

VisualAge C++ Professional for OS/2 and for Windows NT

Build
Concepts and Tasks

Version 4.0



VisualAge C++ Professional for OS/2 and for Windows NT

Build
Concepts and Tasks

Version 4.0

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page v.

First Edition (November 1998)

This edition applies to Version 4.0 of the VisualAge C++ Professional product, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term ″VisualAge″ as used in this publication, refers to the VisualAge C++ product set.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada, M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers’ Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices v
Programming Interface Information . . . v
Trademarks and Service Marks vi
Industry Standards vii

Chapter 1. Introduction to Building Applications Using VisualAge C++ . . . 1
C Compilation: Comparing C and C++ Builds 1
Codestore 2
Build Extensions and Extension Specification Files 4
C++ Source Files and Source Regions . . . 5
Macros in C++ Source Files 6
The One-Definition Rule (C++) 9
Decorated and Mangled Names 11
When to Use Makefiles. 11
Source Files that Contain Embedded SQL 13
Cleaning Up After Builds 13
Inlining 14
Precompiled C Headers 16
Organizing Your C Source for Precompiled Headers. 18
Linker Searches 20
Search Paths for Primary Source Files . . . 21
How Configuration Files Are Processed 22
Conditional Processing in Builds 23
The Configuration Optimizer (C++) . . . 24
Convert Included Sources to Primary Sources (C++). 25
Improve Build Times 27

Chapter 2. Build. 29
Set Build Options 29
Set Build Options for Source Files or Targets 30
Set Build Options for the Project 30
Add or Remove an Option Directive . . . 31
Change an Option Directive 31
Add Source Files to a Project. 32
Remove Source Files from a Project . . . 34
Add a Source Directive. 34
Add a Group Directive. 35
Remove a Source or Group Directive . . . 36
Change a Group Directive's Name 37
Change a Source Directive's Type 37
Change a Source or Group Directive's Source Files 37
Move a Source File 38
Override the Default Type of a Source File 39
Use Resource Source Files as Sources for a Build. 39
Compile and Bind Resources. 40
Compile IPF Help During a Build 41
Use Makefiles as Sources for a Build . . . 42
Build with Existing Object Files and Libraries 44
Add Part Source to the Configuration File (AIX). 45
Precompile Embedded SQL During a Build 46
Add or Change Targets Overview 47
Add or Remove a Target 48
Change a Target's Source Files 49
Build Executable Programs 50
Produce Object Files from a Build 50
Produce Multiple Targets from One Build 51
View Build Messages 52
Control Build Messages 53
Clean Up the Build Environment 53
Build in a Team Environment 53
Build from the VisualAge C++ Command Line Window 55
Redirect Standard Streams 55
Run Command Line Programs During Builds 56
Nominate Inline Functions 57
Check Out, Check In or Extract a Source File 59
Get Mangled Names for C++ Symbols . . 59

© Copyright IBM Corp. 1998

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

IBM may change this publication, the product described herein, or both.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

© Copyright IBM Corp. 1998

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

DATABASE 2
DB2
IBM
Operating System/2
OS/2
VisualAge

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation.

Pentium as a trademark or registered trademark of Intel Corporation in the U.S. and other countries.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

VisualAge C++ supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899-1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C(X3.159-1989) and is technically equivalent to the ANSI** C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ - ISO/IEC 14882:1998.
- A subset of the ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard.
- A subset of the X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

Chapter 1. Introduction to Building Applications Using VisualAge C++

There are several concepts you should understand about how VisualAge C++ Professional Version 4.0 builds your code. This document illustrates the power of the configuration file and outlines some of the thing you need to do to take best advantage of that power.

C Compilation: Comparing C and C++ Builds

VisualAge C++ supports incremental build for C++ code, and provides an IDE that you can use to develop C and C++ code. If you are programming in C, you should be aware of the following differences between the way VisualAge C++ handles C and C++ source code.

C Compilation and Incremental C++ Builds
The C compiler takes C source files as input and, if compilation is successful, creates object files. Like most compilers, it processes source files one at a time and creates an object file from scratch for every source that it compiles.

An incremental C++ build uses a program database called the codestore to record information from your C++ source files and configuration file. If you modify your source files, subsequent builds recompile and relink only the portions of code affected by the changes. Therefore, incremental C++ builds can rebuild much faster than traditional compile and link.

The C compiler is not incremental. A VisualAge C++ build from C source files records the include hierarchy and information from the configuration file in the codestore. It stores no other information about your C source code in the codestore. On subsequent builds, an entire C source file is recompiled if it or a file that it includes has a later timestamp than on the previous build.

You cannot use incremental compilation for C source files or incremental linking on code generated from C source code. However, you can use a number of features that are not available with incremental build. For example, the following actions are possible only for C source files:

- Run just the preprocessor.
- Generate object files automatically.
- Output a variety of listing files.
- Use precompiled headers to speed recompilation in cases where the headers do not change between builds.

© Copyright IBM Corp. 1998

Order of Declarations in C and C++ Source

You must include header files in every C source file that requires them, and ensure that functions, variables, and the like are declared before they are used. VisualAge C++ frees the C++ programmer from this constraint, by performing orderless compilation. VisualAge C++ processes all C++ source files together and searches in the codestore for declarations when they are required.

Mixing C and C++ Source

In your configuration file, you can explicitly specify whether each source file contains C or C++ source code, or accept the default type set by the file extension. VisualAge C++ runs the C compiler for C source files or incorporates code for C++ source files into the codestore. If you have both C and C++ source files in one build, the linker links the object files compiled from C source files with the code in the codestore generated from C++ source files.

Using the IDE

You can use the VisualAge C++ IDE to develop C code and C++ code. You can enter and edit source files for either language with the Editor, in a **Source** view. You can compile and link by clicking the **Build** button. Use the **Project SmartGuide** to create your configuration file and the **Configuration** section of the IDE to modify it.

Not all of the IDE works the same for C and C++. For example, the browsing and debugging features use the representation of C++ code that is stored in the codestore and do not apply to C code.

For both C and C++, you see compile and link messages in a Message view of the IDE. Messages are sent to the console when you build with the **vacbld** command.

Debugging C Code

To debug C code, use a traditional debugger such as the debuggers supplied with earlier versions of VisualAge C++ and predecessor products. Build with options gen(debug) or gen(lineOnlyDebug) to include debugger information in your object and executable files.

Codestore

For incremental C++ build to work, VisualAge C++ must maintain a database that stores information about each build. This program database is called the *codestore*. The codestore is a database that is created or updated during builds. A build synchronizes the codestore with the latest version of source code.

A project has one codestore and at least one configuration file. A codestore can be produced with more than one configuration file only when one master configuration file includes other configuration files with include directives. The codestore file has the same name as the configuration file, or master configuration file, and the file extension .ics. VisualAge C++ records information from the configuration file and from C++ source files in the codestore. On all but an initial build, VisualAge C++ also uses data in the codestore to determine what has changed from one build to the next.

C++ The codestore holds a representation of C++ code that provides rapid access for services such as code generation, debugging, and browsing. Several components of VisualAge C++ use the codestore. The IDE accesses information in the codestore to let you browse code. You do not have to perform a special build or set any build options in your configuration file to browse or debug C++ code. All information required by the C++ debugger is in the codestore, and not in the target file produced by the build. Therefore, you can produce an application that you can ship and code that you can debug from one build. This is a great advantage over traditional compilers.

C++ The *one-definition rule* applies to a codestore. This rule limits when you can redefine variables, functions, class types, enumeration types, and templates.

