Open Watcom C/C++

Compiler and Tools

User's Guide for QNX

3rd Edition

Powersoft®

Notice of Copyright

Portions Copyright © 1984-2002 Sybase, Inc. and its subsidiaries. All rights reserved.

Any part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of anyone.

For more information please visit http://www.openwatcom.org/

ISBN 1-55094-xxx-y

Printed in Canada

Preface

Open Watcom C is an implementation of ISO/ANSI 9899:1990 Programming Language C. The standard was developed by the ANSI X3J11 Technical Committee on the C Programming Language. In addition to the full C language standard, the compiler supports numerous extensions for the Intel 80x86-based personal computer environment. The compiler is also partially compliant with the ISO/IEC 9899:1999 Programming Language C standard.

Open Watcom C++ is an implementation of the Draft Proposed International Standard for Information Systems Programming Language C++ (ANSI X3J16, ISO WG21). In addition to the full C++ language standard, the compiler supports numerous extensions for the Intel 80x86-based personal computer environment.

Open Watcom is well known for its language processors having developed, over the last decade, compilers and interpreters for the APL, BASIC, COBOL, FORTRAN and Pascal programming languages. From the start, Open Watcom has been committed to developing portable software products. These products have been implemented on a variety of processor architectures including the IBM 370, the Intel 8086 family, the Motorola 6809 and 68000, the MOS 6502, and the Digital PDP11 and VAX. In most cases, the tools necessary for porting to these environments had to be created first. Invariably, a code generator had to be written. Assemblers, linkers and debuggers had to be created when none were available or when existing ones were inadequate.

Over the years, much research has gone into developing the "ultimate" code generator for the Intel 8086 family. We have continually looked for new ways to improve the quality of the emitted code, never being quite satisfied with the results. Several major revisions, including some entirely new approaches to code generation, have ensued over the years. Our latest version employs state of the art techniques to produce very high quality code for the 8086 family. We introduced the C compiler in 1987, satisfied that we had a C software development system that would be of major benefit to those developing applications in C for the IBM PC and compatibles.

The *Open Watcom C/C++ Compiler and Tools User's Guide for QNX* describes how to use Open Watcom C/C++ on Intel 80x86-based personal computers to build QNX applications. The User's Guide is covers the following topics:

- The Open Watcom C/C++ compiler including compiler options, benchmarking, include file processing, the preprocessor, predefined macros and keywords, based pointers, precompiled headers, and libraries

- 16-bit memory models, calling conventions, and pragmas

- 32-bit memory models, calling conventions, and pragmas

- In-line assembly language

- The Open Watcom Linker

- The Open Watcom Library Manager

- The Open Watcom Assembler

- The Open Watcom Disassembler

- The Open Watcom Strip Utility

- Environment Variables

- C Diagnostic Messages

- C++ Diagnostic Messages

- Open Watcom C/C++ Run-Time Messages

Additional copies of this documentation may be ordered from:

QNX Software Systems Ltd.
175 Terence Matthews Crescent
Kanata, Ontario
CANADA K2M 1W8
Phone: 613-591-0931
Fax: 613-591-3579

Acknowledgements

This book was produced with the Open Watcom GML electronic publishing system, a software tool developed by WATCOM. In this system, writers use an ASCII text editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The Open Watcom GML software, which runs on a variety of operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

The Plum Hall Validation Suite for C/C++ has been invaluable in verifying the conformance of the Open Watcom C/C++ compilers to the ISO C Language Standard and the Draft Proposed C++ Language Standard.

Many users have provided valuable feedback on earlier versions of the Open Watcom C/C++ compilers and related tools. Their comments were greatly appreciated. If you find problems in the documentation or have some good suggestions, we would like to hear from you.

July, 1997.

Trademarks Used in this Manual

Table of Contents

Open Watcom C/C++ User's Guide 1

1 About This Manual 3

2 Open Watcom C/C++ Compiler Options 7

3 The Open Watcom C/C++ Compilers 9
3.1 Open Watcom C/C++ Command Line Format 9
3.2 Environment Variables 11
3.3 Open Watcom C/C++ Command Line Examples 12
3.4 Benchmarking Hints 14
3.5 Compiler Diagnostics 17
3.6 Open Watcom C/C++ #include File Processing 18
3.7 Open Watcom C/C++ Preprocessor 21
3.8 Open Watcom C/C++ Predefined Macros 23
3.9 Open Watcom C/C++ Extended Keywords 29
3.10 Based Pointers 39
3.10.1 Segment Constant Based Pointers and Objects 40
3.10.2 Segment Object Based Pointers 41
3.10.3 Void Based Pointers 41
3.10.4 Self Based Pointers 42
3.11 The __declspec Keyword 44
3.12 The Open Watcom Code Generator 50

4 Precompiled Headers 53
4.1 Using Precompiled Headers 53
4.2 When to Precompile Header Files 53
4.3 Creating and Using Precompiled Headers 54
4.4 The "-fh[q]" (Precompiled Header) Option 54
4.5 Consistency Rules for Precompiled Headers 55

5 The Open Watcom C/C++ Libraries 57
5.1 Open Watcom C/C++ Library Directory Structure 57
5.2 Open Watcom C/C++ C Libraries 57
5.3 Open Watcom C 16-bit Shared Library 58
5.4 Open Watcom C/C++ Class Libraries 59
5.5 Open Watcom C/C++ Math Libraries 60
5.6 Open Watcom C/C++ 80x87 Math Libraries 61
5.7 Open Watcom C/C++ Alternate Math Libraries 61
5.8 The Open Watcom C/C++ Run-time Initialization Routines 62

Table of Contents

16-bit Topics 65

6 16-bit Memory Models 67
6.1 Introduction 67
6.2 16-bit Code Models 67
6.3 16-bit Data Models 68
6.4 Summary of 16-bit Memory Models 69
6.5 Mixed 16-bit Memory Model 69
6.6 Linking Applications for the Various 16-bit Memory Models 69
6.7 Memory Layout 70

7 16-bit Assembly Language Considerations 73
7.1 Introduction 73
7.2 Data Representation 73
7.2.1 Type "char" 74
7.2.2 Type "short int" 74
7.2.3 Type "long int" 74
7.2.4 Type "int" 75
7.2.5 Type "float" 75
7.2.6 Type "double" 76
7.3 Memory Layout 77
7.4 Calling Conventions for Non-80x87 Applications 79
7.4.1 Passing Arguments Using Register-Based Calling Conventions ... 79
7.4.2 Sizes of Predefined Types 80
7.4.3 Size of Enumerated Types 81
7.4.4 Effect of Function Prototypes on Arguments 81
7.4.5 Interfacing to Assembly Language Functions 82
7.4.6 Functions with Variable Number of Arguments 86
7.4.7 Returning Values from Functions 86
7.5 Calling Conventions for 80x87-based Applications 90
7.5.1 Passing Values in 80x87-based Applications 90
7.5.2 Returning Values in 80x87-based Applications 92

8 16-bit Pragmas 93
8.1 Introduction 93
8.2 Using Pragmas to Specify Options 94
8.3 Using Pragmas to Specify Default Libraries 96
8.4 The ALLOC_TEXT Pragma (C Only) 97
8.5 The CODE_SEG Pragma 98
8.6 The COMMENT Pragma 99
8.7 The DATA_SEG Pragma 99
8.8 The DISABLE_MESSAGE Pragma (C Only) 100

Table of Contents

8.9 The DUMP_OBJECT_MODEL Pragma (C++ Only) 101
8.10 The ENABLE_MESSAGE Pragma (C Only) 101
8.11 The ENUM Pragma 102
8.12 The ERROR Pragma 103
8.13 The EXTREF Pragma 103
8.14 The FUNCTION Pragma 104
8.15 Setting Priority of Static Data Initialization (C++ Only) 105
8.16 The INLINE_DEPTH Pragma (C++ Only) 106
8.17 The INLINE_RECURSION Pragma (C++ Only) 107
8.18 The INTRINSIC Pragma 107
8.19 The MESSAGE Pragma 108
8.20 The ONCE Pragma 108
8.21 The PACK Pragma 109
8.22 The READ_ONLY_FILE Pragma 110
8.23 The TEMPLATE_DEPTH Pragma (C++ Only) 111
8.24 The WARNING Pragma (C++ Only) 112
8.25 Auxiliary Pragmas 112
- 8.25.1 Specifying Symbol Attributes 112
- 8.25.2 Alias Names 113
- 8.25.3 Predefined Aliases 115
  - 8.25.3.1 Predefined "__cdecl" Alias 116
  - 8.25.3.2 Predefined "__pascal" Alias 117
- 8.25.4 Alternate Names for Symbols 117
- 8.25.5 Describing Calling Information 118
  - 8.25.5.1 Loading Data Segment Register 121
  - 8.25.5.2 Defining Exported Symbols in Dynamic Link Libraries 122
  - 8.25.5.3 Defining Windows Callback Functions 122
  - 8.25.5.4 Forcing a Stack Frame 123
- 8.25.6 Describing Argument Information 123
  - 8.25.6.1 Passing Arguments in Registers 123
  - 8.25.6.2 Forcing Arguments into Specific Registers 127
  - 8.25.6.3 Passing Arguments to In-Line Functions 127
  - 8.25.6.4 Removing Arguments from the Stack 128
  - 8.25.6.5 Passing Arguments in Reverse Order 128
- 8.25.7 Describing Function Return Information 129
  - 8.25.7.1 Returning Function Values in Registers 130
  - 8.25.7.2 Returning Structures 131
  - 8.25.7.3 Returning Floating-Point Data 132
- 8.25.8 A Function that Never Returns 134
- 8.25.9 Describing How Functions Use Memory 134
- 8.25.10 Describing the Registers Modified by a Function 139

Table of Contents

8.25.11 An Example 140
8.25.12 Auxiliary Pragmas and the 80x87 141
8.25.12.1 Using the 80x87 to Pass Arguments 142
8.25.12.2 Using the 80x87 to Return Function Values 145
8.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls 146

32-bit Topics 147

9 32-bit Memory Models 149
9.1 Introduction 149
9.2 32-bit Code Models 149
9.3 32-bit Data Models 150
9.4 Summary of 32-bit Memory Models 150
9.5 Flat Memory Model 151
9.6 Mixed 32-bit Memory Model 151
9.7 Linking Applications for the Various 32-bit Memory Models 152
9.8 Memory Layout 152

10 32-bit Assembly Language Considerations 155
10.1 Introduction 155
10.2 Data Representation 155
10.2.1 Type "char" 156
10.2.2 Type "short int" 156
10.2.3 Type "long int" 156
10.2.4 Type "int" 157
10.2.5 Type "float" 157
10.2.6 Type "double" 158
10.3 Memory Layout 159
10.4 Calling Conventions for Non-80x87 Applications 161
10.4.1 Passing Arguments Using Register-Based Calling Conventions 161
10.4.2 Sizes of Predefined Types 162
10.4.3 Size of Enumerated Types 163
10.4.4 Effect of Function Prototypes on Arguments 163
10.4.5 Interfacing to Assembly Language Functions 164
10.4.6 Using Stack-Based Calling Conventions 168
10.4.7 Functions with Variable Number of Arguments 172
10.4.8 Returning Values from Functions 172
10.5 Calling Conventions for 80x87-based Applications 176
10.5.1 Passing Values in 80x87-based Applications 176

Table of Contents

10.5.2 Returning Values in 80x87-based Applications 178

11 32-bit Pragmas 179
11.1 Introduction 179
11.2 Using Pragmas to Specify Options 180
11.3 Using Pragmas to Specify Default Libraries 182
11.4 The ALLOC_TEXT Pragma (C Only) 183
11.5 The CODE_SEG Pragma 184
11.6 The COMMENT Pragma 185
11.7 The DATA_SEG Pragma 185
11.8 The DISABLE_MESSAGE Pragma (C Only) 186
11.9 The DUMP_OBJECT_MODEL Pragma (C++ Only) 187
11.10 The ENABLE_MESSAGE Pragma (C Only) 187
11.11 The ENUM Pragma 188
11.12 The ERROR Pragma 189
11.13 The EXTREF Pragma 189
11.14 The FUNCTION Pragma 190
11.15 Setting Priority of Static Data Initialization (C++ Only) 191
11.16 The INLINE_DEPTH Pragma (C++ Only) 192
11.17 The INLINE_RECURSION Pragma (C++ Only) 193
11.18 The INTRINSIC Pragma 193
11.19 The MESSAGE Pragma 194
11.20 The ONCE Pragma 194
11.21 The PACK Pragma 195
11.22 The READ_ONLY_FILE Pragma 196
11.23 The TEMPLATE_DEPTH Pragma (C++ Only) 197
11.24 The WARNING Pragma (C++ Only) 198
11.25 Auxiliary Pragmas 198
11.25.1 Specifying Symbol Attributes 198
11.25.2 Alias Names 199
11.25.3 Predefined Aliases 202
11.25.3.1 Predefined "__cdecl" Alias 202
11.25.3.2 Predefined "__pascal" Alias 203
11.25.3.3 Predefined "__stdcall" Alias 203
11.25.3.4 Predefined "__syscall" Alias 204
11.25.4 Alternate Names for Symbols 205
11.25.5 Describing Calling Information 206
11.25.5.1 Loading Data Segment Register 208
11.25.5.2 Defining Exported Symbols in Dynamic Link Libraries 209
11.25.5.3 Forcing a Stack Frame 209
11.25.6 Describing Argument Information 209

Table of Contents

11.25.6.1 Passing Arguments in Registers ... 210
11.25.6.2 Forcing Arguments into Specific Registers ... 213
11.25.6.3 Passing Arguments to In-Line Functions ... 214
11.25.6.4 Removing Arguments from the Stack ... 215
11.25.6.5 Passing Arguments in Reverse Order ... 215
11.25.7 Describing Function Return Information ... 216
11.25.7.1 Returning Function Values in Registers ... 216
11.25.7.2 Returning Structures ... 218
11.25.7.3 Returning Floating-Point Data ... 219
11.25.8 A Function that Never Returns ... 220
11.25.9 Describing How Functions Use Memory ... 221
11.25.10 Describing the Registers Modified by a Function ... 226
11.25.11 An Example ... 227
11.25.12 Auxiliary Pragmas and the 80x87 ... 228
11.25.12.1 Using the 80x87 to Pass Arguments ... 228
11.25.12.2 Using the 80x87 to Return Function Values ... 232
11.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls ... 233

In-line Assembly Language ... 235

12 In-line Assembly Language ... 237
12.1 In-line Assembly Language Default Environment ... 237
12.2 In-line Assembly Language Tutorial ... 238
12.3 Labels in In-line Assembly Code ... 245
12.4 Variables in In-line Assembly Code ... 245
12.5 In-line Assembly Language using _asm ... 248
12.6 In-line Assembly Directives and Opcodes ... 250

Open Watcom Tools ... 259

The Open Watcom Linker ... 261

13 The Open Watcom Linker ... 263
13.1 Using the SYSTEM Directive ... 265
13.2 Linking 16-bit QNX Executable Files ... 267
13.3 Linking 32-bit QNX Executable Files ... 267

14 Linker Directives and Options ... 269

Table of Contents

14.1 The ALIAS Directive 272
14.2 The ARTIFICIAL Option 273
14.3 The CACHE Option 274
14.4 The CASEEXACT Option 275
14.5 The # Directive 276
14.6 The CVPACK Option 277
14.7 The DEBUG Directive 278
    14.7.1 Line Numbering Information - DEBUG WATCOM LINES 280
    14.7.2 Local Symbol Information - DEBUG WATCOM LOCALS 281
    14.7.3 Typing Information - DEBUG WATCOM TYPES 281
    14.7.4 All Debugging Information - DEBUG WATCOM ALL 282
    14.7.5 Global Symbol Information 282
    14.7.6 Global Symbols for the NetWare Debugger - DEBUG NOVELL 282
    14.7.7 The ONLYEXPORTS Debugging Option 282
    14.7.8 Using DEBUG Directives 283
    14.7.9 Removing Debugging Information from an Executable File 284
14.8 The DISABLE Directive 285
14.9 The DOSSEG Option 287
14.10 The ELIMINATE Option 289
14.11 The ENDLINK Directive 290
14.12 The FARCALLS Option 291
14.13 The FILE Directive 292
14.14 The FILLCHAR Option 294
14.15 The FORMAT Directive 295
14.16 The @ Directive 304
14.17 The LANGUAGE Directive 307
14.18 The LIBFILE Directive 308
14.19 The LIBPATH Directive 310
14.20 The LIBRARY Directive 312
    14.20.1 Searching for Libraries Specified in Environment Variables 313
    14.20.2 Converting Libraries Created using Phar Lap 386|LIB 313
14.21 The LINEARRELOCS Option 315
14.22 The LONGLIVED Option 316
14.23 The MANGLEDNAMES Option 317
14.24 The MAP Option 318
14.25 The MAXERRORS Option 319
14.26 The MODFILE Directive 320
14.27 The MODTRACE Directive 321
14.28 The NAME Directive 322
14.29 The NAMELEN Option 323
14.30 The NODEFAULTLIBS Option 324

Table of Contents

14.31 The OPTION Directive 325
14.32 The OPTLIB Directive 326
14.32.1 Searching for Optional Libraries Specified in Environment Variables 327
14.33 The ORDER Directive 328
14.34 The OSNAME Option 332
14.35 The OUTPUT Directive 333
14.36 The PATH Directive 336
14.37 The PRIVILEGE Option 338
14.38 The QUIET Option 339
14.39 The REDEFSOK Option 340
14.40 The REFERENCE Directive 341
14.41 The SHOWDEAD Option 342
14.42 The SORT Directive 343
14.43 The STACK Option 344
14.44 The START Option 345
14.45 The STARTLINK Directive 346
14.46 The STATICS Option 347
14.47 The SYMFILE Option 348
14.48 The SYMTRACE Directive 350
14.49 The SYSTEM Directive 351
14.49.1 Special System Names 353
14.50 The UNDEFSOK Option 355
14.51 The VERBOSE Option 356
14.52 The VFREMOVAL Option 357

15 The QNX Executable File Format 359
15.1 Memory Layout 361

16 Open Watcom Linker Diagnostic Messages 363

The Open Watcom Library Manager 385

17 The Open Watcom Library Manager 387
17.1 Introduction 387
17.2 The Open Watcom Library Manager Command Line 388
17.3 Open Watcom Library Manager Module Commands 389
17.4 Adding Modules to a Library File 390
17.5 Deleting Modules from a Library File 390
17.6 Replacing Modules in a Library File 391
17.7 Extracting a Module from a Library File 392

Table of Contents

17.8 Creating Import Libraries 393
17.9 Creating Import Library Entries 393
17.10 Commands from a File or Environment Variable 394
17.11 Open Watcom Library Manager Options 394
17.11.1 Suppress Creation of Backup File - "b" Option 394
17.11.2 Case Sensitive Symbol Names - "c" Option 395
17.11.3 Specify Output Directory - "d" Option 395
17.11.4 Specify Output Format - "f" Option 395
17.11.5 Generating Imports - "i" Option 396
17.11.6 Creating a Listing File - "l" Option 397
17.11.7 Display C++ Mangled Names - "m" Option 397
17.11.8 Always Create a New Library - "n" Option 397
17.11.9 Specifying an Output File Name - "o" Option 398
17.11.10 Specifying a Library Record Size - "p" Option 398
17.11.11 Operate Quietly - "q" Option 399
17.11.12 Strip Line Number Records - "s" Option 399
17.11.13 Trim Module Name - "t" Option 399
17.11.14 Operate Verbosely - "v" Option 399
17.11.15 Explode Library File - "x" Option 400
17.12 Librarian Error Messages 400

The Open Watcom Assembler 405

18 The Open Watcom Assembler 407
18.1 Introduction 407
18.2 Assembly Directives and Opcodes 409
18.3 Unsupported Directives 418
18.4 Open Watcom Assembler Specific 418
18.4.1 Naming convention 419
18.4.2 Open Watcom "C" name mangler 419
18.4.3 Calling convention 420
18.5 Open Watcom Assembler Diagnostic Messages 420

The Open Watcom Disassembler 431

19 The Object File Disassembler 433
19.1 Introduction 433
19.2 Changing the Internal Label Character - "i=<char>" 434
19.3 The Assembly Format Option - "a" 434
19.4 The External Symbols Option - "e" 435

Table of Contents

19.5 The No Instruction Name Pseudonyms Option - "fp" 436
19.6 The No Register Name Pseudonyms Option - "fr" 436
19.7 The Alternate Addressing Form Option - "fi" 436
19.8 The Uppercase Instructions/Registers Option - "fu" 436
19.9 The Listing Option - "l[=<list_file>]" 437
19.10 The Public Symbols Option - "p" 437
19.11 Retain C++ Mangled Names - "m" 438
19.12 The Source Option - "s[=<source_file>]" 438
19.13 An Example 439

20 Optimization of Far Calls 445

The Open Watcom Strip Utility 447

21 The Open Watcom Strip Utility 449
21.1 Introduction 449
21.2 The Open Watcom Strip Utility Command Line 449
21.3 Strip Utility Messages 451

Appendices 453

A. Use of Environment Variables 455
A.1 FORCE 455
A.2 INCLUDE 455
A.3 LIB 456
A.4 PATH 456
A.5 TMPDIR 457
A.6 WATCOM 457
A.7 WCC 458
A.8 WCC386 458
A.9 WCGMEMORY 459
A.10 WD 459
A.11 WD_PATH 460
A.12 WPP 460
A.13 WPP386 461

B. Open Watcom C Diagnostic Messages 463
B.1 Warning Level 1 Messages 464
B.2 Warning Level 2 Messages 471
B.3 Warning Level 3 Messages 471

Table of Contents

B.4 Error Messages 473
B.5 Informational Messages 497
B.6 Pre-compiled Header Messages 499
B.7 Miscellaneous Messages and Phrases 500

C. Open Watcom C++ Diagnostic Messages 501
C.1 Diagnostic Messages 502

D. Open Watcom C/C++ Run-Time Messages 747
D.1 Run-Time Error Messages 747
D.2 errno Values and Their Meanings 748
D.2.1 Shared Library Errors 751
D.2.2 Non-blocking and Interrupt I/O 752
D.2.3 IPC/Network Software -- Argument Errors 752
D.2.4 IPC/Network Software -- Operational Errors 753
D.2.5 QNX Specific 753
D.3 Math Run-Time Error Messages 754

Open Watcom C/C++ User's Guide

1 About This Manual

This manual contains the following chapters:

Chapter 1 — "About This Manual".

This chapter provides an overview of the contents of this guide.

Chapter 2 — "Open Watcom C/C++ Compiler Options" on page 7.

This chapter provides a summary and reference section for all the C and C++ compiler options.

Chapter 3 — "The Open Watcom C/C++ Compilers" on page 9.

This chapter describes how to compile an application from the command line. This chapter also describes compiler environment variables, benchmarking hints, compiler diagnostics, #include file processing, the preprocessor, predefined macros, extended keywords, and the code generator.

Chapter 4 — "Precompiled Headers" on page 53.

This chapter describes the use of precompiled headers to speed up compilation.

Chapter 5 — "The Open Watcom C/C++ Libraries" on page 57.

This chapter describes the Open Watcom C/C++ library directory structure, C libraries, class libraries, math libraries, 80x87 math libraries, alternate math libraries, the "NO87" environment variable, and the run-time initialization routines.

Chapter 6 — "16-bit Memory Models" on page 67.

This chapter describes the Open Watcom C/C++ memory models (including code and data models), the tiny memory model, the mixed memory model, linking applications for the various memory models, creating a tiny memory model application, and memory layout in an executable.

Chapter 7 — "16-bit Assembly Language Considerations" on page 73.

This chapter describes issues relating to 16-bit interfacing such as parameter passing conventions.

Chapter 8 — "16-bit Pragmas" on page 93.

This chapter describes the use of pragmas with the 16-bit compilers.

Chapter 9 — "32-bit Memory Models" on page 149.

This chapter describes the Open Watcom C/C++ memory models (including code and data models), the flat memory model, the mixed memory model, linking applications for the various memory models, and memory layout in an executable.

Chapter 10 — "32-bit Assembly Language Considerations" on page 155.

This chapter describes issues relating to 32-bit interfacing such as parameter passing conventions.

Chapter 11 — "32-bit Pragmas" on page 179.

This chapter describes the use of pragmas with the 32-bit compilers.

Chapter 12 — "In-line Assembly Language" on page 237.

This chapter describes in-line assembly language programming using the auxiliary pragma.

Chapter 13 — "The Open Watcom Linker" on page 263.

This chapter introduces the Open Watcom Linker.

Chapter 14 — "Linker Directives and Options" on page 269.

This chapter describes the Open Watcom Linker directives and options that apply to QNX in alphabetical order.

Chapter 15 — "The QNX Executable File Format" on page 359.

This chapter describes the QNX executable file format.

Chapter 16 — "Open Watcom Linker Diagnostic Messages" on page 363.

This chapter explains the Open Watcom Linker error messages.

Chapter 17 — "The Open Watcom Library Manager" on page 387.

This chapter describe the Open Watcom Library Manager.

Chapter 18 — "The Open Watcom Assembler" on page 407.

This chapter describe the Open Watcom Assembler.

Chapter 19 — "The Object File Disassembler" on page 433.

This chapter describe the Open Watcom Disassembler.

Chapter 20 — "Optimization of Far Calls" on page 445.

This chapter describes the optimization of far calls.

Chapter 21 — "The Open Watcom Strip Utility" on page 449.

This chapter describe the Open Watcom Strip Utility.

Appendix A. — "Use of Environment Variables" on page 455.

This appendix describes all the environment variables used by the compilers and related tools.

Appendix B. — "Open Watcom C Diagnostic Messages" on page 463.

This appendix lists all of the Open Watcom C diagnostic messages with an explanation for each.

Appendix C. — "Open Watcom C++ Diagnostic Messages" on page 501.

This appendix lists all of the Open Watcom C++ diagnostic messages with an explanation for each.

Appendix D. — "Open Watcom C/C++ Run-Time Messages" on page 747.

This appendix lists all of the C/C++ run-time diagnostic messages with an explanation for each.

2 Open Watcom C/C++ Compiler Options

This chapter describes all the compiler options that are available.

3 The Open Watcom C/C++ Compilers

This chapter covers the following topics.

- Command line syntax (see "Open Watcom C/C++ Command Line Format")
- Environment variables used by the compilers (see "Environment Variables" on page 11)
- Examples of command line syntax (see "Open Watcom C/C++ Command Line Examples" on page 12)
- Interpreting diagnostic messages (see "Compiler Diagnostics" on page 17)
- #include file handling (see "Open Watcom C/C++ #include File Processing" on page 18)
- Using the preprocessor built into the compilers (see "Open Watcom C/C++ Preprocessor" on page 21)
- System-dependent macros predefined by the compilers (see "Open Watcom C/C++ Predefined Macros" on page 23)
- Additional keywords supported by the compilers (see "Open Watcom C/C++ Extended Keywords" on page 29)
- Based pointer support in the compilers (see "Based Pointers" on page 39)
- Notes about the Code Generator (see "The Open Watcom Code Generator" on page 50)

3.1 Open Watcom C/C++ Command Line Format

The formal Open Watcom C/C++ command line syntax is shown below.

compiler_name [options] [file_spec] [options] [@extra_opts]

The square brackets [ ] denote items which are optional.

compiler_name is one of the Open Watcom C/C++ compiler command names.

wcc	is the Open Watcom C compiler for 16-bit Intel platforms.
wpp	is the Open Watcom C++ compiler for 16-bit Intel platforms.
wcc386	is the Open Watcom C compiler for 32-bit Intel platforms.
wpp386	is the Open Watcom C++ compiler for 32-bit Intel platforms.

file_spec is the file name specification of one or more files to be compiled. If file_spec is specified as the single character ".", an input file is read from standard input and the output file name defaults to stdin.obj.

If no path is specified, the current working directory is assumed. If the file is not in the current directory, an adjacent "C" directory (i.e., `../c`) is searched if it exists.

If no file extension is specified, the compiler will check for a file with one of the following extensions in the order listed:

.cpp	(C++ only)
.cc	(C++ only)
.c	(C/C++)

A QNX filename extension consists of that portion of a filename containing the last "." and any characters which follow it.

Example:

```
File Specification              Extension
/home/john.doe/foo              (none)
/home/john.doe/foo.             .
/home/john.doe/foo.bar          .bar
/home/john.doe/foo.goo.bar      .bar
```

If a period "." is specified but not the extension, the file is assumed to have no filename extension.

If only the compiler name is specified then the compiler will display a list of available options.

options is a list of valid compiler options, each preceded by a dash ("−"). Options may be specified in any order.

extra_opts is the name of an environment variable or file which contains additional command line options to be processed. If the specified environment variable does not exist, a search is made for a file with the specified name. If no file extension is included in the specified name, the default file extension is ".occ". A search of the current directory is made. If not successful, an adjacent "OCC" directory (i.e., `../occ`) is searched if it exists.

3.2 Environment Variables

Environment variables can be used to specify commonly used compiler options. There is one environment variable for each compiler (the name of the environment variable is the same as the compiler name). The Open Watcom C/C++ environment variable names are:

WCC used with the Open Watcom C compiler for 16-bit Intel platforms

Example:
```
    $ export "WCC=-d1 -ot"
```

WPP used with the Open Watcom C++ compiler for 16-bit Intel platforms

Example:
```
    $ export "WPP=-d1 -ot"
```

WCC386 used with the Open Watcom C compiler for 32-bit Intel platforms

Example:
```
    $ export "WCC386=-d1 -ot"
```

WPP386 used with the Open Watcom C++ compiler for 32-bit Intel platforms

Example:
```
    $ export "WPP386=-d1 -ot"
```

The options specified in environment variables are processed before options specified on the command line. The above examples define the default options to be "d1" (include line number debugging information in the object file), and "ot" (favour time optimizations over size optimizations).

Once a particular environment variable has been defined, those options listed become the default each time the associated compiler is used. The compiler command line can be used to override any options specified in the environment string.

> ***Hint:*** If you use the same compiler options all the time, you may find it handy to define the environment variable in your user initialization file.

3.3 Open Watcom C/C++ Command Line Examples

The following are some examples of using Open Watcom C/C++ to compile C/C++ source programs.

Example:

```
$ compiler_name report -d1 -s
```

The compiler processes `report.c(pp)` producing an object file which contains source line number information. Stack overflow checking is omitted from the object code.

Example:

```
$ compiler_name -mm -fpc calc
```

The compiler compiles `calc.c(pp)` for the Intel "medium" memory model and generates calls to floating-point library emulation routines for all floating-point operations. Memory models are described in the chapter entitled "16-bit Memory Models" on page 67.

Example:

```
$ compiler_name kwikdraw -2 -fpi87 -oaxt
```

The compiler processes `kwikdraw.c(pp)` producing 16-bit object code for an Intel 286 system equipped with an Intel 287 numeric data processor (or any upward compatible 386/387, 486DX, or Pentium system). While the choice of these options narrows the number of microcomputer systems where this code will execute, the resulting code will be highly optimized for this type of system.

Example:

```
$ compiler_name -mf -3s calc
```

The compiler compiles `calc.c(pp)` for the Intel 32-bit "flat" memory model. The compiler will generate 386 instructions based on 386 instruction timings using the stack-based argument passing convention. The resulting code will be optimized for Intel 386 systems.

Memory models are described in the chapter entitled "32-bit Memory Models" on page 149. Argument passing conventions are described in the chapter entitled "32-bit Assembly Language Considerations" on page 155.

Example:
```
$ compiler_name kwikdraw -4r -fpi87 -oaimxt
```

The compiler processes `kwikdraw.c(pp)` producing 32-bit object code for an Intel 386-compatible system equipped with a 387 numeric data processor. The compiler will generate 386 instructions based on 486 instruction timings using the register-based argument passing convention. The resulting code will be highly optimized for Intel 486 systems.

Example:
```
$ compiler_name ../source/modabs -d2
```

The compiler processes `../source/modabs.c(pp)` (a file in a directory which is adjacent to the current one). The object file is placed in the current directory. Included with the object code and data is information on local symbols and data types. The code generated is straight-forward, unoptimized code which can be readily debugged with the Open Watcom Debugger.

Example:
```
$ export "compiler_name=-i=/includes -mc"
$ compiler_name /cprogs/grep.tst -fi=iomods.c
```

The compiler processes the program contained in the file `/cprogs/grep.tst.` The file `iomods.c` is included as if it formed part of the source input stream. The include search path and memory model options are defaults each time the compiler is invoked. The memory model option could be overridden on the command line. After looking for an "include" file in the current directory, the compiler will search each directory listed in the "i" path. See the section entitled "Open Watcom C/C++ #include File Processing" on page 18 for more information.

Example:
```
$ compiler_name grep -fo=../obj/
```

The compiler processes the program contained in the file `grep.c(pp)` which is located in the current directory. The object file is placed in the directory `../obj` under the name `grep.o.`

Example:
```
$ compiler_name -dDBG=1 grep -fo=../obj/.dbo
```

The compiler processes the program contained in the file `grep.c(pp)` which is located in the current directory. The macro "DBG" is defined so that conditional debugging statements that have been placed in the source are compiled. The object file is placed in the directory `../obj` and its filename extension will be ".dbo" (instead of ".o"). Selection of a different filename extension permits easy identification of object files that have been compiled with debugging statements.

Example:

```
$ compiler_name -g=GKS -s /gks/gopks
```

The compiler generates code for `gopks.c(pp)` and places it into the "GKS" group. If the "g" option had not been specified, the code would not have been placed in any group. Assume that this file contains the definition of the routine `gopengks` as follows:

```
void far gopengks( int workstation, long int h )
{
     .
     .
     .
}
```

For a small code model, the routine `gopengks` must be defined in this file as `far` since it is placed in another group. The "s" option is also specified to prevent a run-time call to the stack overflow check routine which will be placed in a different code segment at link time. The `gopengks` routine must be prototyped by C routines in other groups as

```
void far gopengks( int workstation, long int h );
```

since it will appear in a different code segment.

3.4 Benchmarking Hints

The Open Watcom C/C++ compiler contains many options for controlling the code to be produced. It is impossible to have a certain set of compiler options that will produce the absolute fastest execution times for all possible applications. With that said, we will list the compiler options that we think will give the best execution times for most applications. You may have to experiment with different options to see which combination of options generates the fastest code for your particular application.

The recommended options for generating the fastest 16-bit Intel code are:

Pentium Pro -onatx -oh -oi+ -ei -zp8 -6 -fpi87 -fp6

Pentium -onatx -oh -oi+ -ei -zp8 -5 -fpi87 -fp5

486 -onatx -oh -oi+ -ei -zp8 -4 -fpi87 -fp3

386 -onatx -oh -oi+ -ei -zp8 -3 -fpi87 -fp3

286 -onatx -oh -oi+ -ei -zp8 -2 -fpi87 -fp2

186 -onatx -oh -oi+ -ei -zp8 -1 -fpi87

8086 -onatx -oh -oi+ -ei -zp8 -0 -fpi87

The recommended options for generating the fastest 32-bit Intel code are:

Pentium Pro -onatx -oh -oi+ -ei -zp8 -6 -fp6

Pentium -onatx -oh -oi+ -ei -zp8 -5 -fp5

486 -onatx -oh -oi+ -ei -zp8 -4 -fp3

386 -onatx -oh -oi+ -ei -zp8 -3 -fp3

The "oi+" option is for C++ only. Under some circumstances, the "ob" and "ol+" optimizations may also give better performance with 32-bit Intel code.

Option "on" causes the compiler to replace floating-point divisions with multiplications by the reciprocal. This generates faster code (multiplication is faster than division), but the result may not be the same because the reciprocal may not be exactly representable.

Option "oe" causes small user written functions to be expanded in-line rather than generating a call to the function. Expanding functions in-line can further expose other optimizations that couldn’t otherwise be detected if a call was generated to the function.

Option "oa" causes the compiler to relax alias checking.

Option "ot" must be specified to cause the code generator to select code sequences which are faster without any regard to the size of the code. The default is to select code sequences which strike a balance between size and speed.

Option "ox" is equivalent to "obmiler" and "s" which causes the compiler/code generator to do branch prediction ("ob"), generate 387 instructions in-line for math functions such as sin, cos, sqrt ("om"), expand intrinsic functions in-line ("oi"), perform loop optimizations ("ol"),

expand small user functions in-line ("oe"), reorder instructions to avoid pipeline stalls ("or"), and to not generate any stack overflow checking ("s"). Option "or" is very important for generating fast code for the Pentium and Pentium Pro processors.

Option "oh" causes the compiler to attempt repeated optimizations (which can result in longer compiles but more optimal code).

Option "oi+" causes the C++ compiler to expand intrinsic functions in-line (just like "oi") but also sets the *inline_depth* to its maximum (255). By default, *inline_depth* is 3. The *inline_depth* can also be changed by using the C++ `inline_depth` pragma.

Option "ei" causes the compiler to allocate at least an "int" for all enumerated types.

Option "zp8" causes all data to be aligned on 8 byte boundaries. The default is "zp2" for the 16-bit compiler and "zp8" for 32-bit compiler. If, for example, "zp1" packing was specified then this would pack all data which would reduce the amount of data memory required but would require extra clock cycles to access data that is not on an appropriate boundary.

Options "0", "1", "2", "3", "4", "5" and "6" emit Intel code sequences optimized for processor-specific instruction set features and timings. For 16-bit Intel applications, the use of these options may limit the range of systems on which the application will run but there are execution performance improvements.

Options "fp2", "fp3", "fp5" and "fp6" emit Intel floating-point operations targetted at specific features of the math coprocessor in the Intel series. For 16-bit Intel applications, the use of these options may limit the range of systems on which the application will run but there are execution performance improvements.

Option "fpi87" causes in-line Intel 80x87 numeric data processor instructions to be generated into the object code for floating-point operations. Floating-point instruction emulation is not included so as to obtain the best floating-point performance in 16-bit Intel applications.

For 32-bit Intel applications, the use of the "fp5" option will give good performance on the Intel Pentium but less than optimal performance on the 386 and 486. The use of the "5" option will give good performance on the Pentium and minimal, if any, impact on the 386 and 486. Thus, the following set of options gives good overall performance for the 386, 486 and Pentium processors.

```
-onatx -oh -oi+ -ei -zp8 -5 -fp3
```

3.5 Compiler Diagnostics

If the compiler prints diagnostic messages to the screen, it will also place a copy of these messages in a file in your current directory. The file will have the same file name as the source file and an extension of ".err". The compiler issues two types of diagnostic messages, namely warnings or errors. A warning message does not prevent the production of an object file. However, error messages indicate that a problem is severe enough that it must be corrected before the compiler will produce an object file. The error file is a handy reference when you wish to correct the errors in the source file.

Just to illustrate the diagnostic features of Open Watcom C/C++, we will modify the "hello" program in such a way as to introduce some errors.

Example:

```
#include <stdio.h>

int main()
  {
     int x;
     printf( "Hello world\n" );
     return( y );
  }
```

The equivalent C++ program follows:

Example:

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
    int x;
    cout << "Hello world" << endl;
    return( y );
}
```

In this example, we have added the lines:

```
int x;
```

and

```
return( y );
```

and changed the keyword `void` to `int`.

We compile the program with the "warning" option.

Example:

$ *compiler_name* `hello -w3`

For the C program, the following output appears on the screen.

```
hello.c(7): Error! E1011: Symbol 'y' has not been declared
hello.c(5): Warning! W202: Symbol 'x' has been defined, but not
                           referenced
hello.c: 8 lines, included 174, 1 warnings, 1 errors
```

For the C++ program, the following output appears on the screen.

```
hello.cpp(8): Error! E029: (col 13) symbol 'y' has not been declared
hello.cpp(9): Warning! W014: (col 1) no reference to symbol 'x'
hello.cpp(9): Note! N392: (col 1) 'int x' in 'int main( void )'
              defined in: hello.cpp(6) (col 9)
hello.cpp: 9 lines, included 1628, 1 warning, 1 error
```

Here we see an example of both types of messages. An error and a warning message have been issued. As indicated by the error message, we require a declarative statement for the identifier `y`. The warning message indicates that, while it is not a violation of the rules of C/C++ to define a variable without ever using it, we probably did not intend to do so. Upon examining the program, we find that:

1. the variable `x` should have been assigned a value, and
2. the variable `y` has probably been incorrectly typed and should have been entered as `x`.

The complete list of Open Watcom C/C++ diagnostic messages is presented in an appendix of this guide.

3.6 Open Watcom C/C++ #include File Processing

When using the `#include` preprocessor directive, a header is identified by a sequence of characters placed between the "<" and ">" delimiters (e.g., <file>) and a source file is identified by a sequence of characters enclosed by quotation marks (e.g., "file"). Open Watcom C/C++ makes a distinction between the use of "<>" or quotation marks to surround the name of the file to be included. The search techniques for header files and source files are slightly different. Consider the following example.

Example:
```
#include <stdio.h>   /* a system header file */
#include "stdio.h"   /* your own header or source file */
```

You should use "<" and ">" when referring to standard or system header files and quotation marks when referring to your own header and source files.

The character sequence placed between the delimiters in an `#include` directive represents the name of the file to be included. The file name may include node, path, and extension.

It is not necessary to include the node and path specifiers in the file specification when the file resides on a different node or in a different directory. Open Watcom C/C++ provides a mechanism for looking up include files which may be located in various directories and disks of the computer system. Open Watcom C/C++ searches directories for header and source files in the following order (the search stops once the file has been located):

1. If the file specification enclosed in quotation marks ("file-spec") or angle brackets (<file-spec>) contains the complete node and path specification, that file is included (provided it exists). No other searching is performed. The node need not be specified in which case the current node is assumed.

2. If the file specification is enclosed in quotation marks, the current directory is searched.

3. Next, if the file specification is enclosed in quotation marks, the directory of the file containing the `#include` directive is searched. If the current file is also an `#include` file, the directory of the parent file is searched next. This search continues recursively through all the nested `#include` files until the original source file's directory is searched.

4. Next, if the file specification enclosed in quotation marks ("file-spec") or in angle brackets (<file-spec>), each directory listed in the "i" path is searched (in the order that they were specified).

5. Next, each directory listed in the **<os>_INCLUDE** environment variable is searched (in the order that they were specified). The environment variable name is constructed from the current build target name. The default build targets are:

   DOS when the host operating system is DOS,

   OS2 when the host operating system is OS/2,

   NT when the host operating system is Windows NT/95, or

QNX when the host operating system is QNX.

For example, the environment variable **OS2_INCLUDE** will be searched if the build target is "OS2". The build target would be OS/2 if:

1. the host operating system is OS/2 and the "bt" option was not specified, or
2. the "bt=OS2" option was explicitly specified.

6. Next, each directory listed in the **INCLUDE** environment variable is searched (in the order that they were specified).

7. Finally, if the file specification is enclosed in quotation marks, an adjacent "H" directory (i.e., `../h`) is searched if it exists.

In the above example, `<stdio.h>` and `"stdio.h"` could refer to two different files if there is a `stdio.h` in the current directory and one in the Open Watcom C/C++ include file directory `(/usr/include)` and the current directory is not listed in an "i" path or the **INCLUDE** environment variable.

The compiler will search the directories listed in "i" paths (see description of the "i" option) and the **INCLUDE** environment variable in a manner analogous to that which the operating system shell will use when searching for programs by using the **PATH** environment variable.

The "export" command is used to define an **INCLUDE** environment variable that contains a list of directories. A command of the form

```
export INCLUDE=path:path...
```

is issued before running Open Watcom C/C++ the first time.

We illustrate the use of the `#include` directive in the following example.

Example:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>
```

```
#include "common.c"

int main()
  {
    initialize();
    update_files();
    create_report();
    finalize();
  }

#include "part1.c"
#include "part2.c"
```

If the above text is stored in the source file `report.c` in the current directory then we might issue the following commands to compile the application.

Example:

```
$ export INCLUDE=/usr/include://1/headers
$ compiler_name report -fo=../obj/ -i=../source
```

In the above example, the "export" command is used to define the **INCLUDE** environment variable. It specifies that the `/usr/include` directory (of the current node) and the `/headers` directory (a directory on node 1) are to be searched.

The Open Watcom C/C++ "i" option defines a third place to search for include files. The advantage of the **INCLUDE** environment variable is that it need not be specified each time the compiler is run.

3.7 Open Watcom C/C++ Preprocessor

The Open Watcom C/C++ preprocessor forms an integral part of Open Watcom C/C++. When any form of the "p" option is specified, only the preprocessor is invoked. No code is generated and no object file is produced. The output of the preprocessor is written to the standard output file, although it can also be redirected to a file using the "fo" option. Suppose the following C/C++ program is contained in the file `msgid.c`.

Example:
```
#define _IBMPC 0
#define _IBMPS2 1

#if _TARGET == _IBMPS2
char *SysId = { "IBM PS/2" };
#else
char *SysId = { "IBM PC" };
#endif

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

We can use the Open Watcom C/C++ preprocessor to generate the C/C++ code that would actually be compiled by the compiler by issuing the following command.

Example:
```
$ compiler_name msgid -plc -fo -d_TARGET=_IBMPS2
```

The file `msgid.i` will be created and will contain the following C/C++ code.

```
#line 1 "msgid.c"




char *SysId = { "IBM PS/2" };
#line 9 "msgid.c"

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

Note that the file `msgid.i` can be used as input to the compiler.

Example:
```
$ compiler_name msgid.i
```

Since `#line` directives are present in the file, the compiler can issue error messages in terms of the original source file line numbers.

3.8 Open Watcom C/C++ Predefined Macros

In addition to the standard ISO-defined macros supported by the Open Watcom C/C++ compilers, several additional system-dependent macros are also defined. These are described in this section. See the ***Open Watcom C Language Reference*** manual for a description of the standard macros.

The Open Watcom C/C++ compilers run on various host operating systems including DOS, OS/2, Windows NT, Windows 95 and QNX. Any of the supported host operating systems can be used to develop applications for a number of target systems. By default, the target operating system for the application is the same as the host operating system unless some option or combination of options is specified. For example, DOS applications are built on DOS by default, OS/2 applications are built on OS/2 by default, and so on. But the flexibility is there to build applications for other operating systems/environments.

The macros described below may be used to identify the target system for which the application is being compiled. (Note: In several places in the following text, a pair of underscore characters appears as __ which resembles a single, elongated underscore.)

The Open Watcom C/C++ compilers support both 16-bit and 32-bit application development. The following macros are defined for 16-bit and 32-bit target systems.

```
 16-bit      32-bit
========    ========
__X86__     __X86__
__I86__     __386__
M_I86       M_I386
_M_I86      _M_I386
_M_IX86     _M_IX86
```

Notes:

1. The `__X86__` identifies the target as an Intel environment.
2. The `__I86__`, `M_I86` and `_M_I86` macros identify the target as a 16-bit Intel environment.
3. The `__386__`, `M_I386` and `_M_I386` macros identify the target as a 32-bit Intel environment.

4. The _M_IX86 macro is identically equal to 100 times the architecture compiler option value (-0, -1, -2, -3, -4, -5, etc.). If "-5" (Pentium instruction timings) was specified as a compiler option, then the value of _M_IX86 would be 500.

The Open Watcom C/C++ compilers support application development for a variety of operating systems. The following macros are defined for particular target operating systems.

```
Target      Macros
======      ======================================
DOS         __DOS__, _DOS, MSDOS
OS/2        __OS2__
QNX         __QNX__, __UNIX__
Netware     __NETWARE__, __NETWARE_386__
NT          __NT__
Windows     __WINDOWS__, _WINDOWS, __WINDOWS_386__
Linux       __LINUX__, __UNIX__
```

Notes:

1. The __DOS__, _DOS and MSDOS macros are defined when the build target is "DOS" (16-bit DOS or 32-bit extended DOS).

2. The __OS2__ macro is defined when the build target is "OS2" (16-bit or 32-bit OS/2).

3. The __QNX__ and __UNIX__ macros are defined when the build target is "QNX" (16-bit or 32-bit QNX).

4. The __NETWARE__ and __NETWARE_386__ macros are defined when the build target is "NETWARE" (Novell NetWare).

5. The __NT__ macro is defined when the build target is "NT" (Windows NT and Windows 95).

6. The __WINDOWS__ macro is defined when the build target is "WINDOWS" or one of the "zw", "zW", "zWs" options is specified (identifies the target operating system as 16-bit Windows or 32-bit extended Windows but not Windows NT or Windows 95).

7. The _WINDOWS macro is defined when the build target is "WINDOWS" or one of the "zw", "zW", "zWs" options is specified and you are using a 16-bit compiler (identifies the target operating system as 16-bit Windows).

8. The __WINDOWS_386__ macro is defined when the build target is "WINDOWS" or the "zw" option is specified and you are using a 32-bit compiler (identifies the target operating system as 32-bit extended Windows).

9. The __LINUX__ and __UNIX__ macros are defined when the build target is "LINUX" (32-bit Linux).

The following macros are defined for the indicated options.

```
Option    Macro
====== ==================
bm      _MT
br      _DLL
fpi     __FPI__
fpi87   __FPI__
j       __CHAR_SIGNED__
oi      __INLINE_FUNCTIONS__
xr      _CPPRTTI (C++ only)
xs      _CPPUNWIND (C++ only)
xss     _CPPUNWIND (C++ only)
xst     _CPPUNWIND (C++ only)
za      NO_EXT_KEYS
zw      __WINDOWS__
zW      __WINDOWS__
zWs     __WINDOWS__
```

The following memory model macros are defined for the indicated memory model options.

```
Option     All        16-bit only          32-bit only
====== =========== =================    =================
mf      __FLAT__                         M_386FM  _M_386FM
ms      __SMALL__   M_I86SM  _M_I86SM    M_386SM  _M_386SM
mm      __MEDIUM__  M_I86MM  _M_I86MM    M_386MM  _M_386MM
mc      __COMPACT__ M_I86CM  _M_I86CM    M_386CM  _M_386CM
ml      __LARGE__   M_I86LM  _M_I86LM    M_386LM  _M_386LM
mh      __HUGE__    M_I86HM  _M_I86HM
```

The following macros indicate which compiler is compiling the C/C++ source code.

__cplusplus Open Watcom C++ predefines the macro __cplusplus to identify the compiler as a C++ compiler.

__WATCOMC__

Open Watcom C/C++ predefines the macro __WATCOMC__ to identify the compiler as one of the Open Watcom C/C++ compilers.

The value of the macro depends on the version number of the compiler. The value is 100 times the version number (version 8.5 yields 850, version 9.0 yields 900, etc.). Note that for Open Watcom 1.0, the value of this macro is 1200, for Open Watcom 1.1 it is 1210 etc.

__WATCOM_CPLUSPLUS__
Open Watcom C++ predefines the macro `__WATCOM_CPLUSPLUS__` to identify the compiler as one of the Open Watcom C++ compilers.

The value of the macro depends on the version number of the compiler. The value is 100 times the version number (version 10.0 yields 1000, version 10.5 yields 1050, etc.). Note that for Open Watcom 1.0, the value of this macro is 1200, for Open Watcom 1.1 it is 1210 etc.

The following macros are defined for compatibility with Microsoft.

__CPPRTTI Open Watcom C++ predefines the `__CPPRTTI` macro to indicate that C++ Run-Time Type Information (RTTI) is in force. This macro is predefined if the Open Watcom C++ "xr" compile option is specified and is not defined otherwise.

__CPPUNWIND
Open Watcom C++ predefines the `__CPPUNWIND` macro to indicate that C++ exceptions supported. This macro is predefined if any of the Open Watcom C++ "xs", "xss" or "xst" compile options are specified and is not defined otherwise.

_INTEGRAL_MAX_BITS
Open Watcom C/C++ predefines the `_INTEGRAL_MAX_BITS` macro to indicate that maximum number of bits supported in an integral type (see the description of the "__int64" keyword in the next section). Its value is 64 currently.

_PUSHPOP_SUPPORTED
Open Watcom C/C++ predefines the `_PUSHPOP_SUPPORTED` macro to indicate that `#pragma pack(push)` and `#pragma pack(pop)` are supported.

_STDCALL_SUPPORTED
Open Watcom C/C++ predefines the `_STDCALL_SUPPORTED` macro to indicate that the standard 32-bit Win32 calling convention is supported.

The following table summarizes the predefined macros supported by the compilers and the values that the respective compilers assign to them. A "yes" under the column means that the compiler supports the macro with the indicated value. Note that the C and C++ compilers

sometime support the same macro but with different values (including no value which means the symbol is defined without a value).

Predefined Macro and Setting	Supported by Compiler wcc	wcc386	wpp	wpp386
__386__=1		Yes		Yes
__3R__=1				Yes
_based=__based	Yes	Yes	Yes	Yes
_cdecl=__cdecl	Yes	Yes	Yes	Yes
cdecl=__cdecl	Yes	Yes	Yes	Yes
__cplusplus=1			Yes	Yes
_CPPRTTI=1			Yes	Yes
_CPPUNWIND=1			Yes	Yes
_export=__export	Yes	Yes	Yes	Yes
_far16=__far16	Yes	Yes	Yes	Yes
_far=__far	Yes	Yes	Yes	Yes
far=__far	Yes	Yes	Yes	Yes
_fastcall=__fastcall	Yes	Yes	Yes	Yes
__FLAT__=1		Yes		Yes
_fortran=__fortran	Yes	Yes	Yes	Yes
fortran=__fortran	Yes	Yes	Yes	Yes
__FPI__=1	Yes	Yes	Yes	Yes
_huge=__huge	Yes	Yes	Yes	Yes
huge=__huge	Yes	Yes	Yes	Yes
__I86__=1	Yes		Yes	
_inline=__inline	Yes	Yes	Yes	Yes
_INTEGRAL_MAX_BITS=64	Yes	Yes	Yes	Yes
_interrupt=__interrupt	Yes	Yes	Yes	Yes
interrupt=__interrupt	Yes	Yes	Yes	Yes
_loadds=__loadds	Yes	Yes	Yes	Yes
_M_386FM=1				Yes
M_386FM=1				Yes
_M_I386=1		Yes		Yes
M_I386=1		Yes		Yes
_M_I86=1	Yes		Yes	
M_I86=1	Yes		Yes	
_M_I86SM=1	Yes		Yes	
M_I86SM=1	Yes		Yes	
_M_IX86=0	Yes		Yes	
_M_IX86=500		Yes		Yes
_near=__near	Yes	Yes	Yes	Yes
near=__near	Yes	Yes	Yes	Yes
__NT__=1 (on Win32 platform)	Yes	Yes	Yes	Yes
_pascal=__pascal	Yes	Yes	Yes	Yes
pascal=__pascal	Yes	Yes	Yes	Yes
_saveregs=__saveregs	Yes	Yes	Yes	Yes
_segment=__segment	Yes	Yes	Yes	Yes
_segname=__segname	Yes	Yes	Yes	Yes
_self=__self	Yes	Yes	Yes	Yes
__SMALL__=1	Yes		Yes	
SOMDLINK=__far	Yes			
SOMDLINK=_Syscall		Yes		Yes
SOMLINK=__cdecl	Yes			
SOMLINK=_Syscall		Yes		Yes
_STDCALL_SUPPORTED=1		Yes		Yes
__SW_0=1	Yes		Yes	
__SW_3R=1		Yes		Yes
__SW_5=1		Yes		Yes
__SW_FP287=1			Yes	
__SW_FP2=1	Yes			

__SW_FP387=1				Yes
__SW_FP3=1		Yes		
__SW_FPI=1	Yes	Yes	Yes	Yes
__SW_MF=1		Yes		Yes
__SW_MS=1	Yes			
__SW_ZDP=1	Yes	Yes	Yes	Yes
__SW_ZFP=1	Yes	Yes	Yes	Yes
__SW_ZGF=1		Yes		Yes
__SW_ZGP=1	Yes		Yes	
_stdcall=__stdcall	Yes	Yes	Yes	Yes
_syscall=__syscall	Yes	Yes	Yes	Yes
__WATCOM_CPLUSPLUS__=1240			Yes	Yes
__WATCOMC__=1240	Yes	Yes	Yes	Yes
__X86__=1	Yes	Yes	Yes	Yes

3.9 Open Watcom C/C++ Extended Keywords

Open Watcom C/C++ supports the use of some special keywords to describe system dependent attributes of functions and other object names. These attributes are inspired by the Intel processor architecture and the plethora of function calling conventions in use by compilers for this architecture. In keeping with the ISO C and C++ language standards, Open Watcom C/C++ uses the double underscore (i.e., "__") or single underscore followed by uppercase letter (e.g., "_S") prefix with these keywords. To support compatibility with other C/C++ compilers, alternate forms of these keywords are also supported through predefined macros.

__near Open Watcom C/C++ supports the `__near` keyword to describe functions and other object names that are in near memory and pointers to near objects.

Open Watcom C/C++ predefines the macros `near` and `_near` to be equivalent to the `__near` keyword.

__far Open Watcom C/C++ supports the `__far` keyword to describe functions and other object names that are in far memory and pointers to far objects.

Open Watcom C/C++ predefines the macros `far`, `_far` and `SOMDLINK` (16-bit only) to be equivalent to the `__far` keyword.

__huge Open Watcom C/C++ supports the `__huge` keyword to describe functions and other object names that are in huge memory and pointers to huge objects. The 32-bit compilers treat these as equivalent to far objects.

Open Watcom C/C++ predefines the macros `huge` and `_huge` to be equivalent to the `__huge` keyword.

__based Open Watcom C/C++ supports the `__based` keyword to describe pointers to objects that appear in other segments or the objects themselves. See the section entitled "Based Pointers" on page 39 for an explanation of the `__based` keyword.

Open Watcom C/C++ predefines the macro `_based` to be equivalent to the `__based` keyword.

__segment Open Watcom C/C++ supports the `__segment` keyword which is used when describing objects of type segment. See the section entitled "Based Pointers" on page 39 for an explanation of the `__segment` keyword.

Open Watcom C/C++ predefines the macro `_segment` to be equivalent to the `__segment` keyword.

__segname Open Watcom C/C++ supports the `__segname` keyword which is used when describing segname constant based pointers or objects. See the section entitled "Based Pointers" on page 39 for an explanation of the `__segname` keyword.

Open Watcom C/C++ predefines the macro `_segname` to be equivalent to the `__segname` keyword.

__self Open Watcom C/C++ supports the `__self` keyword which is used when describing self based pointers. See the section entitled "Based Pointers" on page 39 for an explanation of the `__self` keyword.

Open Watcom C/C++ predefines the macro `_self` to be equivalent to the `__self` keyword.

_Packed Open Watcom C/C++ supports the `_Packed` keyword which is used when describing a structure. If specified before the ***struct*** keyword, the compiler will force the structure to be packed (no alignment, no gaps) regardless of the setting of the command-line option or the ***#pragma*** controlling the alignment of members.

__cdecl Open Watcom C/C++ supports the `__cdecl` keyword to describe C functions that are called using a special convention.

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine returns a pointer in register AX/EAX to the return value which is stored in the data segment (DGROUP).

4. For the 16-bit compiler, registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

5. For the 32-bit compiler, registers EAX, ECX and EDX are not saved and restored when a call is made.

Open Watcom C/C++ predefines the macros `cdecl`, `_cdecl`, `_Cdecl` and `SOMLINK` (16-bit only) to be equivalent to the `__cdecl` keyword.

__pascal Open Watcom C/C++ supports the `__pascal` keyword to describe Pascal functions that are called using a special convention described by a pragma in the "stddef.h" header file.

Open Watcom C/C++ predefines the macros `pascal`, `_pascal` and `_Pascal` to be equivalent to the `__pascal` keyword.

__fortran Open Watcom C/C++ supports the `__fortran` keyword to describe functions that are called from FORTRAN. It converts the name to uppercase letters and suppresses the "_" which is appended to the function name for certain calling conventions.

Open Watcom C/C++ predefines the macros `fortran` and `_fortran` to be equivalent to the `__fortran` keyword.

__interrupt Open Watcom C/C++ supports the `__interrupt` keyword to describe a function that is an interrupt handler.

Example:
```
#include <i86.h>

void __interrupt int10( union INTPACK r )
{
        .
        .
        .
}
```

The code generator will emit instructions to save all registers. The registers are saved on the stack in a specific order so that they may be referenced using the "INTPACK" union as shown in the DOS example above. The code generator will emit instructions to establish addressability to the program's data segment since the DS segment register contents are unpredictable. The function will return using an "IRET" (16-bit) or "IRETD" (32-bit) (interrupt return) instruction.

Open Watcom C/C++ predefines the macros `interrupt` and `_interrupt` to be equivalent to the `__interrupt` keyword.

__declspec(modifier)
Open Watcom C/C++ supports the `__declspec` keyword for compatibility with Microsoft C++. The `__declspec` keyword is used to modify storage-class attributes of functions and/or data. There are several modifiers that can be specified with the `__declspec` keyword: `thread`, `naked`, `dllimport`, `dllexport`, `__pragma( "string" )`, `__cdecl`, `__pascal`, `__fortran`, `__stdcall`, and `__syscall`. These attributes are a property only of the declaration of the object or function to which they are applied. Unlike the `__near` and `__far` keywords, which actually affect the type of object or function (in this case, 2- and 4-byte addresses), these storage-class attributes do not redefine the type attributes of the object itself. The `__pragma` modifier is supported by Open Watcom C++ only. The `thread` attribute affects data and objects only. The `naked`, `__pragma`, `__cdecl`, `__pascal`, `__fortran`, `__stdcall`, and `__syscall` attributes affect functions only. The `dllimport` and `dllexport` attributes affect functions, data, and objects. For more information on the `__declspec` keyword, please see the section entitled "The __declspec Keyword" on page 44.

__export Open Watcom C/C++ supports the `__export` keyword to describe functions and other object names that are to be exported from a Microsoft Windows DLL, OS/2 DLL, or Netware NLM. See also the description of the "zu" option.

Example:
```
void __export _Setcolor( int color )
{
          .
          .
          .
}
```

Open Watcom C/C++ predefines the macro `_export` to be equivalent to the `__export` keyword.

__loadds Open Watcom C/C++ supports the `__loadds` keyword to describe functions that require specific loading of the DS register to establish addressability to the function's data segment. This keyword is useful in describing a function that will be placed in a Microsoft Windows or OS/2 1.x Dynamic Link Library (DLL). See also the description of the "nd" and "zu" options.

Example:
```
void __export __loadds _Setcolor( int color )
{
          .
          .
          .
}
```

If the function in an OS/2 1.x Dynamic Link Library requires access to private data, the data segment register must be loaded with an appropriate value since it will contain the DS value of the calling application upon entry to the function.

Open Watcom C/C++ predefines the macro `_loadds` to be equivalent to the `__loadds` keyword.

__saveregs Open Watcom C/C++ recognizes the `__saveregs` keyword which is an attribute used by C/C++ compilers to describe a function that must save and restore all registers.

Open Watcom C/C++ predefines the macro `_saveregs` to be equivalent to the `__saveregs` keyword.

__stdcall (32-bit only) The `__stdcall` keyword may be used with function definitions, and indicates that the 32-bit Win32 calling convention is to be used.

Notes:

1. All symbols are preceded by an underscore character.

2. All C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted.

3. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The called routine will remove the arguments from the stack.

4. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

5. Registers EAX, ECX and EDX are not saved and restored when a call is made.

__syscall (32-bit only) The `__syscall` keyword may be used with function definitions, and indicates that the calling convention used is compatible with functions provided by 32-bit OS/2.

Notes:

1. Symbols names are not modified, that is, they are not adorned with leading or trailing underscores.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

Open Watcom C/C++ predefines the macros `_syscall`, `_System`, `SOMLINK` (32-bit only) and `SOMDLINK` (32-bit only) to be equivalent to the `__syscall` keyword.

__far16 (32-bit only) Open Watcom C/C++ recognizes the `__far16` keyword which can be used to define far 16-bit (far16) pointers (16-bit selector with 16-bit offset) or far 16-bit function prototypes. This keyword can be used under 32-bit OS/2 to call 16-bit functions from your 32-bit flat model program. Integer arguments will automatically be converted to 16-bit integers, and 32-bit pointers will be converted to far16 pointers before calling a special thunking layer to transfer control to the 16-bit function.

Open Watcom C/C++ predefines the macros `_far16` and `_Far16` to be equivalent to the `__far16` keyword. This keyword is compatible with Microsoft C.

In the OS/2 operating system (version 2.0 or higher), the first 512 megabytes of the 4 gigabyte segment referenced by the DS register is divided into 8192 areas of 64K bytes each. A far16 pointer consists of a 16-bit selector referring to one of the 64K byte areas, and a 16-bit offset into that area.

A pointer declared as,

    [type] __far16 *name;

defines an object that is a far16 pointer. If such a pointer is accessed in the 32-bit environment, the compiler will generate the necessary code to convert between the far16 pointer and a "flat" 32-bit pointer.

For example, the declaration,

```
char __far16 *bufptr;
```

declares the object `bufptr` to be a far16 pointer to ***char.***

A function declared as,

    [type] __far16 func([arg_list]);

declares a 16-bit function. Any calls to such a function from the 32-bit environment will cause the compiler to convert any 32-bit pointer arguments to far16 pointers, and any ***int*** arguments from 32 bits to 16 bits. (In the 16-bit environment, an object of type ***int*** is only 16 bits.) Any return value from the function will have its return value converted in an appropriate manner.

For example, the declaration,

```
char * __far16 Scan( char *buffer, int len, short err );
```

declares the 16-bit function Scan. When this function is called from the 32-bit environment, the buffer argument will be converted from a flat 32-bit pointer to a far16 pointer (which, in the 16-bit environment, would be declared as char __far *. The len argument will be converted from a 32-bit integer to a 16-bit integer. The err argument will be passed unchanged. Upon returning, the far16 pointer (far pointer in the 16-bit environment) will be converted to a 32-bit pointer which describes the equivalent location in the 32-bit address space.

_Seg16 (32-bit only) Open Watcom C/C++ recognizes the _Seg16 keyword which has a similar but not identical function as the __far16 keyword described above. This keyword is compatible with IBM C Set/2 and IBM VisualAge C++.

In the OS/2 operating system (version 2.0 or higher), the first 512 megabytes of the 4 gigabyte segment referenced by the DS register is divided into 8192 areas of 64K bytes each. A far16 pointer consists of a 16-bit selector referring to one of the 64K byte areas, and a 16-bit offset into that area.

Note that _Seg16 is **not** interchangeable with __far16.

A pointer declared as,

[type] * _Seg16 name;

defines an object that is a far16 pointer. Note that the _Seg16 appears on the right side of the * which is opposite to the __far16 keyword described above.

For example,

```
char * _Seg16 bufptr;
```

declares the object bufptr to be a far16 pointer to ***char*** (the same as above).

The _Seg16 keyword may not be used to describe a 16-bit function. A ***#pragma*** directive must be used instead. A function declared as,

[type] * _Seg16 func([parm_list]);

declares a 32-bit function that returns a far16 pointer.

For example, the declaration,

```
char * _Seg16 Scan( char * buffer, int len, short err );
```

declares the 32-bit function `Scan`. No conversion of the argument list will take place. The return value is a far16 pointer.

__pragma Open Watcom C++ supports the `__pragma` keyword to support in-lining of member functions. The `__pragma` keyword must be followed by parentheses containing a string that names an auxiliary pragma. Here is a simplified example showing usage and syntax.

Example:

```
#pragma aux fast_mul = \
     "imul eax,edx" \
     parm caller [eax] [edx] \
     value struct;

struct fixed {
     unsigned v;
};

fixed __pragma( "fast_mul") operator *( fixed,
fixed );

fixed two = { 2 };
fixed three = { 3 };

fixed foo()
{
     return two * three;
}
```

See the chapters entitled "16-bit Pragmas" on page 93 and "32-bit Pragmas" on page 179 for more information on pragmas.

__int64 Open Watcom C/C++ supports the `__int64` keyword to define 64-bit integer data objects.

Example:
```
static __int64 bigInt;
```

Also supported are signed and unsigned 64-bit integer constants.

signed __int64 Use the "i64" suffix for a signed 64-bit integer constant.

Example:
```
12345i64
12345I64
```

unsigned __int64 Use the "ui64" suffix for a signed 64-bit integer constant.

Example:
```
12345Ui64
12345uI64
```

The run-time library supports formatting of `__int64` items (see the description of the `printf` library function).

Example:
```
#include <stdio.h>
#include <limits.h>

void main()
{
    __int64 bigint;
    __int64 bigint2;

    bigint2 = 8I64 * (LONG_MAX + 1I64);
    for( bigint = 0;
         bigint <= bigint2;
         bigint += bigint2 / 16 ) {
        printf( "Hello world %Ld\n", bigint );
    }
}
```

Restrictions

switch An `__int64` expression cannot be used in a *switch* statement.

bit fields More than 32 bits in a 64-bit bitfield is not supported.

3.10 Based Pointers

Near pointers are generally the most efficient type of pointer because they are small, and the compiler can assume knowledge about what segment of the computer's memory the pointer (offset) refers to. Far pointers are the most flexible because they allow the programmer to access any part of the computer's memory, without limitation to a particular segment. However, far pointers are bigger and slower because of the additional flexibility.

Based pointers are a compromise between the efficiency of near pointers and the flexibility of far pointers. With based pointers, the programmer takes responsibility to tell the compiler which segment a near pointer (offset) belongs to, but may still access segments of the computer's memory outside of the normal data segment (DGROUP). The result is a pointer type which is as small as and almost as efficient as a near pointer, but with most of the flexibility of a far pointer.

An object declared as a based pointer falls into one of the following categories:

- the based pointer is in the segment described by another object,
- the based pointer, used as a pointer to another object of the same type (as in a linked list), refers to the same segment,
- the based pointer is an offset to no particular segment, and must be combined explicitly with a segment value to produce a valid pointer.

To support based pointers, the following keywords are provided:

```
__based
__segment
__segname
__self
```

The following operator is also provided:

```
:>
```

These keywords and operator are described in the following sections.

Two macros, defined in `malloc.h,` are also provided:

```
_NULLSEG
_NULLOFF
```

They are used in a manner similar to `NULL,` but are used with objects declared as `__segment` and `__based` respectively.

3.10.1 Segment Constant Based Pointers and Objects

A segment constant based pointer or object has its segment value based on a specific, named segment. A segment constant based object is specified as:

[type] __based(__segname("segment")) object_name;

and a segment constant based pointer is specified as:

[type] __based(__segname("segment")) *object-name;

where `segment` is the name of the segment in which the pointer or object is based. As shown above, the segment name is always specified as a string. There are three special segment names recognized by the compiler:

```
"_CODE"
"_CONST"
"_DATA"
```

The `"_CODE"` segment is the default code segment. The `"_CONST"` segment is the segment containing constant values. The `"_DATA"` segment is the default data segment. If the segment name is not one of the three recognized names, then a segment will be created with that name. If a segment constant based object is being defined, then it will be placed in the named segment. If a segment constant based pointer is being defined, then it can point at objects in the named segment.

The following examples illustrate segment constant based pointers and objects.

Example:

```
int __based( __segname( "_CODE" ) )  ival = 3;
int __based( __segname( "_CODE" ) ) *iptr;
```

`ival` is an object that resides in the default code segment. `iptr` is an object that resides in the data segment (the usual place for data objects), but points at an integer which resides in the default code segment. `iptr` is suitable for pointing at `ival`.

Example:
```
char __based( __segname( "GOODTHINGS" ) ) thing;
```

`thing` is an object which resides in the segment `GOODTHINGS,` which will be created if it does not already exist. (The creation of segments is done by the linker, and is a method of grouping objects and functions. Nothing is implicitly created during the execution of the program.)

3.10.2 Segment Object Based Pointers

A segment object based pointer derives its segment value from another named object. A segment object based pointer is specified as follows:

[type] __based(segment) *name;

where `segment` is an object defined as type `__segment.`

An object of type `__segment` may contain a segment value. Such an object is particularly designed for use with segment object based pointers.

The following example illustrates a segment object based pointer:

Example:
```
__segment             seg;
char __based( seg ) *cptr;
```

The object `seg` contains only a segment value. Whenever the object `cptr` is used to point to a character, the actual pointer value will be made up of the segment value found in `seg` and the offset value found in `cptr.` The object `seg` might be assigned values such as the following:

- a constant value (e.g., the segment containing screen memory),
- the result of the library function ***_bheapseg,***
- the segment portion of another pointer value, by casting it to the type `__segment.`

3.10.3 Void Based Pointers

A void based pointer must be explicitly combined with a segment value to produce a reference to a memory location. A void based pointer does not infer its segment value from another object. The `:>` (base) operator is used to combine a segment value and a void based pointer.

For example, on a personal computer running DOS with a color monitor, the screen memory begins at segment 0xB800, offset 0. In a video text mode, to examine the first character currently displayed on the screen, the following code could be used:

Example:
```
extern void main()
{
    __segment              screen;
    char __based( void ) *scrptr;

    screen = 0xB800;
    scrptr = 0;
    printf( "Top left character is '%c'.\n",
            *(screen:>scrptr) );
}
```

The general form of the :> operator is:

segment :> offset

where `segment` is an expression of type `__segment`, and `offset` is an expression of type `__based( void ) *`.

3.10.4 Self Based Pointers

A self based pointer infers its segment value from itself. It is particularly useful for structures such as linked lists, where all of the list elements are in the same segment. A self based pointer pointing to one element may be used to access the next element, and the compiler will use the same segment as the original pointer.

The following example illustrates a function which will print the values stored in the last two members of a linked list:

Example:
```
struct a {
    struct a __based( __self ) *next;
    int                         number;
};
```

```
extern void PrintLastTwo( struct a far *list )
{
  __segment                seg;
  struct a __based( seg ) *aptr;

  seg  = FP_SEG( list );
  aptr = FP_OFF( list );
  for( ; aptr != _NULLOFF; aptr = aptr->next ) {
    if( aptr->next == _NULLOFF ) {
      printf( "Last item is %d\n",
              aptr->number );
    } else if( aptr->next->next == _NULLOFF ) {
      printf( "Second last item is %d\n",
              aptr->number );
    }
  }
}
```

The argument to the function `PrintLastTwo` is a far pointer, pointing to a linked list structure anywhere in memory. It is assumed that all members of a particular linked list of this type reside in the same segment of the computer's memory. (Another instance of the linked list might reside entirely in a different segment.) The object `seg` is given the segment portion of the far pointer. The object `aptr` is given the offset portion, and is described as being based in the segment stored in `seg.`

The expression `aptr->next` refers to the `next` member of the structure stored in memory at the offset stored in `aptr` and the segment implied by `aptr,` which is the value stored in `seg.` So far, the behavior is no different than if `next` had been declared as,

```
struct a *next;
```

The expression `aptr->next->next` illustrates the difference of using a self based pointer. The first part of the expression (`aptr->next`) occurs as described above. However, using the result to point to the next member occurs by using the offset value found in the `next` member and combining it with the segment value of the *pointer used to get to that member,* which is still the segment implied by `aptr,` which is the value stored in `seg.` If `next` had not been declared using `__based( __self ),` then the second pointing operation would refer to the offset value found in the `next` member, but with the default data segment (DGROUP), which may or may not be the same segment as stored in `seg.`

3.11 The __declspec Keyword

Open Watcom C/C++ supports the `__declspec` keyword for compatibility with Microsoft C++. The `__declspec` keyword is used to modify storage-class attributes of functions and/or data.

__declspec(thread) is used to define thread local storage (TLS). TLS is the mechanism by which each thread in a multithreaded process allocates storage for thread-specific data. In standard multithreaded programs, data is shared among all threads of a given process, whereas thread local storage is the mechanism for allocating per-thread data.

Example:
```
__declspec(thread) static int tls_data = 0;
```

The following rules apply to the use of the `thread` attribute.

- The `thread` attribute can be used with data and objects only.

- You can specify the `thread` attribute only on data items with static storage duration. This includes global data objects (both `static` and `extern`), local static objects, and static data members of classes. Automatic data objects cannot be declared with the `thread` attribute. The following example illustrates this error:

  Example:
  ```
  #define TLS __declspec( thread )
  void func1()
  {
      TLS int tls_data;            // Wrong!
  }

  int func2( TLS int tls_data )    // Wrong!
  {
      return tls_data;
  }
  ```

- The `thread` attribute must be used for both the declaration and the definition of a thread local object, whether the declaration and definition occur in the same file or separate files. The following example illustrates this error:

Example:
```
#define TLS __declspec( thread )
extern int tls_data;    // This generates an
error, because the
TLS    int tls_data;    // declaration and the
definition differ.
```

- Classes cannot use the `thread` attribute. However, you can instantiate class objects with the `thread` attribute, as long as the objects do not need to be constructed or destructed. For example, the following code generates an error:

Example:
```
#define TLS __declspec( thread )
TLS class A     // Wrong! Classes are not
objects
{
    // Code
};
A AObject;
```

Because the declaration of objects that use the `thread` attribute is permitted, these two examples are semantically equivalent:

Example:
```
#define TLS __declspec( thread )
TLS class B
{
    // Code
} BObject;      // Okay! BObject declared
thread local.

class C
{
    // Code
};
TLS C CObject;  // Okay! CObject declared
thread local.
```

- Standard C permits initialization of an object or variable with an expression involving a reference to itself, but only for objects of non-static extent. Although C++ normally permits such dynamic initialization of an object with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects.

Example:

```
#define TLS  __declspec( thread )
TLS int tls_i = tls_i;            // C and C++
error
int j = j;                        // Okay in
C++; C error
TLS int tls_k = sizeof( tls_k );  // Okay in C
and C++
```

Note that a `sizeof` expression that includes the object being initialized does not constitute a reference to itself and is allowed in C and C++.

__declspec(naked) indicates to the code generator that no prologue or epilogue sequence is to be generated for a function. Any statements other than "_asm" directives or auxiliary pragmas are not compiled. ***_asm*** Essentially, the compiler will emit a "label" with the specified function name into the code.

Example:

```
#include <stdio.h>

int __declspec( naked ) foo( int x )
{
    _asm {
#if defined(__386__)
        inc eax
#else
        inc ax
#endif
        ret
    }
}

void main()
{
    printf( "%d\n", foo( 1 ) );
}
```

The following rules apply to the use of the `naked` attribute.

- The `naked` attribute cannot be used in a data declaration. The following declaration would be flagged in error.

Example:

```
__declspec(naked) static int data_object = 0;
```

__declspec(dllimport) is used to declare functions, data and objects imported from a DLL.

Example:

```
#define DLLImport __declspec(dllimport)

DLLImport void dll_func();
DLLImport int  dll_data;
```

Functions, data and objects are exported from a DLL by use of `__declspec(dllexport)`, the `__export` keyword (for which `__declspec(dllexport)` is the replacement), or through linker "EXPORT" directives.

Note: When calling functions imported from other modules, it is not strictly necessary to use the `__declspec(dllimport)` modifier to declare the functions. This modifier however must always be used when importing data or objects to ensure correct behavior.

__declspec(dllexport) is used to declare functions, data and objects exported from a DLL. Declaring functions as `dllexport` eliminates the need for linker "EXPORT" directives. The `__declspec(dllexport)` attribute is a replacement for the `__export` keyword.

__declspec(__pragma("string")) is used to declare functions which adhere to the conventions described by the pragma identified by "string".

Example:

```
#include <stdio.h>

#pragma aux my_stdcall "_*" \
        parm routine [] \
        value struct struct caller [] \
        modify [eax ecx edx];

struct list {
    struct list *next;
    int          value;
    float        flt_value;
};

#define STDCALL __declspec( __pragma("my_stdcall")
)

STDCALL struct list foo( int x, char *y, double z
);

void main()
{
    int a = 1;
    char *b = "Hello there";
    double c = 3.1415926;
    struct list t;

    t = foo( a, b, c );
    printf( "%d\n", t.value );
}


struct list foo( int x, char *y, double z )
{
    struct list tmp;

    printf( "%s\n", y );
    tmp.next = NULL;
    tmp.value = x;
    tmp.flt_value = z;
    return( tmp );
}
```

The `__pragma` modifier is supported by Open Watcom C++ only.

__declspec(__cdecl) is used to declare functions which conform to the Microsoft compiler calling convention.

__declspec(__pascal) is used to declare functions which conform to the OS/2 1.x and Windows 3.x calling convention.

__declspec(__fortran) is used to declare functions which conform to the ***__fortran*** calling convention.

Example:

```
#include <stdio.h>

#define DLLFunc __declspec(dllimport __fortran)
#define DLLData __declspec(dllimport)

#ifdef __cplusplus
extern "C" {
#endif

DLLFunc int  dll_func( int, int, int );
DLLData int  dll_data;

#ifdef __cplusplus
};
#endif

void main()
{
  printf( "%d %d\n", dll_func( 1,2,3 ), dll_data
);
}
```

__declspec(__stdcall) is used to declare functions which conform to the 32-bit Win32 "standard" calling convention.

Example:

```
#include <stdio.h>

#define DLLFunc __declspec(dllimport __stdcall)
#define DLLData __declspec(dllimport)

DLLFunc int  dll_func( int, int, int );
DLLData int  dll_data;

void main()
{
  printf( "%d %d\n", dll_func( 1,2,3 ), dll_data
);
}
```

__declspec(__syscall) is used to declare functions which conform to the 32-bit OS/2 ***__syscall*** calling convention.

3.12 The Open Watcom Code Generator

The Open Watcom Code Generator performs such optimizations as common subexpression elimination, global flow analysis, and so on.

In some cases, the code generator can do a better job of optimizing code if it could utilize more memory. This is indicated when a

```
Not enough memory to optimize procedure 'xxxx'
```

message appears on the screen as the source program is compiled. In such an event, you may wish to make more memory available to the code generator.

A special environment variable may be used to obtain memory usage information or set memory usage limits on the code generator. The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:
```
$ export "WCGMEMORY=?"
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:
```
$ export "WCGMEMORY=128"
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use

this feature. To generate identical code on two personal computers with different memory configurations, you must ensure that the **WCGMEMORY** environment variable is set identically on both machines.

4 Precompiled Headers

4.1 Using Precompiled Headers

Open Watcom C/C++ supports the use of precompiled headers to decrease the time required to compile several source files that include the same header file.

4.2 When to Precompile Header Files

Using precompiled headers reduces compilation time when:

- You always use a large body of code that changes infrequently.
- Your program comprises multiple modules, all of which use the same first include file and the same compilation options. In this case, the first include file along with all the files that it includes can be precompiled into one precompiled header.

Because the compiler only uses the first include file to create a precompiled header, you may want to create a master or global header file that includes all the other header files that you wish to have precompiled. Then all source files should include this master header file as the first `#include` in the source file. Even if you don't use a master header file, you can benefit from using precompiled headers for Windows programs by using `#include <windows.h>` as the first include file, or by using `#include <afxwin.h>` as the first include file for MFC applications.

The first compilation — the one that creates the precompiled header file — takes a bit longer than subsequent compilations. Subsequent compilations can proceed more quickly by including the precompiled header.

You can precompile C and C++ programs. In C++ programming, it is common practice to separate class interface information into header files which can later be included in programs that use the class. By precompiling these headers, you can reduce the time a program takes to compile.

> ***Note:*** Although you can use only one precompiled header (.pch) file per source file, you can use multiple .pch files in a project.

4.3 Creating and Using Precompiled Headers

Precompiled code is stored in a file called a precompiled header when you use the precompiled header option (**-fh** or **-fhq**) on the command line. The **-fh** option causes the compiler to either create a precompiled header or use the precompiled header if it already exists. The **-fhq** option is similar but prevents the compiler from issuing informational or warning messages about precompiled header files. The default name of the precompiled header file is one of wcc.pch, wcc386.pch, wpp.pch, or wpp386.pch (depending on the compiler used). You can also control the name of the precompiled header that is created or used with the **-fh=filename** or **-fhq=filename** ("specify precompiled header filename") options.

Example:

```
-fh=projectx.pch
-fhq=projectx.pch
```

4.4 The "-fh[q]" (Precompiled Header) Option

The **-fh** option instructs the compiler to use a precompiled header file with a default name of wcc.pch, wcc386.pch, wpp.pch, or wpp386.pch (depending on the compiler used) if it exists or to create one if it does not. The file is created in the current directory. You can use the **-fh=filename** option to change the default name (and placement) of the precompiled header. Add the letter "q" (for "quiet") to the option name to prevent the compiler from displaying precompiled header activity information.

The following command line uses the **-fh** option to create a precompiled header.

Example:

```
wpp -fh myprog.cpp
wpp386 -fh myprog.cpp
```

The following command line creates a precompiled header named myprog.pch and places it in the /projpch directory.

Example:
```
wpp -fh=/projpch/myprog.pch myprog.cpp
wpp386 -fh=/projpch/myprog.pch myprog.cpp
```

The precompiled header is created and/or used when the compiler encounters the first `#include` directive that occurs in the source file. In a subsequent compilation, the compiler performs a consistency check to see if it can use an existing precompiled header. If the consistency check fails then the compiler discards the existing precompiled header and builds a new one.

The **-fhq** form of the precompiled header option prevents the compiler from issuing warning or informational messages about precompiled header files. For example, if you change a header file, the compiler will tell you that it changed and that it must regenerate the precompiled header file. If you specify **-fhq** then the compiler just generates the new precompiled header file without displaying a message.

4.5 Consistency Rules for Precompiled Headers

If a precompiled header file exists (either the default file or one specified by **-fh=filename**), it is compared to the current compilation for consistency. A new precompiled header file is created and the new file overwrites the old unless the following requirements are met:

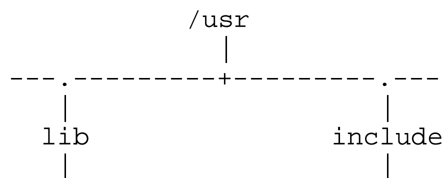
- The current compiler options must match those specified when the precompiled header was created.
- The current working directory must match that specified when the precompiled header was created.
- The name of the first `#include` directive must match the one that was specified when the precompiled header was created.
- All macros defined prior to the first `#include` directive must have the same values as the macros defined when the precompiled header was created. A sequence of `#define` directives need not occur in exactly the same order because there are no semantic order dependencies for `#define` directives.
- The value and order of include paths specified on the command line with **-i** options must match those specified when the precompiled header was created.
- The time stamps of all the header files (all files specified with `#include` directives) used to build the precompiled header must match those that existed when the precompiled header was created.

5 The Open Watcom C/C++ Libraries

The Open Watcom C/C++ library routines are described in the ***Open Watcom C Library Reference*** manual, and the ***Open Watcom C++ Class Library Reference*** manual.

5.1 Open Watcom C/C++ Library Directory Structure

The Open Watcom C/C++ libraries are located under the `/usr/lib` directory.



5.2 Open Watcom C/C++ C Libraries

Due to the many code generation strategies possible in the 80x86 family of processors, a number of versions of the libraries are provided. You must use the libraries which coincide with the particular code generation strategy or model that you have selected. For the type of code generation strategy or model that you intend to use, refer to the description of the "m?" memory model compiler option. The various code models supported by Open Watcom C/C++ are described in the chapters entitled "16-bit Memory Models" on page 67 and "32-bit Memory Models" on page 149.

We have selected a simple naming convention for the libraries that are provided with Open Watcom C/C++. Letters are affixed to the file name to indicate the particular strategy with which the modules in the library have been compiled.

16-bit only

S denotes a version of the Open Watcom C/C++ libraries which have been compiled for the "small" memory model (small code, small data).

M denotes a version of the Open Watcom C/C++ libraries which have been compiled for the "medium" memory model (big code, small data).

C denotes a version of the Open Watcom C/C++ libraries which have been compiled for the "compact" memory model (small code, big data).

L denotes a version of the Open Watcom C/C++ libraries which have been compiled for the "large" memory model (big code, big data).

H denotes a version of the Open Watcom C/C++ libraries which have been compiled for the "huge" memory model (big code, huge data).

32-bit only

3R denotes a version of the Open Watcom C/C++ libraries that will be used by programs which have been compiled for the "flat/small" memory models using the "3r", "4r" or "5r" option.

3S denotes a version of the Open Watcom C/C++ libraries that will be used by programs which have been compiled for the "flat/small" memory models using the "3s", "4s" or "5s" option.

The Open Watcom C/C++ 16-bit libraries are listed below.

```
clibs.lib    (small model support)
clibm.lib    (medium model support)
clibc.lib    (compact model support)
clibl.lib    (large model support)
clibh.lib    (huge model support)
```

The Open Watcom C/C++ 32-bit libraries are listed below.

```
clib3r.lib   (flat/small models, "3r", "4r" or "5r" option)
clib3s.lib   (flat/small models, "3s", "4s" or "5s" option)
```

5.3 Open Watcom C 16-bit Shared Library

A portion of the 16-bit Open Watcom C Library is also stored in a memory-resident library called the system *shared library.* On multi-tasking systems, it makes sense that commonly-used library routines such as *read* and *write* be shared among processes. By sharing the same code, the memory requirement for applications is reduced. The functions in the shared library are memory model independent so they can be used by any small/large code, small/large/huge data applications.

5.4 Open Watcom C/C++ Class Libraries

The Open Watcom C/C++ Class Library routines are described in the ***Open Watcom C++ Class Library Reference*** manual.

The Open Watcom C++ 16-bit Class Libraries are listed below.

```
    (iostream and string class libraries)
plibs.lib    (small model support)
plibm.lib    (medium model support)
plibc.lib    (compact model support)
plibl.lib    (large model support)
plibh.lib    (huge model support)
    (complex class library for "fpc" option)
cplxs.lib    (small model support)
cplxm.lib    (medium model support)
cplxc.lib    (compact model support)
cplxl.lib    (large model support)
cplxh.lib    (huge model support)
    (complex class library for "fpi..." options)
cplx7s.lib   (small model support)
cplx7m.lib   (medium model support)
cplx7c.lib   (compact model support)
cplx7l.lib   (large model support)
cplx7h.lib   (huge model support)
```

These libraries are independent of the operating system. The "7" designates a library compiled with the "7" option.

The Open Watcom C++ 32-bit Class Libraries are listed below.

```
    (iostream and string class libraries)
plib3r.lib   (flat models, "3r", "4r" or "5r" option)
plib3s.lib   (flat models, "3s", "4s" or "5s" option)
    (complex class library for "fpc" option)
cplx3r.lib   (flat models, "3r", "4r" or "5r" option)
cplx3s.lib   (flat models, "3s", "4s" or "5s" option)
    (complex class library for "fpi..." options)
cplx73r.lib  (flat models, "3r", "4r" or "5r" option)
cplx73s.lib  (flat models, "3s", "4s" or "5s" option)
```

These libraries are independent of the operating system. The "3r" and "3s" suffixes refer to the argument passing convention used. The "7" designates a library compiled with the "7" option.

5.5 Open Watcom C/C++ Math Libraries

In general, a Math library is required when floating-point computations are included in the application. The Math libraries are operating-system independent. The Math libraries are placed under the `/usr/lib` directory.

The following situations indicate that one of the Math libraries should be included when linking the application.

1. When one or more of the functions described in the `math.h` header file is referenced, then a Math library must be included.

2. If an application is linked and the message

   ```
   "_fltused_ is an undefined reference"
   ```

   appears, then a Math library must be included.

3. (16-bit only) If an application is linked and the message

   ```
   "__init_87_emulator is an undefined reference"
   ```

   appears, then one of the modules in the application was compiled with one of the "fpi", "fpi87", "fp2", "fp3" or "fp5" options. If the "fpi" option was used, the 80x87 emulator library (`emu87.lib`) or the 80x87 fixup library (`noemu87.lib`) should be included when linking the application.

   If the "fpi87" option was used, the 80x87 fixup library `noemu87.lib` should be included when linking the application.

4. (32-bit only) If an application is linked and the message

   ```
   "__init_387_emulator is an undefined reference"
   ```

   appears, then one of the modules in the application was compiled with one of the "fpi", "fpi87", "fp2", "fp3" or "fp5" options. If the "fpi" option was used, the 80x87 emulator library (`emu387.lib`) should be included when linking the application.

   If the "fpi87" option was used, the empty 80x87 emulator library `noemu387.lib` should be included when linking the application.

Normally, the compiler and linker will automatically take care of this. Simply ensure that the **WATCOM** environment variable includes the location of the Open Watcom C/C++ libraries.

5.6 Open Watcom C/C++ 80x87 Math Libraries

One of the following Math libraries must be used if any of the modules of your application were compiled with one of the Open Watcom C/C++ "fpi", "fpi87", "fp2", "fp3" or "fp5" options and your application requires floating-point support for the reasons given above.

16-bit libraries:

```
math87s.lib (small model)
math87m.lib (medium model)
math87c.lib (compact model)
math87l.lib (large model)
math87h.lib (huge model)
noemu87.lib
emu87.lib   (QNX dependent)
```

32-bit libraries:

```
math387r.lib (flat/small models, "3r", "4r" or "5r" option)
math387s.lib (flat/small models, "3s", "4s" os "5s" option)
emu387.lib   (QNX dependent)
```

The "fpi" option causes an 80x87 numeric data processor emulator to be linked into your application in addition to any 80x87 math routines that were referenced. For QNX, there is a common 80x87 emulator task that is used so that there is one copy of the emulator in memory at any one time. This emulator will decode and emulate 80x87 instructions when an 80x87 is not present in the system.

When the "fpi87" option is used exclusively, the emulator is not included. In this case, the application must be run on personal computer systems equipped with the numeric data processor.

5.7 Open Watcom C/C++ Alternate Math Libraries

One of the following Math libraries must be used if any of the modules of your application were compiled with the Open Watcom C/C++ "fpc" option and your application requires floating-point support for the reasons given above. The following Math libraries include support for floating-point which is done out-of-line through run-time calls.

16-bit libraries:

```
maths.lib (small model)
mathm.lib (medium model)
mathc.lib (compact model)
mathl.lib (large model)
mathh.lib (huge model)
```

32-bit libraries:

```
math3r.lib (flat/small models, "3r", "4r" or "5r" option)
math3s.lib (flat/small models, "3s", "4s" os "5s" option)
```

Applications which are linked with one of these libraries do not require a numeric data processor for floating-point operations. If one is present in the system, it will be used; otherwise floating-point operations are simulated in software.

5.8 The Open Watcom C/C++ Run-time Initialization Routines

Source files are included in the package for the Open Watcom C/C++ application startup (or initialization) sequence.

(16-bit only) These files are located in the directory:

```
/usr/lib/src/startup (QNX initialization)
```

The following is a summary list of the startup files for QNX.

```
cstart_s.asm  (startup for small memory model)
cstart_m.asm  (startup for medium memory model)
cstart_c.asm  (startup for compact memory model)
cstart_l.asm  (startup for large memory model)
cstart_h.asm  (startup for huge memory model)
models.inc    (included by cstart_*.asm)
cstart.asm    (included by cstart_*.asm)
mdef.inc      (macros included by cstart.asm)
cmain.c       (final part of initialization sequence)
```

The assembler file `cstart.asm` contains the first part of the initialization code and the remainder is continued in the file `cmain.c`. The assembler files, `cstart_*.asm`, define the type of memory model and include `cstart.asm`. It is `cmain.c` that calls your mainline routine `(main)`.

(32-bit only) These files are located in the directory:

```
/usr/lib/src/startup (QNX initialization)
```

The following is a summary list of the startup files for QNX.

```
cstrt386.asm  (startup for small memory model)
mdef.inc      (macros included by cstrt386.asm)
cmain.c       (final part of initialization sequence)
```

The assembler file `cstrt386.asm` contains the first part of the initialization code and the remainder is continued in the file `cmain.c`. It is `cmain.c` that calls your mainline routine `(main)`.

The source code is provided for those who wish to customize the initialization sequence for special applications.

16-bit Topics

6 16-bit Memory Models

6.1 Introduction

This chapter describes the various 16-bit memory models supported by Open Watcom C/C++. Each memory model is distinguished by two properties; the code model used to implement function calls and the data model used to reference data.

6.2 16-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to functions are made with *near calls.* In a near call, the destination address is 16 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library functions, must be less than 64K.

A big code model is one in which all calls to functions are made with *far calls.* In a far call, the destination address is 32 bits (a segment value and an offset relative to the segment value). This model allows the size of the code comprising your program to exceed 64K.

> ***Note:*** If your program contains less than 64K of code, you should use a memory model that employs the small code model. This will result in smaller and faster code since near calls are smaller instructions and are processed faster by the CPU.

6.3 16-bit Data Models

There are three data models;

1. the small data model,
2. the big data model and
3. the huge data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 16 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 64K.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 32 bits (a segment value and an offset relative to the segment value). This removes the 64K limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Open Watcom C/C++ assumes that the offset portion of a far pointer will not be incremented beyond 64K. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 64K bytes. For example, an array containing 40,000 integers does not fit into the big data model. An object such as this should be described as *huge.*

A huge data model is one in which all references to data are made with far pointers. This is similar to the big data model. However, in the huge data model, incrementing a far pointer will adjust the offset *and* the segment if necessary. The limit on the size of an object pointed to by a far pointer imposed by the big data model is removed in the huge data model.

Notes:

1. If your program contains less than 64K of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

2. The huge data model should be used only if needed. The code generated in the huge data model is not very efficient since a run-time routine is called in order to increment far pointers. This increases the size of the code significantly and increases execution time.

6.4 Summary of 16-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Open Watcom C/C++.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
tiny	small	small	near	near
small	small	small	near	near
medium	big	small	far	near
compact	small	big	near	far
large	big	big	far	far
huge	big	huge	far	huge

6.5 Mixed 16-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that uses the ***near,*** ***far,*** or ***huge*** keywords when describing some of its functions or data objects.

For example, a medium memory model application that uses some far pointers to data can be described as a mixed memory model. In an application such as this, most of the data is in a 64K segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model. Data objects outside of the DGROUP segment are described with the ***far*** keyword.

6.6 Linking Applications for the Various 16-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model.

```
Memory  Run-time      Floating-Point  Floating-Point
Model   Library       Calls Library   Library (80x87)
------- --------      --------------  ---------------

small   clibs.lib     maths.lib       math87s.lib
                                       +(no)emu87.lib*

medium  clibm.lib     mathm.lib       math87m.lib
                                       +(no)emu87.lib*

compact clibc.lib     mathc.lib       math87c.lib
                                       +(no)emu87.lib*

large   clibl.lib     mathl.lib       math87l.lib
                                       +(no)emu87.lib*

huge    clibh.lib     mathh.lib       math87h.lib
                                       +(no)emu87.lib*
```

* One of emu87.lib or noemu87.lib will be used with the 80x87 math libraries depending on the use of the "fpi" (include emulation) or "fpi87" (do not include emulation) options.

6.7 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"

2. all other segments not belonging to group "DGROUP"

3. all segments belonging to group "DGROUP" with class "BEGDATA"

4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

5. all segments belonging to group "DGROUP" with class "BSS"

6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Open Watcom C/C++ to name segments.

1. The Open Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Open Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

7 16-bit Assembly Language Considerations

7.1 Introduction

This chapter will deal with the following topics.

1. The data representation of the basic types supported by Open Watcom C/C++.

2. The memory layout of a Open Watcom C/C++ program.

3. The method for passing arguments and returning values.

4. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Open Watcom C/C++ "fpi" or "fpi87" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Open Watcom C/C++ "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

7.2 Data Representation

This section describes the internal or machine representation of the basic types supported by Open Watcom C/C++.

7.2.1 Type "char"

An item of type "char" occupies 1 byte of storage. Its value is in the following range.

```
0 <= n <= 255
```

Note that "char" is, by default, unsigned. The Open Watcom C/C++ compiler option "j" can be used to change the default from unsigned to signed. If "char" is signed, an item of type "char" is in the following range.

```
-128 <= n <= 127
```

You can force an item of type "char" to be unsigned or signed regardless of the default by defining them to be of type "unsigned char" or "signed char" respectively.

7.2.2 Type "short int"

An item of type "short int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "short int" is signed and hence "short int" and "signed short int" are equivalent. If an item of type "short int" is to be unsigned, it must be defined as "unsigned short int". In this case, its value is in the following range.

```
0 <= n <= 65535
```

7.2.3 Type "long int"

An item of type "long int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "long int" is signed and hence "long int" and "signed long int" are equivalent. If an item of type "long int" is to be unsigned, it must be defined as "unsigned long int". In this case, its value is in the following range.

```
0 <= n <= 4294967295
```

7.2.4 Type "int"

An item of type "int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "int" is signed and hence "int" and "signed int" are equivalent. If an item of type "int" is to be unsigned, it must be defined as "unsigned int". In this case its value is in the following range.

```
0 <= n <= 65535
```

If you are generating code that executes in 16-bit mode, "short int" and "int" are equivalent, "unsigned short int" and "unsigned int" are equivalent, and "signed short int" and "signed int" are equivalent. This may not be the case in other environments where "int" and "long int" are 4 bytes.

7.2.5 Type "float"

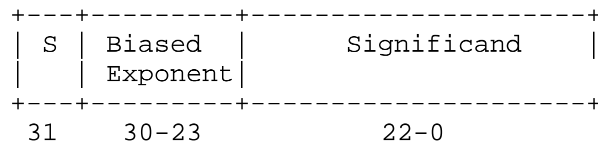
A datum of type "float" is an approximate representation of a real number. Each datum of type "float" occupies 4 bytes. If `m` is the magnitude of `x` (an item of type "float") then `x` can be approximated if

$$2^{-126} <= m < 2^{128}$$

or in more approximate terms if

```
1.175494e-38 <= m <= 3.402823e38
```

Data of type "float" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



```
+---+---------+---------------------+
| S | Biased  |     Significand     |
|   | Exponent|                     |
+---+---------+---------------------+
 31   30-23            22-0
```

Notes

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 127 (i.e., exponent value 1 represents 2^{-126}; exponent value 127 represents 2^{0}; exponent value 254 represents 2^{127}; etc.). The exponent field is 8 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 23 bits long.

Zero A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

7.2.6 Type "double"

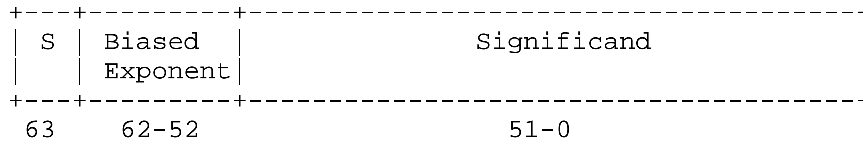
A datum of type "double" is an approximate representation of a real number. The precision of a datum of type "double" is greater than or equal to one of type "float". Each datum of type "double" occupies 8 bytes. If `m` is the magnitude of `x` (an item of type "double") then `x` can be approximated if

$$2^{-1022} <= m < 2^{1024}$$

or in more approximate terms if

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Data of type "double" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



Notes:

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 1023 (i.e., exponent value 1 represents 2^{-1022}; exponent value 1023 represents 2^{0}; exponent value 2046 represents 2^{1023}; etc.). The exponent field is 11 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 52 bits long.

Zero A double precision zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

7.3 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"

4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

5. all segments belonging to group "DGROUP" with class "BSS"

6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Open Watcom C/C++ to name segments.

1. The Open Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Open Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

7.4 Calling Conventions for Non-80x87 Applications

The following sections describe the calling convention used when compiling with the "fpc" compiler option.

7.4.1 Passing Arguments Using Register-Based Calling Conventions

How arguments are passed to a function with register-based calling conventions is determined by the size (in bytes) of the argument and where in the argument list the argument appears. Depending on the size, arguments are either passed in registers or on the stack. Arguments such as structures are almost always passed on the stack since they are generally too large to fit in registers. Since arguments are processed from left to right, the first few arguments are likely to be passed in registers (if they can fit) and, if the argument list contains many arguments, the last few arguments are likely to be passed on the stack.

The registers used to pass arguments to a function are AX, BX, CX and DX. The following algorithm describes how arguments are passed to functions.

Initially, we have the following registers available for passing arguments: AX, DX, BX and CX. Note that registers are selected from this list in the order they appear. That is, the first register selected is AX and the last is CX. For each argument *Ai,* starting with the left most argument, perform the following steps.

1. If the size of *Ai* is 1 byte, convert it to 2 bytes and proceed to the next step. If *Ai* is of type "unsigned char", it is converted to an "unsigned int". If *Ai* is of type "signed char", it is converted to a "signed int". If *Ai* is a 1-byte structure, the padding is determined by the compiler.

2. If an argument has already been assigned a position on the stack, *Ai* will also be assigned a position on the stack. Otherwise, proceed to the next step.

3. If the size of *Ai* is 2 bytes, select a register from the list of available registers. If a register is available, *Ai* is assigned that register. The register is then removed from the list of available registers. If no registers are available, *Ai* will be assigned a position on the stack.

4. If the size of *Ai* is 4 bytes, select a register pair from the following list of combinations: [DX AX] or [CX BX]. The first available register pair is assigned to *Ai* and removed from the list of available pairs. The high-order 16 bits of the argument are assigned to the first register in the pair; the low-order 16 bits are assigned to the second register in the pair. If none of the above register pairs is available, *Ai* will be assigned a position on the stack.

5. If the type of *Ai* is "double" or "float" (in the absence of a function prototype), select [AX BX CX DX] from the list of available registers. All four registers are removed from the list of available registers. The high-order 16 bits of the argument are assigned to the first register and the low-order 16 bits are assigned to the fourth register. If any of the four registers is not available, *Ai* will be assigned a position on the stack.

6. All other arguments will be assigned a position on the stack.

Notes:

1. Arguments that are assigned a position on the stack are padded to a multiple of 2 bytes. That is, if a 3-byte structure is assigned a position on the stack, 4 bytes will be pushed on the stack.

2. Arguments that are assigned a position on the stack are pushed onto the stack starting with the rightmost argument.

7.4.2 Sizes of Predefined Types

The following table lists the predefined types, their size as returned by the "sizeof" function, the size of an argument of that type and the registers used to pass that argument if it was the only argument in the argument list.

Basic Type	*"sizeof"*	*Argument Size*	*Registers Used*
char	1	2	[AX]
short int	2	2	[AX]
int	2	2	[AX]
long int	4	4	[DX AX]
float	4	8	[AX BX CX DX]
double	8	8	[AX BX CX DX]
near pointer	2	2	[AX]
far pointer	4	4	[DX AX]
huge pointer	4	4	[DX AX]

Note that the size of the argument listed in the table assumes that no function prototypes are specified. Function prototypes affect the way arguments are passed. This will be discussed in the section entitled "Effect of Function Prototypes on Arguments".

Notes:

1. Provided no function prototypes exist, an argument will be converted to a default type as described in the following table.

Argument Type	***Passed As***
char	unsigned int
signed char	signed int
unsigned char	unsigned int
float	double

7.4.3 Size of Enumerated Types

The integral type of an enumerated type is determined by the values of the enumeration constants. In strict ISO/ANSI C mode, all enumerated constants are of type `int`. In the extensions mode, the compiler will use the smallest integral type possible (excluding `long` ints) that can represent all values of the enumerated type. For instance, if the minimum and maximum values of the enumeration constants are in the range −128 and 127, the enumerated type will be equivalent to a `signed char` (size = 1 byte). All references to enumerated constants in the previous instance will have type `signed char`. An enumerated constant is always promoted to an `int` when passed as an argument.

7.4.4 Effect of Function Prototypes on Arguments

Function prototypes define the types of the formal parameters of a function. Their appearance affects the way in which arguments are passed. An argument will be converted to the type of the corresponding formal parameter in the function prototype. Consider the following example.

```
void prototype( float x, int i );

void main()
{
  float x;
  int   i;

  x = 3.14;
  i = 314;
  prototype( x, i );
  rtn( x, i );
}
```

The function prototype for `prototype` specifies that the first argument is to be passed as a "float" and the second argument is to be passed as an "int". This results in the first argument being passed in registers DX and AX and the second argument being passed in register BX.

If no function prototype is given, as is the case for the function `rtn`, the first argument will be passed as a "double" and the second argument would be passed as an "int". This results in the first argument being passed in registers AX, BX, CX and DX and the second argument being passed on the stack.

Note that even though both `prototype` and `rtn` were called with identical argument lists, the way in which the arguments were passed was completely different simply because a function prototype for `prototype` was specified. Function prototyping is an excellent way to guarantee that arguments will be passed as expected to your assembly language function.

7.4.5 Interfacing to Assembly Language Functions

Consider the following example.

Example:

```
void main()
{
    long int x;
    int      i;
    long int y;

    x = 7;
    i = 77;
    y = 777;
    myrtn( x, i, y );
}
```

`myrtn` is an assembly language function that requires three arguments. The first argument is of type "long int", the second argument is of type "int" and the third argument is again of type "long int". Using the rules for register-based calling conventions, these arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in registers DX and AX leaving BX and CX as available registers for other arguments.

2. The second argument will be passed in register BX leaving CX as an available register for other arguments.

3. The third argument will not fit in register CX (its size is 4 bytes) and hence will be pushed on the stack.

Let us look at the stack upon entry to `myrtn.`

Small Code Model

```
Offset
          +----------------+
   0      | return address | <- SP points here
          +----------------+
   2      | argument #3    |
          |                |
          +----------------+
   6      |                |
```

Big Code Model

```
Offset
          +----------------+
   0      | return address | <- SP points here
          |                |
          +----------------+
   4      | argument #3    |
          |                |
          +----------------+
   8      |                |
```

Notes:

1. The return address is the top element on the stack. In a small code model, the return address is 1 word (16 bits); in a big code model, the return address is 2 words (32 bits).

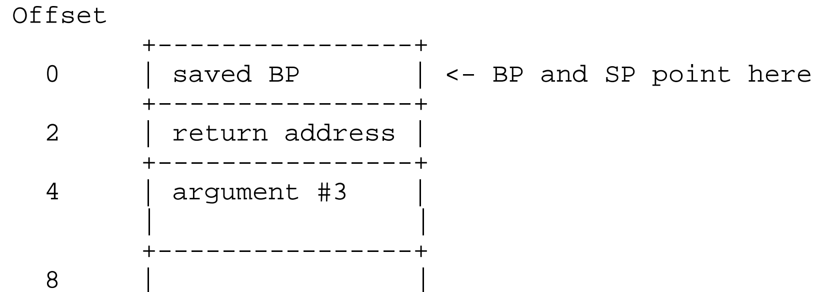
Register SP cannot be used as a base register to address the third argument on the stack. Register BP is normally used to address arguments on the stack. Upon entry to the function,

register BP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

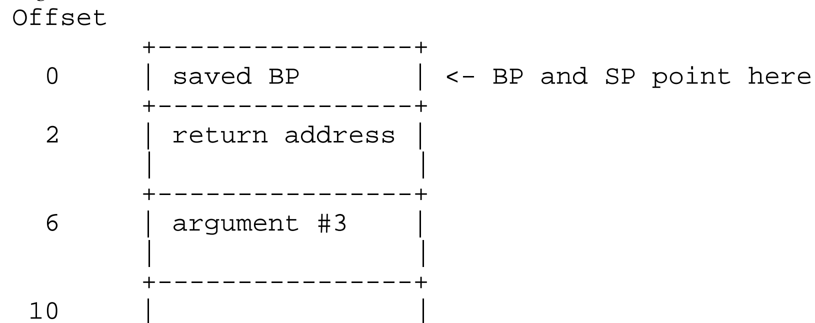
```
push    BP          ; save current value of BP
mov     BP,SP       ; get access to arguments
```

After executing these instructions, the stack looks like this.

Small Code Model



Big Code Model



As the above diagrams show, the third argument is at offset 4 from register BP in a small code model and offset 6 in a big code model.

Upon exit from `myrtn,` we must restore the value of BP. The following two instructions achieve this.

```
mov     SP,BP       ; restore stack pointer
pop     BP          ; restore BP
```

The following is a sample assembly language function which implements `myrtn.`

Small Memory Model (small code, small data)

```
DGROUP     group     _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume    CS:_TEXT
           assume    DS:DGROUP
           public    myrtn_
myrtn_     proc      near
           push      BP            ; save BP
           mov       BP,SP         ; get access to arguments
;
; body of function
;
           mov       SP,BP         ; restore SP
           pop       BP            ; restore BP
           ret       4             ; return and pop last arg
myrtn_     endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group     _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume    CS:MYRTN_TEXT
           public    myrtn_
myrtn_     proc      far
           push      BP            ; save BP
           mov       BP,SP         ; get access to arguments
;
; body of function
;
           mov       SP,BP         ; restore SP
           pop       BP            ; restore BP
           ret       4             ; return and pop last arg
myrtn_     endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must be followed with an underscore. Global variable names must be preceded with an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

8. If any of the arguments was pushed onto the stack, the called routine must pop those arguments off the stack in the "ret" instruction.

7.4.6 Functions with Variable Number of Arguments

A function prototype with a parameter list that ends with ",..." has a variable number of arguments. In this case, all arguments are passed on the stack. Since no prototyping information exists for arguments represented by ",...", those arguments are passed as described in the section "Passing Arguments".

7.4.7 Returning Values from Functions

The way in which function values are returned depends on the size of the return value. The following examples describe how function values are to be returned. They are coded for a small code model.

1. 1-byte values are to be returned in register AL.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret1_
Ret1_   proc    near   ; char Ret1()
        mov     AL,'G'
        ret
Ret1_   endp
_TEXT   ends
        end
```

2. 2-byte values are to be returned in register AX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret2_
Ret2_   proc    near   ; short int Ret2()
        mov     AX,77
        ret
Ret2_   endp
_TEXT   ends
        end
```

3. 4-byte values are to be returned in registers DX and AX with the most significant word in register DX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret4_
Ret4_   proc    near   ; long int Ret4()
        mov     AX,word ptr CS:Val4+0
        mov     DX,word ptr CS:Val4+2
        ret
Val4    dd      7777777
Ret4_   endp
_TEXT   ends
        end
```

4. 8-byte values, except structures, are to be returned in registers AX, BX, CX and DX with the most significant word in register AX.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret8_
Ret8_   proc    near   ; double Ret8()
        mov     DX,word ptr CS:Val8+0
        mov     CX,word ptr CS:Val8+2
        mov     BX,word ptr CS:Val8+4
        mov     AX,word ptr CS:Val8+6
        ret
Val8:   dq      7.7
Ret8_   endp
_TEXT   ends
        end
```

The ".8087" pseudo-op must be specified so that all floating-point constants are generated in 8087 format. When using the "fpc" (floating-point calls) option, "float" and "double" are returned in registers. See section "Returning Values in 80x87-based Applications" when using the "fpi" or "fpi87" options.

5. Otherwise, the caller allocates space on the stack for the return value and sets register SI to point to this area. In a big data model, register SI contains an offset relative to the segment value in segment register SS.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  RetX_
;
; struct int_values {
;     int value1, value2, value3, value4, value5;
;                   };
;
RetX_   proc    near ; struct int_values RetX()
        mov     word ptr SS:0[SI],71
        mov     word ptr SS:4[SI],72
        mov     word ptr SS:8[SI],73
        mov     word ptr SS:12[SI],74
        mov     word ptr SS:16[SI],75
        ret
RetX_   endp
_TEXT   ends
        end
```

When returning values on the stack, remember to use a segment override to the stack segment (SS).

The following is an example of a Open Watcom C/C++ program calling the above assembly language subprograms.

```
#include <stdio.h>

struct int_values {
    int value1;
    int value2;
    int value3;
    int value4;
    int value5;
};

extern  char              Ret1(void);
extern  short int         Ret2(void);
extern  long int          Ret4(void);
extern  double            Ret8(void);
extern  struct int_values RetX(void);

void main()
{
    struct int_values x;

    printf( "Ret1 = %c\n", Ret1() );
    printf( "Ret2 = %d\n", Ret2() );
    printf( "Ret4 = %ld\n", Ret4() );
    printf( "Ret8 = %f\n", Ret8() );
    x = RetX();
    printf( "RetX1 = %d\n", x.value1 );
    printf( "RetX2 = %d\n", x.value2 );
    printf( "RetX3 = %d\n", x.value3 );
    printf( "RetX4 = %d\n", x.value4 );
    printf( "RetX5 = %d\n", x.value5 );
}
```

The above function should be compiled for a small code model (use the "ms" or "mc" compiler option).

7.5 Calling Conventions for 80x87-based Applications

When a source file is compiled by Open Watcom C/C++ with one of the "fpi" or "fpi87" options, all floating-point arguments are passed on the 80x86 stack. The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, it is assigned a position on the 80x86 stack.

7.5.1 Passing Values in 80x87-based Applications

Consider the following example.

Example:

```
extern  void    myrtn(int,float,double,long int);

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77
    j = 77;
    myrtn( i, x, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument is of type "int" (2 bytes), the second argument is of type "float" (4 bytes), the third argument is of type "double" (8 bytes) and the fourth argument is of type "long int" (4 bytes). These arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in register AX leaving BX, CX and DX as available registers for other arguments.

2. The second argument will be passed on the 80x86 stack since it is a floating-point argument.

3. The third argument will also be passed on the 80x86 stack since it is a floating-point argument.

4. The fourth argument will be passed on the 80x86 stack since a previous argument has been assigned a position on the 80x86 stack.

Remember, arguments are pushed on the stack from right to left. That is, the rightmost argument is pushed first.

Any assembly language function must obey the following rule.

1. All arguments passed on the stack must be removed by the called function.

The following is a sample assembly language function which implements `myrtn`.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  myrtn_
myrtn_  proc    near
;
; body of function
;
        ret 16           ; return and pop arguments
myrtn_  endp
_TEXT   ends
        end
```

Notes:

1. Function names must be followed by an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option. In this example, AX does not have to be saved as it was used to pass the first argument. Floating-point registers can be modified without saving their contents.

3. The direction flag must be clear before returning to the caller.

4. This function has been written for a small code model. Any segment containing executable code must belong to the class "CODE" and the segment "_TEXT". On entry, CS contains the segment address of the segment "_TEXT". The above restrictions do not apply in a big code memory model.

5. When writing assembly language functions for a small code model, you must declare them as "near". If you wish to write assembly language functions for a big code model, you must declare them as "far".

7.5.2 Returning Values in 80x87-based Applications

Floating-point values are returned in ST(0) when using the "fpi" or "fpi87" options. All other values are returned in the manner described earlier in this chapter.

8 16-bit Pragmas

8.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas allow you to specify certain compiler options.
- Pragmas can be used to direct the Open Watcom C/C++ code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Open Watcom C/C++.
- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the C/C++ language level. The code generator can use this information to generate more efficient code.
- Any sequence of in-line machine language instructions, including QNX function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc] The item *abc* is optional.

{abc} The item *abc* may be repeated zero or more times.

a|b|c One of *a, b* or *c* may be specified.

a ::= b The item *a* is defined in terms of *b*.

(a) Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify options
- pragmas that specify default libraries
- pragmas that describe the way structures are stored in memory
- pragmas that provide auxiliary information used for code generation

8.2 Using Pragmas to Specify Options

Currently, the following options can be specified with pragmas:

unreferenced The "unreferenced" option controls the way Open Watcom C/C++ handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause Open Watcom C/C++ to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause Open Watcom C/C++ to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if "unreferenced" was specified.

check_stack The "check_stack" option controls the way stack overflows are to be handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When "check_stack" is on, Open Watcom C/C++ will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

If a stack overflow does occur during execution and you are sure that your program is not in error (i.e. it is not unnecessarily recursing), you must increase the stack size. This is done by linking your application again and specifying the "STACK" option to the Open Watcom Linker with a larger stack size.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

reuse_duplicate_strings (C only) (C Only) The "reuse_duplicate_strings" option controls the way Open Watcom C handles identical strings in an expression. For example,

```
#pragma on (reuse_duplicate_strings);
```

will cause Open Watcom C to reuse identical strings in an expression. This is the default. Specifying

```
#pragma off (reuse_duplicate_strings);
```

will cause Open Watcom C to generate additional copies of the identical string. The following example shows where this may be of importance to the way the application behaves.

Example:

```
#include <stdio.h>

#pragma off (reuse_duplicate_strings)


void poke( char *, char * );

void main()
  {
    poke( "Hello world\n", "Hello world\n" );
  }

void poke( char *x, char *y )
  {
    x[3] = 'X';
    printf( x );
    y[4] = 'Y';
    printf( y );
  }
/*
Default output:
HelXo world
HelXY world
*/
```

8.3 Using Pragmas to Specify Default Libraries

Default libraries are specified in special object module records. Library names are extracted from these special records by the Open Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

By default, that is if no library pragma is specified, the Open Watcom C/C++ compiler generates, in the object file defining the main program, default libraries corresponding to the memory model and floating-point model used to compile the file. For example, if you have compiled the source file containing the main program for the medium memory model and the floating-point calls floating-point model, the libraries "clibm" and "mathm" will be placed in the object file.

If you wish to add your own default libraries to this list, you can do so with a library pragma. Consider the following example.

```
#pragma library (mylib);
```

The name "mylib" will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as '/', ':' or ',' (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("/usr/lib/graph.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "/usr/lib/graph.lib");
```

8.4 The ALLOC_TEXT Pragma (C Only)

The "alloc_text" pragma can be used to specify the name of the text segment into which the generated code for a function, or a list of functions, is to be placed. The following describes the form of the "alloc_text" pragma.

```
#pragma alloc_text ( seg_name, fn {, fn} ) [;]
```

where	*description:*
seg_name	is the name of the text segment.
fn	is the name of a function.

Consider the following example.

```
extern int fn1(int);
extern int fn2(void);
#pragma alloc_text ( my_text, fn1, fn2 );
```

The code for the functions `fn1` and `fn2` will be placed in the segment `my_text`. Note: function prototypes for the named functions must exist prior to the "alloc_text" pragma.

8.5 The CODE_SEG Pragma

The "code_seg" pragma can be used to specify the name of the text segment into which the generated code for functions is to be placed. The following describes the form of the "code_seg" pragma.

```
#pragma code_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the text segment optionally enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_CODE_SEG"
#pragma code_seg ( seg_name );
```

class_name is the optional class name of the text segment and may be enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma code_seg ( "MY_CODE_SEG", class_name );
```

Consider the following example.

```
#pragma code_seg ( my_text );

int incr( int i )
{
    return( i + 1 );
}

int decr( int i )
{
    return( i - 1 );
}
```

The code for the functions `incr` and `decr` will be placed in the segment `my_text`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma code_seg ();
```

8.6 The COMMENT Pragma

The "comment" pragma can be used to place a comment record in an object file or executable file. The following describes the form of the "comment" pragma.

```
#pragma comment ( comment_type [, "comment_string"] ) [;]
```

where ***description:***

comment_type specifies the type of comment record. The allowable comment types are:

lib Default libraries are specified in special object module records. Library names are extracted from these special records by the Open Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

The "lib" form of this pragma offers the same features as the "library" pragma. See the section entitled "Using Pragmas to Specify Default Libraries" on page 96 for more information.

"comment_string" is an optional string literal that provides additional information for some comment types.

Consider the following example.

```
#pragma comment ( lib, "mylib" );
```

8.7 The DATA_SEG Pragma

The "data_seg" pragma can be used to specify the name of the segment into which data is to be placed. The following describes the form of the "data_seg" pragma.

```
#pragma data_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the data segment and may be enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_DATA_SEG"
#pragma data_seg ( seg_name );
```

class_name is the optional class name of the data segment and may be enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma data_seg ( "MY_DATA_SEG", class_name );
```

Consider the following example.

```
#pragma data_seg ( my_data );

static int i;
static int j;
```

The data for `i` and `j` will be placed in the segment `my_data`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma data_seg ();
```

8.8 The DISABLE_MESSAGE Pragma (C Only)

The "disable_message" pragma disables the issuance of specified diagnostic messages. The form of the "disable_message" pragma is as follows.

```
#pragma disable_message ( msg_num {, msg_num} ) [;]
```

where	*description:*
msg_num	is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C Diagnostic Messages" on page 463. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The ENABLE_MESSAGE Pragma (C Only)".

8.9 The DUMP_OBJECT_MODEL Pragma (C++ Only)

The "dump_object_model" pragma causes the C++ compiler to print information about the object model for an indicated class or an enumeration name to the diagnostics file. For class names, this information includes the offsets and sizes of fields within the class and within base classes. For enumeration names, this information consists of a list of all the enumeration constants with their values.

The general form of the "dump_object_model" pragma is as follows.

```
#pragma dump_object_model class [;]
#pragma dump_object_model enumeration [;]
class ::= a defined C++ class free of errors
enumeration ::= a defined C++ enumeration name
```

This pragma is designed to be used for information purposes only.

8.10 The ENABLE_MESSAGE Pragma (C Only)

The "enable_message" pragma re-enables the issuance of specified diagnostic messages that have been previously disabled. The form of the "enable_message" pragma is as follows.

```
#pragma enable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C Diagnostic Messages" on page 463. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The DISABLE_MESSAGE Pragma (C Only)" on page 100.

8.11 The ENUM Pragma

The "enum" pragma affects the underlying storage-definition for subsequent *enum* declarations. The forms of the "enum" pragma are as follows.

```
#pragma enum int [;]
#pragma enum minimum [;]
#pragma enum original [;]
#pragma enum pop [;]
```

where ***description:***

int Make *int* the underlying storage definition (same as the "ei" compiler option).

minimum Minimize the underlying storage definition (same as not specifying the "ei" compiler option).

original Reset back to the original compiler option setting (i.e., what was or was not specified on the command line).

pop Restore the previous setting.

The first three forms all push the previous setting before establishing the new setting.

8.12 The ERROR Pragma

The "error" pragma can be used to issue an error message with the specified text. The following describes the form of the "error" pragma.

```
#pragma error "error text" [;]
```

where ***description:***

"error text" is the text of the message that you wish to display.

You should use the ISO ***#error*** directive rather than this pragma. This pragma is provided for compatibility with legacy code. The following is an example.

```
#if defined(__386__)
    ...
#elseif defined(__86__)
    ...
#else
#pragma error ( "neither __386__ or __86__ defined" );
#endif
```

8.13 The EXTREF Pragma

The "extref" pragma is used to generate a reference to an external function or data item. The form of the "extref" pragma is as follows.

```
#pragma extref name [;]
```

where ***description:***

name is the name of an external function or data item. It must be declared to be an external function or data item before the pragma is encountered. In C++, when *name* is a function, it must not be overloaded.

This pragma causes an external reference for the function or data item to be emitted into the object file even if that function or data item is not referenced in the module. The external

reference will cause the linker to include the module containing that name in the linked program or DLL.

This is useful for debugging since you can cause debugging routines (callable from within debugger) to be included into a program or DLL to be debugged.

In C++, you can also force constructors and/or destructors to be called for a data item without necessarily referencing the data item anywhere in your code.

8.14 The FUNCTION Pragma

Certain functions, such as those listed in the description of the "oi" and "om" options, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" or "om" option or using an "intrinsic" pragma. The "function" pragma can be used to remove the intrinsic attribute for a specified list of functions.

The following describes the form of the "function" pragma.

```
#pragma function ( fn {, fn} ) [;]
```

where ***description:***

fn is the name of a function.

Suppose the following source code was compiled using the "om" option so that when one of the special math functions is referenced, the intrinsic form will be used. In our example, we have referenced the function `sin` which does have an intrinsic form. By specifying `sin` in a "function" pragma, the intrinsic attribute will be removed, causing the function `sin` to be treated as a regular user-defined function.

```
#include <math.h>
#pragma function( sin );

double test( double x )
{
    return( sin( x ) );
}
```

8.15 Setting Priority of Static Data Initialization (C++ Only)

The "initialize" pragma sets the priority for initialization of static data in the file. This priority only applies to initialization of static data that requires the execution of code. For example, the initialization of a class that contains a constructor requires the execution of the constructor. Note that if the sequence in which initialization of static data in your program takes place has no dependencies, the "initialize" pragma need not be used.

The general form of the "initialize" pragma is as follows.

```
#pragma initialize [before | after] priority [;]

priority ::= n | library | program
```

where	*description:*
n	is a number representing the priority and must be in the range 0-255. The larger the priority, the later the point at which initialization will occur.

Priorities in the range 0-20 are reserved for the C++ compiler. This is to ensure that proper initialization of the C++ run-time system takes place before the execution of your program. The "library" keyword represents a priority of 32 and can be used for class libraries that require initialization before the program is initialized. The "program" keyword represents a priority of 64 and is the default priority for any compiled code. Specifying "before" adjusts the priority by subtracting one. Specifying "after" adjusts the priority by adding one.

A source file containing the following "initialize" pragma specifies that the initialization of static data in the file will take place before initialization of all other static data in the program since a priority of 63 will be assigned.

Example:

```
#pragma initialize before program
```

If we specify "after" instead of "before", the initialization of the static data in the file will occur after initialization of all other static data in the program since a priority of 65 will be assigned.

Note that the following is equivalent to the "before" example

Example:
```
#pragma initialize 63
```

and the following is equivalent to the "after" example.

Example:
```
#pragma initialize 65
```

The use of the "before", "after", and "program" keywords are more descriptive in the intent of the pragmas.

It is recommended that a priority of 32 (the priority used when the "library" keyword is specified) be used when developing class libraries. This will ensure that initialization of static data defined by the class library will take place before initialization of static data defined by the program. The following "initialize" pragma can be used to achieve this.

Example:
```
#pragma initialize library
```

8.16 The INLINE_DEPTH Pragma (C++ Only)

When an in-line function is called, the function call may be replaced by the in-line expansion for that function. This in-line expansion may include calls to other in-line functions which can also be expanded. The "inline_depth" pragma can be used to set the number of times this expansion of in-line functions will occur for a call.

The form of the "inline_depth" pragma is as follows.

```
#pragma inline_depth [(] n [)] [;]
```

where	***description:***
n	is the depth of expansion. If `n` is 0, no expansion will occur. If `n` is 1, only the original call is expanded. If `n` is 2, the original call and the in-line functions invoked by the original function will be expanded. The default value for `n` is 3. The maximum value for `n` is 255. Note that no expansion of recursive in-line functions occur unless enabled using the "inline_recursion" pragma.

8.17 The INLINE_RECURSION Pragma (C++ Only)

The "inline_recursion" pragma controls the recursive expansion of inline functions. The form of the "inline_recursion" pragma is as follows.

```
#pragma inline_recursion [(] on | off [)] [;]
```

Specifying "on" will enable expansion of recursive inline functions. The depth of expansion is specified by the "inline_depth" pragma. The default depth is 3. Specifying "off" suppresses expansion of recursive inline functions. This is the default.

8.18 The INTRINSIC Pragma

Certain functions, those listed in the description of the "oi" option, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" option or using an "intrinsic" pragma.

The following describes the form of the "intrinsic" pragma.

```
#pragma intrinsic ( fn {, fn} ) [;]
```

where	***description:***
fn	is the name of a function.

Suppose the following source code was compiled without using the "oi" option so that no function had the intrinsic attribute. If we wanted the intrinsic form of the `sin` function to be used, we could specify the function in an "intrinsic" pragma.

```
#include <math.h>
#pragma intrinsic( sin );

double test( double x )
{
    return( sin( x ) );
}
```

8.19 The MESSAGE Pragma

The "message" pragma can be used to issue a message with the specified text to the standard output without terminating compilation. The following describes the form of the "message" pragma.

```
#pragma message ( "message text" ) [;]
```

where ***description:***

"message text" is the text of the message that you wish to display.

The following is an example.

```
#if defined(__386__)
    ...
#else
#pragma message ( "assuming 16-bit compile" );
#endif
```

8.20 The ONCE Pragma

The "once" pragma can be used to indicate that the file which contains this pragma should only be opened and processed "once". The following describes the form of the "once" pragma.

```
#pragma once [;]
```

Assume that the file "foo.h" contains the following text.

Example:

```
#ifndef _FOO_H_INCLUDED
#define _FOO_H_INCLUDED
#pragma once
    .
    .
    .
#endif
```

The first time that the compiler processes "foo.h" and encounters the "once" pragma, it records the file's name. Subsequently, whenever the compiler encounters a `#include` statement that refers to "foo.h", it will not open the include file again. This can help speed up processing of `#include` files and reduce the time required to compile an application.

8.21 The PACK Pragma

The "pack" pragma can be used to control the way in which structures are stored in memory. There are 4 forms of the "pack" pragma.

The following form of the "pack" pragma can be used to change the alignment of structures and their fields in memory.

```
#pragma pack ( n ) [;]
```

where ***description:***

n is 1, 2, 4, 8 or 16 and specifies the method of alignment.

The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4, 8 or 16, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

```
                    zp1     zp2     zp4     zp8     zp16
sizeof(member)  \---------------------------------------
        1       |   0       0       0       0       0
        2       |   0       2       2       2       2
        4       |   0       2       4       4       4
        8       |   0       2       4       8       8
        16      |   0       2       4       8       16
        x       |   aligned to largest member
```

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8. If the largest member of structure "x" is 16 bytes then "x" is aligned according to row 16.

If no value is specified in the "pack" pragma, a default value of 2 is used. Note that the default value can be changed with the "zp" Open Watcom C/C++ compiler command line option.

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack.

```
#pragma pack ( push ) [;]
```

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack and set the current alignment.

```
#pragma pack ( push, number ) [;]
```

The following form of the "pack" pragma can be used to restore the previous alignment amount from an internal stack.

```
#pragma pack ( pop ) [;]
```

8.22 The READ_ONLY_FILE Pragma

Explicit listing of dependencies in a makefile can often be tedious in the development and maintenance phases of a project. The Open Watcom C/C++ compiler will insert dependency information into the object file as it processes source files so that a complete snapshot of the files necessary to build the object file are recorded. The "read_only_file" pragma can be used to prevent the name of the source file that includes it from being included in the dependency information that is written to the object file.

This pragma is commonly used in system header files since they change infrequently (and, when they do, there should be no impact on source files that have included them).

The form of the "read_only_file" pragma follows.

```
#pragma read_only_file [;]
```

For more information on make dependencies, see the section entitled "Automatic Dependency Detection (.AUTODEPEND)" in the ***Open Watcom C/C++ Tools User's Guide.***

8.23 The TEMPLATE_DEPTH Pragma (C++ Only)

The "template_depth" pragma provides a hard limit for the amount of nested template expansions allowed so that infinite expansion can be detected.

The form of the "template_depth" pragma is as follows.

```
#pragma template_depth [(] n [)] [;]
```

where ***description:***

n is the depth of expansion. If the value of `n` is less than 2, if will default to 2. If `n` is not specified, a warning message will be issued and the default value for `n` will be 100.

The following example of recursive template expansion illustrates why this pragma can be useful.

Example:

```
#pragma template_depth(10);

template <class T>
struct S {
    S<T*> x;
};

S<char> v;
```

8.24 The WARNING Pragma (C++ Only)

The "warning" pragma sets the level of warning messages. The form of the "warning" pragma is as follows.

```
#pragma warning msg_num level [;]
```

where ***description:***

msg_num is the number of the warning message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C++ Diagnostic Messages" on page 501. If `msg_num` is "*", the level of all warning messages is changed to the specified level. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

level is a number from 0 to 9 and represents the level of the warning message. When a value of zero is specified, the warning becomes an error.

8.25 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

8.25.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. a type definition that resolves to a function type
3. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

An example of a type definition that resolves to a function type is the following.

```
typedef void (*func_type)();
```

When an auxiliary pragma refers to a such a type definition, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to each function whose type matches the specified type definition.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
#pragma aux y <attrs_1>;
code in which symbol y is referenced
code in which symbol z is referenced
#pragma aux default <attrs_2>;
#pragma aux x <attrs_3>;
```

Auxiliary attributes are assigned to x, y and z in the following way.

1. Symbol x is assigned the initial default attributes merged with the attributes specified by <attrs_2> and <attrs_3>.

2. Symbol y is assigned the initial default attributes merged with the attributes specified by <attrs_1>.

3. Symbol z is assigned the initial default attributes merged with the attributes specified by <attrs_2>.

8.25.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as

additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
#pragma aux ( sym, alias ) [;]
```

where	*description:*
sym	is any valid C/C++ identifier.
alias	is the alias name and is any valid C/C++ identifier.

Consider the following example.

```
#pragma aux push_args parm [] ;
#pragma aux ( rtn, push_args ) ;
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed on the stack.

Let us look at an example in which the symbol is a type definition.

```
typedef void (func_type)(int);

#pragma aux push_args parm [];
#pragma aux ( func_type, push_args );

extern func_type rtn1;
extern func_type rtn2;
```

The first auxiliary pragma defines an alias name called `push_args` that specifies the mechanism to be used to pass arguments. The mechanism is to pass all arguments on the stack. The second auxiliary pragma associates the attributes specified in the first pragma with the type definition `func_type`. Since `rtn1` and `rtn2` are of type `func_type`, arguments to either of those functions will be passed on the stack.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
#pragma aux ( alias ) sym aux_attrs [;]
```

where	*description:*
alias	is the alias name and is any valid C/C++ identifier.
sym	is any valid C/C++ identifier.
aux_attrs	are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
#pragma aux MS_C "_*"                                    \
                 parm caller []                          \
                 value struct float struct routine [ax]\
                 modify [ax bx cx dx es];
#pragma aux (MS_C) rtn1;
#pragma aux (MS_C) rtn2;
#pragma aux (MS_C) rtn3;
```

The routines `rtn1`, `rtn2` and `rtn3` assume the same attributes as the alias name `MS_C` which defines the calling convention used by the Microsoft C compiler. Whenever calls are made to `rtn1`, `rtn2` and `rtn3`, the Microsoft C calling convention will be used.

Note that if the attributes of `MS_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1`, `rtn2` and `rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

> ***WARNING!*** The alias name `MS_C` is just another symbol. If `MS_C` appeared in your source code, it would assume the attributes specified in the pragma for `MS_C`.

8.25.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` or `cdecl` defines the calling convention used by Microsoft compilers.

__fastcall `__fastcall` or `fastcall` defines the calling convention used by Microsoft compilers.

__fortran `__fortran` or `fortran` defines the calling convention used by Open Watcom FORTRAN compilers.

__pascal `__pascal` or `pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

__stdcall `__stdcall` or `stdcall` defines the calling convention used by Microsoft compilers.

__watcall `__watcall` or `watcall` defines the default calling convention used by Open Watcom compilers.

The following describes the attributes of the above alias names.

8.25.3.1 Predefined "__cdecl" Alias

```
#pragma aux __cdecl "_*" \
           parm caller [] \
           value struct float struct routine [ax] \
           modify [ax bx cx dx es]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register AX.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

8.25.3.2 Predefined "__pascal" Alias

```
#pragma aux __pascal "^" \
           parm reverse routine [] \
           value struct float struct caller [] \
           modify [ax bx cx dx es]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register AX will contain address of the space allocated for the return value.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

8.25.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
#pragma aux sym obj_name [;]
```

where	***description:***
sym	is any valid C/C++ identifier.
obj_name	is any character string enclosed in double quotes.

When specifying `obj_name,` the asterisk character ('*') has a special meaning; it is a placeholder for `sym.`

In the following example, the name "myrtn" will be replaced by "myrtn_" in the object file.

```
#pragma aux myrtn "*_";
```

This is the default for all function names.

In the following example, the name "myvar" will be replaced by "_myvar" in the object file.

```
#pragma aux myvar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

The '^' character also has a special meaning. Whenever it is encountered in `obj_name`, it is replaced by the upper case version of `sym`.

In the following example, the name "myrtn" will be replaced by "MYRTN" in the object file.

```
#pragma aux myrtn "^";
```

8.25.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#pragma aux sym far [;]
    or
#pragma aux sym near [;]
    or
#pragma aux sym = in_line [;]

in_line ::= { const | (seg id) | (offset id) | (reloff id)
          | (float fpinst) | "asm" }
```

where	***description:***
sym	is a function name.
const	is a valid C/C++ integer constant.
id	is any valid C/C++ identifier.
fpinst	is a sequence of bytes that forms a valid 80x87 instruction. The keyword ***float*** must precede `fpinst` so that special fixups are applied to the 80x87 instruction.
seg	specifies the segment of the symbol `id.`
offset	specifies the offset of the symbol `id.`
reloff	specifies the relative offset of the symbol `id` for near control transfers.
asm	is an assembly language instruction or directive.

In the following example, Open Watcom C/C++ will generate a far call to the function `myrtn.`

```
#pragma aux myrtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Open Watcom C/C++ will generate a near call to the function `myrtn.`

```
#pragma aux myrtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Open Watcom C/C++ will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line function.

```
void mode4(void);
#pragma aux mode4 =                   \
    0xb4 0x00       /* mov AH,0 */ \
    0xb0 0x04       /* mov AL,4 */ \
    0xcd 0x10       /* int 10H  */ \
    modify [ AH AL ];
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0       ; select function "set mode"
mov   AL,4       ; specify mode (mode 4)
int   10H        ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your C/C++ program. The C prototype for the function `mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by Open Watcom C/C++.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
void mode4(void);
#pragma aux mode4 =      \
    "mov AH,0",          \
    "mov AL,4",          \
    "int 10H"            \
    modify [ AH AL ];
```

If a sequence of in-line assembly language instructions contains 80x87 floating-point instructions, each floating-point instruction must be preceded by "float". Note that this is only required if you have specified the "fpi" compiler option; otherwise it will be ignored.

The following example generates the 80x87 "square root" instruction.

```
double mysqrt(double);
#pragma aux mysqrt parm [8087] = \
    float 0xd9 0xfa /* fsqrt */;
```

A sequence of in-line assembly language instructions may contain symbolic references. In the following example, a near call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                     \
    0xe8 reloff myalias /* near call */;
```

In the following example, a far call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                                \
    0x9a offset myalias seg myalias /* far call */;
```

8.25.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a function that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified function.

```
#pragma aux sym parm loadds [;]
```

where ***description:***

sym is a function name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified function.

```
#pragma aux sym loadds [;]
```

where	***description:***
sym	is a function name.

8.25.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link libraries are exported using the Open Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
#pragma aux sym export [;]
```

where	***description:***
sym	is a function name.

8.25.5.3 Defining Windows Callback Functions

When compiling a Microsoft Windows application, you must use the "zW" option so that special prologue/epilogue sequences are generated. Furthermore, callback functions require larger prologue/epilogue sequences than those generated when the "zW" compiler option is specified. The following form of the auxiliary pragma will cause a callback prologue/epilogue sequence to be generated for a callback function when compiled using the "zW" option.

```
#pragma aux sym export [;]
```

where	***description:***
sym	is a callback function name.

Alternatively, the "zw" compiler option can be used to generate callback prologue/epilogue sequences. However, all functions contained in a module compiled using the "zw" option will have a callback prologue/epilogue sequence even if the functions are not callback functions.

8.25.5.4 Forcing a Stack Frame

Normally, a function contains a stack frame if arguments are passed on the stack or an automatic variable is allocated on the stack. No stack frame will be generated if the above conditions are not satisfied. The following form of the auxiliary pragma will force a stack frame to be generated under any circumstance.

```
#pragma aux sym frame [;]
```

where	*description:*
sym	is a function name.

8.25.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Open Watcom C/C++ is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#pragma aux sym parm { pop_info | reverse | {reg_set} } [;]

pop_info ::= caller | routine
```

where	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

8.25.6.1 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **float** and **double** are always pushed on the stack when the "fpi" or "fpi87" option is used.

double Arguments of type **double** can only be passed in the following register combination: AX:BX:CX:DX. For example, if the following register set was specified for a routine having an argument of type **double,**

```
[AX BX SI DI]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **double** is supported only when the "fpc" option is used. Note that this argument passing method does not include the passing of 8-byte structures.

far pointer A far pointer can only be passed in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. For example, if a far pointer is passed to a function with the following register set,

```
[ES BP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set.

long int, float

The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **long int,**) are: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX and BX:AX. For example, if the following register set was specified for a routine with one argument of type **long int,**

```
[ES DI]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method includes 4-byte structures. Note that this argument passing method includes arguments of type **float** but only when the "fpc" option is used.

int The only registers that will be assigned to 2-byte arguments (e.g., arguments of type **int**) are: AX, BX, CX, DX, SI and DI. For example, if the following register set was specified for a routine with one argument of type **int,**

```
[BP]
```

the argument would be pushed on the stack since a valid register combination for 2-byte arguments is not contained in the register set.

char Arguments whose size is 1 byte (e.g., arguments of type **char**) are promoted to 2 bytes and are then assigned registers as if they were 2-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [AX BX CX DX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX.

3. If you are compiling for a memory model with a small data model, or the "zdp" compiler option is specified, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment. Note that the "zdf" compiler option can be used to specify that register DS does not contain that segment address of the program's data segment. In this case, register combinations containing register DS are legal.

Consider the following example.

```
#pragma aux myrtn parm [ax bx cx dx] [bp si];
```

Suppose `myrtn` is a routine with 3 arguments each of type **long int.**

1. The first argument will be passed in the register pair DX:AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be pushed on the stack since BP:SI is not a valid register pair for arguments of type **long int.**

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
#pragma aux myrtn parm [ax bx cx dx] [si di];
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **int** and the second and third of type **long int.**

1. The first argument will be passed in the register AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be passed in the register set DI:SI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [AX BX CX DX] is used.

8.25.6.2 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a function, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register SI, the second argument to be passed in register DI and the third argument to be passed in register CX, the following auxiliary pragma can be used.

```
void mycopy( char near *, char *, int );
#pragma aux mycopy parm [SI] [DI] [CX];
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

8.25.6.3 Passing Arguments to In-Line Functions

For functions whose code is generated by Open Watcom C/C++ and whose argument list is described by an auxiliary pragma, Open Watcom C/C++ has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Open Watcom C/C++ assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line function called `scrollactivepgup`.

```
void scrollactivepgup(char,char,char,char,char,char);
#pragma aux scrollactivepgup = \
    "mov AH,6"   \
    "int 10h"    \
    parm [ch] [cl] [dh] [dl] [al] [bh] \
    modify [ah];
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Open Watcom C/C++ will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example,

if the first argument to `scrollactivepgup` was called with an argument whose type was **int**, it would first be converted to **char** before assigning it to register CH. Similarly, if an in-line function required its argument in register pair DX:AX and the argument was of type **short int**, the argument would be converted to **long int** before assigning it to register pair DX:AX.

In general, Open Watcom C/C++ assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **unsigned char.**

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **unsigned short int.**

3. A register set consisting of two 16-bit registers (4 bytes) is assigned a type of **unsigned long int.**

4. A register set consisting of four 16-bit registers (8 bytes) is assigned a type of **double.**

8.25.6.4 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
#pragma aux sym parm (caller | routine) [;]
```

where ***description:***

sym is a function name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

8.25.6.5 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
#pragma aux sym parm reverse [;]
```

where ***description:***

sym is a function name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for functions that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a function.

```
#pragma aux rtn parm reverse [];
```

8.25.7 Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#pragma aux sym value {no8087 | reg_set | struct_info} [;]
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

8.25.7.1 Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Note that the method described below for returning values of type **float** or **double** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set.*

1-byte For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2-byte For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

4-byte For 4-byte return values (except far pointers), only the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, registers DX:AX will be used. This form of the auxiliary pragma is legal for functions of type **float** when using the "fpc" option only.

far pointer For functions that return far pointers, the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. If no register set is specified, the registers DX:AX will be used.

8-byte For 8-byte return values (including functions of type **double**), only the following register combination is allowed: AX:BX:CX:DX. If no register set is specified, the registers AX:BX:CX:DX will be used. This form of the auxiliary pragma is legal for functions of type **double** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

8.25.7.2 Returning Structures

Typically, structures are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register SI to point to it. The called routine then places the return value at the location pointed to by register SI.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
#pragma aux sym value struct (caller|routine) reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register AX by the called routine. It is assumed that the memory for the return value is allocated from the stack segment (the stack segment is contained in segment register SS).

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: AX, DX, BX, CX, SI or DI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, register pair DX:AX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register SI to it.

```
#pragma aux sym value struct struct [;]
```

where ***description:***

sym is a function name.

8.25.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **float** or **double** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register SI to it.

```
#pragma aux sym value struct float [;]
```

where ***description:***

sym is a function name.

In other words, floating-point values are to be returned in the same way structures are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **float** will be returned in registers DX:AX. Function return values whose type is **double** will be returned in registers AX:BX:CX:DX. This is the default method for the "fpc" option.

```
#pragma aux sym value no8087 [;]
```

where ***description:***

sym is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
#pragma aux sym value [8087] [;]
```

where ***description:***

sym is a function name.

8.25.8 A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#pragma aux sym aborts [;]
```

where	description:
sym	is a function name.

Consider the following example.

```
#pragma aux exitrtn aborts;
extern void exitrtn(void);

void rtn()
  {
    exitrtn();
  }
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Open Watcom C/C++ generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn.`

8.25.9 Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym modify nomemory [;]
```

where	description:
sym	is a function name.

Consider the following example.

```
#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
    while( i < 10000 ) {
        i += 383;
    }
    myrtn();
    i += 13143;
};
```

To compile the above program, "rtn.c", we issue the following command.

```
$ wcc rtn -oai -d1
$ wpp rtn -oai -d1
$ wcc386 rtn -oai -d1
$ wpp386 rtn -oai -d1
```

For illustrative purposes, we omit loop optimizations from the list of code optimizations that we want the compiler to perform. The "d1" compiler option is specified so that the object file produced by Open Watcom C/C++ contains source line information.

We can generate a file containing a disassembly of `rtn.o` by issuing the following command.

```
$ wdis rtn -l -s -r
```

The "s" option is specified so that the listing file produced by the Open Watcom Disassembler contains source lines taken from `rtn.c.` The listing file `rtn.lst` appears as follows.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  0026 bytes

#pragma off (check_stack);

extern void MyRtn( void );

int i = { 1033 };
```

```
extern Rtn()
  {
 0000  52                 Rtn_            push    DX
 0001  8b 16 00 00                        mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27        L1              cmp     DX,2710H
 0009  7d 06                              jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                        add     DX,017fH
 000f  eb f4                              jmp     L1

    MyRtn();
 0011  89 16 00 00        L2              mov     _i,DX
 0015  e8 00 00                           call    MyRtn_
 0018  8b 16 00 00                        mov     DX,_i

    i += 13143;
 001c  81 c2 57 33                        add     DX,3357H
 0020  89 16 00 00                        mov     _i,DX

  };
 0024  5a                                 pop     DX
 0025  c3                                 ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                    _i              - ..

No disassembly errors

------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
#pragma aux myrtn modify nomemory;
```

If we compile the source file with the above pragma and disassemble the object file using the Open Watcom Disassembler, we get the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  0022 bytes

#pragma off (check_stack);

extern void MyRtn( void );
#pragma aux MyRtn modify nomemory;

int i = { 1033 };

extern Rtn()
  {
 0000  52                Rtn_            push    DX
 0001  8b 16 00 00                       mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27       L1              cmp     DX,2710H
 0009  7d 06                             jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                       add     DX,017fH
 000f  eb f4                             jmp     L1

    MyRtn();
 0011  89 16 00 00       L2              mov     _i,DX
 0015  e8 00 00                          call    MyRtn_

    i += 13143;
 0018  81 c2 57 33                       add     DX,3357H
 001c  89 16 00 00                       mov     _i,DX

  };
 0020  5a                                pop     DX
 0021  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                   _i              - ..

No disassembly errors

------------------------------------------------------------
```

Notice that the value of `i` is in register DX after completion of the "while" loop. After the call to `myrtn`, the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any memory (i.e., global or static variables) that is used directly or indirectly by `Rtn` and hence register DX contains the correct value of `i`.

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym parm nomemory modify nomemory [;]
```

where ***description:***

sym is a function name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
#pragma aux myrtn parm nomemory modify nomemory;
```

If you now compile our source file and disassemble the object file using wdis, the result is the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  001e bytes

#pragma off (check_stack);

extern void MyRtn( void );
#pragma aux MyRtn parm nomemory modify nomemory;

int i = { 1033 };

extern Rtn()
  {
 0000  52                Rtn_           push    DX
 0001  8b 16 00 00                      mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27       L1             cmp     DX,2710H
 0009  7d 06                            jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                      add     DX,017fH
 000f  eb f4                            jmp     L1
```

```
   MyRtn();
 0011  e8 00 00          L2              call    MyRtn_

   i += 13143;
 0014  81 c2 57 33                       add     DX,3357H
 0018  89 16 00 00                       mov     _i,DX

 };
 001c  5a                                pop     DX
 001d  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                   _i              - ..

No disassembly errors

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register DX before calling `myrtn.` The auxiliary pragma informs the compiler that `myrtn` does not reference any memory (i.e., global or static variables) that is used directly or indirectly by `myrtn` so updating `i` was not necessary before calling `myrtn.`

8.25.10 Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a function will use without saving.

```
#pragma aux sym modify [exact] reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Specifying a register set informs Open Watcom C/C++ that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called function. Also, since the AX register is frequently used to

return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called function does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called function.

```
#pragma aux sym modify exact reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

The above form of the auxiliary pragma tells Open Watcom C/C++ not to assume that the registers used to pass arguments will be modified by the called function. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:

```
unsigned GetSP(void);
#if defined(__386__)
#pragma aux GetSP = value [esp] modify exact [];
#else
#pragma aux GetSP = value [sp] modify exact [];
#endif
```

8.25.11 An Example

As mentioned in an earlier section, the following pragma defines the calling convention for functions compiled by Microsoft C.

```
#pragma aux MS_C "_*"                              \
                 parm caller []                    \
                 value struct float struct routine [ax]\
                 modify [ax bx cx dx es];
```

Let us discuss this pragma in detail.

"_*" specifies that all function and variable names are preceded by the underscore character (_) when translated from source form to object form.

parm caller [] specifies that all arguments are to be passed on the stack (an empty register set was specified) and the caller will remove the arguments from the stack.

value struct marks the section describing how the called routine returns structure information.

- ***float*** specifies that floating-point arguments are returned in the same way as structures are returned.
- ***struct*** specifies that 1, 2 and 4-byte structures are not to be returned in registers.
- ***routine*** specifies that the called routine allocates storage for the return structure and returns with a register pointing at it.
- ***[ax]*** specifies that register AX is used to point to the structure return value.

modify [ax bx cx dx es]

specifies that registers AX, BX, CX, DX and ES are not preserved by the called routine.

Note that the default method of returning integer values is used; 1-byte characters are returned in register AL, 2-byte integers are returned in register AX, and 4-byte integers are returned in the register pair DX:AX.

8.25.12 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile functions. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

8.25.12.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point

registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).

```
            +----------------+
      0     | 4th from top   |  ST(4)
            +----------------+
      1     | 5th from top   |  ST(5)
            +----------------+
      2     | 6th from top   |  ST(6)
            +----------------+
      3     | 7th from top   |  ST(7)
            +----------------+
ST -> 4     | top of stack   |  ST(0)
            +----------------+
      5     | 1st from top   |  ST(1)
            +----------------+
      6     | 2nd from top   |  ST(2)
            +----------------+
      7     | 3rd from top   |  ST(3)
            +----------------+
```

Starting with version 9.5, the Open Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
#pragma aux myrtn parm [8087];

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77;
    j = 77;
    myrtn( x, i, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument of type **float** (4 bytes), the second argument is of type **int** (2 bytes), the third argument is of type **double** (8 bytes) and the fourth argument is of type **long int** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float,** will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
#pragma aux myrtn parm [ax 8087];
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register AX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **double**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

8.25.12.2 Using the 80x87 to Return Function Values

The following form of the auxiliary pragma can be used to describe a function that returns a floating-point value in ST(0).

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

8.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a function unless the "fpr" option is used to generate backward compatible code (older Open Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

```
#pragma aux sym modify reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

This instructs Open Watcom C/C++ to save any local variables that are located in the 80x87 cache before calling the specified routine.

32-bit Topics

9 32-bit Memory Models

9.1 Introduction

This chapter describes the various 32-bit memory models supported by Open Watcom C/C++. Each memory model is distinguished by two properties; the code model used to implement function calls and the data model used to reference data.

9.2 32-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to functions are made with *near calls.* In a near call, the destination address is 32 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library functions, must be less than 4GB.

A big code model is one in which all calls to functions are made with *far calls.* In a far call, the destination address is 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This model allows the size of the code comprising your program to exceed 4GB.

> ***Note:*** If your program contains less than 4GB of code, you should use a memory model that employs the small code model. This will result in smaller and faster code since near calls are smaller instructions and are processed faster by the CPU.

9.3 32-bit Data Models

There are two data models;

1. the small data model and
2. the big data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 32 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 4GB.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This removes the 4GB limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Open Watcom C/C++ assumes that the offset portion of a far pointer will not be incremented beyond 4GB. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 4GB.

> ***Note:*** If your program contains less than 4GB of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

9.4 Summary of 32-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Open Watcom C/C++.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
flat	small	small	near	near
small	small	small	near	near
medium	big	small	far	near
compact	small	big	near	far
large	big	big	far	far

9.5 Flat Memory Model

In the flat memory model, the application's code and data must total less than 4GB in size. Segment registers CS, DS, SS and ES point to the same linear address space (this does not imply that the segment registers contain the same value). That is, a given offset in one segment refers to the same memory location as that offset in another segment. Essentially, a flat model operates as if there were no segments.

9.6 Mixed 32-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that uses the ***near, far,*** or ***huge*** keywords when describing some of its functions or data objects.

For example, a medium memory model application that uses some far pointers to data can be described as a mixed memory model. In an application such as this, most of the data is in a 4GB segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model. Data objects outside of the DGROUP segment are described with the ***far*** keyword.

9.7 Linking Applications for the Various 32-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model. Currently, only libraries for the flat/small memory model are provided.

```
Memory      Run-time    Floating-Point   Floating-Point
Model       Library     Library (80x87)  Library (f-p calls)
----------  ----------  ---------------  -------------------
flat/small  clib3r.lib    math387r.lib     math3r.lib
            clib3s.lib    math387s.lib     math3s.lib
            plib3r.lib    cplx73r.lib      cplx3r.lib
            plib3s.lib    cplx73s.lib      cplx3s.lib
```

The letter "r" or "s" which is affixed to the file name indicates the particular strategy with which the modules in the library have been compiled.

r denotes a version of the Open Watcom C/C++ 32-bit libraries which have been compiled for the "flat/small" memory models using the "3r", "4r" or "5r" option.

s denotes a version of the Open Watcom C/C++ 32-bit libraries which have been compiled for the "flat/small" memory models using the "3s", "4s" or "5s" option.

9.8 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Open Watcom C/C++ to name segments.

1. The Open Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.

2. The Open Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

10 32-bit Assembly Language Considerations

10.1 Introduction

This chapter will deal with the following topics.

1. The data representation of the basic types supported by Open Watcom C/C++.

2. The memory layout of a Open Watcom C/C++ program.

3. The method for passing arguments and returning values.

4. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Open Watcom C/C++ "fpi" or "fpi87" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Open Watcom C/C++ "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

10.2 Data Representation

This section describes the internal or machine representation of the basic types supported by Open Watcom C/C++.

10.2.1 Type "char"

An item of type "char" occupies 1 byte of storage. Its value is in the following range.

```
0 <= n <= 255
```

Note that "char" is, by default, unsigned. The Open Watcom C/C++ compiler option "j" can be used to change the default from unsigned to signed. If "char" is signed, an item of type "char" is in the following range.

```
-128 <= n <= 127
```

You can force an item of type "char" to be unsigned or signed regardless of the default by defining them to be of type "unsigned char" or "signed char" respectively.

10.2.2 Type "short int"

An item of type "short int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "short int" is signed and hence "short int" and "signed short int" are equivalent. If an item of type "short int" is to be unsigned, it must be defined as "unsigned short int". In this case, its value is in the following range.

```
0 <= n <= 65535
```

10.2.3 Type "long int"

An item of type "long int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "long int" is signed and hence "long int" and "signed long int" are equivalent. If an item of type "long int" is to be unsigned, it must be defined as "unsigned long int". In this case, its value is in the following range.

```
0 <= n <= 4294967295
```

10.2.4 Type "int"

An item of type "int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "int" is signed and hence "int" and "signed int" are equivalent. If an item of type "int" is to be unsigned, it must be defined as "unsigned int". In this case its value is in the following range.

```
0 <= n <= 4294967295
```

If you are generating code that executes in 32-bit mode, "long int" and "int" are equivalent, "unsigned long int" and "unsigned int" are equivalent, and "signed long int" and "signed int" are equivalent. This may not be the case in other environments where "int" and "short int" are 2 bytes.

10.2.5 Type "float"

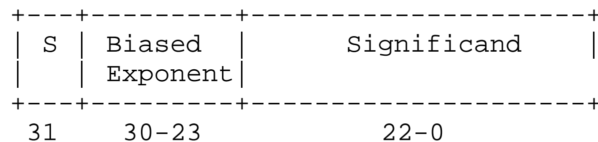
A datum of type "float" is an approximate representation of a real number. Each datum of type "float" occupies 4 bytes. If `m` is the magnitude of `x` (an item of type "float") then `x` can be approximated if

$$2^{-126} <= m < 2^{128}$$

or in more approximate terms if

```
1.175494e-38 <= m <= 3.402823e38
```

Data of type "float" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



```
+---+---------+---------------------+
| S | Biased  |      Significand    |
|   | Exponent|                     |
+---+---------+---------------------+
 31    30-23           22-0
```

Notes

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 127 (i.e., exponent value 1 represents 2^{-126}; exponent value 127 represents 2^0; exponent value 254 represents 2^{127}; etc.). The exponent field is 8 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 23 bits long.

Zero A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

10.2.6 Type "double"

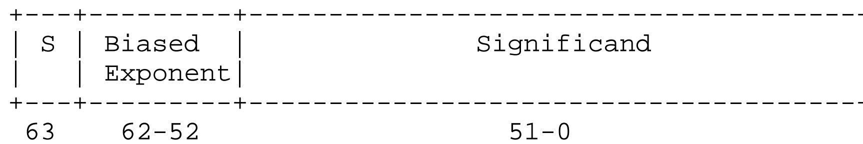
A datum of type "double" is an approximate representation of a real number. The precision of a datum of type "double" is greater than or equal to one of type "float". Each datum of type "double" occupies 8 bytes. If `m` is the magnitude of `x` (an item of type "double") then `x` can be approximated if

$$2^{-1022} <= m < 2^{1024}$$

or in more approximate terms if

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Data of type "double" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.


```
+---+---------+-----------------------------------------+
| S | Biased  |              Significand                |
|   | Exponent|                                         |
+---+---------+-----------------------------------------+
 63   62-52                   51-0
```

Notes:

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 1023 (i.e., exponent value 1 represents 2^{-1022}; exponent value 1023 represents 2^{0}; exponent value 2046 represents 2^{1023}; etc.). The exponent field is 11 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 52 bits long.

Zero A double precision zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

10.3 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Open Watcom C/C++ to name segments.

1. The Open Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Open Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

10.4 Calling Conventions for Non-80x87 Applications

The following sections describe the calling convention used when compiling with the "fpc" compiler option.

10.4.1 Passing Arguments Using Register-Based Calling Conventions

How arguments are passed to a function with register-based calling conventions is determined by the size (in bytes) of the argument and where in the argument list the argument appears. Depending on the size, arguments are either passed in registers or on the stack. Arguments such as structures are almost always passed on the stack since they are generally too large to fit in registers. Since arguments are processed from left to right, the first few arguments are likely to be passed in registers (if they can fit) and, if the argument list contains many arguments, the last few arguments are likely to be passed on the stack.

The registers used to pass arguments to a function are EAX, EBX, ECX and EDX. The following algorithm describes how arguments are passed to functions.

Initially, we have the following registers available for passing arguments: EAX, EDX, EBX and ECX. Note that registers are selected from this list in the order they appear. That is, the first register selected is EAX and the last is ECX. For each argument *Ai,* starting with the left most argument, perform the following steps.

1. If the size of *Ai* is 1 byte or 2 bytes, convert it to 4 bytes and proceed to the next step. If *Ai* is of type "unsigned char" or "unsigned short int", it is converted to an "unsigned int". If *Ai* is of type "signed char" or "signed short int", it is converted to a "signed int". If *Ai* is a 1-byte or 2-byte structure, the padding is determined by the compiler.

2. If an argument has already been assigned a position on the stack, *Ai* will also be assigned a position on the stack. Otherwise, proceed to the next step.

3. If the size of *Ai* is 4 bytes, select a register from the list of available registers. If a register is available, *Ai* is assigned that register. The register is then removed from

the list of available registers. If no registers are available, *Ai* will be assigned a position on the stack.

4. If the type of *Ai* is "far pointer", select a register pair from the following list of combinations: [EDX EAX] or [ECX EBX]. The first available register pair is assigned to *Ai* and removed from the list of available pairs. The segment value will actually be passed in register DX or CX and the offset in register EAX or EBX. If none of the above register pairs is available, *Ai* will be assigned a position on the stack. Note that 8 bytes will be pushed on the stack even though the size of an item of type "far pointer" is 6 bytes.

5. If the type of *Ai* is "double" or "float" (in the absence of a function prototype), select a register pair from the following list of combinations: [EDX EAX] or [ECX EBX]. The first available register pair is assigned to *Ai* and removed from the list of available pairs. The high-order 32 bits of the argument are assigned to the first register in the pair; the low-order 32 bits are assigned to the second register in the pair. If none of the above register pairs is available, *Ai* will be assigned a position on the stack.

6. All other arguments will be assigned a position on the stack.

Notes:

1. Arguments that are assigned a position on the stack are padded to a multiple of 4 bytes. That is, if a 3-byte structure is assigned a position on the stack, 4 bytes will be pushed on the stack.

2. Arguments that are assigned a position on the stack are pushed onto the stack starting with the rightmost argument.

10.4.2 Sizes of Predefined Types

The following table lists the predefined types, their size as returned by the "sizeof" function, the size of an argument of that type and the registers used to pass that argument if it was the only argument in the argument list.

Basic Type	*"sizeof"*	*Argument Size*	*Registers Used*
char	1	4	[EAX]
short int	2	4	[EAX]
int	4	4	[EAX]
long int	4	4	[EAX]
float	4	8	[EDX EAX]

double	8	8	[EDX EAX]
near pointer	4	4	[EAX]
far pointer	6	8	[EDX EAX]

Note that the size of the argument listed in the table assumes that no function prototypes are specified. Function prototypes affect the way arguments are passed. This will be discussed in the section entitled "Effect of Function Prototypes on Arguments".

Notes:

1. Provided no function prototypes exist, an argument will be converted to a default type as described in the following table.

Argument Type	***Passed As***
char	unsigned int
signed char	signed int
unsigned char	unsigned int
short	unsigned int
signed short	signed int
unsigned short	unsigned int
float	double

10.4.3 Size of Enumerated Types

The integral type of an enumerated type is determined by the values of the enumeration constants. In strict ISO/ANSI C mode, all enumerated constants are of type `int.` In the extensions mode, the compiler will use the smallest integral type possible (excluding `long` ints) that can represent all values of the enumerated type. For instance, if the minimum and maximum values of the enumeration constants are in the range −128 and 127, the enumerated type will be equivalent to a `signed char` (size = 1 byte). All references to enumerated constants in the previous instance will have type `signed char.` An enumerated constant is always promoted to an `int` when passed as an argument.

10.4.4 Effect of Function Prototypes on Arguments

Function prototypes define the types of the formal parameters of a function. Their appearance affects the way in which arguments are passed. An argument will be converted to the type of the corresponding formal parameter in the function prototype. Consider the following example.

```
void prototype( float x, int i );

void main()
{
  float x;
  int   i;

  x = 3.14;
  i = 314;
  prototype( x, i );
  rtn( x, i );
}
```

The function prototype for `prototype` specifies that the first argument is to be passed as a "float" and the second argument is to be passed as an "int". This results in the first argument being passed in register EAX and the second argument being passed in register EDX.

If no function prototype is given, as is the case for the function `rtn,` the first argument will be passed as a "double" and the second argument would be passed as an "int". This results in the first argument being passed in registers EDX and EAX and the second argument being passed in register EBX.

Note that even though both `prototype` and `rtn` were called with identical argument lists, the way in which the arguments were passed was completely different simply because a function prototype for `prototype` was specified. Function prototyping is an excellent way to guarantee that arguments will be passed as expected to your assembly language function.

10.4.5 Interfacing to Assembly Language Functions

Consider the following example.

Example:

```
void main()
{
    double   x;
    int      i;
    double   y;

    x = 7;
    i = 77;
    y = 777;
    myrtn( x, i, y );
}
```

`myrtn` is an assembly language function that requires three arguments. The first argument is of type "double", the second argument is of type "int" and the third argument is again of type "double". Using the rules for register-based calling conventions, these arguments will be passed to `myrtn` in the following way:

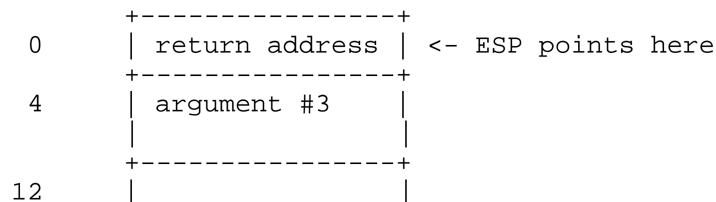
1. The first argument will be passed in registers EDX and EAX leaving EBX and ECX as available registers for other arguments.

2. The second argument will be passed in register EBX leaving ECX as an available register for other arguments.

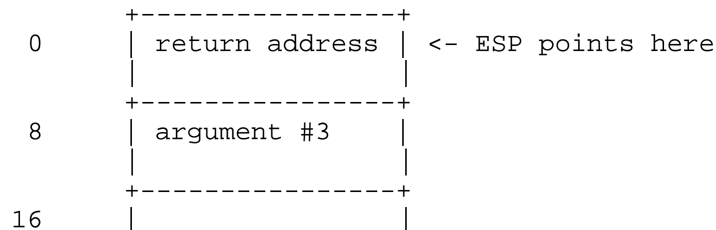
3. The third argument will not fit in register ECX (its size is 8 bytes) and hence will be pushed on the stack.

Let us look at the stack upon entry to `myrtn`.