Every time you build in the IDE or run the**vacbld** command, you create or update a transient or persistent codestore. A transient codestore is a temporary codestore that resides in transient memory. It is used during one run of vacbld or one IDE session and is erased when you close the IDE or vacbld ends. A persistent codestore is a codestore that is saved in a file. It remains after you close your IDE session or **vacbld** ends, and is available for future builds. By default, a persistent codestore is created at the end of an IDE session during which at least one build was performed and by the **vacbld** command.

To increase the speed of builds, the IDE can use both transient and persistent codestores. By default, the IDE uses a transient codestore and creates or updates the persistent codestore at regular intervals.

When you build by selecting **Build in Migrate Mode** from the **Project Workbook** menu in the IDE or by running **vacbld** with the -migrationbuild or -noc options, a transient codestore is used during the build and no persistent codestore file created.

The build process does not modify your source files. You can delete the codestore file at any time. The best way to remove an existing codestore is to select **Clean** from the **Project Workbook** menu of the IDE, because Clean

performs additional cleanup. The next time you build, VisualAge C++ performs an initial build and creates a new codestore.

Build Extensions and Extension Specification Files

Build extensions are programs that add capabilities to the VisualAge C++ build process. These programs often are utilities or executable programs that you also can run from the command line. When used as build extensions, they are started by the build process and are controlled by your configuration file.

The VisualAge C++ kernel performs incremental C++ builds, and includes many of the processes that operate on the codestore. Build extensions are integrated with the build process so that they appear to work like parts of the kernel. All build extensions access the codestore, but many do not use the representation of C++ code that is stored in the codestore.

Do not use the run directive to run build extensions. You control most extensions with source, target, and option directives. Use the tool directive for extensions that are not associated with particular source file types. The description of each build extension that is supplied with VisualAge C++ tells you how to use the program.

Build extensions may produce intermediate or temporary files that remain after the build is complete. Usually, they do no harm and you can ignore them. The best way to clean up your build environment, including deleting the codestore, is to select **Clean** from the **Project Workbook** menu of the IDE.

Extension Specification (.ice) files
VisualAge C++ knows about build extensions because they are registered in extension specification files. These files have the file extension .ice, which stands for 'incremental compiler extension'. They have a format similar to Windows .ini files. They contain internal information required to load and run the program.

A number of extensions are supplied with VisualAge C++ and some extension specification files are installed with the product. Do not modify the .ice files supplied with VisualAge C++.

C++ Source Files and Source Regions

VisualAge C++ uses the four terms listed below to describe user code that is processed in a C++ incremental build. The first three terms describe C++ source files. The last term relates to preprocessed C++ source code that is stored in the codestore.

- Primary Sources
- Included Sources
- Macro Sources
- Source Regions

Primary Sources Primary sources are source files listed in the configuration file. They can be ordinary source files or header files. During a build, these files are subject to orderless compile.

A primary source file is not the same as a translation unit. In a traditional compiler, each source file is a translation unit, but VisualAge C++ builds from all sources at once. As a result, the one-definition rule of the C++ standard applies to the entire set of sources for a build.

VisualAge C++ applies the conventional rules of internal and external linkage to primary sources. Variables and functions declared with internal linkage are visible only within the primary source file in which they are declared. Variables and functions declared with external linkage are visible to all source files.

Included Sources
Included sources are source files not listed in the configuration file. They are compiled because they are included, either directly or indirectly, in primary files with #include preprocessor directives.

A C++ source file can be named in an #include directive and also be listed in the configuration file. When that happens, the file is treated as a primary source and the #include directive is ignored. You can leave the #include directive in place so that your source files can also be compiled by traditional C++ compilers.

Included sources are treated as if their contents appeared at the inclusion point. For example, the file boundaries have no effect on source region boundaries.

For faster builds and to simplify your source file organization, use primary sources rather than included sources as much as possible. You should convert included sources that are semantically the same in all places where they are included to primary or macro sources.

Macro Sources
Macro sources are primary sources to which the build option macros(global) applies. During a build, they are preprocessed before other sources, and in the order of their appearance in the configuration file. For purposes other than controlling the visibility for macros, macro sources are identical to primary sources.

Use macro sources for files that contain macro definitions that must be visible in other source files. The scope of macros defined in macro sources includes all macro sources listed subsequently in the configuration file and all sources that are not macro sources. If a subsequent macro source undefines a macro with a #undef directive, the scope of that macro ends.

Macros defined in primary sources that are not macro sources are local to the file in which they are defined.

Source Regions
Source regions are not source files. Source regions are stored in the codestore and contain preprocessed tokens. They are the smallest units of preprocessed source that can be recompiled and relinked during an incremental build. Usually a source region contains a representation of a much smaller section of code than a source file. For example, each high level declaration in your code is represented in the codestore in a separate source region. If you change a declaration between builds, only the regions that represent the declaration and regions that depend upon it are rebuilt.

Source regions are considered to be unordered with respect to other source regions. This means that the source code from which one source region is generated can refer to names declared in source code from which other source regions are generated, regardless of the order in which the source files are processed. Within a source region, declarations and statements are ordered using the normal C++ rules of ordering. This means that within a class declaration or within a function body, names must be declared before they are used.

Macros in C++ Source Files

You can include preprocessor macros in C++ source files, exactly like you do in source files that you prepare for conventional C++ compilers.

Unlike conventional compilers, VisualAge C++ performs an orderless compile and you can specify many source files in any order. Functions, classes, and variables do not have to be declared before they are referenced, because the build process finds the declarations as it needs them. However, preprocessor macros do have intrinsic position dependencies. The order in which they are

defined, undefined, or redefined affects your code. For example, constructs such as the following are common, especially in header files:

```
#ifndef MY_GUARD
#define MY_GUARD 1
...
#endif
```

Because of the ordering requirements of macros, changes to macros cause more recompilation than changes to C++ code.

How VisualAge C++ treats a macro depends on whether the source file in which it is defined is a macro source. A macro source is a C++ source file that is listed in the configuration file and to which the option macros(global, yes) applies.

If a macro is defined in a source that is not a macro source, it is local to the source file in which it is defined. The macro is visible only to code that appears after the macro definition in the same file. The macro definition is not visible from other source files.

Macros in macro sources are global. They are visible from the point of definition to the end of the file and continue to be visible in all macro source files that are listed subsequently in your configuration file. Global macros are visible to all sources that are not macro sources.

When you convert a header file to a primary source file, you should also make it a macro source if the header defines macros that are used by other source files. Apply macros(global) to the header file, so that macros defined in it are visible in other sources that previously included the header file.

You should specify source files that are not header files as primary sources and either accept the default setting of macros(global, no) or explicitly apply macros(global, no) to them. Do this to give code in these files the same behavior as in regular C++ compilers.

The only difference between macro sources and other primary sources is in the way macros are preprocessed. The order of macro sources in the configuration file controls the scope of macros defined within them. For processing C++ declarations, macro source files are treated exactly like other source files.

When you optimize your configuration file using the configuration optimizer, compile errors may be reported. This may be due to changes the configuration optimizer makes to your configuration file. Part of the configuration optimizations made by this tool involve automatically promoting system run-time header files that are #included in your application.

In some cases this may cause incorporation errors if these runtime header files contain a reference to a macro name that is defined in your application. There are two ways you can resolve this problem:

- Edit the new configuration file and manually promote your application header file to precede the system header file.
- Edit the new configuration file and enclose the promoted system header file with the necessary options to define the macros it requires. We recommend you use the latter approach.