Small Code Model
Offset



Big Code Model
Offset



Notes:

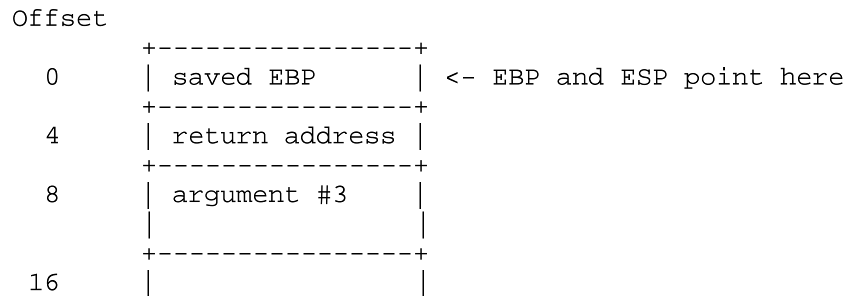
1. The return address is the top element on the stack. In a small code model, the return address is 1 double word (32 bits); in a big code model, the return address is 2 double words (64 bits).

Register EBP is normally used to address arguments on the stack. Upon entry to the function, register EBP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

```
push    EBP            ; save current value of EBP
mov     EBP,ESP        ; get access to arguments
```

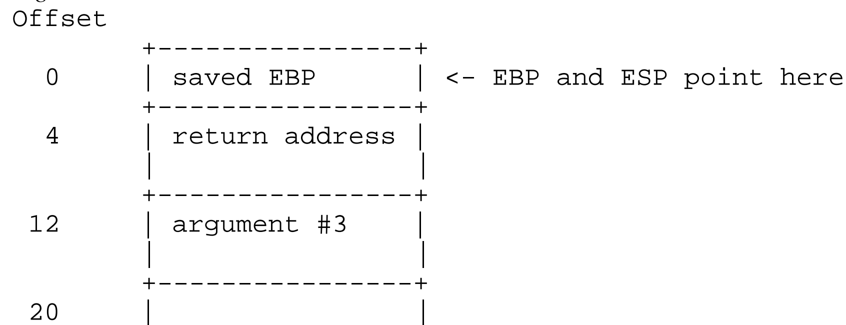
After executing these instructions, the stack looks like this.

Small Code Model



```
Offset
        +----------------+
  0     | saved EBP      | <- EBP and ESP point here
        +----------------+
  4     | return address |
        +----------------+
  8     | argument #3    |
        |                |
        +----------------+
 16     |                |
```

Big Code Model



```
Offset
        +----------------+
  0     | saved EBP      | <- EBP and ESP point here
        +----------------+
  4     | return address |
        |                |
        +----------------+
 12     | argument #3    |
        |                |
        +----------------+
 20     |                |
```

As the above diagrams show, the third argument is at offset 8 from register EBP in a small code model and offset 12 in a big code model.

Upon exit from `myrtn,` we must restore the value of EBP. The following two instructions achieve this.

```
mov     ESP,EBP        ; restore stack pointer
pop     EBP            ; restore EBP
```

The following is a sample assembly language function which implements `myrtn.`

Small Memory Model (small code, small data)

```
DGROUP     group    _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume   CS:_TEXT
           assume   DS:DGROUP
           public   myrtn_
myrtn_     proc     near
           push     EBP          ; save EBP
           mov      EBP,ESP      ; get access to arguments
;
; body of function
;
           mov      ESP,EBP      ; restore ESP
           pop      EBP          ; restore EBP
           ret      8            ; return and pop last arg
myrtn_     endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group    _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume   CS:MYRTN_TEXT
           public   myrtn_
myrtn_     proc     far
           push     EBP          ; save EBP
           mov      EBP,ESP      ; get access to arguments
;
; body of function
;
           mov      ESP,EBP      ; restore ESP
           pop      EBP          ; restore EBP
           ret      8            ; return and pop last arg
myrtn_     endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must be followed with an underscore. Global variable names must be preceded with an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

8. If any of the arguments was pushed onto the stack, the called routine must pop those arguments off the stack in the "ret" instruction.

10.4.6 Using Stack-Based Calling Conventions

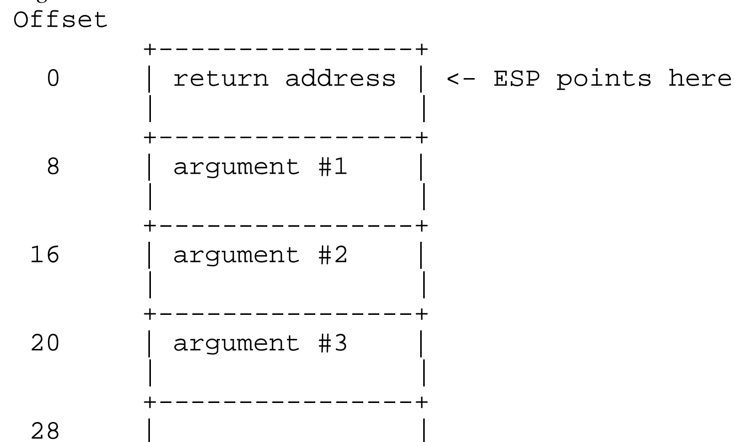
Let us now consider the example in the previous section except this time we will use the stack-based calling convention. The most significant difference between the stack-based calling convention and the register-based calling convention is the way the arguments are passed. When using the stack-based calling conventions, no registers are used to pass arguments. Instead, all arguments are passed on the stack.

Let us look at the stack upon entry to `myrtn.`

Small Code Model

```
Offset
        +----------------+
  0     | return address | <- ESP points here
        +----------------+
  4     | argument #1    |
        |                |
        +----------------+
 12     | argument #2    |
        |                |
        +----------------+
 16     | argument #3    |
        |                |
        +----------------+
 24     |                |
```

Big Code Model



```
Offset
        +----------------+
  0     | return address | <- ESP points here
        |                |
        +----------------+
  8     | argument #1    |
        |                |
        +----------------+
 16     | argument #2    |
        |                |
        +----------------+
 20     | argument #3    |
        |                |
        +----------------+
 28     |                |
```

Notes:

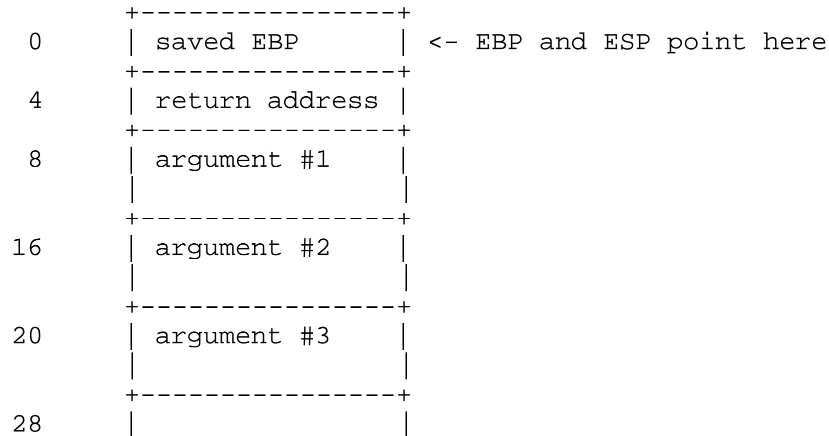
1. The return address is the top element on the stack. In a small code model, the return address is 1 double word (32 bits); in a big code model, the return address is 2 double words (64 bits).

Register EBP is normally used to address arguments on the stack. Upon entry to the function, register EBP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

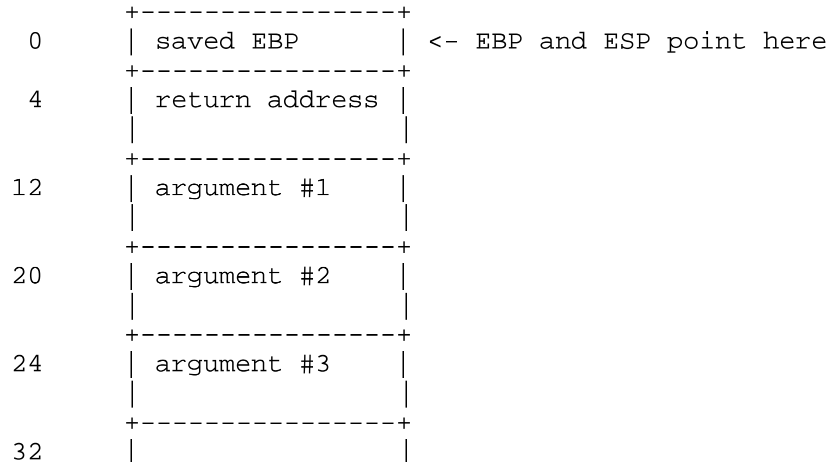
```
push    EBP             ; save current value of EBP
mov     EBP,ESP         ; get access to arguments
```

After executing these instructions, the stack looks like this.

Small Code Model
Offset



Big Code Model
Offset



As the above diagrams show, the argument are all on the stack and are referenced by specifying an offset from register EBP.

Upon exit from `myrtn,` we must restore the value of EBP. The following two instructions achieve this.

```
mov     ESP,EBP         ; restore stack pointer
pop     EBP             ; restore EBP
```

The following is a sample assembly language function which implements `myrtn`.

Small Memory Model (small code, small data)

```
DGROUP     group    _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume   CS:_TEXT
           assume   DS:DGROUP
           public   myrtn
myrtn      proc     near
           push     EBP          ; save EBP
           mov      EBP,ESP      ; get access to arguments
;
; body of function
;
           mov      ESP,EBP      ; restore ESP
           pop      EBP          ; restore EBP
           ret
           ; return
myrtn      endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group    _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume   CS:MYRTN_TEXT
           public   myrtn
myrtn      proc     far
           push     EBP          ; save EBP
           mov      EBP,ESP      ; get access to arguments
;
; body of function
;
           mov      ESP,EBP      ; restore ESP
           pop      EBP          ; restore EBP
           ret
           ; return
myrtn      endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must not be followed with an underscore as was the case with the register-based calling convention. Global variable names must not be preceded with an underscore as was the case with the register-based calling convention.

2. All used 80x86 registers except registers EAX, ECX and EDX must be saved on entry and restored on exit. Segment registers DS and ES must also be saved on entry and restored on exit. Segment register ES does not have to be saved and restored when using a memory model that is not a small data model. Note that segment registers only have to be saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

8. The caller is responsible for removing arguments from the stack.

10.4.7 Functions with Variable Number of Arguments

A function prototype with a parameter list that ends with ",..." has a variable number of arguments. In this case, all arguments are passed on the stack. Since no prototyping information exists for arguments represented by ",...", those arguments are passed as described in the section "Passing Arguments".

10.4.8 Returning Values from Functions

The way in which function values are returned depends on the size of the return value. The following examples describe how function values are to be returned. They are coded for a small code model.

1. 1-byte values are to be returned in register AL.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret1_
Ret1_   proc    near   ; char Ret1()
        mov     AL,'G'
        ret
Ret1_   endp
_TEXT   ends
        end
```

2. 2-byte values are to be returned in register AX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret2_
Ret2_   proc    near   ; short int Ret2()
        mov     AX,77
        ret
Ret2_   endp
_TEXT   ends
        end
```

3. 4-byte values are to be returned in register EAX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret4_
Ret4_   proc    near   ; int Ret4()
        mov     EAX,7777777
        ret
Ret4_   endp
_TEXT   ends
        end
```

4. 8-byte values, except structures, are to be returned in registers EDX and EAX. When using the "fpc" (floating-point calls) option, "float" and "double" are returned in registers. See section "Returning Values in 80x87-based Applications" when using the "fpi" or "fpi87" options.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret8_
Ret8_   proc    near   ; double Ret8()
        mov     EDX,dword ptr CS:Val8+4
        mov     EAX,dword ptr CS:Val8
        ret
Val8:   dq      7.7
Ret8_   endp
_TEXT   ends
        end
```

The ".8087" pseudo-op must be specified so that all floating-point constants are generated in 8087 format.

5. Otherwise, the caller allocates space on the stack for the return value and sets register ESI to point to this area. In a big data model, register ESI contains an offset relative to the segment value in segment register SS.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  RetX_
;
; struct int_values {
;     int value1, value2, value3, value4, value5;
;                     };
;
RetX_   proc    near ; struct int_values RetX()
        mov     dword ptr SS:0[ESI],71
        mov     dword ptr SS:4[ESI],72
        mov     dword ptr SS:8[ESI],73
        mov     dword ptr SS:12[ESI],74
        mov     dword ptr SS:16[ESI],75
        ret
RetX_   endp
_TEXT   ends
        end
```

When returning values on the stack, remember to use a segment override to the stack segment (SS).

The following is an example of a Open Watcom C/C++ program calling the above assembly language subprograms.

```
#include <stdio.h>

struct int_values {
    int value1;
    int value2;
    int value3;
    int value4;
    int value5;
};

extern  char               Ret1(void);
extern  short int          Ret2(void);
extern  long int           Ret4(void);
extern  double             Ret8(void);
extern  struct int_values RetX(void);

void main()
{
    struct int_values x;

    printf( "Ret1 = %c\n", Ret1() );
    printf( "Ret2 = %d\n", Ret2() );
    printf( "Ret4 = %ld\n", Ret4() );
    printf( "Ret8 = %f\n", Ret8() );
    x = RetX();
    printf( "RetX1 = %d\n", x.value1 );
    printf( "RetX2 = %d\n", x.value2 );
    printf( "RetX3 = %d\n", x.value3 );
    printf( "RetX4 = %d\n", x.value4 );
    printf( "RetX5 = %d\n", x.value5 );
}
```

The above function should be compiled for a small code model (use the "mf", "ms" or "mc" compiler option).

> ***Note:*** Returning values from functions in the stack-based calling convention is the same as returning values from functions in the register-based calling convention when using the "fpc" option.

10.5 Calling Conventions for 80x87-based Applications

When a source file is compiled by Open Watcom C/C++ with one of the "fpi" or "fpi87" options, all floating-point arguments are passed on the 80x86 stack. The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, it is assigned a position on the 80x86 stack.

> ***Note:*** When compiling using the "fpi" or "fpi87" options, the method used for passing floating-point arguments in the stack-based calling convention is identical to the method used in the register-based calling convention. However, when compiling using the "fpi" or "fpi87" options, the method used for returning floating-point values in the stack-based calling convention is different from the method used in the register-based calling convention. The register-based calling convention returns floating-point values in ST(0), whereas the stack-based calling convention returns floating-point values in EDX and EAX.

10.5.1 Passing Values in 80x87-based Applications

Consider the following example.

Example:

```
extern  void    myrtn(int,float,double,long int);

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77
    j = 77;
    myrtn( i, x, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument is of type "int" (4 bytes), the second argument is of type "float" (4 bytes), the third argument is of type "double" (8 bytes) and the fourth argument is of type "long int" (4 bytes).

When using the stack-based calling conventions, all of the arguments will be passed on the stack. When using the register-based calling conventions, the above arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in register EAX leaving EBX, ECX and EDX as available registers for other arguments.

2. The second argument will be passed on the 80x86 stack since it is a floating-point argument.

3. The third argument will also be passed on the 80x86 stack since it is a floating-point argument.

4. The fourth argument will be passed on the 80x86 stack since a previous argument has been assigned a position on the 80x86 stack.

Remember, arguments are pushed on the stack from right to left. That is, the rightmost argument is pushed first.

Any assembly language function must obey the following rule.

1. All arguments passed on the stack must be removed by the called function.

The following is a sample assembly language function which implements `myrtn`.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  myrtn_
myrtn_  proc    near
;
; body of function
;
        ret 16          ; return and pop arguments
myrtn_  endp
_TEXT   ends
        end
```

Notes:

1. Function names must be followed by an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option. In this example, EAX does not have to be saved as it was used to pass the first argument. Floating-point registers can be modified without saving their contents.

3. The direction flag must be clear before returning to the caller.

4. This function has been written for a small code model. Any segment containing executable code must belong to the class "CODE" and the segment "_TEXT". On entry, CS contains the segment address of the segment "_TEXT". The above restrictions do not apply in a big code memory model.

5. When writing assembly language functions for a small code model, you must declare them as "near". If you wish to write assembly language functions for a big code model, you must declare them as "far".

10.5.2 Returning Values in 80x87-based Applications

When using the stack-based calling conventions with "fpi" or "fpi87", floating-point values are returned in registers. Single precision values are returned in EAX, and double precision values are returned in EDX:EAX.

When using the register-based calling conventions with "fpi" or "fpi87", floating-point values are returned in ST(0). All other values are returned in the manner described earlier in this chapter.

11 32-bit Pragmas

11.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas allow you to specify certain compiler options.
- Pragmas can be used to direct the Open Watcom C/C++ code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Open Watcom C/C++.
- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the C/C++ language level. The code generator can use this information to generate more efficient code.
- Any sequence of in-line machine language instructions, including QNX function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc] The item *abc* is optional.

{abc} The item *abc* may be repeated zero or more times.

a|b|c One of *a, b* or *c* may be specified.

a ::= b The item *a* is defined in terms of *b.*

(a) Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify options
- pragmas that specify default libraries
- pragmas that describe the way structures are stored in memory
- pragmas that provide auxiliary information used for code generation

11.2 Using Pragmas to Specify Options

Currently, the following options can be specified with pragmas:

unreferenced The "unreferenced" option controls the way Open Watcom C/C++ handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause Open Watcom C/C++ to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause Open Watcom C/C++ to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if "unreferenced" was specified.

check_stack The "check_stack" option controls the way stack overflows are to be handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When "check_stack" is on, Open Watcom C/C++ will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

If a stack overflow does occur during execution and you are sure that your program is not in error (i.e. it is not unnecessarily recursing), you must increase the stack size. This is done by linking your application again and specifying the "STACK" option to the Open Watcom Linker with a larger stack size.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

reuse_duplicate_strings (C only) (C Only) The "reuse_duplicate_strings" option controls the way Open Watcom C handles identical strings in an expression. For example,

```
#pragma on (reuse_duplicate_strings);
```

will cause Open Watcom C to reuse identical strings in an expression. This is the default. Specifying

```
#pragma off (reuse_duplicate_strings);
```

will cause Open Watcom C to generate additional copies of the identical string. The following example shows where this may be of importance to the way the application behaves.

Example:

```
#include <stdio.h>

#pragma off (reuse_duplicate_strings)


void poke( char *, char * );

void main()
  {
    poke( "Hello world\n", "Hello world\n" );
  }

void poke( char *x, char *y )
  {
    x[3] = 'X';
    printf( x );
    y[4] = 'Y';
    printf( y );
  }
/*
Default output:
HelXo world
HelXY world
*/
```

11.3 Using Pragmas to Specify Default Libraries

Default libraries are specified in special object module records. Library names are extracted from these special records by the Open Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

By default, that is if no library pragma is specified, the Open Watcom C/C++ compiler generates, in the object file defining the main program, default libraries corresponding to the memory model and floating-point model used to compile the file. For example, if you have compiled the source file containing the main program for the flat memory model and the floating-point calls floating-point model, the libraries "clib3r" and "math3r" will be placed in the object file.

If you wish to add your own default libraries to this list, you can do so with a library pragma. Consider the following example.

```
#pragma library (mylib);
```

The name "mylib" will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as '/', ':' or ',' (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("/usr/lib/graph.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "/usr/lib/graph.lib");
```

11.4 The ALLOC_TEXT Pragma (C Only)

The "alloc_text" pragma can be used to specify the name of the text segment into which the generated code for a function, or a list of functions, is to be placed. The following describes the form of the "alloc_text" pragma.

```
#pragma alloc_text ( seg_name, fn {, fn} ) [;]
```

where ***description:***

seg_name is the name of the text segment.

fn is the name of a function.

Consider the following example.

```
extern int fn1(int);
extern int fn2(void);
#pragma alloc_text ( my_text, fn1, fn2 );
```

The code for the functions `fn1` and `fn2` will be placed in the segment `my_text`. Note: function prototypes for the named functions must exist prior to the "alloc_text" pragma.

11.5 The CODE_SEG Pragma

The "code_seg" pragma can be used to specify the name of the text segment into which the generated code for functions is to be placed. The following describes the form of the "code_seg" pragma.

```
#pragma code_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the text segment optionally enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_CODE_SEG"
#pragma code_seg ( seg_name );
```

class_name is the optional class name of the text segment and may be enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma code_seg ( "MY_CODE_SEG", class_name );
```

Consider the following example.

```
#pragma code_seg ( my_text );

int incr( int i )
{
    return( i + 1 );
}

int decr( int i )
{
    return( i - 1 );
}
```

The code for the functions `incr` and `decr` will be placed in the segment `my_text`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma code_seg ();
```

11.6 The COMMENT Pragma

The "comment" pragma can be used to place a comment record in an object file or executable file. The following describes the form of the "comment" pragma.

```
#pragma comment ( comment_type [, "comment_string"] ) [;]
```

where ***description:***

comment_type specifies the type of comment record. The allowable comment types are:

lib Default libraries are specified in special object module records. Library names are extracted from these special records by the Open Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

The "lib" form of this pragma offers the same features as the "library" pragma. See the section entitled "Using Pragmas to Specify Default Libraries" on page 182 for more information.

"comment_string" is an optional string literal that provides additional information for some comment types.

Consider the following example.

```
#pragma comment ( lib, "mylib" );
```

11.7 The DATA_SEG Pragma

The "data_seg" pragma can be used to specify the name of the segment into which data is to be placed. The following describes the form of the "data_seg" pragma.

```
#pragma data_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the data segment and may be enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_DATA_SEG"
#pragma data_seg ( seg_name );
```

class_name is the optional class name of the data segment and may be enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma data_seg ( "MY_DATA_SEG", class_name );
```

Consider the following example.

```
#pragma data_seg ( my_data );

static int i;
static int j;
```

The data for `i` and `j` will be placed in the segment `my_data`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma data_seg ();
```

11.8 The DISABLE_MESSAGE Pragma (C Only)

The "disable_message" pragma disables the issuance of specified diagnostic messages. The form of the "disable_message" pragma is as follows.

```
#pragma disable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C Diagnostic Messages" on page 463. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The ENABLE_MESSAGE Pragma (C Only)".

11.9 The DUMP_OBJECT_MODEL Pragma (C++ Only)

The "dump_object_model" pragma causes the C++ compiler to print information about the object model for an indicated class or an enumeration name to the diagnostics file. For class names, this information includes the offsets and sizes of fields within the class and within base classes. For enumeration names, this information consists of a list of all the enumeration constants with their values.

The general form of the "dump_object_model" pragma is as follows.

```
#pragma dump_object_model class [;]
#pragma dump_object_model enumeration [;]
class ::= a defined C++ class free of errors
enumeration ::= a defined C++ enumeration name
```

This pragma is designed to be used for information purposes only.

11.10 The ENABLE_MESSAGE Pragma (C Only)

The "enable_message" pragma re-enables the issuance of specified diagnostic messages that have been previously disabled. The form of the "enable_message" pragma is as follows.

```
#pragma enable_message ( msg_num {, msg_num} ) [;]
```

where	*description:*
msg_num	is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C Diagnostic Messages" on page 463. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The DISABLE_MESSAGE Pragma (C Only)" on page 186.

11.11 The ENUM Pragma

The "enum" pragma affects the underlying storage-definition for subsequent *enum* declarations. The forms of the "enum" pragma are as follows.

```
#pragma enum int [;]
#pragma enum minimum [;]
#pragma enum original [;]
#pragma enum pop [;]
```

where	*description:*
int	Make *int* the underlying storage definition (same as the "ei" compiler option).
minimum	Minimize the underlying storage definition (same as not specifying the "ei" compiler option).
original	Reset back to the original compiler option setting (i.e., what was or was not specified on the command line).
pop	Restore the previous setting.

The first three forms all push the previous setting before establishing the new setting.

11.12 The ERROR Pragma

The "error" pragma can be used to issue an error message with the specified text. The following describes the form of the "error" pragma.

```
#pragma error "error text" [;]
```

where ***description:***

"error text" is the text of the message that you wish to display.

You should use the ISO ***#error*** directive rather than this pragma. This pragma is provided for compatibility with legacy code. The following is an example.

```
#if defined(__386__)
    ...
#elseif defined(__86__)
    ...
#else
#pragma error ( "neither __386__ or __86__ defined" );
#endif
```

11.13 The EXTREF Pragma

The "extref" pragma is used to generate a reference to an external function or data item. The form of the "extref" pragma is as follows.

```
#pragma extref name [;]
```

where ***description:***

name is the name of an external function or data item. It must be declared to be an external function or data item before the pragma is encountered. In C++, when *name* is a function, it must not be overloaded.

This pragma causes an external reference for the function or data item to be emitted into the object file even if that function or data item is not referenced in the module. The external

reference will cause the linker to include the module containing that name in the linked program or DLL.

This is useful for debugging since you can cause debugging routines (callable from within debugger) to be included into a program or DLL to be debugged.

In C++, you can also force constructors and/or destructors to be called for a data item without necessarily referencing the data item anywhere in your code.

11.14 The FUNCTION Pragma

Certain functions, such as those listed in the description of the "oi" and "om" options, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" or "om" option or using an "intrinsic" pragma. The "function" pragma can be used to remove the intrinsic attribute for a specified list of functions.

The following describes the form of the "function" pragma.

```
#pragma function ( fn {, fn} ) [;]
```

where	***description:***
fn	is the name of a function.

Suppose the following source code was compiled using the "om" option so that when one of the special math functions is referenced, the intrinsic form will be used. In our example, we have referenced the function `sin` which does have an intrinsic form. By specifying `sin` in a "function" pragma, the intrinsic attribute will be removed, causing the function `sin` to be treated as a regular user-defined function.

```
#include <math.h>
#pragma function( sin );

double test( double x )
{
    return( sin( x ) );
}
```

11.15 Setting Priority of Static Data Initialization (C++ Only)

The "initialize" pragma sets the priority for initialization of static data in the file. This priority only applies to initialization of static data that requires the execution of code. For example, the initialization of a class that contains a constructor requires the execution of the constructor. Note that if the sequence in which initialization of static data in your program takes place has no dependencies, the "initialize" pragma need not be used.

The general form of the "initialize" pragma is as follows.

```
#pragma initialize [before | after] priority [;]

priority ::= n | library | program
```

where	*description:*
n	is a number representing the priority and must be in the range 0-255. The larger the priority, the later the point at which initialization will occur.

Priorities in the range 0-20 are reserved for the C++ compiler. This is to ensure that proper initialization of the C++ run-time system takes place before the execution of your program. The "library" keyword represents a priority of 32 and can be used for class libraries that require initialization before the program is initialized. The "program" keyword represents a priority of 64 and is the default priority for any compiled code. Specifying "before" adjusts the priority by subtracting one. Specifying "after" adjusts the priority by adding one.

A source file containing the following "initialize" pragma specifies that the initialization of static data in the file will take place before initialization of all other static data in the program since a priority of 63 will be assigned.

Example:

```
#pragma initialize before program
```

If we specify "after" instead of "before", the initialization of the static data in the file will occur after initialization of all other static data in the program since a priority of 65 will be assigned.

Note that the following is equivalent to the "before" example

Example:

```
#pragma initialize 63
```

and the following is equivalent to the "after" example.

Example:

```
#pragma initialize 65
```

The use of the "before", "after", and "program" keywords are more descriptive in the intent of the pragmas.

It is recommended that a priority of 32 (the priority used when the "library" keyword is specified) be used when developing class libraries. This will ensure that initialization of static data defined by the class library will take place before initialization of static data defined by the program. The following "initialize" pragma can be used to achieve this.

Example:

```
#pragma initialize library
```

11.16 The INLINE_DEPTH Pragma (C++ Only)

When an in-line function is called, the function call may be replaced by the in-line expansion for that function. This in-line expansion may include calls to other in-line functions which can also be expanded. The "inline_depth" pragma can be used to set the number of times this expansion of in-line functions will occur for a call.

The form of the "inline_depth" pragma is as follows.

```
#pragma inline_depth [(] n [)] [;]
```

where	***description:***
n	is the depth of expansion. If `n` is 0, no expansion will occur. If `n` is 1, only the original call is expanded. If `n` is 2, the original call and the in-line functions invoked by the original function will be expanded. The default value for `n` is 3. The maximum value for `n` is 255. Note that no expansion of recursive in-line functions occur unless enabled using the "inline_recursion" pragma.

11.17 The INLINE_RECURSION Pragma (C++ Only)

The "inline_recursion" pragma controls the recursive expansion of inline functions. The form of the "inline_recursion" pragma is as follows.

```
#pragma inline_recursion [(] on | off [)] [;]
```

Specifying "on" will enable expansion of recursive inline functions. The depth of expansion is specified by the "inline_depth" pragma. The default depth is 3. Specifying "off" suppresses expansion of recursive inline functions. This is the default.

11.18 The INTRINSIC Pragma

Certain functions, those listed in the description of the "oi" option, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" option or using an "intrinsic" pragma.

The following describes the form of the "intrinsic" pragma.

```
#pragma intrinsic ( fn {, fn} ) [;]
```

where	***description:***
fn	is the name of a function.

Suppose the following source code was compiled without using the "oi" option so that no function had the intrinsic attribute. If we wanted the intrinsic form of the `sin` function to be used, we could specify the function in an "intrinsic" pragma.

```
#include <math.h>
#pragma intrinsic( sin );

double test( double x )
{
    return( sin( x ) );
}
```

11.19 The MESSAGE Pragma

The "message" pragma can be used to issue a message with the specified text to the standard output without terminating compilation. The following describes the form of the "message" pragma.

```
#pragma message ( "message text" ) [;]
```

where ***description:***

"message text" is the text of the message that you wish to display.

The following is an example.

```
#if defined(__386__)
    ...
#else
#pragma message ( "assuming 16-bit compile" );
#endif
```

11.20 The ONCE Pragma

The "once" pragma can be used to indicate that the file which contains this pragma should only be opened and processed "once". The following describes the form of the "once" pragma.

```
#pragma once [;]
```

Assume that the file "foo.h" contains the following text.

Example:

```
#ifndef _FOO_H_INCLUDED
#define _FOO_H_INCLUDED
#pragma once
     .
     .
     .
#endif
```

The first time that the compiler processes "foo.h" and encounters the "once" pragma, it records the file's name. Subsequently, whenever the compiler encounters a `#include` statement that refers to "foo.h", it will not open the include file again. This can help speed up processing of `#include` files and reduce the time required to compile an application.

11.21 The PACK Pragma

The "pack" pragma can be used to control the way in which structures are stored in memory. There are 4 forms of the "pack" pragma.

The following form of the "pack" pragma can be used to change the alignment of structures and their fields in memory.

```
#pragma pack ( n ) [;]
```

where ***description:***

n is 1, 2, 4, 8 or 16 and specifies the method of alignment.

The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4, 8 or 16, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

```
                      zp1     zp2     zp4     zp8     zp16
sizeof(member)  \---------------------------------------
        1       |   0       0       0       0       0
        2       |   0       2       2       2       2
        4       |   0       2       4       4       4
        8       |   0       2       4       8       8
        16      |   0       2       4       8       16
        x       |   aligned to largest member
```

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8. If the largest member of structure "x" is 16 bytes then "x" is aligned according to row 16.

If no value is specified in the "pack" pragma, a default value of 8 is used. Note that the default value can be changed with the "zp" Open Watcom C/C++ compiler command line option.

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack.

```
#pragma pack ( push ) [;]
```

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack and set the current alignment.

```
#pragma pack ( push, number ) [;]
```

The following form of the "pack" pragma can be used to restore the previous alignment amount from an internal stack.

```
#pragma pack ( pop ) [;]
```

11.22 The READ_ONLY_FILE Pragma

Explicit listing of dependencies in a makefile can often be tedious in the development and maintenance phases of a project. The Open Watcom C/C++ compiler will insert dependency information into the object file as it processes source files so that a complete snapshot of the files necessary to build the object file are recorded. The "read_only_file" pragma can be used to prevent the name of the source file that includes it from being included in the dependency information that is written to the object file.

This pragma is commonly used in system header files since they change infrequently (and, when they do, there should be no impact on source files that have included them).

The form of the "read_only_file" pragma follows.

```
#pragma read_only_file [;]
```

For more information on make dependencies, see the section entitled "Automatic Dependency Detection (.AUTODEPEND)" in the ***Open Watcom C/C++ Tools User's Guide.***

11.23 The TEMPLATE_DEPTH Pragma (C++ Only)

The "template_depth" pragma provides a hard limit for the amount of nested template expansions allowed so that infinite expansion can be detected.

The form of the "template_depth" pragma is as follows.

```
#pragma template_depth [(] n [)] [;]
```

where ***description:***

n is the depth of expansion. If the value of `n` is less than 2, if will default to 2. If `n` is not specified, a warning message will be issued and the default value for `n` will be 100.

The following example of recursive template expansion illustrates why this pragma can be useful.

Example:

```
#pragma template_depth(10);

template <class T>
struct S {
    S<T*> x;
};

S<char> v;
```

11.24 The WARNING Pragma (C++ Only)

The "warning" pragma sets the level of warning messages. The form of the "warning" pragma is as follows.

```
#pragma warning msg_num level [;]
```

where	*description:*
msg_num	is the number of the warning message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Open Watcom C++ Diagnostic Messages" on page 501. If `msg_num` is "*", the level of all warning messages is changed to the specified level. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).
level	is a number from 0 to 9 and represents the level of the warning message. When a value of zero is specified, the warning becomes an error.

11.25 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

11.25.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. a type definition that resolves to a function type
3. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

An example of a type definition that resolves to a function type is the following.

```
typedef void (*func_type)();
```

When an auxiliary pragma refers to a such a type definition, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to each function whose type matches the specified type definition.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
#pragma aux y <attrs_1>;
code in which symbol y is referenced
code in which symbol z is referenced
#pragma aux default <attrs_2>;
#pragma aux x <attrs_3>;
```

Auxiliary attributes are assigned to x, y and z in the following way.

1. Symbol x is assigned the initial default attributes merged with the attributes specified by <attrs_2> and <attrs_3>.

2. Symbol y is assigned the initial default attributes merged with the attributes specified by <attrs_1>.

3. Symbol z is assigned the initial default attributes merged with the attributes specified by <attrs_2>.

11.25.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as

additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
#pragma aux ( sym, [far16] alias ) [;]
```

where	*description:*
sym	is any valid C/C++ identifier.
alias	is the alias name and is any valid C/C++ identifier.

The `far16` attribute should only be used on systems that permit the calling of 16-bit code from 32-bit code. Currently, the only supported operating system that allows this is 32-bit OS/2. If you have any libraries of functions or APIs that are only available as 16-bit code and you wish to access these functions and APIs from 32-bit code, you must specify the `far16` attribute. If the `far16` attribute is specified, the compiler will generate special code which allows the 16-bit code to be called from 32-bit code. Note that a `far16` function must be a function whose attributes are those specified by one of the alias names `__cdecl` or `__pascal`. These alias names will be described in a later section.

Consider the following example.

```
#pragma aux push_args parm [] ;
#pragma aux ( rtn, push_args ) ;
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed on the stack.

Let us look at an example in which the symbol is a type definition.

```
typedef void (func_type)(int);

#pragma aux push_args parm [];
#pragma aux ( func_type, push_args );

extern func_type rtn1;
extern func_type rtn2;
```

The first auxiliary pragma defines an alias name called `push_args` that specifies the mechanism to be used to pass arguments. The mechanism is to pass all arguments on the stack. The second auxiliary pragma associates the attributes specified in the first pragma with

the type definition `func_type`. Since `rtn1` and `rtn2` are of type `func_type`, arguments to either of those functions will be passed on the stack.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
#pragma aux ( alias ) sym aux_attrs [;]
```

where	*description:*
alias	is the alias name and is any valid C/C++ identifier.
sym	is any valid C/C++ identifier.
aux_attrs	are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
#pragma aux HIGH_C "*"                                \
                   parm caller []                     \
                   value no8087                       \
                   modify [eax ecx edx fs gs];
#pragma aux (HIGH_C) rtn1;
#pragma aux (HIGH_C) rtn2;
#pragma aux (HIGH_C) rtn3;
```

The routines `rtn1`, `rtn2` and `rtn3` assume the same attributes as the alias name `HIGH_C` which defines the calling convention used by the MetaWare High C compiler. Note that register ES must also be specified in the "modify" register set when using a memory model that is not a small data model. Whenever calls are made to `rtn1`, `rtn2` and `rtn3`, the MetaWare High C calling convention will be used.

Note that if the attributes of `HIGH_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1`, `rtn2` and `rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

WARNING! The alias name `HIGH_C` is just another symbol. If `HIGH_C` appeared in your source code, it would assume the attributes specified in the pragma for `HIGH_C`.

11.25.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` or `cdecl` defines the calling convention used by Microsoft compilers.

__fastcall `__fastcall` or `fastcall` defines the calling convention used by Microsoft compilers.

__fortran `__fortran` or `fortran` defines the calling convention used by Open Watcom FORTRAN compilers.

__pascal `__pascal` or `pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

__stdcall `__stdcall` or `stdcall` defines a special calling convention used by the Win32 API functions.

__syscall `__syscall` or `syscall` defines the calling convention used by the 32-bit OS/2 API functions.

__system `__system` or `system` are identical to `__syscall`.

__watcall `__watcall` or `watcall` defines the default calling convention used by Open Watcom compilers.

The following describes the attributes of the above alias names.

11.25.3.1 Predefined "__cdecl" Alias

```
#pragma aux __cdecl "_*" \
           parm caller [] \
           value struct float struct routine [eax] \
           modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register EAX.

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.3.2 Predefined "__pascal" Alias

```
#pragma aux __pascal "^" \
           parm reverse routine [] \
           value struct float struct caller [] \
           modify [eax ebx ecx edx]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value.

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

11.25.3.3 Predefined "__stdcall" Alias

```
#pragma aux __stdcall "_*@nnn" \
           parm routine [] \
           value struct struct caller [] \
           modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. All C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted.

3. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The called routine will remove the arguments from the stack.

4. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

5. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.3.4 Predefined "__syscall" Alias

```
#pragma aux __syscall "*" \
           parm caller [] \
           value struct struct caller [] \
           modify [eax ecx edx]
```

Notes:

1. Symbols names are not modified, that is, they are not adorned with leading or trailing underscores.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
#pragma aux sym obj_name [;]
```

where	*description:*
sym	is any valid C/C++ identifier.
obj_name	is any character string enclosed in double quotes.

When specifying `obj_name,` the asterisk character ('*') has a special meaning; it is a placeholder for `sym.`

In the following example, the name "myrtn" will be replaced by "myrtn_" in the object file.

```
#pragma aux myrtn "*_";
```

This is the default for all function names.

In the following example, the name "myvar" will be replaced by "_myvar" in the object file.

```
#pragma aux myvar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

The '^' character also has a special meaning. Whenever it is encountered in `obj_name,` it is replaced by the upper case version of `sym.`

In the following example, the name "myrtn" will be replaced by "MYRTN" in the object file.

```
#pragma aux myrtn "^";
```

11.25.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#pragma aux sym far [;]
     or
#pragma aux sym near [;]
     or
#pragma aux sym = in_line [;]

in_line ::= { const | (seg id) | (offset id) | (reloff id)
          | "asm" }
```

where	*description:*
sym	is a function name.
const	is a valid C/C++ integer constant.
id	is any valid C/C++ identifier.
seg	specifies the segment of the symbol `id`.
offset	specifies the offset of the symbol `id`.
reloff	specifies the relative offset of the symbol `id` for near control transfers.
asm	is an assembly language instruction or directive.

In the following example, Open Watcom C/C++ will generate a far call to the function `myrtn`.

```
#pragma aux myrtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Open Watcom C/C++ will generate a near call to the function `myrtn`.

```
#pragma aux myrtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Open Watcom C/C++ will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line function.

```
void mode4(void);
#pragma aux mode4 =                     \
    0xb4 0x00       /* mov AH,0 */ \
    0xb0 0x04       /* mov AL,4 */ \
    0xcd 0x10       /* int 10H  */ \
    modify [ AH AL ];
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0        ; select function "set mode"
mov   AL,4        ; specify mode (mode 4)
int   10H         ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your C/C++ program. The C prototype for the function `mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by Open Watcom C/C++.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
void mode4(void);
#pragma aux mode4 =     \
    "mov AH,0",         \
    "mov AL,4",         \
    "int 10H"           \
    modify [ AH AL ];
```

A sequence of in-line assembly language instructions may contain symbolic references. In the following example, a near call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                     \
    0xe8 reloff myalias /* near call */;
```

In the following example, a far call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                                \
    0x9a offset myalias seg myalias /* far call */;
```

11.25.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a function that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified function.

```
#pragma aux sym parm loadds [;]
```

where	***description:***
sym	is a function name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified function.

```
#pragma aux sym loadds [;]
```

where *description:*

sym is a function name.

11.25.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link libraries are exported using the Open Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
#pragma aux sym export [;]
```

where *description:*

sym is a function name.

11.25.5.3 Forcing a Stack Frame

Normally, a function contains a stack frame if arguments are passed on the stack or an automatic variable is allocated on the stack. No stack frame will be generated if the above conditions are not satisfied. The following form of the auxiliary pragma will force a stack frame to be generated under any circumstance.

```
#pragma aux sym frame [;]
```

where *description:*

sym is a function name.

11.25.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Open Watcom C/C++ is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#pragma aux sym parm { pop_info|reverse|{reg_set} } [;]

pop_info ::= caller | routine
```

where	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.25.6.1 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#pragma aux sym parm {reg_set} [;]
```

where	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **float** and **double** are always pushed on the stack when the "fpi" or "fpi87" option is used.

double Arguments of type **double** can only be passed in one of the following register pairs: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. For example, if the following register set was specified for a routine having an argument of type **double,**

```
[EBP EBX]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **double** is supported only when the "fpc" option is used. Note that this argument passing method does not include the passing of 8-byte structures.

far pointer A far pointer can only be passed in one of the following register pairs: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. For example, if a far pointer is passed to a function with the following register set,

```
[ES EBP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set.

int The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **int**) are: EAX, EBX, ECX, EDX, ESI and EDI. For example, if the following register set was specified for a routine with one argument of type **int,**

```
[EBP]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method includes 4-byte structures. Note that this argument passing method also includes arguments of type **float** but only when the "fpc" option is used.

char, short int

Arguments whose size is 1 byte or 2 bytes (e.g., arguments of type **char** and **short int** as well as 2-byte structures) are promoted to 4 bytes and are then assigned registers as if they were 4-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [EAX EBX ECX EDX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX. Specifying register EAX implies that register AX has been specified. Specifying register EBX implies that register BX has been specified. Specifying register ECX implies that register CX has been specified. Specifying register EDX implies that register DX has been specified. Specifying register EDI implies that register DI has been specified. Specifying register ESI implies that register SI has been specified. Specifying register EBP implies that register BP has been specified. Specifying register ESP implies that register SP has been specified.

3. If you are compiling for a memory model with a small data model, or the "zdp" compiler option is specified, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment. Note that the "zdf" compiler option can be used to specify that register DS does not contain that segment address of the program's data segment. In this case, register combinations containing register DS are legal.

4. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [ebp esi];
```

Suppose `myrtn` is a routine with 3 arguments each of type **double.**

1. The first argument will be passed in the register pair EDX:EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be pushed on the stack since EBP:ESI is not a valid register pair for arguments of type **double.**

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [esi edi];
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **int** and the second and third of type **double.**

1. The first argument will be passed in the register EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be passed in the register set EDI:ESI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [EAX EBX ECX EDX] is used.

11.25.6.2 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a function, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register ESI, the second argument to be passed in register EDI and the third argument to be passed in register ECX, the following auxiliary pragma can be used.

```
void mycopy( char near *, char *, int );
#pragma aux mycopy parm [ESI] [EDI] [ECX];
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

11.25.6.3 Passing Arguments to In-Line Functions

For functions whose code is generated by Open Watcom C/C++ and whose argument list is described by an auxiliary pragma, Open Watcom C/C++ has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Open Watcom C/C++ assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line function called `scrollactivepgup`.

```
void scrollactivepgup(char,char,char,char,char,char);
#pragma aux scrollactivepgup = \
    "mov AH,6"   \
    "int 10h"    \
    parm [ch] [cl] [dh] [dl] [al] [bh] \
    modify [ah];
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Open Watcom C/C++ will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **int**, it would first be converted to **char** before assigning it to register CH. Similarly, if an in-line function required its argument in register EAX and the argument was of type **short int**, the argument would be converted to **long int** before assigning it to register EAX.

In general, Open Watcom C/C++ assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **unsigned char.**

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **unsigned short int.**

3. A register set consisting of a single 32-bit register (4 bytes) is assigned a type of **unsigned long int.**

4. A register set consisting of two 32-bit registers (8 bytes) is assigned a type of **double.**

11.25.6.4 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
#pragma aux sym parm (caller | routine) [;]
```

where ***description:***

sym is a function name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

11.25.6.5 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
#pragma aux sym parm reverse [;]
```

where ***description:***

sym is a function name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for functions that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a function.

```
#pragma aux rtn parm reverse [];
```

11.25.7 Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#pragma aux sym value {no8087 | reg_set | struct_info} [;]
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.25.7.1 Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
#pragma aux sym value reg_set [;]
```

where	*description:*
sym	is a function name.
reg_set	is a register set.

Note that the method described below for returning values of type **float** or **double** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set*.

1-byte For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2-byte For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

4-byte For 4-byte return values (including near pointers), only the following register are allowed: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used. This form of the auxiliary pragma is legal for functions of type **float** when using the "fpc" option only.

far pointer For functions that return far pointers, the following register pairs are allowed: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. If no register set is specified, the registers DX:EAX will be used.

8-byte For 8-byte return values (including functions of type **double**), only the following register pairs are allowed: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. If no register set is specified, the registers EDX:EAX will be used. This form of the auxiliary pragma is legal for functions of type **double** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

3. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

11.25.7.2 Returning Structures

Typically, structures are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register ESI to point to it. The called routine then places the return value at the location pointed to by register ESI.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
#pragma aux sym value struct (caller|routine) reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register EAX by the called routine.

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: EAX, EDX, EBX, ECX, ESI or EDI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following registers: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register ESI to it.

```
#pragma aux sym value struct struct [;]
```

where ***description:***

sym is a function name.

11.25.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **float** or **double** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register ESI to it.

```
#pragma aux sym value struct float [;]
```

where ***description:***

sym is a function name.

In other words, floating-point values are to be returned in the same way structures are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **float** will be returned in register EAX. Function return values whose type is **double** will be returned in registers EDX:EAX. This is the default method for the "fpc" option.

```
#pragma aux sym value no8087 [;]
```

where ***description:***

sym is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
#pragma aux sym value [8087] [;]
```

where ***description:***

sym is a function name.

11.25.8 A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#pragma aux sym aborts [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma aux exitrtn aborts;
extern void exitrtn(void);

void rtn()
  {
    exitrtn();
  }
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Open Watcom C/C++ generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn.`

11.25.9 Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym modify nomemory [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
    while( i < 10000 ) {
        i += 383;
    }
    myrtn();
    i += 13143;
};
```

To compile the above program, "rtn.c", we issue the following command.

```
$ wcc rtn -oai -d1
$ wpp rtn -oai -d1
$ wcc386 rtn -oai -d1
$ wpp386 rtn -oai -d1
```

For illustrative purposes, we omit loop optimizations from the list of code optimizations that we want the compiler to perform. The "d1" compiler option is specified so that the object file produced by Open Watcom C/C++ contains source line information.

We can generate a file containing a disassembly of `rtn.o` by issuing the following command.

```
$ wdis rtn -l -s -r
```

The "s" option is specified so that the listing file produced by the Open Watcom Disassembler contains source lines taken from `rtn.c`. The listing file `rtn.lst` appears as follows.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000036 bytes

#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };
```

```
extern Rtn() {
 0000  52                Rtn_            push    EDX
 0001  8b 15 00 00 00 00                 mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1              cmp     EDX,00002710H
 000d  7d 08                             jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                 add     EDX,0000017fH
 0015  eb f0                             jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2              mov     _i,EDX
 001d  e8 00 00 00 00                    call    myrtn_
 0022  8b 15 00 00 00 00                 mov     EDX,_i

    i += 13143;
 0028  81 c2 57 33 00 00                 add     EDX,00003357H
 002e  89 15 00 00 00 00                 mov     _i,EDX

}
 0034  5a                                pop     EDX
 0035  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
#pragma aux myrtn modify nomemory;
```

If we compile the source file with the above pragma and disassemble the object file using the Open Watcom Disassembler, we get the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000030 bytes

#pragma off (check_stack);
#pragma aux myrtn modify nomemory;
```

```
extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                  Rtn_             push    EDX
 0001  8b 15 00 00 00 00                    mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1                 cmp     EDX,00002710H
 000d  7d 08                                jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                    add     EDX,0000017fH
 0015  eb f0                                jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2                 mov     _i,EDX
 001d  e8 00 00 00 00                       call    myrtn_

    i += 13143;
 0022  81 c2 57 33 00 00                    add     EDX,00003357H
 0028  89 15 00 00 00 00                    mov     _i,EDX

}
 002e  5a                                   pop     EDX
 002f  c3                                   ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Notice that the value of `i` is in register EDX after completion of the "while" loop. After the call to `myrtn`, the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any memory (i.e., global or static variables) that is used directly or indirectly by `Rtn` and hence register EDX contains the correct value of `i`.

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym parm nomemory modify nomemory [;]
```

where ***description:***

sym is a function name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
#pragma aux myrtn parm nomemory modify nomemory;
```

If you now compile our source file and disassemble the object file using wdis, the result is the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  0000002a bytes

#pragma off (check_stack);
#pragma aux myrtn parm nomemory modify nomemory;

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                Rtn_            push    EDX
 0001  8b 15 00 00 00 00                 mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1              cmp     EDX,00002710H
 000d  7d 08                             jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                 add     EDX,0000017fH
 0015  eb f0                             jmp     L1
```

```
    myrtn();
 0017  e8 00 00 00 00    L2              call    myrtn_

    i += 13143;
 001c  81 c2 57 33 00 00                 add     EDX,00003357H
 0022  89 15 00 00 00 00                 mov     _i,EDX

}
 0028  5a                                pop     EDX
 0029  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register EDX before calling `myrtn.` The auxiliary pragma informs the compiler that `myrtn` does not reference any memory (i.e., global or static variables) that is used directly or indirectly by `myrtn` so updating `i` was not necessary before calling `myrtn.`

11.25.10 Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a function will use without saving.

```
#pragma aux sym modify [exact] reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Specifying a register set informs Open Watcom C/C++ that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called function. Also, since the EAX register is frequently used to return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called function does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called function.

```
#pragma aux sym modify exact reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

The above form of the auxiliary pragma tells Open Watcom C/C++ not to assume that the registers used to pass arguments will be modified by the called function. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:
```
unsigned GetSP(void);
#if defined(__386__)
#pragma aux GetSP = value [esp] modify exact [];
#else
#pragma aux GetSP = value [sp] modify exact [];
#endif
```

11.25.11 An Example

As mentioned in an earlier section, the following pragma defines the calling convention for functions compiled by MetaWare’s High C compiler.

```
#pragma aux HIGH_C "*"                              \
                   parm caller []                   \
                   value no8087                     \
                   modify [eax ecx edx fs gs];
```

Note that register ES must also be specified in the "modify" register set when using a memory model with a non-small data model. Let us discuss this pragma in detail.

""* specifies that all function and variable names appear in object form as they do in source form.

parm caller [] specifies that all arguments are to be passed on the stack (an empty register set was specified) and the caller will remove the arguments from the stack.

value no8087 specifies that floating-point values are to be returned using 80x86 registers and not 80x87 floating-point registers.

modify [eax ecx edx fs gs] specifies that registers EAX, ECX, EDX, FS and GS are not preserved by the called routine.

Note that the default method of returning integer values is used; 1-byte characters are returned in register AL, 2-byte integers are returned in register AX, and 4-byte integers are returned in register EAX.

11.25.12 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile functions. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

11.25.12.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).

```
                +----------------+
        0       | 4th from top   |  ST(4)
                +----------------+
        1       | 5th from top   |  ST(5)
                +----------------+
        2       | 6th from top   |  ST(6)
                +----------------+
        3       | 7th from top   |  ST(7)
                +----------------+
  ST -> 4       | top of stack   |  ST(0)
                +----------------+
        5       | 1st from top   |  ST(1)
                +----------------+
        6       | 2nd from top   |  ST(2)
                +----------------+
        7       | 3rd from top   |  ST(3)
                +----------------+
```

Starting with version 9.5, the Open Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
#pragma aux myrtn parm [8087];

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77;
    j = 77;
    myrtn( x, i, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument of type **float** (4 bytes), the second argument is of type **int** (4 bytes), the third argument is of type **double** (8 bytes) and the fourth argument is of type **long int** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float,** will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
#pragma aux myrtn parm [eax 8087];
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register EAX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **double**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

11.25.12.2 Using the 80x87 to Return Function Values

The following form of the auxiliary pragma can be used to describe a function that returns a floating-point value in ST(0).

```
#pragma aux sym value reg_set [;]
```

where	*description:*
sym	is a function name.
reg_set	is a register set containing the string "8087", i.e. [8087].

11.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a function unless the "fpr" option is used to generate backward compatible code (older Open Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

```
#pragma aux sym modify reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

This instructs Open Watcom C/C++ to save any local variables that are located in the 80x87 cache before calling the specified routine.

In-line Assembly Language

12 In-line Assembly Language

The chapters entitled "16-bit Pragmas" on page 93 and "32-bit Pragmas" on page 179 briefly describe the use of the auxiliary pragma to create a sequence of assembly language instructions that can be placed anywhere executable C/C++ statements can appear in your source code. This chapter is devoted to an in-depth look at in-line assembly language programming.

The reasons for resorting to in-line assembly code are varied:

- Speed - You may be interested in optimizing a heavily-used section of code.
- Size - You may wish to optimize a module for size by replacing a library function call with a direct system call.
- Architecture - You may want to access certain features of the Intel x86 architecture that cannot be done so with C/C++ statements.

There are also some reasons for not resorting to in-line assembly code.

- Portability - The code is not portable to different architectures.
- Optimization - Sometimes an optimizing compiler can do a better job of arranging the instruction stream so that it is optimal for a particular processor (such as the 486 or Pentium).

12.1 In-line Assembly Language Default Environment

In next table is description of the default in-line assembler environment in dependency on C/C++ compilers CPU switch for x86 target platform.

```
Compiler  CPU        FPU        CPU extension
          directive  directive  directives
--------  ---------  ---------  --------------------------
-0        .8086      .8087
-1        .186       .8087
-2        .286p      .287
-3        .386p      .387
-4        .486p      .387
-5        .586p      .387       .K3D+.MMX
-6        .686p      .387       .K3D+.MMX+.XMM+.XMM2+.XMM3
```

This environment can be simply changed by appropriate directives.

Note:

This change is valid only for the block of assembly source code. After this block, default setting is restored.

12.2 In-line Assembly Language Tutorial

Doing in-line assembly is reasonably straight-forward with Open Watcom C/C++ although care must be exercised. You can generate a sequence of in-line assembly anywhere in your C/C++ code stream. The first step is to define the sequence of instructions that you wish to place in-line. The auxiliary pragma is used to do this. Here is a simple example based on a DOS function call that returns a far pointer to the Double-Byte Character Set (DBCS) encoding table.

Example:

```
extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "mov ax,6300h"   \
        "int 21h"        \
        value   [ds si]  \
        modify  [ax];
```

To set up the DOS call, the AH register must contain the hexadecimal value "63" (63h). A DOS function call is invoked by interrupt 21h. DOS returns a far pointer in DS:SI to a table of byte pairs in the form (start of range, end of range). On a non-DBCS system, the first pair will be (0,0). On a Japanese DBCS system, the first pair will be (81h,9Fh).

With each pragma, we define a corresponding function prototype that explains the behaviour of the function in terms of C/C++. Essentially, it is a function that does not take any arguments and that returns a far pointer to a unsigned short item.

The pragma indicates that the result of this "function" is returned in DS:SI (value [ds si]). The pragma also indicates that the AX register is modified by the sequence of in-line assembly code (modify [ax]).

Having defined our in-line assembly code, let us see how it is used in actual C code.

Example:

```
#include <stdio.h>

extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "mov ax,6300h"   \
        "int 21h"        \
        value   [ds si]  \
        modify  [ax];

void main()
{
    if( *dbcs_table() != 0 ) {
        /*
            we are running on a DOS system that
            supports double-byte characters
        */
        printf( "DBCS supported\n" );
    }
}
```

Before you attempt to compile and run this example, consider this: The program will not work! At least, it will not work in most 16-bit memory models. And it doesn't work at all in 32-bit protected mode using a DOS extender. What is wrong with it?

We can examine the disassembled code for this program in order to see why it does not always work in 16-bit real-mode applications.

```
    if( *dbcs_table() != 0 ) {
        /*
            we are running on a DOS system that
            supports double-byte characters
        */
 0007  b8 00 63                        mov     ax,6300H
 000a  cd 21                           int     21H
 000c  83 3c 00                        cmp     word ptr [si],0000H
 000f  74 0a                           je      L1

        printf( "DBCS supported\n" );
    }
 0011  be 00 00                        mov     si,offset L2
 0014  56                              push    si
 0015  e8 00 00                        call    printf_
 0018  83 c4 02                        add     sp,0002H

}
```

After the DOS interrupt call, the DS register has been altered and the code generator does nothing to recover the previous value. In the small memory model, the contents of the DS register never change (and any code that causes a change to DS must save and restore its value). It is the programmer's responsibility to be aware of the restrictions imposed by certain memory models especially with regards to the use of segmentation registers. So we must make a small change to the pragma.

```
extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "push ds"        \
        "mov ax,6300h"   \
        "int 21h"        \
        "mov di,ds"      \
        "pop ds"         \
        value   [di si]  \
        modify  [ax];
```

If we compile and run this example with a 16-bit compiler, it will work properly. We can examine the disassembled code for this revised program.

```
    if( *dbcs_table() != 0 ) {
        /*
            we are running on a DOS system that
            supports double-byte characters
        */
0008  1e                                push    ds
0009  b8 00 63                          mov     ax,6300H
000c  cd 21                             int     21H
000e  8c df                             mov     di,ds
0010  1f                                pop     ds
0011  8e c7                             mov     es,di
0013  26 83 3c 00                       cmp     word ptr es:[si],0000H
0017  74 0a                             je      L1

        printf( "DBCS supported\n" );
    }
0019  be 00 00                          mov     si,offset L2
001c  56                                push    si
001d  e8 00 00                          call    printf_
0020  83 c4 02                          add     sp,0002H
```

If you examine this code, you can see that the DS register is saved and restored by the in-line assembly code. The code generator, having been informed that the far pointer is returned in (DI:SI), loads up the ES register from DI in order to reference the far data correctly.

That takes care of the 16-bit real-mode case. What about 32-bit protected mode? When using a DOS extender, you must examine the accompanying documentation to see if the system call that you wish to make is supported by the DOS extender. One of the reasons that this particular DOS call is not so clear-cut is that it returns a 16-bit real-mode segment:offset pointer. A real-mode pointer must be converted by the DOS extender into a protected-mode pointer in order to make it useful. As it turns out, neither the Tenberry Software DOS/4G(W) nor Phar Lap DOS extenders support this particular DOS call (although others may). The issues with each DOS extender are complex enough that the relative merits of using in-line assembly code are not worth it. We present an excerpt from the final solution to this problem.

Example:

```
#ifndef __386__

extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "push ds"        \
        "mov ax,6300h"   \
        "int 21h"        \
        "mov di,ds"      \
        "pop ds"         \
        value   [di si]  \
        modify  [ax];
```

```
#else

unsigned short far * dbcs_table( void )
{
    union REGPACK       regs;
    static short        dbcs_dummy = 0;

    memset( &regs, 0, sizeof( regs ) );
    if( _IsPharLap() ) {
        PHARLAP_block pblock;

        memset( &pblock, 0, sizeof( pblock ) );
        pblock.real_eax = 0x6300;       /* get DBCS vector table */
        pblock.int_num = 0x21;          /* DOS call */
        regs.x.eax = 0x2511;            /* issue real-mode interrupt */
        regs.x.edx = FP_OFF( &pblock ); /* DS:EDX -> parameter block */
        regs.w.ds = FP_SEG( &pblock );
        intr( 0x21, &regs );
        return( firstmeg( pblock.real_ds, regs.w.si ) );
    } else if( _IsDOS4G() ) {
        DPMI_block dblock;

        memset( &dblock, 0, sizeof( dblock ) );
        dblock.eax = 0x6300;            /* get DBCS vector table */
        regs.w.ax = 0x300;              /* DPMI Simulate R-M intr */
        regs.h.bl = 0x21;               /* DOS call */
        regs.h.bh = 0;                  /* flags */
        regs.w.cx = 0;                  /* # bytes from stack */
        regs.x.edi = FP_OFF( &dblock );
        regs.x.es = FP_SEG( &dblock );
        intr( 0x31, &regs );
        return( firstmeg( dblock.ds, dblock.esi ) );
    } else {
        return( &dbcs_dummy );
    }
}

#endif
```

The 16-bit version will use in-line assembly code but the 32-bit version will use a C function that has been crafted to work with both Tenberry Software DOS/4G(W) and Phar Lap DOS extenders. The `firstmeg` function used in the example is shown below.

```
#define REAL_SEGMENT    0x34

void far *firstmeg( unsigned segment, unsigned offset )
{
    void far    *meg1;

    if( _IsDOS4G() ) {
        meg1 = MK_FP( FP_SEG( &meg1 ), ( segment << 4 ) + offset );
    } else {
        meg1 = MK_FP( REAL_SEGMENT, ( segment << 4 ) + offset );
    }
    return( meg1 );
}
```

We have taken a brief look at two features of the auxiliary pragma, the "modify" and "value" attributes.

The "modify" attribute describes those registers that are modified by the execution of the sequence of in-line code. You usually have two choices here; you can save/restore registers that are affected by the code sequence in which case they need not appear in the modify list or you can let the code generator handle the fact that the registers are modified by the code sequence. When you invoke a system function (such as a DOS or BIOS call), you should be careful about any side effects that the call has on registers. If a register is modified by a call and you have not listed it in the modify list or saved/restored it, this can have a disastrous affect on the rest of the code in the function where you are including the in-line code.

The "value" attribute describes the register or registers in which a value is returned (we use the term "returned", not in the sense that a function returns a value, but in the sense that a result is available after execution of the code sequence).

This leads the discussion into the third feature of the auxiliary pragma, the feature that allows us to place the results of C expressions into specific registers as part of the "setup" for the sequence of in-line code. To illustrate this, let us look at another example.

Example:

```
extern void BIOSSetCurPos( unsigned short __rowcol,
                           unsigned char __page );
#pragma aux BIOSSetCurPos =     \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        parm   [dx] [bh]        \
        modify [ah];
```

The "parm" attribute specifies the list of registers into which values are to be placed as part of the prologue to the in-line code sequence. In the above example, the "set cursor position" function requires three pieces of information. It requires that the cursor row value be placed in the DH register, that the cursor column value be placed in the DL register, and that the screen page number be placed in the BH register. In this example, we have decided to combine the row and column information into a single "argument" to the function. Note that the function prototype for `BIOSSetCurPos` is important. It describes the types and number of arguments to be set up for the in-line code. It also describes the type of the return value (in this case there is none).

Once again, having defined our in-line assembly code, let us see how it is used in actual C code.

Example:

```
#include <stdio.h>

extern void BIOSSetCurPos( unsigned short __rowcol,
                           unsigned char __page );
#pragma aux BIOSSetCurPos =     \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        parm   [dx] [bh]        \
        modify [ah];

void main()
{
    BIOSSetCurPos( (5 << 8) | 20, 0 );
    printf( "Hello world\n" );
}
```

To see how the code generator set up the register values for the in-line code, let us take a look at the disassembled code.

```
   BIOSSetCurPos( (5 << 8) | 20, 0 );
0008  ba 14 05                        mov     dx,0514H
000b  30 ff                           xor     bh,bh
000d  55                              push    bp
000e  b4 02                           mov     ah,02H
0010  cd 10                           int     10H
0012  5d                              pop     bp
```

As we expected, the result of the expression for the row and column is placed in the DX register and the page number is placed in the BH register. The remaining instructions are our in-line code sequence.

Although our examples have been simple, you should be able to generalize them to your situation.

To review, the "parm", "value" and "modify" attributes are used to:

1. convey information to the code generator about the way data values are to be placed in registers in preparation for the code burst (parm),

2. convey information to the code generator about the result, if any, from the code burst (value), and

3. convey information to the code generator about any side effects to the registers after the code burst has executed (modify). It is important to let the code generator

know all of the side effects on registers when the in-line code is executed; otherwise it assumes that all registers, other than those used for parameters, are preserved. In our examples, we chose to push/pop some of the registers that are modified by the code burst.

12.3 Labels in In-line Assembly Code

Labels can be used in in-line assembly code. Here is an example.

Example:
```
extern void _disable_video( unsigned );
#pragma aux _disable_video =    \
 "again: in al,dx"              \
        "test al,8"             \
        "jz again"              \
        "mov dx,03c0h"          \
        "mov al,11h"            \
        "out dx,al"             \
        "mov al,0"              \
        "out dx,al"             \
        parm [dx]               \
        modify [al dx];
```

12.4 Variables in In-line Assembly Code

To finish our discussion, we provide examples that illustrate the use of variables in the in-line assembly code. The following example illustrates the use of static variable references in the auxiliary pragma.

Example:
```
#include <stdio.h>

static short         _rowcol;
static unsigned char _page;
```

```
extern void BIOSSetCurPos( void );
#pragma aux BIOSSetCurPos =     \
        "mov  dx,_rowcol"       \
        "mov  bh,_page"         \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        modify [ah bx dx];


void main()
{
    _rowcol = (5 << 8) | 20;
    _page = 0;
    BIOSSetCurPos();
    printf( "Hello world\n" );
}
```

The only rule to follow here is that the auxiliary pragma must be defined after the variables are defined. The in-line assembler is passed information regarding the sizes of variables so they must be defined first.

If we look at a fragment of the disassembled code, we can see the result.

```
   _rowcol = (5 << 8) | 20;
0008  c7 06 00 00 14 05              mov     word ptr __rowcol,0514H

   _page = 0;
000e  c6 06 00 00 00                 mov     byte ptr __page,00H

   BIOSSetCurPos();
0013  8b 16 00 00                    mov     dx,__rowcol
0017  8a 3e 00 00                    mov     bh,__page
001b  55                             push    bp
001c  b4 02                          mov     ah,02H
001e  cd 10                          int     10H
0020  5d                             pop     bp
```

The following example illustrates the use of automatic variable references in the auxiliary pragma. Again, the auxiliary pragma must be defined after the variables are defined so the pragma is placed in-line with the function.

Example:

```
#include <stdio.h>

void main()
{
    short         _rowcol;
    unsigned char _page;

    extern void BIOSSetCurPos( void );
#   pragma aux  BIOSSetCurPos = \
        "mov  dx,_rowcol"       \
        "mov  bh,_page"         \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        modify [ah bx dx];

    _rowcol = (5 << 8) | 20;
    _page = 0;
    BIOSSetCurPos();
    printf( "Hello world\n" );
}
```

If we look at a fragment of the disassembled code, we can see the result.

```
   _rowcol = (5 << 8) | 20;
000e  c7 46 fc 14 05                    mov     word ptr -4H[bp],0514H

   _page = 0;
0013  c6 46 fe 00                       mov     byte ptr -2H[bp],00H

   BIOSSetCurPos();
0017  8b 96 fc ff                       mov     dx,-4H[bp]
001b  8a be fe ff                       mov     bh,-2H[bp]
001f  55                                push    bp
0020  b4 02                             mov     ah,02H
0022  cd 10                             int     10H
0024  5d                                pop     bp
```

You should try to avoid references to automatic variables as illustrated by this last example. Referencing automatic variables in this manner causes them to be marked as volatile and the optimizer will not be able to do a good job of optimizing references to these variables.

12.5 In-line Assembly Language using _asm

There is an alternative to Open Watcom's auxiliary pragma method for creating in-line assembly code. You can use one of the ***_asm*** or ***__asm*** keywords to imbed assembly code into the generated code. The following is a revised example of the cursor positioning example introduced above.

Example:

```
#include <stdio.h>

void main()
{
    unsigned short _rowcol;
    unsigned char _page;

    _rowcol = (5 << 8) | 20;
    _page = 0;
    _asm {
        mov     dx,_rowcol
        mov     bh,_page
        push    bp
        mov     ah,2
        int     10h
        pop     bp
    };
    printf( "Hello world\n" );
}
```

The assembly language sequence can reference program variables to retrieve or store results. There are a few incompatibilities between Microsoft and Open Watcom implementation of this directive.

__LOCAL_SIZE is not supported by Open Watcom C/C++. This is illustrated in the following example.

Example:

```
void main()
{
    int i;
    int j;

    _asm {
        push    bp
        mov     bp,sp
        sub     sp,__LOCAL_SIZE
    };
}
```

structure references are not supported by Open Watcom C/C++. This is illustrated in the following example.

Example:

```
#include <stdio.h>

struct rowcol {
    unsigned char col;
    unsigned char row;
};

void main()
{
    struct rowcol _pos;
    unsigned char _page;

    _pos.row = 5;
    _pos.col = 20;
    _page = 0;
    _asm {
        mov     dl,_pos.col
        mov     dh,_pos.row
        mov     bh,_page
        push    bp
        mov     ah,2
        int     10h
        pop     bp
    };
    printf( "Hello world\n" );
}
```

12.6 In-line Assembly Directives and Opcodes

It is not the intention of this chapter to describe assembly-language programming in any detail. You should consult a book that deals with this topic. However, we present a list of the directives, opcodes and register names that are recognized by the assembler built into the compiler's auxiliary pragma processor.

```
.186        .286        .286c       .286p       .287
.386
.386p       .387        .486        .486p       .586
.586p
.686        .686p       .8086       .8087       aaa
aad
aam         aas         adc         add         addpd
addps
addsd       addss       addsubpd    addsubps    ah
al
and         andnpd      andnps      andpd       andps
arpl
ax          bh          bl          bound       bp
bsf
bsr         bswap       bt          btc         btr
bts
bx          byte        call        callf       cbw
cdq
ch          cl          clc         cld         clflush
cli
clts        cmc         cmova       cmovae      cmovb
cmovbe
cmovc       cmove       cmovg       cmovge      cmovl
cmovle
cmovna      cmovnae     cmovnb      cmovnbe     cmovnc
cmovne
cmovng      cmovnge     cmovnl      cmovnle     cmovno
cmovnp
cmovns      cmovnz      cmovo       cmovp       cmovpe
cmovpo
cmovs       cmovz       cmp         cmpeqpd     cmpeqps
cmpeqsd
cmpeqss     cmplepd     cmpleps     cmplesd     cmpless
cmpltpd
cmpltps     cmpltsd     cmpltss     cmpneqpd    cmpneqps
cmpneqsd
cmpneqss    cmpnlepd    cmpnleps    cmpnlesd    cmpnless
cmpnltpd
cmpnltps    cmpnltsd    cmpnltss    cmpordpd    cmpordps
cmpordsd
cmpordss    cmppd       cmpps       cmps        cmpsb
cmpsd
cmpss       cmpsw       cmpunordpd  cmpunordps  cmpunordsd
cmpunordss
cmpxchg     cmpxchg8b   comisd      comiss      cpuid
cr0
cr2         cr3         cr4         cs          cvtdq2pd
```

```
cvtdq2ps
cvtpd2dq    cvtpd2pi    cvtpd2ps    cvtpi2pd    cvtpi2ps
cvtps2dq
cvtps2pd    cvtps2pi    cvtsd2si    cvtsd2ss    cvtsi2sd
cvtsi2ss
cvtss2sd    cvtss2si    cvttpd2dq   cvttpd2pi   cvttps2dq
cvttps2pi
cvttsd2si   cvttss2si   cwd         cwde        cx
daa
das         db          dd          dec         df
dh
di          div         divpd       divps       divsd
divss
dl          dp          dr0         dr1         dr2
dr3
dr6         dr7         ds          dup         dw
dword
dx          eax         ebp         ebx         ecx
edi
edx         emms        enter       es          esi
esp
f2xm1       fabs        fadd        faddp       far
fbld
fbstp       fchs        fclex       fcmovb      fcmovbe
fcmove
```

fcmovnb fcmovnbe fcmovne fcmovnu fcmovu
fcom
fcomi fcomip fcomp fcompp fcos
fdecstp
fdisi fdiv fdivp fdivr fdivrp
femms
feni ffree fiadd ficom ficomp
fidiv
fidivr fild fimul fincstp finit
fist
fistp fisttp fisub fisubr fld
fld1
fldcw fldenv fldenvd fldenvw fldl2e
fldl2t
fldlg2 fldln2 fldpi fldz fmul
fmulp
fnclex fndisi fneni fninit fnop
fnrstor
fnrstord fnrstorw fnsave fnsaved fnsavew
fnstcw
fnstenv fnstenvd fnstenvw fnstsw fpatan
fprem
fprem1 fptan frndint frstor frstord
frstorw
fs fsave fsaved fsavew fscale
fsetpm
fsin fsincos fsqrt fst fstcw
fstenv
fstenvd fstenvw fstp fstsw fsub
fsubp
fsubr fsubrp ftst fucom fucomi
fucomip
fucomp fucompp fwait fword fxam
fxch
fxrstor fxsave fxtract fyl2x fyl2xp1
gs
haddpd haddps hsubpd hsubps hlt
idiv
imul in inc ins insb
insd
insw int into invd invlpg
iret
iretd iretdf iretf ja jae
jb
jbe jc jcxz je jecxz
jg
jge jl jle jmp jmpf
jna

```
jnae        jnb         jnbe        jnc         jne
jng
jnge        jnl         jnle        jno         jnp
jns
jnz         jo          jp          jpe         jpo
js
jz          .k3d        lahf        lar         lddqu
ldmxcsr
lds         lea         leave       les         lfence
lfs
lgdt        lgs         lidt        lldt        lmsw
lock
lods        lodsb       lodsd       lodsw       loop
loopd
loope       looped      loopew      loopne      loopned
loopnew
loopnz      loopnzd     loopnzw     loopw       loopz
loopzd
loopzw      lsl         lss         ltr         maskmovdqu
maskmovq
maxpd       maxps       maxsd       maxss       mfence
minpd
minps       minsd       minss       mm0         mm1
mm2
```

mm3	mm4	mm5	mm6	mm7
.mmx				
monitor	mov	movapd	movaps	movd
movddup				
movdq2q	movdqa	movdqu	movhlps	movhpd
movhps				
movlhps	movlpd	movlps	movmskpd	movmskps
movntdq				
movnti	movntpd	movntps	movntq	movq
movq2dq				
movs	movsb	movsd	movshdup	movsldup
movss				
movsw	movsx	movupd	movups	movzx
mul				
mulpd	mulps	mulsd	mulss	mwait
near				
neg	.no87	nop	not	offset
or				
orpd	orps	out	outs	outsb
outsd				
outsw	oword	packssdw	packsswb	packuswb
paddb				
paddd	paddq	paddsb	paddsw	paddusb
paddusw				
paddw	pand	pandn	pause	pavgb
pavgusb				
pavgw	pcmpeqb	pcmpeqd	pcmpeqw	pcmpgtb
pcmpgtd				
pcmpgtw	pextrw	pf2id	pf2iw	pfacc
pfadd				
pfcmpeq	pfcmpge	pfcmpgt	pfmax	pfmin
pfmul				
pfnacc	pfpnacc	pfrcp	pfrcpit1	pfrcpit2
pfrsqit1				
pfrsqrt	pfsub	pfsubr	pi2fd	pi2fw
pinsrw				
pmaddwd	pmaxsw	pmaxub	pminsw	pminub
pmovmskb				
pmulhrw	pmulhuw	pmulhw	pmullw	pmuludq
pop				
popa	popad	popf	popfd	por
prefetch				
prefetchnta	prefetcht0	prefetcht1	prefetcht2	prefetchw
psadbw				
pshufd	pshufhw	pshuflw	pshufw	pslld
pslldq				
psllq	psllw	psrad	psraw	psrld
psrldq				

```
psrlq        psrlw        psubb        psubd        psubq
psubsb
psubsw       psubusb      psubusw      psubw        pswapd
ptr
punpckhbw    punpckhdq    punpckhqdq   punpckhwd    punpcklbw
punpckldq
punpcklqdq   punpcklwd    push         pusha        pushad
pushd
pushf        pushfd       pushw        pword        pxor
qword
rcl          rcpps        rcpss        rcr          rdmsr
rdpmc
rdtsc        rep          repe         repne        repnz
repz
ret          retd         retf         retfd        retn
rol
ror          rsm          rsqrtps      rsqrtss      sahf
sal
sar          sbb          scas         scasb        scasd
scasw
seg          seta         setae        setb         setbe
setc
sete         setg         setge        setl         setle
setna
```

```
setnae      setnb       setnbe      setnc       setne
setng
setnge      setnl       setnle      setno       setnp
setns
setnz       seto        setp        setpe       setpo
sets
setz        sfence      sgdt        shl         shld
short
shr         shrd        shufpd      shufps      si
sidt
sldt        smsw        sp          sqrtpd      sqrtps
sqrtsd
sqrtss      ss          st          stc         std
sti
stmxcsr     stos        stosb       stosd       stosw
str
sub         subpd       subps       subsd       subss
sysenter
sysexit     tbyte       test        tr3         tr4
tr5
tr6         tr7         ucomisd     ucomiss     unpckhpd
unpckhps
unpcklpd    unpcklps    verr        verw        wait
wbinvd
word        wrmsr       xadd        xchg        xlat
xlatb
xorpd       xorps       .xmm        xmm0        xmm1
.xmm2
xmm2        .xmm3       xmm3        xmm4        xmm5
xmm6
xmm7        xor
```

A separate assembler is also included with this product and is described in the ***Open Watcom C/C++ Tools User's Guide***

Open Watcom Tools

The Open Watcom Linker

13 The Open Watcom Linker

The Open Watcom Linker is a linkage editor (linker) that takes object and library files as input and produces executable files as output. The following object module and library formats are supported by the Open Watcom Linker.

- The standard Intel Object Module Format (OMF).
- Microsoft's extensions to the standard Intel OMF.
- Phar Lap's Easy OMF-386 object module format for linking 386 applications.
- The COFF object module format.
- The ELF object module format.
- The OMF library format.
- The AR (Microsoft compatible) object library format.

The Open Watcom Linker is capable of producing a number of executable file formats. The following lists these executable file formats.

- DOS executable files
- ELF executable files
- executable files that run under FlashTek's DOS extender
- executable files that run under Phar Lap's 386|DOS-Extender
- executable files that run under CauseWay DOS extender, Tenberry Software's DOS/4G and DOS/4GW DOS extenders, and compatible products
- NetWare Loadable Modules (NLMs) that run under Novell's NetWare operating system
- OS/2 executable files including Dynamic Link Libraries

- QNX executable files
- 16-bit Windows (Win16) executable files including Dynamic Link Libraries
- 32-bit Windows (Win32) executable files including Dynamic Link Libraries
- raw binary images
- Intel Hex files (Hex80, Hex86 and extended linear)

In addition to being able to generate the above executable file formats, the Open Watcom Linker also runs under a variety of operating systems. Currently, the Open Watcom Linker runs under the following operating systems.

- DOS
- OS/2
- QNX
- Windows NT/2000/XP
- Windows 95/98/Me

This guide describes only the QNX executable file format.

The Open Watcom Linker command line format is as follows.

```
wlink {directive}
```

where ***directive*** is a series of Open Watcom Linker directives specified on the command line or in one or more files. If the directives are contained within a file, the "@" character is used to reference that file. If no file extension is specified, a file extension of "lnk" is assumed.

Example:

```
wlink name testprog @first @second option map
```

In the above example, directives are specified on the command line (e.g., "name testprog" and "option map") and in files (e.g., `first.lnk` and `second.lnk`).

13.1 Using the SYSTEM Directive

For each executable file format that can be created using the Open Watcom Linker, a specific SYSTEM directive may be used. The SYSTEM directive selects a subset of the available directives necessary to create each specific executable file format.

System	***Description***
com	16-bit x86 DOS ".COM" executable
dos	16-bit x86 DOS executable
dos4g	32-bit x86 DOS/4GW executable
dos4gnz	non-zero based 32-bit x86 DOS/4GW executable
netware	32-bit x86 NetWare Loadable Module
novell	synonym for "netware"
os2	16-bit x86 OS/2 executable
os2_dll	16-bit x86 OS/2 Dynamic Link Library
os2_pm	16-bit x86 OS/2 Presentation Manager executable
os2v2	32-bit x86 OS/2 executable
os2v2_dll	32-bit x86 OS/2 Dynamic Link Library
os2v2_pm	32-bit x86 OS/2 Presentation Manager executable
pharlap	32-bit x86 Phar Lap executable
tnt	32-bit x86 Phar Lap TNT dos style executable

qnx 16-bit x86 QNX executable

qnx386 32-bit x86 QNX executable

x32r 32-bit x86 FlashTek executable using register-based calling conventions

x32rv 32-bit x86 virtual-memory FlashTek executable using register-based calling conventions

x32s 32-bit x86 FlashTek executable using stack-based calling conventions

x32sv 32-bit x86 virtual-memory FlashTek executable using stack-based calling conventions

windows 16-bit x86 Windows 3.x executable

windows_dll 16-bit x86 Windows 3.x Dynamic Link Library

win95 32-bit x86 Windows 9x executable

win95 dll 32-bit x86 Windows 9x Dynamic Link Library

nt 32-bit x86 Windows NT character-mode executable

nt_win 32-bit x86 Windows NT windowed executable

win32 synonym for "nt_win"

nt_dll 32-bit x86 Windows NT Dynamic Link Library

win386 32-bit x86 Open Watcom extended Windows 3.x executable or Dynamic Link Library

The various systems that we have listed above are defined in special linker directive files which are plain ASCII text files that you can edit. These files are called `wlink.lnk` and `wlsystem.lnk`.

The file `wlink.lnk` is a special linker directive file that is automatically processed by the Open Watcom Linker before processing any other directives. On a DOS, OS/2, or Windows-hosted system, this file must be located in one of the paths specified in the **PATH** environment variable. On a QNX-hosted system, this file should be located in the `/etc` directory. A default version of this file is located in the `\watcom\binw` directory on DOS-hosted systems, the `\watcom\binp` directory on OS/2-hosted systems, the `/etc` directory on QNX-hosted systems, and the `\watcom\binnt` directory on Windows 95 or

Windows NT-hosted systems. Note that the file `wlink.lnk` includes the file `wlsystem.lnk` which is located in the `\watcom\binw` directory on DOS, OS/2, or Windows-hosted systems and the `/etc` directory on QNX-hosted systems.

The files `wlink.lnk` and `wlsystem.lnk` reference the **WATCOM** environment variable which must be set to the directory in which you installed your software.

13.2 Linking 16-bit QNX Executable Files

To create this type of file, use the following structure.

```
system   qnx
option   map
name     app_name
file     obj1, obj2, ...
library  lib1, lib2, ...
```

For more information, see the chapter entitled "The QNX Executable File Format" on page 359.

13.3 Linking 32-bit QNX Executable Files

To create this type of file, use the following structure.

```
system   qnx386
option   map
name     app_name
file     obj1, obj2, ...
library  lib1, lib2, ...
```

For more information, see the chapter entitled "The QNX Executable File Format" on page 359.

14 Linker Directives and Options

The Open Watcom Linker supports a large set of directives and options. The following sections present these directives and options in alphabetical order.

Directives tell the Open Watcom Linker how to create your program. For example, using directives you can tell the Open Watcom Linker which object files are to be included in the program, which library files to search to resolve undefined references, and the name of the executable file.

The file `wlink.lnk` is a special linker directive file that is automatically processed by the Open Watcom Linker before processing any other directives. On a DOS, OS/2, or Windows-hosted system, this file must be located in one of the paths specified in the **PATH** environment variable. On a QNX-hosted system, this file should be located in the `/etc` directory. A default version of this file is located in the `\watcom\binw` directory on DOS-hosted systems, the `\watcom\binp` directory on OS/2-hosted systems, the `/etc` directory on QNX-hosted systems, and the `\watcom\binnt` directory on Windows 95 or Windows NT-hosted systems. Note that the file `wlink.lnk` includes the file `wlsystem.lnk` which is located in the `\watcom\binw` directory on DOS, OS/2, or Windows-hosted systems and the `/etc` directory on QNX-hosted systems.

The files `wlink.lnk` and `wlsystem.lnk` reference the **WATCOM** environment variable which must be set to the directory in which you installed your software.

It is also possible to use environment variables when specifying a directive. For example, if the **LIBDIR** environment variable is defined as follows,

```
export libdir=/test
```

then the linker directive

```
library $libdir/mylib
```

is equivalent to the following linker directive.

```
library /test/mylib
```

Note that a space must precede a reference to an environment variable.