Here is an example of the problem, followed by the two possible solutions:

Problem

t.cpp

```
#include 't.h'
#include <windows.h>
```

t.h

```
#define _WIN32_WINNT 0x0400
```

t.icc (config optimizer generated)

```
target 't.exe'
{
  option macros(global,yes)
  {
    source type(cpp) 'windows.h'
  }
  source type(cpp) 't.cpp'
}
```

Solution 1

t.icc

```
target 't.exe'
{
  option macros(global,yes)
  {
    source type(cpp) 't.h'
    source type(cpp) 'windows.h'
  }
  source type(cpp) 't.cpp'
}
```

Solution 2 (recommended)

t.icc

```
target 't.exe'
{
  option macros(global,yes)
  {
    option define('_WIN32_WINNT','0x0400')
    {
      source type(cpp) 'windows.h'
    }
  }
  source type(cpp) 't.cpp'
}
```

The following build options can affect how macros are processed:

Option	Definition
macros(global[, yes]) macros(global, no)	Determines whether source files are macro sources. By default, source files are not macro sources.
macros(redefine[, yes]) macros(redefine, no)	Controls whether macros can be redefined without using the #undef preprocessor directive first.

Option	Definition
define(*macro_name*[, *string*])	Defines macros that can be used in source files to which this option applies.
undefine(*macro_name*)	Undefines a macro in the of source files to which this option applies.

The One-Definition Rule (C++)

The one-definition rule is specified in the C++ standard. The one-definition rule has two aspects:

1. A translation unit must not contain more than one definition of any variable, function, class type, enumeration type, or template.
2. External definitions must not be redefined within a program.

Traditional C++ compilers enforce only the first part of the one-definition rule. VisualAge C++ enforces the full rule, to conform to the C++ standard and to remove the possibility of two or more conflicting definitions being visible at the same time during a build. As implemented in VisualAge C++, the one-definition rule means that declarations with external linkage cannot be defined in more than one source file. The one-definition rule allows some declarations, such as class and enum definitions, templates, and inline functions, to be defined in different source files as long as they consist of the same sequence of tokens. VisualAge C++ accepts these multiple definitions.

Because VisualAge C++ performs C++ incremental builds, in which all source files are considered at once, the one-definition rule applies to the set of C++ source files that are input to one build. As a result, VisualAge C++ may not accept code that other compilers accept in some circumstances. The circumstances relate to following constructs and are described in detail below:

- Classes, enums, Templates, and Inline Functions
- Enumerations and extern Variables
- typedef Statements

If your C++ sources are rejected by VisualAge C++ due to the one-definition rule, you should consider using one of the following methods to correct the problem:

1. Change the name of one of the conflicting definitions.
2. Enclose the code that depends on each of the conflicting definitions in different namespaces.
3. Split your source files so that each of the conflicting definitions is in a separate codestore. Do this by using a separate project and creating a

separate configuration file for each codestore. For example, you can build DLLs and your executable program from different configuration files.

If you specify the same source file more than once in your configuration file, VisualAge C++ treats each occurrence of the file as a separate and unrelated source file. The one-definition rule limits the way you can reuse C++ source files in one codestore.

Classes, enums, Templates, and Inline Functions
The C++ language does not allow classes, enums, templates, and inline functions to be defined more than once in a program, unless all three of the following conditions are satisfied:

- The definitions appear in different source files.
- The definitions consist of the same sequence of identical tokens.
- The definitions are semantically identical.

Traditional compilers test for the first condition. For example, they make sure that classes and enums are defined only once within a source file. VisualAge C++ tests for the first two conditions and lets you redefine classes and enums only with definitions that are identical to the original definitions. Therefore, if a class or enum name is used with different definitions in different source files, VisualAge C++ reports an error that traditional compilers do not find. While traditional compilers only check this rule within a source file, VisualAge C++ checks it across all source files.

Enumerators and extern Variables
The C++ language allows only one definition of a given enumerator or variable name with external linkage in namespace scope, which includes global scope, and insists that multiple declarations of an external variable match. While traditional compilers only check this rule within a source file, VisualAge C++ checks it across all source files.

typedef Statements
Standard C++ does not allow definitions with external linkage to be defined differently in different translation units. VisualAge C++ applies this rule to types defined in typedef statements, which have no linkage according to standard C++. Unlike traditional compilers, VisualAge C++ does not let you redefine a type with typedef statements in more than one source file unless the typedef statements are identical.

If VisualAge C++ rejects a typedef statement in your C++ source because of the one-definition rule, you have an additional recovery technique to the three methods listed above. You can tell VisualAge C++ to treat all typedef statements as local to the source file in which the type is defined, by building with option lang(localTypedefs, yes).

Decorated and Mangled Names

The internal representation of function names and other other symbols in your code may not be the same as the identifier in your source code. The internal names are based on your identifier, but may be *decorated* and *mangled*.

Decorated Names

OS/2 WIN The internal names may be decorated on OS/2 and Windows NT platforms. The decoration is a leading character that represents the linkage convention of the function. A question mark indicates __cdecl linkage, and an underscore indicates __stdcall linkage.

WIN On Windows, the names of functions with __stdcall linkage are followed by a commercial at symbol, @, and a number. The number indicates how many bytes are required for parameters of the function.

Mangled Names

C++ When VisualAge C++ builds from C++ code, it mangles all C++ symbolic names that are not declared with extern 'C'. Mangling involves adding characters before and after the name to indicate type a number of arguments of a function and its and scope.The scope can indicate the class or namespace of the function. Mangled names are used to resolve references to functions when the function name has be overloaded.

OS/2 WIN If you specify symbols to export from a DLL, by using build options link(exportList, *fileName*) or link(export, *external_name*, ...), you must specify the internal representation of the name.

If you build with option link(map) the link map produced by the build shows symbols in their internal representation.

When to Use Makefiles

You never have to use a makefile when you develop an application with VisualAge C++. Nevertheless, the make utility is available to process makefiles. The main difference between using makefiles now and using makefiles in earlier versions of VisualAge C++ and predecessor products is that now you cannot issue commands to compile and link.

In VisualAge C++ configuration files perform much of the role played by makefiles in traditional C and C++ development environments. The only way to compile and link C code or perform an incremental C++ build is to process a configuration file. Makefiles cannot do this essential part of most builds. As a result, makefiles are no longer used to define dependencies on C and C++

source files or to ensure that C and C++ sources are recompiled if they have timestamps later than specified target files.

You no longer need makefiles. You may find that not only is it possible to do everything you require from configuration files, but also that using configuration files is simpler and less error-prone than maintaining makefiles. For example, the VisualAge C++ build process implicitly runs the processes required to build the targets you specify in your configuration file from sources you list in your configuration file.

There are two ways to combine processing makefiles with VisualAge C++ builds:

1. You can issue the **vacbld** command inside a makefile to perform a command line build that processes a configuration file.
2. You can specify your makefile as a source in your configuration file. The VisualAge C++ build process can run the make utility implicitly before the compile and link step.

You may want to use makefiles in the following circumstances:

- To set up your build environment, you can set environment variables and run programs from a makefile before processing your configuration file. Using a makefile to set environment variables for the build works only if you start the **vacbld** command from your makefile. When you process a makefile as a source in your configuration file, the environment variable settings made from the makefile are lost when you return to the main build process.
- If your build is complex and requires more than one configuration file, you can issue a sequence of **vacbld** commands from your makefile.
- If you must create directories or move files at specific times during the build process, or run tools that monitor the build, you can encode the necessary commands in a makefile.
- For builds that do not involve any C or C++ code, you can eliminate the overhead of loading the VisualAge C++ kernel by using the command line utilities. For example, you can run the resource compiler directly from the command line, from a makefile, or as part of a build.

If you use a configuration file for some steps in a build and a makefile for other steps, the program database for your build, the codestore, may not accurately reflect the state of your source files. This may cause problems for future builds.

OS/2 WIN For example, on OS/2 and Windows NT platforms you can run the library manager from the command line or from a makefile to update libraries. If the library was originally created by a build, the codestore is not updated to incorporate changes made to the library outside the build process.

If you have makefiles from earlier versions of VisualAge C++ and predecessor products, you should use the **make2cfg** utility to convert them into configuration files. To manually convert involves more steps than deleting the compile and link commands.