Many directives can take a list of one or more arguments separated by commas. Instead of a comma-delimited list, you can specify a space-separated list provided the list is enclosed in braces (e.g., { space delimited list }). For example, the "FILE" directive can take a list of object file names as an argument.

```
file first,second,third,fourth
```

The alternate way of specifying this is as follows.

```
file {first second third fourth}
```

Where this comes in handy is in make files, where a list of dependents is usually a space-delimited list.

```
OBJS = first second third fourth
     .
     .
     .
     wlink file {$(objs)}
```

The following notation is used to describe the syntax of linker directives and options.

ABC All items in upper case are required.

[abc] The item *abc* is optional.

{abc} The item *abc* may be repeated zero or more times.

{abc}+ The item *abc* may be repeated one or more times.

a|b|c One of *a, b* or *c* may be specified.

a ::= b The item *a* is defined in terms of *b.*

Certain characters have special meaning to the linker. When a special character must appear in a name, you can imbed the string that makes up the name inside apostrophes (e.g., 'name@8'). This prevents the linker from interpreting the special character in its usual manner. This is also true for file or path names that contain spaces (e.g., '\program files\software\mylib'). Normally, the linker would interpret a space or blank in a file name as a separator. The special characters are listed below:

Character	Name of Character
	Blank
=	Equals
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Period
{	Left Brace
}	Right Brace
@	At Sign
#	Hash Mark
%	Percentage Symbol

14.1 The ALIAS Directive

The "ALIAS" directive is used to specify an equivalent name for a symbol name. The format of the "ALIAS" directive (short form "A") is as follows.

```
ALIAS alias_name=symbol_name{, alias_name=symbol_name}
```

where ***description:***

alias_name is the alias name.

symbol_name is the symbol name to which the alias name is mapped.

Consider the following example.

```
alias sine=mysine
```

When the linker tries to resolve the reference to `sine`, it will immediately substitute the name `mysine` for `sine` and begin searching for the symbol `mysine`.

14.2 The ARTIFICIAL Option

The "ARTIFICIAL" option should only be used if you are developing a Open Watcom C++ application. A Open Watcom C++ application contains many compiler-generated symbols. By default, the linker does not include these symbols in the map file. The "ARTIFICIAL" option can be used if you wish to include these compiler-generated symbols in the map file.

The format of the "ARTIFICIAL" option (short form "ART") is as follows.

```
OPTION ARTIFICIAL
```

14.3 The CACHE Option

The "CACHE" and "NOCACHE" options can be used to control caching of object and library files in memory by the linker. When neither the "CACHE" nor "NOCACHE" option is specified, the linker will only cache small libraries. Object files and large libraries are not cached. The "CACHE" and "NOCACHE" options can be used to alter this default behaviour. The "CACHE" option enables the caching of object files and large library files while the "NOCACHE" option disables all caching.

The format of the "CACHE" option (short form "CAC") is as follows.

```
OPTION CACHE
```

The format of the "NOCACHE" option (short form "NOCAC") is as follows.

```
OPTION NOCACHE
```

When linking large applications with many object files, caching object files will cause extensive use of memory by the linker. On virtual memory systems such as OS/2, Windows NT or Windows 95, this can cause extensive page file activity when real memory resources have been exhausted. This can degrade the performance of other tasks on your system. For this reason, the OS/2 and Windows-hosted versions of the linker do not perform object file caching by default. This does not imply that object file caching is not beneficial. If your system has lots of real memory or the linker is running as the only task on the machine, object file caching can certainly improve the performance of the linker.

On single-tasking environments such as DOS, the benefits of improved linker performance outweighs the memory demands associated with object file caching. For this reason, object file caching is performed by default on these systems. If the memory requirements of the linker exceed the amount of memory on your system, the "NOCACHE" option can be specified.

The QNX operating system is a multi-tasking real-time operating system. However, it is not a virtual memory system. Caching object files can consume large amounts of memory. This may prevent other tasks on the system from running, a problem that may be solved by using the "NOCACHE" option.

14.4 The CASEEXACT Option

The "CASEEXACT" option tells the Open Watcom Linker to respect case when resolving references to global symbols. That is, "ScanName" and "SCANNAME" represent two different symbols. This is the default because the most commonly used languages (C, C++, FORTRAN) are case sensitive. The format of the "CASEEXACT" option (short form "C") is as follows.

```
OPTION CASEEXACT
```

It is possible to override the default by using the "NOCASEEXACT" option. The "NOCASEEXACT" option turns off case-sensitive linking. The format of the "NOCASEEXACT" option (short form "NOCASE") is as follows.

```
OPTION NOCASEEXACT
```

You can specify the "NOCASEEXACT" option in the default directive files `wlink.lnk` or `wlsystem.lnk` if required.

The file `wlink.lnk` is a special linker directive file that is automatically processed by the Open Watcom Linker before processing any other directives. On a DOS, OS/2, or Windows-hosted system, this file must be located in one of the paths specified in the **PATH** environment variable. On a QNX-hosted system, this file should be located in the `/etc` directory. A default version of this file is located in the `\watcom\binw` directory on DOS-hosted systems, the `\watcom\binp` directory on OS/2-hosted systems, the `/etc` directory on QNX-hosted systems, and the `\watcom\binnt` directory on Windows 95 or Windows NT-hosted systems. Note that the file `wlink.lnk` includes the file `wlsystem.lnk` which is located in the `\watcom\binw` directory on DOS, OS/2, or Windows-hosted systems and the `/etc` directory on QNX-hosted systems.

The files `wlink.lnk` and `wlsystem.lnk` reference the **WATCOM** environment variable which must be set to the directory in which you installed your software.

14.5 The # Directive

The "#" directive is used to mark the start of a comment. All text from the "#" character to the end of the line is considered a comment. The format of the "#" directive is as follows.

 # comment

where ***description:***

comment is any sequence of characters.

The following directive file illustrates the use of comments.

```
file main, trigtest

# Use my own version of "sin" instead of the
# library version.

file mysin
library /math/trig
```

14.6 The CVPACK Option

This option is only meaningful when generating Microsoft CodeView debugging information. This option causes the linker to automatically run the Open Watcom CodeView 4 Symbolic Debugging Information Compactor, CVPACK, on the executable that it has created. This is necessary to get the CodeView debugging information into a state where the Microsoft CodeView debugger will accept it.

The format of the "CVPACK" option (short form "CVP") is as follows.

OPTION CVPACK

For more information on generating CodeView debugging information into the executable, see the section entitled "The DEBUG Directive" on page 278

14.7 The DEBUG Directive

The "DEBUG" directive is used to tell the Open Watcom Linker to generate debugging information in the executable file. This extra information in the executable file is used by the Open Watcom Debugger. The format of the "DEBUG" directive (short form "D") is as follows.

```
    DEBUG dbtype [dblist] |
    DEBUG [dblist]

    dbtype ::= DWARF | WATCOM | CODEVIEW | NOVELL
    dblist ::= [db_option{,db_option}]
    db_option ::= LINES | TYPES | LOCALS | ALL

DEBUG NOVELL only
    db_option ::= ONLYEXPORTS | REFERENCED
```

The Open Watcom Linker supports four types of debugging information, "DWARF" (the default), "WATCOM", "CODEVIEW", or "NOVELL".

DWARF (short form "D") specifies that all object files contain DWARF format debugging information and that the executable file will contain DWARF debugging information.

This debugging format is assumed by default when none is specified.

WATCOM (short form "W") specifies that all object files contain Watcom format debugging information and that the executable file will contain Watcom debugging information. This format permits the selection of specific classes of debugging information (*dblist*) which are described below.

CODEVIEW

(short form "C") specifies that all object files contain CodeView (CV4) format debugging information and that the executable file will contain CodeView debugging information.

It will be necessary to run the Microsoft Debugging Information Compactor, CVPACK, on the executable that it has created. For information on requesting the linker to automatically run CVPACK, see the section entitled "The CVPACK Option" on page 277 Alternatively, you can run CVPACK from the command line.

NOVELL (short form "N") specifies a form of global symbol information that can only be processed by the NetWare debugger.

For the Watcom debugging information format, we can be selective about the types of debugging information that we include with the executable file. We can categorize the types of debugging information as follows:

- global symbol information
- line numbering information
- local symbol information
- typing information
- NetWare global symbol information

The following options can be used with the "DEBUG WATCOM" directive to control which of the above classes of debugging information is included in the executable file.

LINES (short form "LI") specifies line numbering and global symbol information.

LOCALS (short form "LO") specifies local and global symbol information.

TYPES (short form "T") specifies typing and global symbol information.

ALL (short form "A") specifies all of the above debugging information.

ONLYEXPORTS
(short form "ONL") restricts the generation of global symbol information to exported symbols. This option may only be used with Netware executable formats.

The following options can be used with the "DEBUG NOVELL" directive to control which of the above classes of debugging information is included in the executable file.

ONLYEXPORTS
(short form "ONL") restricts the generation of global symbol information to exported symbols.

REFERENCED
(short form "REF") restricts the generation of symbol information to referenced symbols only.

> ***Note:*** The position of the "DEBUG" directive is important. The level of debugging information specified in a "DEBUG" directive only applies to object files and libraries that appear in *subsequent* "FILE" or "LIBRARY" directives. For example, if "DEBUG WATCOM ALL" was the only "DEBUG" directive specified and was also the last linker directive, no debugging information would appear in the executable file.

Only global symbol information is actually produced by the Open Watcom Linker; the other three classes of debugging information are extracted from object modules and copied to the executable file. Therefore, at compile time, you must instruct the compiler to generate local symbol, line numbering and typing information in the object file so that the information can be transferred to the executable file. If you have asked the Open Watcom Linker to produce a particular class of debugging information and it appears that none is present, one of the following conditions may exist.

1. The debugging information is not present in the object files.
2. The "DEBUG" directive has been misplaced.

The following sections describe the classes of debugging information.

14.7.1 Line Numbering Information - DEBUG WATCOM LINES

The "DEBUG WATCOM LINES" option controls the processing of line numbering information. Line numbering information is the line number and address of the generated code for each line of source code in a particular module. This allows Open Watcom Debugger to perform source-level debugging. When the Open Watcom Linker encounters a "DEBUG WATCOM" directive with a "LINES" or "ALL" option, line number information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which line numbering information is requested must have been compiled with the "d1" or "d2" option.

A subsequent "DEBUG WATCOM" directive without a "LINES" or "ALL" option terminates the processing of line numbering information.

14.7.2 Local Symbol Information - DEBUG WATCOM LOCALS

The "DEBUG WATCOM LOCALS" option controls the processing of local symbol information. Local symbol information is the name and address of all symbols local to a particular module. This allows Open Watcom Debugger to locate these symbols so that you can reference local data and routines by name. When the Open Watcom Linker encounters a "DEBUG WATCOM" directive with a "LOCALS" or "ALL" option, local symbol information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which local symbol information is requested must have been compiled with the "d2" option.

A subsequent "DEBUG WATCOM" directive without a "LOCALS" or "ALL" option terminates the processing of local symbol information.

14.7.3 Typing Information - DEBUG WATCOM TYPES

The "DEBUG WATCOM TYPES" option controls the processing of typing information. Typing information includes a description of all types, structures and arrays that are defined in a module. This allows Open Watcom Debugger to display variables according to their type. When the Open Watcom Linker encounters a "DEBUG WATCOM" directive with a "TYPES" or "ALL" option, typing information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which typing information is requested must have been compiled with the "d2" option.

A subsequent "DEBUG WATCOM" directive without a "TYPES" or "ALL" option terminates the processing of typing information.

14.7.4 All Debugging Information - DEBUG WATCOM ALL

The "DEBUG WATCOM ALL" option specifies that "LINES", "LOCALS", and "TYPES" options are requested. The "LINES" option controls the processing of line numbering information. The "LOCALS" option controls the processing of local symbol information. The "TYPES" option controls the processing of typing information. Each of these options is described in a previous section. A subsequent "DEBUG WATCOM " directive without an "ALL" option discontinues those options which are not specified in the list of debug options.

14.7.5 Global Symbol Information

Global symbol information consists of all the global symbols in your program and their address. This allows Open Watcom Debugger to locate these symbols so that you can reference global data and routines by name. When the Open Watcom Linker encounters a "DEBUG" directive, global symbol information for all the global symbols appearing in your program is placed in the executable file.

14.7.6 Global Symbols for the NetWare Debugger - DEBUG NOVELL

The NetWare operating system has a built-in debugger that can be used to debug programs. When "DEBUG NOVELL" is specified, the Open Watcom Linker will generate global symbol information that can be used by the NetWare debugger. Note that any line numbering, local symbol, and typing information generated in the executable file will not be recognized by the NetWare debugger. Also, ***wstrip*** cannot be used to remove this form of global symbol information from the executable file.

14.7.7 The ONLYEXPORTS Debugging Option

The "ONLYEXPORTS" option (short form "ONL") restricts the generation of global symbol information to exported symbols (symbols appearing in an "EXPORT" directive). If "DEBUG WATCOM ONLYEXPORTS" is specified, Open Watcom Debugger global symbol information is generated only for exported symbols. If "DEBUG NOVELL ONLYEXPORTS" is specified, NetWare global symbol information is generated only for exported symbols.

14.7.8 Using DEBUG Directives

Consider the following directive file.

```
debug watcom all
file module1
debug watcom lines
file module2, module3
debug watcom
library mylib
```

It specifies that the following debugging information is to be generated in the executable file.

1. global symbol information for your program

2. line numbering, typing and local symbol information for the following object files:

   ```
   module1.o
   ```

3. line numbering information for the following object files:

   ```
   module2.o
   module3.o
   ```

Note that if the "DEBUG WATCOM" directive before the "LIBRARY" directive is not specified, line numbering information for all object modules extracted from the library "mylib.lib" would be generated in the executable file provided the object modules extracted from the library have line numbering information present.

> ***Note:*** A "DEBUG WATCOM" directive with no option suppresses the processing of line numbering, local symbol and typing information for all subsequent object modules.

Debugging information can use a significant amount of disk space. As shown in the above example, you can select only the class of debugging information you want and for those modules you wish to debug. In this way, the amount of debugging information in the executable file is minimized and hence the amount of disk space used by the executable file is kept to a minimum.

As you can see from the above example, the position of the "DEBUG WATCOM" directive is important when describing the debugging information that is to appear in the executable file.

> ***Note:*** If you want all classes of debugging information for all files to appear in the executable file you must specify "DEBUG WATCOM ALL" before any "FILE" and "LIBRARY" directives.

14.7.9 Removing Debugging Information from an Executable File

A utility called ***wstrip*** has been provided which takes as input an executable file and removes the debugging information placed in the executable file by the Open Watcom Linker. Note that global symbol information generated using "DEBUG NOVELL" cannot be removed by ***wstrip.***

For more information on this utility, see the chapter entitled "The Open Watcom Strip Utility" in the ***Open Watcom C/C++ Tools User's Guide*** or ***Open Watcom FORTRAN 77 Tools User's Guide.***

14.8 The DISABLE Directive

The "DISABLE" directive is used to disable the display of linker messages.

The Open Watcom Linker issues three classes of messages; fatal errors, errors and warnings. Each message has a 4-digit number associated with it. Fatal messages start with the digit 3, error messages start with the digit 2, and warning messages start with the digit 1. It is possible for a message to be issued as a warning or an error.

If a fatal error occurs, the linker will terminate immediately and no executable file will be generated.

If an error occurs, the linker will continue to execute so that all possible errors are issued. However, no executable file will be generated since these errors do not permit a proper executable file to be generated.

If a warning occurs, the linker will continue to execute. A warning message is usually informational and does not prevent the creation of a proper executable file. However, all warnings should eventually be corrected.

Note that the behaviour of the linker does not change when a message is disabled. For example, if a message that normally terminates the linker is disabled, the linker will still terminate but the message describing the reason for the termination will not be displayed. For this reason, you should only disable messages that are warnings.

The linker will ignore the severity of the message number. For example, some messages can be displayed as errors or warnings. It is not possible to disable the message when it is issued as a warning and display the message when it is issued as an error. In general, do not specify the severity of the message when specifying a message number.

The format of the "DISABLE" directive (short form "DISA") is as follows.

```
DISABLE msg_num{, msg_num}
```

where ***description:***

msg_num is a message number. See the chapter entitled "Open Watcom Linker Diagnostic Messages" on page 363 for a list of messages and their corresponding numbers.

The following "DISABLE" directive will disable message 28 (an undefined symbol has been referenced).

```
disable 28
```

14.9 The DOSSEG Option

The "DOSSEG" option tells the Open Watcom Linker to order segments in a special way. The format of the "DOSSEG" option (short form "D") is as follows.

```
OPTION DOSSEG
```

When the "DOSSEG" option is specified, segments will be ordered in the following way.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"
6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

When using Open Watcom run-time libraries, it is not necessary to specify the "DOSSEG" option. One of the object files in the Open Watcom run-time libraries contains a special record that specifies the "DOSSEG" option.

If no "DOSSEG" option is specified, segments are ordered in the order they are encountered by the Open Watcom Linker.

When the "DOSSEG" option is specified, the Open Watcom Linker defines two special variables. `_edata` defines the start of the "BSS" class of segments and `_end` defines the end of the "BSS" class of segments. Your program must not redefine these symbols.

14.10 The ELIMINATE Option

The "ELIMINATE" option can be used to enable dead code elimination. Dead code elimination is a process the linker uses to remove unreferenced segments from the application. The linker will only remove segments that contain code; unreferenced data segments will not be removed.

The format of the "ELIMINATE" option (short form "EL") is as follows.

```
OPTION ELIMINATE
```

Linking C/C++ Applications

Typically, a module of C/C++ code contains a number of functions. When this module is compiled, all functions will be placed in the same code segment. The chances of each function in the module being unreferenced are remote and the usefulness of the "ELIMINATE" option is greatly reduced.

In order to maximize the effect of the "ELIMINATE" option, the "zm" compiler option is available to tell the Open Watcom C/C++ compiler to place each function in its own code segment. This allows the linker to remove unreferenced functions from modules that contain many functions.

Note, that if a function is referenced by data, as in a jump table, the linker will not be able to eliminate the code for the function even if the data that references it is unreferenced.

Linking FORTRAN 77 Applications

The Open Watcom FORTRAN 77 compiler always places each function and subroutine in its own code segment, even if they are contained in the same module. Therefore when linking with the "ELIMINATE" option the linker will be able to eliminate code on a function/subroutine basis.

14.11 The ENDLINK Directive

The "ENDLINK" directive is used to indicate the end of a new set of linker commands that are to be processed after the current set of commands has been processed. The format of the "ENDLINK" directive (short form "ENDL") is as follows.

```
ENDLINK
```

The "STARTLINK" directive, described in "The STARTLINK Directive" on page 346, is used to indicate the start of the set of commands.

14.12 The FARCALLS Option

The "FARCALLS" option tells the Open Watcom Linker to optimize Far Calls. This is the default setting for Open Watcom Linker The format of the "FARCALLS" option (short form "FAR") is as follows.

```
OPTION FARCALLS
```

The "NOFARCALLS" option turns off Far Calls optimization. The format of the "NOFARCALLS" option (short form "NOFAR") is as follows.

```
OPTION NOFARCALLS
```

You can specify the "NOFARCALLS" option in the default directive files `wlink.lnk` or `wlsystem.lnk` if required.

The file `wlink.lnk` is a special linker directive file that is automatically processed by the Open Watcom Linker before processing any other directives. On a DOS, OS/2, or Windows-hosted system, this file must be located in one of the paths specified in the **PATH** environment variable. On a QNX-hosted system, this file should be located in the `/etc` directory. A default version of this file is located in the `\watcom\binw` directory on DOS-hosted systems, the `\watcom\binp` directory on OS/2-hosted systems, the `/etc` directory on QNX-hosted systems, and the `\watcom\binnt` directory on Windows 95 or Windows NT-hosted systems. Note that the file `wlink.lnk` includes the file `wlsystem.lnk` which is located in the `\watcom\binw` directory on DOS, OS/2, or Windows-hosted systems and the `/etc` directory on QNX-hosted systems.

The files `wlink.lnk` and `wlsystem.lnk` reference the **WATCOM** environment variable which must be set to the directory in which you installed your software.

14.13 The FILE Directive

The "FILE" directive is used to specify the object files and library modules that the Open Watcom Linker is to process. The format of the "FILE" directive (short form "F") is as follows.

```
FILE obj_spec{,obj_spec}

obj_spec ::= obj_file[(obj_module)]
           | library_file[(obj_module)]
```

where ***description:***

obj_file is a file specification for the name of an object file. If no file extension is specified, a file extension of "o" is assumed.

library_file is a file specification for the name of a library file. Note that the file extension of the library file (usually "lib") must be specified; otherwise an object file will be assumed. When a library file is specified, all object files in the library are included (whether required or not).

obj_module is the name of an object module defined in an object or library file.

Consider the following example.

Example:

```
wlink system my_os f /math/sin, mycos
```

The Open Watcom Linker is instructed to process the following object files:

```
/math/sin.o
mycos.o
```

The object file "mycos.o" is located in the current directory since no path was specified.

More than one "FILE" directive may be used. The following example is equivalent to the preceding one.

Example:
```
wlink system my_os f /math/sin f mycos
```

Thus, other directives may be placed between lists of object files.

The "FILE" directive can also specify object modules from a library file or object file. Consider the following example.

Example:
```
wlink system my_os f /math/math.lib(sin)
```

The Open Watcom Linker is instructed to process the object module "sin" contained in the library file "math.lib" in the directory "/math".

In the following example, the Open Watcom Linker will process the object module "sin" contained in the object file "math.o" in the directory "/math".

Example:
```
wlink system my_os f /math/math(sin)
```

In the following example, the Open Watcom Linker will include all object modules contained in the library file "math.lib" in the directory "/math".

Example:
```
wlink system my_os f /math/math.lib
```

14.14 The FILLCHAR Option

The "FILLCHAR" option (short form "FILL") specifies the byte value used to fill gaps in the output image.

OPTION FILLCHAR=n

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

n specifies the value to be used in blank areas of the output image. The value must be in the range of 0 to 255, inclusive.

This option is most useful for raw binary output that will be programmed into an (E)EPROM where a value of 255 (0xff) is preferred. The default value of *n* is zero.

14.15 The FORMAT Directive

The "FORMAT" directive is used to specify the format of the executable file that the Open Watcom Linker is to generate. The format of the "FORMAT" directive (short form "FORM") is as follows.

```
FORMAT form

form ::= DOS [COM]
         | WINDOWS [win_dll] [MEMORY] [FONT]
         | WINDOWS VXD [DYNAMIC]
         | WINDOWS NT [TNT] [dll_attrs]
         | OS2 [os2_type] [dll_attrs | os2_attrs]
         | PHARLAP [EXTENDED | REX | SEGMENTED]
         | NOVELL [NLM | LAN | DSK | NAM | 'number'] 'description'
         | QNX [FLAT]
         | ELF [DLL]

win_dll ::= DLL [INITGLOBAL | INITINSTANCE]

dll_attrs ::= DLL [INITGLOBAL | INITINSTANCE]
               [TERMINSTANCE | TERMGLOBAL]

os2_type ::= FLAT | LE | LX

os2_attrs ::= PM | PMCOMPATIBLE | FULLSCREEN
               | PHYSDEVICE | VIRTDEVICE
```

where ***description:***

DOS (short form "D") tells the Open Watcom Linker to generate a DOS "EXE" file.

The name of the executable file will have extension "exe". If "COM" is specified, a DOS "COM" file will be generated in which case the name of the executable file will have extension "com". Note that these default extensions can be overridden by using the "NAME" directive to name the executable file.

Not all programs can be generated in the "COM" format. The following rules must be followed.

1. The program must consist of only one physical segment. This implies that the size of the program (code and data) must be less than 64k.

2. The program must not contain any segment relocation. A warning message will be issued by the Open Watcom Linker each time a segment relocation is encountered.

A DOS "COM" file cannot contain debugging information. If you wish to debug a DOS "COM" file, you must use the "SYMFILE" option to instruct the Open Watcom Linker to place the debugging information in a separate file.

WINDOWS tells the Open Watcom Linker to generate a Win16 (16-bit Windows) executable file.

The name of the executable file will have extension "exe". If "DLL" (short form "DL") is specified, a Dynamic Link Library will be generated; the name of the executable file will also have extension "exe". Note that these default extensions can be overridden by using the "NAME" directive to name the executable file.

Specifying "INITGLOBAL" (short form "INITG") will cause Windows to call an initialization routine the first time the Dynamic Link Library is loaded. The "INITGLOBAL" option should be used with "OPTION ONEAUTODATA" (the default for Dynamic Link Libraries). If the "INITGLOBAL" option is used with "OPTION MANYAUTODATA", the initialization code will be called once for the first data segment allocated but not for subsequent allocations (this is generally not desirable behaviour and will likely cause a program fault).

Specifying "INITINSTANCE" (short form "INITI") will cause Windows to call an initialization routine each time the Dynamic Link Library is used by a process. The "INITINSTANCE" option should be used with "OPTION MANYAUTODATA" (the default for executable programs).

In either case, the initialization routine is defined by the start address. If neither "INITGLOBAL" or "INITINSTANCE" is specified, "INITGLOBAL" is assumed.

Specifying "MEMORY" (short form "MEM") indicates that the application will run in standard or enhanced mode. If Windows 3.0 is running in standard and enhanced mode, and "MEMORY" is not specified, a warning message will be issued. The "MEMORY" specification was used in the transition from Windows 2.0 to Windows 3.0. The "MEMORY" specification is ignored in Windows 3.1 or later.

Specifying "FONT" (short form "FO") indicates that the proportional-spaced system font can be used. Otherwise, the old-style mono-spaced system font will be used. The "FONT" specification was used in the transition from Windows 2.0 to Windows 3.0. The "FONT" specification is ignored in Windows 3.1 or later.

WINDOWS VXD tells the Open Watcom Linker to generate a Windows VxD file (Virtual Device Driver).

The name of the file will have extension "386". Note that this default extension can be overridden by using the "NAME" directive to name the driver file.

Specifying "DYNAMIC" (short form "DYN") , dynamicaly loadable driver will be generated (only for Windows 3.11 or 9x). By default the Open Watcom Linker generate staticaly loadable driver (for Windows 3.x or 9x).

WINDOWS NT tells the Open Watcom Linker to generate a Win32 executable file ("PE" format).

If "TNT" is specified, an executable for the Phar Lap TNT DOS extender is created. A "PL" format (rather than "PE") executable is created so that the Phar Lap TNT DOS extender will always run the application (including under Windows NT).

If "DLL" (short form "DL") is specified, a Dynamic Link Library will be generated in which case the name of the executable file will have extension "dll". Note that these default extensions can be overridden by using the "NAME" directive to name the executable file.

Specifying "INITGLOBAL" (short form "INITG") will cause the initialization routine to be called the first time the Dynamic Link Library is loaded.

Specifying "INITINSTANCE" (short form "INITI") will cause the initialization routine to be called each time the Dynamic Link Library is referenced by a process.

In either case, the initialization routine is defined by the start address. If neither "INITGLOBAL" or "INITINSTANCE" is specified, "INITGLOBAL" is assumed.

It is also possible to specify whether the initialization routine is to be called at DLL termination or not. Specifying "TERMGLOBAL" (short form "TERMG") will cause the initialization routine to be called when the last instance of the Dynamic Link Library is terminated. Specifying "TERMINSTANCE" (short

form "TERMI") will cause the initialization routine to be called each time an instance of the Dynamic Link Library is terminated. Note that the initialization routine is passed an argument indicating whether it is being called during DLL initialization or DLL termination. If "INITINSTANCE" is used and no termination option is specified, "TERMINSTANCE" is assumed. If "INITGLOBAL" is used and no termination option is specified, "TERMGLOBAL" is assumed.

OS2 tells the Open Watcom Linker to generate an OS/2 executable file format.

The name of the executable file will have extension "exe". If "LE" is specified, an early form of the OS/2 32-bit linear executable will be generated. This executable file format is required by CauseWay DOS extender, Tenberry Software's DOS/4G and DOS/4GW DOS extenders, and similar products.

In order to improve load time and minimize the size of the executable file, the OS/2 32-bit linear executable file format was changed. If "LX" or "FLAT" (short form "FL") is specified, the new form of the OS/2 32-bit linear executable will be generated. This executable file format is required by the FlashTek DOS extender and 32-bit OS/2 executables.

If "FLAT", "LX" or "LE" is not specified, an OS/2 16-bit executable will be generated.

If "DLL" (short form "DL") is specified, a Dynamic Link Library will be generated in which case the name of the executable file will have extension "dll". Note that these default extensions can be overridden by using the "NAME" directive to name the executable file.

Specifying "INITGLOBAL" (short form "INITG") will cause the initialization routine to be called the first time the Dynamic Link Library is loaded. The "INITGLOBAL" option should be used with "OPTION ONEAUTODATA" (the default for Dynamic Link Libraries). If the "INITGLOBAL" option is used with "OPTION MANYAUTODATA", the initialization code will be called once for the first data segment allocated but not for subsequent allocations (this is generally not desirable behaviour and will likely cause a program fault).

Specifying "INITINSTANCE" (short form "INITI") will cause the initialization routine to be called each time the Dynamic Link Library is referenced by a process. The "INITINSTANCE" option should be used with "OPTION MANYAUTODATA" (the default for executable programs).

In either case, the initialization routine is defined by the start address. If neither "INITGLOBAL" or "INITINSTANCE" is specified, "INITGLOBAL" is assumed.

For OS/2 32-bit linear executable files, it is also possible to specify whether the initialization routine is to be called at DLL termination or not. Specifying "TERMGLOBAL" (short form "TERMG") will cause the initialization routine to be called when the last instance of the Dynamic Link Library is terminated. Specifying "TERMINSTANCE" (short form "TERMI") will cause the initialization routine to be called each time an instance of the Dynamic Link Library is terminated. Note that the initialization routine is passed an argument indicating whether it is being called during DLL initialization or DLL termination. If "INITINSTANCE" is used and no termination option is specified, "TERMINSTANCE" is assumed. If "INITGLOBAL" is used and no termination option is specified, "TERMGLOBAL" is assumed.

If "PM" is specified, a Presentation Manager application will be created. The application uses the API provided by the Presentation Manager and must be executed in the Presentation Manager environment.

lf "PMCOMPATIBLE" (short form "PMC") is specified, an application compatible with Presentation Manager will be created. The application can run inside the Presentation Manager or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications. This is the default.

If "FULLSCREEN" (short form "FULL") is specified, an OS/2 full screen application will be created. The application will run in a separate screen group from the Presentation Manager.

If "PHYSDEVICE" (short form "PHYS") is specified, the executable file is marked as a physical device driver.

If "VIRTDEVICE" (short form "VIRT") is specified, the executable file is marked as a virtual device driver.

PHARLAP (short form "PHAR") tells the Open Watcom Linker to generate an executable file that will run under Phar Lap’s 386|DOS-Extender.

There are 4 forms of executable files: simple, extended, relocatable and segmented. If "EXTENDED" (short form "EXT") is specified, an extended form of the executable file with file extension "exp" will be generated. If "REX" is specified, a relocatable executable file with file extension "rex" will be

generated. If "SEGMENTED" (short form "SEG") is specified, a segmented executable file with file extension "exp" will be generated. If neither "EXTENDED", "REX" or "SEGMENTED" is specified, a simple executable file with file extension "exp" will be generated. Note that the default file extensions can be overridden by using the "NAME" directive to name the executable file.

The simple form is for flat model 386 applications. It is the only format that can be loaded by earlier versions of 386|DOS-Extender (earlier than 1.2).

The extended form is used for flat model applications that have been linked in a way which requires a method of specifying more information for 386|DOS-Extender than possible with the simple form.

The relocatable form is similar to the simple form. Unique to the relocatable form is an offset relocation table. This allows the loader to load the program at any location it chooses.

The segmented form is used for embedded system applications like Intel RMX. These executables cannot be loaded by 386|DOS-Extender.

A simple form of the executable file is generated in all but the following cases.

1. "EXTENDED" is specified in the "FORMAT" directive.

2. The "RUNTIME" directive is specified. Options specified by the "RUNTIME" directive can only be specified in the extended form of the executable file.

3. The "OFFSET" option is specified. The value specified in the "OFFSET" option can only be specified in the extended form of the executable file.

4. "REX" is specified in the "FORMAT" directive. In this case, the relocatable form will be generated. You must not specify the "RUNTIME" directive or the "OFFSET" option when generating the relocatable form.

5. "SEGMENTED" is specified in the "FORMAT" directive. In this case, the segmented form will be generated.

NOVELL (short form "NOV") tells the Open Watcom Linker to generate a NetWare executable file, more commonly called a NetWare Loadable Module (NLM).

NLMs are further classified according to their function. The executable file will have a file extension that depends on the class of the NLM being generated. The following describes the classification of NLMs.

LAN instructs the Open Watcom Linker to generate a LAN driver. A LAN driver is a device driver for Local Area Network hardware. A file extension of "lan" is used for the name of the executable file.

DSK instructs the Open Watcom Linker to generate a disk driver. A file extension of "dsk" is used for the name of the executable file.

NAM instructs the Open Watcom Linker to generate a file system name-space support module. A file extension of "nam" is used for the name of the executable file.

MSL instructs the Open Watcom Linker to generate a Mirrored Server Link module. The default file extension is "msl"

CDM instructs the Open Watcom Linker to generate a Custom Device module. The default file extension is "cdm"

HAM instructs the Open Watcom Linker to generate a Host Adapter module. The default file extension is "ham"

NLM instructs the Open Watcom Linker to generate a utility or server application. This is the default. A file extension of "nlm" is used for the name of the executable file.

’number’ instructs the Open Watcom Linker to generate a specific type of NLM using ’number’. This is a 32 bit value that corresponds to Novell allocated NLM types.

These are the current defined values:

0 Specifies a standard NLM (default extension .NLM)

1 Specifies a disk driver module (default extension .DSK)

2 Specifies a namespace driver module (default extension .NAM)

3 Specifies a LAN driver module (default extension .LAN)

4	Specifies a utility NLM (default extension .NLM)
5	Specifies a Mirrored Server Link module (default .MSL)
6	Specifies an Operating System module (default .NLM)
7	Specifies a Page High OS module (default .NLM)
8	Specifies a Host Adapter module (default .HAM)
9	Specifies a Custom Device module (default .CDM)
10	Reserved for Novell usage
11	Reserved for Novell usage
12	Specifies a Ghost module (default .NLM)
13	Specifies an SMP driver module (default .NLM)
14	Specifies a NIOS module (default .NLM)
15	Specifies a CIOS CAD type module (default .NLM)
16	Specifies a CIOS CLS type module (default .NLM)
21	Reserved for Novell NICI usage
22	Reserved for Novell NICI usage
23	Reserved for Novell NICI usage
24	Reserved for Novell NICI usage
25	Reserved for Novell NICI usage
26	Reserved for Novell NICI usage
27	Reserved for Novell NICI usage
28	Reserved for Novell NICI usage

description is a textual description of the program being linked.

QNX tells the Open Watcom Linker to generate a QNX executable file.

If "FLAT" (short form "FL") is specified, a 32-bit flat executable file is generated.

Under QNX, no file extension is added to the executable file name.

Under other operating systems, the name of the executable file will have the extension "qnx". Note that this default extension can be overridden by using the "NAME" directive to name the executable file.

For more information on QNX executable file formats, see the chapter entitled "The QNX Executable File Format" on page 359.

ELF tells the Open Watcom Linker to generate an ELF format executable file.

ELF format DLLs can also be created.

If no "FORMAT" directive is specified, the executable file format will be selected for each of the following host systems in the way described.

DOS If 16-bit object files are encountered, a 16-bit DOS executable will be created. If 32-bit object files are encountered, a 32-bit DOS/4G executable will be created.

OS/2 If 16-bit object files are encountered, a 16-bit OS/2 executable will be created. If 32-bit object files are encountered, a 32-bit OS/2 executable will be created.

QNX If 16-bit object files are encountered, a 16-bit QNX executable will be created. If 32-bit object files are encountered, a 32-bit QNX executable will be created.

Windows NT If 16-bit object files are encountered, a 16-bit Windows executable will be created. If 32-bit object files are encountered, a 32-bit Win32 executable will be created.

Windows 95 If 16-bit object files are encountered, a 16-bit Windows executable will be created. If 32-bit object files are encountered, a 32-bit Win32 executable will be created.

14.16 The @ Directive

The "@" directive instructs the Open Watcom Linker to process directives from an alternate source. The format of the "@" directive is as follows.

> ***@directive_var***
> ***or***
> ***@directive_file***

where ***description:***

directive_var is the name of an environment variable. The directives specified by the value of `directive_var` will be processed.

directive_file is a file specification for the name of a linker directive file. A file extension of "lnk" is assumed if no file extension is specified.

The environment variable approach to specifying linker directives allows you to specify commonly used directives without having to specify them each time you invoke the Open Watcom Linker. If the environment variable "wlink" is set as in the following example,

```
export wlink=debug watcom all option map, verbose library
math
wlink @wlink
```

then each time the Open Watcom Linker is invoked, full debugging information will be generated, a verbose map file will be created, and the library file "math.lib" will be searched for undefined references.

A linker directive file is useful, for example, when the linker input consists of a large number of object files and you do not want to type their names on the command line each time you link your program. Note that a linker directive file can also include other linker directive files.

Let the file "memos.lnk" be a directive file containing the following lines.

```
system my_os
name memos
file memos
file actions
file read
file msg
file prompt
file memmgr
library /termio/screen
library /termio/keyboard
```

Win16 only: We must also use the "EXPORT" directive to define the window function. This is done using the following directive.

```
export window_function
```

Consider the following example.

Example:

```
wlink @memos
```

The Open Watcom Linker is instructed to process the contents of the directive file "memos.lnk". The executable image file will be called "memos.exe". The following object files will be loaded from the current directory.

```
memos.o
actions.o
read.o
msg.o
prompt.o
memmgr.o
```

If any unresolved symbol references remain after all object files have been processed, the library files "screen.lib" and "keyboard.lib" in the directory "/termio" will be searched (in the order listed).

Notes:

1. In the above example, we did not provide the file extension when the directive file was specified. The Open Watcom Linker assumes a file extension of "lnk" if none is present.

2. It is not necessary to list each object file and library with a separate directive. The following linker directive file is equivalent.

```
system my_os
name memos
file memos,actions,read,msg,prompt,memmgr
library /termio/screen,/termio/keyboard
```

However, if you want to selectively specify what debugging information should be included, the first style of directive file will be easier to use. This is illustrated in the following sample directive file.

```
system my_os
name memos
debug watcom lines
file memos
debug watcom all
file actions
debug watcom lines
file read
file msg
file prompt
file memmgr
debug watcom
library /termio/screen
library /termio/keyboard
```

3. Information for a particular directive can span directive files. This is illustrated in the following sample directive file.

```
system my_os
file memos, actions, read, msg, prompt, memmgr
file @dbgfiles
library /termio/screen
library /termio/keyboard
```

The directive file "dbgfiles.lnk" contains, for example, those object files that are used for debugging purposes.

14.17 The LANGUAGE Directive

The "LANGUAGE" directive is used to specify the language in which strings in the Open Watcom Linker directives are specified. The format of the "LANGUAGE" directive (short form "LANG") is as follows.

```
LANGUAGE lang

lang ::= JAPANESE | CHINESE | KOREAN
```

JAPANESE (short form "JA") specifies that strings are to be handled as if they contained characters from the Japanese Double-Byte Character Set (DBCS).

CHINESE (short form "CH") specifies that strings are to be handled as if they contained characters from the Chinese Double-Byte Character Set (DBCS).

KOREAN (short form "KO") specifies that strings are to be handled as if they contained characters from the Korean Double-Byte Character Set (DBCS).

14.18 The LIBFILE Directive

The "LIBFILE" directive is used to specify the object files that the Open Watcom Linker is to process. The format of the "LIBFILE" directive (short form "LIBF") is as follows.

> ***LIBFILE obj_spec{,obj_spec}***
>
> ***obj_spec ::= obj_file | library_file***

where ***description:***

obj_file is a file specification for the name of an object file. If no file extension is specified, a file extension of "o" is assumed.

library_file is a file specification for the name of a library file. Note that the file extension of the library file (usually "lib") must be specified; otherwise an object file will be assumed. When a library file is specified, all object files in the library are included (whether required or not).

The difference between the "LIBFILE" directive and the "FILE" directive is as follows.

1. When searching for an object or library file specified in a "LIBFILE" directive, the current working directory will be searched first, followed by the paths specified in the "LIBPATH" directive, and finally the paths specified in the "LIB" environment variable. Note that if the object or library file name contains a path, only the specified path will be searched.
2. Object or library file names specified in a "LIBFILE" directive will not be used to create the name of the executable file when no "NAME" directive is specified.

Essentially, object files that appear in "LIBFILE" directives are viewed as components of a library that have not been explicitly placed in a library file.

Consider the following linker directive file.

```
libpath /libs
libfile mystart
path /objs
file file1, file2
```

The Open Watcom Linker is instructed to process the following object files:

```
/libs/mystart.o
/objs/file1.o
/objs/file2.o
```

Note that the executable file will have file name "file1" and not "mystart".

14.19 The LIBPATH Directive

The "LIBPATH" directive is used to specify the directories that are to be searched for library files appearing in subsequent "LIBRARY" directives and object files appearing in subsequent "LIBFILE" directives. The format of the "LIBPATH" directive (short form "LIBP") is as follows.

LIBPATH [path_name{:path_name}]

where ***description:***

path_name is a path name.

Consider a directive file containing the following linker directives.

```
file test
libpath /math
library trig
libfile newsin
```

First, the Open Watcom Linker will process the object file "test.o" from the current working directory. The object file "newsin.o" will then be processed, searching the current working directory first. If "newsin.o" is not in the current working directory, the "/math" directory will be searched. If any unresolved references remain after processing the object files, the library file "trig.lib" will be searched. If the file "trig.lib" does not exist in the current working directory, the "/math" directory will be searched.

It is also possible to specify a list of paths in a "LIBPATH" directive. Consider the following example.

```
libpath /newmath:/math
library trig
```

When processing undefined references, the Open Watcom Linker will attempt to process the library file "trig.lib" in the current working directory. If "trig.lib" does not exist in the current working directory, the "/newmath" directory will be searched. If "trig.lib" does not exist in the "/newmath" directory, the "/math" directory will be searched.

If the name of a library file appearing in a "LIBRARY" directive or the the name of an object file appearing in a "LIBFILE" directive contains a path specification, only the specified path will be searched.

Note that

```
libpath path1
libpath path2
```

is equivalent to the following.

```
libpath path2:path1
```

14.20 The LIBRARY Directive

The "LIBRARY" directive is used to specify the library files to be searched when unresolved symbols remain after processing all specified input object files. The format of the "LIBRARY" directive (short form "L") is as follows.

```
LIBRARY library_file{,library_file}
```

where	*description:*
library_file	is a file specification for the name of a library file. If no file extension is specified, a file extension of "lib" is assumed.

Consider the following example.

Example:
```
wlink system my_os file trig lib /math/trig, /cmplx/trig
```

The Open Watcom Linker is instructed to process the following object file:

```
trig.o
```

If any unresolved symbol references remain after all object files have been processed, the following library files will be searched:

```
/math/trig.lib
/cmplx/trig.lib
```

More than one "LIBRARY" directive may be used. The following example is equivalent to the preceding one.

Example:
```
wlink system my_os f trig lib /math/trig lib /cmplx/trig
```

Thus other directives may be placed between lists of library files.

14.20.1 Searching for Libraries Specified in Environment Variables

The "LIB" environment variable can be used to specify a list of paths that will be searched for library files. The "LIB" environment variable can be set using the "export" command as follows:

```
export lib=/graphics/lib:/utility
```

Consider the following "LIBRARY" directive and the above definition of the "LIB" environment variable.

```
library /mylibs/util, graph
```

If undefined symbols remain after processing all object files specified in all "FILE" directives, the Open Watcom Linker will resolve these references by searching the following libraries in the specified order.

1. the library file "/mylibs/util.lib"
2. the library file "graph.lib" in the current directory
3. the library file "/graphics/lib/graph.lib"
4. the library file "/utility/graph.lib"

Notes:

1. If a library file specified in a "LIBRARY" directive contains an absolute path specification, the Open Watcom Linker will not search any of the paths specified in the "LIB" environment string for the library file. Under QNX, an absolute path specification is one that begins the "/" character. Under all other operating systems, an absolute path specification is one that begins with a drive specification or the "\" character.

2. Once a library file has been found, no further elements of the "LIB" environment variable are searched for other libraries of the same name. That is, if the library file "/graphics/lib/graph.lib" exists, the library file "/utility/graph.lib" will not be searched even though unresolved references may remain.

14.20.2 Converting Libraries Created using Phar Lap 386|LIB

Phar Lap's librarian, 386|LIB, creates libraries whose dictionary is a different format from the one used by other librarians. For this reason, linking an application using the Open Watcom Linker with libraries created using 386|LIB will not work. Library files created using 386|LIB must be converted to the form recognized by the Open Watcom Linker. This is achieved by issuing the following wlib command.

```
wlib newlib +pharlib.lib
```

The library file "pharlib.lib" is a library created using 386|LIB. The library file "newlib.lib" will be created so that the Open Watcom Linker can now process it.

14.21 The LINEARRELOCS Option

The "LINEARRELOCS" option instructs the linker to generate offset fixups in addition to the normal segment fixups. The offset fixups allow the system to move pieces of code and data that were loaded at a particular offset within a segment to another offset within the same segment.

The format of the "LINEARRELOCS" option (short form "LI") is as follows.

```
OPTION LINEARRELOCS
```

14.22 The LONGLIVED Option

The "LONGLIVED" option specifies that the application being linked will reside in memory, or be active, for a long period of time (e.g., background tasks). The memory manager, knowing an application is "LONGLIVED", allocates memory for the application so as to reduce fragmentation.

The format of the "LONGLIVED" option (short form "LO") is as follows.

```
OPTION LONGLIVED
```

14.23 The MANGLEDNAMES Option

The "MANGLEDNAMES" option should only be used if you are developing a Open Watcom C++ application. Due to the nature of C++, the Open Watcom C++ compiler generates mangled names for symbols. A mangled name for a symbol includes the following.

1. symbol name
2. scoping information
3. typing information

This information is stored in a cryptic form with the symbol. When the linker encounters a mangled name in an object file, it formats the above information and produces this name in the map file.

If you would like the linker to produce the mangled name as it appeared in the object file, specify the "MANGLEDNAMES" option.

The format of the "MANGLEDNAMES" option (short form "MANG") is as follows.

```
OPTION MANGLEDNAMES
```

14.24 The MAP Option

The "MAP" option controls the generation of a map file. The format of the "MAP" option (short form "M") is as follows.

OPTION MAP[=map_file]

where ***description:***

map_file is a file specification for the name of the map file. If no file extension is specified, a file extension of "map" is assumed.

By default, no map file is generated. Specifying this option causes the Open Watcom Linker to generate a map file. The map file is simply a memory map of your program. That is, it specifies the relative location of all global symbols in your program. The map file also contains the size of your program.

If no file name is specified, the map file will have a default file extension of "map" and the same file name as the executable file. Note that the map file will be created in the current directory even if the executable file name specified in the "NAME" directive contains a path specification.

Alternatively, a file name can be specified. The following directive instructs the linker to generate a map file and call it "myprog.map" regardless of the name of the executable file.

```
option map=myprog
```

You can also specify a path and/or file extension when using the "MAP=" form of the "MAP" option.

14.25 The MAXERRORS Option

The "MAXERRORS" option can be used to set a limit on the number of error messages generated by the linker. Note that this does not include warning messages. When this limit is reached, the linker will issue a fatal error and terminate.

The format of the "MAXERRORS" option (short form "MAXE") is as follows.

```
OPTION MAXERRORS=n
```

where	***description:***
n	is the maximum number of error messages issued by the linker.

14.26 The MODFILE Directive

The "MODFILE" directive instructs the linker that only the specified object files have changed. The format of the "MODFILE" directive (short form "MODF") is as follows.

MODFILE obj_file{,obj_file}

where	***description:***
obj_file	is a file specification for the name of an object file. If no file extension is specified, a file extension of "o" is assumed.

This directive is used only in concert with incremental linking. This directive tells the linker that only the specified object files have changed. When this option is specified, the linker will not check the dates on any of the object files or libraries when incrementally linking.

14.27 The MODTRACE Directive

The "MODTRACE" directive instructs the Open Watcom Linker to print a list of all modules that reference the symbols defined in the specified modules. The format of the "MODTRACE" directive (short form "MODT") is as follows.

MODTRACE module_name{,module_name}

where ***description:***

module_name is the name of an object module defined in an object or library file.

The information is displayed in the map file. Consider the following example.

Example:
```
wlink system my_os op map file test lib math modt trig
```

If the module "trig" defines the symbols "sin" and "cos", the Open Watcom Linker will list, in the map file, all modules that reference the symbols "sin" and "cos".

14.28 The NAME Directive

The "NAME" directive is used to provide a name for the executable file generated by the Open Watcom Linker. The format of the "NAME" directive (short form "N") is as follows.

```
NAME exe_file
```

where ***description:***

exe_file is a file specification for the name of the executable file. Under UNIX, no file extension is appended. For all other operating systems, a file extension suitable for the current executable file format is appended if no file extension is specified.

Consider the following example.

Example:
```
wlink system my_os name myprog file test, test2, test3
```

The linker is instructed to generate an executable file called "myprog.exe" if you are running a DOS, OS/2 or Windows-hosted version of the linker. If you are running a UNIX-hosted version of the linker, an executable file called "myprog" will be generated.

Notes:

1. No file extension was given when the executable file name was specified. The linker assumes a file extension that depends on the format of the executable file being generated. If you are running a UNIX-hosted version of the linker, no file extension will be assumed. The section entitled "The FORMAT Directive" on page 295 describes the "FORMAT" directive and how the file extension is chosen for each executable file format.

2. If no "NAME" directive is present, the executable file will have the file name of the first object file processed by the linker. If the first object file processed is called "test.o" and no "NAME" directive is specified, an executable file called "test.exe" will be generated if you are running a DOS or OS/2-hosted version of the linker. If you are running a UNIX-hosted version of the linker, an executable file called "test" will be generated.

14.29 The NAMELEN Option

The "NAMELEN" option tells the Open Watcom Linker that all symbols must be uniquely identified in the number of characters specified or less. If any symbol fails to satisfy this condition, a warning message will be issued. The warning message will state that a symbol has been defined more than once.

The format of the "NAMELEN" option (short form "NAMEL") is as follows.

```
OPTION NAMELEN=n
```

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

Some computer systems, for example, require that all global symbols be uniquely identified in 8 characters. By specifying an appropriate value for the "NAMELEN" option, you can ease the task of porting your application to other computer systems.

14.30 The NODEFAULTLIBS Option

Special object module records that specify default libraries are placed in object files generated by Open Watcom compilers. These libraries reflect the memory and floating-point model that a source file was compiled for and are automatically searched by the Open Watcom Linker when unresolved symbols are detected. These libraries can exist in the current directory, in one of the paths specified in "LIBPATH" directives, or in one of the paths specified in the **LIB** environment variable.

Note that all library files that appear in a "LIBRARY" directive are searched before default libraries. The "NODEFAULTLIBS" option instructs the Open Watcom Linker to ignore default libraries. That is, only libraries appearing in a "LIBRARY" directive are searched.

The format of the "NODEFAULTLIBS" option (short form "NOD") is as follows.

```
OPTION NODEFAULTLIBS
```

14.31 The OPTION Directive

The "OPTION" directive is used to specify options to the Open Watcom Linker. The format of the "OPTION" directive (short form "OP") is as follows.

```
OPTION option{,option}
```

where ***description:***

option is any of the linker options available for the executable format that is being generated.

14.32 The OPTLIB Directive

The "OPTLIB" directive is used to specify the library files to be searched when unresolved symbols remain after processing all specified input object files. The format of the "OPTLIB" directive (no short form) is as follows.

```
OPTLIB library_file{,library_file}
```

where ***description:***

library_file is a file specification for the name of a library file. If no file extension is specified, a file extension of "lib" is assumed.

This directive is similar to the "LIBRARY" directive except that the linker will not issue a warning message if the library file cannot be found.

Consider the following example.

Example:

```
wlink system my_os file trig optlib /math/trig, /cmplx/trig
```

The Open Watcom Linker is instructed to process the following object file:

```
trig.o
```

If any unresolved symbol references remain after all object files have been processed, the following library files will be searched:

```
/math/trig.lib
/cmplx/trig.lib
```

More than one "OPTLIB" directive may be used. The following example is equivalent to the preceding one.

Example:
```
wlink system my_os f trig optlib /math/trig optlib
/cmplx/trig
```

Thus other directives may be placed between lists of library files.

14.32.1 Searching for Optional Libraries Specified in Environment Variables

The "LIB" environment variable can be used to specify a list of paths that will be searched for library files. The "LIB" environment variable can be set using the "export" command as follows:

```
export lib=/graphics/lib:/utility
```

Consider the following "OPTLIB" directive and the above definition of the "LIB" environment variable.

```
optlib /mylibs/util, graph
```

If undefined symbols remain after processing all object files specified in all "FILE" directives, the Open Watcom Linker will resolve these references by searching the following libraries in the specified order.

1. the library file "/mylibs/util.lib"
2. the library file "graph.lib" in the current directory
3. the library file "/graphics/lib/graph.lib"
4. the library file "/utility/graph.lib"

Notes:

1. If a library file specified in a "OPTLIB" directive contains an absolute path specification, the Open Watcom Linker will not search any of the paths specified in the "LIB" environment string for the library file. On UNIX platforms, an absolute path specification is one that begins the "/" character. On all other hosts, an absolute path specification is one that begins with a drive specification or the "\" character.

2. Once a library file has been found, no further elements of the "LIB" environment variable are searched for other libraries of the same name. That is, if the library file "/graphics/lib/graph.lib" exists, the library file "/utility/graph.lib" will not be searched even though unresolved references may remain.

14.33 The ORDER Directive

The "ORDER" directive is used to specify the order in which classes are placed into the output image, and the order in which segments are linked within a class. The directive can optionally also specify the starting address of a class or segment, control whether the segment appears in the output image, and facilitate copying of data from one segment to another. The "ORDER" Directive is primarily intended for embedded (ROMable) targets that do not run under an operating system, or for other special purpose applications. The format of the "ORDER" directive (short form "ORD") is as follows.

```
ORDER {CLNAME class_name [class_options]}+

class_options ::= [SEGADDR=n][OFFSET=n][copy_option][NOEMIT]{seglist}
copy_option ::= [COPY source_class_name]
seglist := {SEGMENT seg_name [SEGADDR=n][OFFSET=n][NOEMIT]}+
```

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

class_name is the name of a class defined in one or more object files. If the class is not defined in an object file, the *class_name* and all associated options are ignored. Note that the "ORDER" directive does *not* create classes or segments. Classes specified with "CLNAME" keywords will be placed in the output image in the order listed. Any classes that are not listed will be placed after the listed ones.

SEGADDR=n (short form "SEGA") specifies the segment portion of the starting address of the class or segment in the output image. It is combined with "OFFSET" to represent a unique linear address. "SEGADDR" is only valid for segmented formats. Its use in other contexts is undefined. The "HSHIFT" value affects how the segment value is converted to a linear address.

OFFSET=n (short form "OFF") specifies the offset portion of the starting address of the class or segment in the output image. It is combined with "SEGADDR" to represent a unique linear address. Offset is limited to a range of 0 to 65535 in

segmented architectures, but can be a larger value for non-segmented architectures, up to the limits of the architecture.

When "SEGADDR" and/or "OFFSET" are specified, the location counter used to generate the executable is advanced to that address. Any gaps are filled with the "FILLCHAR" value, except for HEX output format, in which case they are simply skipped. If the location counter is already beyond the specified location, an error message is generated. This would likely be the result of having specified classes or segments in incorrect order, or not providing enough room for preceding ones. Without the "SEGADDR" and "OFFSET" options, classes and segment are placed in the the executable consecutively, possibly with a small gap in between if required by the alignment specified for the class.

COPY (short form "CO") indicates that the data from the segment named *source_class_name* is to be used in this segment.

NOEMIT (short form "NOE") indicates that the data in this segment should not be placed in the executable.

SEGMENT indicates the order of segments within a class, and possibly other options associated with that segment. Segments listed are placed in the executable in the order listed. They must be part of the class just named. Any segments in that class not listed will follow the last listed segment. The segment options are a subset of the class options and conform to the same specifications.

In ROM-based applications it is often necessary to:

- Fix the program location
- Separate code and data to different fixed parts of memory
- Place a copy of initialized data in ROM (usually right after the code)
- Prevent the original of the initialized data from being written to the loadfile, since it resides in RAM and cannot be saved there.

The "ORDER" directive caters for these requirements. Classes can be placed in the executable in a specific order, with absolute addresses specified for one or more classes, and segments within a class can be forced into a specified order with absolute addresses specified for one or more of them. Initialized data can be omitted at its target address, and a copy included at a different address.

Following is a sample "ORDER" directive for an embedded target (AM186ER). The bottom 32K of memory is RAM for data. A DGROUP starting address of 0x80:0 is required. The

upper portion of memory is FLASH ROM. Code starts at address 0xD000:0. The initialized data from DGROUP is placed immediately after the code.

```
order clname BEGDATA NOEMIT segaddr=0x80 segment _NULL
segment _AFTERNULL
      clname DATA NOEMIT segment _DATA
      clname BSS
      clname STACK
      clname START segaddr=0xD000
      clname CODE segment BEGTEXT segment _TEXT
      clname ROMDATA COPY BEGDATA
      clname ROMDATAE
```

DGROUP consists of classes "BEGDATA", "DATA", "BSS", "BSS2" and "STACK". Note that these are marked "NOEMIT" (except for the BSS classes and STACK which are not initialized, and therefore have no data in them anyway) to prevent data from being placed in the loadfile at 0x80:0. The first class of DGROUP is given the fixed starting segment address of 0x80 (offset is assumed to be 0). The segments "_NULL", "_AFTERNULL" and "_DATA" will be allocated consecutively in that order, and because they are part of DGROUP, will all share the same segment portio of the address, with offsets adjusted accordingly.

The code section consists of classes "START" and "CODE". These are placed beginning at 0xD000:0. "START" contains only one segment, which will be first. It will have a CS value of 0xD000. Code has two segments, "BEGTEXT" and "_TEXT" which will be placed after "START", in that order, and packed into a single CS value of their own (perhaps 0xD001 in this example), unless they exceed 64K in size, which should not be the case if the program was compiled using the small memory model.

The classes "ROMDATA" and "ROMDATAE" were created in assembly with one segment each and no symbols or data in them. The class names can be used to identify the beginning and end of initialized data so it can be copied to RAM by the startup code.

The "COPY" option actually works at the group level, because that is the way it is generally needed. The entire data is in DGROUP. "ROMDATA" will be placed in a group of its own called "AUTO". (Note: each group mentioned in the map file under the name "AUTO" is a separate group. They are not combined or otherwise related in any way, other than they weren't explicitly created by the programmer, compiler or assembler, but rather automatically created by the linker in the course of its work.) Therefore there is a unique group associated with this class. The "COPY" option finds the group associated with "BEGDATA" and copies all the object data from there to "ROMDATA". Specifically, it places a copy of this data in the executable at the location assigned to "ROMDATA", and adjusts the length of "ROMDATA" to account for this. All symbol references to this data are to its execution address (0x80:0), not where it ended up in the executable (for instance 0xD597:0). The starting address of "ROMDATAE" is also adjusted to account for the data assigned to

"ROMDATA". That way, the program can use the symbol "ROMDATAE" to identify the end of the copy of DGROUP. It is also necessary in case more than one "COPY" class exists consecutively, or additional code or data need to follow it.

It should also be noted that the "DOSSEG" option (whether explicitly given to the linker, or passed in an object file) performs different class and segment ordering. If the "ORDER" directive is used, it overrides the "DOSSEG" option, causing it to be ignored.

14.34 The OSNAME Option

The "OSNAME" option can be used to set the name of the target operating system of the executable file generated by the linker. The format of the "OSNAME" option (short form "OSN") is as follows.

OPTION OSNAME='string'

where ***description:***

string is any sequence of characters.

The information specified by the "OSNAME" option will be displayed in the *creating a ? executable* message. This is the last line of output produced by the linker, provided the "QUIET" option is not specified. Consider the following example.

```
option osname='SuperOS'
```

The last line of output produced by the linker will be as follows.

```
creating a SuperOS executable
```

Some executable formats have a stub executable file that is run under 16-bit DOS. The message displayed by the default stub executable file will be modified when the "OSNAME" option is used. The default stub executable displays the following message:

OS/2: `this is an OS/2 executable`

Win16: `this is a Windows executable`

Win32: `this is a Windows NT executable`

If the "OSNAME" option used in the previous example was specified, the default stub executable would generate the following message.

```
this is a SuperOS executable
```

14.35 The OUTPUT Directive

The "OUTPUT" directive overrides the normal operating system specific executable format and creates either a raw binary image or an Intel Hex file. The format of the "OUTPUT" directive (short form "OUT") is as follows.

OUTPUT RAW|HEX [OFFSET=n][HSHIFT=n][STARTREC]

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

RAW specifies the output file to be a raw binary and will contain an absolute image of the executable's code and data. Default file extension is "bin".

HEX specifies the output file to contain a representation of the absolute image of the code and data using the Intel standard hex file format. Default file extension is "hex".

OFFSET=n (short form "OFF") specifies that the linear address *n* should be subtracted from all addresses being output to the executable image.

HSHIFT defines the relationship between segment values for type 02 records and linear addresses. The value *n* is the number of digits to right shift a 32-bit value containing a segment address in its upper 16 bits in order to convert it to part of a linear address. In more conventional terms, (16 - *n*) is the amount to shift a segment value left in order to convert it to part of a linear address.

STARTREC (short form "ST") specifies that a Starting Address record will be included in Intel Hex output. This option is ignored if output type is not Intel hex.

For raw binary files, the position in the file is the linear address after the offset is subtracted from it. Any gaps filled with the value specified through "OPTION FILLCHAR" (default is 0).

For hex files, the linear address (after subtracting the offset) is used to determine the output record generated. Records contain 16 bytes, unless a gap occurs prior to that in which case the record is shorter, and a new record starts after the gap. There are three types of Intel Hex records. The oldest and most widely used is HEX80, which can only deal with 16-bit addresses. For many ROM-based applications, this is enough, especially once an offset has been subtracted. For maximum versatility, all addresses less than 65536 are generated in this form.

The HEX86 standard creates a segmentation that mirrors the CPU segmentation. Type 02 records define the segment, and all subsequent addresses are based on that segment value. For addresses above 64K, This form is used. A program that understands HEX86 should assume the segment value is zero until an 02 record is encountered. This preserves backward compatibility with HEX80, and allows the automatic selection algorithm used in Open Watcom Linker to work properly.

Type 02 records are assumed to have segment values that, when shifted left four bits, form a linear address. However, this is not suitable for 24-bit segmented addressing schemes. Therefore, Open Watcom Linker uses the value specified through "OPTION HSHIFT" to determine the relationship between segments and offsets. This approach can work with any 16:16 segmented architecture regardless of the segment alignment. The default shift value is 12, representing the conventional 8086 architecture. This is not to be confused with the optional "OUTPUT HSHIFT" value discussed below.

Of course, PROM programmers or third-party tools probably were *not* designed to work with unconventional shift values, hence for cases where code for a 24-bit (or other non-standard) target needs to be programmed into a PROM or processed by a third-party tool, the "OUTPUT HSHIFT" option can be used to override the "OPTION HSHIFT" value. This would usually be of the form "OUTPUT HSHIFT=12" to restore the industry standard setting. The default for "OUTPUT HSHIFT" is to follow "OPTION HSHIFT". When neither is specified, the default "OPTION HSHIFT" value of 12 applies, providing industry standard compliance.

If the address exceeds the range of type 02 records (1 MB for HSHIFT=12 and 16 MB for HSHIFT=8), type 04 extended linear records are generated, again ensuring seamless compatibility and migration to large file sizes.

If "STARTREC" is specified for "OUTPUT HEX", the penultimate record in the file (just before the end record) will be a start address record. The value of the start address will be determined by the module start record in an object file, typically the result of an "END start" assembler directive. If the start address is less than 65536 (always for 16-bit applications, and where applicable for 32-bit applications), a type 03 record with segment and offset values will be emitted. If the start address is equal to or greater than 65536, then a type 05 linear starting address record will be generated. Note that neither of these cases depends directly on the "HSHIFT" or "OUTPUT HSIFT" settings. If HSHIFT=8, then the segment and offset values for the start symbol will be based on that number and used accordingly, but unlike other

address information in a hex file, this is not derived from a linear address and hence not converted based on the HSHIFT value.

14.36 The PATH Directive

The "PATH" directive is used to specify the directories that are to be searched for object files appearing in subsequent "FILE" directives. When the "PATH" directive is specified, the current directory will no longer be searched unless it appears in the "PATH" directive. The format of the "PATH" directive (short form "P") is as follows.

PATH path_name{:path_name}

where ***description:***

path_name is a path name.

Consider a directive file containing the following linker directives.

```
path /math
file sin
path /stats
file mean, variance
```

It instructs the Open Watcom Linker to process the following object files:

```
/math/sin.o
/stats/mean.o
/stats/variance.o
```

It is also possible to specify a list of paths in a "PATH" directive. Consider the following example.

```
path /math:/stats
file sin
```

First, the linker will attempt to load the file "/math/sin.o". If unsuccessful, the linker will attempt to load the file "/stats/sin.o".

It is possible to override the path specified in a "PATH" directive by preceding the object file name in a "FILE" directive with an absolute path specification. On UNIX platforms, an absolute path specification is one that begins the "/" character. On all other hosts, an absolute path specification is one that begins with a drive specification or the "\" character.

```
path /math
file sin
path /stats
file mean, /mydir/variance
```

The above directive file instructs the linker to process the following object files:

```
/math/sin.o
/stats/mean.o
/mydir/variance.o
```

14.37 The PRIVILEGE Option

The "PRIVILEGE" option specifies the privilege level (0, 1, 2 or 3) at which the application will run. The format of the "PRIVILEGE" option (short form "PRIV") is as follows.

OPTION PRIVILEGE=n

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

The default privilege level is 0.

14.38 The QUIET Option

The "QUIET" option tells the Open Watcom Linker to suppress all informational messages. Only warning, error and fatal messages will be issued. By default, the Open Watcom Linker issues informational messages. The format of the "QUIET" option (short form "Q") is as follows.

```
OPTION QUIET
```

14.39 The REDEFSOK Option

The "REDEFSOK" option tells the Open Watcom Linker to ignore redefined symbols and to generate an executable file anyway. By default, warning messages are displayed and an executable file is generated if redefined symbols are present.

The format of the "REDEFSOK" option (short form "RED") is as follows.

```
OPTION REDEFSOK
```

The "NOREDEFSOK" option tells the Open Watcom Linker to treat redefined symbols as an error and to not generate an executable file. By default, warning messages are displayed and an executable file is generated if redefined symbols are present.

The format of the "NOREDEFSOK" option (short form "NORED") is as follows.

```
OPTION NOREDEFSOK
```

14.40 The REFERENCE Directive

The "REFERENCE" directive is used to explicitly reference a symbol that is not referenced by any object file processed by the linker. If any symbol appearing in a "REFERENCE" directive is not resolved by the linker, an error message will be issued for that symbol specifying that the symbol is undefined.

The "REFERENCE" directive can be used to force object files from libraries to be linked with the application. Also note that a symbol appearing in a "REFERENCE" directive will not be eliminated by dead code elimination. For more information on dead code elimination, see the section entitled "The ELIMINATE Option" on page 289.

The format of the "REFERENCE" directive (short form "REF") is as follows.

REFERENCE symbol_name{, symbol_name}

where ***description:***

symbol_name is the symbol for which a reference is made.

Consider the following example.

```
reference domino
```

The symbol `domino` will be searched for. The object module that defines this symbol will be linked with the application. Note that the linker will also attempt to resolve symbols referenced by this module.

14.41 The SHOWDEAD Option

The "SHOWDEAD" option instructs the linker to list, in the map file, the symbols associated with dead code and unused C++ virtual functions that it has eliminated from the link. The format of the "SHOWDEAD" option (short form "SHO") is as follows.

```
OPTION SHOWDEAD
```

The "SHOWDEAD" option works best in concert with the "ELIMINATE" and "VFREMOVAL" options.

14.42 The SORT Directive

The "SORT" directive is used to sort the symbols in the "Memory Map" section of the map file. By default, symbols are listed on a per module basis in the order the modules were encountered by the linker. That is, a module header is displayed followed by the symbols defined by the module.

The format of the "SORT" directive (short form "SO") is as follows.

SORT [GLOBAL] [ALPHABETICAL]

If the "SORT" directive is specified without any options, as in the following example, the module headers will be displayed each followed by the list of symbols it defines sorted by address.

```
sort
```

If only the "GLOBAL" sort option (short form "GL") is specified, as in the following example, the module headers will not be displayed and all symbols will be sorted by address.

```
sort global
```

If only the "ALPHABETICAL" sort option (short form "ALP") is specified, as in the following example, the module headers will be displayed each followed by the list of symbols it defines sorted alphabetically.

```
sort alphabetical
```

If both the "GLOBAL" and "ALPHABETICAL" sort options are specified, as in the following example, the module headers will not be displayed and all symbols will be sorted alphabetically.

```
sort global alphabetical
```

If you are linking a Open Watcom C++ application, mangled names are sorted by using the base name. The base name is the name of the symbol as it appeared in the source file. See the section entitled "The MANGLEDNAMES Option" on page 317 for more information on mangled names.

14.43 The STACK Option

The "STACK" option can be used to increase the size of the stack. The format of the "STACK" option (short form "ST") is as follows.

OPTION STACK=n

where ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k|m]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024. If *m* is specified, the value is multiplied by 1024*1024.

The default stack size varies for both 16-bit and protected-mode 32-bit applications depending on the executable format. You can determine the default stack size by looking at the map file that can be generated when an application is linked ("OPTION MAP"). During execution of your program, you may get an error message indicating your stack has overflowed. If you encounter such an error, you must link your application again, this time specifying a larger stack size using the "STACK" option.

Example:

```
option stack=8192
```

14.44 The START Option

The format of the "START" option is as follows.

```
OPTION START=symbol_name
```

where ***description:***

symbol_name specifies the name of the procedure where execution begins.

For the Netware executable format, the default name of the start procedure is "_Prelude".

14.45 The STARTLINK Directive

The "STARTLINK" directive is used to indicate the start of a new set of linker commands that are to be processed after the current set of commands has been processed. The format of the "STARTLINK" directive (short form "STARTL") is as follows.

```
STARTLINK
```

The "ENDLINK" directive is used to indicate the end of the set of commands identified by the "STARTLINK" directive.

14.46 The STATICS Option

The "STATICS" option should only be used if you are developing a Open Watcom C or C++ application. The Open Watcom C and C++ compilers produce definitions for static symbols in the object file. By default, these static symbols do not appear in the map file. If you want static symbols to be displayed in the map file, use the "STATICS" option.

The format of the "STATICS" option (short form "STAT") is as follows.

```
OPTION STATICS
```

14.47 The SYMFILE Option

The "SYMFILE" option provides a method for specifying an alternate file for debugging information. The format of the "SYMFILE" option (short form "SYMF") is as follows.

```
OPTION SYMFILE[=symbol_file]
```

where ***description:***

symbol_file is a file specification for the name of the symbol file. If no file extension is specified, a file extension of "sym" is assumed.

By default, no symbol file is generated; debugging information is appended at the end of the executable file. Specifying this option causes the Open Watcom Linker to generate a symbol file. The symbol file contains the debugging information generated by the linker when the "DEBUG" directive is used. The symbol file can then be used by Open Watcom Debugger. If no debugging information is requested, no symbol file is created, regardless of the presence of the "SYMFILE" option.

If no file name is specified, the symbol file will have a default file extension of "sym" and the same path and file name as the executable file. Note that the symbol file will be placed in the same directory as the executable file.

Alternatively, a file name can be specified. The following directive instructs the linker to generate a symbol file and call it "myprog.sym" regardless of the name of the executable file.

```
option symf=myprog
```

You can also specify a path and/or file extension when using the "SYMFILE=" form of the "SYMFILE" option.

Notes:

1. This option should be used to debug a DOS "COM" executable file. A DOS "COM" executable file must not contain any additional information other than the executable information itself since DOS uses the size of the file to determine what to load.

2. This option should be used when creating a Microsoft Windows executable file. Typically, before an executable file can be executed as a Microsoft Windows application, a resource compiler takes the Windows executable file and a resource

file as input and combines them. If the executable file contains debugging information, the resource compiler will strip the debugging information from the executable file. Therefore, debugging information must not be part of the executable file created by the linker.

14.48 The SYMTRACE Directive

The "SYMTRACE" directive instructs the Open Watcom Linker to print a list of all modules that reference the specified symbols. The format of the "SYMTRACE" directive (short form "SYMT") is as follows.

```
SYMTRACE  symbol_name{,symbol_name}
```

where ***description:***

symbol_name is the name of a symbol.

The information is displayed in the map file. Consider the following example.

Example:
```
wlink system my_os op map file test lib math symt sin, cos
```

The Open Watcom Linker will list, in the map file, all modules that reference the symbols "sin" and "cos".

14.49 The SYSTEM Directive

There are three forms of the "SYSTEM" directive.

The first form of the "SYSTEM" directive (short form "SYS") is called a system definition directive. It allows you to associate a set of linker directives with a specified name called the *system name*. This set of linker directives is called a system definition block. The format of a system definition directive is as follows.

SYSTEM BEGIN system_name {directive} END

where ***description:***

system_name is a unique system name.

directive is a linker directive.

A system definition directive cannot be specified within another system definition directive.

The second form of the "SYSTEM" directive is called a system deletion directive. It allows you to remove the association of a set of linker directives with a *system name*. The format of a system deletion directive is as follows.

SYSTEM DELETE system_name

where ***description:***

system_name is a defined system name.

The third form of the "SYSTEM" directive is as follows.

SYSTEM system_name

where ***description:***

system_name is a defined system name.

When this form of the "SYSTEM" directive is encountered, all directives specified in the system definition block identified by `system_name` will be processed.

Let us consider an example that demonstrates the use of the "SYSTEM" directive. The following linker directives define a system called *statistics.*

```
system begin statistics
format dos
libpath /libs
library stats, graphics
option stack=8k
end
```

They specify that a *statistics* application is to be created by using the libraries "stats.lib" and "graphics.lib". These library files are located in the directory "/libs". The application requires a stack size of 8k and the specified format of executable will be generated.

Suppose the linker directives in the above example are contained in the file "stats.lnk". If we wish to create a *statistics* application, we can issue the following command.

```
wlink @stats system statistics file myappl
```

As demonstrated by the above example, the "SYSTEM" directive can be used to localize the common attributes that describe a class of applications.

The system deletion directive can be used to redefine a previously defined system. Consider the following example.

```
system begin at_dos
    libpath %WATCOM%\lib286
    libpath %WATCOM%\lib286\dos
    format dos ^
end
system begin n98_dos
    sys at_dos ^
    libpath %WATCOM%\lib286\dos\n98
end
system begin dos
sys at_dos ^
end
```

If you wish to redefine the definition of the "dos" system, you can specify the following set of directives.

```
system delete dos
system begin dos
sys n98_dos ^
end
```

This effectively redefines a "dos" system to be equivalent to a "n98_dos" system (NEC PC-9800 DOS), rather than the previously defined "at_dos" system (AT-compatible DOS).

For additional examples on the use of the "SYSTEM" directive, examine the contents of the `wlink.lnk` and `wlsystem.lnk` files.

The file `wlink.lnk` is a special linker directive file that is automatically processed by the Open Watcom Linker before processing any other directives. On a DOS, OS/2, or Windows-hosted system, this file must be located in one of the paths specified in the **PATH** environment variable. On a QNX-hosted system, this file should be located in the `/etc` directory. A default version of this file is located in the `\watcom\binw` directory on DOS-hosted systems, the `\watcom\binp` directory on OS/2-hosted systems, the `/etc` directory on QNX-hosted systems, and the `\watcom\binnt` directory on Windows 95 or Windows NT-hosted systems. Note that the file `wlink.lnk` includes the file `wlsystem.lnk` which is located in the `\watcom\binw` directory on DOS, OS/2, or Windows-hosted systems and the `/etc` directory on QNX-hosted systems.

The files `wlink.lnk` and `wlsystem.lnk` reference the **WATCOM** environment variable which must be set to the directory in which you installed your software.

14.49.1 Special System Names

There are two special system names. When the linker has processed all object files and the executable file format has not been determined, and a system definition block has not been processed, the directives specified in the "286" or "386" system definition block will be processed. The "386" system definition block will be processed if a 32-bit object file has been processed. Furthermore, only a restricted set of linker directives is allowed in a "286" and "386" system definition block. They are as follows.

- FORMAT
- LIBFILE
- LIBPATH

- LIBRARY
- NAME
- OPTION
- RUNTIME (for Phar Lap executable files only)
- SEGMENT (for OS/2 and QNX executable files only)

14.50 The UNDEFSOK Option

The "UNDEFSOK" option tells the Open Watcom Linker to generate an executable file even if undefined symbols are present. By default, no executable file will be generated if undefined symbols are present.

The format of the "UNDEFSOK" option (short form "U") is as follows.

```
OPTION UNDEFSOK
```

The "NOUNDEFSOK" option tells the Open Watcom Linker to not generate an executable file if undefined symbols are present. This is the default behaviour.

The format of the "NOUNDEFSOK" option (short form "NOU") is as follows.

```
OPTION NOUNDEFSOK
```

14.51 The VERBOSE Option

The "VERBOSE" option controls the amount of information produced by the Open Watcom Linker in the map file. The format of the "VERBOSE" option (short form "V") is as follows.

```
OPTION VERBOSE
```

If the "VERBOSE" option is specified, the linker will list, for each object file, all segments it defines and their sizes. By default, this information is not produced in the map file.

14.52 The VFREMOVAL Option

The "VFREMOVAL" option instructs the linker to remove unused C++ virtual functions. The format of the "VFREMOVAL" option (short form "VFR") is as follows.

```
OPTION VFREMOVAL
```

If the "VFREMOVAL" option is specified, the linker will attempt to eliminate unused virtual functions. In order for the linker to do this, the Open Watcom C++ "zv" compiler option must be used for *all* object files in the executable. The "VFREMOVAL" option works best in concert with the "ELIMINATE" option.

15 The QNX Executable File Format

This chapter deals specifically with aspects of QNX executable files. The QNX executable file format will only run under the QNX operating system.

Input to the Open Watcom Linker is specified on the command line and can be redirected to one or more files or environment strings. The Open Watcom Linker command line format is as follows.

```
wlink {directive}
```

where ***directive*** is any of the following:

ALIAS symbol_name=symbol_name{,symbol_name=symbol_name}
DEBUG dbtype [dblist] | DEBUG [dblist]
DISABLE msg_num{,msg_num}
ENDLINK
FILE obj_spec{,obj_spec}
FORMAT QNX [FLAT]
LANGUAGE
LIBFILE obj_file{,obj_file}
LIBPATH path_name{:path_name}
LIBRARY library_file{,library_file}
MODFILE obj_file{,obj_file}
MODTRACE obj_spec{,obj_spec}
NAME exe_file
NEWSEGMENT
OPTION option{,option}

> ***ARTIFICIAL***
> ***[NO]CACHE***
> ***[NO]CASEEXACT***
> ***CVPACK***
> ***DOSSEG***
> ***ELIMINATE***

[NO]FARCALLS
HEAPSIZE=n
INCREMENTAL
LINEARRELOCS
LONGLIVED
MANGLEDNAMES
MAP[=map_file]
MAXERRORS=n
NAMELEN=n
NODEFAULTLIBS
NORELOCS
OFFSET=n
OSNAME='string'
PACKCODE=n
PACKDATA=n
PRIVILEGE=n
QUIET
REDEFSOK
RESOURCE[=resource_file | 'string']
SHOWDEAD
STACK=n
START=symbol_name
STATICS
SYMFILE[=symbol_file]
[NO]UNDEFSOK
VERBOSE
VFREMOVAL

OPTLIB library_file{,library_file}
PATH path_name{:path_name}
REFERENCE symbol_name{,symbol_name}
SEGMENT seg_desc{,seg_desc}
SORT [GLOBAL] [ALPHABETICAL]
STARTLINK
SYMTRACE symbol_name{,symbol_name}
SYSTEM BEGIN system_name {directive} END
SYSTEM system_name
 # comment
@ directive_file

You can view all the directives specific to QNX executable files by simply typing the following:

```
wlink ? qnx
```

Notes:

1. If the file `/etc/wlink.hlp` exists, the contents of that file will be displayed when the following command is issued.

```
wlink ?
```

2. If all of the directive information does not fit on the command line, type the following.

```
wlink
```

The prompt "WLINK>" will appear on the next line. You can enter as many lines of directive information as required. Press "Ctrl/D" to terminate the input of directive information.

15.1 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"
6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments

belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

16 Open Watcom Linker Diagnostic Messages

The Open Watcom Linker issues three classes of messages; fatal errors, errors and warnings. Each message has a 4-digit number associated with it. Fatal messages start with the digit 3, error messages start with the digit 2, and warning messages start with the digit 1. It is possible for a message to be issued as a warning or an error.

If a fatal error occurs, the linker will terminate immediately and no executable file will be generated.

If an error occurs, the linker will continue to execute so that all possible errors are issued. However, no executable file will be generated since these errors do not permit a proper executable file to be generated.

If a warning occurs, the linker will continue to execute. A warning message is usually informational and does not prevent the creation of a proper executable file. However, all warnings should eventually be corrected.

The messages listed contain references to `%s, %S, %a, %x, %d, %l,` and `%f.` They represent strings that are substituted by the Open Watcom Linker to make the error message more precise.

1. `%s` represents a string. This may be a segment or group name, or the name of a linker directive or option.

2. `%S` represents the name of a symbol.

3. `%a` represents an address. The format of the address depends on the format of the executable file being generated.

4. `%x` represents a hexadecimal number.

5. `%d` represents integers in the range -32768 and 32767.

6. `%l` represents integers in the range -2147483648 and 2147483647.

7. %f represents an executable file format such as DOS, WINDOWS, PHARLAP, NOVELL, OS2, QNX or ELF.

The following is a list of all warning and error messages produced by the Open Watcom Linker followed by a description of the message. A message may contain more than one reference to "%s". In such a case, the description will reference them as "%sn" where n is the occurrence of "%s" in the message.

MSG 2002 **** internal ** - %s**

If this message occurs, you have found a bug in the linker and should report it.

MSG 2008 **cannot open %s1 : %s2**

An error occurred while trying to open the file "%s1". The reason for the error is given by "%s2". Generally this error message is issued when the linker cannot open a file (e.g., an object file or an executable file).

MSG 3009 **dynamic memory exhausted**

The linker uses all available memory when linking an application. When all available memory is used, a spill file will be used. Therefore, unless you are low on disk space, the linker will always be able to generate the executable file. Dynamic memory is the memory the linker uses to build its internal data structures and symbol table. A spill file is not used for dynamic memory. If the linker issues this message, it cannot link your application. The following are suggestions that may help you in this situation.

1. Concatenate all your object files into one and specify only the resulting object file as input to the linker. For example, you can issue the following command.

   ```
   % cat *.obj > all.tmp
   % mv all.tmp all.obj
   ```

   This technique only works for OMF-type object files. This significantly reduces the size of the file list the linker must maintain.

2. Object files may contain a record which specifies the module name. This information is used by Open Watcom Debugger to locate modules during a debugging session and usually contains the full path of the source file. This can consume a significant amount of memory when many such object files are being linked. If your source is being compiled by the Open Watcom C or C++ compiler, you can use the

"nm" option to set the module name to just the file name. This reduces the amount of memory required by the linker. If your are using Open Watcom Debugger to debug your application, you may have to use the "set source" command so that the source corresponding to a module can be located.

3. Typically, when you are compiling a program for a large code model, each module defines a different "text" segment. If you are compiling your application using the Open Watcom C or C++ compiler, you can reduce the number of "text" segments that the linker has to process by specifying the "nt" option. The "nt" option allows you to specify the name of the "text" segment so that a group of object files define the same "text" segment.

MSG 2010,3010 **I/O error processing %s1 : %s2**

An error has occurred while processing the file "%s1". The cause of the error is given by "%s2". This error is usually detected while reading from object and library files or writing to the spill file or executable file. For example, this error would be issued if a "disk full" condition existed.

MSG 2011 **invalid object file attribute**

The linker encountered an object file that was not of the format required of an object file.

MSG 2012 **invalid library file attribute**

The linker encountered a library file that was not of the format required of a library file.

MSG 3013 **break key detected**

The linking process was interrupted by the user from the keyboard.

MSG 1014 **stack segment not found**

The linker identifies the stack segment by a segment defined as having the "STACK" attribute. This message is issued if no such segment is encountered. This usually happens if the linker cannot find the run-time libraries required to link your application.

MSG 2015 **bad relocation type specified**

This message is issued if a a relocation is found in an object file which the linker does not support.