Source Files that Contain Embedded SQL

Your C and C++ source files can have SQL statements embedded within the source code.

If you are using Data Access Builder, you never have to deal with embedded SQL directly. Data Access Builder generates source code that uses DB2 embedded SQL or ODBC drivers to access databases. Data Access Builder can also create parts that you can use in visual programming. Data Access Builder updates your configuration file as required to automate the process of building an application that accesses databases, including the SQL precompilation step.You can greatly increase your productivity by using Data Access Builder rather than programming your database operations manually.

You can also use DB2 directly, if it is installed on your workstation. In this situation, you do write C or C++ code that contains embedded SQL, and must explicitly precompile that code as a step in your builds. VisualAge C++ provides a build extension that you can use to run the DB2 SQL precompile function easily from your configuration file. The DB2 SQL precompile function is a command line function to precompile C or C++ source files that contain SQL statements. You can use the build extension only only when DB2 is available.

The DB2 SQL precompile build extension produces C or C++ source files that you can use as ordinary C or C++ source files for your build.

Cleaning Up After Builds

A clean build is a build that starts with no input other than source files, including the configuration file. The first time you build a new project, VisualAge C++ performs a clean build. Cleaning up the build environment after builds involves removing all files produced by previous builds so that the next build is the equivalent of the first build for the project.

By default, the build process creates a codestore. If the build is successful, the target files specified in your configuration file are produced. Other intermediate or temporary files may also be produced by the build process, build extensions that you use, or by command line programs that you start with the run directive in your configuration file.

You can clean up the build environment after you have performed a number of builds. This is analogous to including a 'clean' directive block in makefiles in traditional build environments.You may want clean up the build environment for a number of reasons:

- To package the source files for archiving or distribution.
- During development, to eliminate any problems that you suspect may relate to old information in the codestore or intermediate files.
- To determine how long it takes to rebuild your project from scratch, using the final versions of the source files.

You can always delete the codestore file directly, but cleaning up the build environment is preferable because it ensures that all files produced by a build are deleted.

To clean up the environment, select **Clean** from the **Project Workbook** menu of the IDE. The cleanup flag is passed on to all steps in the build process, including build extensions. For example the resource compiler removes old temporary resource source files during a cleanup. You should take the following additional steps to make sure the environment is completely reset:

- If your configuration file includes run directives that create files, move files, or do other tasks from which you must recover, include one or more run directives that take the form:

run cleanup *command_string_list*

The command_string_list contains command that perform the recovery.

- If your build procedure include steps that happen outside the VisualAge C++ build process, check whether these steps must be repeated or altered after the cleanup.
  OS/2 WIN For example, on OS/2 and Windows NT platforms you can use the ilib command to add individual object modules to a library. You may need to take some action if the files involved are affected by the cleanup.

When the cleanup is complete, the IDE displays a box that asks you whether you want to build your project. If you reply yes, VisualAge C++ performs an initial build as though you are building this project for the first time. If you reply no, a small new codestore is created for the project because it remains open in the IDE.

Inlining

When a compiler inlines a function, it replaces the function call with the actual code for the function.

Benefits of Inlining
Inlining generally improves program performance in the following ways:

- Eliminating the overhead of the function calls and linkage.
- Exposing the function code to the optimizer.

The best candidates for inlining are small functions that are called often from a few places. Here are some more ways to improve performance with inlining:

- To increase opportunities for optimization, use constant arguments in inlined functions whenever possible.
- If you have a function that is called many times from a few functions, but infrequently from others, create a copy of the function with a different name and inline it only in the functions that call it often.
- Look for inlining opportunities when the overhead for the function is significant. For example, when functions are called within loops, function code can be reduced or repeated evaluation of the expressions may be eliminated.

Drawbacks of Inlining
An expected result of inlining is a larger executable module, because the code for the function is included at each call site. However, the difference in size may be less than the size of the function multiplied by the number of calls due to extra optimizations that can be performed.

In some cases, inlining may slow program performance, especially if you build with the build options that allow automatic inlining. Automatic inlining does not always select the best candidates for inlining. It is more effective to nominate specific functions to be inlined by including the following qualifiers in the function declarations:

C _Inline
C++ inline.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends.

You should turn off inlining if you plan to debug your executable module. Inlining can make debugging difficult; for example, if you set an entry breakpoint for a function call but the function is inlined, the breakpoint will not work.

Build Options that Affect Inlining
Build with opt(inline, yes) to enable the inlining of functions that are qualified

with the keyword _Inline in C source code or with the keyword inline in C++ source code. Build with opt(inline, no) to prevent this inlining.

Build with opt(autoInline, yes) to enable the automatic inlining of C functions that are not qualified with _Inline or C++ functions that are not qualified with inline. In automatic inlining, the compiler decides which functions to inline. Build with opt(autoInline, no) to prevent this inlining.

To gain some control over automatic inlining, you can use the following options:

AIX opt(autoInlineLimit, *number*) on AIX platforms
OS/2 WIN opt(autoInlineThreshold, *number*) on OS/2 and Windows NT platforms

By default, inlining is not performed unless you are optimizing. To optimize, build with opt(level, *level*) and specify a value greater than zero for *level*. Building with opt(level,0) implicitly sets opt(inline, no) and opt(autoInline, no). Building with any other optimization level implicitly sets opt(inline, yes) and opt(autoInline, yes).

Precompiled C Headers

Precompiled headers are a feature of the VisualAge C++ C compiler that you can use to reduce the time required to compile C programs. Using headers that have been precompiled during previous compilations eliminates the need to process header files that do not change between compilations.

To create precompiled headers, build with file(genPCH). To use them build with file(usePCH). Usually you use these options together. Alternatively you can use #pragma hdrfile in your source code.

If you always use file(genPCH) and file(usePCH) together, precompiled header files are created if they do not exist, and used if they do. When a source file is changed, the precompiled version is automatically regenerated.

Before you use precompiled headers, you must understand initial sequencesof #include directives:

- How the Compiler Uses Initial Sequences
- How You Should Use Initial Sequences
- Examples

C++ VisualAge C++ relieves you of much of the overhead of header files for C++ and compiles C++ incrementally. Precompiled headers are of no advantage in C++ and are not supported.

How the Compiler Uses Initial Sequences

The compiler generates a single precompiled object if the first lines in a source file contain a valid initial sequence of #include directives. Such an initial sequence of headers can consist of the following:

- #include directives
  The first #include directive can be preceded only by comments and preprocessing directives.
- comments
- #error directives
- null directives
- false conditional compilation blocks beginning with #elif or #else
- #endif directives

When a header contains conditional compilation directives to prevent it from being included a second time, it is only counted once in the initial sequence, even if it appears multiple times.

The initial sequence ends when the source contains any construct not in the above list or #pragma hdrstop. If a #pragma hdrstop, or any other construct than a comment or preprocessing directive appears before the first #include in the source, the compiler cannot produce a precompiled header.

Any #include directives after the initial sequence are not precompiled. They must be compiled every time the source file is compiled.

In subsequent compiles, the precompiled object can be used in all compilation units that satisfy the following conditions:

- A compilation unit must have a matching initial sequence. If it has a longer initial sequence, #include directives beyond the initial matching portion are compiled normally.
- The files that make up the precompiled header object have not changed. The compiler checks the modification date on each file.
- All macros that were expanded or tested while generating the precompiled header object are defined with the same replacement tokens. To match macros, the compiler checks all names defined or undefined before the start of the initial sequence or predefined by the compiler. Macros that are not expanded or tested during the precompile do not have to match.
- No additional macros have been defined.
- The same compiler options are in effect.

How You should Use Initial Sequences
Consider the following techniques to get the most benefit from precompiled headers:

- Use the same initial sequence of #include directives wherever possible. The more files that share the same initial sequence, the greater the improvement in your compile time.
- Reorganize your source files.
  Using precompiled headers without organizing your source files can actually **slow** your compilation.
- Use multiple initial sequences
  If the conditions that precompiled header objects must satisfy prevent their use in subsequent compiles, you may be able work around the restrictions by creating more than one precompiled header. Do this by specifying an alternative file name or directory with #pragma hdrfile or build options file(genPCH, *fileName*) and file(usePCH, *fileName*).

ExamplesExample of an Initial Sequence for Precompiled C Headers
Example Matching Initial Sequences for Precompiled C Headers
Examples of Multiple Initial Sequences for Precompiled C Headers

Organizing Your C Source for Precompiled Headers

Precompiled headers can reduce the time required to compile C source. However, to get the most improvement, you may need to reorganize your source files. Using precompiled headers without organizing your source files can actually **slow** your compilation.

There are several strategies you can use to organize your files:

- Use a precompiled header for each compilation unit
- Use a single header file
- Create a global header file
- Group headers
- Examples

Use a precompiled header for each compilation unit
Use #pragma hdrfile in each primary source file to specify a distinct precompiled header object for each compilation unit.

Benefits: Each compilation unit has its own precompiled headers, so you can create the longest possible initial sequence for each compilation unit. You do not need to match the initial sequences in other compilation units, because you are not sharing the precompiled header object.

Drawbacks: If you change one header file, the compiler regenerates every precompiled header object that includes that header. This method can also require large amounts of disk space, since many precompiled headers are generated: a common header is precompiled separately for every compilation unit that includes it.

Use a single header file Create a single header file that has #include directives for every header in your application, and include it in each primary source file.

Benefits: You get the maximum possible benefit from using precompiled header files, and the maximum possible improvement in compile time. You have an exact match of the initial sequence for every compilation unit.

Drawbacks: You may have to recompile the entire application every time you change even a single header file. For larger applications, this is probably unacceptable.

Create a global header file
Create a single header file that has #include directives for those header files that are shared by many different compilation units. Include this global header file as the first step of the initial sequence in each primary source file, followed immediately by #pragma hdrstop.

Benefits: You get the benefit from precompiling common and shared headers, without having to recompile when you change less common headers.

Drawbacks: There is only one group of precompiled headers. Any headers outside of that group are compiled normally, and the precompiled header object must be regenerated every time you change even one of the common headers.

Group headers Do the following:

1. Identify headers that are common throughout your source, and divide them into smaller groups of associated headers. A header can be included in more than one group.
2. For each group, create a header that contains #include directives for each header in the group.
3. In each primary source file, identify the precompiled header file name with #pragma hdrfile, #include the appropriate group header, and end the initial sequence with #pragma hdrstop. If a source file does not use any of the precompiled header groups, put #pragma hdrstop as the first directive in the file.

Benefits: Common headers are precompiled. Changing a header only affects the groups it belongs to, rather than requiring all precompiled headers to be regenerated. This method does not use as much disk space as having a precompiled header object for each compilation unit, and is more maintainable than a single global header file.

Drawbacks: Requires additional work to organize headers into groups.

ExamplesExample of an Initial Sequence for Precompiled C Headers
Example Matching Initial Sequences for C Precompiled Headers
Examples of Multiple Initial Sequences for C Precompiled Headers

Linker Searches

You can specify libraries and object files as sources in configuration files. To create a target, the linker satisfies references to symbols in these files by searching them in the following order:

1. Object files are searched first, in the order in which they appear inside the target directive in your configuration file.
2. Libraries are searched next, in the order in which they appear inside the target directive in your configuration file.
3. If you name libraries in source directives that are not subdirectives of any target directives, these libraries are searched after sources listed in the target directive for the target being built. The libraries are searched in the order in which they appear in your configuration file, regardless of whether they appear before or after the target directive.
4. The libraries supplied with VisualAge C++ are searched last.

For example, consider the following excerpt from a configuration file:

```
source 'a.lib'
target 'myprog.exe' { source 'myprog.cpp',
                              'b.lib',
                              'e.obj',
                              'c.lib',
                              'd.obj'}
source 'f.lib'
```

In this example, the files are searched in the following order:

1. e.obj
2. d.obj
3. b.lib
4. c.lib
5. a.lib
6. f.lib

OS/2 WIN On OS/2 and Windows NT platforms, if a library contains a reference to a symbol that is not known the linker searches first the library that contains the reference, and then looks all the other libraries in the order described above.

The paths that VisualAge C++ searches for libraries and object files are established as follows:

- OS/2 WIN On OS/2 and Windows NT platforms, the LIB environment variable also stores the search path for libraries. This environment variable is set when you open a VisualAge C++ command line window or click on one of the icons that are put the VisualAge C++ program group during installation. You can specify paths to be searched before the string in the environment variable using the option link(libSearchPath, *path*).
- AIX On AIX the default path is usr/vacpp/xlC/lib followed by /usr/lib:/lib, but you can override this path with link(libSearchPath, *path*).

Search Paths for Primary Source Files

Primary source files are C or C++ source files that are named in the configuration file.

When you specify a library or object file as a source file, the linker searches for the files to satisfy references from C and C++ source code.

When you specify a C or C++ source file in your configuration file you can include an absolute or relative path. If you do not supply an absolute path VisualAge C++ looks for the source file in the following order:

1. Look first in the directory that contains the configuration file.
2. Searched next in the path indicated by the build option incl(searchPath, *path*).
3. OS/2 WIN On OS/2 and Windows NT platforms, search the path stored in the INCLUDE environment variable.

The INCLUDE environment variable is set when you run open the VisualAge C++ command line window, or when you select and open an icon that is placed in the VisualAge C++ program group during installation.

How Configuration Files Are Processed

A configuration file defines a configuration for a project, rather than specifying an ordered set of steps for a build. Unlike makefiles, from which a make process is driven by ordered dependencies based on targets, most configuration file directives establish a relationship between source files and target files.

When you build, VisualAge C++ processes directives in a configuration file. Targets are built incrementally, and building targets drives the build process. Target directives name the output files. Source directives supplied as subdirectives of a target directive list the source files to use to create the target file. These files determine whether C++ incremental compile, C compile, link, resource bind, or other actions are performed during the build process.

Option directives apply options to the source and target files. The set of sources and options can be built up before and within the target directive. Options that are in effect when the target directive is processed, and that apply to the files used to produce the target file, are passed to the processes that operate on those files.

For example, the directives below forward options that apply to C source files and to the target executable to the C compiler and the linker. The group directive creates a group name for the source files. When the source and target directives are processed, the options defined as sourceopts and targetopts are applied to the source and target files. Options that have no meaning for the target and source file types are ignored. In this example, any source options that are specific to C++ source files, resources, or any types of files other than C source files are ignored. Any target options that do not apply to executable targets are also ignored.

```
option sourceopts = /* ... */
option targetopts = /* ... */
group MySourceList = myprog.c, myfunc1.c, myfunc2.c
option sourceopts, targetopts {
   target myprog.exe { source MySourceList }
}
```

If an option is specified more than once for the same files, second and subsequent uses of the option either append to the value of the setting or reset the option, depending on the option. Similarly, some options implicitly set or override others.

Some steps in the build process do not require a target directive to be activated. Build extensions add capabilities to the build process that are not necessarily driven by target directives. To use a build extension specify a source file that the build extension processes in a source directive, or use a

tool directive. The description of each build extension tells you what types of source files they are associated with, and how to prepare your configuration file to use them.