MSG 2016 **%a: absolute target invalid for self-relative relocation**

This message is issued, for example, if a near call or jump is made to an external symbol which is defined using the "EQU" assembler directive. "%a" identifies the location of the near call or jump instruction.

MSG 2017 **bad location specified for self-relative relocation at %a**

This message is issued if a bad fixup is encountered. "%a" defines the location of the fixup.

MSG 2018 **relocation offset at %a is out of range**

This message is issued when the offset part of a relocation exceeds 64K in a 16-bit executable or an Alpha executable. "%a" defines the location of the fixup. The error is most commonly caused by errors in coding assembly language routines. Consider a module that references an external symbol that is defined in a segment different from the one in which the reference occurred. The module, however, specifies that the segment in which the symbol is defined is the same segment as the segment that references the symbol. This error is most commonly caused when the "EXTRN" assembler directive is placed after the "SEGMENT" assembler directive for the segment referencing the symbol. If the segment that references the symbol is allocated far enough away from the segment that defines the symbol, the linker will issue this message.

MSG 1019 **segment relocation at %a**

This message is issued when a 16-bit segment relocation is encountered and "FORMAT DOS COM", "FORMAT PHARLAP" or "FORMAT NOVELL" has been specified. None of the above executable file formats allow segment relocation. "%a" identifies the location of the segment relocation.

MSG 2020 **size of group %s exceeds 64k by %l bytes**

The group "%s" has exceeded the maximum size (64K) allowed for a group in a 16-bit executable by "%l" bytes. Usually, the group is "DGROUP" (the default data segment) and your application has placed too much data in this group. One of the following may solve this problem.

1. If you are using the Open Watcom C or C++ compiler, you can place some of your data in a far segment by using the "far" keyword when

defining data. You can also decrease the value of the data threshold by using the "zt" compiler option. Any datum whose size exceeds the value of the data threshold will be placed in a far segment.

2. If you are using the Open Watcom FORTRAN 77 compiler, you can decrease the value of the data threshold by using the "dt" compiler option. Any datum whose size exceeds the value of the data threshold will be placed in a far segment.

MSG 2021 **size of segment %s exceeds 64k by %l bytes**

The segment "%s" has exceeded the maximum size (64K) for a segment in a 16-bit executable. This usually occurs if you are linking a 16-bit application that has been compiled for a small code model and the size of the application has grown in such a way that the size of the code segment ("_TEXT") has exceeded 64K. You can overlay your application or compile it for a large code model if you cannot reduce the amount of code in your application.

MSG 2022 **cannot have a starting address with an imported symbol**

When generating an OS/2 executable file, a symbol imported from a DLL cannot be a start address. When generating a NetWare executable file, a symbol imported from an NLM cannot be a start address.

MSG 1023 **no starting address found, using %a**

The starting address defines the location where execution is to begin and must be defined by a special "module end" record in one of the object files linked into your application. This message is issued if no such record is encountered in which case a default starting address, namely "%a", will be used. This usually happens if the linker cannot find the run-time libraries required to link your application.

MSG 2024 **missing overlay loader**

This message is issued when an overlayed 16-bit DOS executable is being linked and the overlay manager has not been encountered. This usually happens if the linker cannot find the run-time libraries required to link your application.

MSG 2025 **short vector %d is out of range**

This message is issued when the linker is creating an overlayed 16-bit DOS executable and "OPTION SMALL" is specified. Since an overlay vector contains a near call to the overlay loader followed by a near jump to the routine

corresponding to the overlay vector, all code including the overlay manager and all overlay vectors must be less than 64K. This message is issued if the offset of an overlay vector from the overlay loader or the corresponding routine exceeds 64K.

MSG 2026 **redefinition of reserved symbol %s**

The linker defines certain reserved symbols. These symbols are "_edata", "_end", "__OVLTAB__", "__OVLSTARTVEC__", "__OVLENDVEC__", "__LOVLLDR__", "__NOVLLDR__", "__SOVLLDR__", "__LOVLINIT__", "__NOVLINIT__" and "__SOVLINIT__". The symbols "__OVLTAB__", "__OVLSTARTVEC__", "__OVLENDVEC__", "__LOVLLDR__", "__NOVLLDR__", "__SOVLLDR__", "__LOVLINIT__", "__NOVLINIT__" and "__SOVLINIT__" are defined only if you are using overlays in 16-bit DOS executables. The symbols "_edata" and "_end" are defined only if the "DOSSEG" option is specified. Your application must not attempt to define these symbols. "%s" identifies the reserved symbol.

MSG 1027 **redefinition of %S ignored**

The symbol "%S" has been defined by more that one module; the first definition is used. This is only a warning message. Note that if a symbol is defined more than once and its address is the same in both cases, no warning will be issued. This prevents the warning message from being issued when linking FORTRAN 77 modules that contain common blocks.

MSG 1028,2028 **%S is an undefined reference**

The symbol "%S" has been referenced but not defined. Check that the spelling of the symbol is consistent. If you wish the linker to ignore undefined references, use the "UNDEFSOK" option.

MSG 2029 **premature end of file encountered**

This error is issued while processing object files and object modules from libraries and is caused if the end of the file or module is reached before the "module end" record is encountered. The probable cause is a truncated object file.

MSG 2030 **multiple starting addresses found**

The starting address defines the location where execution is to begin and is defined by a "module end" record in a particular object file. This message is

issued if more than one object file contains a "module end" record that defines a starting address.

MSG 2031 **segment %s is in group %s and group %s**

The segment "%s1" has been defined to be in group "%s2" in one module and in group "%s3" in another module. A segment can only belong to one group.

MSG 1032 **record (type 0x%x) not processed**

An object record type not supported by the linker has been encountered. This message is issued when linking object modules created by other compilers or assemblers that create object files with records that the linker does not support.

MSG 2033,3033 **directive error near '%s'**

A syntax error occurred while the linker was processing directives. "%s" specifies where the error occurred.

MSG 2034 **%a cannot have an offset with an imported symbol**

An imported symbol is one that was specified in an "IMPORT" directive. Imported symbols are defined in Windows or OS/2 16-bit DLLs and in Netware NLMs. References to imported symbols must always have an offset value of 0. If "DosWrite" is an imported symbol, then referencing "DosWrite+2" is illegal. "%a" defines the location of the illegal reference.

MSG 1038 **DEBUG directive appears after object files**

This message is issued if the first "DEBUG" directive appears after a "FILE" directive. A common error is to specify a "DEBUG" directive after the "FILE" directives in which case no debugging information for those object files is generated in the executable file.

MSG 2039 **ALIGNMENT value too small**

The value specified in the "ALIGNMENT" option refers to the alignment of segments in the executable file. For 16-bit Windows or 16-bit OS/2, segments in the executable file are pointed to by a segment table. An entry in the segment table contains a 16-bit value which is a multiple of the alignment value. Together they form the offset of the segment from the start of the segment table. The smaller the alignment, the bigger the value required in the segment table to point to the segment. If this value exceeds 64K, then a larger alignment value is required to decrease the size that goes in the segment table.

MSG 2040 **ordinal in IMPORT directive not valid**

The specified ordinal in the "IMPORT" directive is incorrect (e.g., -1). An ordinal number must be in the range 0 to 65535.

MSG 2041 **ordinal in EXPORT directive not valid**

The specified ordinal in the "EXPORT" directive is incorrect (e.g., -1). An ordinal number must be in the range 0 to 65535.

MSG 2042 **too many IOPL words in EXPORT directive**

The maximum number of IOPL words for a 16-bit executable is 63.

MSG 1043 **duplicate exported ordinal**

This message is issued for ordinal numbers specified in an "EXPORT" directive for symbols belonging to DLLs. This message is issued if an ordinal number is assigned to two different symbols. A warning is issued and the linker assigns a non-used ordinal number to the symbol that caused the warning.

MSG 1044,2044 **exported symbol %s not found**

This message is issued when generating a DLL or NetWare NLM. An attempt has been made to define an entry point into a DLL or NLM that does not exist.

MSG 1045 **segment attribute defined more than once**

A segment appearing in a "SEGMENT" directive has been given conflicting or duplicate attributes.

MSG 1046 **segment name %s not found**

The segment name specified in a "SEGMENT" directive has not been defined.

MSG 1047 **class name %s not found**

The class name specified in a "SEGMENT" directive has not been defined.

MSG 1048 **inconsistent attributes for automatic data segment**

This message is issued for Windows or OS/2 16-bit executable files. Two conflicting attributes were specified for the automatic data segment. For

example, "LOADONCALL" and "PRELOAD" are conflicting attributes. Only the first attribute is used.

MSG 2049 **invalid STUB file**

The stub file is not a valid executable file. The stub file is only used for OS/2 executable files and Windows (both Win16 and Win32) executable files.

MSG 1050 **invalid DLL specified in OLDLIBRARY option**

The DLL specified in an "OLDLIBRARY" option is not a valid dynamic link library.

MSG 2051 **STUB file name same as executable file name**

When generating an OS/2 or Windows (Win16, Win32) executable file, the stub file name must not be same as the executable file name.

MSG 2052 **relocation at %a not in the same segment**

This message is only issued for Windows (Win16), OS/2, Phar Lap, and QNX executables. A relative fixup must relocate to the same segment. "%a" defines the location of the fixup.

MSG 2053 **%a: cannot reach a DLL with a relative relocation**

A reference to a symbol in an OS/2 or Windows 16-bit DLL must not be relative. "%a" defines the location of the reference.

MSG 1054 **debugging information incompatible: using line numbers only**

An attempt has been made to link an object file with out-of-date debugging information.

MSG 2055 **%a: frame must be the same as the target in protected mode**

Each relocation consists of three components; the location being relocated, the target (or address being referenced), and the frame (the segment to which the target is adjusted). In protected mode, the segment of the target must be the same as the frame. "%a" defines the location of the fixup. This message does not apply to 32-bit OS/2 and Windows (Win32).

MSG 2056 **cannot find library member %s(%s)**

Library member "%s2" in library file "%s1" could not be found. This message is issued if the library file could not be found or the library file did not contain the specified member.

MSG 3057 **executable format has been established**

This message is issued if there is more than one "FORMAT" directive.

MSG 1058 **%s option not valid for %s executable**

The option "%s1" can only be specified if an executable file whose format is "%s2" is being generated.

MSG 1059,2059 **value for %s too large**

The value specified for option "%s" exceeds its limit.

MSG 1060 **value for %s incorrect**

The value specified for option "%s" is not in the allowable range.

MSG 1061 **multiple values specified for REALBREAK**

The "REALBREAK" option for Phar Lap executables can only be specified once.

MSG 1062 **export and import records not valid for %f**

This message is issued if a reference to a DLL is encountered and the executable file format is not one that supports DLLs. The file format is represented by "%f".

MSG 2063 **invalid relocation for flat memory model at %a**

A segment relocation in the flat memory model was encountered. "%a" defines the location of the fixup.

MSG 2064 **cannot combine 32-bit segments (%s1) with 16-bit segments (%s2)**

A 32-bit segment "%s1" and a 16-bit segment "%s2" have been encountered. Mixing object files created by a 286 compiler and object files created by a 386 compiler is the most probable cause of this error.

MSG 2065 **REALBREAK symbol %s not found**

The symbol specified in the "REALBREAK" option for Phar Lap executables has not been defined.

MSG 2066 **invalid relative relocation type for an import at %a**

This message is issued only if a NetWare executable file is being generated. An imported symbol is one that was specified in an "IMPORT" directive or an import library. Any reference to an imported symbol must not refer to the segment of the imported symbol. "%a" defines the location of the reference.

MSG 2067 **%a: cannot relocate between code and data in Novell formats**

This message is issued only if a NetWare executable file is being generated. Segment relocation is not permitted. "%a" defines the location of the fixup.

MSG 2068 **absolute segment fixup not valid in protected mode**

A reference to an absolute location is not allowed in protected mode. A protected-mode application is one that is being generated for OS/2, FlashTek's DOS extender, Phar Lap's 386|DOS-Extender, Tenberry Software's DOS/4G or DOS/4GW DOS extender, Novell's NetWare operating systems, Windows NT, or Windows 95. An absolute location is most commonly defined by the "EQU" assembler directive.

MSG 1069 **unload CHECK procedure not found**

This message is issued only if a NetWare executable file is being generated. The symbol specified in the "CHECK" option has not been defined.

MSG 2070 **START procedure not found**

This message is issued only if a NetWare executable file is being generated. The symbol specified in the "START" option has not been defined. The default "START" symbol is "_Prelude".

MSG 2071 **EXIT procedure not found**

This message is issued only if a NetWare executable file is being generated. The symbol specified in the "EXIT" option has not been defined. The default "STOP" symbol is "_Stop".

MSG 1072 **SECTION directive not allowed in root**

When describing 16-bit overlays, "SECTION" directives must appear between a "BEGIN" directive and its corresponding "END" directive.

MSG 2073 **bad Novell file format specified**

An invalid NetWare executable file format was specified. Valid formats are NLM, DSK, NAM, LAN, MSL, HAM, CDM or a numerical module type.

MSG 2074 **circular alias found for %s**

An attempt was made to circularly define the symbol name specified in an ALIAS directive. For example:

```
ALIAS foo1=foo2, foo2=foo1
```

MSG 2075 **expecting an END directive**

A "BEGIN" directive is missing its corresponding "END" directive.

MSG 1076 **%s option multiply specified**

The option "%s" can only be specified once.

MSG 1080 **file %s is a %d-bit object file**

A 32-bit attribute was encountered while generating a 16-bit executable file format, or a 16-bit attribute was encountered while generating a 32-bit executable file format.

MSG 2082 **invalid record type 0x%x**

An object record type not recognized by the linker has been encountered. This message is issued when linking object modules created by other compilers or assemblers that create object files with records that the linker does not recognize.

MSG 2083 **cannot reference address %a from frame %x**

When generating a 16-bit executable, the offset of a referenced symbol was greater than 64K from the location referencing it.

MSG 2084 **target offset exceeds 64K at %a**

When generating a 16-bit executable, the computed offset for a symbol exceeds 64K. "%a" defines the location of the fixup.

MSG 2086 **invalid starting address for .COM file**

The value of the segment of the starting address for a 16-bit DOS "COM" file, as specified in the map file, must be 0.

MSG 1087 **stack segment ignored in .COM file**

A stack segment must not be defined when generating a 16-bit DOS "COM" file. Only a single physical segment is allowed in a DOS "COM" file. The stack is allocated from the high end of the physical segment. That is, the initial value of SP is hexadecimal FFFE.

MSG 3088 **virtual memory exhausted**

This message is similar to the "dynamic memory exhausted" message. The DOS-hosted version of the linker has run out of memory trying to keep track of virtual memory blocks. Virtual memory blocks are allocated from expanded memory, extended memory and the spill file.

MSG 2089 **program too large for a .COM file**

The total size of a 16-bit DOS "COM" program must not exceed 64K. That is, the total amount of code and data must be less than 64K since only a single physical segment is allowed in a DOS "COM" file. You must decrease the size of your program or generate a DOS "EXE" file.

MSG 1090 **redefinition of %s by %s ignored**

The symbol "%s1" has been redefined by module "%s2". This message is issued when the size specified in the "NAMELEN" option has caused two symbols to map to the same symbol. For example, if the symbols *routine1* and *routine2* are encountered and "OPTION NAMELEN=7" is specified, then this message will be issued since the first seven characters of the two symbols are identical.

MSG 2091 **group %s is in more than one overlay**

A group that spans more than one section in a 16-bit DOS executable has been detected.

MSG 2092 **NEWSEGMENT directive appears before object files**

The 16-bit "NEWSEGMENT" directive must appear after a "FILE" directive.

MSG 2093 **cannot open %s**

This message is issued when the linker is unable to open a file and is unable to determine the cause.

MSG 2094 **i/o error processing %s**

This message is issued when the linker has encountered an i/o error while processing the file and is unable to determine the cause. This message may be issued when reading from object and library files, or writing to the executable and spill file.

MSG 3097 **too many library modules**

This message is similar to the "dynamic memory exhausted" message. This message if issued when the "DISTRIBUTE" option for 16-bit DOS executables is specified. The linker has run out of memory trying to keep track of the relationship between object modules extracted from libraries and the overlays they should be placed in.

MSG 1098 **Offset option must be a multiple of %dK**

The value specified with the "OFFSET" option must be a multiple of 4K (4096) for Phar Lap and QNX executables and a multiple of 64K (65536) for OS/2 and Windows 32-bit executables.

MSG 2099 **symbol name too long: %s**

The maximum size (approximately 2048) of a symbol has been exceeded. Reduce the size of the symbol to avoid this error.

MSG 1101 **invalid incremental information file**

The incremental information file is corrupt or from an older version of the compiler. The old information file and the executable will be deleted and new ones will be generated.

MSG 1102 **object file %s not found for tracing**

A "SYMTRACE" or "MODTRACE" directive contained an object file (namely %s) that could not be found.

MSG 1103 **library module %s(%s) not found for tracing**

A "SYMTRACE" or "MODTRACE" directive contained an object module (namely module %s1 in library %s2) that could not be found.

MSG 1105 **cannot reserve %l bytes of extra overlay space**

The value specified with the "AREA" option for 16-bit DOS executables results in an executable file that requires more than 1 megabyte of memory to execute.

MSG 1107 **undefined system name: %s**

The name %s was referenced in a "SYSTEM" directive but never defined by a system block definition.

MSG 1108 **system %s defined more than once**

The name %s has appeared in a system definition block more than once.

MSG 1109 **OFFSET option is less than the stack size**

For the QNX operating system, the stack is placed at the front of the executable image and thus the initial load address must leave enough room for the stack.

MSG 1110 **library members not allowed in libfile**

Only object files are allowed in a "LIBFILE" directive. This message will be issued if a module from a library file is specified in a "LIBFILE" directive.

MSG 1111 **error in default system block**

The default system block definition (system name "286" for 16-bit applications) and (system name "386" for 32-bit applications) contains a directive error. The system name "286" or "386" is automatically referenced by the linker when the format of the executable cannot be determined (i.e. no "FORMAT" directive has been specified).

MSG 3114 **environment name specified incorrectly**

This message is specified if the environment variable is not properly enclosed between two percent (%) characters.

MSG 1115 **environment name %s not found**

The environment variable %s has not been defined in the environment space.

MSG 1116 **overlay area must be at least %l bytes**

This message is issued if the size of the largest overlay exceeds the size of the overlay area specified by the "AREA" option for 16-bit DOS executables.

MSG 1117 **segment number too high for a movable entry point**

The segment number of a moveable segment must not exceed 255 for 16-bit executables. Reduce the number of segments or use the "PACKCODE" option.

MSG 1118 **heap size too large**

This message is issued if the size of the heap, stack and the default data segment (group DGROUP) exceeds 64K for 16-bit executables.

MSG 2119 **wlib import statement incorrect**

The "EXPORT" directive allows you to specify a library command file. This command file is scanned for any librarian commands that create import library entries. An invalid command was detected. See the section entitled "The EXPORT Directive" for the correct format of these commands.

MSG 2120 **application too large to run under DOS**

This message is issued if the size of the 16-bit DOS application exceeds 1M.

MSG 1121 **'%s' has already been exported**

The linker has detected an attempt to export a symbol more than once. For example, a name appearing in more than one "EXPORT" directive will cause this message to be issued. Also, if you have declared a symbol as an export in your source and have also specified the same symbol in an "EXPORT" directive, this message will be issued. This message is only a warning.

MSG 3122 **no FILE directives found**

This message is issued if no "FILE" directive has been specified. In other words, you have specified no object files to link.

MSG 3123 **overlays are not supported in this version of the linker**

This version of the linker does not support the creation of overlaid 16-bit executables.

MSG 1124 **lazy reference for %S has different default resolutions**

A lazy external reference is one which has two resolutions: a preferred one and a default one which is used if the preferred one is not found. In this case, the linker has found two lazy references that have the same preferred resolution but different default resolutions.

MSG 1125 **multiple aliases found for %S**

The linker has found a name which has been aliased to two different symbols.

MSG 1126 **%s has been modified: doing full relink**

The linker has determined that the time stamps on the executable file and symbolic information file (.sym) are different. An incremental link will not be done.

MSG 2127 **cannot export symbol %S**

An attempt was made to export a symbol defined with an absolute address or to export an imported symbol. It is not possible to export these symbols with the "EXPORT" directive.

MSG 3128 **directive error near beginning of input**

The linker detected an error at the start of the command line.

MSG 3129 **address information too large**

The linker has encountered a segment that appears in more than 11000 object files. An empty segment does not affect this limit. This can only occur with Watcom debugging information. If this message appears, switch to DWARF debugging information.

MSG 1130 **%s is an invalid shared nlm file**

The NLM specified in a "SHAREDNLM" option is not valid.

MSG 3131 **cannot open spill file: file already exists**

All 26 of the DOS-hosted linker's possible spill file names are in use. Spill files can accumulate when linking on a multi-tasking system and the directory in which the spill file is created is identical for each invocation of the linker.

MSG 2132 **curly brace delimited list incorrect**

A list delimited by curly braces is not correct. The most likely cause is a missing right brace.

MSG 1133 **no realbreak specified for 16-bit code**

While generating a Phar Lap executable file, both 16-bit and 32-bit code was linked together and no "REALBREAK" option has been specified. A warning message is issued since this may be a potential problem.

MSG 1134 **%s is an invalid message file**

The file specified in a "MESSAGE" option for NetWare executable files is invalid.

MSG 3135 **need exactly 1 overlay area with dynamic overlay manager**

Only a single overlay area is supported by the 16-bit dynamic overlay manager.

MSG 1136 **segment relocation to a read/write data segment found at %a(%S)**

The "RWRELOCCHECK" option for 16-bit Windows (Win16) executables has been specified and the linker has detected a segment relocation to a read/write data segment. Where the name of the offending symbol is not available, "identifier unavailable" is used.

MSG 3137 **too many errors encountered**

This message is issued when the number of error messages issued by the linker exceeds the number specified by the "MAXERRORS" option.

MSG 3138 **invalid filename '%s'**

The linker performs a simple filename validation whenever a filename is specified to the linker. For example, a directory specification is not a valid filename.

MSG 3139 **cannot have both 16-bit and 32-bit object files**

It is impossible to mix 16-bit code and 32-bit code in the same executable when generating a QNX executable file.

MSG 1140 **invalid message number**

An invalid message number has been specified in a "DISABLE" directive.

MSG 1141 **virtual function table record for %s mismatched**

The linker performs a consistency check to ensure that the C++ compiler has not generated incorrect virtual function information. If the message is issued, please report this problem.

MSG 1143 **not enough memory to sort map file symbols**

There was not enough memory for the linker to sort the symbols in the "Memory Map" portion of the map file. This will only occur when the "SORT GLOBAL" option has been specified.

MSG 1145 **%S is both pure virtual and non-pure virtual**

A function has been declared both as "pure" and "non-pure" virtual.

MSG 2146 **%s is an invalid object file**

Something was encountered in the object file that cannot be processed by the linker.

MSG 3147 **Ambiguous format specified**

Not enough of the FORMAT directive attributes were specified to enable the linker to determine the executable file format. For example,

```
FORMAT OS2
```

will generate this message.

MSG 1148 **Invalid segment type specified**

The segment type must be one of CODE or DATA.

MSG 1149 **Only one debugging format can be specified**

The debugging format must be one of Watcom, CodeView, DWARF (default), or Novell. You cannot specify multiple debugging formats.

MSG 1150 **file %s has code for a different processor**

An object file has been encountered which contains code compiled for a different processor (e.g., an Intel application and an Alpha object file).

MSG 2151 **big endian code not supported**

Big endian code is not supported by the linker.

MSG 2152 **no dictionary found**

No symbol search dictionary was found in a library that the linker attempted to process.

MSG 2154 **cannot execute %s1 : %s2**

An attempt by the linker to spawn another application failed. The application is specified by "%s1" and the reason for the failure is specified by "%s2".

MSG 2155 **relocation at %a to an improperly aligned target**

Some relocations in Alpha executables require that the object be aligned on a 4 byte boundary.

MSG 2156 **OPTION INCREMENTAL must be one of the first directives specified**

The option must be specified before any option or directive which modifies the linker's symbol table (e.g., IMPORT, EXPORT, REFERENCE, ALIAS).

MSG 3157 **no code or data present**

The linker requires that there be at least 1 byte of either code or data in the executable.

MSG 1158 **problem adding resource information**

The resource file is invalid or corrupt.

MSG 3159 **incremental linking only supports DWARF debugging information**

When OPTION INCREMENTAL is used, you cannot specify non-DWARF debugging information for the executable. You must specify DEBUG DWARF when requesting debugging information.

MSG 3160 **incremental linking does not support dead code elimination**

When OPTION INCREMENTAL is used, you cannot specify OPTION ELIMINATE.

MSG 1162 **relocations on iterated data not supported**

An object file was encountered that contained an iterated data record that requires relocation. This is most commonly caused by a module coded in assembly language.

MSG 1163 **module has not been compiled with the "zv" option**

When OPTION VFREMOVAL is used, all object files must be compiled with the "zv" option. The linker has detected an object file that has not been compiled with this option.

MSG 3164 **incremental linking does not support virtual function removal**

When OPTION INCREMENTAL is used, you cannot also specify OPTION VFREMOVAL.

MSG 1165 **resource file %s too big**

The resource file specified in OPTION RESOURCE was too big to fit inside the QNX executable. The maximum size is approximately 32000 bytes.

MSG 2166 **both %s1 and %s2 marked as starting symbols**

If the linker sees that there is more than one starting address specified in the program and they have symbol names associated with them, it will emit this error message. If there is more than one starting address specified and at least one of them is unnamed, it will issue message 2030.

MSG 1167 **The NLM internal name (%s) has been truncated as it exceeds the maximum size.**

This message is issued when generating a NetWare NLM. The output file name as specified by the NAME directive has specified a long file name (exceeds 8.3). The linker will truncate the generated file name by using the first eight characters of the specified file name and the first three characters of the file extension (if supplied), separated by a period.

The Open Watcom Library Manager

17 The Open Watcom Library Manager

17.1 Introduction

The Open Watcom Library Manager can be used to create and update object library files. It takes as input an object file or a library file and creates or updates a library file. For OS/2, Win16 and Win32 applications, it can also create import libraries from Dynamic Link Libraries.

An object library is essentially a collection of object files. These object files generally contain utility routines that can be used as input to the Open Watcom Linker to create an application. The following are some of the advantages of using library files.

1. Only those modules that are referenced will be included in the executable file. This eliminates the need to know which object files should be included and which ones should be left out when linking an application.

2. Libraries are a good way of organizing object files. When linking an application, you need only list one library file instead of several object files.

The Open Watcom Library Manager currently runs under the following operating systems.

- DOS
- OS/2
- QNX
- Windows

17.2 The Open Watcom Library Manager Command Line

The following describes the Open Watcom Library Manager command line.

wlib [options_1] lib_file [cmd_list]

The square brackets "[]" denote items which are optional.

lib_file is the file specification for the library file to be processed. If no file extension is specified, a file extension of "lib" is assumed.

options_1 is a list of valid options. Options may be specified in any order. Options are preceded by a "—" character.

cmd_list is a list of commands to the Open Watcom Library Manager specifying what operations are to be performed. Each command in *cmd_list* is separated by a space.

The following is a summary of valid options. Items enclosed in square brackets "[]" are optional. Items separated by an or-bar "|" and enclosed in parentheses "()" indicate that one of the items must be specified. Items enclosed in angle brackets "<>" are to be replaced with a user-supplied name or value (the "<>" are not included in what you specify).

? display the usage message
b suppress creation of backup file
c perform case sensitive comparison
d=<output_directory>
directory in which extracted object modules will be placed
fa output AR format library
fm output MLIB format library
fo output OMF format library
h display the usage message
ia generate AXP import records
ii generate X86 import records
ip generate PPC import records
ie generate ELF import records
ic generate COFF import records

io generate OMF import records

i(r|n)(n|o) imports for the resident/non-resident names table are to be imported by name/ordinal.

l[=<list_file>]
create a listing file

m display C++ mangled names

n always create a new library

o=<output_file>
set output file name for library

p=<record_size>
set library page size (supported for "OMF" library format only)

q suppress identification banner

s strip line number records from object files (supported for "OMF" library format only)

t remove path information from module name specified in THEADR records (supported for "OMF" library format only)

v do not suppress identification banner

x extract all object modules from library

zld strip file dependency info from object files (supported for "OMF" library format only)

The following sections describe the operations that can be performed on a library file. Note that before making a change to a library file, the Open Watcom Library Manager makes a backup copy of the original library file unless the "o" option is used to specify an output library file whose name is different than the original library file, or the "b" option is used to suppress the creation of the backup file. The backup copy has the same file name as the original library file but has a file extension of "bak". Hence, **lib_file** should not have a file extension of "bak".

17.3 Open Watcom Library Manager Module Commands

The following is a summary of basic Open Watcom Library Manager module manipulation commands:

+ add module to a library

- remove module from a library

**** or :*** extract module from a library (**:** is used with a UNIX-hosted version of the Open Watcom Library Manager, otherwise ***** is used)

++ add import library entry

17.4 Adding Modules to a Library File

An object file can be added to a library file by specifying a **+obj_file** command where **obj_file** is the file specification for an object file. A file extension of "o" is assumed if none is specified. If the library file does not exist, a warning message will be issued and the library file will be created.

Example:
```
wlib mylib +myobj
```

In the above example, the object file "myobj" is added to the library file "mylib.lib".

When a module is added to a library, the Open Watcom Library Manager will issue a warning if a symbol redefinition occurs. This will occur if a symbol in the module being added is already defined in another module that already exists in the library file. Note that the module will be added to the library in any case.

It is also possible to combine two library files together. The following example adds all modules in the library "newlib.lib" to the library "mylib.lib".

Example:
```
wlib mylib +newlib.lib
```

Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are adding an object file.

17.5 Deleting Modules from a Library File

A module can be deleted from a library file by specifying a **-mod_name** command where **mod_name** is the file name of the object file when it was added to the library with the directory and file extension removed.

Example:
```
wlib mylib -myobj
```

In the above example, the Open Watcom Library Manager is instructed to delete the module "myobj" from the library file "mylib.lib".

It is also possible to specify a library file instead of a module name.

Example:
```
wlib mylib -oldlib.lib
```

In the above example, all modules in the library file "oldlib.lib" are removed from the library file "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are removing an object module.

17.6 Replacing Modules in a Library File

A module can be replaced by specifying a **-+mod_name** or **+-mod_name** command. The module **mod_name** is deleted from the library. The object file "mod_name" is then added to the library.

Example:
```
wlib mylib -+myobj
```

In the above example, the module "myobj" is replaced by the object file "myobj".

It is also possible to merge two library files.

Example:
```
wlib mylib -+updlib.lib
```

In the above example, all modules in the library file "updlib.lib" replace the corresponding modules in the library file "mylib.lib". Any module in the library "updlib.lib" not in library "mylib.lib" is added to the library "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are replacing an object module.

17.7 Extracting a Module from a Library File

A module can be extracted from a library file by specifying a **:mod_name** [=file_name] command. The module **mod_name** is not deleted but is copied to a disk file. If **mod_name** is preceded by a path specification, the output file will be placed in the directory identified by the path specification. If **mod_name** is followed by a file extension, the output file will contain the specified file extension.

Example:
```
wlib mylib :myobj
```

In the above example, the module "myobj" is copied to a disk file. The disk file will be an object file with file name "myobj". A file extension of "o" will be used.

Example:
```
wlib mylib :myobj.out
```

In the above example, the module "myobj" will be extracted from the library file "mylib.lib" and placed in the file "myobj.out"

The following form of the extract command can be used if the module name is not the same as the output file name.

Example:
```
wlib mylib :myobj=newmyobj.out
```

You can extract a module from a file and have that module deleted from the library file by specifying a **:-mod_name** command. The following example performs the same operations as in the previous example but, in addition, the module is deleted from the library file.

Example:
```
wlib mylib :-myobj.out
```

Note that the same result is achieved if the delete operator precedes the extract operator.

17.8 Creating Import Libraries

The Open Watcom Library Manager can also be used to create import libraries from Dynamic Link Libraries. Import libraries are used when linking OS/2, Win16 or Win32 applications.

Example:
```
wlib implib +dynamic.dll
```

In the above example, the following actions are performed. For each external symbol in the specified Dynamic Link Library, a special object module is created that identifies the external symbol and the actual name of the Dynamic Link Library it is defined in. This object module is then added to the specified library. The resulting library is called an import library.

Note that you must specify the "dll" file extension. Otherwise, the Open Watcom Library Manager will assume you are adding an object file.

17.9 Creating Import Library Entries

An import library entry can be created and added to a library by specifying a command of the following form.

```
++sym.dll_name[.[altsym].export_name][.ordinal]
```

where ***description:***

sym is the name of a symbol in a Dynamic Link Library.

dll_name is the name of the Dynamic Link Library that defines `sym`.

altsym is the name of a symbol in a Dynamic Link Library. When omitted, the default symbol name is `sym`.

export_name is the name that an application that is linking to the Dynamic Link Library uses to reference `sym`. When omitted, the default export name is `sym`.

ordinal is the ordinal value that can be used to identify `sym` instead of using the name `export_name`.

Example:
```
wlib math ++__sin.trig.sin.1
```

In the above example, an import library entry will be created for symbol `sin` and added to the library "math.lib". The symbol `sin` is defined in the Dynamic Link Library called "trig.dll" as `__sin`. When an application is linked with the library "math.lib", the resulting executable file will contain an import by ordinal value 1. If the ordinal value was omitted, the resulting executable file would contain an import by name `sin`.

17.10 Commands from a File or Environment Variable

The Open Watcom Library Manager can be instructed to process all commands in a disk file or environment variable by specifying the **@name** command where **name** is a file specification for the command file or the name of an environment variable. A file extension of "lbc" is assumed for files if none is specified. The commands must be one of those previously described.

Example:
```
wlib mylib @mycmd
```

In the above example, all commands in the environment variable "mycmd" or file "mycmd.lbc" are processed by the Open Watcom Library Manager.

17.11 Open Watcom Library Manager Options

The following sections describe the list of options allowed when invoking the Open Watcom Library Manager.

17.11.1 Suppress Creation of Backup File - "b" Option

The "b" option tells the Open Watcom Library Manager to not create a backup library file. In the following example, the object file identified by "new" will be added to the library file "mylib.lib".

Example:
```
wlib -b mylib +new
```

If the library file "mylib.lib" already exits, no backup library file ("mylib.bak") will be created.

17.11.2 Case Sensitive Symbol Names - "c" Option

The "c" option tells the Open Watcom Library Manager to use a case sensitive compare when comparing a symbol to be added to the library to a symbol already in the library file. This will cause the names "myrtn" and "MYRTN" to be treated as different symbols. By default, comparisons are case insensitive. That is the symbol "myrtn" is the same as the symbol "MYRTN".

17.11.3 Specify Output Directory - "d" Option

The "d" option tells the Open Watcom Library Manager the directory in which all extracted modules are to be placed. The default is to place all extracted modules in the current directory.

In the following example, the module "mymod" is extracted from the library "mylib.lib". The module will be placed in the file "/o/mymod.o".

Example:
```
wlib -d=/o mymod
```

17.11.4 Specify Output Format - "f" Option

The "f" option tells the Open Watcom Library Manager the format of the output library. The default output format is determined by the type of object files that are added to the library when it is created. The possible output format options are:

fa output AR format library

fm output MLIB format library

fo output OMF format library

17.11.5 Generating Imports - "i" Option

The "i" option can be used to describe type of import library to create.

ia generate AXP import records

ii generate X86 import records

ip generate PPC import records

ie generate ELF import records

ic generate COFF import records

io generate OMF import records

When creating import libraries from Dynamic Link Libraries, import entries for the names in the resident and non-resident names tables are created. The "i" option can be used to describe the method used to import these names.

iro Specifying "iro" causes imports for names in the resident names table to be imported by ordinal.

irn Specifying "irn" causes imports for names in the resident names table to be imported by name. This is the default.

ino Specifying "ino" causes imports for names in the non-resident names table to be imported by ordinal. This is the default.

inn Specifying "inn" causes imports for names in the non-resident names table to be imported by name.

Example:

```
wlib -iro -inn implib +dynamic.dll
```

Note that you must specify the "dll" file extension for the Dynamic Link Library. Otherwise an object file will be assumed.

17.11.6 Creating a Listing File - "l" Option

The "l" (lower case "L") option instructs the Open Watcom Library Manager to produce a list of the names of all symbols that can be found in the library file to a listing file. The file name of the listing file is the same as the file name of the library file. The file extension of the listing file is "lst".

Example:
```
wlib -l mylib
```

In the above example, the Open Watcom Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.lst".

An alternate form of this option is `-l=list_file`. With this form, you can specify the name of the listing file. When specifying a listing file name, a file extension of "lst" is assumed if none is specified.

Example:
```
wlib -l=mylib.out mylib
```

In the above example, the Open Watcom Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.out".

You can get a listing of the contents of a library file to the terminal by specifying only the library name on the command line as demonstrated by the following example.

Example:
```
wlib mylib
```

17.11.7 Display C++ Mangled Names - "m" Option

The "m" option instructs the Open Watcom Library Manager to display C++ mangled names rather than displaying their demangled form. The default is to interpret mangled C++ names and display them in a somewhat more intelligible form.

17.11.8 Always Create a New Library - "n" Option

The "n" option tells the Open Watcom Library Manager to always create a new library file. If the library file already exists, a backup copy is made (unless the "b" option was specified). The original contents of the library are discarded and a new library is created. If the "n" option was not specified, the existing library would be updated.

Example:
```
wlib -n mylib +myobj
```

In the above example, a library file called "mylib.lib" is created. It will contain a single object module, namely "myobj", regardless of the contents of "mylib.lib" prior to issuing the above command. If "mylib.lib" already exists, it will be renamed to "mylib.bak".

17.11.9 Specifying an Output File Name - "o" Option

The "o" option can be used to specify the output library file name if you want the original library to remain unchanged and a new library created.

Example:
```
wlib -o=newlib lib1 +lib2.lib
```

In the above example, the modules from "lib1.lib" and "lib2.lib" are added to the library "newlib.lib". Note that since the original library remains unchanged, no backup copy is created. Also, if the "l" option is used to specify a listing file, the listing file will assume the file name of the output library.

17.11.10 Specifying a Library Record Size - "p" Option

The "p" option specifies the record size in bytes for each record in the library file. The record size must be a power of 2 and in the range 16 to 32768. If the record size is less than 16, it will be rounded up to 16. If the record size is greater than 16 and not a power of 2, it will be rounded up to the nearest power of 2. The default record size is 256 bytes.

Each entry in the dictionary of a library file contains an offset from the start of the file which points to a module. The offset is 16 bits and is a multiple of the record size. Since the default record size is 256, the maximum size of a library file for a record size of 256 is 256*64K. If the size of the library file increases beyond this size, you must increase the record size.

Example:
```
wlib -p=512 lib1 +lib2.lib
```

In the above example, the Open Watcom Library Manager is instructed to create/update the library file "lib1.lib" by adding the modules from the library file "lib2.lib". The record size of the resulting library file is 512 bytes.

17.11.11 Operate Quietly - "q" Option

The "q" option suppressing the banner and copyright notice that is normally displayed when the Open Watcom Library Manager is invoked.

Example:
```
wlib -q -l mylib
```

17.11.12 Strip Line Number Records - "s" Option

The "s" option tells the Open Watcom Library Manager to remove line number records from object files that are being added to a library. Line number records are generated in the object file if the "d1" option is specified when compiling the source code.

Example:
```
wlib -s mylib +myobj
```

17.11.13 Trim Module Name - "t" Option

The "t" option tells the Open Watcom Library Manager to remove path information from the module name specified in THEADR records in object files that are being added to a library. The module name is created from the file name by the compiler and placed in the THEADR record of the object file. The module name will contain path information if the file name given to the compiler contains path information.

Example:
```
wlib -t mylib +myobj
```

17.11.14 Operate Verbosely - "v" Option

The "v" option enables the display of the banner and copyright notice when the Open Watcom Library Manager is invoked.

Example:
```
wlib -v -l mylib
```

17.11.15 Explode Library File - "x" Option

The "x" option tells the Open Watcom Library Manager to extract all modules from the library. Note that the modules are not deleted from the library. Object modules will be placed in the current directory unless the "d" option is used to specify an alternate directory.

In the following example all modules will be extracted from the library "mylib.lib" and placed in the current directory.

Example:
```
wlib -x mylib
```

In the following example, all modules will be extracted from the library "mylib.lib". The module will be placed in the file "/o" directory.

Example:
```
wlib -x -d=/o mylib
```

17.12 Librarian Error Messages

The following messages may be issued by the Open Watcom Library Manager.

Error! Could not open object file '%s'.

Object file '%s' could not be found. This message is usually issued when an attempt is made to add a non-existent object file to the library.

Error! Could not open library file '%s'.

The specified library file could not be found. This is usually issued for input library files. For example, if you are combining two library files, the library file you are adding is an input library file and the library file you are adding to or creating is an output library file.

Error! Invalid object module in file '%s' not added.

The specified file contains an invalid object module.

Error! Dictionary too large. Recommend split library into two libraries.

The size of the dictionary in a library file cannot exceed 64K. You must split the library file into two separate library files.

Error! Redefinition of module '%s' in file '%s'.

This message is usually issued when an attempt is made to add a module to a library that already contains a module by that name.

Warning! Redefinition of symbol '%s' in file '%s' ignored.

This message is issued if a symbol defined by a module already in the library is also defined by a module being added to the library.

Error! Library too large. Recommend split library into two libraries or try a larger page_bound than %xH.

The record size of the library file does not allow the library file to increase beyond its current size. The record size of the library file must be increased using the "p" option.

Error! Expected '%s' in '%s' but found '%s'.

An error occurred while scanning command input.

Warning! Could not find module '%s' for deletion.

This message is issued if an attempt is made to delete a module that does not exist in the library.

Error! Could not find module '%s' for extraction.

This message is issued if an attempt is made to extract a module that does not exist in the library.

Error! Could not rename old library for backup.

The Open Watcom Library Manager creates a backup copy before making any changes (unless the "b" option is specified). This message is issued if an error occurred while trying to rename the original library file to the backup file name.

Warning! Could not open library '%s' : will be created.

The specified library does not exist. It is usually issued when you are adding to a non-existent library. The Open Watcom Library Manager will create the library.

Warning! Output library name specification ignored.

This message is issued if the library file specified by the "o" option could not be opened.

Warning! Could not open library '%s' and no operations specified: will not be created.

This message is issued if the library file specified on the command line does not exist and no operations were specified. For example, asking for a listing file of a non-existent library will cause this message to be issued.

Warning! Could not open listing file '%s'.
The listing file could not be opened. For example, this message will be issued when a "disk full" condition is present.

Error! Could not open output library.
The output library could not be opened.

Error! Unable to write to output library.
An error occurred while writing to the output library.

Error! Unable to write to extraction file '%s'.
This message is issued when extracting an object module from a library file and an error occurs while writing to the output file.

Error! Out of Memory.
There was not enough memory to process the library file.

Error! Could not open file '%s'.
This message is issued if the output file for a module that is being extracted from a library could not be opened.

Error! Library '%s' is invalid. Contents ignored.
The library file does not contain the correct header information.

Error! Library '%s' has an invalid page size. Contents ignored.
The library file has an invalid record size. The record size is contained in the library header and must be a power of 2.

Error! Invalid object record found in file '%s'.
The specified file contains an invalid object record.

Error! No library specified on command line.
This message is issued if a library file name is not specified on the command line.

Error! Expecting library name.
This message is issued if the location of the library file name on the command line is incorrect.

Warning! Invalid file name '%s'.
This message is issued if an invalid file name is specified. For example, a file name longer that 127 characters is not allowed.

Error! Could not open command file '%s'.

The specified command file could not be opened.

Error! Could not read from file '%s'. Contents ignored as command input.

An error occurred while reading a command file.

The Open Watcom Assembler

18 The Open Watcom Assembler

18.1 Introduction

This chapter describes the Open Watcom Assembler. It takes as input an assembler source file (a file with extension ".a") and produces, as output, an object file.

The Open Watcom Assembler command line syntax is the following.

wasm [options] asm_file [options] [@env_var]

The square brackets [ ] denote items which are optional.

wasm is the name of the Open Watcom Assembler.

asm_file is the filename specification of the assembler source file to be assembled. A default filename extension of ".a" is assumed when no extension is specified. A filename extension consists of that portion of a filename containing the last "." and any characters which follow it.

Example:

```
File Specification                Extension
/home/john.doe/foo                (none)
/home/john.doe/foo.               .
/home/john.doe/foo.bar            .bar
/home/john.doe/foo.goo.bar        .bar
```

options is a list of valid Open Watcom Assembler options, each preceded by a dash (";.ct .sf7 -;.esf "). Options may be specified in any order.

The options supported by the Open Watcom Assembler are:

{0,1,2,3,4,5,6}{p}{r,s}

0	same as ".8086"
1	same as ".186"
2{p}	same as ".286" or ".286p"
3{p}	same as ".386" or ".386p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
4{p}	same as ".486" or ".486p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
5{p}	same as ".586" or ".586p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
6{p}	same as ".686" or ".686p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
p	protect mode
add r	defines "__REGISTER__"
add s	defines "__STACK__"

Example:

```
    -2      -3p      -4pr     -5p
```

bt=<os> defines "__<os>__" and checks the "<os>_INCLUDE" environment variable for include files

c do not output OMF COMENT records that allow WDISASM to figure out when data bytes have been placed in a code segment

d<name>[=text] define text macro

d1 line number debugging support

e stop reading assembler source file at END directive. Normally, anything following the END directive will cause an error.

e<number> set error limit number

fe=<file_name> set error file name

fo=<file_name> set object file name

fi=<file_name> force <file_name> to be included

fpc same as ".no87"

fpi inline 80x87 instructions with emulation

fpi87 inline 80x87 instructions

fp0 same as ".8087"

fp2 same as ".287" or ".287p"

fp3 same as ".387" or ".387p"

fp5 same as ".587" or ".587p"

fp6 same as ".687" or ".687p"

i=<directory> add directory to list of include directories

j or s force signed types to be used for signed values

m{t,s,m,c,l,h,f} memory model: (Tiny, Small, Medium, Compact, Large, Huge, Flat)

-mt	Same as ".model tiny"
-ms	Same as ".model small"
-mm	Same as ".model medium"
-mc	Same as ".model compact"
-ml	Same as ".model large"
-mh	Same as ".model huge"
-mf	Same as ".model flat"

Each of the model directives also defines "__<model>__" (e.g., ".model small" defines "__SMALL__"). They also affect whether something like "foo proc" is considered a "far" or "near" procedure.

nd=<name> set data segment name
nm=<name> set module name
nt=<name> set name of text segment
o allow C form of octal constants
zcm set C name mangler to MASM compatible mode
zld remove file dependency information
zq or q operate quietly
zz remove "@size" from STDCALL function names
zzo don't mangle STDCALL symbols (WASM backward compatible)
? or h print this message
w<number> set warning level number
we treat all warnings as errors
wx set warning level to maximum setting

18.2 Assembly Directives and Opcodes

It is not the intention of this chapter to describe assembly-language programming in any detail. You should consult a book that deals with this topic. However, we present an alphabetically ordered list of the directives, opcodes and register names that are recognized by the assembler.

```
.186         .286         .286c        .286p        .287
.386
.386p        .387         .486         .486p        .586
.586p
.686         .686p        .8086        .8087        aaa
aad
aam          aas          abs          adc          add
addpd
addps        addr         addsd        addss
addsubpd     addsubps
ah           al           alias        align        .alpha
and
andnpd       andnps       andpd        andps        arpl
assume
ax           basic        bh           bl           bound
bp
.break       bsf          bsr          bswap        bt
btc
btr          bts          bx           byte         c
call
callf        casemap      catstr       cbw          cdq
ch
cl           clc          cld          clflush      cli
clts
cmc          cmova        cmovae       cmovb        cmovbe
cmovc
cmove        cmovg        cmovge       cmovl        cmovle
cmovna
cmovnae      cmovnb       cmovnbe      cmovnc       cmovne
cmovng
cmovnge      cmovnl       cmovnle      cmovno       cmovnp
cmovns
cmovnz       cmovo        cmovp        cmovpe       cmovpo
cmovs
cmovz        cmp          cmpeqpd      cmpeqps
cmpeqsd      cmpeqss
cmplepd      cmpleps      cmplesd      cmpless
cmpltpd      cmpltps
cmpltsd      cmpltss      cmpneqpd     cmpneqps
cmpneqsd     cmpneqss
cmpnlepd     cmpnleps     cmpnlesd     cmpnless
cmpnltpd     cmpnltps
cmpnltsd     cmpnltss     cmpordpd     cmpordps
cmpordsd     cmpordss
cmppd        cmpps        cmps         cmpsb        cmpsd
cmpss
cmpsw        cmpunordpd   cmpunordps   cmpunordsd
```

```
cmpunordss   cmpxchg
cmpxchg8b    .code        comisd       comiss       comm
comment
common       compact      .const       .continue    cpuid
cr0
cr2          cr3          cr4          .cref        cs
cvtdq2pd
cvtdq2ps     cvtpd2dq     cvtpd2pi     cvtpd2ps
cvtpi2pd     cvtpi2ps
cvtps2dq     cvtps2pd     cvtps2pi     cvtsd2si
cvtsd2ss     cvtsi2sd
cvtsi2ss     cvtss2sd     cvtss2si     cvttpd2dq
cvttpd2pi    cvttps2dq
cvttps2pi    cvttsd2si    cvttss2si    cwd          cwde
cx
daa          das          .data        .data?       db
dd
dec          df           dh           di           div
divpd
divps        divsd        divss        dl
.dosseg      dosseg
dp           dq           dr0          dr1          dr2
dr3
dr6          dr7          ds           dt           dup
dw
```

```
dword        dx           eax          ebp          ebx
echo
ecx          edi          edx          .else        else
elseif
emms         end          .endif       endif        endm
endp
ends         .endw        enter        eq           equ
equ2
.err         .errb        .errdef      .errdif
.errdifi     .erre
.erridn      .erridni     .errnb       .errndef     .errnz
error
es           esi          esp          even         .exit
exitm
export       extern       externdef    extrn        f2xm1
fabs
fadd         faddp        far          .fardata
.fardata?    farstack
fbld         fbstp        fchs         fclex        fcmovb
fcmovbe
fcmove       fcmovnb      fcmovnbe     fcmovne
fcmovnu      fcmovu
fcom         fcomi        fcomip       fcomp        fcompp
fcos
fdecstp      fdisi        fdiv         fdivp        fdivr
fdivrp
femms        feni         ffree        fiadd        ficom
ficomp
fidiv        fidivr       fild         fimul
fincstp      finit
fist         fistp        fisttp       fisub        fisubr
flat
fld          fld1         fldcw        fldenv
fldenvd      fldenvw
fldl2e       fldl2t       fldlg2       fldln2       fldpi
fldz
fmul         fmulp        fnclex       fndisi       fneni
fninit
fnop         fnrstor      fnrstord     fnrstorw     fnsave
fnsaved
fnsavew      fnstcw       fnstenv      fnstenvd
fnstenvw     fnstsw
for          forc         fortran      fpatan       fprem
fprem1
fptan        frndint      frstor       frstord
frstorw      fs
fsave        fsaved       fsavew       fscale       fsetpm
fsin
```

```
fsincos      fsqrt        fst          fstcw        fstenv
fstenvd
fstenvw      fstp         fstsw        fsub         fsubp
fsubr
fsubrp       ftst         fucom        fucomi
fucomip      fucomp
fucompp      fwait        fword        fxam         fxch
fxrstor
fxsave       fxtract      fyl2x        fyl2xp1      ge
global
group        gs           gt           haddpd       haddps
high
highword     hlt          hsubpd       hsubps       huge
idiv
.if          if           if1          if2          ifb
ifdef
ifdif        ifdifi       ife          ifidn        ifidni
ifnb
ifndef       ignore       imul         in           inc
include
includelib   ins          insb         insd         insw
int
into         invd         invlpg       invoke       iret
iretd
```

```
iretdf        iretf         irp           irpc          ja
jae
jb            jbe           jc            jcxz          je
jecxz
jg            jge           jl            jle           jmp
jmpf
jna           jnae          jnb           jnbe          jnc
jne
jng           jnge          jnl           jnle          jno
jnp
jns           jnz           jo            jp            jpe
jpo
js            jz            .k3d          label         lahf
lar
large         lddqu         ldmxcsr       lds           le
lea
leave         length        lengthof      les
.lfcond       lfence
lfs           lgdt          lgs           lidt          .list
.listall
.listif       .listmacro    .listmacroall lldt          lmsw
local
lock          lods          lodsb         lodsd         lodsw
loop
loopd         loope         looped        loopew        loopne
loopned
loopnew       loopnz        loopnzd       loopnzw       loopw
loopz
loopzd        loopzw        low           lowword
lroffset      lsl
lss           lt            ltr           macro         mask
maskmovdqu
maskmovq      maxpd         maxps         maxsd         maxss
medium
memory        mfence        minpd         minps         minsd
minss
mm0           mm1           mm2           mm3           mm4
mm5
mm6           mm7           .mmx          mod           .model
monitor
mov           movapd        movaps        movd
movddup       movdq2q
movdqa        movdqu        movhlps       movhpd        movhps
movlhps
movlpd        movlps        movmskpd      movmskps
movntdq       movnti
movntpd       movntps       movntq        movq
movq2dq       movs
```

```
movsb       movsd       movshdup    movsldup    movss
movsw
movsx       movupd      movups      movzx       mul
mulpd
mulps       mulsd       mulss       mwait       name
ne
near        nearstack   neg         .no87
.nocref     .nolist
nop         not         nothing     offset      opattr
option
or          org         orpd        orps        os_dos
os_os2
out         outs        outsb       outsd       outsw
oword
packssdw    packsswb    packuswb    paddb       paddd
paddq
paddsb      paddsw      paddusb     paddusw     paddw
page
pand        pandn       para        pascal      pause
pavgb
pavgusb     pavgw       pcmpeqb     pcmpeqd
pcmpeqw     pcmpgtb
pcmpgtd     pcmpgtw     pextrw      pf2id       pf2iw
pfacc
```

```
pfadd        pfcmpeq      pfcmpge      pfcmpgt      pfmax
pfmin
pfmul        pfnacc       pfpnacc      pfrcp
pfrcpit1     pfrcpit2
pfrsqit1     pfrsqrt      pfsub        pfsubr       pi2fd
pi2fw
pinsrw       pmaddwd      pmaxsw       pmaxub       pminsw
pminub
pmovmskb     pmulhrw      pmulhuw      pmulhw       pmullw
pmuludq
pop          popa         popad        popcontext   popf
popfd
por          prefetch     prefetchnta  prefetcht0
prefetcht1   prefetcht2
prefetchw    private      proc         proto        psadbw
pshufd
pshufhw      pshuflw      pshufw       pslld        pslldq
psllq
psllw        psrad        psraw        psrld        psrldq
psrlq
psrlw        psubb        psubd        psubq        psubsb
psubsw
psubusb      psubusw      psubw        pswapd       ptr
public
punpckhbw    punpckhdq    punpckhqdq   punpckhwd
punpcklbw    punpckldq
punpcklqdq   punpcklwd    purge        push         pusha
pushad
pushcontext  pushd        pushf        pushfd       pushw
pword
pxor         qword        .radix       rcl          rcpps
rcpss
rcr          rdmsr        rdpmc        rdtsc
readonly     record
rep          repe         .repeat      repeat       repne
repnz
rept         repz         ret          retd         retf
retfd
retn         rol          ror          rsm
rsqrtps      rsqrtss
sahf         sal          .sall        sar          sbb
sbyte
scas         scasb        scasd        scasw        sdword
seg
segment      .seq         seta         setae        setb
setbe
setc         sete         setg         setge        setl
setle
```

setna	setnae	setnb	setnbe	setnc
setne				
setng	setnge	setnl	setnle	setno
setnp				
setns	setnz	seto	setp	setpe
setpo				
sets	setz	.sfcond	sfence	sgdt
shl				
shld	short	shr	shrd	shufpd
shufps				
si	sidt	size	sizeof	sldt
small				
smsw	sp	sqrtpd	sqrtps	sqrtsd
sqrtss				
ss	st	.stack	.startup	stc
std				
stdcall	sti	stmxcsr	stos	stosb
stosd				
stosw	str	struc	struct	sub
subpd				
subps	subsd	subss	subtitle	subttl
sword				
syscall	sysenter	sysexit	tbyte	test
textequ				
.tfcond	this	tiny	title	tr3
tr4				
tr5	tr6	tr7	typedef	
ucomisd	ucomiss			
union	unpckhpd	unpckhps	unpcklpd	
unpcklps	.until			
use16	use32	uses	vararg	verr
verw				
wait	watcom_c	wbinvd	.while	width
word				
wrmsr	xadd	xchg	.xcref	xlat
xlatb				
.xlist	.xmm	xmm0	xmm1	.xmm2
xmm2				
.xmm3	xmm3	xmm4	xmm5	xmm6
xmm7				
xor	xorpd	xorps		

18.3 Unsupported Directives

Other assemblers support directives that this assembler does not. The following is a list of directives that are ignored by the Open Watcom Assembler (use of these directives results in a warning message).

```
.alpha          .cref           .lfcond         .list
.listall        .listif         .listmacro      .listmacroall
.nocref         .nolist         page            .sall
.seq            .sfcond         subtitle        subttl
.tfcond         title           .xcref          .xlist
```

The following is a list of directives that are flagged by the Open Watcom Assembler (use of these directives results in an error message).

```
addr            .break          casemap         catstr
.continue       echo            .else           endmacro
.endif          .endw           .exit           high
highword        .if             invoke          low
lowword         lroffset        mask            opattr
option          popcontext      proto           purge
pushcontext     .radix          record          .repeat
.startup        this            typedef         union
.until          .while          width
```

18.4 Open Watcom Assembler Specific

There are a few specific features in Open Watcom Assembler

18.4.1 Naming convention

```
                  Procedure   Variable
Convention          Name        Name
---------------  ----------  ---------
C                   '*'         '*'
C (MASM)           '_*'        '_*'     see note 1
WATCOM_C           '*_'        '_*'
SYSCALL             '*'         '*'
STDCALL           '_*@nn'      '_*'
STDCALL            '_*'        '_*'     see note 2
STDCALL             '*'         '*'     see note 3
BASIC               '^'         '^'
FORTRAN             '^'         '^'
PASCAL              '^'         '^'
```

Notes:

1. WASM uses MASM compatible names when -zcm command line option is used.

2. In STDCALL procedures name 'nn' is overall parametrs size in bytes. '@nn' is suppressed when -zz command line option is used (WATCOM 10.0 compatibility).

3. STDCALL symbols mangling is suppressed by -zzo command line option (WASM backward compatible).

18.4.2 Open Watcom "C" name mangler

```
Command line      Procedure     Others
  option            Name        Names
---------------  ----------  ---------
0,1,2              '*_'        '_*'
3,4,5,6 with r     '*_'        '_*'
3,4,5,6 with s      '*'        '*'
```

18.4.3 Calling convention

Convention	Vararg	Parameters passed by	Parameters order	Cleanup caller stack
C	yes	stack	right to left	no
WATCOM_C	yes	registers	right to left	no
SYSCALL	yes	stack	right to left	no
STDCALL	yes	stack	right to left	yes see note 1
BASIC	no	stack	left to right	yes
FORTRAN	no	stack	left to right	yes
PASCAL	no	stack	left to right	yes

Notes:

1. For STDCALL procedures WASM automaticaly cleanup caller stack, except case when vararg parameter is used.

18.5 Open Watcom Assembler Diagnostic Messages

1 Size doesn't match with previous definition

2 Invalid instruction with current CPU setting

3 LOCK prefix is not allowed on this instruction

4 REP prefix is not allowed on this instruction

5 Invalid memory pointer

6 Cannot use 386 addressing mode with current CPU setting

7 Too many base registers

8 Invalid index register

9 Scale factor must be 1, 2, 4 or 8

10 invalid addressing mode with current CPU setting

11 ESP cannot be used as index

12 Too many base/index registers

13 Memory offset cannot reference to more than one label

14 Offset must be relocatable

15 Memory offset expected

16 Invalid indirect memory operand

17 Cannot mix 16 and 32-bit registers

18 CPU type already set

19 Unknown directive

20 Expecting comma

21 Expecting number

22 Invalid label definition

23 Invalid use of SHORT, NEAR, FAR operator

24 No memory

25 Cannot use 386 segment register with current CPU setting

26 POP CS is not allowed

27 Cannot use 386 register with current CPU setting

28 Only MOV can use special register

29 Cannot use TR3, TR4, TR5 in current CPU setting

30 Cannot use SHORT with CALL

31 Only SHORT displacement is allowed

32 Syntax error

33 Prefix must be followed by an instruction

34 No size given before 'PTR' operator

35 Invalid IMUL format

36 Invalid SHLD/SHRD format

37 Too many commas

38 Syntax error: Unexpected colon

39 Operands must be the same size

40 Invalid instruction operands

41 Immediate constant too large

42 Can not use short or near modifiers with this instruction

43 Jump out of range

44 Displacement cannot be larger than 32k

45 Initializer value too large

46 Symbol already defined

47 Immediate data too large

48 Immediate data out of range

49 Can not transfer control to stack symbol

50 Offset cannot be smaller than WORD size

51 Can not take offset of stack symbol

52 Can not take segment of stack symbol

53 Segment too large

54 Offset cannot be larger than 32k

55 Operand 2 too big

56 Operand 1 too small

57 Too many arithmetic operators

58 Too many open square brackets

59 Too many close square brackets

60 Too many open brackets

61 Too many close brackets

62 Invalid number digit

63 Assembler Code is too long

64 Brackets are not balanced

65 Operator is expected

66 Operand is expected

67 Too many tokens in a line

68 Bracket is expected

69 Illegal use of register

70 Illegal use of label

71 Invalid operand in addition

72 Invalid operand in subtraction

73 One operand must be constant

74 Constant operand is expected

75 A constant operand is expected in addition

76 A constant operand is expected in subtraction

77 A constant operand is expected in multiplication

78 A constant operand is expected in division

79 A constant operand is expected after a positive sign

80 A constant operand is expected after a negative sign

81 Label is not defined

82 More than one override

83 Label is expected

84 Only segment or group label is allowed

85 Only register or label is expected in override

86 Unexpected end of file

87 Label is too long

88 This feature has not been implemented yet

89 Internal Error #1

90 Can not take offset of group

91 Can not take offset of segment

92 Invalid character found

93 Invalid operand size for instruction

94 This instruction is not supported

95 size not specified -- BYTE PTR is assumed

96 size not specified -- WORD PTR is assumed

97 size not specified -- DWORD PTR is assumed

500 Segment parameter is defined already

501 Model parameter is defined already

502 Syntax error in segment definition

503 'AT' is not supported in segment definition

504 Segment definition is changed

505 Lname is too long

506 Block nesting error

507 Ends a segment which is not opened

508 Segment option is undefined

509 Model option is undefined

510 No segment is currently opened

511 Lname is used already

512 Segment is not defined

513 Public is not defined

514 Colon is expected

515 A token is expected after colon

516 Invalid qualified type

517 Qualified type is expected

518 External definition different from previous one

519 Memory model is not found in .MODEL

520 Cannot open include file

521 Name is used already

522 Library name is missing

523 Segment name is missing

524 Group name is missing

525 Data emitted with no segment

526 Seglocation is expected

527 Invalid register

528 Cannot address with assumed register

529 Invalid start address

530 Label is already defined

531 Token is too long

532 The line is too long after expansion

533 A label is expected after colon

534 Must be associated with code

535 Procedure must have a name

536 Procedure is alreadly defined

537 Language type must be specified

538 End of procedure is not found

539 Local variable must immediately follow PROC or MACRO statement

540 Extra character found

541 Cannot nest procedures

542 No procedure is currently defined

543 Procedure name does not match

544 Vararg requires C calling convention

545 Model declared already

546 Model is not declared

547 Backquote expected

548 COMMENT delimiter expected

549 End directive required at end of file

550 Nesting level too deep

551 Symbol not defined

552 Insert Stupid warning #1 here

553 Insert Stupid warning #2 here

554 Spaces not allowed in command line options

555 Error:

556 Source File

557 No filename specified.

558 Out of Memory

559 Cannot Open File -

560 Cannot Close File -

561 Cannot Get Start of Source File -

562 Cannot Set to Start of Source File -

563 Command Line Contains More Than 1 File To Assemble

564 include path %s.

565 Unknown option %s. Use /? for list of options.

566 read more command line from %s.

567 Internal error in %s(%u)

568 OBJECT WRITE ERROR !!

569 NO LOR PHARLAP !!

570 Parameter Required

571 Expecting closing square bracket

572 Expecting file name

573 Floating point instruction not allowed with /fpc

574 Too many errors

575 Build target not recognised

576 Public constants should be numeric

577 Expecting symbol

578 Do not mix simplified and full segment definitions

579 Parms passed in multiple registers must be accessed separately, use %s

580 Ten byte variables not supported in register calling convention

581 Parameter type not recognised

582 forced error:

583 forced error: Value not equal to 0 : %d

584 forced error: Value equal to 0: %d

585 forced error: symbol defined: %s

586 forced error: symbol not defined: %s

587 forced error: string blank : <%s>

588 forced error: string not blank : <%s>

589 forced error: strings not equal : <%s> : <%s>

590 forced error: strings equal : <%s> : <%s>

591 included by file %s(%d)

592 macro called from file %s(%d)

593 Symbol %s not defined

594 Extending jump

595 Ignoring inapplicable directive

596 Unknown symbol class '%s'

597 Symbol class for '%s' already established

598 number must be a power of 2

599 alignment request greater than segment alignment

600 '%s' is already defined

601 %u unclosed conditional directive(s) detected

The Open Watcom Disassembler

19 The Object File Disassembler

19.1 Introduction

This chapter describes the Open Watcom Disassembler. It takes as input an object file (a file with extension ".o") and produces, as output, the Intel assembly language equivalent. The Open Watcom compilers do not produce an assembly language listing directly from a source program. Instead, the Open Watcom Disassembler can be used to generate an assembly language listing from the object file generated by the compiler.

The Open Watcom Disassembler command line syntax is the following.

wdis [options] filespec [options]

The square brackets [ ] denote items which are optional.

wdis is the name of the Open Watcom Disassembler.

filespec is the filename specification of the object file to be disassembled. A default filename extension of ".o" is assumed when no extension is specified. A filename extension consists of that portion of a filename containing the last "." and any characters which follow it.

Example:

```
File Specification              Extension
/home/john.doe/foo              (none)
/home/john.doe/foo.             .
/home/john.doe/foo.bar          .bar
/home/john.doe/foo.goo.bar      .bar
```

options is a list of valid Open Watcom Disassembler options, each preceded by a dash (";.ct .sf7 -;.esf "). Options may be specified in any order.

The options supported by the Open Watcom Disassembler are:

a write assembly instructions only to the listing file
e include list of external names
fp do not use instruction name pseudonyms
fr do not use register name pseudonyms [Alpha only]
fi use alternate indexing format [80(x)86 only]
fu instructions/registers in upper case
i=<char> redefine the initial character of internal labels (default: L)
l[=<list_file>]
create a listing file
m leave C++ names mangled
o print list of operands beside instructions
p include list of public names
s[=<source_file>]
using object file source line information, imbed original source lines into the output file

The following sections describe the list of options.

19.2 Changing the Internal Label Character - "i=<char>"

The "i" option permits you to specify the first character to be used for internal labels. Internal labels take the form "Ln" where "n" is one or more digits. The default character "L" can be changed using the "i" option. The replacement character must be a letter (a-z, A-Z). A lowercase letter is converted to uppercase.

Example:
```
$ wdis calendar -i=x
```

19.3 The Assembly Format Option - "a"

The "a" option controls the format of the output produced to the listing file. When specified, the Open Watcom Disassembler will produce a listing file that can be used as input to an assembler.

Example:
```
$ wdis calendar -a -l=calendar.asm
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.o` and produce the output to the file `calendar.asm` so that it can be assembled by an assembler.

19.4 The External Symbols Option - "e"

The "e" option controls the amount of information produced in the listing file. When specified, a list of all externally defined symbols is produced in the listing file.

Example:
```
$ wdis calendar -e
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.o` and produce the output, with a list of all external symbols, on the screen. A sample list of external symbols is shown below.

```
List of external symbols

Symbol
----------------
___iob           0000032f 00000210 000001f4 00000158 00000139
__CHK            00000381 00000343 000002eb 00000237 000000cb 00000006
Box_             000000f2
Calendar_        000000a7 00000079 00000049
ClearScreen_     00000016
fflush_          00000334 00000215 000001f9 0000015d 0000013e
int386_          000003af 00000372
Line_            000002db 000002b5 00000293 00000274 0000025a
localtime_       00000028
memset_          00000308
PosCursor_       0000031e 000001e1 00000148 00000123 000000b6
printf_          00000327 00000208 000001ec 00000150 00000131
strlen_          00000108
time_            0000001d
------------------------------------------------------------
```

Each externally defined symbol is followed by a list of location counter values indicating where the symbol is referenced.

The "e" option is ignored when the "a" option is specified.

19.5 The No Instruction Name Pseudonyms Option - "fp"

By default, AXP instruction name pseudonyms are emitted in place of actual instruction names. The Open Watcom AXP Assembler accepts instruction name pseudonyms. The "fp" option instructs the Open Watcom Disassembler to emit the actual instruction names instead.

19.6 The No Register Name Pseudonyms Option - "fr"

By default, AXP register names are emitted in pseudonym form. The Open Watcom AXP Assembler accepts register pseudonyms. The "fr" option instructs the Open Watcom Disassembler to display register names in their non-pseudonym form.

19.7 The Alternate Addressing Form Option - "fi"

The "fi" option causes an alternate syntactical form of the based or indexed addressing mode of the 80x86 to be used in an instruction. For example, the following form is used by default for Intel instructions.

```
mov ax,-2[bp]
```

If the "fi" option is specified, the following form is used.

```
mov ax,[bp-2]
```

19.8 The Uppercase Instructions/Registers Option - "fu"

The "fu" option instructs the Open Watcom Disassembler to display instruction and register names in uppercase characters. The default is to display them in lowercase characters.

19.9 The Listing Option - "l[=<list_file>]"

By default, the Open Watcom Disassembler produces its output to the terminal. The "l" (lowercase L) option instructs the Open Watcom Disassembler to produce the output to a listing file. The default file name of the listing file is the same as the file name of the object file. The default file extension of the listing file is `.lst`.

Example:
```
$ wdis calendar -l
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.o` and produce the output to a listing file called `calendar.lst`.

An alternate form of this option is "l=<list_file>". With this form, you can specify the name of the listing file. When specifying a listing file, a file extension of `.lst` is assumed if none is specified.

Example:
```
$ wdis calendar -l=calendar.lis
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.o` and produce the output to a listing file called `calendar.lis`.

19.10 The Public Symbols Option - "p"

The "p" option controls the amount of information produced in the listing file. When specified, a list of all public symbols is produced in the listing file.

Example:
```
$ wdis calendar -p
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.o` and produce the output, with a list of all exported symbols, to the screen. A sample list of public symbols is shown below.

The following is a list of public symbols in 80x86 code.

```
List of public symbols

SYMBOL                  SECTION                 OFFSET
--------------------------------------------------------
main_                   _TEXT                   000002C0
void near Box( int, int, int, int )
                        _TEXT                   00000093
void near Calendar( int, int, int, int, int, char near * )
                        _TEXT                   0000014A
void near ClearScreen() _TEXT                   00000000
void near Line( int, int, int, char, char, char )
                        _TEXT                   00000036
void near PosCursor( int, int )
                        _TEXT                   0000001A
```

The following is a list of public symbols in Alpha AXP code.

```
List of public symbols

SYMBOL                  SECTION                 OFFSET
--------------------------------------------------------
main                    .text                   000004F0
void near Box( int, int, int, int )
                        .text                   00000148
void near Calendar( int, int, int, int, int, char near * )
                        .text                   00000260
void near ClearScreen() .text                   00000000
void near Line( int, int, int, char, char, char )
                        .text                   00000060
void near PosCursor( int, int )
                        .text                   00000028
```

The "p" option is ignored when the "a" option is specified.

19.11 Retain C++ Mangled Names - "m"

The "m" option instructs the Open Watcom Disassembler to retain C++ mangled names rather than displaying their demangled form. The default is to interpret mangled C++ names and display them in a somewhat more intelligible form.

19.12 The Source Option - "s[=<source_file>]"

The "s" option causes the source lines corresponding to the assembly language instructions to be produced in the listing file. The object file must contain line numbering information. That is, the "d1" or "d2" option must have been specified when the source file was compiled. If no line numbering information is present in the object file, the "s" option is ignored.

The following defines the order in which the source file name is determined when the "s" option is specified.

1. If present, the source file name specified on the command line.
2. The name from the module header record.
3. The object file name.

In the following example, we have compiled the source file `mysrc.c` with "d1" debugging information. We then disassemble it as follows:

Example:
```
$ wdis mysrc -s -l
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `mysrc.o` and produce the output to the listing file `mysrc.lst.` The source lines are extracted from the file `mysrc.c.`

An alternate form of this option is "s=<source_file>". With this form, you can specify the name of the source file.

Example:
```
$ wdis mysrc -s=myprog.c -l
```

The above example produces the same result as in the previous example except the source lines are extracted from the file `myprog.c.`

19.13 An Example

Consider the following program contained in the file `hello.c.`

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

Compile it with the "d1" option. An object file called `hello.o` will be produced. The "d1" option causes line numbering information to be generated in the object file. We can use the Open Watcom Disassembler to disassemble the contents of the object file by issuing the following command.

```
$ wdis hello -l -e -p -s -fu
```

The output will be written to a listing file called `hello.lst` (the "l" option was specified"). It will contain a list of external symbols (the "e" option was specified), a list of public symbols (the "p" option was specified) and the source lines corresponding to the assembly language instructions (the "s" option was specified). The source input file is called `hello.c.` The register names will be displayed in upper case (the "fu" option was specified). The output, shown below, is the result of using the Open Watcom C++ compiler.

The following is a disassembly of 80x86 code.

```
Module: HELLO.C
GROUP: 'DGROUP' CONST,CONST2,_DATA,_BSS

Segment: _TEXT DWORD USE32 0000001A bytes

#include <stdio.h>

void main()
0000                    main_:
0000    68 08 00 00 00      PUSH      0x00000008
0005    E8 00 00 00 00      CALL      __CHK

{
    printf( "Hello world\n" );
000A    68 00 00 00 00      PUSH      offset L$1
000F    E8 00 00 00 00      CALL      printf_
0014    83 C4 04            ADD       ESP,0x00000004

}
0017    31 C0               XOR       EAX,EAX
0019    C3                  RET

Routine Size: 26 bytes,    Routine Base: _TEXT + 0000

No disassembly errors

List of external references

SYMBOL
-------
__CHK                   0006
printf_                 0010

Segment: CONST DWORD USE32 0000000D bytes
0000                    L$1:
0000    48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00          Hello world..



BSS Size: 0 bytes
```

```
List of public symbols

SYMBOL                   SECTION                   OFFSET
---------------------------------------------------------
main_                    _TEXT                     00000000
```

The following is a disassembly of Alpha AXP code.

```
                                 .new_section .text, "crx4"

#include <stdio.h>

void main()
0000                     main:
0000    23DEFFF0             LDA           SP,-0x10(SP)
0004    B75E0000             STQ           RA,(SP)

{
    printf( "Hello world\n" );
0008    261F0000             LDAH          A0,h^L$0(R31)
000C    22100000             LDA           A0,l^L$0(A0)
0010    43F00010             SEXTL         A0,A0
0014    D3400000             BSR           RA,j^printf

}
0018    201F0000             MOV           0x00000000,V0
001C    A75E0000             LDQ           RA,(SP)
0020    23DE0010             LDA           SP,0x10(SP)
0024    6BFA8001             RET           (RA)

Routine Size: 40 bytes,    Routine Base: .text + 0000

No disassembly errors
```

```
List of external references

SYMBOL
-------
printf                  0014

                                .new_section .const, "drw4"
0000                    L$0:
0000    48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00 Hello world.....

                                .new_section .const2, "drw4"

                                .new_section .data, "drw4"

                                .new_section .bss, "urw4"
0000                    .bss:

BSS Size: 0 bytes


                                .new_section .pdata, "dr2"

0000                    //  Procedure descriptor for main
        main                                    //  BeginAddress      : 0
        main+0x28                               //  EndAddress        : 40
        00000000                                //  ExceptionHandler  : 0
        00000000                                //  HandlerData       : 0
        main+0x8                                //  PrologEnd         : 8

                                .new_section .drectve, "iRr0"
0000    2D 64 65 66 61 75 6C 74 6C 69 62 3A 63 6C 69 62 -defaultlib:clib
0010    20 2D 64 65 66 61 75 6C 74 6C 69 62 3A 70 6C 69  -defaultlib:pli
0020    62 20 2D 64 65 66 61 75 6C 74 6C 69 62 3A 6D 61 b -defaultlib:ma
0030    74 68 20 00                                     th .

List of public symbols

SYMBOL                  SECTION                 OFFSET
--------------------------------------------------------
main                    .text                   00000000
```

Let us create a form of the listing file that can be used as input to an assembler.

```
$ wdis hello -l=hello.asm -r -a
```

The output will be produced in the file `hello.asm`. The output, shown below, is the result of using the Open Watcom C++ compiler.

The following is a disassembly of 80x86 code.

```
.387
.386p
                PUBLIC  main_
                EXTRN   __CHK:BYTE
                EXTRN   printf_:BYTE
                EXTRN   ___wcpp_3_data_init_fs_root_:BYTE
                EXTRN   _cstart_:BYTE
DGROUP          GROUP   CONST,CONST2,_DATA,_BSS
_TEXT           SEGMENT DWORD PUBLIC USE32 'CODE'
                ASSUME CS:_TEXT, DS:DGROUP, SS:DGROUP
main_:
    PUSH      0x00000008
    CALL      near ptr __CHK
    PUSH      offset L$1
    CALL      near ptr printf_
    ADD       ESP,0x00000004
    XOR       EAX,EAX
    RET
_TEXT           ENDS
CONST           SEGMENT DWORD PUBLIC USE32 'DATA'
L$1:
                DB      0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f
                DB      0x72, 0x6c, 0x64, 0x0a, 0x00

CONST           ENDS
CONST2          SEGMENT DWORD PUBLIC USE32 'DATA'
CONST2          ENDS
_DATA           SEGMENT DWORD PUBLIC USE32 'DATA'
_DATA           ENDS
_BSS            SEGMENT DWORD PUBLIC USE32 'BSS'
_BSS            ENDS

                END
```

The following is a disassembly of Alpha AXP code.

```
.globl          main
.extrn          printf
.extrn          _cstart_
.new_section .text, "crx4"
main:
    LDA           $SP,-0x10($SP)
    STQ           $RA,($SP)
    LDAH          $A0,h^`L$0`($ZERO)
    LDA           $A0,l^`L$0`($A0)
    SEXTL         $A0,$A0
    BSR           $RA,j^printf
    MOV           0x00000000,$V0
    LDQ           $RA,($SP)
    LDA           $SP,0x10($SP)
    RET           $ZERO,($RA),0x00000001
```

```
.new_section .const, "drw4"
`L$0`:
                .asciiz "Hello world\n"
                .byte   0x00, 0x00, 0x00



.new_section .pdata, "dr2"

                //  0000                Procedure descriptor for main
                .long   main                    //  BeginAddress      : 0
                .long   main+0x28               //  EndAddress        : 40
                .long   00000000                //  ExceptionHandler  : 0
                .long   00000000                //  HandlerData       : 0
                .long   main+0x8                //  PrologEnd         : 8

.new_section .drectve, "iRr0"
                .asciiz "-defaultlib:clib -defaultlib:plib
-defaultlib:math "
```

20 Optimization of Far Calls

Optimization of far calls can result in smaller executable files and improved performance. It is most useful when the automatic grouping of logical segments into physical segments takes place. Note that, by default, automatic grouping is performed by the Open Watcom Linker.

The Open Watcom C, C++ and FORTRAN 77 compilers automatically enable the far call optimization. The Open Watcom Linker will optimize far calls to procedures that reside in the same physical segment as the caller. For example, a large code model program will probably contain many far calls to procedures in the same physical segment. Since the segment address of the caller is the same as the segment address of the called procedure, only a near call is necessary. A near call does not require a relocation entry in the relocation table of the executable file whereas a far call does. Thus, the far call optimization will result in smaller executable files that will load faster. Furthermore, a near call will generally execute faster than a far call, particularly on 286 and 386-based machines where, for applications running in protected mode, segment switching is fairly expensive.

The following describes the far call optimization. The **call far label** instruction is converted to one of the following sequences of code.

```
push   cs                  seg    ss
call   near label         push   cs
nop                        call   near label
```

Notes:

1. The **nop** or **seg ss** instruction is present since a **call far label** instruction is five bytes. The **push cs** instruction is one byte and the **call near label** instruction is three bytes. The **seg ss** instruction is used because it is faster than the **nop** instruction.

2. The called procedure will still use a **retf** instruction but since the code segment and the near address are pushed on the stack, the far return will execute correctly.

3. The position of the padding instruction is chosen so that the return address is word aligned. A word aligned return address improves performance.

4. When two consecutive **call far label** instructions are optimized and the first **call far label** instruction is word aligned, the following sequence replaces both **call far label** instructions.

```
push    cs
call    near label1
seg     ss
push    cs
seg     cs
call    near label2
```

5. If your program contains only near calls, this optimization will have no effect.

A far jump optimization is also performed by the Open Watcom Linker. This has the same benefits as the far call optimization. A **jmp far label** instruction to a location in the same segment will be replaced by the following sequence of code.

```
jmp    near label
mov    ax,ax
```

Note that for 32-bit segments, this instruction becomes `mov eax,eax`.

The Open Watcom Strip Utility

21 The Open Watcom Strip Utility

21.1 Introduction

The Open Watcom Strip Utility may be used to manipulate information that is appended to the end of an executable file. The information can be either one of two things:

1. Symbolic debugging information
2. Resource information

This information can be added or removed from the executable file. Symbolic debugging information is placed at the end of an executable file by the Open Watcom Linker or the Open Watcom Strip Utility. Resource information is placed at the end of an executable by a resource compiler or the Open Watcom Strip Utility.

Once a program has been debugged, the Open Watcom Strip Utility allows you to remove the debugging information from the executable file so that you do not have to remove the debugging directives from the linker directive file and link your program again. Removal of the debugging information reduces the size of the executable image.

All executable files generated by the Open Watcom Linker can be specified as input to the Open Watcom Strip Utility.

21.2 The Open Watcom Strip Utility Command Line

The Open Watcom Strip Utility command line syntax is:

```
wstrip [options] input_file [output_file] [info_file]
```

where:

[] The square brackets denote items which are optional.

options

- **-n** (noerrors) Do not issue any diagnostic message.
- **-q** (quiet) Do not print any informational messages.
- **-r** (resources) Process resource information rather than debugging information.
- **-a** (add) Add information rather than remove information.

input_file is a file specification for the name of an executable file. If no file extension is specified, the Open Watcom Strip Utility will assume one of the following extensions: "exe", "dll", "exp", "rex", "nlm", "dsk", "lan", "nam", "msl", "cdm", "ham", "qnx" or no file extension. Note that the order specified in the list of file extensions is the order in which the Open Watcom Strip Utility will select file extensions.

output_file is an optional file specification for the output file. If no file extension is specified, the file extension specified in the input file name will be used for the output file name. If "." is specified, the input file name will be used.

info_file is an optional file specification for the file in which the debugging or resource information is to be stored (when removing information) or read (when adding information). If no file extension is specified, a file extension of "sym" is assumed for debugging information and "res" for resource information. To specify the name of the information file but not the name of an output file, a "." may be specified in place of *output_file.*

Description:

1. If the "r" (resource) option is not specified then the default action is to add/remove symbolic debugging information.
2. If the "a" (add) option is not specified then the default action is to remove information.
3. If *output_file* is not specified, the debugging or resource information is added to or removed from *input_file.*

4. If *output_file* is specified, *input_file* is copied to *output_file* and the debugging or resource information is added to or removed from *output_file*. *input_file* remains unchanged.

5. If *info_file* is specified then the debugging or resource information that is added to or removed from the executable file is read from or written to this file. The debugging or resource information may be appended to the executable by specifying the "a" (add) option. Also, the debugging information may be appended to the executable by concatenating the debugging information file to the end of the executable file (the files must be treated as binary files).

6. During processing, the Open Watcom Strip Utility will create a temporary file, ensuring that a file by the chosen name does not already exist.

21.3 Strip Utility Messages

The following messages may be issued by the Open Watcom Strip Utility.

Usage: wstrip [options] input_file [output_file] [info_file]
options: (-option is also accepted)
/n don't print warning messages
/q don't print informational messages
/r process resource information rather than debugging information
/a add information rather than delete information
input_file: executable file
output_file: optional output executable or '.'
info_file: optional output debugging or resource information file
or input debugging or resource informational file

The command line was entered with no arguments.