Other factors can alter the order in which directives in configuration files are handled:

- The if directive allows conditional processing of blocks of directives.
- The perform directive can limit the scope of variables to within blocks of directives.
- The include directive textually imbeds directives from another configuration file.
- The run directive schedules execution of command line programs. Whether execution occurs before or after compile and link stages, and before or after other run directives, depends on the targets, sources, and other parameters of the run directive.
- A configuration file can start another build as a synchronous child process, using a run directive.
- A non-zero return code from a run directive can terminate the build.
- Build errors suppress the processing of subsequent directives that depend on targets of the build.
- The error directive stops processing of the configuration file.

Example of a Configuration File

Conditional Processing in Builds

You can set up one configuration file so that it can perform different builds in different conditions. For example, you can set different build options depending on a condition you test during the build process. Any action that you can control in a configuration file can be conditional on any expression that you can specify in a configuration file. Often, the conditions depend on environment variables that are set outside the configuration file.

Use the if directive to code conditions in your configuration file. You can edit the configuration file from a **Source** view of the IDE.

Here is an example of an if directive:

```
if $OS2 {    // OS2 build
  option targetopts = commontargetopts, link(pmType, PM)
}
else {       // Windows build
```

```
   option targetopts = commontargetopts,
   link(linkWithSharedLib),
   link(subsystem, 'WINDOWS')
}
```

You can process all directives conditionally by using them as subdirectives of an if directive.

The Configuration Optimizer (C++)

The configuration optimizer is a build extension that you can use as a tool to optimize the configuration of C++ source files in a project. In an optimized configuration, the amount of preprocessing and reprocessing of identical declarations is minimized. The configuration optimizer attempts to achieve this goal by converting included source files into primary source files, whenever possible, and by applying the build option macros(global) to converted files that define global macros. It replaces the existing configuration file and the replaced file is backed up.

Before you use the configuration optimizer, make sure that you can build successfully with your configuration file and source files. If you are migrating existing source files to VisualAge C++, you should complete a successful migration build *before* you run the configuration optimizer. There are two ways to run the configuration optimizer:

- Select **Optimize Configuration** from the **Tools** menu in the IDE.
- If you are modifying your configuration file directly, use the tool directive and insert the following line in your configuration file:

```
tool 'config_opt'
```

This directive does not appear in the configuration file produced by the configuration optimizer.

The configuration optimizer does not require an existing codestore, and uses the codestore of the current build. At the end of the build, the codestore reflects the final state of the configuration file.

The configuration optimizer works iteratively, making as many passes through an internal representation of the configuration file as are required to complete all possible conversions. On each pass it tries to convert top-level included files into primary files. When optimization is complete, original configuration file is copied to a file with the same base file name as the original configuration file and the first unused file extension in the set .bk0, .bk1, and so on. The configuration optimizer then creates a new configuration file with the same base file name as the original configuration file and the file extension

.icc. The structure of the original configuration file and any include configuration file directives in it are preserved.

The configuration optimizer promotes included files to primary files by listing their file names in source directives in the configuration file. It does not delete #include directives in your source code or modify your source files in any way.

No included file is promoted if doing so can change the meaning of targets built from the including source. A header is *not* promoted in the following circumstances:

- The header is included more than once with different, non-empty content.
- A macro name defined in the header may be expanded in a primary file in a situation where it would not be expanded in an included file.
- A macro name defined in the header is already defined differently at the point where it would be preprocessed in a primary file.
- The header contains a reference to a macro name that is defined in the including file before the point of inclusion.
- The header undefines a macro name that is defined in the including file before the point of inclusion.
- The header contains at least one macro that must be global, but also defines another macro name that would be expanded in a macro source file in a situation where it does not expand in an included source.
- The header contains at least one macro that must be global, but also defines another macro name whose definition conflicts with a different macro of the same name.

Convert Included Sources to Primary Sources (C++)

When you build with VisualAge C++, you can specify C++ header files like any other source files in your configuration file. Source files listed in the configuration file are called primary sources. Alternatively, you can include header files in other source files, in the traditional manner, with #include preprocessor directives.

If you are planning to migrate source files by building in migration mode, you should do so before you convert included source files to primary source files. Migration builds process source files individually, in the order in which they appear in the configuration file. Therefore, functions, classes, and variables must be declared and macros must be declared in each file before they are used. For migration builds, include header files in the traditional manner. Migration builds may not be successful in the following circumstances:

- Any source files have dependencies on declarations in other primary source files.
- The option macros(global) is applied to any primary source files.

Because regular VisualAge C++ builds perform orderless compile and process all source files at once, the order of declarations in header files and other source files does not matter. Therefore, the declarations you traditionally put in header files can be in primary sources. You should convert included header files to primary sources for the following reasons:

- To minimize maintenance of header files. You no longer have to worry about organizing header files, setting up precompiled header files to reduce recompilation time, maintaining multiple copies of a declaration, or eliminating circular inclusion.
- To build faster. A header file that is listed as a primary source is processed only once, not every time it is included.

The converted source files can be compatible with traditional compilers because you can leave the #include directives in place. VisualAge C++ ignores #include directives if the included file is also a primary source.

You can use the configuration optimizer tool to convert header files, or you can modify your configuration file manually.

Convert the Configuration File Manually

Do not underestimate the task of manual conversion. You must determine which included files can be safely converted to primary sources without changing the meaning of targets built from the including source. If any of your converted header files define macros that must be visible in other sources, you must apply option macros(global) to them to convert them to macro sources and list them in the proper order in the configuration file, because global macros are preprocessed in the order in which the sources that define them appear in the configuration file. The configuration optimizer can do all this for you.

To convert included sources manually, make the following changes to your configuration file:

- List your header files as source files in source directives.
- Apply the option macros(global) to all header files that define macros which are used in other source files. VisualAge C++ treats the source files to which this option applies as macro sources. Macro sources are preprocessed before other primary sources and in the order in which they appear in the configuration file.
- Make sure you build with lang(staticInlineLinkage, no) and lang(staticConstLinkage, no). These are the default settings. When you

convert an included file to a primary file, variables and functions that are not declared external may acquire internal linkage. Because this is not desirable when you are converting a header file, the default setting of these build options is to give inline functions and const variables extern linkage.

- If you define types in typedef statements in header files, make sure you build with lang(localTypedefs, no). No is the default setting of this option, and makes the types defined in typedef statements available to source files.

Improve Build Times

The differences between the previous versions of VisualAge C++ and this version will initially affect the build times of existing and new applications. These differences include such things as:

- The use of configuration files rather than makefiles for application builds
- New technology related to incremental compilation
- Full ANSI C++ language support

While you are adjusting to this new technology, you may experience slow build times. This is most noticeable with newly created configuration files. The configuration files specify C++ source files which, if written in the traditional way, specify files as #include statements. If the files specified in the #include statements are also directly specified as macro source files in the configuration files, build times will improve. By using the configuration optimizer you can automatically promote many of the header files and #include files to configuration files as macro source files. When you next perform a build, the amount of preprocessing and reprocessing of identical declarations is minimized. After the files are specified as macro source files in the configuration files you may remove the #include statements from the source files, but this action is not mandatory. (For a description of primary source files, included source files, and macro source files, click on the link for *C++ Source Files and Source Regions* under Related Concepts at the bottom of this file.)

Note that when you run the configuration optimizer, a build is taking place. This configuration-optimizer build may take some time, however subsequent normal builds with the newly optimized configuration file are always faster.

Faster build times may still be possible by further optimizing the configuration file. During the initial optimization not all header files may have been promoted to the configuration file as source files. Look for header files included in multiple C++ source files that are semantically the same and are not present in the configuration file. Edit the configuration file and convert these header files to primary and macro source files. Use macro source files for files that contain macro definitions that must be visible in other source

files. You may want to adjust the order of these macro source files in the configuration file depending on the macro dependencies between them. For example, if header file `A` depends on macros defined in header file `B`, header file `B` should be specified as a macro source file and it should precede header file `A` in the configuration file.