Too low on memory

There is not enough free memory to allocate file buffers.

Unable to find '%s'

The specified file could not be located.

Cannot create temporary file

All the temporary file names are in use.

Unable to open '%s' to read

The input executable file cannot be opened for reading.

'%s' is not a valid executable file
The input file has invalid executable file header information.

'%s' does not contain debugging information
There is nothing to strip from the specified executable file.

Seek error on '%s'
An error occurred during a seek operation on the specified file.

Unable to create output file '%s'
The output file could not be created. Check that the output disk is not write-protected or that the specified output file is not marked "read-only".

Unable to create symbol file '%s'
The symbol file could not be created.

Error reading '%s'
An error occurred while reading the input executable file.

Error writing to '%s'
An error occurred while writing the output executable file or the symbol file. Check the amount of free space on the output disk. If the input and output files reside on the same disk, there might not be enough room for a second copy of the executable file during processing.

Cannot erase file '%s'
The input executable file is probably marked "read-only" and therefore could not be erased (the input file is erased whenever the output file has the same name).

Cannot rename file '%s'
The output executable file could not be renamed. Ordinarily, this should never occur.

Appendices

A. Use of Environment Variables

In the Open Watcom C/C++ software development package, a number of environment variables are used. This appendix summarizes their use with a particular component of the package.

A.1 FORCE

The **FORCE** environment variable identifies a file that is to be included as part of the source input stream. This variable is used by Open Watcom C/C++.

```
export "FORCE=filespec"
```

The specified file is included as if a

```
#include "filespec"
```

directive were placed at the start of the source file.

Example:
```
$ export "FORCE=/usr/include/common.cnv"
$ wcc report
```

The **FORCE** environment variable can be overridden by use of the Open Watcom C/C++ "fi" option.

A.2 INCLUDE

The **INCLUDE** environment variable describes the location of the C and C++ header files (files with the ".h" filename extension). This variable is used by Open Watcom C/C++.

```
export "INCLUDE=path:path..."
```

The **INCLUDE** environment string is like the **PATH** string in that you can specify one or more directories separated by colons (":").

A.3 LIB

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

The **LIB** environment variable is used to select the libraries that will be used when the application is linked. This variable is used by the Open Watcom Linker (wlink). The **LIB** environment string is like the **PATH** string in that you can specify one or more directories separated by colons (":").

A.4 PATH

The **PATH** environment variable is used by the QNX shell to locate programs.

```
export "PATH=path:path..."
```

The **PATH** environment variable should include the directory of the Open Watcom C/C++ binary program files when using Open Watcom C/C++ and its related tools.

The default installation directory for Open Watcom C/C++ QNX binaries is called "/bin".

Example:
```
$ export "PATH=/bin"
```

The **PATH** environment variable is also used by the following programs in the described manner.

1. cc to locate the 16-bit Open Watcom C/C++ and 32-bit Open Watcom C/C++ compilers and the Open Watcom Linker.
2. "WD" to locate programs.

A.5 TMPDIR

The **TMPDIR** environment variable describes the location (path) for temporary files created by the 16-bit Open Watcom C/C++ and 32-bit Open Watcom C/C++ compilers and the Open Watcom Linker.

```
export "TMPDIR=path"
```

Normally, 16-bit Open Watcom C/C++ and 32-bit Open Watcom C/C++ will create temporary spill files in the current directory. However, by defining the **TMPDIR** environment variable to be a certain path, you can tell Open Watcom C/C++ where to place its temporary files. The same is true of the Open Watcom Linker temporary file.

Consider the following definition of the **TMPDIR** environment variable.

Example:
```
$ export "TMPDIR=//2/hd/tmp"
```

The Open Watcom C/C++ compiler and Open Watcom Linker will create its temporary files in `//2/hd/tmp`.

A.6 WATCOM

In order for the Open Watcom Linker to locate the 16-bit Open Watcom C/C++ and 32-bit Open Watcom C/C++ library files, the **WATCOM** environment variable should be defined. When using ***cc***, it is not necessary to define this environment variable since it uses another technique for identifying the location of the library files to the Open Watcom Linker. However, you should do so when you begin to use the Open Watcom Linker directly without the aid of this utility program. The **WATCOM** environment variable is used to locate the libraries that will be used when the application is linked. The default directory for 16-bit Open Watcom C/C++ and 32-bit Open Watcom C/C++ files is "/usr".

Example:
```
$ export "WATCOM=//0/hd/usr"
```

A.7 WCC

The **WCC** environment variable can be used to specify commonly-used options for the 16-bit C compiler.

```
export "WCC=-option1 -option2 ..."
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:
```
$ export "WCC=-d1 -ox"
```

Once the **WCC** environment variable has been defined, those options listed become the default each time the ***wcc*** command is used.

A.8 WCC386

The **WCC386** environment variable can be used to specify commonly-used options for the 32-bit C compiler.

```
export "WCC386=-option1 -option2 ..."
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:
```
$ export "WCC386=-d1 -ox"
```

Once the **WCC386** environment variable has been defined, those options listed become the default each time the ***wcc386*** command is used.

A.9 WCGMEMORY

The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:
```
$ export "WCGMEMORY=?"
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:
```
$ export "WCGMEMORY=128"
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure that the **WCGMEMORY** environment variable is set identically on both machines.

A.10 WD

The **WD** environment variable can be used to specify commonly-used Open Watcom Debugger options.

```
export "WD=-option1 -option2 ..."
```

These options are processed before options specified on the command line. The following example defines the default options to be "noinvoke" (do not execute the `profile.dbg` file) and "reg=10" (retain up to 10 register sets while tracing).

Example:
```
$ export "WD=-noinvoke -reg=10"
```

Once the **WD** environment variable has been defined, those options listed become the default each time the WD command is used.

A.11 WD_PATH

The **WD_PATH** environment variable is used by wd to locate Open Watcom Debugger support files. These files fall into five categories.

1. Open Watcom Debugger command files (files with the ".dbg" suffix).

2. Open Watcom Debugger trap files (files with the ".trp" suffix).

3. Open Watcom Debugger parser files (files with the ".prs" suffix).

4. Open Watcom Debugger help files (files with the ".hlp" suffix).

5. Open Watcom Debugger symbolic debugging information files (files with the ".sym" suffix).

```
export "WD_PATH=path:path..."
```

By default, Open Watcom Debugger looks in the `/usr/watcom/wd` directory for command files so it is not necessary to include this directory in the **WD_PATH** environment variable string.

A.12 WPP

The **WPP** environment variable can be used to specify commonly-used options for the 16-bit C++ compiler.

```
export "WPP=-option1 -option2 ..."
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:
```
$ export "WPP=-d1 -ox"
```

Once the **WPP** environment variable has been defined, those options listed become the default each time the ***wpp*** command is used.

A.13 WPP386

The **WPP386** environment variable can be used to specify commonly-used options for the 32-bit C++ compiler.

```
export "WPP386=-option1 -option2 ..."
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:
```
$ export "WPP386=-d1 -ox"
```

Once the **WPP386** environment variable has been defined, those options listed become the default each time the ***wpp386*** command is used.

B. Open Watcom C Diagnostic Messages

The following is a list of all warning and error messages produced by the Open Watcom C compilers. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%s`, `%d` and `%u`. They represent strings that are substituted by the Open Watcom C compilers to make the error message more exact. `%d` and `%u` represent a string of digits; `%s` a string, usually a symbolic name.

Consider the following program, named `err.c`, which contains errors.

Example:

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
err.c(6): Error! E1034: Symbol 'i' already defined
err.c(9): Error! E1011: Symbol 'x' has not been declared
err.c: 12 lines, included 191, 0 warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number of the line containing the error (in parentheses),
3. a message number, and
4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `err.c.` Error number 1034 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `err.c.` Error number 1011 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

B.1 Warning Level 1 Messages

W100 *Parameter %d contains inconsistent levels of indirection*

The function is expecting something like `char **` and it is being passed a `char *` for instance.

W101 *Non-portable pointer conversion*

This message is issued whenever you convert a non-zero constant to a pointer.

W102 *Type mismatch (warning)*

This message is issued for a function return value or an assignment where both types are pointers, but they are pointers to different kinds of objects.

W103 *Parameter count does not agree with previous definition (warning)*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you can ignore this warning, or you can change the function declaration and prototypes to use the ",..." to indicate that the function indeed takes a variable number of parameters.

W104 *Inconsistent levels of indirection*

This occurs in an assignment or return statement when one of the operands has more levels of indirection than the other operand. For example, a `char **` is being assigned to a `char *.`

Solution: Correct the levels of indirection or use a `void *.`

W105 *Assignment found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

W106 *Constant out of range - truncated*

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

W107 *Missing return value for function '%s'*

A function has been declared with a function return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

W108 *Duplicate typedef already defined*

A duplicate typedef is not allowed in ANSI C. This warning is issued when compiling with extensions enabled. You should delete the duplicate typedef definition.

W109 *not used*

unused message

W110 *'fortran' pragma not defined*

You have used the ***fortran*** keyword in your program, but have not defined a #pragma for ***fortran.***

W111 *Meaningless use of an expression*

The line contains an expression that does nothing useful. In the example "i = (1,5);", the expression "1," is meaningless.

W112 *Pointer truncated*

A far pointer is being passed to a function that is expecting a near pointer, or a far pointer is being assigned to a near pointer.

W113 *Pointer type mismatch*

You have two pointers that either point to different objects, or the pointers are of different size, or they have different modifiers.

W114 *Missing semicolon*

You are missing the semicolon ";" on the field definition just before the right curly brace "}".

W115 *&array may not produce intended result*

The type of the expression "&array" is different from the type of the expression "array". Suppose we have the declaration `char buffer[80]` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what most people expect to happen. The address of operator "&" is not required for getting the address of an array.

W116 *Attempt to return address of auto variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

W117 *'##' tokens did not generate a single token (rest discarded)*

When two tokens are pasted together using ##, they must form a string that can be parsed as a single token.

W118 *Label '%s' has been defined but not referenced*

You have defined a label that is not referenced in a ***goto*** statement. It is possible that you are missing the ***case*** keyword when using an enumerated type name as a case in a ***switch*** statement. If not, then the label can be deleted.

W119 *Address of static function '%s' has been taken*

This warning may indicate a potential problem when the program is overlayed.

W120 *lvalue cast is not standard C*

A cast operation does not yield an lvalue in ANSI standard C. However, to provide compatibility with code written prior to the availability of ANSI standard C compilers, if an expression was an lvalue prior to the cast operation, and the cast operation does not cause any conversions, the compiler treats the result as an lvalue and issues this warning.

W121 *Text following pre-processor directives is not standard C*

Arbitrary text is not allowed following a pre-processor directive. Only comments are allowed following a pre-processor directive.

W122 *Literal string too long for array - truncated*

The supplied literal string contains more characters than the specified dimension of the array. Either shorten the literal string, or increase the dimension of the array to hold all of the characters from the literal string.

W123 *'//' style comment continues on next line*

The compiler has detected a line continuation during the processing of a C++ style comment ("//"). The warning can be removed by switching to a C style comment ("/**/"). If you require the comment to be terminated at the end of the line, make sure that the backslash character is not the last character in the line.

Example:

```
#define XX 23 // comment start \
comment \
end

int x = XX; // comment start ...\
comment end
```

W124 *Comparison result always %d*

The line contains a comparison that is always true (1) or false (0). For example comparing an unsigned expression to see if it is >= 0 or < 0 is redundant. Check to see if the expression should be signed instead of unsigned.

W125 *Nested include depth of %d exceeded*

The number of nested include files has reached a preset limit, check for recursive include statements.

W126 *Constant must be zero for pointer compare*

A pointer is being compared using == or != to a non-zero constant.

W127 *trigraph found in string*

Trigraph expansion occurs inside a string literal. This warning can be disabled via the command line or ***#pragma warning*** directive.

Example:

```
// string expands to "(?]?????"!
char *e = "(???)???-????";
// possible work-arounds
char *f = "(" "???" ")" "???" "-" "????";
char *g = "(\?\?\?)\?\?\?-\?\?\?\?";
```

W128 *%d padding byte(s) added*

The compiler has added slack bytes to align a member to the correct offset.

W129 *#endif matches #if in different source file '%s'*

This warning may indicate a ***#endif*** nesting problem since the traditional usage of ***#if*** directives is confined to the same source file. This warning may often come before an error and it is hoped will provide information to solve a preprocessing directive problem.

W130 *Possible loss of precision*

This warning indicates that you may be converting a argument of one size to another, different size. For instance, you may be losing precision by passing a long argument to a function that takes a short. This warning is initially disabled. It must be explicitly enabled with *#pragma enable_message(130)* or option "-wce=130". It can be disabled later by using *#pragma disable_message(130).*

W131 *No prototype found for function '%s'*

A reference for a function appears in your program, but you do not have a prototype for that function defined. Implicit prototype will be used, but this will cause problems if the assumed prototype does not match actual function definition.

W132 *No storage class or type specified*

When declaring a data object, either storage class or data type must be given. If no type is specified, ***int*** is assumed. If no storage class is specified, the default depends on scope (see the *C Language Reference* for details). For instance

Example:
```
auto i;
```

is a valid declaration, as is

Example:
```
short i;
```

However,

Example:
```
    i;
```

is not a correctly formed declaration.

W133 *Symbol name truncated for '%s'*

Symbol is longer than the object file format allows and has been truncated to fit. Maximum length is 255 characters for OMF and 1024 characters for COFF or ELF object files.

W134 *Shift amount negative*

The right operand of a left or right shift operator is a negative value. The result of the shift operation is undefined.

Example:
```
    int a = 1 << -2;
```

The value of 'a' in the above example is undefined.

W135 *Shift amount too large*

The right operand of a left or right shift operator is a value greater than or equal to the width in bits of the type of the promoted left operand. The result of the shift operation is undefined.

Example:
```
    int a = 1 >> 123;
```

The value of 'a' in the above example is undefined.

W136 *Comparison equivalent to 'unsigned == 0'*

Comparing an unsigned expression to see whether it is <= 0 is equivalent to testing for == 0. Check to see if the expression should be signed instead of unsigned.

B.2 Warning Level 2 Messages

W200 *'%s' has been referenced but never assigned a value*

You have used the variable in an expression without previously assigning a value to that variable.

W201 *Unreachable code*

The statement will never be executed, because there is no path through the program that causes control to reach this statement.

W202 *Symbol '%s' has been defined, but not referenced*

There are no references to the declared variable. The declaration for the variable can be deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced).*

W203 *Preprocessing symbol '%s' has not been declared*

The symbol has been used in a preprocessor expression. The compiler assumes the symbol has a value of 0 and continues. A `#define` may be required for the symbol, or you may have forgotten to include the file which contains a `#define` for the symbol.

B.3 Warning Level 3 Messages

W300 *Nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ANSI C. You may be missing the `*/` for the previous comment.

W301 *not used*

unused message

W302 *Expression is only useful for its side effects*

You have an expression that would have generated the warning "Meaningless use of an expression", except that it also contains a side-effect, such as ++, −−, or a function call.

W303 *Parameter '%s' has been defined, but not referenced*

There are no references to the declared parameter. The declaration for the parameter can be deleted. Since it is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the parameter. You can prevent the message from being issued through use of *#pragma off(unreferenced).*

This warning is initially disabled. It must be specifically enabled with *#pragma enable_message(303)* or option "-wce=303". It can be disabled later by using *#pragma disable_message(303).*

W304 *Return type 'int' assumed for function '%s'*

If a function is declared without specifying return type, such as

Example:
```
    foo( void );
```

then its return type will be assumed to be ***int***

W305 *Type 'int' assumed in declaration of '%s'*

If an object is declared without specifying its type, such as

Example:

```
register count;
```

then its type will be assumed to be ***int***

W306 *Assembler warning: '%s'*

A problem has been detected by the in-line assembler. The message indicates the problem detected.

B.4 Error Messages

E1000 *BREAK must appear in while, do, for or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace { for a ***while, do, for*** or ***switch*** statement.

E1001 *CASE must appear in switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

E1002 *CONTINUE must appear in while, do or for statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many } between the ***while, do*** or ***for*** statement and the ***continue*** statement.

E1003 *DEFAULT must appear in switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many } between the start of the ***switch*** and the ***default*** label.

E1004 *Misplaced '}' or missing earlier '{'*

An extra } has been found which cannot be matched up with an earlier { .

E1005 *Misplaced #elif directive*

The `#elif` directive must be inside an `#if` preprocessing group and before the `#else` directive if present.

E1006 *Misplaced #else directive*

The `#else` directive must be inside an `#if` preprocessing group and follow all `#elif` directives if present.

E1007 *Misplaced #endif directive*

A preprocessing directive has been found without a matching `#if` directive. You either have an extra or you are missing an `#if` directive earlier in the file.

E1008 *Only 1 DEFAULT per switch allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

E1009 *Expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

E1010 *Type mismatch*

For pointer subtraction, both pointers must point to the same type. For other operators, both expressions must be assignment compatible.

E1011 *Symbol '%s' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to `#include` a header file that contains the declaration.

E1012 *Expression is not a function*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

E1013 *Constant variable cannot be modified*

An expression or statement has been found which modifies a variable which has been declared with the ***const*** keyword.

E1014 *Left operand must be an 'lvalue'*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

E1015 *'%s' is already defined as a variable*

You are trying to declare a function with the same name as a previously declared variable.

E1016 *Expecting identifier*

The token following "->" and "." operators must be the name of an identifier which appears in the struct or union identified by the operand preceding the "->" and "." operators.

E1017 *Label '%s' already defined*

All labels within a function must be unique.

E1018 *Label '%s' not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

E1019 *Tag '%s' already defined*

All ***struct, union*** and ***enum*** tag names must be unique.

E1020 *Dimension cannot be 0 or negative*

The dimension of an array must be positive and non-zero.

E1021 *Dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

E1022 *Missing or misspelled data type near '%s'*

The compiler has found an identifier that is not a predefined type or the name of a "typedef". Check the identifier for a spelling mistake.

E1023 *Storage class of parameter must be register or unspecified*

The only storage class allowed for a parameter declaration is ***register.***

E1024 *Declared symbol '%s' is not in parameter list*

Make sure that all the identifiers in the parameter list match those provided in the declarations between the start of the function and the opening brace "{".

E1025 *Parameter '%s' already declared*

A declaration for the specified parameter has already been processed.

E1026 *Invalid declarator*

A syntax error has occurred while parsing a declaration.

E1027 *Invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

E1028 *Variable '%s' cannot be void*

You cannot declare a ***void*** variable.

E1029 *Expression must be 'pointer to ...'*

An attempt has been made to de-reference (*) a variable or expression which is not declared to be a pointer.

E1030 *Cannot take the address of an rvalue*

You can only take the address of a variable or memory location.

E1031 *Name '%s' not found in struct/union %s*

The specified identifier is not one of the fields declared in the ***struct*** or ***union.*** Check that the field name is spelled correctly, or that you are pointing to the correct ***struct*** or ***union.***

E1032 *Expression for '.' must be a 'structure' or 'union'*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***struct*** or ***union*** type.

E1033 *Expression for '->' must be 'pointer to struct or union'*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***struct*** or ***union*** type.

E1034 *Symbol '%s' already defined*

The specified symbol has already been defined.

E1035 *static function '%s' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

E1036 *Right operand of '%s' is a pointer*

The right operand of "+=" and "−=" cannot be a pointer. The right operand of "−" cannot be a pointer unless the left operand is also a pointer.

E1037 *Type cast must be a scalar type*

You cannot type cast an expression to be a ***struct, union,*** array or function.

E1038 *Expecting label for goto statement*

The ***goto*** statement requires the name of a label.

E1039 *Duplicate case value '%s' found*

Every case value in a ***switch*** statement must be unique.

E1040 *Field width too large*

The maximum field width allowed is 16 bits.

E1041 *Field width of 0 with symbol not allowed*

A bit field must be at least one bit in size.

E1042 *Field width must be positive*

You cannot have a negative field width.

E1043 *Invalid type specified for bit field*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

E1044 *Variable '%s' has incomplete type*

A full definition of a ***struct*** or ***union*** has not been given.

E1045 *Subscript on non-array*

One of the operands of "[]" must be an array.

E1046 *Incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

E1047 *Argument for # must be a macro parm*

The argument for the stringize operator "#" must be a macro parameter.

E1048 *Unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

E1049 *Invalid #include directive*

A syntax error has been encountered in a `#include` directive.

E1050 *Not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

E1051 *Not expecting a return value for function '%s'*

The specified function is declared as a ***void*** function. Delete the ***return*** statement, or change the type of the function.

E1052 *Expression has void type*

You tried to use the value of a ***void*** expression inside another expression.

E1053 *Cannot take the address of a bit field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

E1054 *Expression must be constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

E1055 *Unable to open '%s'*

The file specified in an `#include` directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the `INCLUDE` environment variable or the "-I" option on the command line.

E1056 *Too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro.

E1057 *Modifiers disagree with previous definition of '%s'*

You have more than one definition or prototype for the variable or function which have different type modifiers.

E1058 *Cannot use typedef '%s' as a variable*

The name of a typedef has been found when an operand or operator is expected. If you are trying to use a type cast, make sure there are parentheses around the type, otherwise check for a spelling mistake.

E1059 *Invalid storage class for non-local variable*

A variable with module scope cannot be defined with the storage class of ***auto*** or ***register.***

E1060 *Invalid type*

An invalid combination of the following keywords has been specified in a type declaration: ***const, volatile, signed, unsigned, char, int, short, long, float*** and ***double.***

E1061 *Expecting data or function declaration, but found '%s'*

The compiler is expecting the start of a data or function declaration. If you are only part way through a function, then you have too many closing braces "}".

E1062 *Inconsistent return type for function '%s'*

Two prototypes for the same function disagree.

E1063 *Missing operand*

An operand is required in the expression being parsed.

E1064 *Out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not `#include` header files that are not required.

For the 16-bit C compiler, the "-d2" option causes the compiler to use more memory. Try compiling with the "-d1" option instead.

E1065 *Invalid character constant*

This message is issued for an improperly formed character constant.

E1066 *Cannot perform operation with pointer to void*

You cannot use a "pointer to void" with the operators +, −, ++, −−, += and −=.

E1067 *Cannot take address of variable with storage class 'register'*

If you want to take the address of a local variable, change the storage class from ***register*** to ***auto.***

E1068 *Variable '%s' already initialized*

The specified variable has already been statically initialized.

E1069 *Ending \" missing for string literal*

The compiler did not find a second double quote to end the string literal.

E1070 *Data for aggregate type must be enclosed in curly braces*

When an array, struct or union is statically initialized, the data must be enclosed in curly braces {}.

E1071 *Type of parameter %d does not agree with previous definition*

The type of the specified parameter is incompatible with the prototype for that function. The following example illustrates a problem that can arise when the sequence of declarations is in the wrong order.

Example:

```
/* Uncommenting the following line will
   eliminate the error */
/* struct foo; */

void fn1( struct foo * );

struct foo {
    int     a,b;
};

void fn1( struct foo *bar )
{
    fn2( bar );
}
```

The problem can be corrected by reordering the sequence in which items are declared (by moving the description of the structure `foo` ahead of its first reference or by adding the indicated statement). This will assure that the first instance of structure `foo` is defined at the proper outer scope.

E1072 *Storage class disagrees with previous definition of '%s'*

The previous definition of the specified variable has a storage class of ***static.*** The current definition must have a storage class of ***static*** or ***extern.***

E1073 *Invalid option '%s'*

The specified option is not recognized by the compiler.

E1074 *Invalid optimization option '%s'*

The specified option is an unrecognized optimization option.

E1075 *Invalid memory model '%s'*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

E1076 *Missing semicolon at end of declaration*

You are missing a semicolon ";" on the declaration just before the left curly brace "{".

E1077 *Missing '}'*

The compiler detected end of file before finding a right curly brace "}" to end the current function.

E1078 *Invalid type for switch expression*

The type of a switch expression must be integral.

E1079 *Expression must be integral*

An integral expression is required.

E1080 *Expression must be arithmetic*

Both operands of the "*", "/" and "%" operators must be arithmetic. The operand of the unary minus must also be arithmetic.

E1081 *Expression must be scalar type*

A scalar expression is required.

E1082 *Statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

E1083 *Statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

E1084 *Statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

E1085 *Statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

E1086 *Expression too complicated, split it up and try again*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

E1087 *Missing matching #endif directive*

You are missing a to terminate a `#if, #ifdef` or `#ifndef` preprocessing directive.

E1088 *Invalid macro definition, missing)*

The right parenthesis ")" is required for a function-like macro definition.

E1089 *Missing) for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

E1090 *Invalid conversion*

A ***struct*** or ***union*** cannot be converted to anything. A ***float*** or ***double*** cannot be converted to a pointer and a pointer cannot be converted to a ***float*** or ***double.***

E1091 *%s*

This is a user message generated with the `#error` preprocessing directive.

E1092 *Cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

E1093 *Function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

E1094 *Function cannot return a function*

You cannot return a function. You can return a pointer to a function.

E1095 *Cannot take address of local variable in static initialization*

You cannot take the address of an ***auto*** variable at compile time.

E1096 *Inconsistent use of return statements*

The compiler has found a ***return*** statement which returns a value and a ***return*** statement that does not return a value both in the same function. The ***return*** statement which does not return a value needs to have a value specified to be consistent with the other ***return*** statement in the function.

E1097 *Missing ? or misplaced :*

The compiler has detected a syntax error related to the "?" and ":" operators. You may need parenthesis around the expressions involved so that it can be parsed correctly.

E1098 *Maximum struct or union size is 64K*

The size of a ***struct*** or ***union*** is limited to 64K so that the compiler can represent the offset of a member in a 16-bit register.

E1099 *Statement must be inside function. Probable cause: missing {*

The compiler has detected a statement such as ***for, while, switch,*** etc., which must be inside a function. You either have too many closing braces "}" or you are missing an opening brace "{" earlier in the function.

E1100 *Definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must `#undef` it before you can define it with a new definition.

E1101 *Cannot #undef '%s'*

The special macros `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__`, `__FUNCTION__` and `__func__`, and the identifier "defined", cannot be deleted by the `#undef` directive.

E1102 *Cannot #define the name 'defined'*

You cannot define a macro called `defined`.

E1103 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

E1104 *Type cast not allowed in #if or #elif expression*

A type cast is not allowed in a preprocessor expression.

E1105 *'sizeof' not allowed in #if or #elif expression*

The ***sizeof*** operator is not allowed in a preprocessor expression.

E1106 *Cannot compare a struct or union*

A ***struct*** or ***union*** cannot be compared with "==" or "!=". You must compare each member of a ***struct*** or ***union*** to determine equality or inequality. If the ***struct*** or ***union*** is packed (has no holes in it for alignment purposes) then you can compare two structs using `memcmp`.

E1107 *Enumerator list cannot be empty*

You must have at least one identifier in an ***enum*** list.

E1108 *Invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

E1109 *Cannot take sizeof a bit field*

The smallest object that you can ask for the size of is a char.

E1110 *Cannot initialize variable with storage class of extern*

A storage class of ***extern*** is used to associate the variable with its actual definition somewhere else in the program.

E1111 *Invalid storage class for parameter*

The only storage class allowed for a parameter is ***register.***

E1112 *Initializer list cannot be empty*

An initializer list must have at least one item specified.

E1113 *Expression has incomplete type*

An attempt has been made to access a struct or union whose definition is not known, or an array whose dimensions are not known.

E1114 *Struct or union cannot contain itself*

You cannot have a ***struct*** or ***union*** contain itself. You can have a pointer in the ***struct*** which points to an instance of itself. Check for a missing "*" in the declaration.

E1115 *Incomplete enum declaration*

The enumeration tag has not been previously defined.

E1116 *An id list not allowed except for function definition*

A function prototype must contain type information.

E1117 *Must use 'va_start' macro inside function with variable parameters*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

E1118 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

E1119 *Internal compiler error %d*

A bug has been encountered in the C compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to compiler developers so that we can fix the problem.

E1120 *Parameter number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

E1121 *Procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

E1122 *Illegal register modified by '%s' #pragma*

For the 16-bit C compiler: The BP, CS, DS, and SS registers cannot be modified in small data models. The BP, CS, and SS registers cannot be modified in large data models.

For the 32-bit C compiler: The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data models. The EBP, CS, and SS registers cannot be modified in large data models.

E1123 *File must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function). This message is only issued in strict ANSI mode (-za).

E1124 *Out of macro space*

The compiler ran out of memory for storing macro definitions.

E1125 *Keyboard interrupt detected*

The compile has been aborted with Ctrl/C or Ctrl/Break.

E1126 *Array, struct or union cannot be placed in a register*

Only scalar objects can be specified with the ***register*** class.

E1127 *Type required in parameter list*

If the first parameter in a function definition or prototype is defined with a type, then all of the parameters must have a type specified.

E1128 *Enum constant is out of range %s*

All of the constants must fit into appropriate value range.

E1129 *Type does not agree with previous definition of '%s'*

You have more than one definition of a variable or function that do not agree.

E1130 *Duplicate name '%s' not allowed in struct or union*

All the field names in a ***struct*** or ***union*** must be unique.

E1131 *Duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

E1132 *Unable to open work file: error code = %d*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

E1133 *Write error on work file: error code = %d*

An error was encountered trying to write information to the work file. The disk could be full.

E1134 *Read error on work file: error code = %d*

An error was encountered trying to read information from the work file.

E1135 *Seek error on work file: error code = %d*

An error was encountered trying to seek to a position in the work file.

E1136 *not used*

unused message

E1137 *Out of enum space*

The compiler has run out of space allocated to store information on all of the ***enum*** constants defined in your program.

E1138 *Filename required on command line*

The name of a file to be compiled must be specified on the command line.

E1139 *Command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the cc utility to compile multiple files with a single command.

E1140 *_leave must appear in a _try statement*

The ***_leave*** keyword must be inside a ***_try*** statement. The ***_leave*** keyword causes the program to jump to the start of the ***_finally*** block.

E1141 *Expecting end of line but found '%s'*

A syntax error has been detected. The token displayed in the message should help you determine the problem.

E1142 *Too many bytes specified in #pragma*

There is an internal limit on the number of bytes for in-line code that can be specified with a pragma. Try splitting the function into two or more smaller functions.

E1143 *Cannot resolve linkage conventions for routine '%s' #pragma*

The compiler cannot generate correct code for the specified routine because of register conflicts. Change the registers used by the parameters of the pragma.

E1144 *Symbol '%s' in pragma must be global*

The in-line code for a pragma can only reference a global variable or function. You can only reference a parameter or local variable by passing it as a parameter to the in-line code pragma.

E1145 *Internal compiler limit exceeded, break module into smaller pieces*

The compiler can handle 65535 quadruples, 65535 leaves, and 65535 symbol table entries and literal strings. If you exceed one of these limits, the program must be broken into smaller pieces until it is capable of being processed by the compiler.

E1146 *Invalid initializer for integer data type*

Integer data types (int and long) can be initialized with numeric expressions or address expressions that are the same size as the integer data type being initialized.

E1147 *Too many errors: compilation aborted*

The compiler stops compiling when the number of errors generated exceeds the error limit. The error limit can be set with the "-e" option. The default error limit is 20.

E1148 *Expecting identifier but found '%s'*

A syntax error has been detected. The token displayed in the message should help you determine the problem.

E1149 *Expecting constant but found '%s'*

The #line directive must be followed by a constant indicating the desired line number.

E1150 *Expecting \"filename\" but found '%s'*

The second argument of the #line directive must be a filename enclosed in quotes.

E1151 *Parameter count does not agree with previous definition*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you are missing the ", ..." in the function prototype.

E1152 *Segment name required*

A segment name must be supplied in the form of a literal string to the __segname() directive.

E1153 *Invalid __based declaration*

The compiler could not recognize one of the allowable forms of __based declarations. See the *C Language Reference* document for description of all the allowable forms of __based declarations.

E1154 *Variable for __based declaration must be of type __segment or pointer*

A based pointer declaration must be based on a simple variable of type __segment or pointer.

E1155 *Duplicate external symbol %s*

Duplicate external symbols will exist when the specified symbol name is truncated to 8 characters.

E1156 *Assembler error: '%s'*

An error has been detected by the in-line assembler. The message indicates the error detected.

E1157 *Variable must be 'huge'*

A variable or an array that requires more than 64K of storage in the 16-bit compiler must be declared as ***huge.***

E1158 *Too many parm sets*

Too many parameter register sets have been specified in the pragma.

E1159 *I/O error reading '%s': %s*

An I/O error has been detected by the compiler while reading the source file. The system dependent reason is also displayed in the message.

E1160 *Attempt to access far memory with all segment registers disabled in '%s'*

The compiler does not have any segment registers available to access the desired far memory location.

E1161 *No identifier provided for '-D' option*

The command line option "-D" must be followed by the name of the macro to be defined.

E1162 *Invalid register pegged to a segment in '%s'*

The register specified in a #pragma data_seg, or a ***__segname*** expression must be a valid segment register.

E1163 *Invalid octal constant*

An octal constant cannot contain the digits 8 or 9.

E1164 *Invalid hexadecimal constant*

The token sequence "0x" must be followed by a hexadecimal character (0-9, a-f, or A-F).

E1165 *Unexpected ')'. Probable cause: missing '('*

A closing parenthesis was found in an expression without a corresponding opening parenthesis.

E1166 *Symbol '%s' is unreachable from #pragma*

The in-line assembler found a jump instruction to a label that is too far away.

E1167 *Division or remainder by zero in a constant expression*

The compiler found a constant expression containing a division or remainder by zero.

E1168 *Cannot end string literal with backslash*

The argument to a macro that uses the stringize operator '#' on that argument must not end in a backslash character.

Example:

```
    #define str(x) #x
    str(@#\)
```

E1169 *Invalid __declspec declaration*

The only valid __declspec declarations are "__declspec(thread)", "__declspec(dllexport)", and "__declspec(dllimport)".

E1170 *Too many storage class specifiers*

You can only specify one storage class specifier in a declaration.

E1171 *Expecting '%s' but found end of file*

A syntax error has been detected. The compiler is still expecting more input when it reached the end of the source program.

E1172 *Expecting struct/union tag but found '%s'*

The compiler expected to find an identifier following the ***struct*** or ***union*** keyword.

E1173 *Operand of __builtin_isfloat() must be a type*

The __builtin_isfloat() function is used by the ***va_arg*** macro to determine if a type is a floating-point type.

E1174 *Invalid constant*

The token sequence does not represent a valid numeric constant.

E1175 *Too many initializers*

There are more initializers than objects to initialize. For example int X[2] = { 0, 1, 2 }; The variable "X" requires two initializers not three.

E1176 *Parameter %d, pointer type mismatch*

You have two pointers that either point to different objects, or the pointers are of different size, or they have different modifiers.

E1177 *Modifier repeated in declaration*

You have repeated the use of a modifier like "const" (an error) or "far" (a warning) in a declaration.

E1178 *Type qualifier mismatch*

You have two pointers that have different "const" or "volatile" qualifiers.

E1179 *Parameter %d, type qualifier mismatch*

You have two pointers that have different const or "volatile" qualifiers.

E1180 *Sign specifier mismatch*

You have two pointers that point to types that have different sign specifiers.

E1181 *Parameter %d, sign specifier mismatch*

You have two pointers that point to types that have different sign specifiers.

E1182 *Missing \\ for string literal*

You need a '\' to continue a string literal across a line.

E1183 *Expecting '%s' after '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

E1184 *Expecting '%s' after '%s' but found end of file*

A syntax error has been detected. The compiler is still expecting more input when it reached the end of the source program.

E1185 *Invalid register name '%s' in #pragma*

The register name is invalid/unknown.

B.5 Informational Messages

I2000 *Not enough memory to fully optimize procedure '%s'*

The compiler did not have enough memory to fully optimize the specified procedure. The code generated will still be correct and execute properly. This message is purely informational.

I2001 *Not enough memory to maintain full peephole*

Certain optimizations benefit from being able to store the entire module in memory during optimization. All functions will be individually optimized but the optimizer will not be able to share code between functions if this message appears. The code generated will still be correct and execute properly. This message is purely informational. It is only printed if the warning level is greater than or equal to 4.

The main reason for this message is for those people who are concerned about reproducing the exact same object code when the same source file is compiled on a different machine. You may not be able to reproduce the exact same object code from one compile to the next unless the available memory is exactly the same.

I2002 *'%s' defined in: %s(%u)*

This informational message indicates where the symbol in question was defined. The message is displayed following an error or warning diagnostic for the symbol in question.

Example:

```
static int a = 9;
int b = 89;
```

The variable 'a' is not referenced in the preceding example and so will cause a warning to be generated. Following the warning, the informational message indicates the line at which 'a' was declared.

I2003 *source conversion type is '%s'*

This informational message indicates the type of the source operand, for the preceding conversion diagnostic.

I2004 *target conversion type is '%s'*

This informational message indicates the target type of the conversion, for the preceding conversion diagnostic.

I2005 *Including file '%s'*

This informational message indicates that the specified file was opened as a result of `#include` directive processing.

B.6 Pre-compiled Header Messages

H3000 *Error reading PCH file*

The pre-compiled header file does not follow the correct format.

H3001 *PCH file header is out of date*

The pre-compiled header file is out of date with the compiler. The current version of the compiler is expecting a different format.

H3002 *Compile options differ with PCH file*

The command line options are not the same as used when making the pre-compiled header file. This can effect the values of the pre-compiled information.

H3003 *Current working directory differs with PCH file*

The pre-compiled header file was compiled in a different directory.

H3004 *Include file '%s' has been modified since PCH file was made*

The include files have been modified since the pre-compiled header file was made.

H3005 *PCH file was made from a different include file*

The pre-compiled header file was made using a different include file.

H3006 *Include path differs with PCH file*

The include paths have changed.

H3007 *Preprocessor macro definition differs with PCH file*

The definition of a preprocessor macro has changed.

H3008 *PCH cannot have data or code definitions.*

The include files used to build the pre-compiled header contain function or data definitions. This is not currently supported.

B.7 Miscellaneous Messages and Phrases

M4000 *Code size*

String used in message construction.

M4001 *Error!*

String used in message construction.

M4002 *Warning!*

String used in message construction.

M4003 *Note!*

String used in message construction.

M4004 *(Press return to continue)*

String used in message construction.

C. Open Watcom C++ Diagnostic Messages

The following is a list of all warning and error messages produced by the Open Watcom C++ compilers. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%N, %S, %T, %s, %d` and `%u.` They represent strings that are substituted by the Open Watcom C++ compilers to make the error message more exact. `%d` and `%u` represent a string of digits; `%N, %S, %T` and `%s` a string, usually a symbolic name.

Consider the following program, named `err.cpp`, which contains errors.

Example:

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
File: err.cpp
(6,12): Error! E042: symbol 'i' already defined
  'i' declared at: (5,9)
(9,5): Error! E029: symbol 'x' has not been declared
err.cpp: 12 lines, included 174, no warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number and column of the line containing the error (in parentheses),
3. a message number, and

4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `err.cpp.` Error number 042 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `err.cpp.` Error number 029 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

A number of messages contain a reference to the ARM. This is the "Annotated C++ Reference Manual" written by Margaret A. Ellis and Bjarne Stroustrup and published by Addison-Wesley (ISBN 0-201-51459-1).

C.1 Diagnostic Messages

000 *internal compiler error*

If this message appears, please report the problem directly to the Open Watcom development team. See http://www.openwatcom.org/.

001 *assignment of constant found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

002 *constant out of range; truncated*

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

Example:

```
int a = 12345678901234567890;
```

003 *missing return value*

A function has been declared with a non-void return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

Example:
```
int foo( int a )
{
    int b = a + a;
}
```

The message will be issued at the end of the function.

004 *base class '%T' does not have a virtual destructor*

A virtual destructor has been declared in a class with base classes. However, one of those base classes does not have a virtual destructor. A ***delete*** of a pointer cast to such a base class will not function properly in all circumstances.

Example:
```
struct Base {
    ~Base();
};
struct Derived : Base {
    virtual ~Derived();
};
```

It is considered good programming practice to declare virtual destructors in all classes used as base classes of classes having virtual destructors.

005 *pointer or reference truncated*

The expression contains a transfer of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., assigning a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. This message indicates that some information is being lost so check the code carefully.

Example:
```
extern int __far *foo();
int __far *p_far = foo();
int __near *p_near = p_far; // truncated
```

006 *syntax error; probable cause: missing ';'*

The compiler has found a complete expression (or declaration) during parsing but could not continue. The compiler has detected that it could have continued if a semicolon was present so there may be a semicolon missing.

Example:
```
enum S {
}   // missing ';'

class X {
};
```

007 *'&array' may not produce intended result*

The type of the expression '&array' is different from the type of the expression 'array'. Suppose we have the declaration `char buffer[80].` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what one may have expected. The address-of operator '&' is not required for getting the address of an array.

008 *returning address of function argument or of auto or register variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

Example:
```
int *foo()
{
    int k = 123;
    return &k;      // k is automatic variable
}
```

009 *option requires a file name*

The specified option is not recognized by the compiler since there was no file name after it (i.e., "-fo=my.obj").

010 *asm directive ignored*

The asm directive (e.g., asm("mov r0,1");) is a non-portable construct. The Open Watcom C++ compiler treats all asm directives like comments.

011 *all members are private*

This message warns the programmer that there will be no way to use the contents of the class because all accesses will be flagged as erroneous (i.e., accessing a private member).

Example:
```
class Private {
    int a;
    Private();
    ~Private();
    Private( const Private& );
};
```

012 *template argument cannot be type '%T'*

A template argument can be either a generic type (e.g., `template < class T >`), a pointer, or an integral type. These types are required for expressions that can be checked at compile time.

013 *unreachable code*

The indicated statement will never be executed because there is no path through the program that causes control to reach that statement.

Example:
```
void foo( int *p )
{
    *p = 4;
    return;
    *p = 6;
}
```

The statement following the ***return*** statement cannot be reached.

014 *no reference to symbol '%S'*

There are no references to the declared variable. The declaration for the variable can be deleted. If the variable is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced),* or adding a statement that assigns the variable to itself.

015 *nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ISO/ANSI C. You may be missing the `*/` for the previous comment.

016 *template argument list cannot be empty*

An empty template argument list would result in a template that could only define a single class or function.

017 *label '%s' has not been referenced by a goto*

The indicated label has not been referenced and, as such, is useless. This warning can be safely ignored.

Example:

```
int foo( int a, int b )
{
un_refed:
    return a + b;
}
```

018 *no reference to anonymous union member '%S'*

The declaration for the anonymous member can be safely deleted without any effect.

019 *'break' may only appear in a for, do, while, or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace { for a ***while, do, for*** or ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    break;  // illegal
    return a+b;
}
```

020 *'case' may only appear in a switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    case 4:    // illegal
    return a+b;
}
```

021 *'continue' may only appear in a for, do, or while statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many } between the ***while, do*** or ***for*** statement and the ***continue*** statement.

Example:

```
int foo( int a, int b )
{
    continue;   // illegal
    return a+b;
}
```

022 *'default' may only appear in a switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many } between the start of the ***switch*** and the ***default*** label.

Example:

```
int foo( int a, int b )
{
    default: // illegal
    return a+b;
}
```

023 *misplaced '}' or missing earlier '{'*

An extra } has been found which cannot be matched up with an earlier { .

024 *misplaced #elif directive*

The ***#elif*** directive must be inside an ***#if*** preprocessing group and before the ***#else*** directive if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

025 *misplaced #else directive*

The ***#else*** directive must be inside an ***#if*** preprocessing group and follow all ***#elif*** directives if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

026 *misplaced #endif directive*

A ***#endif*** preprocessing directive has been found without a matching ***#if*** directive. You either have an extra ***#endif*** or you are missing an ***#if*** directive earlier in the file.

Example:
```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

027 *only one 'default' per switch statement is allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

Example:
```
int translate( int a )
{
    switch( a ) {
      case 1:
    a = 8;
    break;
      default:
    a = 9;
    break;
      default: // illegal
    a = 10;
    break;
    }
    return a;
}
```

028 *expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

029 *symbol '%N' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to ***#include*** a header file that contains the declaration.

Example:

```
int a = b;  // b has not been declared
```

030 *left expression must be a function or a function pointer*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

Example:

```
int a;
int b = a( 12 );
```

031 *operand must be an lvalue*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

Example:

```
void foo( int a )
{
    ( a + 1 ) = 7;
    int b = ++ ( a + 6 );
}
```

Both statements within the function are erroneous, since lvalues are expected where the additions are shown.

032 *label ’%s’ already defined*

All labels within a function must be unique.

Example:
```
void bar( int *p )
{
label:
    *p = 0;
label:
    return;
}
```

The second label is illegal.

033 *label ’%s’ is not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

Example:
```
void bar( int *p )
{
labl:
    *p = 0;
    goto label;
}
```

The label referenced in the ***goto*** is not defined.

034 *dimension cannot be zero*

The dimension of an array must be non-zero.

Example:
```
int array[0];   // not allowed
```

035 *dimension cannot be negative*

The dimension of an array must be positive.

Example:
```
    int array[-1];  // not allowed
```

036 *dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

Example:
```
    int array[][];   // not allowed
```

037 *invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

Example:
```
    auto void foo()
    {
    }
```

038 *expression must have pointer type*

An attempt has been made to de-reference a variable or expression which is not declared to be a pointer.

Example:
```
    int a;
    int b = *a;
```

039 *cannot take address of an rvalue*

You can only take the address of a variable or memory location.

Example:
```
char c;
char *p1 = & & c;    // not allowed
char *p2 = & (c+1); // not allowed
```

040 *expression for '.' must be a class, struct or union*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***class, struct*** or ***union*** type.

Example:
```
struct S
{
    int a;
};
int &fun();
int a = fun().a;
```

041 *expression for '->' must be pointer to class, struct or union*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***class, struct*** or ***union*** type.

Example:
```
struct S
{
    int a;
};
int *fun();
int a = fun()->a;
```

042 *symbol '%S' already defined*

The specified symbol has already been defined.

Example:
```
char a = 2;
char a = 2; // not allowed
```

043 *static function '%S' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

Example:
```
static int fun( void );
int k = fun();
// fun not defined by end of program
```

044 *expecting label for goto statement*

The ***goto*** statement requires the name of a label.

Example:
```
int fun( void )
{
    goto;
}
```

045 *duplicate case value '%s' found*

Every case value in a ***switch*** statement must be unique.

Example:
```
int fun( int a )
{
    switch( a ) {
      case 1:
    return 7;
      case 2:
    return 9;
      case 1: // duplicate not allowed
    return 7;
    }
    return 79;
}
```

046 *bit-field width is too large*

The maximum field width allowed is 16 bits in the 16-bit compiler and 32 bits in the 32-bit compiler.

Example:
```
struct S
{
    unsigned bitfield :48;  // too wide
};
```

047 *width of a named bit-field must not be zero*

A bit field must be at least one bit in size.

Example:
```
struct S {
    int bitfield :10;
    int :0;   // okay, aligns to int
    int h :0; // error, field is named
};
```

048 *bit-field width must be positive*

You cannot have a negative field width.

Example:
```
struct S
{
    unsigned bitfield :-10; // cannot be negative
};
```

049 *bit-field base type must be an integral type*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

Example:
```
    struct S
    {
        float bitfield : 10;    // must be integral
    };
```

050 *subscript on non-array*

One of the operands of '[]' must be an array or a pointer.

Example:
```
    int array[10];
    int i1 = array[0];  // ok
    int i2 = 0[array];  // same as above
    int i3 = 0[1];      // illegal
```

051 *incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

052 *argument for # must be a macro parm*

The argument for the stringize operator '#' must be a macro parameter.

053 *unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

Example:
```
    #i_goofed   // not valid
```

054 *invalid #include directive*

A syntax error has been encountered in a ***#include*** directive.

Example:
```
    #include    // no header file
    #include stdio.h
```

Both examples are illegal.

055 *not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

Example:
```
#define mac(a,b) a+b
int i = mac(123);   // needs 2 parameters
```

056 *not expecting a return value*

The specified function is declared as a ***void*** function. Delete the ***return*** value, or change the type of the function.

Example:
```
void fun()
{
    return 14;  // not expecting return value
}
```

057 *cannot take address of a bit-field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

Example:
```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
void* p = &var.bitfield;    // illegal
```

058 *expression must be a constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

059 *unable to open '%s'*

The file specified in an ***#include*** directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the **INCLUDE** or **INCLUDE** environment variables or in the "i=" option on the command line.

060 *too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro. The extra parameters are ignored.

Example:

```
#define mac(a,b) a+b
int i = mac(1,2,3); // needs 2 parameters
```

061 *cannot use __based or __far16 pointers in this context*

The use of ***__based*** and ***__far16*** pointers is prohibited in ***throw*** expressions and ***catch*** statements.

Example:

```
extern int __based( __segname( "myseg" ) ) *pi;

void bad()
{
    try {
    throw pi;
    } catch( int __far16 *p16 ) {
    *p16 = 87;
    }
}
```

Both the ***throw*** expression and ***catch*** statements cause this error to be diagnosed.

062 *only one type is allowed in declaration specifiers*

Only one type is allowed for the first part of a declaration. A common cause of this message is that there may be a missing semi-colon (';') after a class definition.

Example:

```
class C
{
public:
    C();
}           // needs ";"

int foo() { return 7; }
```

063 *out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not ***#include*** header files that are not required.

064 *invalid character constant*

This message is issued for an improperly formed character constant.

Example:

```
char c = '12345';
char d = ''';
```

065 *taking address of variable with storage class 'register'*

You can take the address of a ***register*** variable in C++ (but not in ISO/ANSI C). If there is a chance that the source will be compiled using a C compiler, change the storage class from ***register*** to ***auto.***

Example:

```
extern int foo( char* );
int bar()
{
    register char c = 'c';
    return foo( &c );
}
```

066 *'delete' expression size is not allowed*

The C++ language has evolved to the point where the ***delete*** expression size is no longer required for a correct deletion of an array.

Example:
```
void fn( unsigned n, char *p ) {
    delete [n] p;
}
```

067 *ending " missing for string literal*

The compiler did not find a second double quote to end the string literal.

Example:
```
char *a = "no_ending_quote;
```

068 *invalid option*

The specified option is not recognized by the compiler.

069 *invalid optimization option*

The specified option is an unrecognized optimization option.

070 *invalid memory model*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

071 *expression must be integral*

An integral expression is required.

Example:

```
int foo( int a, float b, int *p )
{
    switch( a ) {
      case 1.3:     // must be integral
    return p[b];    // index not integer
      case 2:
    b <<= 2;    // can only shift integers
      default:
    return b;
    }
}
```

072 *expression must be arithmetic*

Arithmetic operations, such as "/" and "*", require arithmetic operands unless the operation has been overloaded or unless the operands can be converted to arithmetic operands.

Example:

```
class C
{
public:
    int c;
};
C cv;
int i = cv / 2;
```

073 *statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

Example:

```
extern int bar( int );
void foo( int a )
{
    if( a ) goto ending;
    bar( a );
ending:
    // needs statement following
}
```

074 *statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

075 *statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
    int foo( int a )
    {
        switch( a ) {
          default:
        return 7;
          case 1: // needs statement following
        }
        return 18;
    }
```

076 *statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
    int foo( int a )
    {
        switch( a ) {
          case 7:
        return 7;
          default:
        // needs statement following
        }
        return 18;
    }
```

077 *missing matching #endif directive*

You are missing a ***#endif*** to terminate a ***#if, #ifdef*** or ***#ifndef*** preprocessing directive.

Example:
```
#if 1
int a;
// needs #endif
```

078 *invalid macro definition, missing ')'*

The right parenthesis ")" is required for a function-like macro definition.

Example:
```
#define bad_mac( a, b
```

079 *missing ')' for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

Example:
```
#define mac( a, b) a+b
int d = mac( 1, 2
```

080 *%s*

This is a user message generated with the ***#error*** preprocessing directive.

Example:
```
#error my very own error message
```

081 *cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

Example:

```
typedef int TD(float);
TD array[12];
```

082 *function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

Example:

```
typedef int ARR[10];
ARR fun( float );
```

083 *function cannot return a function*

You cannot return a function. You can return a pointer to a function.

Example:

```
typedef int TD();
TD fun( float );
```

084 *function templates can only have type arguments*

A function template argument can only be a generic type (e.g., `template < class` T >). This is a restriction in the C++ language that allows compilers to automatically instantiate functions purely from the argument types of calls.

085 *maximum class size has been exceeded*

The 16-bit compiler limits the size of a ***struct*** or ***union*** to 64K so that the compiler can represent the offset of a member in a 16-bit register. This error also occurs if the size of a structure overflows the size of an ***unsigned*** integer.

Example:

```
struct S
{
    char arr1[ 0xfffe ];
    char arr2[ 0xfffe ];
    char arr3[ 0xfffe ];
    char arr4[ 0xfffffffe ];
};
```

086 *definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must ***#undef*** it before you can define it with a new definition.

Example:

```
#define CON 123
#define CON 124     // not same as previous
```

087 *initialization of '%S' must be in file scope*

A file scope variable must be initialized in file scope.

Example:

```
void fn()
{
    extern int v = 1;
}
```

088 *default argument for '%S' declared outside of class definition*

Problems can occur with member functions that do not declare all of their default arguments during the class definition. For instance, a copy constructor is declared if a class does not define a copy constructor. If a default argument is added later on to a constructor that makes it a copy constructor, an ambiguity results.

Example:

```
struct S {
    S( S const &, int );
    // S( S const & );  <-- declared by compiler
};
// ambiguity with compiler
// generated copy constructor
// S( S const & );
S::S( S const &, int = 0 )
{
}
```

089 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

Example:
```
#define badmac( a, b ) ## a ## b
```

090 *invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

Example:
```
float f = 123.9E+Q;
```

091 *'sizeof' is not allowed for a bit-field*

The smallest object that you can ask for the size of is a char.

Example:
```
struct S
{   int a;
    int b :10;
} v;
int k = sizeof( v.b );
```

092 *option requires a path*

The specified option is not recognized by the compiler since there was no path after it (i.e., "-i=d:\include;d:\path").

093 *must use 'va_start' macro inside function with variable arguments*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

Example:

```
#include <stdarg.h>
int foo( int a, int b )
{
    va_list args;
    va_start( args, a );
    va_end( args );
    return b;
}
```

094 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

095 *internal compiler error %d*

A bug has been encountered in the compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to the Open Watcom development team so that we can fix the problem. See http://www.openwatcom.org/.

096 *argument number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

097 *procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

098 *illegal register modified by '%s' #pragma*

For the 16-bit Open Watcom C/C++ compiler: The BP, CS, DS, and SS registers cannot be modified in small data models. The BP, CS, and SS registers cannot be modified in large data models.

For the 32-bit Open Watcom C/C++ compiler: The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data models. The EBP, CS, and SS registers cannot be modified in large data models.

099 *file must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function).

Note: This message has been disabled starting with Open Watcom v1.4. The ISO 1998 C++ standard allows empty translation units.

100 *out of macro space*

The compiler ran out of memory for storing macro definitions.

101 *keyboard interrupt detected*

The compilation has been aborted with Ctrl/C or Ctrl/Break.

102 *duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

Example:

```
    #define badmac( a, b, a ) a ## b
```

103 *unable to open work file: error code = %d*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

104 *write error on work file: error code = %d*

An error was encountered trying to write information to the work file. The disk could be full.

105 *read error on work file: error code = %d*

An error was encountered trying to read information from the work file.

106 *token too long; truncated*

The token must be less than 510 bytes in length.

107 *filename required on command line*

The name of a file to be compiled must be specified on the command line.

108 *command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the cc utility to compile multiple files with a single command.

109 *virtual member functions are not allowed in a union*

A union can only be used to overlay the storage of data. The storage of virtual function information (in a safe manner) cannot be done if storage is overlaid.

Example:

```
struct S1{ int f( int ); };
struct S2{ int f( int ); };
union un { S1 s1;
      S2 s2;
      virtual int vf( int );
    };
```

110 *union cannot be used as a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and achieve the same effect.

Example:

```
union U { int a; int b; };
class S : public U { int s; };
```

111 *union cannot have a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and inherit the base classes normally.

Example:
```
class S { public: int s; };
union U : public S { int a; int b; };
```

112 *cannot inherit an undefined base class '%T'*

The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:
```
class Undefined;
class C : public Undefined {
    int c;
};
```

113 *repeated direct base class will cause ambiguities*

Almost all accesses will be ambiguous. This restriction is useful in catching programming errors. The repeated base class can be encapsulated in another class if the repetition is required.

Example:
```
class Dup
{
    int d;
};
class C : public Dup, public Dup
{
    int c;
};
```

114 *templates may only be declared in namespace scope*

Currently, templates can only be declared in namespace scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

115 *linkages may only be declared in file scope*

A common source of errors for C and C++ result from the use of prototypes inside of functions. This restriction attempts to prevent such errors.

116 *unknown linkage '%s'*

Only the linkages "C" and "C++" are supported by Open Watcom C++.

Example:
```
    extern "APL" void AplFunc( int* );
```

117 *too many storage class specifiers*

This message is a result of duplicating a previous storage class or having a different storage class. You can only have one of the following storage classes, ***extern, static, auto, register,*** or ***typedef.***

Example:
```
    extern typedef int (*fn)( void );
```

118 *nameless declaration is not allowed*

A type was used in a declaration but no name was given.

Example:
```
    static int;
```

119 *illegal combination of type specifiers*

An incorrect scalar type was found. Either a scalar keyword was repeated or the combination is illegal.

Example:
```
short short x;
short long y;
```

120 *illegal combination of type qualifiers*

A repetition of a type qualifier has been detected. Some compilers may ignore repetitions but strictly speaking it is incorrect code.

Example:
```
const const x;
struct S {
    int virtual virtual fn();
};
```

121 *syntax error*

The C++ compiler was unable to interpret the text starting at the location of the message. The C++ language is sufficiently complicated that it is difficult for a compiler to correct the error itself.

122 *parser stack corrupted*

The C++ parser has detected an internal problem that usually indicates a compiler problem. Please report this directly to the Open Watcom development team. See http://www.openwatcom.org/.

123 *template declarations cannot be nested within each other*

Currently, templates can only be declared in namespace scope. Furthermore, a template declaration must be finished before another template can be declared.

124 *expression is too complicated*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

125 *invalid redefinition of the typedef name '%S'*

Redefinition of typedef names is only allowed if you are redefining a typedef name to itself. Any other redefinition is illegal. You should delete the duplicate ***typedef*** definition.

Example:
```
    typedef int TD;
    typedef float TD;   // illegal
```

126 *class '%T' has already been defined*

This message usually results from the definition of two classes in the same scope. This is illegal regardless of whether the class definitions are identical.

Example:
```
    class C {
    };
    class C {
    };
```

127 *'sizeof' is not allowed for an undefined type*

If a type has not been defined, the compiler cannot know how large it is.

Example:
```
    class C;
    int x = sizeof( C );
```

128 *initializer for variable '%S' cannot be bypassed*

The variable may not be initialized when code is executing at the position indicated in the message. The C++ language places these restrictions to prevent the use of uninitialized variables.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 1:
    int b = 2;
    return b;
      default: // b bypassed
    return b + 5;
    }
}
```

129 *division by zero in a constant expression*

Division by zero is not allowed in a constant expression. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 4 / 0:  // illegal
    return a;
    }
    return a + 2;
}
```

130 *arithmetic overflow in a constant expression*

The multiplication of two integral values cannot be represented. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 0x7FFF * 0x7FFF * 0x7FFF:  // overflow
    return a;
    }
    return a + 2;
}
```

131 *not enough memory to fully optimize procedure '%s'*

The indicated procedure cannot be fully optimized with the amount of memory available. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

132 *not enough memory to maintain full peephole*

Certain optimizations benefit from being able to store the entire module in memory during optimization. All functions will be individually optimized but the optimizer will not be able to share code between functions if this message appears. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

133 *too many errors: compilation aborted*

The Open Watcom C++ compiler sets a limit to the number of error messages it will issue. Once the number of messages reaches the limit the above message is issued. This limit can be changed via the "/e" command line option.

134 *too many parm sets*

An extra parameter passing description has been found in the aux pragma text. Only one parameter passing description is allowed.

135 *'friend', 'virtual' or 'inline' modifiers may only be used on functions*

This message indicates that you are trying to declare a strange entity like an ***inline*** variable. These qualifiers can only be used on function declarations and definitions.

136 *more than one calling convention has been specified*

A function cannot have more than one #pragma modifier applied to it. Combine the pragmas into one pragma and apply it once.

137 *pure member function constant must be '0'*

The constant must be changed to '0' in order for the Open Watcom C++ compiler to accept the pure virtual member function declaration.

Example:
```
struct S {
    virtual int wrong( void ) = 91;
};
```

138 *based modifier has been repeated*

A repeated based modifier has been detected. There are no semantics for combining base modifiers so this is not allowed.

Example:
```
char *ptr;
char __based( void ) __based( ptr ) *a;
```

139 *enumeration variable is not assigned a constant from its enumeration*

In C++ (as opposed to C), enums represent values of distinct types. Thus, the compiler will not automatically convert an integer value to an enum type if you are compiling your source in strict ISO/ANSI C++ mode. If you have extensions enabled, this message is treated as a warning.

Example:
```
enum Days { sun, mod, tues, wed, thur, fri, sat };
enum Days day = 2;
```

140 *bit-field declaration cannot have a storage class specifier*

Bit-fields (along with most members) cannot have storage class specifiers in their declaration. Remove the storage class specifier to correct the code.

Example:
```
class C
{
public:
    extern unsigned bitf :10;
};
```

141 *bit-field declaration must have a base type specified*

A bit-field cannot make use of a default integer type. Specify the type ***int*** to correct the code.

Example:
```
class C
{
public:
    bitf :10;
};
```

142 *illegal qualification of a bit-field declaration*

A bit-field can only be declared ***const*** or ***volatile.*** Qualifications like ***friend*** are not allowed.

Example:
```
struct S {
    friend int bit1 :10;
    inline int bit2 :10;
    virtual int bit3 :10;
};
```

All three declarations of bit-fields are illegal.

143 *duplicate base qualifier*

The compiler has found a repetition of base qualifiers like ***protected*** or ***virtual.***

Example:
```
struct Base { int b; };
struct Derived : public public Base { int d; };
```

144 *only one access specifier is allowed*

The compiler has found more than one access specifier for a base class. Since the compiler cannot choose one over the other, remove the unwanted access specifier to correct the code.

Example:
```
struct Base { int b; };
struct Derived : public protected Base { int d; };
```

145 *unexpected type qualifier found*

Type specifiers cannot have ***const*** or ***volatile*** qualifiers. This shows up in ***new*** expressions because one cannot allocate a ***const*** object.

146 *unexpected storage class specifier found*

Type specifiers cannot have ***auto*** or ***static*** storage class specifiers. This shows up in ***new*** expressions because one cannot allocate a ***static*** object.

147 *access to '%S' is not allowed because it is ambiguous*

There are two ways that this error can show up in C++ code. The first way a member can be ambiguous is that the same name can be used in two different classes. If these classes are combined with multiple inheritance, accesses of the name will be ambiguous.

Example:
```
struct S1 { int s; };
struct S2 { int s; };
struct Der : public S1, public S2
{
    void foo() { s = 2; };  // s is ambiguous
};
```

The second way a member can be ambiguous involves multiple inheritance. If a class is inherited non-virtually by two different classes which then get combined with multiple inheritance, an access of the member is faced with deciding which copy of the member is intended. Use the '::' operator to clarify what member is being accessed or access the member with a different class pointer or reference.

Example:
```
struct Top { int t; };
struct Mid : public Top { int m; };
struct Bot : public Top, public Mid
{
    void foo() { t = 2; };  // t is ambiguous
};
```

148 *access to private member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access private members of the class.

Example:
```
struct Top { int t; };
class Bot : private Top
{
    int foo() { return t; };  // t is private
};
Bot b;
int k = b.foo();    // foo is private
```

149 *access to protected member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access protected members of the class. The compiler also requires that ***protected*** members be accessed through a derived class to ensure that an unrelated base class cannot be quietly modified. This is a fairly recent change to the C++ language that may cause Open Watcom C++ to not accept older C++ code. See Section 11.5 in the ARM for a discussion of protected access.

Example:
```
struct Top { int t; };
struct Mid : public Top { int m; };
class Bot : protected Mid
{
protected:
    // t cannot be accessed
    int foo() { return t; };
};
Bot b;
int k = b.foo(); // foo is protected
```

150 *operation does not allow both operands to be pointers*

There may be a missing indirection in the code exhibiting this error. An example of this error is adding two pointers.

Example:
```
void fn()
{
    char *p, *q;

    p += q;
}
```

151 *operand is neither a pointer nor an arithmetic type*

An example of this error is incrementing a class that does not have any overloaded operators.

Example:
```
struct S { } x;
void fn()
{
    ++x;
}
```

152 *left operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:
```
struct S { } x;
void fn()
{
    x = x + 1;
}
```

153 *right operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    x = 1 + x;
}
```

154 *cannot subtract a pointer from an arithmetic operand*

The subtract operands are probably in the wrong order.

Example:

```
int fn( char *p )
{
    return( 10 - p );
}
```

155 *left expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = x * 1;
}
```

156 *right expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = 1 * x;
}
```

157 *left expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:
```
struct S { } x;
void fn()
{
    x = x ^ 1;
}
```

158 *right expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:
```
struct S { } x;
void fn()
{
    x = 1 ^ x;
}
```

159 *cannot assign a pointer value to an arithmetic item*

The pointer value must be cast to the desired type before the assignment takes place.

Example:
```
void fn( char *p )
{
    int a;

    a = p;
}
```

160 *attempt to destroy a far object when data model is near*

Destructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    ~Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

161 *attempt to call member function for far object when the data model is near*

Member functions cannot be called for objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    int foo();
};

Obj far obj;
int integer = obj.foo();
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

162 *template type argument cannot have a default argument*

This message was produced by earlier versions of the Open Watcom C++ compiler. Support for default template arguments was added in version 1.3 and this message was removed at that time.

163 *attempt to delete a far object when the data model is near*

delete cannot be used to deallocate objects which are stored in far memory when the default memory model for data is near.

Example:
```
struct Obj
{   char *p;
};

void foo( Obj far *p )
{
    delete p;
}
```

The second last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

164 *first operand is not a class, struct or union*

The ***offsetof*** operation can only be performed on a type that can have members. It is meaningless for any other type.

Example:
```
#include <stddef.h>

int fn( void )
{
    return offsetof( double, sign );
}
```

165 *syntax error: class template cannot be processed*

The class template contains unbalanced braces. The class definition cannot be processed in this form.

166 *cannot convert right pointer to type of left operand*

The C++ language will not allow the implicit conversion of unrelated class pointers. An explicit cast is required.

Example:

```
class C1;
class C2;

void fun( C1* pc1, C2* pc2 )
{
    pc2 = pc1;
}
```

167 *left operand must be an lvalue*

The left operand must be an expression that is valid on the left side of an assignment. Examples of incorrect lvalues include constants and the results of most operators.

Example:

```
int i, j;
void fn()
{
    ( i - 1 ) = j;
    1 = j;
}
```

168 *static data members are not allowed in an union*

A union should only be used to organize memory in C++. Enclose the union in a class if you need a static data member associated with the union.

Example:

```
union U
{
    static int a;
    int b;
    int c;
};
```

169 *invalid storage class for a member*

A class member cannot be declared with ***auto, register,*** or ***extern*** storage class.

Example:
```
class C
{
    auto int a;     // cannot specify auto
};
```

170 *declaration is too complicated*

The declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

Example:
```
int ************p;
```

Example:
```
// transform this to ...
typedef int ****PD1;
typedef PD1 ****PD2;
PD2 ****p;
```

171 *exception declaration is too complicated*

The exception declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

172 *floating-point constant too large to represent*

The Open Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the positive exponent is too large.

Example:
```
float f = 1.2e78965;
```

173 *floating-point constant too small to represent*

The Open Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the negative exponent is too large.

Example:
```
float f = 1.2e-78965;
```

174 *class template '%S' cannot be overloaded*

A class template name must be unique across the entire C++ program. Furthermore, a class template cannot coexist with another class template of the same name.

175 *range of enum constants cannot be represented*

If one integral type cannot be chosen to represent all values of an enumeration, the values cannot be used reliably in the generated code. Shrink the range of enumerator values used in the ***enum*** declaration.

Example:
```
enum E
{   e1 = 0xFFFFFFFF
,   e2 = -1
};
```

176 *'%S' cannot be in the same scope as a class template*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

177 *invalid storage class in file scope*

A declaration in file scope cannot have a storage class of ***auto*** or ***register.***

Example:
```
auto int a;
```

178 *const object must be initialized*

Constant objects cannot be modified so they must be initialized before use.

Example:

```
const int a;
```

179 *declaration cannot be in the same scope as class template '%S'*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

180 *template arguments must be named*

A member function of a template class cannot be defined outside the class declaration unless all template arguments have been named.

181 *class template '%S' is already defined*

A class template cannot have its definition repeated regardless of whether it is identical to the previous definition.

182 *invalid storage class for an argument*

An argument declaration cannot have a storage class of ***extern, static,*** or ***typedef.***

Example:

```
int foo( extern int a )
{
    return a;
}
```

183 *unions cannot have members with constructors*

A union should only be used to organize memory in C++. Allowing union members to have constructors would mean that the same piece of memory could be constructed twice.

Example:

```
class C
{
    C();
};
union U
{
    int a;
    C c;    // has constructor
};
```

184 *statement is too complicated*

The statement contains too many nested constructs. Break up the statement into multiple statements.

185 *'%s' is not the name of a class or namespace*

The right hand operand of a '::' operator turned out not to reference a class type or namespace. Because the name is followed by another '::', it must name a class or namespace.

186 *attempt to modify a constant value*

Modification of a constant value is not allowed. If you must force this to work, take the address and cast away the constant nature of the type.

Example:

```
static int const con = 12;
void foo()
{
    con = 13;        // error
    *(int*)&con = 13;   // ok
}
```

187 *'offsetof' is not allowed for a bit-field*

A bit-field cannot have a simple offset so it cannot be referenced in an ***offsetof*** expression.

Example:

```
#include <stddef.h>
struct S
{
    unsigned b1 :10;
    unsigned b2 :15;
    unsigned b3 :11;
};
int k = offsetof( S, b2 );
```

188 *base class is inherited with private access*

This warning indicates that the base class was originally declared as a ***class*** as opposed to a ***struct.*** Furthermore, no access was specified so the base class defaults to ***private*** inheritance. Add the ***private*** or ***public*** access specifier to prevent this message depending on the intended access.

189 *overloaded function cannot be selected for arguments used in call*

Either conversions were not possible for an argument to the function or a function with the right number of arguments was not available.

Example:
```
class C1;
class C2;
int foo( C1* );
int foo( C2* );
int k = foo( 5 );
```

190 *base operator operands must be " __segment :> pointer "*

The base operator (:>) requires the left operand to be of type __segment and the right operand to be a pointer.

Example:
```
char _based( void ) *pcb;
char __far *pcf = pcb;      // needs :> operator
```

Examples of typical uses are as follows:

Example:
```
const __segment mySegAbs = 0x4000;
char __based( void ) *c_bv = 24;
char __far *c_fp_1 = mySegAbs :> c_bv;
char __far *c_fp_2 = __segname( "_DATA" ) :> c_bv;
```

191 *expression must be a pointer or a zero constant*

In a conditional expression, if one side of the ':' is a pointer then the other side must also be a pointer or a zero constant.

Example:
```
extern int a;
int *p = ( a > 7 ) ? &a : 12;
```

192 *left expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:
```
void *p;
void *q = p + 2;
```

193 *right expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:
```
void *p;
void *q = 2 + p;
```

194 *expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:
```
void *p;
void *q = ++p;
```

195 *'sizeof' is not allowed for a function*

A function has no size defined for it by the C++ language specification.

Example:
```
typedef int FT( int );

unsigned y = sizeof( FT );
```

196 *'sizeof' is not allowed for type void*

The type ***void*** has no size defined for it by the C++ language specification.

Example:
```
void *p;
unsigned size = sizeof( *p );
```

197 *type cannot be defined in this context*

A type cannot be defined in certain contexts. For example, a new type cannot be defined in an argument list, a ***new*** expression, a conversion function identifier, or a catch handler.

Example:
```
extern int goop();
int foo()
{
    try {
    return goop();
    } catch( struct S { int s; } ) {
    return 2;
    }
}
```

198 *expression cannot be used as a class template parameter*

The compiler has to be able to compare expressions during compilation so this limits the complexity of expressions that can be used for template parameters. The only types of expressions that can be used for template parameters are constant integral expressions and addresses. Any symbols must have external linkage or must be static class members.

199 *premature end-of-file encountered during compilation*

The compiler expects more source code at this point. This can be due to missing parentheses (')') or missing closing braces ('}').

200 *duplicate case value '%s' after conversion to type of switch expression*

A duplicate ***case*** value has been found. Keep in mind that all case values must be converted to the type of the switch expression. Constants that may be different initially may convert to the same value.

Example:

```
enum E { e1, e2 };
void foo( short a )
{
    switch( a ) {
      case 1:
      case 0x10001:    // converts to 1 as short
    break;
    }
}
```

201 *declaration statement follows an if statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    if( a )
    int b = 14;
}
```

202 *declaration statement follows an else statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:
```
void foo( int a )
{
    if( a )
    int c = 15;
    else
    int b = 14;
}
```

203 *declaration statement follows a switch statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:
```
void foo( int a )
{
    switch( a )
    int b = 14;
}
```

204 *'this' pointer is not defined*

The ***this*** value can only be used from within non-static member functions.

Example:
```
void *fn()
{
    return this;
}
```

205 *declaration statement cannot follow a while statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    while( a )
    int b = 14;
}
```

206 *declaration statement cannot follow a do statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    do
    int b = 14;
    while( a );
}
```

207 *declaration statement cannot follow a for statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended. A ***for*** loop with an initial declaration is allowed to be used within another ***for*** loop, so this code is legal C++:

Example:

```
void fn( int **a )
{
    for( int i = 0; i < 10; ++i )
    for( int j = 0; j < 10; ++j )
        a[i][j] = i + j;
}
```

The following example, however, illustrates a potentially erroneous situation.

Example:
```
void foo( int a )
{
    for( ; a<10; )
    int b = 14;
}
```

208 *pointer to virtual base class converted to pointer to derived class*

Since the relative position of a virtual base can change through repeated derivations, this conversion is very dangerous. All C++ translators must report an error for this type of conversion.

Example:
```
struct VBase { int v; };
struct Der : virtual public VBase { int   d; };
extern VBase *pv;
Der *pd = (Der *)pv;
```

209 *cannot use far pointer in this context*

Only near pointers can be thrown when the data memory model is near.

Example:
```
extern int __far *p;
void foo()
{
    throw p;
}
```

When the small memory model (-ms switch) is selected, the ***throw*** expression is diagnosed as erroneous. Similarly, only near pointers can be specified in ***catch*** statements when the data memory model is near.

210 *returning reference to function argument or to auto or register variable*

The storage for the automatic variable will be destroyed immediately upon function return. Returning a reference effectively allows the caller to modify storage which does not exist.

Example:

```
class C
{
    char *p;
public:
    C();
    ~C();
};

C& foo()
{
    C auto_var;
    return auto_var;    // not allowed
}
```

211 *#pragma attributes for '%S' may be inconsistent*

A pragma attribute was changed to a value which matches neither the current default not the previous value for that attribute. A warning is issued since this usually indicates an attribute is being set twice (or more) in an inconsistent way. The warning can also occur when the default attribute is changed between two pragmas for the same object.

212 *function arguments cannot be of type void*

Having more than one ***void*** argument is not allowed. The special case of one ***void*** argument indicates that the function accepts no parameters.

Example:

```
void fn1( void )        // OK
{
}
void fn2( void, void, void )    // Error!
{
}
```

213 *class template requires more parameters for instantiation*

The class template instantiation has too few parameters supplied so the class cannot be instantiated properly.

214 *class template requires fewer parameters for instantiation*

The class template instantiation has too many parameters supplied so the class cannot be instantiated properly.

215 *no declared 'operator new' has arguments that match*

An ***operator new*** could not be found to match the ***new*** expression. Supply the correct arguments for special ***operator new*** functions that are defined with the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void *operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

216 *wide character string concatenated with a simple character string*

There are no semantics defined for combining a wide character string with a simple character string. To correct the problem, make the simple character string a wide character string by prefixing it with a ***L.***

Example:

```
char *p = "1234" L"5678";
```

217 *'offsetof' is not allowed for a static member*

A ***static*** member does not have an offset like simple data members. If this is required, use the address of the ***static*** member.

Example:
```
#include <stddef.h>
class C
{
public:
    static int stat;
    int memb;
};

int size_1 = offsetof( C, stat );   // not allowed
int size_2 = offsetof( C, memb );   // ok
```

218 *cannot define an array of void*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot declare an array of ***void.***

Example:
```
void array[24];
```

219 *cannot define an array of references*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating an array of references is currently not allowed in the C++ language.

Example:
```
int& array[24];
```

220 *cannot define a reference to void*

One cannot create a reference to a ***void*** because there can be no ***void*** variables to supply for initializing the reference.

Example:
```
void& ref;
```

221 *cannot define a reference to another reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a reference to another reference is currently not allowed in the C++ language.

Example:
```
int & & ref;
```

222 *cannot define a pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a pointer to a reference is currently not allowed in the C++ language.

Example:
```
char& *ptr;
```

223 *cannot initialize array with 'operator new'*

The initialization of arrays created with ***operator new*** can only be done with default constructors. The capability of using another constructor with arguments is currently not allowed in the C++ language.

Example:
```
struct S
{
    S( int );
};
S *p = new S[10] ( 12 );
```

224 *'%N' is a variable of type void*

A variable cannot be of type ***void.*** The ***void*** type can only be used in restricted circumstances because it has no size. For instance, a function returning ***void*** means that it does not return any value. A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

225 *cannot define a member pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a member pointer to a reference is currently not allowed in the C++ language.

Example:
```
struct S
{
    S();
    int &ref;
};

int& S::* p;
```

226 *function '%S' is not distinct*

The function being declared is not distinct enough from the other functions of the same name. This means that all function overloads involving the function's argument types will be ambiguous.

Example:
```
struct S {
    int s;
};
extern int foo( S* );
extern int foo( S* const ); // not distinct enough
```

227 *overloaded function is ambiguous for arguments used in call*

The compiler could not find an unambiguous choice for the function being called.

Example:
```
extern int foo( char );
extern int foo( short );
int k = foo( 4 );
```

228 *declared 'operator new' is ambiguous for arguments used*

The compiler could not find an unambiguous choice for ***operator new.***

Example:
```
#include <stdlib.h>
struct Der
{
    int s[2];
    void* operator new( size_t, char );
    void* operator new( size_t, short );
};
Der *p = new(10) Der;
```

229 *function '%S' has already been defined*

The function being defined has already been defined elsewhere. Even if the two function bodies are identical, there must be only one definition for a particular function.

Example:
```
int foo( int s ) { return s; }
int foo( int s ) { return s; }  // illegal
```

230 *expression on left is an array*

The array expression is being used in a context where only pointers are allowed.

Example:
```
void fn( void *p )
{
    int a[10];

    a = 0;
    a = p;
    a++;
}
```

231 *user-defined conversion has a return type*

A user-defined conversion cannot be declared with a return type. The "return type" of the user-defined conversion is implicit in the name of the user-defined conversion.

Example:
```
struct S {
    int operator int(); // cannot have return type
};
```

232 *user-defined conversion must be a function*

The operator name describing a user-defined conversion can only be used to designate functions.

Example:
```
// operator char can only be a function
int operator char = 9;
```

233 *user-defined conversion has an argument list*

A user-defined conversion cannot have an argument list. Since user-defined conversions can only be non-static member functions, they have an implicit ***this*** argument.

Example:
```
struct S {
    operator int( S& ); // cannot have arguments
};
```

234 *destructor cannot have a return type*

A destructor cannot have a return type (even ***void***). The destructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:
```
struct S {
    void* ~S();
};
```

235 *destructor must be a function*

The tilde ('~') style of name is reserved for declaring destructor functions. Variable names cannot make use of the destructor style of names.

Example:
```
struct S {
    int ~S; // illegal
};
```

236 *destructor has an argument list*

A destructor cannot have an argument list. Since destructors can only be non-static member functions, they have an implicit ***this*** argument.

Example:
```
struct S {
    ~S( S& );
};
```

237 *'%N' must be a function*

The ***operator*** style of name is reserved for declaring operator functions. Variable names cannot make use of the ***operator*** style of names.

Example:
```
struct S {
    int operator+;  // illegal
};
```

238 *'%N' is not a function*

The compiler has detected what looks like a function body. The message is a result of not finding a function being declared. This can happen in many ways, such as dropping the ':' before defining base classes, or dropping the '=' before initializing a structure via a braced initializer.

Example:

```
struct D B { int i; };
```

239 *nested type class '%s' has not been declared*

A nested class has not been found but is required by the use of repeated '::' operators. The construct "A::B::C" requires that 'A' be a class type, and 'B' be a nested class within the scope of 'A'.

Example:

```
struct B {
    static int b;
};
struct A : public B {
};
int A::B::b = 2;    // B not nested in A
```

The preceding example is illegal; the following is legal

Example:

```
struct A {
    struct B {
    static int b;
    };
};
int A::B::b = 2;    // B nested in A
```

240 *enum '%s' has not been declared*

An elaborated reference to an ***enum*** could not be satisfied. All enclosing scopes have been searched for an ***enum*** name. Visible variable declarations do not affect the search.

Example:

```
struct D {
    int i;
    enum E { e1, e2, e3 };
};
enum E enum_var;    // E not visible
```

241 *class or namespace '%s' has not been declared*

The construct "A::B::C" requires that 'A' be a class type or a namespace, and 'B' be a nested class or namespace within the scope of 'A'. The reference to 'A' could not be satisfied. All enclosing scopes have been searched for a ***class*** or ***namespace*** name. Visible variable declarations do not affect the search.

Example:
```
struct A{ int a; };

int b;
int c = B::A::b;
```

242 *only one initializer argument allowed*

The comma (',') in a function like cast is treated like an argument list comma (','). If a comma expression is desired, use parentheses to enclose the comma expression.

Example:
```
void fn()
{
    int a;

    a = int( 1, 2 );         // Error!
    a = int( ( 1, 2 ) );     // OK
}
```

243 *default arguments are not part of a function's type*

This message indicates that a declaration has been found that requires default arguments to be part of a function's type. Either declaring a function ***typedef*** or a pointer to a function with default arguments are examples of incorrect declarations.

Example:
```
typedef int TD( int, int a = 14 );
int (*p)( int, int a = 14 ) = 0;
```

244 *missing default arguments*

Gaps in a succession of default arguments are not allowed in the C++ language.

Example:
```
void fn( int = 1, int, int = 3 );
```

245 *overloaded operator cannot have default arguments*

Preventing overloaded operators from having default arguments enforces the property that binary operators will only be called from a use of a binary operator. Allowing default arguments would allow a binary ***operator*** **+** to function as a unary ***operator*** **+.**

Example:
```
class C
{
public:
    C operator +( int a = 10 );
};
```

246 *left expression is not a pointer to a constant object*

One cannot assign a pointer to a constant type to a pointer to a non-constant type. This would allow a constant object to be modified via the non-constant pointer. Use a cast if this is absolutely necessary.

Example:
```
char* fun( const char* p )
{
    char* q;
    q = p;
    return q;
}
```

247 *cannot redefine default argument for '%S'*

Default arguments can only be defined once in a program regardless of whether the value of the default argument is identical.

Example:
```
static int foo( int a = 10 );
static int foo( int a = 10 )
{
    return a+a;
}
```

248 *using default arguments would be overload ambiguous with '%S'*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name.

Example:
```
void fn( int );
void fn( int, int = 1 );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' without any default arguments or the second 'fn' with a default argument applied.

249 *using default arguments would be overload ambiguous with '%S' using default arguments*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name with default arguments.

Example:
```
void fn( int, int = 1 );
void fn( int, char = 'a' );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' with a default argument or the second 'fn' with a default argument applied.

250 *missing default argument for '%S'*

In C++, one is allowed to add default arguments to the right hand arguments of a function declaration in successive declarations. The message indicates that the declaration is only valid if there was a default argument previously declared for the next argument.

Example:

```
void fn1( int    , int     );
void fn1( int    , int = 3 );
void fn1( int = 2, int     );   // OK

void fn2( int    , int     );
void fn2( int = 2, int     );   // Error!
```

251 *enum references must have an identifier*

There is no way to reference an anonymous ***enum.*** If all enums are named, the cause of this message is most likely a missing identifier.

Example:

```
enum   { X, Y, Z }; // anonymous enum
void fn()
{
    enum *p;
}
```

252 *class declaration has not been seen for '~%s'*

A destructor has been used in a context where its class is not visible.