The configuration optimizer may generate messages at completion that indicate what is not being promoted to the new configuration file. These messages may give you hints about further changes you can consider. For example, IBM Open Class header files that you create and directly include in your C++ source files are not promoted to the configuration file. You can add these to the configuration file.

As you develop and change your application code, you should run the configuration optimizer periodically to ensure that any new files are properly promoted to the configuration file. Every time you run a build using the IDE, a heuristic runs in the background and keeps track of the changes you make to your application. It will issue a reminder periodically to run the configuration optimizer.

You will also find that as your application grows in size, the overhead associated with the internal representation of your application (referred to as the codestore) becomes smaller. As this occurs your build-time performance will improve.

The following references link you to specific topics that can help you take advantage of the technology of this product. These files also contain links to related information. You may also find it useful to review the online tutorial. The tutorial is divided into sections that describe the features of the product (What's Cool!), that describe the new IDE (Tour of VisualAge), and that give you an opportunity to use the product while developing applications (Try VisualAge C++).

Chapter 2. Build

There are three ways to start a build of your project in the IDE:

- click the Build button  on the toolbar
- select **Build** from the **Project Workbook** menu
- press **Ctrl+Shift+B**

All of your source file changes are saved automatically before the build starts. It does not matter which page is displayed when you start the build.

For performance reasons, the codestore is by default not saved to disk on every build. The automatic saving of a codestore to disk is controlled by the **Number of builds before automatically saving the codestore** setting on the Settings page of the Workbook section. You can also force a save by selecting **Save Codestore** from the **Project Workbook** menu.

Progress messages and the results of your build are displayed in the information area at the bottom of the IDE window. Build messages are displayed in a Messages view.

Set Build Options

Build options are contained in the configuration file. Using the views in the Configuration section of the IDE, you can add, change, and remove build options. You can define options for targets or source files or for the entire project.

Add options to the configuration file using option directives. An option directive defines a list of one or more build options. If you name an option directive, the group of options it defines can be used in other option directives.

Option values are viewed and set using either the Change Options, Change Options Groups, or Project Options views. If an option has a default value, the option is not specified in the configuration file, but the value is used during the build process. The options that are added to the configuration file are those that have a value that is not the default value. Options that have been set are marked with green text in the views. You can remove an option from the configuration file by resetting its value back to the default.

- If the option control is an entry field, erase the text and press Enter.

© Copyright IBM Corp. 1998

- If the option control is a radio button, click the radio button that is currently selected.
- If the option control is a check box, click the check box until the default state, either off or on appears with diagonal lines.
- If the option control has a cumulative **Items** list, select each value and click **Remove** until the list is empty.

Set Build Options for Source Files or Targets

Use the Options page to set the build options for either source files or targets.

1. From the Sources and Targets view, select a source file or target.
2. In the Change Options view, set the options for that source file or target.
3. If you decide you need to start over while you are setting options, click **Reset** to restore the options to the values in the configuration file.
4. Click **Apply** to save the changes to the configuration file. An option directive containing the build options you set will be added to the configuration file. The option directive will apply to the source file or target you selected.

Set Build Options for the Project

Set build options that apply to the entire project using the Project Options view on the Project Options page in the Configuration section.

1. In the Project Options view, set the values for options that you want to apply to the project.
2. If you decide you need to start over while you are setting options, click **Reset** to restore the options to the values in the configuration file.
3. When you have finished setting options, click **Apply** to save the changes to the configuration file. An option directive with the name ProjectOptions is added to the top of the configuration file if it does not exist already. If the ProjectOptions option directive exists, the options are added to or changed in that directive. ProjectOptions contains the options you set and applies to the entire configuration file. If you add source files or targets outside the ProjectOptions options directive, the project options will not be applied to those files.

Add or Remove an Option Directive

The Configuration section of the IDE provides views and actions to add option directives to or remove them from your configuration file.

To add an option directive:

1. Select the directive that you want to either add the new option directive after or into from whatever Configuration view it is displayed in. Right-click the directive to display a menu. To add the option directive to the top of the configuration file, right-click on the background of a view to display a menu. Views that offer this pop-up menu include Source and Groups, Targets, Source and Targets, or Options Groups.
2. Select the action that adds the option directive to where you want it in the configuration file:
   - Select **Add Option after...** to add an option directive after the selected directive or select **Add Option into...** to add an option directive within a directive that has a directive block.
   - Select **Add Option...** to add an option directive to the top of the configuration file. If the ProjectOptions directive exists, the Add option action will add the appropriate directive inside the ProjectOptions block, as the first item in that block.

   A dialog box is displayed.
3. Type a name in the **Group** field if you want to create an option group that can be reused in subsequent option directives.
4. In the dialog box, set the values for the options you want in the option directive.
5. To undo the selections you have made, click **Reset**. Click **Cancel** to close the dialog box without adding the option directive.
6. Click **Apply**. The new option directive is added to the configuration file. To update the codestore and the other views in the IDE, you must build the project.

To remove an options directive, select the options directive, and then select **Remove** from the pop-up menu. The option directive is deleted from the file, and the views in the Configuration section are refreshed. If the option directive is named, and if that option group is referenced in other option directives, those references are removed as well.

Change an Option Directive

The Configuration section of the IDE workbook provides views and actions to change the options in an option directive or the name of an option directive.

To change the values of the options defined for an option directive use the Options Groups page:

1. In the Options Groups view, select the directive you want to change. If you select an option specifier, the Change Options Groups view positions you to the control for that option.
2. In the Change Options Group view, select the options you want to change for the group, and change them. You can also set new options.
3. Click **Apply** to save the the options to the configuration files, and to update the views in the Configuration section.

To change or add a name to an options directive use the Options Groups page:

1. In the Options Groups view on, select the directive you want to change.
2. In the Change Options Groups view, type the name in the **Group** field.
3. Click **Apply** to save the changed name to the configuration file, and to update the views in the Configuration section.

Add Source Files to a Project

You can add either existing or new source files to your project.

Add Existing Source Files to a Project

To add existing source files to your project, use the **Files** view in the **Host** section.

1. Select the file you want to add to your project, and click mouse button 2. A menu is displayed. You can also select multiple files by holding down the **Ctrl** key and selecting the files with mouse button 1.
2. Click mouse button 2 over the file or files you have selected. A pop-up menu appears.
3. Select the action that adds the source file to where you want it in the configuration file:
   - Select **Add Source to Configuration**. to add the source file to the top of the configuration file in a source directive.
   - Select **Add Source to Source Directive...** to add the source file to an existing source directive. A dialog box is displayed.
   - Select **Add Source to Target Directive...** to add the source file to an existing target directive in the configuration file. The source file is added in a source directive. A dialog box is displayed.

- In the dialog box, select the directive that you want to add this file to. The complete list of directives in the current configuration file and any included configuration files is displayed by clicking the arrow at the end of the **Source** or **Target** field.
- Ensure **Use Relative Path** is selected to add the source files with a path name relative to the configuration file directory. When this option is not selected, the source files are added with full path names.
- To undo the selections you have made and to reset the **Use Relative Path** to its default, click **Reset**. Click **Cancel** to close the dialog box without adding the source file.
- Click **Apply** to save the change to the configuration file. To update the codestore and the other views in the IDE, you must build the project.

Add New Source Files to a Project

To add a new source file to your project, use the Source and Groups page of the Configuration section.

1. In the Source and Groups view, select the source or group directive that you want to add the new source file to.
2. In the Change Source and Groups view, type the path and file name of the source file in the **Source** entry field.
3. Ensure **Use Relative Path** is selected to add the source file with a path relative to configuration file directory. If you do not select this option, the source will be added with the path name you specified.
4. Click **Add** to add the file to the **Files** list. The **Files** list is updated.
5. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**.
6. Click **Apply** to save the change to the configuration file. To update the other views in the IDE, you must build the project.