Example:

```
class C;

void fun( C* p )
{
    p->~S();
}
```

253 *'::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** member functions. The Open Watcom C++ compiler also allows code that is qualified with its own class so that declarations can be moved in and out of class definitions easily.

Example:
```
struct N {
    void bar();
};
struct S {
    void S::foo() { // OK
    }
    void N::bar() { // error
    }
};
```

254 *'%S' has not been declared as a member*

In a definition of a class member, the indicated declaration must already have been declared when the class was defined.

Example:
```
class C
{
public:
    int c;
    int goop();
};
int C::x = 1;
C::not_decled() { }
```

255 *default argument expression cannot use function argument '%S'*

Default arguments must be evaluated at each call. Since the order of evaluation for arguments is undefined, a compiler must diagnose all default arguments that depend on other arguments.

Example:
```
void goop( int d )
{
    struct S {
    // cannot access "d"
    int foo( int c, int b = d )
        {
        return b + c;
        };
    };
}
```

256 *default argument expression cannot use local variable '%S'*

Default arguments must be evaluated at each call. Since a local variable is not always available in all contexts (e.g., file scope initializers), a compiler must diagnose all default arguments that depend on local variables.

Example:

```
void goop( void )
{
    int a;
    struct S {
    // cannot access "a"
    int foo( int c, int b = a )
        {
        return b + c;
        };
    };
}
```

257 *access declarations may only be 'public' or 'protected'*

Access declarations are used to increase access. A ***private*** access declaration is useless because there is no access level for which ***private*** is an increase in access.

Example:

```
class Base
{
    int pri;
protected:
    int pro;
public:
    int pub;
};
class Derived : public Base
{
    private: Base::pri;
};
```

258 *cannot declare both a function and variable of the same name ('%N')*

Functions can be overloaded in C++ but they cannot be overloaded in the presence of a variable of the same name. Likewise, one cannot declare a variable in the same scope as a set of overloaded functions of the same name.

Example:

```
int foo();
int foo;
struct S {
    int bad();
    int bad;
};
```

259 *class in access declaration ('%T') must be a direct base class*

Access declarations can only be applied to direct (immediate) base classes.

Example:

```
struct B {
    int f;
};
struct C : B {
    int g;
};
struct D : private C {
    B::f;
};
```

In the above example, "C" is a direct base class of "D" and "B" is a direct base class of "C", but "B" is not a direct base class of "D".

260 *overloaded functions ('%N') do not have the same access*

If an access declaration is referencing a set of overloaded functions, then they all must have the same access. This is due to the lack of a type in an access declaration.

Example:

```
class C
{
    static int foo( int );      // private
public:
    static int foo( float );    // public
};

class B : private C
{
public: C::foo;
};
```

261 *cannot grant access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : private Base
{
    public: Base::pub;       // ok
    public: Base::pro;       // changes access
};
```

262 *cannot reduce access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : public Base
{
    protected: Base::pub;   // changes access
    protected: Base::pro;   // ok
};
```

263 *nested class '%N' has not been defined*

The current state of the C++ language supports nested types. Unfortunately, this means that some working C code will not work unchanged.

Example:

```
struct S {
    struct T;
    T *link;
};
```

In the above example, the class "T" will be reported as not being defined by the end of the class declaration. The code can be corrected in the following manner.

Example:

```
struct S {
    struct T;
    T *link;
    struct T {
    };
};
```

264 *user-defined conversion must be a non-static member function*

A user-defined conversion is a special member function that allows the class to be converted implicitly (or explicitly) to an arbitrary type. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:
```
struct S
{
    static operator int();
};
```

265 *destructor must be a non-static member function*

A destructor is a special member function that will perform cleanup on a class before the storage for the class will be released. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:
```
struct S
{
    static ~S();
};
```

266 *'%N' must be a non-static member function*

The operator function in the message is restricted to being a non-static member function. This usually means that the operator function is treated in a special manner by the compiler.

Example:
```
class C
{
public:
    static operator =( C&, int );
};
```

267 *'%N' must have one argument*

The operator function in the message is only allowed to have one argument. An operator like ***operator ~*** is one such example because it represents a unary operator.

Example:
```
class C
{
public: int c;
};
C& operator~( const C&, int );
```

268 *'%N' must have two arguments*

The operator function in the message must have two arguments. An operator like ***operator*** += is one such example because it represents a binary operator.

Example:
```
class C
{
public: int c;
};
C& operator += ( const C& );
```

269 *'%N' must have either one argument or two arguments*

The operator function in the message must have either one argument or two arguments. An operator like ***operator*** + is one such example because it represents either a unary or a binary operator.

Example:
```
class C
{
public: int c;
};
C& operator+( const C&, int, float );
```

270 *'%N' must have at least one argument*

The ***operator new*** and ***operator new []*** member functions must have at least one argument for the size of the allocation. After that, any arguments are up to the programmer. The extra arguments can be supplied in a ***new*** expression via the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void * operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

271 *'%N' must have a return type of void*

The C++ language requires that ***operator delete*** and ***operator delete []*** have a return type of ***void.***

Example:

```
class C
{
public:
    int c;
    C* operator delete( void* );
    C* operator delete []( void* );
};
```

272 *'%N' must have a return type of pointer to void*

The C++ language requires that both ***operator new*** and ***operator new []*** have a return type of `void *`.

Example:

```
#include <stddef.h>
class C
{
public:
    int c;
    C* operator new( size_t size );
    C* operator new []( size_t size );
};
```

273 *the first argument of '%N' must be of type size_t*

The C++ language requires that the first argument for ***operator new*** and ***operator new []*** be of the type "size_t". The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:
```
    void *operator new( int size );
    void *operator new( double size, char c );
    void *operator new []( int size );
    void *operator new []( double size, char c );
```

274 *the first argument of '%N' must be of type pointer to void*

The C++ language requires that the first argument for ***operator delete*** and ***operator delete []*** be a `void *`.

Example:
```
    class C;
    void operator delete( C* );
    void operator delete []( C* );
```

275 *the second argument of '%N' must be of type size_t*

The C++ language requires that the second argument for ***operator delete*** and ***operator delete []*** be of type "size_t". The two argument form of ***operator delete*** and ***operator delete []*** is optional and it can only be present inside of a class declaration. The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:
```
    struct S {
        void operator delete( void *, char );
        void operator delete []( void *, char );
    };
```

276 *the second argument of 'operator ++' or 'operator --' must be int*

The C++ language requires that the second argument for ***operator ++*** be ***int.*** The two argument form of ***operator ++*** is used to overload the postfix operator "++". The postfix operator "--" can be overloaded similarly.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( C&, unsigned );
```

277 *return type of '%S' must allow the '->' operator to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.***

Example:

```
struct S {
    int a;
    S *operator ->();
};

void fn( S &q )
{
    q->a = 1; // becomes (q.operator ->())->a = 1;
}
```

278 *'%N' must take at least one argument of a class/enum or a reference to a class/enum*

Overloaded operators can only be defined for classes and enumerations. At least one argument, must be a class or an enum type in order for the C++ compiler to distinguish the operator from the built-in operators.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( unsigned, int );
```

279 *too many initializers*

The compiler has detected extra initializers.

Example:
```
int a[3] = { 1, 2, 3, 4 };
```

280 *too many initializers for character string*

A string literal used in an initialization of a character array is viewed as providing the terminating null character. If the number of array elements isn't enough to accept the terminating character, this message is output.

Example:
```
char ac[3] = "abc";
```

281 *expecting '%s' but found expression*

This message is output when some bracing or punctuation is expected but an expression was encountered.

Example:
```
int b[3] = 3;
```

282 *anonymous struct/union member '%N' cannot be declared in this class*

An anonymous member cannot be declared with the same name as its containing class.

Example:
```
struct S {
    union {
    int S;        // Error!
    char b;
    };
};
```

283 *unexpected '%s' during initialization*

This message is output when some unexpected bracing or punctuation is encountered during initialization.

Example:
```
int e = { { 1 };
```

284 *nested type '%N' cannot be declared in this class*

A nested type cannot be declared with the same name as its containing class.

Example:
```
struct S {
    typedef int S;  // Error!
};
```

285 *enumerator '%N' cannot be declared in this class*

An enumerator cannot be declared with the same name as its containing class.

Example:
```
struct S {
    enum E {
    S,  // Error!
    T
    };
};
```

286 *static member '%N' cannot be declared in this class*

A static member cannot be declared with the same name as its containing class.

Example:
```
struct S {
    static int S;   // Error!
};
```

287 *constructor cannot have a return type*

A constructor cannot have a return type (even ***void***). The constructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:
```
class C {
public:
    C& C( int );
};
```

288 *constructor cannot be a static member*

A constructor is a special member function that takes raw storage and changes it into an instance of a class. In order to do this, it must have access to storage for the instance of the class so it is restricted to being a non-static member function.

Example:
```
class C {
public:
    static C( int );
};
```

289 *invalid copy constructor argument list (causes infinite recursion)*

A copy constructor's first argument must be a reference argument. Furthermore, any default arguments must also be reference arguments. Without the reference, a copy constructor would require a copy constructor to execute in order to prepare its arguments. Unfortunately, this would be calling itself since it is the copy constructor.

Example:
```
struct S {
    S( S const & );    // copy constructor
};
```

290 *constructor cannot be declared const or volatile*

A constructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

Example:
```
class C {
public:
    C( int ) const;
    C( float ) volatile;
};
```

291 *constructor cannot be virtual*

Virtual functions cannot be called for an object before it is constructed. For this reason, a virtual constructor is not allowed in the C++ language. Techniques for simulating a virtual constructor are known, one such technique is described in the ARM p.263.

Example:
```
class C {
public:
    virtual C( int );
};
```

292 *types do not match in simple type destructor*

A simple type destructor is available for "destructing" simple types. The destructor has no effect. Both of the types must be identical, for the destructor to have meaning.

Example:
```
void foo( int *p )
{
    p->int::~double();
}
```

293 *overloaded operator is ambiguous for operands used*

The Open Watcom C++ compiler performs exhaustive analysis using formalized techniques in order to decide what implicit conversions should be applied for overloading operators. Because of this, Open Watcom C++ detects ambiguities that may escape other C++ compilers. The most common ambiguity that Open Watcom C++ detects involves classes having constructors with single arguments and a user-defined conversion.

Example:
```
struct S {
    S(int);
    operator int();
    int a;
};

int fn( int b, int i, S s )
{
    //    i    : s.operator int()
    // OR S(i) : s
    return b ? i : s;
}
```

In the above example, "i" and "s" must be brought to a common type. Unfortunately, there are two common types so the compiler cannot decide which one it should choose, hence an ambiguity.

294 *feature not implemented*

The compiler does not support the indicated feature.

295 *invalid friend declaration*

This message indicates that the compiler found extra declaration specifiers like ***auto, float,*** or ***const*** in the friend declaration.

Example:
```
class C
{
    friend float;
};
```

296 *friend declarations may only be declared in a class*

This message indicates that a ***friend*** declaration was found outside a class scope (i.e., a class definition). Friends are only meaningful for class types.

Example:
```
extern void foo();
friend void foo();
```

297 *class friend declaration needs 'class' or 'struct' keyword*

The C++ language has evolved to require that all friend class declarations be of the form "class S" or "struct S". The Open Watcom C++ compiler accepts the older syntax with a warning but rejects the syntax in pure ISO/ANSI C++ mode.

Example:
```
struct S;
struct T {
    friend S;   // should be "friend class S;"
};
```

298 *class friend declarations cannot contain a class definition*

A class friend declaration cannot define a new class. This is a restriction required in the C++ language.

Example:
```
struct S {
    friend struct X {
    int f;
    };
};
```

299 *'%T' has already been declared as a friend*

The class in the message has already been declared as a friend. Remove the extra friend declaration.

Example:
```
class S;
class T {
    friend class S;
    int tv;
    friend class S;
};
```

300 *function '%S' has already been declared as a friend*

The function in the message has already been declared as a friend. Remove the extra friend declaration.

Example:
```
extern void foo();
class T {
    friend void foo();
    int tv;
    friend void foo();
};
```

301 *'friend', 'virtual' or 'inline' modifiers are not part of a function's type*

This message indicates that the modifiers may be incorrectly placed in the declaration. If the declaration is intended, it cannot be accepted because the modifiers can only be applied to functions that have code associated with them.

Example:
```
typedef friend (*PF)( void );
```

302 *cannot assign right expression to element on left*

This message indicates that the assignment cannot be performed. It usually arises in assignments of a class type to an arithmetic type.

Example:
```
struct S
{   int sv;
};
S s;
int foo()
{
    int k;
    k = s;
    return k;
}
```

303 *constructor is ambiguous for operands used*

The operands provided for the constructor did not select a unique constructor.

Example:
```
struct S {
    S(int);
    S(char);
};

S x = S(1.0);
```

304 *class '%s' has not been defined*

The name before a '::' scope resolution operator must be defined unless a member pointer is being declared.

Example:
```
struct S;

int S::* p; // OK
int S::a = 1;   // Error!
```

305 *all bit-fields in a union must be named*

This is a restriction in the C++ language. The same effect can be achieved with a named bitfield.

Example:
```
union u
{   unsigned bit1 :10;
    unsigned :6;
};
```

306 *cannot convert expression to type of cast*

The cast is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended cast.

Example:

```
struct T {
};

void fn()
{
    T y = (T) 0;
}
```

307 *conversion ambiguity: [expression] to [cast type]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

void fn()
{
    S x = (S) 1.0;
}
```

308 *an anonymous class without a declarator is useless*

There is no way to reference the type in this kind of declaration. A name must be provided for either the class or a variable using the class as its type.

Example:

```
struct {
    int a;
    int b;
};
```

309 *global anonymous union must be declared static*

This is a restriction in the C++ language. Since there is no unique name for the anonymous union, it is difficult for C++ translators to provide a correct implementation of external linkage anonymous unions.

Example:
```
static union {
    int a;
    int b;
};
```

310 *anonymous struct/union cannot have storage class in this context*

Anonymous unions (or structs) declared in class scopes cannot be ***static.*** Any other storage class is also disallowed.

Example:
```
struct S {
    static union {
    int iv;
    unsigned us;
    };
};
```

311 *union contains a protected member*

A union cannot have a ***protected*** member because a union cannot be a base class.

Example:
```
static union {
    int iv;
protected:
    unsigned sv;
} u;
```

312 *anonymous struct/union contains a private member '%S'*

An anonymous union (or struct) cannot have member functions or friends so it cannot have ***private*** members since no code could access them.

Example:
```
static union {
    int iv;
private:
    unsigned sv;
};
```

313 *anonymous struct/union contains a function member '%S'*

An anonymous union (or struct) cannot have any function members. This is a restriction in the C++ language.

Example:

```
static union {
     int iv;
     void foo();     // error
     unsigned sv;
};
```

314 *anonymous struct/union contains a typedef member '%S'*

An anonymous union (or struct) cannot have any nested types. This is a restriction in the C++ language.

Example:

```
static union {
     int iv;
     unsigned sv;
     typedef float F;
     F fv;
};
```

315 *anonymous struct/union contains an enumeration member '%S'*

An anonymous union (or struct) cannot have any enumeration members. This is a restriction in the C++ language.

Example:

```
static union {
     int iv;
     enum choice { good, bad, indifferent };
     choice c;
     unsigned sv;
};
```

316 *anonymous struct/union member '%s' is not distinct in enclosing scope*

Since an anonymous union (or struct) provides its member names to the enclosing scope, the names must not collide with other names in the enclosing scope.

Example:
```
int iv;
unsigned sv;
static union {
    int iv;
    unsigned sv;
};
```

317 *unions cannot have members with destructors*

A union should only be used to organize memory in C++. Allowing union members to have destructors would mean that the same piece of memory could be destructed twice.

Example:
```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    ~T();
};
static union
{
    S su;
    T tu;
};
```

318 *unions cannot have members with user-defined assignment operators*

A union should only be used to organize memory in C++. Allowing union members to have assignment operators would mean that the same piece of memory could be assigned twice.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    int tv;
    operator = ( int );
    operator = ( float );
};
static union
{
    S su;
    T tu;
} u;
```

319 *anonymous struct/union cannot have any friends*

An anonymous union (or struct) cannot have any friends. This is a restriction in the C++ language.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
static union {
    S su1;
    S su2;
    friend class S;
};
```

320 *specific versions of template classes can only be defined in file scope*

Currently, specific versions of class templates can only be declared at file scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

Example:

```
template <class G> class S {
    G x;
};

struct Q {
    struct S<int> {
    int x;
    };
};

void foo()
{
    struct S<double> {
    double x;
    };
}
```

321 *anonymous union in a function may only be static or auto*

The current C++ language definition only allows ***auto*** anonymous unions. The Open Watcom C++ compiler allows ***static*** anonymous unions. Any other storage class is not allowed.

322 *static data members are not allowed in a local class*

Static data members are not allowed in a local class because there is no way to define the static member in file scope.

Example:

```
int foo()
{
    struct local {
    static int s;
    };

    local lv;

    lv.s = 3;
    return lv.s;
}
```

323 *conversion ambiguity: [return value] to [return type of function]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:
```
struct S {
    S(int);
    S(char);
};

S fn()
{
    return 1.0;
}
```

324 *conversion of return value is impossible*

The return is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended return type.

Example:
```
struct T {
};

T fn()
{
    return 0;
}
```

325 *function cannot return a pointer based on __self*

A function cannot return a pointer that is based on ***__self.***

Example:
```
void __based(__self) *fn( unsigned );
```

326 *defining '%S' is not possible because its type has unknown size*

In order to define a variable, the size must be known so that the correct amount of storage can be reserved.

Example:
```
class S;
S sv;
```

327 *typedef cannot be initialized*

Initializing a ***typedef*** is meaningless in the C++ language.

Example:
```
typedef int INT = 15;
```

328 *storage class of '%S' conflicts with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to storage class. A symbol cannot be both ***static*** and ***extern.***

329 *modifiers of '%S' conflict with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to modifiers. Correct the program by using the same modifiers for both declarations.

330 *function cannot be initialized*

A function cannot be initialized with an initializer syntax intended for variables. A function body is the only way to provide a definition for a function.

331 *access permission of nested class '%T' conflicts with previous declaration*

Example:
```
struct S {
    struct N;    // public
private:
    struct N {   // private
    };
};
```

332 **** FATAL *** internal error in front end*

If this message appears, please report the problem directly to the Open Watcom development team. See http://www.openwatcom.org/.

333 *cannot convert argument to type specified in function prototype*

It is impossible to convert the indicated argument in the function.

Example:
```
extern int foo( int& );

extern int m;
extern int n;

int k = foo( m + n );
```

In the example, the value of "m+n" cannot be converted to a reference (it could be converted to a constant reference), as shown in the following example.

Example:
```
extern int foo( const int& );

extern int m;
extern int n;

int k = foo( m + n );
```

334 *conversion ambiguity: [argument] to [argument type in prototype]*

An argument in the function call could not be converted since there is more than one constructor or user-defined conversion which could be used to convert the argument.

Example:

```
struct S;

struct T
{
    T( S& );
};

struct S
{
    operator T();
};

S s;
extern int foo( T );
int k = foo( s );   // ambiguous
```

In the example, the argument "s" could be converted by both the constructor in class "T" and by the user-conversion in class "S".

335 *cannot be based on based pointer '%S'*

A based pointer cannot be based on another based pointer.

Example:

```
__segment s;
void __based(s) *p;
void __based(p) *q;
```

336 *declaration specifiers are required to declare '%N'*

The compiler has detected that the name does not represent a function. Only function declarations can leave out declaration specifiers. This error also shows up when a typedef name declaration is missing.

Example:

```
x;
typedef int;
```

337 *static function declared in block scope*

The C++ language does not allow static functions to be declared in block scope. This error can be triggered when the intent is to define a ***static*** variable. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as function prototypes. A work-around for this problem is contained in the example.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    static S a( C() );  // function prototype!
    static S b( (C()) );// variable definition
}
```

338 *cannot define a __based reference*

A C++ reference cannot be based on anything. Based modifiers can only be used with pointers.

Example:

```
__segment s;
void fn( int __based(s) & x );
```

339 *conversion ambiguity: conversion to common pointer type*

A conversion to a common base class of two different pointers has been attempted. The pointer conversion could not be performed because the destination type points to an ambiguous base class of one of the source types.

340 *cannot construct object from argument(s)*

There is not an appropriate constructor for the set of arguments provided.

341 *number of arguments for function '%S' is incorrect*

The number of arguments in the function call does not match the number declared for the indicated non-overloaded function.

Example:
```
extern int foo( int, int );
int k = foo( 1, 2, 3 );
```

In the example, the function was declared to have two arguments. Three arguments were used in the call.

342 *private base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:
```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

extern Der *pd;
Priv *pp = (Priv*)pd;
```

343 *private base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

Priv *foo( Der *p )
{
    return p;
}
```

344 *cannot subtract pointers to different objects*

Pointer subtraction can be performed only for objects of the same type.

Example:

```
#include <stddef.h>
ptrdiff_t diff( float *fp, int *ip )
{
    return fp - ip;
}
```

In the example, a diagnostic results from the attempt to subtract a pointer to an ***int*** object from a pointer to a ***float*** object.

345 *private base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

int foo( Der *pd, Priv *pp )
{
    return pd == pp;
}
```

346 *protected base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

extern Der *pd;
Prot *pp = (Prot*)pd;
```

347 *protected base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

Prot *foo( Der *p )
{
    return p;
}
```

348 *cannot define a member pointer with a memory model modifier*

A member pointer describes how to access a field from a class. Because of this a member pointer must be independent of any memory model considerations.

Example:

```
struct S;

int near S::*mp;
```

349 *protected base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

int foo( Der *pd, Prot *pp )
{
    return pd == pp;
}
```

350 *non-type parameter supplied for a type argument*

A non-type parameter (e.g., an address or a constant expression) has been supplied for a template type argument. A type should be used instead.

351 *type parameter supplied for a non-type argument*

A type parameter (e.g., ***int***) has been supplied for a template non-type argument. An address or a constant expression should be used instead.

352 *cannot access enclosing function's auto variable '%S'*

A local class member function cannot access its enclosing function's automatic variables.

Example:

```
void goop( void )
{
    int a;
    struct S
    {
    int foo( int c, int b )
        {
        return b + c + a;
        };
    };
}
```

353 *cannot initialize pointer to non-constant with a pointer to constant*

A pointer to a non-constant type cannot be initialized with a pointer to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:

```
extern const int *pic;
extern int *pi = pic;
```

354 *pointer expression is always >= 0*

The indicated pointer expression will always be true because the pointer value is always treated as an unsigned quantity, which will be greater or equal to zero.

Example:
```
extern char *p;
unsigned k = ( 0 <= p );    // always 1
```

355 *pointer expression is never < 0*

The indicated pointer expression will always be false because the pointer value is always treated as an unsigned quantity, which will be greater or equal zero.

Example:
```
extern char *p;
unsigned k = ( 0 >= p );   // always 0
```

356 *type cannot be used in this context*

This message is issued when a type name is being used in a context where a non-type name should be used.

Example:
```
struct S {
    typedef int T;
};

void fn( S *p )
{
    p->T = 1;
}
```

357 *virtual function may only be declared in a class*

Virtual functions can only be declared inside of a class. This error may be a result of forgetting the "C::" qualification of a virtual function's name.

Example:

```
virtual void foo();
struct S
{
    int f;
    virtual void bar();
};
virtual void bar()
{
    f = 9;
}
```

358 *'%T' referenced as a union*

A class type defined as a ***class*** or ***struct*** has been referenced as a ***union*** (i.e., union S).

Example:

```
struct S
{
    int s1, s2;
};
union S var;
```

359 *union '%T' referenced as a class*

A class type defined as a ***union*** has been referenced as a ***struct*** or a ***class*** (i.e., class S).

Example:

```
union S
{
    int s1, s2;
};
struct S var;
```

360 *typedef '%N' defined without an explicit type*

The typedef declaration was found to not have an explicit type in the declaration. If ***int*** is the desired type, use an explicit ***int*** keyword to specify the type.

Example:

```
typedef T;
```

361 *member function was not defined in its class*

Member functions of local classes must be defined in their class if they will be defined at all. This is a result of the C++ language not allowing nested function definitions.

Example:

```
void fn()
{
    struct S {
    int bar();
    };
}
```

362 *local class can only have its containing function as a friend*

A local class can only be referenced from within its containing function. It is impossible to define an external function that can reference the type of the local class.

Example:

```
extern void ext();
void foo()
{
    class S
    {
    int s;
    public:
    friend void ext();
    int q;
    };
}
```

363 *local class cannot have '%S' as a friend*

The only classes that a local class can have as a friend are classes within its own containing scope.

Example:
```
struct ext
{
    goop();
};
void foo()
{
    class S
    {
    int s;
    public:
    friend class ext;
    int q;
    };
}
```

364 *adjacent >=, <=, >, < operators*

This message is warning about the possibility that the code may not do what was intended. An expression like "a > b > c" evaluates one relational operator to a 1 or a 0 and then compares it against the other variable.

Example:
```
extern int a;
extern int b;
extern int c;
int k = a > b > c;
```

365 *cannot access enclosing function's argument '%S'*

A local class member function cannot access its enclosing function's arguments.

Example:
```
void goop( int d )
{
    struct S
    {
    int foo( int c, int b )
        {
        return b + c + d;
        };
    };
}
```

366 *support for switch '%s' is not implemented*

Actions for the indicated switch have not been implemented. The switch is supported for compatibility with the Open Watcom C compiler.

367 *conditional expression in if statement is always true*

The compiler has detected that the expression will always be true. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially true expressions.

Example:
```
#define TEST 143
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

368 *conditional expression in if statement is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

Example:
```
#define TEST 14-14
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

369 *selection expression in switch statement is a constant value*

The expression in the ***switch*** statement is a constant. This means that only one case label will be executed. If this is not the expected behaviour, check the switch expression.

Example:
```
#define TEST 0
int foo( int a, int b )
{
    switch ( TEST ) {
      case 0:
    return a;
      default:
    return b;
    }
}
```

370 *constructor is required for a class with a const member*

If a class has a constant member, a constructor is required in order to initialize it.

Example:
```
struct S
{
    const int s;
    int i;
};
```

371 *constructor is required for a class with a reference member*

If a class has a reference member, a constructor is required in order to initialize it.

Example:
```
struct S
{
    int& r;
    int i;
};
```

372 *inline member friend function '%S' is not allowed*

A friend that is a member function of another class cannot be defined. Inline friend rules are currently in flux so it is best to avoid inline friends.

373 *invalid modifier for auto variable*

An automatic variable cannot have a memory model adjustment because they are always located on the stack (or in a register). There are also other types of modifiers that are not allowed for auto variables such as thread-specific data modifiers.

Example:

```
int fn( int far x )
{
    int far y = x + 1;
    return y;
}
```

374 *object (or object pointer) required to access non-static data member*

A reference to a member in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m;
    static void fn()
    {
    m = 1;  // Error!
    }
};
```

375 *user-defined conversion has not been declared*

The named user-defined conversion has not been declared in the class of any of its base classes.

Example:

```
struct S {
    operator int();
    int a;
};

double fn( S *p )
{
    return p->operator double();
}
```

376 *virtual function must be a non-static member function*

A member function cannot be both a ***static*** function and a ***virtual*** function. A static member function does not have a ***this*** argument whereas a ***virtual*** function must have a ***this*** argument so that the virtual function table can be accessed in order to call it.

Example:

```
struct S
{
    static virtual int foo();   // error
    virtual int bar();      // ok
    static int stat();      // ok
};
```

377 *protected base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
protected:
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is protected.

378 *private base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is private.

379 *delete expression will invoke a non-virtual destructor*

In C++, it is possible to assign a base class pointer the value of a derived class pointer so that code that makes use of base class virtual functions can be used. A problem that occurs is that a ***delete*** has to know the correct size of the type in some instances (i.e., when a two argument version of ***operator delete*** is defined for a class). This problem is solved by requiring that a destructor be defined as ***virtual*** if polymorphic deletes must work. The ***delete*** expression will virtually call the correct destructor, which knows the correct size of the complete object. This message informs you that the class you are deleting has virtual functions but it has a non-virtual destructor. This means that the delete will not work correctly in all circumstances.

Example:

```
#include <stddef.h>

struct B {
    int b;
    void operator delete( void *, size_t );
    virtual void fn();
    ~B();
};
struct D : B {
    int d;
    void operator delete( void *, size_t );
    virtual void fn();
    ~D();
};

void dfn( B *p )
{
    delete p;   // could be a pointer to D!
}
```

380 *'offsetof' is not allowed for a function*

A member function does not have an offset like simple data members. If this is required, use a member pointer.

Example:

```
#include <stddef.h>

struct S
{
    int fun();
};

int s = offsetof( S, fun );
```

381 *'offsetof' is not allowed for an enumeration*

An enumeration does not have an offset like simple data members.

Example:

```
#include <stddef.h>

struct S
{
    enum SE { S1, S2, S3, S4 };
    SE var;
};

int s = offsetof( S, SE );
```

382 *could not initialize for code generation*

The source code has been parsed and fully analysed when this error is emitted. The compiler attempted to start generating object code but due to some problem (e.g., out of memory, no file handles) could not initialize itself. Try changing the compilation environment to eliminate this error.

383 *'offsetof' is not allowed for an undefined type*

The class type used in ***offsetof*** must be completely defined, otherwise data member offsets will not be known.

Example:

```
#include <stddef.h>

struct S {
    int a;
    int b;
    int c[ offsetof( S, b ) ];
};
```

384 *attempt to override virtual function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading. The exception to this rule in the C++ language involves restricted changes in the return type of virtual functions. The derived virtual function's return type can be derived from the return type of the base virtual function.

Example:

```
struct B {
    virtual B *fn();
};
struct D : B {
    virtual D *fn();
};
```

385 *attempt to overload function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading.

Example:

```
int foo( char );
unsigned foo( char );
```

386 *attempt to use pointer to undefined class*

An attempt was made to indirect or increment a pointer to an undefined class. Since the class is undefined, the size is not known so the compiler cannot compile the expression properly.

Example:

```
class C;
extern C* pc1;
C* pc2 = ++pc1;     // C not defined

int foo( C*p )
{
    return p->x;    // C not defined
}
```

387 *expression is useful only for its side effects*

The indicated expression is not meaningful. The expression, however, does contain one or more side effects.

Example:

```
extern int* i;
void func()
{
    *(i++);
}
```

In the example, the expression is a reference to an integer which is meaningless in itself. The incrementation of the pointer in the expression is a side effect.

388 *integral constant will be truncated during assignment or initialization*

This message indicates that the compiler knows that a constant value will not be preserved after the assignment. If this is acceptable, cast the constant value to the appropriate type in the assignment.

Example:

```
unsigned char c = 567;
```

389 *integral value may be truncated during assignment or initialization*

This message indicates that the compiler knows that all values will not be preserved after the assignment. If this is acceptable, cast the value to the appropriate type in the assignment.

Example:

```
extern unsigned s;
unsigned char c = s;
```

390 *cannot generate default constructor to initialize '%T' since constructors were declared*

A default constructor will not be generated by the compiler if there are already constructors declared. Try using default arguments to change one of the constructors to a default constructor or define a default constructor explicitly.

Example:
```
class C {
    C( const C& );
public :
    int c;
};
C cv;
```

391 *assignment found in boolean expression*

This is a construct that can lead to errors if it was intended to be an equality (using "==") test.

Example:
```
int foo( int a, int b )
{
    if( a = b ) {
    return b;
    }
    return a;       // always return 1 ?
}
```

392 *definition: '%F'*

This informational message indicates where the symbol in question was defined. The message is displayed following an error or warning diagnostic for the symbol in question.

Example:
```
static int a = 9;
int b = 89;
```

The variable 'a' is not referenced in the preceding example and so will cause a warning to be generated. Following the warning, the informational message indicates the line at which 'a' was declared.

393 *included from %s(%u)*

This informational message indicates the line number of the file including the file in which an error or warning was diagnosed. A number of such messages will allow you to trace back through the ***#include*** directives which are currently being processed.

394 *reference object must be initialized*

A reference cannot be set except through initialization. Also references cannot be 0 so they must always be initialized.

Example:
```
int & ref;
```

395 *option requires an identifier*

The specified option is not recognized by the compiler since there was no identifier after it (i.e., "-nt=module").

396 *'main' cannot be overloaded*

There can only be one entry point for a C++ program. The "main" function cannot be overloaded.

Example:
```
int main();
int main( int );
```

397 *'new' expression cannot allocate a void*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot allocate an instance of ***void.***

Example:
```
void *p = new void;
```

398 *'new' expression cannot allocate a function*

A function type cannot be allocated since there is no meaningful size that can be used. The ***new*** expression can allocate a pointer to a function.

Example:
```
typedef int tdfun( int );
tdfun *tdv = new tdfun;
```

399 *'new' expression allocates a const or volatile object*

The pool of raw memory cannot be guaranteed to support ***const*** or ***volatile*** semantics. Usually ***const*** and ***volatile*** are used for statically allocated objects.

Example:
```
typedef const int con_int;
con_int* p = new con_int;
```

400 *cannot convert right expression for initialization*

The initialization is trying to convert an argument expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended conversion.

Example:
```
struct T {
};

T x = 0;
```

401 *conversion ambiguity: [initialization expression] to [type of object]*

The initialization caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:
```
struct S {
    S(int);
    S(char);
};

S x = 1.0;
```

402 *class template '%S' has already been declared as a friend*

The class template in the message has already been declared as a friend. Remove the extra friend declaration.

Example:
```
template <class T>
    class S;

class X {
    friend class S;
    int f;
    friend class S;
};
```

403 *private base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

404 *protected base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

405 *cannot return a pointer or reference to a constant object*

A pointer or reference to a constant object cannot be returned.

Example:
```
int *foo( const int *p )
{
    return p;
}
```

406 *cannot pass a pointer or reference to a constant object*

A pointer or reference to a constant object could not be passed as an argument.

Example:
```
int *bar( int * );
int *foo( const int *p )
{
    return bar( p );
}
```

407 *class templates must be named*

There is no syntax in the C++ language to reference an unnamed class template.

Example:

```
template <class T>
    class {
    };
```

408 *function templates can only name functions*

Variables cannot be overloaded in C++ so it is not possible to have many different instances of a variable with different types.

Example:

```
template <class T>
    T x[1];
```

409 *template argument '%S' is not used in the function argument list*

This restriction ensures that function templates can be bound to types during overload resolution. Functions currently can only be overloaded based on argument types.

Example:

```
template <class T>
    int foo( int * );
template <class T>
    T bar( int * );
```

410 *destructor cannot be declared const or volatile*

A destructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

411 *static member function cannot be declared const or volatile*

A static member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

412 *only member functions can be declared const or volatile*

A non-member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

413 *'const' or 'volatile' modifiers are not part of a function's type*

The ***const*** and ***volatile*** qualifiers for a function cannot be used in typedefs or pointers to functions. The trailing qualifiers are used to change the type of the implicit ***this*** argument so that member functions that do not modify the object can be declared accurately.

Example:

```
// const is illegal
typedef void (*baddcl)() const;

struct S {
    void fun() const;
    int a;
};

// "this" has type "S const *"
void S::fun() const
{
    this->a = 1;    // Error!
}
```

414 *type cannot be defined in an argument*

A new type cannot be defined in an argument because the type will only be visible within the function. This amounts to defining a function that can never be called because C++ uses name equivalence for type checking.

Example:

```
extern foo( struct S { int s; } );
```

415 *type cannot be defined in return type*

This is a restriction in the current C++ language. A function prototype should only use previously declared types in order to guarantee that it can be called from other functions. The restriction is required for templates because the compiler would have to wait until the end of a class definition before it could decide whether a class template or function template is being defined.

Example:
```
template <class T>
    class C {
    T value;
    } fn( T x ) {
    C y;

    y.x = 0;
    return y;
    };
```

A common problem that results in this error is to forget to terminate a class or enum definition with a semicolon.

Example:
```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

416 *data members cannot be initialized inside a class definition*

This message appears when an initialization is attempted inside of a class definition. In the case of static data members, initialization must be done outside the class definition. Ordinary data members can be initialized in a constructor.

Example:
```
struct S {
    static const int size = 1;
};
```

417 *only virtual functions may be declared pure*

The C++ language requires that all pure functions be declared virtual. A pure function establishes an interface that must consist of virtual functions because the functions are required to be defined in the derived class.

Example:
```
struct S {
    void foo() = 0;
};
```

418 *destructor is not declared in its proper class*

The destructor name is not declared in its own class or qualified by its own class. This is required in the C++ language.

419 *cannot call non-const function for a constant object*

A function that does not promise to not modify an object cannot be called for a constant object. A function can declare its intention to not modify an object by using the ***const*** qualifier.

Example:
```
struct S {
    void fn();
};

void cfn( const S *p )
{
    p->fn();    // Error!
}
```

420 *memory initializer list may only appear in a constructor definition*

A memory initializer list should be declared along with the body of the constructor function.

421 *cannot initialize member '%N' twice*

A member cannot be initialized twice in a member initialization list.

422 *cannot initialize base class '%T' twice*

A base class cannot be constructed twice in a member initialization list.

423 *'%T' is not a direct base class*

A base class initializer in a member initialization list must either be a direct base class or a virtual base class.

424 *'%N' cannot be initialized because it is not a member*

The name used in the member initialization list does not name a member in the class.

425 *'%N' cannot be initialized because it is a member function*

The name used in the member initialization list does not name a non-static data member in the class.

426 *'%N' cannot be initialized because it is a static member*

The name used in the member initialization list does not name a non-static data member in the class.

427 *'%N' has not been declared as a member*

This message indicates that the member does not exist in the qualified class. This usually occurs in the context of access declarations.

428 *const/reference member '%S' must have an initializer*

The ***const*** or reference member does not have an initializer so the constructor is not completely defined. The member initialization list is the only way to initialize these types of members.

429 *abstract class '%T' cannot be used as an argument type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as an argument type.

430 *abstract class '%T' cannot be used as a function return type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as a return type.

431 *defining '%S' is not possible because '%T' is an abstract class*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as either a member or a variable.

432 *cannot convert to an abstract class '%T'*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as the destination type in a conversion.

433 *mangled name for '%S' has been truncated*

The name used in the object file that encodes the name and full type of the symbol is often called a mangled name. The warning indicates that the mangled name had to be truncated due to limitations in the object file format.

434 *cannot convert to a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

435 *cannot convert a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

436 *cannot construct an abstract class*

An instance of an abstract class cannot be created because an abstract class can only be used as a base class.

437 *cannot construct an undefined class*

An instance of an undefined class cannot be created because the size is not known.

438 *string literal concatenated during array initialization*

This message indicates that a missing comma (',') could have made a quiet change in the program. Otherwise, ignore this message.

439 *maximum size of segment '%s' has been exceeded for '%S'*

The indicated symbol has grown in size to a point where it has caused the segment it is defined inside of to be exhausted.

440 *maximum data item size has been exceeded for '%S'*

A non-huge data item is larger than 64k bytes in size. This message only occurs during 16-bit compilation of C++ code.

441 *function attribute has been repeated*

A function attribute (like the ***__export*** attribute) has been repeated. Remove the extra attribute to correct the declaration.

442 *modifier has been repeated*

A modifier (like the ***far*** modifier) has been repeated. Remove the extra modifier to correct the declaration.

443 *illegal combination of memory model modifiers*

Memory model modifiers must be used individually because they cannot be combined meaningfully.

444 *argument name '%N' has already been used*

The indicated argument name has already been used in the same argument list. This is not allowed in the C++ language.

445 *function definition for '%S' must be declared with an explicit argument list*

A function cannot be defined with a typedef. The argument list must be explicit.

446 *user-defined conversion cannot convert to its own class or base class*

A user-defined conversion cannot be declared as a conversion either to its own class or to a base class of itself.

Example:
```
struct B {
};
struct D : private B {
    operator B();
};
```

447 *user-defined conversion cannot convert to void*

A user-defined conversion cannot be declared as a conversion to ***void.***

Example:
```
struct S {
    operator void();
};
```

448 *expecting identifier*

An identifier was expected during processing.

449 *symbol '%S' does not have a segment associated with it*

A pointer cannot be based on a member because it has no segment associated with it. A member describes a layout of storage that can occur in any segment.

450 *symbol '%S' must have integral or pointer type*

If a symbol is based on another symbol, it must be integral or a pointer type. An integral type indicates the segment value that will be used. A pointer type means that all accesses will be added to the pointer value to construct a full pointer.

451 *symbol '%S' cannot be accessed in all contexts*

The symbol that the pointer is based on is in another class so it cannot be accessed in all contexts that the based pointer can be accessed.

452 *cannot convert class expression to be copied*

A convert class expression could not be copied.

453 *conversion ambiguity: multiple copy constructors*

More than one constructor could be used to copy a class object.

454 *function template '%S' already has a definition*

The function template has already been defined with a function body. A function template cannot be defined twice even if the function body is identical.

Example:

```
template <class T>
    void f( T *p )
    {
    }
template <class T>
    void f( T *p )
    {
    }
```

455 *function templates cannot have default arguments*

A function template must not have default arguments because there are certain types of default arguments that do not force the function argument to be a specific type.

Example:

```
template <class T>
    void f2( T *p = 0 )
    {
    }
```

456 *'main' cannot be a function template*

This is a restriction in the C++ language because "main" cannot be overloaded. A function template provides the possibility of having more than one "main" function.

457 *'%S' was previously declared as a typedef*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:

```
typedef int T;
int T( int )        // error!
{
}

enum E { A, B, C };
void E()
{
    enum E x = A;   // use "enum E"
}

class C { };
void C()
{
    class C x;      // use "class C"
}
```

458 *'%S' was previously declared as a variable/function*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:
```
int T( int )
{
}
typedef int T;      // error!

void E()
{
}
enum E { A, B, C };

enum E x = A;       // use "enum E"

void C()
{
}
class C { };

class C x;      // use "class C"
```

459 *private base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

460 *protected base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

461 *maximum size of DGROUP has been exceeded for '%S' in segment '%s'*

The indicated symbol's size has caused the DGROUP contribution of this module to exceed 64k. Changing memory models or declaring some data as ***far*** data are two ways of fixing this problem.

462 *type of return value is not the enumeration type of function*

The return value does not have the proper enumeration type. Keep in mind that integral values are not automatically converted to enum types like the C language.

463 *linkage must be first in a declaration; probable cause: missing ';'*

This message usually indicates a missing semicolon (';'). The linkage specification must be the first part of a declaration if it is used.

464 *'main' cannot be a static function*

This is a restriction in the C++ language because "main" must have external linkage.

465 *'main' cannot be an inline function*

This is a restriction in the C++ language because "main" must have external linkage.

466 *'main' cannot be referenced*

This is a restriction in the C++ language to prevent implementations from having to work around multiple invocations of "main". This can occur if an implementation has to generate special code in "main" to construct all of the statically allocated classes.

467 *cannot call a non-volatile function for a volatile object*

A function that does not promise to not modify an object using ***volatile*** semantics cannot be called for a volatile object. A function can declare its intention to modify an object only through ***volatile*** semantics by using the ***volatile*** qualifier.

Example:

```
struct S {
    void fn();
};

void cfn( volatile S *p )
{
    p->fn();    // Error!
}
```

468 *cannot convert pointer to constant or volatile objects to pointer to void*

You cannot convert a pointer to constant or volatile objects to 'void*'.

Example:
```
extern const int* pci;
extern void *vp;

int k = ( pci == vp );
```

469 *cannot convert pointer to constant or non-volatile objects to pointer to volatile void*

You cannot convert a pointer to constant or non-volatile objects to 'volatile void*'.

Example:
```
extern const int* pci;
extern volatile void *vp;

int k = ( pci == vp );
```

470 *address of function is too large to be converted to pointer to void*

The address of a function can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the function pointer.

Example:
```
void __far foo();
void __near *v = &foo;
```

471 *address of data object is too large to be converted to pointer to void*

The address of an object can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the pointer.

Example:
```
int __far *ip;
void __near *v = ip;
```

472 *expression with side effect in sizeof discarded*

The indicated expression will be discarded; consequently, any side effects in that expression will not be executed.

Example:
```
int a = 14;
int b = sizeof( a++ );
```

In the example, the variable `a` will still have a value 14 after `b` has been initialized.

473 *function argument(s) do not match those in prototype*

The C++ language requires great precision in specifying arguments for a function. For instance, a pointer to `char` is considered different than a pointer to `unsigned char` regardless of whether `char` is an unsigned quantity. This message occurs when a non-overloaded function is invoked and one or more of the arguments cannot be converted. It also occurs when the number of arguments differs from the number specified in the prototype.

474 *conversion ambiguity: [expression] to [class object]*

The conversion of the expression to a class object is ambiguous.

475 *cannot assign right expression to class object*

The expression on the right cannot be assigned to the indicated class object.

476 *argument count is %d since there is an implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function with a ***this*** argument so it may have one more argument than expected.

477 *argument count is %d since there is no implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function without a ***this*** argument so it may have one less argument than expected.

478 *argument count is %d for a non-member function*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is not a member function but it could be declared as a ***friend*** function.

479 *conversion ambiguity: multiple copy constructors to copy array '%S'*

More than one constructor to copy the indicated array exists.

480 *variable/function has the same name as the class/enum '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the typedef name was declared in another file. This usually means that there are two unrelated uses of the same name.

481 *class/enum has the same name as the function/variable '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the function/variable name was declared in another file. This usually means that there are two unrelated uses of the same name. Furthermore, all references to the class or enum must be elaborated (i.e., use 'class C' instead of 'C') in order for subsequent references to compile properly.

482 *cannot create a default constructor*

A default constructor could not be created, because other constructors were declared for the class in question.

Example:

```
struct X {
    X(X&);
};
struct Y {
    X a[10];
};
Y yvar;
```

In the example, the variable "yvar" causes a default constructor for the class "Y" to be generated. The default constructor for "Y" attempts to call the default constructor for "X" in order to initialize the array "a" in class "Y". The default

constructor for "X" cannot be defined because another constructor has been declared.

483 *attempting to access default constructor for %T*

This informational message indicates that a default constructor was referenced but could not be generated.

484 *cannot align symbol '%S' to segment boundary*

The indicated symbol requires more than one segment of storage and the symbol's components cannot be aligned to the segment boundary.

485 *friend declaration does not specify a class or function*

A class or function must be declared as a friend.

Example:

```
struct T {
    // should be class or function declaration
    friend int;
};
```

486 *cannot take address of overloaded function*

This message indicates that an overloaded function's name was used in a context where a final type could not be found. Because a final type was not specified, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the expression.

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );
int k = ( p == &foo );          // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );

// introduce temporary
static int (*temp)( char ) = &foo;

// ok
int k = ( p == temp );
```

487 *cannot use address of overloaded function as a variable argument*

This message indicates that an overloaded function's name was used as a argument for a "..." style function. Because a final function type is not present, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the call.

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );
int k = ellip_fun( 14, &foo );      // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );

static int (*temp)( char ) = &foo;  // introduce
temporary

int k = ellip_fun( 14, temp );      // ok
```

488 *'%N' cannot be overloaded*

The indicated function cannot be overloaded. Functions that fall into this category include ***operator delete.***

489 *symbol '%S' has already been initialized*

The indicated symbol has already been initialized. It cannot be initialized twice even if the initialization value is identical.

490 *delete expression is a pointer to a function*

A pointer to a function cannot be allocated so it cannot be deleted.

491 *delete of a pointer to const data*

Since deleting a pointer may involve modification of data, it is not always safe to delete a pointer to const data.

Example:
```
struct S { };
void fn( S const *p, S const *q ) {
    delete p;
    delete [] q;
}
```

492 *delete expression is not a pointer to data*

A ***delete*** expression can only delete pointers. For example, trying to delete an ***int*** is not allowed in the C++ language.

Example:
```
void fn( int a )
{
    delete a;   // Error!
}
```

493 *template argument is not a constant expression*

The compiler has found an incorrect expression provided as the value for a constant value template argument. The only expressions allowed for scalar template arguments are integral constant expressions.

494 *template argument is not an external linkage symbol*

The compiler has found an incorrect expression provided as the value for a pointer value template argument. The only expressions allowed for pointer template arguments are addresses of symbols. Any symbols must have external linkage or must be static class members.

495 *conversion of const reference to volatile reference*

The constant value can be modified by assigning into the volatile reference. This would allow constant data to be modified quietly.

Example:

```
void fn( const int &rci )
{
    int volatile &r = rci;  // Error!
}
```

496 *conversion of volatile reference to const reference*

The volatile value can be read incorrectly by accessing the const reference. This would allow volatile data to be accessed without correct volatile semantics.

Example:

```
void fn( volatile int &rvi )
{
    int const &r = rvi; // Error!
}
```

497 *conversion of const or volatile reference to plain reference*

The constant value can be modified by assigning into the plain reference. This would allow constant data to be modified quietly. In the case of volatile data, any access to the plain reference will not respect the volatility of the data and thus would be incorrectly accessing the data.

Example:

```
void fn( const int &rci, volatile int &rvi )
{
    int &r1 = rci;  // Error!
    int &r2 = rvi;  // Error!
}
```

498 *syntax error before '%s'; probable cause: incorrectly spelled type name*

The identifier in the error message has not been declared as a type name in any scope at this point in the code. This may be the cause of the syntax error.

499 *object (or object pointer) required to access non-static member function*

A reference to a member function in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m();
    static void fn()
    {
    m();    // Error!
    }
};
```

500 *object (or object pointer) cannot be used to access function*

The indicated object (or object pointer) cannot be used to access function.

501 *object (or object pointer) cannot be used to access data*

The indicated object (or object pointer) cannot be used to access data.

502 *cannot access member function in enclosing class*

A member function in enclosing class cannot be accessed.

503 *cannot access data member in enclosing class*

A data member in enclosing class cannot be accessed.

504 *syntax error before type name '%s'*

The identifier in the error message has been declared as a type name at this point in the code. This may be the cause of the syntax error.

505 *implementation restriction: cannot generate thunk from '%S'*

This implementation restriction is due to the use of a shared code generator between Open Watcom compilers. The virtual ***this*** adjustment thunks are generated as functions linked into the virtual function table. The functions rely on knowing the correct number of arguments to pass on to the overriding virtual function but in the case of ellipsis (...) functions, the number of arguments cannot be known when the thunk function is being generated by the compiler. The target symbol is listed in a diagnostic message. The work around for this problem is to recode the source so that the virtual functions make use of the va_list type found in the stdarg header file.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    virtual void fun( char *, ... );
};
struct D : B {
    virtual void fun( char *, ... );
};
void B::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
    cout << va_arg( args, char ) << endl;
    ++f;
    }
    va_end( args );
}
void D::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
    cout << va_arg( args, int ) << endl;
    ++f;
    }
    va_end( args );
}
```

The previous example can be changed to the following code with corresponding changes to the contents of the virtual functions.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    void fun( char *f, ... )
    {
    va_list args;

    va_start( args, f );
    _fun( f, args );
    va_end( args );
    }
    virtual void _fun( char *, va_list );
};
~b
struct D : B {
    // this can be removed since using B::fun
    // will result in the same behaviour
    // since _fun is a virtual function
    void fun( char *f, ... )
    {
    va_list args;

    va_start( args, f );
    _fun( f, args );
    va_end( args );
    }
    virtual void _fun( char *, va_list );
};
~b
void B::_fun( char *f, va_list args )
{
    while( *f ) {
    cout << va_arg( args, char ) << endl;
    ++f;
    }
}
~b
void D::_fun( char *f, va_list args )
{
    while( *f ) {
    cout << va_arg( args, int ) << endl;
    ++f;
    }
}
```

```
~b
// no changes are required for users of the class
B x;
D y;

void dump( B *p )
{
    p->fun( "1234", 'a', 'b', 'c', 'd' );
    p->fun( "12", 'a', 'b' );
}

~b
void main()
{
    dump( &x );
    dump( &y );
}
```

506 *conversion of __based(void) pointer to virtual base class*

An __based(void) pointer to a class object cannot be converted to a pointer to virtual base class, since this conversion applies only to specific objects.

Example:

```
struct Base {};
struct Derived : virtual Base {};
Derived __based( void ) *p_derived;
Base __based( void ) *p_base = p_derived; // error
```

The conversion would be allowed if the base class were not virtual.

507 *class for target operand is not derived from class for source operand*

A member pointer conversion can only be performed safely when converting a base class member pointer to a derived class member pointer.

508 *conversion ambiguity: [pointer to class member] to [assignment object]*

The base class in the original member pointer is not a unique base class of the derived class.

509 *conversion of pointer to class member involves a private base class*

The member pointer conversion required access to a private base class. The access check did not succeed so the conversion is not allowed.

510 *conversion of pointer to class member involves a protected base class*

The member pointer conversion required access to a protected base class. The access check did not succeed so the conversion is not allowed.

511 *item is neither a non-static member function nor data member*

A member pointer can only be created for non-static member functions and non-static data members. Static members can have their address taken just like their file scope counterparts.

512 *function address cannot be converted to pointer to class member*

The indicated function address cannot be converted to pointer to class member.

513 *conversion ambiguity: [address of function] to [pointer to class member]*

The indicated conversion is ambiguous.

514 *addressed function is in a private base class*

The addressed function is in a private base class.

515 *addressed function is in a protected base class*

The addressed function is in a protected base class.

516 *class for object is not defined*

The left hand operand for the "." or ".*" operator must be of a class type that is completely defined.

Example:

```
class C;

int fun( C& x )
{
    return x.y;     // class C not defined
}
```

517 *left expression is not a class object*

The left hand operand for the ".*" operator must be of a class type since member pointers can only be used with classes.

518 *right expression is not a pointer to class member*

The right hand operand for the ".*" operator must be a member pointer type.

519 *cannot convert pointer to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

520 *conversion ambiguity: [pointer] to [class of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

521 *conversion of pointer to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

522 *conversion of pointer to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

523 *cannot convert object to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

524 *conversion ambiguity: [object] to [class object of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

525 *conversion of object to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

526 *conversion of object to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

527 *conversion of pointer to class member from a derived to a base class*

A member pointer can only be converted from a base class to a derived class. This is the opposite of the conversion rule for pointers.

528 *form is '#pragma inline_recursion en' where 'en' is 'on' or 'off'*

This ***pragma*** indicates whether inline expansion will occur for an inline function which is called (possibly indirectly) a subsequent time during an inline expansion. Either 'on' or 'off' must be specified.

529 *expression for number of array elements must be integral*

The expression for the number of elements in a ***new*** expression must be integral because it is used to calculate the size of the allocation (which is an integral quantity). The compiler will not automatically convert to an integer because of rounding and truncation issues with floating-point values.

530 *function accessed with '.*' or '->*' can only be called*

The result of the ".*" and "->*" operators can only be called because it is often specific to the instance used for the left hand operand.

531 *left operand must be a pointer, pointer to class member, or arithmetic*

The left operand must be a pointer, pointer to class member, or arithmetic.

532 *right operand must be a pointer, pointer to class member, or arithmetic*

The right operand must be a pointer, pointer to class member, or arithmetic.

533 *neither pointer to class member can be converted to the other*

The two member pointers being compared are from two unrelated classes. They cannot be compared since their members can never be related.

534 *left operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the left operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( p == mp )
    p[0] = 1;
}
```

535 *right operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the right operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( mp == p )
    p[0] = 1;
}
```

536 *cannot use '.*' nor '->*' with pointer to class member with zero value*

The compiler has detected a NULL pointer use with a member pointer dereference.

537 *operand is not a valid pointer to class member*

The operand cannot be converted to a valid pointer to class member.

Example:

```
struct S;
int S::* fn()
{
    int a;
    return a;
}
```

538 *destructor can be invoked only with '.' or '->'*

This is a restriction in the C++ language. An explicit invocation of a destructor is not recommended for objects that have their destructor called automatically.

539 *class of destructor must be class of object being destructed*

Destructors can only be called for the exact static type of the object being destroyed.

540 *destructor is not properly qualified*

An explicit destructor invocation can only be qualified with its own class.

541 *pointers to class members reference different object types*

Conversion of member pointers can only occur if the object types are identical. This is necessary to ensure type safety.

542 *operand must be pointer to class or struct*

The left hand operand of a '->*' operator must be a pointer to a class. This is a restriction in the C++ language.

543 *expression must have void type*

If one operand of the ':' operator has ***void*** type, then the other operand must also have ***void*** type.

544 *expression types do not match for ':' operator*

The compiler could not bring both operands to a common type. This is necessary because the result of the conditional operator must be a unique type.

545 *cannot create an undefined type with 'operator new'*

A ***new*** expression cannot allocate an undefined type because it must know how large an allocation is required and it must also know whether there are any constructors to execute.

546 *delete of a pointer to an undefined type*

A ***delete*** expression cannot safely deallocate an undefined type because it must know whether there are any destructors to execute. In spite of this, the ISO/ANSI C++ Working Paper requires that an implementation support this usage.

Example:

```
struct U;

void foo( U *p, U *q ) {
    delete p;
    delete [] q;
}
```

547 *cannot access '%S' through a private base class*

The indicated symbol cannot be accessed because it requires access to a private base class.

548 *cannot access '%S' through a protected base class*

The indicated symbol cannot be accessed because it requires access to a protected base class.

549 *'sizeof' operand contains compiler generated information*

The type used in the 'sizeof' operand contains compiler generated information. Clearing a struct with a call to memset() would invalidate all of this information.

550 *cannot convert ':' operands to a common reference type*

The two reference types cannot be converted to a common reference type. This can happen when the types are not related through base class inheritance.

551 *conversion ambiguity: [reference to object] to [type of opposite ':' operand]*

One of the reference types is an ambiguous base class of the other. This prevents the compiler from converting the operand to a unique common type.

552 *conversion of reference to ':' object involves a private base class*

The conversion of the reference operands requires a conversion through a private base class.

553 *conversion of reference to ':' object involves a protected base class*

The conversion of the reference operands requires a conversion through a protected base class.

554 *expression must have type arithmetic, pointer, or pointer to class member*

This message means that the type cannot be converted to any of these types, also. All of the mentioned types can be compared against zero ('0') to produce a true or false value.

555 *expression for 'while' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

556 *testing expression for 'for' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

557 *message number '%d' is invalid*

The message number used in the #pragma does not match the message number for any warning message. This message can also indicate that a number or '*' (meaning all warnings) was not found when it was expected.

558 *warning level must be an integer in range 0 to 9*

The new warning level that can be used for the warning can be in the range 0 to 9. The level 0 means that the warning will be treated as an error (compilation will not succeed). Levels 1 up to 9 are used to classify warnings. The -w option sets an upper limit on the level for warnings. By setting the level above the command line limit, you effectively ignore all cases where the warning shows up.

559 *function '%S' cannot be defined because it is generated by the compiler*

The indicated function cannot be defined because it is generated by the compiler. The compiler will automatically generate default constructors, copy constructors, assignment operators, and destructors according to the rules of the C++ language. This message indicates that you did not declare the function in the class definition.

560 *neither environment variable nor file found for '@' name*

The indirection operator for the command line will first check for an environment variable of the name and use the contents for the command line. If an environment variable is not found, a check for a file with the same name will occur.

561 *more than 5 indirections during command line processing*

The Open Watcom C++ compiler only allows a fixed number nested indirections using files or environment variables, to prevent runaway chains of indirections.

562 *cannot take address of non-static member function*

The only way to create a value that described the non-static member function is to use a member pointer.

563 *cannot generate default '%S' because class contains either a constant or a reference member*

An assignment operator cannot be generated because the class contains members that cannot be assigned into.

564 *cannot convert pointer to non-constant or volatile objects to pointer to const void*

A pointer to non-constant or volatile objects cannot be converted to 'const void*'.

565 *cannot convert pointer to non-constant or non-volatile objects to pointer to const volatile void*

A pointer to non-constant or non-volatile objects cannot be converted to 'const volatile void*'.

566 *cannot initialize pointer to non-volatile with a pointer to volatile*

A pointer to a non-volatile type cannot be initialized with a pointer to a volatile type because this would allow volatile data to be modified without volatile semantics via the non-volatile pointer to it.

567 *cannot pass a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be passed in this context.

568 *cannot return a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be returned.

569 *left expression is not a pointer to a volatile object*

One cannot assign a pointer to a volatile type to a pointer to a non-volatile type. This would allow a volatile object to be modified via the non-volatile pointer. Use a cast if this is absolutely necessary.

570 *virtual function override for '%S' is ambiguous*

This message indicates that there are at least two overrides for the function in the base class. The compiler cannot arbitrarily choose one so it is up to the programmer to make sure there is an unambiguous choice. Two of the overriding functions are listed as informational messages.

571 *initialization priority must be number 0-255, 'library', or 'program'*

An incorrect module initialization priority has been provided. Check the User's Guide for the correct format of the priority directive.

572 *previous case label defined %L*

This informational message indicates where a preceding ***case*** label is defined.

573 *previous default label defined %L*

This informational message indicates where a preceding ***default*** label is defined.

574 *label defined %L*

This informational message indicates where a label is defined.

575 *label referenced %L*

This informational message indicates where a label is referenced.

576 *object thrown has type: %T*

This informational message indicates the type of the object being thrown.

577 *object thrown has an ambiguous base class %T*

It is illegal to throw an object with a base class to which a conversion would be ambiguous.

Example:
```
struct ambiguous{ };
struct base1 : public ambiguous { };
struct base2 : public ambiguous { };
struct derived : public base1, public base2 { };

foo( derived &object )
{
    throw object;
}
```

The ***throw*** will cause an error to be displayed because an object of type "derived" cannot be converted to an object of type "ambiguous".

578 *form is '#pragma inline_depth level' where 'level' is 0 to 255*

This ***pragma*** sets the number of times inline expansion will occur for an inline function which contains calls to inline functions. The level must be a number from zero to 255. When the level is zero, no inline expansion occurs.

579 *pointer or reference truncated by cast*

The cast expression causes a conversion of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., casting a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. Because this message indicates that some information is being lost, check the code carefully.

580 *cannot find a constructor for given initializer argument list*

The initializer list provided for the ***new*** expression does not uniquely identify a single constructor.

581 *variable '%N' can only be based on a string in this context*

All of the based modifiers can only be applied to pointer types. The only based modifier that can be applied to non-pointer types is the '__based(__segname("WATCOM"))' style.

582 *memory model modifiers are not allowed for class members*

Class members describe the arrangement and interpretation of memory and, as such, assume the memory model of the address used to access the member.

583 *redefinition of the typedef name '%S' ignored*

The compiler has detected that a slightly different type has been assigned to a typedef name. The type is functionally equivalent but typedef redefinitions should be precisely identical.

584 *constructor for variable '%S' cannot be bypassed*

The variable may not be constructed when code is executing at the position the message indicated. The C++ language places these restrictions to prevent the use of unconstructed variables.

585 *syntax error; missing start of function body after constructor initializer*

Member initializers can only be used in a constructor's definition.

Example:

```
struct S {
    int a;
    S( int x = 1 ) : a(x)
    {
    }
};
```

586 *conversion ambiguity: [expression] to [type of default argument]*

A conversion to an ambiguous base class was detected in the default argument expression.

587 *conversion of expression for default argument is impossible*

A conversion to a unrelated class was detected in the default argument expression.

588 *syntax error before template name '%s'*

The identifier in the error message has been declared as a template name at this point in the code. This may be the cause of the syntax error.

589 *private base class accessed to convert default argument*

A conversion to a private base class was detected in the default argument expression.

590 *protected base class accessed to convert default argument*

A conversion to a protected base class was detected in the default argument expression.

591 *operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

592 *left operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

593 *right operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

594 *construct resolved as a declaration/type*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of a type. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

595 *construct resolved as an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of an expression (a function-like cast). The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

596 *construct cannot be resolved*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct could not be resolved by the compiler. Please report this to the Open

Watcom developement team so that the problem can be analysed. See http://www.openwatcom.org/.

597 *encountered another ambiguous construct during disambiguation*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that another ambiguous construct was found inside an ambiguous construct. The compiler will correctly disambiguate the construct. The programmer is advised to change code that exhibits this warning because this is definitely uncharted territory in the C++ language.

598 *ellipsis (...) argument contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern int bar( S, ... );

static int test = bar( sv, 14, 64 );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

599 *cannot convert argument for ellipsis (...) argument*

This argument cannot be used as an ellipsis (...) argument to a function.

600 *conversion ambiguity: [argument] to [ellipsis (...) argument]*

A conversion ambiguity was detected while converting an argument to an ellipsis (...) argument.

601 *converted function type has different #pragma from original function type*

Since a #pragma can affect calling conventions, one must be very careful performing casts involving different calling conventions.

602 *class value used as return value or argument in converted function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

603 *class value used as return value or argument in original function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

604 *must look ahead to determine whether construct is a declaration/type or an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been used. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous.

605 *assembler: '%s'*

An error has been detected by the #pragma inline assembler.

606 *default argument expression cannot reference 'this'*

The order of evaluation for function arguments is unspecified in the C++ language document. Thus, a default argument must be able to be evaluated before the 'this' argument (or any other argument) is evaluated.

607 *#pragma aux must reference a "C" linkage function '%S'*

The method of assigning pragma information via the #pragma syntax is provided for compatibility with Open Watcom C. Because C only allows one function per name, this was adequate for the C language. Since C++ allows functions to be overloaded, a new method of referencing pragmas has been introduced.

Example:

```
#pragma aux this_in_SI parm caller [si] [ax];

struct S {
    void __pragma("this_in_SI") foo( int );
    void __pragma("this_in_SI") foo( char );
};
```

608 *assignment is ambiguous for operands used*

An ambiguity was detected while attempting to convert the right operand to the type of the left operand.

Example:

```
struct S1 {
    int a;
};

struct S2 : S1 {
    int b;
};

struct S3 : S2, S1 {
    int c;
};

S1* fn( S3 *p )
{
    return p;
}
```

In the example, ***class*** `S1` occurs ambiguously for an object or pointer to an object of type `S3.` A pointer to an `S3` object cannot be converted to a pointer to an `S1` object.

609 *pragma name '%s' is not defined*

Pragmas are defined with the #pragma aux syntax. See the User's Guide for the details of defining a pragma name. If the pragma has been defined then check the spelling between the definition and the reference of the pragma name.

610 *'%S' could not be generated by the compiler*

An error occurred while the compiler tried to generate the specified function. The error prevented the compiler from generating the function properly so the compilation cannot continue.

611 *'catch' does not immediately follow a 'try' or 'catch'*

The catch handler syntax must be used in conjunction with a try block.

Example:

```
void f()
{
    try {
    // code that may throw an exception
    } catch( int x ) {
    // handle 'int' exceptions
    } catch( ... ) {
    // handle all other exceptions
    }
}
```

612 *preceding catch specified '...'*

Since an ellipsis "..." catch handler will handle any type of exception, no further catch handlers can exist afterwards because they will never execute. Reorder the catch handlers so that the "..." catch handler is the last handler.

613 *argument to extern "C" function contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern "C" int bar( S );

static int test = bar( sv );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

614 *previous try block defined %L*

This informational message indicates where a preceding ***try*** block is defined.

615 *previous catch block defined %L*

This informational message indicates where a preceding ***catch*** block is defined.

616 *catch handler can never be invoked*

Because the handlers for a ***try*** block are tried in order of appearance, the type specified in a preceding ***catch*** can ensure that the current handler will never be invoked. This occurs when a base class (or reference) precedes a derived class (or reference); when a pointer to a base class (or reference to the pointer) precedes a pointer to a derived class (or reference to the pointer); or, when "void*" or "void*&" precedes a pointer or a reference to the pointer.

Example:

```
struct BASE {};
struct DERIVED : public BASE {};

foo()
{
    try {
    // code for try
    } catch( BASE b ) {     // [1]
    // code
    } catch( DERIVED ) {    // warning: [1]
    // code
    } catch( BASE* pb ) {   // [2]
    // code
    } catch( DERIVED* pd ) {// warning: [2]
    // code
    } catch( void* pv ) {   // [3]
    // code
    } catch( int* pi ) {    // warning: [3]
    // code
    } catch( BASE& br ) {   // warning: [1]
    // code
    } catch( float*& pfr ) {// warning: [3]
    // code
    }
}
```

Each erroneous catch specification indicates the preceding catch block which caused the error.

617 *cannot overload extern "C" functions (the other function is '%S')*

The C++ language only allows you to overload functions that are strictly C++ functions. The compiler will automatically generate the correct code to distinguish each particular function based on its argument types. The extern "C" linkage mechanism only allows you to define one "C" function of a particular name because the C language does not support function overloading.

618 *function will be overload ambiguous with ’%S’ using default arguments*

The declaration declares a function that is indistinguishable from another function of the same name with default arguments.

Example:
```
void fn( int, int = 1 );
void fn( int );
```

Calling the function ’fn’ with one argument is ambiguous because it could match either the first ’fn’ with a default argument applied or the second ’fn’ without any default arguments.

619 *linkage specification is different than previous declaration ’%S’*

The linkage specification affects the binding of names throughout a program. It is important to maintain consistency because subtle problems could arise when the incorrect function is called. Usually this error prevents an unresolved symbol error during linking because the name of a declaration is affected by its linkage specification.

Example:
```
extern "C" void fn( void );
void fn( void )
{
}
```

620 *not enough segment registers available to generate ’%s’*

Through a combination of options, the number of available segment registers is too small. This can occur when too many segment registers are pegged. This can be fixed by changing the command line options to only peg the segment registers that must absolutely be pegged.

621 *pure virtual destructors must have a definition*

This is an anomaly for pure virtual functions. A destructor is the only special function that is inherited and allowed to be virtual. A derived class must be able to call the base class destructor so a pure virtual destructor must be defined in a C++ program.

622 *jump into try block*

Jumps cannot enter ***try*** blocks.

Example:
```
foo( int a )
{
    if(a) goto tr_lab;

    try {
tr_lab:
    throw 1234;
    } catch( int ) {
    if(a) goto tr_lab;
    }

    if(a) goto tr_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

623 *jump into catch handler*

Jumps cannot enter ***catch*** handlers.

Example:
```
foo( int a )
{
    if(a)goto ca_lab;

    try {
    if(a)goto ca_lab;
    } catch( int ) {
ca_lab:
    }

    if(a)goto ca_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

624 *catch block does not immediately follow try block*

At least one ***catch*** handler must immediately follow the "}" of a ***try*** block.

Example:
```
extern void goop();
void foo()
{
    try {
    goop();
    }       // a catch block should follow!
}
```

In the example, there were no catch blocks after the ***try*** block.

625 *exceptions must be enabled to use feature (use 'xs' option)*

Exceptions are enabled by specifying the 'xs' option when the compiler is invoked. The error message indicates that a feature such as ***try, catch, throw,*** or function exception specification has been used without enabling exceptions.

626 *I/O error reading '%s': %s"*

When attempting to read data from a source or header file, the indicated system error occurred. Likely there is a hardware problem, or the file system has become corrupt.

627 *text following pre-processor directive*

A ***#else*** or ***#endif*** directive was found which had tokens following it rather than an end of line. Some UNIX style preprocessors allowed this, but it is not legal under standard C or C++. Make the tokens into a comment.

628 *expression is not meaningful*

This message indicates that the indicated expression is not meaningful. An expression is meaningful when a function is invoked, when an assignment or initialization is performed, or when the expression is casted to void.

Example:
```
void foo( int i, int j )
{
    i + j;  // not meaningful
}
```

629 *expression has no side effect*

The indicated expression does not cause a side effect. A side effect is caused by invoking a function, by an assignment or an initialization, or by reading a ***volatile*** variable.

Example:
```
int k;
void foo( int i, int j )
{
    i + j,  // no side effect (note comma)
    k = 3;
}
```

630 *source conversion type is '%T'*

This informational message indicates the type of the source operand, for the preceding conversion diagnostic.

631 *target conversion type is '%T'*

This informational message indicates the target type of the conversion, for the preceding conversion diagnostic.

632 *redeclaration of '%S' has different attributes*

A function cannot be made ***virtual*** or pure ***virtual*** in a subsequent declaration. All properties of a function should be described in the first declaration of a function. This is especially important for member functions because the properties of a class are affected by its member functions.

Example:
```
struct S {
    void fun();
};

virtual void S::fun()
{
}
```

633 *template class instantiation for '%T' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a class template. The final type of the template class is shown as well as the location in the source where the instantiation was initiated.

634 *template function instantiation for '%S' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a function template. The final type of the template function is shown as well as the location in the source where the instantiation was initiated.

635 *template class member instantiation was %L*

This informational message indicates that the error or warning was detected during the instantiation of a member of a class template. The location in the source where the instantiation was initiated is shown.

636 *function template binding for '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where arguments are analysed in order to infer what types should be used in a function template instantiation. The function template in question is shown along with the location in the source code that initiated the binding process.

637 *function template binding of '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where a function prototype is analysed in order to see if the prototype matches any function template of the same name. The function template in question is shown along with the location in the source code that initiated the binding process.

638 *'%s' defined %L*

This informational message indicates where the class in question was defined. The message is displayed following an error or warning diagnostic for the class in question.

Example:

```
class S;
int foo( S*p )
{
    return p->x;
}
```

The variable `p` is a pointer to an undefined class and so will cause an error to be generated. Following the error, the informational message indicates the line at which the class S was declared.

639 *form is '#pragma template_depth level' where 'level' is a non-zero number*

This ***pragma*** sets the number of times templates will be instantiated for nested instantiations. The depth check prevents infinite compile times for incorrect programs.

640 *possible non-terminating template instantiation (use "#pragma template_depth %d" to increase depth)*

This message indicates that a large number of expansions were required to complete a template class or template function instantiation. This may indicate that there is an erroneous use of a template. If the program will complete given more depth, try using the suggested #pragma in the error message to increase the depth. The number provided is double the previous value.

641 *cannot inherit a partially defined base class '%T'*

This message indicates that the base class was in the midst of being defined when it was inherited. The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:

```
struct Partial {
    struct Nested : Partial {
    int n;
    };
};
```

642 *ambiguous function: %F defined %L*

This informational message shows the functions that were detected to be ambiguous.

Example:

```
int amb( char );        // will be ambiguous
int amb( unsigned char );   // will be ambiguous
int amb( char, char );
int k = amb( 14 );
```

The constant value 14 has an ***int*** type and so the attempt to invoke the function `amb` is ambiguous. The first two functions are ambiguous (and will be displayed); the third is not considered (nor displayed) since it is declared to have a different number of arguments.

643 *cannot convert argument %d defined %L*

This informational message indicates the first argument which could not be converted to the corresponding type for the declared function. It is displayed when there is exactly one function declared with the indicated name.

644 *'this' cannot be converted*

This informational message indicates the ***this*** pointer for the function which could not be converted to the type of the ***this*** pointer for the declared function. It is displayed when there is exactly one function declared with the indicated name.

645 *rejected function: %F defined %L*

This informational message shows the overloaded functions which were rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function with the indicated name.

646 *'%T' operator can be used*

Following a diagnosis of operator ambiguity, this information message indicates that the operator can be applied with operands of the type indicated in the message.

Example:

```
struct S {
    S( int );
    operator int();
    S operator+( int );
};
S s(15);
int k = s + 123;    // "+" is ambiguous
```

In the example, the "+" operation is ambiguous because it can implemented as by the addition of two integers (with `S::operator int` applied to the second operand) or by a call to `S::operator+`. This informational message indicates that the first is possible.

647 *cannot #undef '%s'*

The predefined macros `__cplusplus, __DATE__, __FILE__, __LINE__, __STDC__, __TIME__, __FUNCTION__` and `__func__` cannot be undefined using the ***#undef*** directive.

Example:

```
#undef __cplusplus
#undef __DATE__
#undef __FILE__
#undef __LINE__
#undef __STDC__
#undef __TIME__
#undef __FUNCTION__
#undef __func__
```

All of the preceding directives are not permitted.

648 *cannot #define '%s'*

The predefined macros `__cplusplus`, `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, and `__TIME__` cannot be defined using the ***#define*** directive.

Example:

```
#define __cplusplus     1
#define __DATE__        2
#define __FILE__        3
#define __LINE__        4
#define __STDC__        5
#define __TIME__        6
```

All of the preceding directives are not permitted.

649 *template function '%F' defined %L*

This informational message indicates where the function template in question was defined. The message is displayed following an error or warning diagnostic for the function template in question.

Example:

```
template <class T>
    void foo( T, T * )
    {
    }

void bar()
{
    foo(1);     // could not instantiate
}
```

The function template for `foo` cannot be instantiated for a single argument causing an error to be generated. Following the error, the informational message indicates the line at which `foo` was declared.

650 *ambiguous function template: %F defined %L*

This informational message shows the function templates that were detected to be ambiguous for the arguments at the call point.

651 *cannot instantiate %S*

This message indicates that the function template could not be instantiated for the arguments supplied. It is displayed when there is exactly one function template declared with the indicated name.

652 *rejected function template: %F defined %L*

This informational message shows the overloaded function template which was rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function or function template with the indicated name.

653 *operand cannot be a function*

The indicated operation cannot be applied to a function.

Example:

```
int Fun();
int j = ++Fun;  // illegal
```

In the example, the attempt to increment a function is illegal.

654 *left operand cannot be a function*

The indicated operation cannot be applied to the left operand which is a function.

Example:

```
extern int Fun();
void foo()
{
    Fun = 0;    // illegal
}
```

In the example, the attempt to assign zero to a function is illegal.

655 *right operand cannot be a function*

The indicated operation cannot be applied to the right operand which is a function.

Example:
```
extern int Fun();
void foo()
{
    void* p = 3[Fun];   // illegal
}
```

In the example, the attempt to subscript a function is illegal.

656 *define this function inside its class definition (may improve code quality)*

The Open Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations.

Example:
```
struct S {
    ~S();
};

S::~S() {
}
```

657 *define this function inside its class definition (could have improved code quality)*

The Open Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations. This particular warning indicates that the compiler has already found an opportunity in previous code but it found out too late that the constructor or destructor had an empty function body.

Example:

```
struct S {
    ~S();
};
struct T : S {
    ~T() {}
};

S::~S() {
}
```

658 *cannot convert address of overloaded function '%S'*

This information message indicates that an address of an overloaded function cannot be converted to the indicated type.

Example:

```
int ovload( char );
int ovload( float );
int routine( int (*)( int );
int k = routine( ovload );
```

The first argument for the function `routine` cannot be converted, resulting in the informational message.

659 *expression cannot have void type*

The indicated expression cannot have a ***void*** type.

Example:

```
main( int argc, char* argv )
{
    if( (void)argc ) {
    return 5;
    } else {
    return 9;
    }
}
```

Conditional expressions, such as the one illustrated in the ***if*** statement cannot have a ***void*** type.

660 *cannot reference a bit field*

The smallest addressable unit is a byte. You cannot reference a bit field.

Example:

```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
int& ref = var.bitfield;    // illegal
```

661 *cannot assign to object having an undefined class*

An assignment cannot be be made to an object whose class has not been defined.

Example:

```
class X;        // declared, but not defined
extern X& foo();    // returns reference (ok)
extern X obj;
void goop()
{
    obj = foo();    // error
}
```

662 *cannot create member pointer to constructor*

A member pointer value cannot reference a constructor.

Example:

```
class C {
    C();
};
int foo()
{
    return 0 == &C::C;
}
```

663 *cannot create member pointer to destructor*

A member pointer value cannot reference a destructor.

Example:

```
class C {
    ~C();
};
int foo()
{
    return 0 == &C::~C;
}
```

664 *attempt to initialize a non-constant reference with a temporary object*

A temporary value cannot be converted to a non-constant reference type.

Example:

```
struct C {
    C( C& );
    C( int );
};

C & c1 = 1;
C c2 = 2;
```

The initializations of `c1` and `c2` are erroneous, since temporaries are being bound to non-const references. In the case of `c1,` an implicit constructor call is required to convert the integer to the correct object type. This results in a temporary object being created to initialize the reference. Subsequent code can modify this temporary's state. The initialization of `c2,` is erroneous for a similar reason. In this case, the temporary is being bound to the non-const reference argument of the copy constructor.

665 *temporary object used to initialize a non-constant reference*

Ordinarily, a temporary value cannot be bound to a non-constant reference. There is enough legacy code present that the Open Watcom C++ compiler issues a warning in cases that should be errors. This may change in the future so it is advisable to correct the code as soon as possible.

666 *assuming unary 'operator &' not overloaded for type '%T'*

An explicit address operator can be applied to a reference to an undefined class. The Open Watcom C++ compiler will assume that the address is required but it does not know whether this was the programmer's intention because the class definition has not been seen.

Example:
```
struct S;

S * fn( S &y ) {
    // assuming no operator '&' defined
    return &y;
}
```

667 *'va_start' macro will not work without an argument before '...'*

The warning indicates that it is impossible to access the arguments passed to the function without declaring an argument before the "..." argument. The "..." style of argument list (without any other arguments) is only useful as a prototype or if the function is designed to ignore all of its arguments.

Example:
```
void fn( ... )
{
}
```

668 *'va_start' macro will not work with a reference argument before '...'*

The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the reference argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

void fn( int &r, ... )
{
    va_list args;

    // address of 'r' is address of
    // object 'r' references so
    // 'va_start' will not work properly
    va_start( args, r );
    va_end( args );
}
```

669 *'va_start' macro will not work with a class argument before '...'*

This warning is specific to C++ compilers that quietly convert class arguments to class reference arguments. The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the class argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

struct S {
    S();
};

void fn( S c, ... )
{
    va_list args;

    // Open Watcom C++ passes a pointer to
    // the temporary created for passing
    // 'c' rather than pushing 'c' on the
    // stack so 'va_start' will not work
    // properly
    va_start( args, c );
    va_end( args );
}
```

670 *function modifier conflicts with previous declaration '%S'*

The symbol declaration conflicts with a previous declaration with regard to function modifiers. Either the previous declaration did not have a function modifier or it had a different one.

Example:
```
#pragma aux never_returns aborts;

void fn( int, int );
void __pragma("never_returns") fn( int, int );
```

671 *function modifier cannot be used on a variable*

The symbol declaration has a function modifier being applied to a variable or non-function. The cause of this may be a declaration with a missing function argument list.

Example:
```
int (* __pascal ok)();
int (* __pascal not_ok);
```

672 *'%T' contains the following pure virtual functions*

This informational message indicates that the class contains pure virtual function declarations. The class is definitely abstract as a result and cannot be used to declare variables. The pure virtual functions declared in the class are displayed immediately following this message.

Example:
```
struct A {
    void virtual fn( int ) = 0;
};

A x;
```

673 *'%T' has no implementation for the following pure virtual functions*

This informational message indicates that the class is derived from an abstract class but the class did not override enough virtual function declarations. The pure virtual functions declared in the class are displayed immediately following this message.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

674 *pure virtual function '%F' defined %L*

This informational message indicates that the pure virtual function has not been overridden. This means that the class is abstract.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

675 *restriction: standard calling convention required for '%S'*

The indicated function may be called by the C++ run-time system using the standard calling convention. The calling convention specified for the function is incompatible with the standard convention. This message may result when `__pascal` is specified for a default constructor, a copy constructor, or a destructor. It may also result when `parm reverse` is specified in a ***#pragma*** for the function.

676 *number of arguments in function call is incorrect*

The number of arguments in the function call does not match the number declared for the function type.

Example:
```
extern int (*pfn)( int, int );
int k = pfn( 1, 2, 3 );
```

In the example, the function pointer was declared to have two arguments. Three arguments were used in the call.

677 *function has type '%T'*

This informational message indicates the type of the function being called.

678 *invalid octal constant*

The constant started with a '0' digit which makes it look like an octal constant but the constant contained the digits '8' and '9'. The problem could be an incorrect octal constant or a missing '.' for a floating constant.

Example:
```
int i = 0123456789; // invalid octal constant
double d = 0123456789;  // missing '.'?
```

679 *class template definition started %L*

This informational message indicates where the class template definition started so that any problems with missing braces can be fixed quickly and easily.

Example:
```
template <class T>
    struct S {
    void f1() {
    // error missing '}'
    };

template <class T>
    struct X {
    void f2() {
    }
    };
```

680 *constructor initializer started %L*

This informational message indicates where the constructor initializer started so that any problems with missing parenthesis can be fixed quickly and easily.

Example:
```
struct S {
    S( int x ) : a(x), b(x // missing parenthesis
    {
    }
};
```

681 *zero size array must be the last data member*

The language extension that allows a zero size array to be declared in a class definition requires that the array be the last data member in the class.

Example:
```
struct S {
    char a[];
    int b;
};
```

682 *cannot inherit a class that contains a zero size array*

The language extension that allows a zero size array to be declared in a class definition disallows the use of the class as a base class. This prevents the programmer from corrupting storage in derived classes through the use of the zero size array.

Example:
```
struct B {
    int b;
    char a[];
};
struct D : B {
    int d;
};
```

683 *zero size array '%S' cannot be used in a class with base classes*

The language extension that allows a zero size array to be declared in a class definition requires that the class not have any base classes. This is required because the C++ compiler must be free to organize base classes in any manner for optimization purposes.

Example:
```
struct B {
    int b;
};
struct D : B {
    int d;
    char a[];
};
```

684 *cannot catch abstract class object*

C++ does not allow abstract classes to be instantiated and so an abstract class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an abstract class.

Example:
```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
    throw Derived();
    } catch( Abstract abstract ) {   // object
    xyz = 1;
    }
}
```

The catch clause in the preceding example would be diagnosed as improper, since an abstract class is specified. The example could be coded as follows.

Example:

```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
    throw Derived();
    } catch( Abstract & abstract ) {  // reference
    xyz = 1;
    }
}
```

685 *non-static member function '%S' cannot be specified*

The indicated non-static member function cannot be used in this context. For example, such a function cannot be used as the second or third operand of the conditional operator.

Example:

```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return (i ? foo : bar)();
}
```

Neither `foo` nor `bar` can be specified as shown in the example. The example can be properly coded as follows:

Example:
```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return i ? foo() : bar();
}
```

686 *attempt to convert pointer or reference from a base to a derived class*

A pointer or reference to a base class cannot be converted to a pointer or reference, respectively, of a derived class, unless there is an explicit cast. The `return` statements in the following example will be diagnosed.

Example:
```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return &b; }
Derived& ReturnRef() { return b; }
```

The following program would be acceptable:

Example:
```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return (Derived*)&b; }
Derived& ReturnRef() { return (Derived&)b; }
```

687 *expression for 'while' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ; )` without causing warnings.

688 *testing expression for 'for' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ; )` without causing warnings.

689 *conditional expression is always true (non-zero)*

The indicated expression is a non-zero constant and so will always be true.

690 *conditional expression is always false (zero)*

The indicated expression is a zero constant and so will always be false.

691 *expecting a member of '%T' to be defined in this context*

A class template member definition must define a member of the associated class template. The complexity of the C++ declaration syntax can make this error hard to identify visually.

Example:

```
template <class T>
    struct S {
    typedef int X;
    static X fn( int );
    static X qq;
    };

template <class T>
    S<T>::X fn( int ) {// should be 'S<T>::fn'

    return fn( 2 );
    }

template <class T>
    S<T>::X qq = 1; // should be 'S<T>::q'

S<int> x;
```

692 *cannot throw an abstract class*

An abstract class cannot be thrown since copies of that object may have to be made (which is impossible);

Example:
```
struct abstract_class {
    abstract_class( int );
    virtual int foo() = 0;
};

void goop()
{
    throw abstract_class( 17 );
}
```

The ***throw*** expression is illegal since it specifies an abstract class.

693 *cannot create pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for write access.

694 *error occurred while writing pre-compiled header file*

The compiler has detected a problem while trying to write some data to the pre-compiled header file.

695 *error occurred while reading pre-compiled header file*

The compiler has detected a problem while trying to read some data from the pre-compiled header file.

696 *pre-compiled header file being recreated*

The existing pre-compiled header file may either be corrupted or is a version that the compiler cannot use due to updates to the compiler. A new version of the pre-compiled header file will be created.

697 *pre-compiled header file being recreated (different compile options)*

The compiler has detected that the command line options have changed enough so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

698 *pre-compiled header file being recreated (different #include file)*

The compiler has detected that the first ***#include*** file name is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

699 *pre-compiled header file being recreated (different current directory)*

The compiler has detected that the working directory is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

700 *pre-compiled header file being recreated (different INCLUDE path)*

The compiler has detected that the INCLUDE path is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

701 *pre-compiled header file being recreated ('%s' has been modified)*

The compiler has detected that an include file has changed so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

702 *pre-compiled header file being recreated (macro '%s' is different)*

The compiler has detected that a macro definition is different so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

703 *pre-compiled header file being recreated (macro '%s' is not defined)*

The compiler has detected that a macro has not been defined so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

704 *command line specifies smart windows callbacks and DS not equal to SS*

An illegal combination of switches has been detected. The windows smart callbacks option cannot be combined with either of the build DLL or DS not equal to SS options.

705 *class '%N' cannot be used with #pragma dump_object_model*

The indicated name has not yet been declared or has been declared but not yet been defined as a class. Consequently, the object model cannot be dumped.

706 *repeated modifier is '%s'*

This informational message indicates what modifier was repeated in the declaration.

Example:
```
typedef int __far FARINT;
FARINT __far *p;    // repeated __far modifier
```

707 *semicolon (';') may be missing after class/enum definition*

This informational message indicates that a missing semicolon (';') may be the cause of the error.

Example:
```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

708 *cannot return a type of unknown size*

A value of an unknown type cannot be returned.

Example:
```
class S;
S foo();

int goo()
{
    foo();
}
```

In the example, foo cannot be invoked because the class which it returns has not been defined.

709 *cannot initialize array member '%S'*

An array class member cannot be specified as a constructor initializer.

Example:
```
class S {
public:
    int arr[3];
    S();
};
S::S() : arr( 1, 2, 3 ) {}
```

In the example, `arr` cannot be specified as a constructor initializer. Instead, the array may be initialized within the body of the constructor.

Example:
```
class S {
public:
    int arr[3];
    S();
};
S::S()
{
    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;
}
```

710 *file '%s' will #include itself forever*

The compiler has detected that the file in the message has been ***#include*** from within itself without protecting against infinite inclusion. This can happen if ***#ifndef*** and ***#define*** header file protection has not been used properly.

Example:
```
#include __FILE__
```

711 *'mutable' may only be used for non-static class members*

A declaration in file scope or block scope cannot have a storage class of ***mutable.***

Example:
```
mutable int a;
```

712 *'mutable' member cannot also be const*

A ***mutable*** member can be modified even if its class object is ***const.*** Due to the semantics of ***mutable,*** the programmer must decide whether a member will be ***const*** or ***mutable*** because it cannot be both at the same time.

Example:
```
struct S {
    mutable const int * p;  // OK
    mutable int * const q;  // error
};
```

713 *left operand cannot be of type bool*

The left hand side of an assignment operator cannot be of type ***bool*** except for simple assignment. This is a restriction required in the C++ language.

Example:
```
bool q;

void fn()
{
    q += 1;
}
```

714 *operand cannot be of type bool*

The operand of both postfix and prefix "--" operators cannot be of type ***bool.*** This is a restriction required in the C++ language.

Example:
```
bool q;

void fn()
{
    --q;   // error
    q--;   // error
}
```

715 *member '%N' has not been declared in '%T'*

The compiler has found a member which has not been previously declared. The symbol may be spelled differently than the declaration, or the declaration may simply not be present.

Example:
```
struct X { int m; };

void fn( X *p )
{
    p->x = 1;
}
```

716 *integral value may be truncated*

This message indicates that the compiler knows that all values will not be preserved after the assignment or initialization. If this is acceptable, cast the value to the appropriate type in the assignment or initialization.

Example:
```
char inc( char c )
{
    return c + 1;
}
```

717 *left operand type is '%T'*

This informational message indicates the type of the left hand side of the expression.

718 *right operand type is '%T'*

This informational message indicates the type of the right hand side of the expression.

719 *operand type is '%T'*

This informational message indicates the type of the operand.

720 *expression type is '%T'*

This informational message indicates the type of the expression.

721 *virtual function '%S' cannot have its return type changed*

This restriction is due to the relatively new feature in the C++ language that allows return values to be changed when a virtual function has been overridden. It is not possible to support both features because in order to support changing the return value of a function, the compiler must construct a "wrapper" function that will call the virtual function first and then change the return value and return. It is not possible to do this with "..." style functions because the number of parameters is not known.

Example:

```
struct B {
};
struct D : virtual B {
};

struct X {
    virtual B *fn( int, ... );
};
struct Y : X {
    virtual D *fn( int, ... );
};
```

722 *__declspec('%N') is not supported*

The identifier used in the ***__declspec*** declaration modifier is not supported by Open Watcom C++.

723 *attempt to construct a far object when data model is near*

Constructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:
```
struct Obj
{   char *p;
    Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

724 *-zo is an obsolete switch (has no effect)*

The ***-zo*** option was required in an earlier version of the compiler but is no longer used.

725 *"%s"*

This is a user message generated with the ***#pragma message*** preprocessing directive.

Example:
```
#pragma message( "my very own warning" );
```

726 *no reference to formal parameter '%S'*

There are no references to the declared formal parameter. The simplest way to remove this warning in C++ is to remove the name from the argument declaration.

Example:

```
int fn1( int a, int b, int c )
{
    // 'b' not referenced
    return a + c;
}
int fn2( int a, int /* b */, int c )
{
    return a + c;
}
```

727 *cannot dereference a pointer to void*

A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

Example:

```
void fn( void *p )
{
    return *p;
}
```

728 *class modifiers for '%T' conflict with class modifiers for '%T'*

A conflict between class modifiers for classes related through inheritance has been detected. A conflict will occur if two base classes have class modifiers that are different. The conflict can be resolved by ensuring that all classes related through inheritance have the same class modifiers. The default resolution is to have no class modifier for the derived base.

Example:

```
struct __cdecl B1 {
    void fn( int );
};
struct __stdcall B2 {
    void fn( int );
};
struct D : B1, B2 {
};
```

729 *invalid hexadecimal constant*

The constant started with a '0x' prefix which makes it look like a hexadecimal constant but the constant was not followed by any hexadecimal digits.

Example:
```
unsigned i = 0x;      // invalid hex constant
```

730 *return type of 'operator ->' will not allow '->' to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.*** This warning indicates that the ***operator ->*** can never be used as an overloaded operator. The only way the operator can be used is to explicitly call it by name.

Example:
```
struct S {
    int a;
    void *operator ->();
};

void *fn( S &q )
{
    return q.operator ->();
}
```

731 *class should have a name since it needs a constructor or a destructor*

The class definition does not have a class name but it includes members that have constructors or destructors. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:
```
struct P {
    int x,y;
    P();
};

typedef struct {
    P c;
    int v;
} T;
```

732 *class should have a name since it inherits a class*

The class definition does not have a class name but it inherits a class. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:

```
struct P {
    int x,y;
    P();
};

typedef struct : P {
    int v;
} T;
```

733 *cannot open pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for read/write access.

734 *invalid second argument to va_start*

The second argument to the va_start macro should be the name of the argument just before the "..." in the argument list.

735 *'//' style comment continues on next line*

The compiler has detected a line continuation during the processing of a C++ style comment ("//"). The warning can be removed by switching to a C style comment ("/**/"). If you require the comment to be terminated at the end of the line, make sure that the backslash character is not the last character in the line.

Example:

```
#define XX 23 // comment start \
comment \
end

int x = XX; // comment start ...\
comment end
```

736 *cannot open file '%s' for write access*

The compiler has detected a problem while trying to open the indicated file for write access.

737 *implicit conversion of pointers to integral types of same size*

The compiler allows, when extensions are enabled, implicit conversions between pointers to integral types when the size of the integral types are the same. Thus, conversions from ***unsigned char*** to either ***char*** or ***signed char*** would be allowed. This is an extension as the ISO/ANSI Draft Working Paper permits implicit conversions only when the types pointed at are identical.

According to the ISO/ANSI Draft Working Paper, a string literal is an array of ***char.*** Consequently, it is illegal to initialize or assign the pointer resulting from that literal to a pointer of either ***unsigned char*** or ***signed char,*** since these pointers point at objects of a different type. When extensions are enabled, this condition is diagnosed as a warning; otherwise, it is an error.

738 *option requires a number*

The specified option is not recognized by the compiler since there was no number after it (i.e., "-w=1"). Numbers must be non-negative decimal numbers.

739 *option -fc specified more than once*

The -fc option can be specified at most once on a command line.

740 *option -fc specified in batch file of commands*

The -fc option cannot be specified on a line in the batch file of command lines specified by the -fc option on the command line used to invoke the compiler.

741 *file specified by -fc is empty or cannot be read*

The file specified using the -fc option is either empty or an input/output error was diagnosed for the file.

742 *cannot open file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

743 *input/output error reading the file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

744 *'%N' does not have a return type specified (int assumed)*

In C++, operator functions should have an explicit return type specified. In future revisions of the ISO/ANSI C++ standard, the use of default int type specifiers may be prohibited so removing any dependencies on default int early will prevent problems in the future.

Example:
```
struct S {
    operator = ( S const & );
    operator += ( S const & );
};
```

745 *cannot initialize reference to non-constant with a constant object*

A reference to a non-constant object cannot be initialized with a reference to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:
```
extern const int *pic;
extern int & ref = pic;
```

746 *processing %s*

This informational message indicates where an error or warning was detected while processing the switches specified on the command line, in environment variables, in command files (using the '@' notation), or in the batch command file (specified using the -fc option).

747 *class '%T' has not been defined*

This informational message indicates a class which was not defined. This is noted following an error or warning message because it often helps to a user to determine the cause of that diagnostic.

748 *cannot catch undefined class object*

C++ does not allow abstract classes to be copied and so an undefined class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an undefined class.

749 *class '%T' cannot be used since its definition has errors*

The analysis of the expression could not continue due to previous errors diagnosed in the class definition.

750 *function prototype in block scope missing 'extern'*

This warning can be triggered when the intent is to define a variable with a constructor. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as a function prototype. A work-around for this problem is contained in the example. If a prototype is desired, add the ***extern*** storage class to remove this warning.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    S a( C() ); // function prototype!
    S b( (C()) );// variable definition

    int bar( int );// warning
    extern int sam( int ); // no warning
}
```

751 *function prototype is '%T'*

This informational message indicates what the type of the function prototype is for the message in question.

752 *class '%T' contains a zero size array*

This warning is triggered when a class with a zero sized array is used in an array or as a class member. This is a questionable practice since a zero sized array at the end of a class often indicates a class that is dynamically sized when it is constructed.

Example:

```
struct C {
    C *next;
    char name[];
};

struct X {
    C q;
};

C a[10];
```

753 *invalid 'new' modifier*

The Open Watcom C++ compiler does not support new expression modifiers but allows them to match the ambient memory model for compatibility. Invalid memory model modifiers are also rejected by the compiler.

Example:

```
int *fn( unsigned x )
{
    return new __interrupt int[x];
}
```

754 *'__declspec(thread)' data '%S' must be link-time initialized*

This error message indicates that the data item in question either requires a constructor, destructor, or run-time initialization. This cannot be supported for thread-specific data at this time.

Example:

```
#include <stdlib.h>

struct C {
    C();
};
struct D {
    ~D();
};

C __declspec(thread) c;
D __declspec(thread) d;
int __declspec(thread) e = rand();
```

755 *code may not work properly if this module is split across a code segment*

The "zm" option allows the compiler to generate functions into separate segments that have different names so that more than 64k of code can be generated in one object file. Unfortunately, if an explicit near function is coded in a large code model, the possibility exists that the linker can place the near function in a separate code segment than a function that calls it. This would cause a linker error followed by an execution error if the executable is executed. The "zmf" option can be used if you require explicit near functions in your code.

Example:

```
// These functions may not end up in the
// same code segment if the -zm option
// is used. If this is the case, the near
// call will not work since near functions
// must be in the same code segment to
// execute properly.
static int near near_fn( int x )
{
    return x + 1;
}

int far_fn( int y )
{
    return near_fn( y * 2 );
}
```

756 *#pragma extref: symbol '%N' not declared*

This error message indicates that the symbol referenced by ***#pragma extref*** has not been declared in the context where the pragma was encountered.

757 *#pragma extref: overloaded function '%S' cannot be used*

An external reference can be emitted only for external functions which are not overloaded.

758 *#pragma extref: '%N' is not a function or data*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the referenced symbol is neither a function nor a data item. An external reference can be emitted only for external functions which are not overloaded and for external data items.

759 *#pragma extref: '%S' is not external*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the symbol is not external. An external reference can be emitted only for external functions which are not overloaded and for external data items.

760 *pre-compiled header file being recreated (debugging info may change)*

The compiler has detected that the module being compiled was used to create debugging information for use by other modules. In order to maintain correctness, the pre-compiled header file must be recreated along with the object file.

761 *octal escape sequence out of range; truncated*

This message indicates that the octal escape sequence produces an integer that cannot fit into the required character type.

Example:

```
char *p = "\406";
```

762 *binary operator '%s' missing right operand*

There is no expression to the right of the indicated binary operator.

763 *binary operator '%s' missing left operand*

There is no expression to the left of the indicated binary operator.

764 *expression contains extra operand(s)*

The expression contains operand(s) without an operator

765 *expression contains consecutive operand(s)*

More than one operand found in a row.

766 *unmatched right parenthesis ')'*

The expression contains a right parenthesis ")" without a matching left parenthesis.

767 *unmatched left parenthesis '('*

The expression contains a left parenthesis "(" without a matching right parenthesis.

768 *no expression between parentheses '( )'*

There is a matching set of parenthesis "()" which do not contain an expression.

769 *expecting ':' operator in conditional expression*

A conditional expression exists without the ':' operator.

770 *expecting '?' operator in conditional expression*

A conditional expression exists without the '?' operator.

771 *expecting first operand in conditional expression*

A conditional expression exists without the first operand.

772 *expecting second operand in conditional expression*

A conditional expression exists without the second operand.

773 *expecting third operand in conditional expression*

A conditional expression exists without the third operand.

774 *expecting operand after unary operator '%s'*

A unary operator without being followed by an operand.

775 *'%s' unexpected in constant expression*

'%s' not allowed in constant expression

776 *assembler: '%s'*

A warning has been issued by the #pragma inline assembler.

777 *expecting 'id' after '::' but found '%s'*

The '::' operator has an invalid token following it.

Example:

```
#define fn( x ) ((x)+1)

struct S {
    int inc( int y ) {
    return ::fn( y );
    }
};
```

778 *only constructors can be declared explicit*

Currently, only constructors can be declared with the ***explicit*** keyword.

Example:
```
int explicit fn( int x ) {
    return x + 1;
}
```

779 *const_cast type must be pointer, member pointer, or reference*

The type specified in a ***const_cast*** operator must be a pointer, a pointer to a member of a class, or a reference.

Example:
```
extern int const *p;
long lp = const_cast<long>( p );
```

780 *const_cast expression must be pointer to same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to the same type of object as that specified in the ***const_cast*** operator.

Example:
```
extern int const * ip;
long* lp = const_cast<long*>( ip );
```

781 *const_cast expression must be lvalue of the same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be an lvalue or reference to the same type of object as that specified in the ***const_cast*** operator.

Example:
```
extern int const i;
long& lr = const_cast<long&>( i );
```

782 *expression must be pointer to member from same class in const_cast*

The expression must be a pointer to member from the same class as that specified in the ***const_cast*** operator.

Example:
```
struct B {
    int ib;
};
struct D : public B {
};
extern int const B::* imb;
int D::* imd const_cast<int D::*>( imb );
```

783 *expression must be member pointer to same type as specified in const_cast*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to member of the same type as that specified in the ***const_cast*** operator.

Example:
```
struct B {
    int ib;
    long lb;
};
int D::* imd const_cast<int D::*>( &B::lb );
```

784 *reinterpret_cast expression must be pointer or integral object*

When a pointer type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer or an integer.

Example:
```
extern float fval;
long* lp = const_cast<long*>( fval );
```

The expression has ***float*** type and so is illegal.

785 *reinterpret_cast expression cannot be casted to reference type*

When a reference type is specified in the ***reinterpret_cast*** operator, the expression must be an lvalue (or have reference type). Additionally, constness cannot be casted away.

Example:

```
extern long f;
extern const long f2;
long& lr1 = const_cast<long&>( f + 2 );
long& lr2 = const_cast<long&>( f2 );
```

Both initializations are illegal. The first cast expression is not an lvalue. The second cast expression attempts to cast away constness.

786 *reinterpret_cast expression cannot be casted to pointer to member*

When a pointer to member type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer to member. Additionally, constness cannot be casted away.

Example:

```
extern long f;
struct S {
    const long f2;
    S();
};
long S::* mp1 = const_cast<long S:: *>( f );
long S::* mp2 = const_cast<long S:: *>( &S::f2 );
```

Both initializations are illegal. The first cast expression does not involve a member pointer. The second cast expression attempts to cast away constness.

787 *only integral arithmetic types can be used with reinterpret_cast*

Pointers can only be casted to sufficiently large integral types.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
```

The cast is illegal because ***float*** type is specified.

788 *only integral arithmetic types can be used with reinterpret_cast*

Only integral arithmetic types can be casted to pointer types.

Example:
```
float flt;
void* p = reinterpret_cast<void*>( flt );
```

The cast is illegal because `flt` has ***float*** type which is not integral.

789 *cannot cast away constness*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness from a pointer, reference, or pointer to member.

Example:
```
struct S {
    int s;
};
extern S const * ps;
extern int const S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness.

790 *size of integral type in cast less than size of pointer*

An object of the indicated integral type is too small to contain the value of the indicated pointer.

Example:
```
int x;
char p = reinterpret_cast<char>( &x );
char q = (char)( &x );
```

Both casts are illegal since a ***char*** is smaller than a pointer.

791 *type cannot be used in reinterpret_cast*

The type specified with reinterpret_cast must be an integral type, a pointer type, a pointer to a member of a class, or a reference type.

Example:
```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

792 *only pointers can be casted to integral types with reinterpret_cast*

The expression must be a pointer type.

Example:
```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

793 *only integers and pointers can be casted to pointer types with reinterpret_cast*

The expression must be a pointer or integral type.

Example:
```
void* x;
void* p = reinterpret_cast<void*>( 16 );
void* q = ( reinterpret_cast<void*>( x ), p );
```

The casts specify illegal types.

794 *static_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***static_cast*** operator. Perhaps reinterpret_cast or dynamic_cast should be used instead;

795 *static_cast cannot be used with the type specified*

A static cast cannot be used with a function type or array type.

Example:
```
typedef int fun( int );
extern int poo( long );
int i = ( static_cast<fun)( poo ) )( 22 );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

796 *static_cast cannot be used with the reference type specified*

The expression could not be converted to the specified type using static_cast.

Example:
```
long lng;
int& ref = static_cast<int&>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

797 *static_cast cannot be used with the pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:
```
long lng;
int* ref = static_cast<int*>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

798 *static_cast cannot be used with the member pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:
```
struct S {
    long lng;
};
int S::* mp = static_cast<int S::*>( &S::lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

799 *static_cast type is ambiguous*

More than one constructor and/or used-defined conversion function can be used to convert the expression to the indicated type.

800 *cannot cast from ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the base class to the derived class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Derived* foo( Base* p )
{
    return static_cast<Derived*>( p );
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

801 *cannot cast to ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the derived class to the base class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Base* foo( Derived* p )
{
    return (Base*)p;
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

802 *can only static_cast integers to enumeration type*

When an enumeration type is specified with ***static_cast,*** the expression must be an integer.

Example:
```
enum sex { male, female };
sex father = static_cast<sex>( 1.0 );
```

The cast is illegal because the expression is not an integer.

803 *dynamic_cast cannot be used with the type specified*

A dynamic cast can only specify a reference to a class or a pointer to a class or ***void.*** When a class is referenced, it must have virtual functions defined within that class or a base class of that class.

804 *dynamic_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***dynamic_cast*** operator. Only a pointer or reference to a class object can be converted. When a class object is referenced, it must have virtual functions defined within that class or a base class of that class.

805 *dynamic_cast requires class '%T' to have virtual functions*

The indicated class must have virtual functions defined within that class or a base class of that class.

806 *base class for type in dynamic_cast is ambiguous (will fail)*

The type in the ***dynamic_cast*** is a pointer or reference to an ambiguous base class.

Example:
```
struct A { virtual void f(){}; };
struct D1 : A { };
struct D2 : A { };
struct D : D1, D2 { };

A *foo( D *p ) {
    // will always return NULL
    return( dynamic_cast< A* >( p ) );
}
```

807 *base class for type in dynamic_cast is private (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a private base class.

Example:
```
struct V { virtual void f(){}; };
struct A : private virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

808 *base class for type in dynamic_cast is protected (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a protected base class.

Example:
```
struct V { virtual void f(){}; };
struct A : protected virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

809 *type cannot be used with an explicit cast*

The indicated type cannot be specified as the type of an explicit cast. For example, it is illegal to cast to an array or function type.

810 *cannot cast to an array type*

It is not permitted to cast to an array type.

Example:
```
typedef int array_type[5];
int array[5];
int* p = (array_type)array;
```

811 *cannot cast to a function type*

It is not permitted to cast to a function type.

Example:
```
typedef int fun_type( void );
void* p = (fun_type)0;
```

812 *implementation restriction: cannot generate RTTI info for '%T' (%d classes)*

The information for one class must fit into one segment. If the segment size is restricted to 64k, the compiler may not be able to emit the correct information properly if it requires more than 64k of memory to represent the class hierarchy.

813 *more than one default constructor for '%T'*

The compiler found more than one default constructor signature in the class definition. There must be only one constructor declared that accepts no arguments.

Example:
```
struct C {
    C();
    C( int = 0 );
};
C cv;
```

814 *user-defined conversion is ambiguous*

The compiler found more than one user-defined conversion which could be performed. The indicated functions that could be used are shown.

Example:
```
struct T {
    T( S const& );
};
struct S {
    operator T const& ();
};
extern S sv;
T const & tref = sv;
```

Either the constructor or the conversion function could be used; consequently, the conversion is ambiguous.

815 *range of possible values for type '%T' is %u to %u*

This informational message indicates the range of values possible for the indicated unsigned type.

Example:
```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the possible range of values for the unsigned type involved.

816 *range of possible values for type '%T' is %d to %d*

This informational message indicates the range of values possible for the indicated signed type.

Example:
```
signed char c;
if( c <= 127 );
```

Because the value of signed char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the possible range of values for the signed type involved.

817 *constant expression in comparison has value %d*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:
```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the constant value (0 in this case) involved in the comparison.

818 *constant expression in comparison has value %u*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:
```
signed char c;
if( c <= 127 );
```

Because the value of char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the constant value (127 in this case) involved in the comparison.

819 *conversion of const reference to non-const reference*

A reference to a constant object is being converted to a reference to a non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:
```
extern int const & const_ref;
int & non_const_ref = const_ref;
```

820 *conversion of volatile reference to non-volatile reference*

A reference to a volatile object is being converted to a reference to a non-volatile object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:
```
extern int volatile & volatile_ref;
int & non_volatile_ref = volatile_ref;
```

821 *conversion of const volatile reference to plain reference*

A reference to a constant and volatile object is being converted to a reference to a non-volatile and non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:
```
extern int const volatile & const_volatile_ref;
int & non_const_volatile_ref = const_volatile_ref;
```

822 *current declaration has type '%T'*

This informational message indicates the type of the current declaration that caused the message to be issued.

Example:
```
extern int __near foo( int );
extern int __far foo( int );
```

823 *only a non-volatile const reference can be bound to temporary*

The expression being bound to a reference will need to be converted to a temporary of the type referenced. This means that the reference will be bound to that temporary and so the reference must be a non-volatile const reference.

Example:
```
extern int * pi;
void * & r1 = pi;        // error
void * const & r2 = pi;     // ok
void * volatile & r3 = pi;  // error
void * const volatile & r4 = pi;// error
```

824 *conversion of pointer to member across a virtual base*

In November 1995, the Draft Working Paper was amended to disallow pointer to member conversions when the source class is a virtual base of the target class. This situation is treated as a warning (unless -za is specified to require strict conformance), as a temporary measure. In the future, an error will be diagnosed for this situation.

Example:
```
struct B {
    int b;
};

struct D : virtual B {
    int d;
};
int B::* mp_b = &B::b;
int D::* mp_d = mp_b;        // conversion across a
virtual base
```

825 *declaration cannot be in the same scope as namespace '%S'*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:
```
namespace x {
    int q;
};
int x;
```

826 *'%S' cannot be in the same scope as a namespace*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:
```
int x;
namespace x {
    int q;
};
```

827 *File: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the name of the file in which an error or warning was detected. The message precedes a group of one or more messages written for the file in question. Within each group, references within the file have the format `(line[,column])`.

828 *%s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected either before or after the source file was read during the compilation process.

829 *%s: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while processing the switches specified in a command file or by the contents of an environment variable. The switch that was being processed is displayed following the name of the file or the environment variable.

830 *%s: %S*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while generating a function, such as a constructor, destructor, or assignment operator or while generating the machine instructions for a function which has been analysed. The name of the function is given following text indicating the context from which the message originated.

831 *possible override is '%S'*

The indicated function is ambiguous since that name was defined in more than one base class and one or more of these functions is virtual. Consequently, it cannot be decided which is the virtual function to be used in a class derived from these base classes.

832 *function being overridden is '%S'*

This informational message indicates a function which cannot be overridden by a virtual function which has ellipsis parameters.

833 *name does not reference a namespace*

A ***namespace*** alias definition must reference a ***namespace*** definition.

Example:
```
typedef int T;
namespace a = T;
```

834 *namespace alias cannot be changed*

A ***namespace*** alias definition cannot change which ***namespace*** it is referencing.

Example:
```
namespace ns1 { int x; }
namespace ns2 { int x; }
namespace a = ns1;
namespace a = ns2;
```

835 *cannot throw undefined class object*

C++ does not allow undefined classes to be copied and so an undefined class object cannot be specified in a ***throw*** expression.

836 *symbol has different type than previous symbol in same declaration*

This warning indicates that two symbols in the same declaration have different types. This may be intended but it is often due to a misunderstanding of the C++ declaration syntax.

Example:
```
// change to:
//   char *p;
//   char q;
// or:
//   char *p, *q;
char* p, q;
```

837 *companion definition is '%S'*

This informational message indicates the other symbol that shares a common base type in the same declaration.

838 *syntax error; default argument cannot be processed*

The default argument contains unbalanced braces or parenthesis. The default argument cannot be processed in this form.

839 *default argument started %L*

This informational message indicates where the default argument started so that any problems with missing braces or parenthesis can be fixed quickly and easily.

Example:
```
struct S {
    int f( int t= (4+(3-7), // missing parenthesis
    );
};
```

840 *'%N' cannot be declared in a namespace*

A ***namespace*** cannot contain declarations or definitions of ***operator new*** or ***operator delete*** since they will never be called implicitly in a ***new*** or ***delete*** expression.

Example:
```
namespace N {
    void *operator new( unsigned );
    void operator delete( void * );
};
```

841 *namespace cannot be defined in a non-namespace scope*

A ***namespace*** can only be defined in either the global namespace scope (file scope) or a namespace scope.

Example:
```
struct S {
    namespace N {
    int x;
    };
}
```

842 *namespace '::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** functions. A ***namespace*** qualified name can only be declared in a namespace scope that encloses the qualified name's namespace.

Example:
```
namespace M {
    namespace N {
    void f();
    void g();
    namespace O {
        void N::f() {
          // error
        }
    }
    }
    void N::g() {
      // OK
    }
}
```

843 *cannot cast away volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove volatility from a pointer, reference, or pointer to member.

Example:
```
struct S {
    int s;
};
extern S volatile * ps;
extern int volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove volatility.

844 *cannot cast away constness and volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness and volatility from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S const volatile * ps;
extern int const volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness and volatility.

845 *cannot cast away unaligned*

A cast or implicit conversion is illegal because a conversion to the target type would add alignment to a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S _unaligned * ps;
extern int _unaligned S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to add alignment.

846 *subscript expression must be integral*

Both of the operands of the indicated index expression are pointers. There may be a missing indirection or function call.

Example:

```
int f();
int *p;
int g() {
    return p[f];
}
```

847 *extension: non-standard user-defined conversion*

An extended conversion was allowed. The latest draft of the C++ working paper does not allow a user-defined conversion to be used in this context. As an extension, the WATCOM compiler supports the conversion since substantial legacy code would not compile without the extension.

848 *useless using directive ignored*

This warning indicates that for most purposes, the ***using namespace*** directive can be removed.

Example:

```
namespace A {
    using namespace A;  // useless
};
```

849 *base class virtual function has not been overridden*

This warning indicates that a virtual function name has been overridden but in an incomplete manner, namely, a virtual function signature has been omitted in the overriding class.

Example:

```
struct B {
    virtual void f() const;
};
struct D : B {
    virtual void f();
};
```

850 *virtual function is '%S'*

This message indicates which virtual function has not been overridden.

851 *macro '%s' defined %L*

This informational message indicates where the macro in question was defined. The message is displayed following an error or warning diagnostic for the macro in question.

Example:

```
#define mac(a,b,c) a+b+c

int i = mac(6,7,8,9,10);
```

The expansion of macro `mac` is erroneous because it contains too many arguments. The informational message will indicate where the macro was defined.

852 *expanding macro '%s' defined %L*

These informational messages indicate the macros that are currently being expanded, along with the location at which they were defined. The message(s) are displayed following a diagnostic which is issued during macro expansion.

853 *conversion to common class type is impossible*

The conversion to a common class is impossible. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type is impossible.

854 *conversion to common class type is ambiguous*

The conversion to a common class is ambiguous. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : public A { B(); };
class C : public A { C(); };
class D : public B, public C { D(); };
extern A a;
extern D d;
int i = ( a == d );
```

The last statement is erroneous since a conversion to a common class type is ambiguous.

855 *conversion to common class type requires private access*

The conversion to a common class violates the access permission which was private. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : private A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (private) access permission.

856 *conversion to common class type requires protected access*

The conversion to a common class violates the access permission which was protected. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : protected A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (protected) access permission.

857 *namespace lookup is ambiguous*

A lookup for a name resulted in two or more non-function names being found. This is not allowed according to the C++ working paper.

Example:

```
namespace M {
    int i;
}
namespace N {
    int i;
    using namespace M;
}
void f() {
    using namespace N;
    i = 7;  // error
}
```

858 *ambiguous namespace symbol is '%S'*

This informational message shows a symbol that conflicted with another symbol during a lookup.

859 *attempt to static_cast from a private base class*

An attempt was made to static_cast a pointer or reference to a private base class to a derived class.

Example:
```
struct PrivateBase {
};

struct Derived : private PrivateBase {
};

extern PrivateBase* pb;
extern PrivateBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a private base class.

860 *attempt to static_cast from a protected base class*

An attempt was made to static_cast a pointer or reference to a protected base class to a derived class.

Example:
```
struct ProtectedBase {
};

struct Derived : protected ProtectedBase {
};

extern ProtectedBase* pb;
extern ProtectedBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a protected base class.

861 *qualified symbol cannot be defined in this scope*

This message indicates that the scope of the symbol is not nested in the current scope. This is a restriction in the C++ language.

Example:

```
namespace A {
    struct S {
    void ok();
    void bad();
    };
    void ok();
    void bad();
};
void A::S::ok() {
}
void A::ok() {
}
namespace B {
    void A::S::bad() {
    // error!
    }
    void A::bad() {
    // error!
    }
};
```

862 *using declaration references non-member*

This message indicates that the entity referenced by the ***using*** declaration is not a class member even though the ***using*** declaration is in class scope.

Example:

```
namespace B {
    int x;
};
struct D {
    using B::x;
};
```

863 *using declaration references class member*

This message indicates that the entity referenced by the ***using*** declaration is a class member even though the ***using*** declaration is not in class scope.

Example:

```
struct B {
    int m;
};
using B::m;
```

864 *invalid suffix for a constant*

An invalid suffix was coded for a constant.

Example:

```
__int64 a[] = {
    0i7, // error
    0i8,
    0i15, // error
    0i16,
    0i31, // error
    0i32,
    0i63, // error
    0i64,
};
```

865 *class in using declaration ('%T') must be a base class*

A ***using*** declaration declared in a class scope can only reference entities in a base class.

Example:

```
struct B {
    int f;
};
struct C {
    int g;
};
struct D : private C {
    B::f;
};
```

866 *name in using declaration is already in scope*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:
```
namespace B {
     int f;
     using B::f;
};
```

867 *conflict with a previous using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:
```
namespace B {
     int f;
     using B::f;
};
```

868 *conflict with current using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:
```
namespace B {
     int f;
     using B::f;
};
```

869 *use of '%N' requires build target to be multi-threaded*

The compiler has detected a use of a run-time function that will create a new thread but the current build target indicates only single-threaded C++ source code is expected. Depending on the user's environment, enabling multi-threaded applications can involve using the "-bm" option or selecting multi-threaded applications through a dialogue.

870 *implementation restriction: cannot use 64-bit value in switch statement*

The use of 64-bit values in switch statements has not been implemented.

871 *implementation restriction: cannot use 64-bit value in case statement*

The use of 64-bit values in case statements has not been implemented.

872 *implementation restriction: cannot use __int64 as bit-field base type*

The use of ***__int64*** for the base type of a bit-field has not been implemented.

873 *based function object cannot be placed in non-code segment "%s".*

Use ***__segname*** with the default code segment "_CODE", or a code segment with the appropriate suffix (indicated by informational message).

Example:
```
    int __based(__segname("foo")) f() {return 1;}
```

Example:
```
    int __based(__segname("_CODE")) f() {return 1;}
```

874 *Use a segment name ending in "%s", or the default code segment "_CODE".*

This informational message explains how to use ***__segname*** to name a code segment.

875 *RTTI must be enabled to use feature (use 'xr' option)*

RTTI must be enabled by specifying the 'xr' option when the compiler is invoked. The error message indicates that a feature such as ***dynamic_cast,*** or ***typeid*** has been used without enabling RTTI.

876 *'typeid' class type must be defined*

The compile-time type of the expression or type must be completely defined if it is a class type.

Example:

```
struct S;
void foo( S *p ) {
    typeid( *p );
    typeid( S );
}
```

877 *cast involves unrelated member pointers*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1 {
    int poo();
};

typedef int (C1::* C1mp )();

C1mp fmp = (C1mp)&D1::poo;
```

The cast on the last line of the example would be diagnosed.

878 *unexpected type modifier found*

A ***__declspec*** modifier was found that could not be applied to an object or could not be used in this context.

Example:

```
__declspec(thread) struct S {
};
```

879 *invalid bit-field name '%N'*

A bit-field can only have a simple identifier as its name. A qualified name is also not allowed for a bit-field.

Example:

```
struct S {
    int operator + : 1;
};
```

880 *%u padding byte(s) added*

This warning indicates that some extra bytes have been added to a class in order to align member data to its natural alignment.

Example:

```
#pragma pack(push,8)
struct S {
    char c;
    double d;
};
#pragma pack(pop);
```

881 *cannot be called with a '%T *'*

This message indicates that the virtual function cannot be called with a pointer or reference to the current class.

882 *cast involves an undefined member pointer*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host class of at least one member pointer was not a fully defined class and, as such, it is unknown whether the host classes are related through derivation. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1;

typedef int (C1::* C1mp )();
typedef int (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
// D1 may derive from C1
```

The cast on the last line of the example would be diagnosed.

883 *cast changes both member pointer object and class type*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host classes of the member pointers are related through derivation and the object type is also being changed. The cast can be broken up into two casts, one that changes the host class without changing the object type, and another that changes the object type without changing the host class.

Example:

```
struct C1 {
    int fn1();
};
struct D1 : C1 {
    int fn2();
};

typedef int (C1::* C1mp )();
typedef void (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
```

The cast on the last line of the example would be diagnosed.

884 *virtual function '%S' has a different calling convention*

This error indicates that the calling conventions specified in the virtual function prototypes are different. This means that virtual function calls will not function properly since the caller and callee may not agree on how parameters should be passed. Correct the problem by deciding on one calling convention and change the erroneous declaration.

Example:
```
struct B {
    virtual void __cdecl foo( int, int );
};
struct D : B {
    void foo( int, int );
};
```

885 *#endif matches #if in different source file*

This warning may indicate a ***#endif*** nesting problem since the traditional usage of ***#if*** directives is confined to the same source file. This warning may often come before an error and it is hoped will provide information to solve a preprocessing directive problem.

886 *preprocessing directive found %L*

This informational message indicates the location of a preprocessing directive associated with the error or warning message.

887 *unary '-' of unsigned operand produces unsigned result*

When a unary minus ('-') operator is applied to an unsigned operand, the result has an unsigned type rather than a signed type. This warning often occurs because of the misconception that '-' is part of a numeric token rather than as a unary operator. The work-around for the warning is to cast the unary minus operand to the appropriate signed type.

Example:
```
extern void u( int );
extern void u( unsigned );
void fn( unsigned x ) {
    u( -x );
    u( -2147483648 );
}
```

888 *trigraph expansion produced '%c'*

Trigraph expansion occurs at a very low-level so it can affect string literals that contain question marks. This warning can be disabled via the command line or ***#pragma warning*** directive.

Example:
```
// string expands to "(?]?~????"!
char *e = "(???)???-????";
// possible work-arounds
char *f = "(" "???" ")" "???" "-" "????";
char *g = "(\?\?\?)\?\?\?-\?\?\?\?";
```

889 *hexadecimal escape sequence out of range; truncated*

This message indicates that the hexadecimal escape sequence produces an integer that cannot fit into the required character type.

Example:
```
char *p = "\x0aCache Timings\x0a";
```

890 *undefined macro '%s' evaluates to 0*

The ISO C/C++ standard requires that undefined macros evaluate to zero during preprocessor expression evaluation. This default behaviour can often mask incorrectly spelled macro references. The warning is useful when used in critical environments where all macros will be defined.

Example:
```
#if _PRODUCTI0N // should be _PRODUCTION
#endif
```

891 *char constant has value %u (more than 8 bits)*

The ISO C/C++ standard requires that multi-char character constants be accepted with an implementation defined value. This default behaviour can often mask incorrectly specified character constants.

Example:
```
int x = '\0x1a'; // warning
int y = '\x1a';
```

892 *promotion of unadorned char type to int*

This message is enabled by the hidden -jw option. The warning may be used to locate all places where an unadorned char type (i.e., a type that is specified as ***char*** and neither ***signed char*** nor ***unsigned char***). This may cause portability problems since compilers have freedom to specify whether the unadorned char type is to be signed or unsigned. The promotion to ***int*** will have different values, depending on the choice being made.

893 *switch statement has no case labels*

The switch statement referenced in the warning did not have any case labels. Without case labels, a switch statement will always jump to the default case code.

Example:
```
void fn( int x )
{
    switch( x ) {
    default:
    ++x;
    }
}
```

894 *unexpected character (%u) in source file*

The compiler has encountered a character in the source file that is not in the allowable set of input characters. The decimal representation of the character byte is output for diagnostic purposes.

Example:
```
// invalid char '\0'
```

895 *ignoring whitespace after line splice*

The compiler is ignoring some whitespace characters that occur after the line splice. This warning is useful when the source code must be compiled with other compilers that do not allow this extension.

Example:
```
#define XXXX int \
x;

XXXX
```

896 *empty member declaration*

The compiler is warning about an extra semicolon found in a class definition. The extra semicolon is valid C++ but some C++ compilers do not accept this as valid syntax.

Example:
```
struct S { ; };
```

897 *'%S' makes use of a non-portable feature (zero-sized array)*

The compiler is warning about the use of a non-portable feature in a declaration or definition. This warning is available for environments where diagnosing the use of non-portable features is useful in improving the portability of the code.

Example:
```
struct D {
    int d;
    char a[];
};
```

898 *in-class initialization is only allowed for const static integral members*

Example:
```
struct A {
    static int i = 0;
};
```

899 *cannot convert expression to target type*

The implicit cast is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended cast.

Example:
```
struct T {
};

void fn()
{
    bool b = T;
}
```

900 *unknown template specialization of '%S'*

Example:
```
template<class T>
struct A { };

template<class T>
void A<T *>::f() {
}
```

901 *wrong number of template arguments for '%S'*

Example:
```
template<class T>
struct A { };

template<class T, class U>
struct A<T, U> { };
}
```

902 *cannot explicitly specialize member of '%S'*

Example:
```
template<class T>
struct A { };

template<>
struct A<int> {
    void f();
};

template<>
void A<int>::f() {
}
```

903 *specialization arguments for '%S' match primary template*

Example:
```
template<class T>
struct A { };

template<class T>
struct A<T> { };
```

904 *partial template specialization for '%S' ambiguous*

Example:
```
template<class T, class U>
struct A { };

template<class T, class U>
struct A<T *, U> { };

template<class T, class U>
struct A<T, U *> { };

A<int *, int *> a;
```

905 *static assertion failed '%s'*

Example:
```
static_assert( false, "false" );
```

906 *Exported templates are not supported by Open Watcom C++*

Example:
```
export template< class T >
struct A {
};
```

907 *redeclaration of member function '%S' not allowed*

Example:
```
struct A {
    void f();
    void f();
};
```

908 *candidate defined %L*

909 *Invalid register name '%s' in #pragma*

The register name is invalid/unknown.

D. Open Watcom C/C++ Run-Time Messages

The following is a list of error messages produced by the Open Watcom C/C++ run-time library. These messages can only appear during the execution of an application built with one of the C run-time libraries.

D.1 Run-Time Error Messages

Assertion failed: %s, file %s, line %d

This message is displayed whenever an assertion that you have made in your program is not true.

Stack Overflow!

Your program is trying to use more stack space than is available. If you believe that your program is correct, you can increase the size of the stack by using the "option stack=nnnn" when you link the program. The stack size can also be specified with the "N" option if you are using cc.

Floating-point support not loaded

You have called one of the printf functions with a format of "%e", "%f", or "%g", but have not passed a floating-point value. The compiler generates a reference to the variable "_fltused_" whenever you pass a floating-point value to a function. During the linking phase, the extra floating-point formatting routines will also be brought into your application when "_fltused_" is referenced. Otherwise, you only get the non floating-point formatting routines.

****** NULL assignment detected***

This message is displayed if any of the first 32 bytes of your program's data segment has been modified. The check is performed just before your program exits to the operating system. All this message means is that sometime during the execution of your program, this memory was modified.

To find the problem, you must link your application with debugging information and use Open Watcom Debugger to monitor its execution. First, run the application with Open Watcom Debugger until it completes. Examine the first 16 bytes of the data segment ("examine __nullarea") and press the space bar to see the next 16 bytes. Any values that are not equal to '01' have been modified. Reload the application, set watch points on the modified locations, and start execution. Open Watcom Debugger will stop when the specified location(s) change in value.

D.2 errno Values and Their Meanings

The following errors can be generated by the C run-time library. These error codes correspond to the error types defined in `errno.h.`

EOK (0) *No error*

EPERM (1) *Operation not permitted*

ENOENT (2) *No such file or directory*

The specified file or directory cannot be found.

ESRCH (3) *No such process*

EINTR (4) *Interrupted function call*

EIO (5) *I/O error*

ENXIO (6) *No such device or address*

E2BIG (7) *Arg list too big*

The argument list passed to the `spawn...`, `exec...` or `system` functions exceeds the limit imposed by QNX, or the environment information exceeds 64K.

ENOEXEC (8) *Exec format error*

The executable file has an invalid format.

EBADF (9) *Bad file descriptor*

The file descriptor is not a valid file descriptor value or it does not correspond to an open file.

ECHILD (10) *No child processes*

EAGAIN (11) *Resource unavailable; try again*

ENOMEM (12) *Not enough memory*

There was not enough memory available to perform the specified request.

EACCES (13) *Permission denied*

You do not have the required (or correct) permissions to access a file.

EFAULT (14) *Bad address*

ENOTBLK (15) *Block device required*

EBUSY (16) *Resource busy*

EEXIST (17) *File exists*

An attempt was made to create a file with the O_EXCL (exclusive) flag when the file already exists.

EXDEV (18) *Improper link*

An attempt was made to rename a file to a different device.

ENODEV (19) *No such device*

ENOTDIR (20) *Not a directory*

EISDIR (21) *Is a directory*

EINVAL (22) *Invalid argument*

An invalid value was specified for one of the arguments to a function.

ENFILE (23) — *Too many files in the system*

All the FILE structures are in use, so no more files can be opened.

EMFILE (24) — *Too many open files*

There are no more file descriptors available, so no more files can be opened.

ENOTTY (25) — *Inappropriate I/O control operation*

ETXTBSY (26) — *Text file busy*

EFBIG (27) — *File too large*

ENOSPC (28) — *No space left on device*

No more space is left for writing on the device, which usually means that the disk is full.

ESPIPE (29) — *Invalid seek*

EROFS (30) — *Read-only file system*

EMLINK (31) — *Too many links*

EPIPE (32) — *Broken pipe*

EDOM (33) — *Math arg out of domain of func*

An argument to a math function is not in the domain of the function.

ERANGE (34) — *Result too large*

The result of a math function could not be represented (too small, or too large).

ENOMSG (35) — *No message of desired type*

EIDRM (36) *Identifier removed*

ECHRNG (37) *Channel number out of range*

EL2NSYNC (38) *Level 2 not synchronized*

EL3HLT (39) *Level 3 halted*

EL3RST (40) *Level 3 reset*

ELNRNG (41) *Link number out of range*

EUNATCH (42) *Protocol driver not attached*

ENOCSI (43) *No CSI structure available*

EL2HLT (44) *Level 2 halted*

EDEADLK (45) *Resource deadlock avoided*

A resource deadlock would occur with regards to locked files.

ENOLCK (46) *No locks available*

ELOOP (62) *Too many levels of symbolic links or prefixes*

ENAMETOOLONG (78) *Filename too long*

D.2.1 Shared Library Errors

ELIBACC (83) *Can't access shared library*

ELIBBAD (84) *Accessing a corrupted shared library*

ELIBSCN (85) *.lib section in a.out corrupted*

ELIBMAX (86) *Attempting to link in too many libraries*

ELIBEXEC (87) *Attempting to exec a shared library*

ENOSYS (89) *Function not implemented*

ENOTEMPTY (93)	*Directory not empty*
EOPNOTSUPP (103)	*Operation not supported*
ESTALE (122)	*Potentially recoverable i/o error*

D.2.2 Non-blocking and Interrupt I/O

EWOULDBLOCK (11)	*Operation would block*
EINPROGRESS (236)	*Operation now in progress*
EALREADY (16)	*Operation already in progress*

D.2.3 IPC/Network Software -- Argument Errors

ENOTSOCK (238)	*Socket operation on non-socket*
EDESTADDRREQ (239)	*Destination address required*
EMSGSIZE (240)	*Message too long*
EPROTOTYPE (241)	*Protocol wrong type for socket*
ENOPROTOOPT (242)	*Protocol not available*
EPROTONOSUPPORT (243)	*Protocol not supported*
ESOCKTNOSUPPORT (244)	*Socket type not supported*
EPFNOSUPPORT (246)	*Protocol family not supported*
EAFNOSUPPORT (247)	*Address family not supported by protocol family*
EADDRINUSE (248)	*Address already in use*
EADDRNOTAVAIL (249)	*Can’t assign requested address*

D.2.4 IPC/Network Software -- Operational Errors

ENETDOWN (250)	*Network is down*
ENETUNREACH (251)	*Network is unreachable*
ENETRESET (252)	*Network dropped connection on reset*
ECONNABORTED (253)	*Software caused connection abort*
ECONNRESET (254)	*Connection reset by peer*
ENOBUFS (255)	*No buffer space available*
EISCONN (256)	*Socket is already connected*
ENOTCONN (257)	*Socket is not connected*
ESHUTDOWN (258)	*Can't send after socket shutdown*
ETOOMANYREFS (259)	*Too many references: can't splice*
ETIMEDOUT (260)	*Connection timed out*
ECONNREFUSED (261)	*Connection refused*
EHOSTDOWN (264)	*Host is down*
EHOSTUNREACH (265)	*No route to host*

D.2.5 QNX Specific

ENOREMOTE (1000)	*Must be done on local machine*
ENONDP (1001)	*Need an NDP (8087...) to run*
EBADFSYS (1002)	*Corrupted file system detected*
ENO32BIT (1003)	*32-bit integer fields were used*
ENOVPE (1004)	*No proc entry available for virtual process*

ENONETQ (1005)	*Process manager-to-net enqueuing failed*
ENONETMAN (1006)	*Could not find net manager for node number*
EVIDBUF2SML (1007)	*Told to allocate a vid buf too small*
EVIDBUF2BIG (1008)	*Told to allocate a vid buf too big*
EMORE (1009)	*More to do; send message again*
ECTRLTERM (1010)	*Remap to controlling terminal*
ENOLIC (1011)	*No license*

D.3 Math Run-Time Error Messages

The following errors can be generated by the math functions in the C run-time library. These error codes correspond to the exception types defined in `math.h` and returned by the `matherr` function when a math error occurs.

DOMAIN — *Domain error*

An argument to the function is outside the domain of the function.

OVERFLOW — *Overflow range error*

The function result is too large.

PLOSS — *Partial loss of significance*

A partial loss of significance occurred.

SING — *Argument singularity*

An argument to the function has a bad value (e.g., log(0.0)).

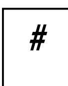
TLOSS — *Total loss of significance*

A total loss of significance occurred. An argument to a function was too large to produce a meaningful result.

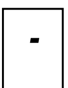
UNDERFLOW

Underflow range error

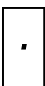
The result is too small to be represented.



directive 276
#define 673, 693
#elif 508-509
#else 508-509, 667
#endif 469, 508-509, 523, 667, 739
#error 103, 189, 523
#if 469, 508-509, 523, 739
#ifdef 523
#ifndef 523, 693
#include 18, 510, 516, 518-519, 617, 690, 693
#pragma 30, 36, 682, 696
#pragma extref 705
#pragma warning 468, 740
#undef 525, 672



-zo 696



.186 409
.286 409
.286c 409
.286p 409
.287 409
.386 409
.386p 409
.387 409
.486 409
.486p 409
.586 409
.586p 409
.686 409
.686p 409
.8086 409
.8087 409
.alpha 409, 418
.break 409, 418
.code 409
.const 409
.continue 409, 418
.cref 409, 418
.data 409
.data? 409
.dosseg 409
.else 418
.endif 418
.endw 409, 418
.err 409
.errb 409
.errdef 409
.errdif 409
.errdifi 409
.erre 409
.erridn 409
.erridni 409
.errnb 409
.errndef 409
.errnz 409
.exit 409, 418
.fardata 409
.fardata? 409
.if 418
.lfcond 409, 418
.list 409, 418
.listall 409, 418
.listif 409, 418
.listmacro 409, 418
.listmacroall 409, 418
.model 409
.nocref 409, 418
.nolist 409, 418
.radix 409, 418
.repeat 409, 418

.sall 409, 418
.seq 409, 418
.sfcond 409, 418
.stack 409
.startup 409, 418
.tfcond 409, 418
.until 409, 418
.while 409, 418
.xcref 409, 418
.xlist 409, 418

/

/include directory 20
/lib 60

1

16-bit QNX executables 267

3

32-bit QNX executables 267
__386__ 23

<

<os>_INCLUDE environment variable 19

A

aborts (pragma) 134, 220
addr 418
addressing arguments 83, 166, 169
ALIAS directive 272
alias name (pragma) 113, 199
alias names
  cdecl 116, 202
  fastcall 116, 202
  fortran 116, 202
  pascal 116, 202
  stdcall 116, 202
  syscall 202
  system 202
  watcall 116, 202
alloc_text pragma 97, 183
apostrophes 270
applications
  creating for QNX 359
AR-format 263
argument list (pragma) 123, 209
arguments
  removing from the stack 128, 215
arguments on the stack 126, 213
ARTIFICIAL option 273
__asm 46, 248
assembler 407
assembly language
  automatic variables 246
  directives 250
  in-line 237
  labels 245
  opcodes 250
  variables 245
auto 480-481, 485, 519, 531, 538, 545, 547, 584, 593
AUTODEPEND 110, 196
auxiliary pragma 112, 198

B

base operator 41
__based 30, 39, 518
based pointers 39
  segment constant 40
  segment object 41
  self 42
  void 41
benchmarking 14
_bheapseg 41
big code model 67, 149
big data model 68, 150
bin directory 456
BIOS call 127, 214
blanks in file names 270
bool 693-694
break 473, 507, 687-688

C

C directory 10
C libraries
  compact 58, 61
  flat 61-62, 152
  huge 58, 61
  large 58, 61
  medium 57, 61
  small 57, 61-62, 152
C/C++ libraries
  flat 58
  small 58
CACHE option 274
callback functions 122
calling convention
  MetaWare High C 201, 227
  Microsoft C 115, 140
calling conventions 73, 155
calling functions
  far 118, 206
  near 118, 206
calling information (pragma) 118, 206
case 467, 473, 484, 507, 522, 553, 654
CASEEXACT option 275
casemap 418
catch 518, 556, 663, 666-667, 685, 702
catstr 418
cc 457
cdecl 30-32, 116, 200, 202
cdecl alias name 116, 202
char 35-36, 478, 480, 515, 700, 711, 741
  size of 80, 162
char type 74, 156
__CHAR_SIGNED__ 25
check_stack option 94, 180
class 513, 529-530, 550, 566, 605, 662, 671
  CODE 71, 78, 153, 160
  FAR_DATA 71, 78, 153, 160
class information 101, 187
clib3r.lib 58
clib3s.lib 58
clibc.lib 58
clibh.lib 58
clibl.lib 58
clibm.lib 58
clibs.lib 58
cmain.c 62-63
CODE class 71, 78, 153, 160
code generation 50
  memory requirements 50, 459
code models
  big 67, 149
  small 67, 149
code_seg pragma 98, 184
CodeView 277
COFF 263
command line format 9
  wasm 407
  wdis 433
  wlib 388
  wlink 264, 359

wstrip 449
command line options
compiler 11
environment variable 11
options file 11
command name
compiler 10
comment (#) directive 276
comment pragma 99, 185
compact memory model 69, 150
compact model
libraries 58, 61
Compactor 277
compiler
features 9
compiling
command line format 9
const 475, 480, 537-538, 583-584, 619, 621-622, 624-625, 693, 708-709
const_cast 708-709
continue 473, 507
conventions
80x87 90, 92, 176, 178
non-80x87 79, 161
__cplusplus 25
cplx3r.lib 59
cplx3s.lib 59
cplx73r.lib 59
cplx73s.lib 59
cplx7c.lib 59
cplx7h.lib 59
cplx7l.lib 59
cplx7m.lib 59
cplx7s.lib 59
cplxc.lib 59
cplxh.lib 59
cplxl.lib 59
cplxm.lib 59
cplxs.lib 59
__CPPRTTI 26
__CPPUNWIND 26
cstart.asm 62
cstart_*.asm 62
cstart_c.asm 62
cstart_h.asm 62
cstart_l.asm 62
cstart_m.asm 62
cstart_s.asm 62
cstrt386.asm 63
CV4 277
CVPACK 277-278
CVPACK option 277

D

data models
big 68, 150
huge 68
small 68, 150
data representation 73, 155
data types 73, 155
data_seg pragma 99, 185
DBCS
Chinese 307
Japanese 307
Korean 307
dead code elimination 289, 341-342
DEBUG directive 278
debug information
removal 449
DEBUG options
ALL 279
CODEVIEW 278
DWARF 278
LINES 279
LOCALS 279
NOVELL 279
ONLYEXPORTS 279, 282
REFERENCED 279
TYPES 279
Watcom 278
debugging information
all 282
for NetWare debugger 282

global symbol 279, 282
line numbering 279-280
local symbol 279, 281
NetWare global symbol 279
strip from "EXE" file 284
typing 279, 281
Debugging Information Compactor 277-278
__declspec 32, 44, 696, 736
__declspec(dllexport) 47
__declspec(dllimport) 47
default 473-474, 484, 507, 509, 522, 654
default directive file 266, 269, 275, 291, 353
wlink.lnk 275, 291
default filename extension 10
default libraries
using pragmas 96, 182
delete 503, 520, 544, 612, 638, 650, 724
diagnostics
errno 748
error 18
matherr 754
Open Watcom C/C++ 17
run-time 748, 754
warning 18
wstrip 451
directives 269
276
ALIAS 272
assembly language 250
comment 276
DEBUG 278
DISABLE 285
ENDLINK 290
FILE 292
FORMAT 295
include 304
LANGUAGE 307
LIBFILE 308
LIBPATH 310
LIBRARY 312
MODFILE 320
MODTRACE 321
NAME 322
OPTION 325
OPTLIB 326
ORDER 328
OUTPUT 333
PATH 336
REFERENCE 341
SORT 343
STARTLINK 346
SYMTRACE 350
SYSTEM 351
directories
C 10
OCC 11
DISABLE directive 285
disable_message pragma 100, 186
disassembler 433
disassembly example 439
DLL 33
exporting functions 32
dllexport 32, 47
dllimport 32
do 473, 484, 507, 522
_DOS 24
DOS/4GW example 241
__DOS__ 24
DOSSEG option 287
double 480, 485
size of 80, 162
double type 76, 158
DPMI example 241
DS segment register 32-33
dump_object_model pragma 101, 187
Dynamic Link Library 33
exporting functions 32
imports 393, 396
dynamic_cast 715-716, 735

E

echo 418
_edata linker symbol 288

ELF 263
ELIMINATE option 289
emu387.lib 60
emu87.lib 60
enable_message pragma 101, 187
_end linker symbol 288
ENDLINK directive 290
endmacro 418
enum 475, 487, 491, 547, 565, 569
enum pragma 102, 188
enumerated types
  size of 81, 163
enumeration
  information 101, 187
  values 101, 187
environment variable
  command line options 11
environment variables 11
  <os>_INCLUDE 19
  FORCE 455
  INCLUDE 20-21, 455-456, 518
  LIB 313, 324, 327, 456
  LIBDIR 269
  OS2_INCLUDE 20
  PATH 20, 266, 269, 275, 291, 353, 456
  TMPDIR 457
  use 455
  WATCOM 60, 267, 269, 275, 291, 353, 456-457
  WCC 11, 458
  WCC386 11, 458
  WCGMEMORY 50-51, 459
  WD 459-460
  WD_PATH 460
  WPP 11, 460-461
  WPP386 11, 461
errno 748
  E2BIG 748
  EACCES 749
  EADDRINUSE 752
  EADDRNOTAVAIL 752
  EAFNOSUPPORT 752
  EAGAIN 749
  EALREADY 752
EBADF 748
EBADFSYS 753
EBUSY 749
ECHILD 749
ECHRNG 751
ECONNABORTED 753
ECONNREFUSED 753
ECONNRESET 753
ECTRLTERM 754
EDEADLK 751
EDESTADDRREQ 752
EDOM 750
EEXIST 749
EFAULT 749
EFBIG 750
EHOSTDOWN 753
EHOSTUNREACH 753
EIDRM 750
EINPROGRESS 752
EINTR 748
EINVAL 749
EIO 748
EISCONN 753
EISDIR 749
EL2HLT 751
EL2NSYNC 751
EL3HLT 751
EL3RST 751
ELIBACC 751
ELIBBAD 751
ELIBEXEC 751
ELIBMAX 751
ELIBSCN 751
ELNRNG 751
ELOOP 751
EMFILE 750
EMLINK 750
EMORE 754
EMSGSIZE 752
ENAMETOOLONG 751
ENETDOWN 753
ENETRESET 753
ENETUNREACH 753
ENFILE 750

ENO32BIT 753
ENOBUFS 753
ENOCSI 751
ENODEV 749
ENOENT 748
ENOEXEC 748
ENOLCK 751
ENOLIC 754
ENOMEM 749
ENOMSG 750
ENONDP 753
ENONETMAN 754
ENONETQ 753
ENOPROTOOPT 752
ENOREMOTE 753
ENOSPC 750
ENOSYS 751
ENOTBLK 749
ENOTCONN 753
ENOTDIR 749
ENOTEMPTY 751
ENOTSOCK 752
ENOTTY 750
ENOVPE 753
ENXIO 748
EOK 748
EOPNOTSUPP 752
EPERM 748
EPFNOSUPPORT 752
EPIPE 750
EPROTONOSUPPORT 752
EPROTOTYPE 752
ERANGE 750
EROFS 750
ESHUTDOWN 753
ESOCKTNOSUPPORT 752
ESPIPE 750
ESRCH 748
ESTALE 752
ETIMEDOUT 753
ETOOMANYREFS 753
ETXTBSY 750
EUNATCH 751
EVIDBUF2BIG 754
  EVIDBUF2SML 754
  EWOULDBLOCK 752
  EXDEV 749
error codes
  errno.h 748
  math.h 754
error file
  .err 17
error messages 463
error pragma 103, 189
errors 285, 363
executable files
  reducing size 449
executable formats 263
explicit 708
__export 11, 32-33, 47, 455, 627
  INCLUDE environment variable 20-21
export (pragma) 122, 209
exporting symbols in dynamic link libraries 122, 209
extension
  default 10
extern 44, 476, 482, 487, 512, 531, 545, 548, 595, 702
external references 103, 189
extref pragma 103, 189

F

far 14, 29, 32, 69, 151, 503, 627, 631, 655
far (pragma) 118, 206
far call 67, 149
far call optimizations 445
far jump optimization 446
far pointer
  size of 80, 162
__far16 35-36, 200, 518
FAR_DATA class 71, 78, 153, 160
FARCALLS option 291
fastcall 116, 202

fastcall alias name 116, 202
fastest 16-bit code 14
fastest 32-bit code 15
fatal errors 285, 363
FILE directive 292
filename extension 10
FILLCHAR option 294
_finally 491
flat memory model 150
flat model
  libraries 58, 61-62, 152
float 119, 480, 485, 584, 600, 709-711
  size of 80, 162
float type 75, 157
floating-point
  fltused 60
  __init_387_emulator 60
  __init_87_emulator 60
fltused 60
for 473, 486, 507, 555
FORCE environment variable 455
FORMAT directive 295
fortran 31-32, 49, 116, 202, 465
fortran alias name 116, 202
__FPI__ 25
frame (pragma) 123, 209
friend 537, 569, 585, 635, 725
function pragma 104, 190
function prototypes
  effect on arguments 81, 163
functions
  returning values 86, 172

G

general directives/options 269
goto 467, 475, 478, 511, 514

H

header file
  including 18
  searching 19
high 418
High C calling convention 227
highword 418
host operating system 264
huge 29, 69, 151, 494
huge data model 68
huge memory model 69
huge model
  libraries 58, 61

I

__I86__ 23
if 676
import library 393, 396
in-line 80x87 floating-point instructions 120
in-line assembly
  in pragmas 118, 206
in-line assembly language 237
  automatic variables 246
  directives 250
  labels 245
  opcodes 250
  variables 245
in-line assembly language instructions
  using mnemonics 120, 207
in-line functions 119, 207
in-line functions (pragma) 127, 214
include
  directive 18
  header file 18
  source file 18

include directive 304
INCLUDE environment variable 20-21, 455-456, 518
include file
  searching 19
__init_387_emulator 60
__init_87_emulator 60
initialize pragma 105, 191
inline 535
inline_depth pragma 106, 192
__INLINE_FUNCTIONS__ 25
inline_recursion pragma 107, 193
int 17, 35, 469, 472-473, 478, 480, 515, 537, 578, 600, 603, 605, 638, 671, 679-680, 741
  size of 80, 162
int type 75, 157
__int64 37-38, 735
_INTEGRAL_MAX_BITS 26
Intel OMF 263
__interrupt 31-32
interrupt routine 31
intrinsic pragma 107, 193
invoke 418
invoking Open Watcom C/C++ 9
invoking Open Watcom Linker 264, 359

K

keywords
  __based 30
  __cdecl 30
  __declspec 32, 44
  __export 32
  __far 29
  __far16 35
  __fortran 31
  __huge 29
  __int64 26, 37
  __interrupt 31
  __loadds 33
  __near 29
  _Packed 30
  __pascal 31
  __pragma 37
  __saveregs 33
  _Seg16 36
  __segment 30
  __segname 30
  __self 30
  __stdcall 33
  __syscall 34

L

L 558
LANGUAGE directive 307
LANGUAGE options
  CHINESE 307
  JAPANESE 307
  KOREAN 307
large memory model 69, 150
large model
  libraries 58, 61
LBC command file 394
_leave 491
LIB environment variable 313, 324, 327, 456
LIBDIR environment variable 269
LIBFILE directive 308
LIBPATH directive 310
libraries 57
  80x87 math 61
  alternate math 61
  class 59
  location 57
  math 60
library
  import 396
LIBRARY directive 312
library file
  adding to a 390

deleting from a 390
extracting from a 392
replacing a module in a 391
library manager 387
library path 457
LINEARRELOCS option 315
linker symbols
_edata 288
_end 288
linking notation 270
__LINUX__ 24-25
__loadds 33
loadds (pragma) 121, 208
loading DS before calling a function 121, 208
loading DS in prologue sequence of a function 121, 208
__LOCAL_SIZE 248
long 480
long double
size of 80, 162
long float
size of 80, 162
long int
size of 80, 162
long int type 74, 156
LONGLIVED option 316
low 418
lowword 418
lroffset 418

M

_M_I386 23
_M_I86 23
_M_IX86 23-24
macros
__386__ 23
__CHAR_SIGNED__ 25
__COMPACT__ 25
__cplusplus 25
__CPPRTTI 26
__CPPUNWIND 26
_DOS 24
__DOS__ 24
__FLAT__ 25
__FPI__ 25
__HUGE__ 25
__I86__ 23
__INLINE_FUNCTIONS__ 25
_INTEGRAL_MAX_BITS 26
__LARGE__ 25
__LINUX__ 24
_M_386CM 25
_M_386FM 25
_M_386LM 25
_M_386MM 25
_M_386SM 25
_M_I386 23
_M_I86 23
_M_I86CM 25
_M_I86HM 25
_M_I86LM 25
_M_I86MM 25
_M_I86SM 25
_M_IX86 23
__MEDIUM__ 25
MSDOS 24
__NETWARE_386__ 24
__NETWARE__ 24
NO_EXT_KEYS 25
__NT__ 24
__OS2__ 24
_PUSHPOP_SUPPORTED 26
__QNX__ 24
__SMALL__ 25
_STDCALL_SUPPORTED 26
__UNIX__ 24
__WATCOM_CPLUSPLUS__ 26
__WATCOMC__ 25
_WINDOWS 24
__WINDOWS_386__ 24
__WINDOWS__ 24
__X86__ 23
mangled names in C++ 317, 343

MANGLEDNAMES option 317
map file 318
MAP option 318
mask 418
math coprocessor 61-62
math387r.lib 61
math387s.lib 61
math3r.lib 62
math3s.lib 62
math87c.lib 61
math87h.lib 61
math87l.lib 61
math87m.lib 61
math87s.lib 61
mathc.lib 61
matherr 754
  DOMAIN 754
  OVERFLOW 754
  PLOSS 754
  SING 754
  TLOSS 754
  UNDERFLOW 755
mathh.lib 61
mathl.lib 61
mathm.lib 61
maths.lib 61
MAXERRORS option 319
mdef.inc 62-63
medium memory model 69, 150
medium model
  libraries 57, 61
memory
  first megabyte 242
memory layout 70, 77, 152, 159, 287, 361
memory model 12
memory models
  16-bit 67
  32-bit 149
  compact 69, 150
  flat 150-151
  huge 69
  large 69, 150
  libraries 69, 152
  medium 69, 150
  mixed 69, 151
  small 69, 150
  tiny 69
message 696
  1014 365
  1019 366
  1023 367
  1027 368
  1028,2028 368
  1032 369
  1038 369
  1043 370
  1044,2044 370
  1045 370
  1046 370
  1047 370
  1048 370
  1050 371
  1054 371
  1058 372
  1059,2059 372
  1060 372
  1061 372
  1062 372
  1069 373
  1072 374
  1076 374
  1080 374
  1087 375
  1090 375
  1098 376
  1101 376
  1102 376
  1103 377
  1105 377
  1107 377
  1108 377
  1109 377
  1110 377
  1111 377
  1115 378
  1116 378
  1117 378
  1118 378

1121 378
1124 379
1125 379
1126 379
1130 379
1133 380
1134 380
1136 380
1140 381
1141 381
1143 381
1145 381
1148 381
1149 382
1150 382
1158 382
1162 383
1163 383
1165 383
1167 383
2002 364
2008 364
2010,3010 365
2011 365
2012 365
2015 365
2016 366
2017 366
2018 366
2020 366
2021 367
2022 367
2024 367
2025 367
2026 368
2029 368
2030 368
2031 369
2033,3033 369
2034 369
2039 369
2040 370
2041 370
2042 370
2049 371
2051 371
2052 371
2053 371
2055 371
2056 371
2063 372
2064 372
2065 373
2066 373
2067 373
2068 373
2070 373
2071 373
2073 374
2074 374
2075 374
2082 374
2083 374
2084 375
2086 375
2089 375
2091 375
2092 376
2093 376
2094 376
2099 376
2119 378
2120 378
2127 379
2132 380
2146 381
2151 382
2152 382
2154 382
2155 382
2156 382
2166 383
3009 364
3013 365
3057 372
3088 375
3097 376
3114 377

- 3122 378
- 3123 379
- 3128 379
- 3129 379
- 3131 380
- 3135 380
- 3137 380
- 3138 380
- 3139 381
- 3147 381
- 3157 382
- 3159 383
- 3160 383
- 3164 383

message pragma 108, 194
messages
- errno 748
- matherr 754
- run-time 748, 754

MetaWare
- High C calling convention 201, 227

Microsoft
- C calling convention 115, 140

Microsoft OMF 263
mixed memory model 69, 151
models.inc 62
MODFILE directive 320
modify exact (pragma) 139-140, 226-227
modify nomemory (pragma) 134, 138, 221, 224
modify reg_set (pragma) 146, 233
MODTRACE directive 321
MSDOS 24
mutable 693

N

naked 32, 46
NAME directive 322
NAMELEN option 323
namespace 566, 723-725
near 29, 32, 69, 151, 503, 655
near (pragma) 118, 206
near call 67, 149
near pointer
- size of 80, 162

NetWare debugger 282
__NETWARE_386__ 24
__NETWARE__ 24
new 538, 552, 558, 576, 618, 647, 650, 656, 724
no8087 (pragma) 129, 216
NO_EXT_KEYS 25
NODEFAULTLIBS option 324
noemu387.lib 60
noemu87.lib 60
NOREDEFSOK option 340
notation 270
NOUNDEFSOK option 355
__NT__ 24
NULL 39
_NULLOFF 39
_NULLSEG 39
numeric data processor 61-62

O

object model 101, 187
OCC directory 11
occ file extension 11
offsetof 544, 549, 614
OMF 263
OMF library 263
once pragma 108, 194
opattr 418
opcodes
- assembly language 250

Open Watcom C/C++ options
- zm 289

operator 564
- :> 41

operator + 567, 576

operator ++ 578
operator += 576
operator -> 579, 698
operator delete 577-578, 612, 637, 724
operator delete [] 577-578
operator new 558, 560, 562, 576-578, 724
operator new [] 576-578
operator ~ 575
optimization 108, 194
option 418
OPTION directive 325
options 7
 ARTIFICIAL 273
 bt 19
 CACHE 274
 CASEEXACT 275
 check_stack 94, 180
 CVPACK 277
 DOSSEG 287
 ELIMINATE 289
 FARCALLS 291
 FILLCHAR 294
 fp2 61
 fp3 61
 fp5 61
 fpc 61, 175
 fpi 61
 fpi87 61
 i 19, 21
 LINEARRELOCS 315
 LONGLIVED 316
 MANGLEDNAMES 317
 MAP 318
 MAXERRORS 319
 NAMELEN 323
 NODEFAULTLIBS 324
 NOREDEFSOK 340
 NOUNDEFSOK 355
 OSNAME 332
 PRIVILEGE 338
 QUIET 339
 r 85, 91, 167, 172, 178
 REDEFSOK 340
 reuse_duplicate_strings 95, 181
 SHOWDEAD 342
 STACK 344
 START 345
 STATICS 347
 SYMFILE 348
 UNDEFSOK 355
 unreferenced 94, 180
 using pragmas 94, 180
 VERBOSE 356
 VFREMOVAL 357
options file
 command line options 11
OPTLIB directive 326
ORDER directive 328
__OS2__ 24
OS2_INCLUDE environment variable 20
OSNAME option 332
OUTPUT directive 333
overview of contents 3

P

pack pragma 109, 195
_Packed 30
page 418
parm (pragma) 123, 210
parm caller (pragma) 128, 215
parm nomemory (pragma) 138, 224
parm reg_set (pragma) 142, 229
parm reverse (pragma) 128, 215
parm routine (pragma) 128, 215
pascal 31-32, 116, 200, 202
pascal alias name 116, 202
passing arguments 79, 161
 1 byte 79, 161
 2 bytes 79, 161
 4 bytes 161
 8 bytes 80, 162
 far pointers 80, 162
 in 80x87 registers 142, 229

in 80x87-based applications 90, 176
in registers 79, 161
of type double 80, 162
PATH directive 336
PATH environment variable 20, 266, 269, 275, 291, 353, 456
PE format executable 297
Phar Lap example 241
Phar Lap OMF-386 263
Phar Lap TNT 297
PL format executable 297
plib3r.lib 59
plib3s.lib 59
plibc.lib 59
plibh.lib 59
plibl.lib 59
plibm.lib 59
plibs.lib 59
popcontext 418
pragma 32, 37, 48, 93, 179, 647, 655, 670
pragma options 94, 180
__pragma("string") 32
pragmas
= const 118, 206
aborts 134, 220
alias name 114, 201
alloc_text 97, 183
alternate name 117, 205
auxiliary 112, 198
calling information 118, 206
code_seg 98, 184
comment 99, 185
data_seg 99, 185
describing argument lists 123, 209
describing return value 129, 216
disable_message 100, 186
dump_object_model 101, 187
enable_message 101, 187
enum 102, 188
error 103, 189
export 122, 209
extref 103, 189
far 118, 206
frame 123, 209
function 104, 190
in-line assembly 118, 206
in-line functions 127, 214
initialize 105, 191
inline_depth 106, 192
inline_recursion 107, 193
intrinsic 107, 193
loadds 121, 208
message 108, 194
modify exact 139-140, 226-227
modify nomemory 134, 138, 221, 224
modify reg_set 146, 233
near 118, 206
no8087 129, 216
notation used to describe 93, 179
once 108, 194
pack 109, 195
parm 123, 210
parm caller 128, 215
parm nomemory 138, 224
parm reg_set 142, 229
parm reverse 128, 215
parm routine 128, 215
read_only_file 110, 196
specifying default libraries 96, 182
struct caller 129, 131, 216, 218
struct float 129, 132, 216, 219
struct routine 129, 131, 216, 218
template_depth 111, 197
value 129-132, 216, 218-219
value [8087] 133, 220
value no8087 133, 220
value reg_set 145, 232
warning 112, 198
precompiled headers 53
compiler options 54
rules 55
uses 53
using 54
predefined types
size of 80, 162
predictable code size 50, 459
preprocessor 21
printf 38

private 550, 571, 589
PRIVILEGE option 338
protected 537, 539, 589
proto 418
public 550
punctuation characters 270
purge 418
pushcontext 418
_PUSHPOP_SUPPORTED 26

Q

QNX applications
  creating 359
__QNX__ 24
QUIET option 339

R

read_only_file pragma 110, 196
real-mode memory 242
record 418
REDEFSOK option 340
REFERENCE directive 341
register 476, 480-481, 487, 489, 519, 531, 545, 547
reinterpret_cast 709-710
removing debug information 449
return 465, 479, 485, 503, 505, 517
return value (pragma) 129, 216
returning values from functions 86, 172
reuse_duplicate_strings option 95, 181
run-time
  error messages 464, 502, 747-748
  messages 747
run-time initialization 62

S

__saveregs 33
_Seg16 36
segment 30, 39, 41-42
  _TEXT 71, 78, 153, 160
segment ordering 70, 77, 152, 159, 287, 361
segment references 30
__segname 30, 39, 494, 735
segname references 30
__self 30, 39, 594
self references 30
shared library 58
short 478, 480, 515
short int
  size of 80, 162
short int type 74, 156
SHOWDEAD option 342
side effects of functions 134, 221
signed 478, 480, 515
signed char 700, 741
  size of 80, 162
signed int
  size of 80, 162
signed long int
  size of 80, 162
signed short int
  size of 80, 162
size of
  char 80, 162
  double 80, 162
  enumerated types 81, 163
  far pointer 80, 162
  float 80, 162
  int 80, 162
  long double 80, 162
  long float 80, 162
  long int 80, 162
  near pointer 80, 162
  predefined types 80, 162

short int 80, 162
signed char 80, 162
signed int 80, 162
signed long int 80, 162
signed short int 80, 162
unsigned char 80, 162
unsigned int 80, 162
unsigned long int 80, 162
unsigned short int 80, 162
sizeof 46, 487
small code model 67, 149
small data model 68, 150
small memory model 69, 150
small model
libraries 57-58, 61-62, 152
software quality assurance 50, 459
SOMDLINK 29, 35
SOMLINK 31, 35
SORT directive 343
source file
including 18
searching 19
space character 270
special characters 270
stack frame 123, 209
stack frame (pragma) 123, 209
STACK option 344
stack-based calling convention 168
80x87 considerations 176
returning values from functions 175
stacking arguments 126, 213
START option 345
STARTLINK directive 346
static 44, 476-477, 482, 512, 514, 531, 538, 548, 558, 589, 593, 595, 598, 611
static_cast 712, 715, 731
STATICS option 347
stdcall 32-33, 116, 202
stdcall alias name 116, 202
_STDCALL_SUPPORTED 26
strip utility 449
struct 30, 475, 477-478, 485-488, 490, 496, 513, 524, 550, 605
struct caller (pragma) 129, 131, 216, 218
struct float (pragma) 129, 132, 216, 219
struct routine (pragma) 129, 131, 216, 218
subtitle 418
subttl 418
support files
dbg 460
hlp 460
prs 460
sym 460
trp 460
switch 467, 473-474, 478, 486, 507, 509, 514, 609
symbol attributes 112, 198
symbol file 348
symbolic references in in-line code sequences 120, 207
SYMFILE option 348
SYMTRACE directive 350
__syscall 32, 34-35, 50, 202
syscall alias name 202
system 35, 202
system alias name 202
SYSTEM directive 265, 351
system name 351

T

template_depth pragma 111, 197
_TEXT segment 71, 78, 153, 160
this 418, 554, 563-564, 611, 621-622, 634, 641, 671
thread 32, 44-45
throw 518, 556, 655, 667, 687-689, 723
tiny memory model 69
title 418
TMPDIR environment variable 457
TNT DOS extender 297
try 491, 663, 666-667
typedef 418, 531, 533, 548, 566, 595
typeid 735

types
  char 74, 156
  double 76, 158
  float 75, 157
  int 75, 157
  long int 74, 156
  short int 74, 156

U

UNDEFSOK option 355
union 418, 475, 477-478, 485-488, 490, 496, 513, 524, 529-530, 605
__UNIX__ 24-25
unreferenced option 94, 180
unsigned 478, 480, 515, 524
unsigned char 700, 741
  size of 80, 162
unsigned int
  size of 80, 162
unsigned long int
  size of 80, 162
unsigned short int
  size of 80, 162
USE16 segments 152, 159
user initialization file 12
using 732-734
using environment variables in directives 269
using namespace 727

V

va_arg 496
value (pragma) 129-132, 216, 218-219
value [8087] (pragma) 133, 220
value no8087 (pragma) 133, 220
value reg_set (pragma) 145, 232
variable argument lists 86, 172
VERBOSE option 356
VFREMOVAL option 357
virtual 537, 611-612, 668
virtual functions 342, 357
void 17, 465, 476, 479, 503, 517, 551-552, 557, 559-560, 563, 577, 582, 618, 628, 650, 676, 697, 715
volatile 480, 537-538, 583, 619, 621-622, 632, 668, 708-709
VxD format executable 297

W

warning messages 463
warning pragma 112, 198
warnings 285, 363
wasm
  command line format 407
watcall 116, 202
watcall alias name 116, 202
WATCOM environment variable 60, 267, 269, 275, 291, 353, 456-457
__WATCOM_CPLUSPLUS__ 26
__WATCOMC__ 25
wcc 458
WCC environment variable 11, 458
WCC options
  nm 71, 79, 153, 161
  nt 71, 79, 154, 161
wcc386 458
WCC386 environment variable 11, 458
WCC386 options
  nm 71, 79, 153, 161
  nt 71, 79, 154, 161
WCGMEMORY environment variable 50-51, 459
WD environment variable 459-460
WD_PATH environment variable 460

wdis
  command line format 433
wdis example 439
wdis options 434
  a 434
  e 435
  fi 436
  fp 436
  fr 436
  fu 436
  i 434
  l (lowercase L) 437
  m 438
  p 437
  s 438
while 473, 484, 486, 507, 522
width 418
window function 305
_WINDOWS 24
__WINDOWS_386__ 24-25
__WINDOWS__ 24
wlib
  command file 394
  command line format 388
  operations 389
wlib options 394
  b 394
  c 395
  d 395
  f 395
  i 396
  l (lower case L) 397
  m 397
  n 397
  o 398
  p 398
  q 399
  s 399
  t 399
  v 399
  x 400
wlink
  command line format 264, 359
wlink command line
  invoking wlink 264, 359
wlink notation 270
wlink.lnk
  default directive file 266, 269, 275, 291, 353
wlsystem.lnk
  directive file 267, 269, 275, 291, 353
wpp 461
WPP environment variable 11, 460-461
WPP options
  nm 71, 79, 153, 161
  nt 71, 79, 154, 161
wpp386 461
WPP386 environment variable 11, 461
WPP386 options
  nm 71, 79, 153, 161
  nt 71, 79, 154, 161
wstrip 282, 284, 449
  command line format 449
  diagnostics 451
wstrip command 284

X

__X86__ 23

Z

zm compiler option (Open Watcom C/C++) 289