The new file will not exist in the file system until you edit it. To edit the new file, use the Source and Groups page:

1. In the Source and Groups view, select the new file.
2. In the Source view, click the arrow for the file name field. Select the new file from the list.
3. The new file will be displayed with one empty line for you to edit.

Remove Source Files from a Project

To remove source files from your configuration file, use any Configuration section view where the file is listed (such as the Source and Groups view on the Source and Groups page). Select a source file and then select **Remove** from the pop-up menu. The source file is deleted from the configuration file, and the views in the Configuration section are refreshed when you rebuild the project. The source files remains in your file system. You can also select multiple files to be removed.

Add a Source Directive

The Configuration section of the IDE workbook provides views and actions to add a source directive to your configuration file.

To add a source directive to the top of the configuration file:

1. To add a new source directive to the top of the configuration file, click mouse button 2 on the background of a Source and Groups, Targets, Source and Targets, or Options Groups view to display a menu.
2. Select **Add Source...**. If the ProjectOptions directive exists, the Add source action will add the appropriate directive inside the ProjectOptions block, as the first item in that block.
3. Select the type of the source files that are included in the source directive from the **Type** list.
4. Type the name of a source file in the **Source** field, or click **Find** to select one or more source files from your file system. If the file name is not enclosed in quotes or does not have a file extension, then it is assumed to be a variable which must be already defined in the configuration file.
5. Ensure **Use Relative Path** is selected to add the source files with a path name relative to the configuration file directory. When this option is not selected, the source files are added with full path names.
6. Click **Add** to add the files to the **Files** list.
7. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**. Click **Cancel** to exit the dialog box without adding the source directive.
8. Click **Apply**. The new source directive is added to the configuration file. To update the codestore and the other views in the IDE, you must build the project.

To add a source directive into an existing directive:

1. Select a directive from any Configuration view it is displayed in. Click mouse button 2 over the directive to display a menu.

2. Select one of the following options:
   - Select **Add Source after...** to add a source directive after the directive
   - Select **Add Source into...** to add a source directive within a directive that has a directive block.

   A dialog box is displayed.
3. Select the type of the source files that are included in the source directive from the **Type** list.
4. Type the name of a source file in the **Source** field, or click **Find** to select one or more source files from your file system. If the file name is not enclosed in quotes or does not have a file extension, then it is assumed to be a variable which must be already defined in the configuration file.
5. Ensure **Use Relative Path** is selected to add the source files with a path name relative to the configuration file directory. When this option is not selected, the source files are added with full path names.
6. Click **Add** to add the files to the **Files** list.
7. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**. Click **Cancel** to exit the dialog box without adding the source directive.
8. Click **Apply**. The new source directive is added to the configuration file. To update the codestore and the other views in the IDE, you must build the project.

Add a Group Directive

The Configuration section of the IDE workbook provides views and actions to add a group directive to your configuration file. The group directive does not specify source files for a project, so the group still needs to be listed in a source directive. The group directive makes dealing with source files that are processed in the same manner easier than dealing with the files individually. You can add descriptive names to a group, improving the readability of your configuration file.

To add a group directive:

1. From a Configuration view, select the directive after which or into which you want to add the new group directive. Click mouse button 2 on the directive to display a menu. To add the group directive to the top of the configuration file, click mouse button 2 on the background of a view to display a menu. Views that provide these pop-up menus include Source and Groups, Targets, Source and Targets, and Options Groups.
2. Select the action that adds the group directive to where you want it in the configuration file:

- Select **Add Group after...**. to add a group directive after the directive or select **Add Group into...**. to add a group directive within a directive that has a directive block.
- Select **Add Group...** to add a group directive to the top of the configuration file. If the ProjectOptions directive exists, the Add group action will add the appropriate directive inside the ProjectOptions block, as the first item in that block.

A dialog box is displayed.

3. In the **Group** field, type the name of the group.
4. Type the name of a source file in the **Source** field, or click **Find** to select one or more source files from your file system. If the file name is not enclosed in quotes or does not have a file extension, then it is assumed to be a variable which must be already defined in the configuration file.
5. Ensure **Use Relative Path** is selected to add the source files with a path name relative to the configuration file directory. When this option is not selected, the source files are added with full path names.
6. Click **Add** to add the files to the **Files** list.
7. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**. Click **Cancel** to exit the dialog box without adding the group directive.
8. Click **Apply**. The new group directive is added to the configuration file. The views in the Configuration section are refreshed to show the new directive. To update the views in other sections, you must build the project. For a source files that are part of this group directive to be part of a build, the group directive must be specified in a source directive.

Remove a Source or Group Directive

You can remove source or group directives from any Configuration section view where the source or group directive object is displayed. To remove the directive, click mouse button 2 on it, and select **Remove** from the pop-up menu. The directive is deleted from the configuration file, and the views in the Configuration section are refreshed. If the group directive is used in any other source or group directives, those references are also removed.

You can select to remove either multiple source directive or multiple group directives, but you cannot mix the directives for a single remove action.

Change a Group Directive's Name

You change the name of a group directive using the Source and Groups page of the Configuration section.

To change the name of a group directive:

1. In the Source and Groups view, select the group directive you want to change.
2. In the Change Source and Groups view, type the new name in the **Group** field.
3. Click **Apply** to save the changed name to the configuration file, and to update the views in the Configuration section.

If the group directive is used in any other source or group directives, those references to the name are also changed.

Change a Source Directive's Type

Use the Source and Groups page of the Configuration section to change the type of a source directive.

To change the type of the source directive:

1. In the Source and Groups view, select the source directive you want to change.
2. In the Change Source and Groups view, select a source file type from the **Type** list. If the source file type you want is not in the list, you can type your own into the field.
3. Click **Apply** to save the changed type to the configuration file, and to update the views in the Configuration section.

Change a Source or Group Directive's Source Files

Use the Source and Groups page of the Configuration section to add , change, and remove source files from a source or group directive.

To add a source file to a source or group directive:

1. In the Source and Groups view, select the source or group directive to which you want to add the source file.
2. In the Change Source and Groups view, type the file name of the source file in the **Source** entry field, or click **Find** to select files from your file system. You can select multiple files in the Find dialog box.

3. Ensure **Use Relative Path** is selected to add the source file with a path relative to configuration file directory. If you do not select this option, the source will be added with the full path name.
4. Click **Add** to add the file to the **Files** list. The **Files** list is updated.
5. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**.
6. Click **Apply** to save the change to the configuration file, and to update the other views in the Configuration section.

To change or remove a file from a source or group directive:

1. In the Source and Groups view, select the source or group directive that contains the file you want to change or remove.
2. In the Change Source and Groups view, select the file to be changed or removed from the **Files** list.
3. If you are changing the file name, make the changes in the **Source** field. You can also change whether a relative path is used to specify the file. If the file name is not enclosed in quotes or does not have a file extension, then it is assumed to be a variable which must be already defined in the configuration file.
4. Click **Change** or **Remove** to change or remove the file. The **Files** list is updated.
5. Click **Apply** to save the change to the configuration file, and to update the other views in the Configuration section. To update the views in other sections, you must build the project.

Move a Source File

The Configuration section of the IDE provides actions to move a source file from a source directive to another source directive or to a new source directive in a target directive

To move a source file:

1. Click mouse button 2 on the file. A menu is displayed.
2. Select **Move to Source...** to move the file to a source or group directive, or select **Move to Target...** to move the file to a a new source directive in a target directive. The **Move to Source** or the **Move to Target** dialog box is displayed.
3. In the dialog box, select the directive to which want to move this file from the **Source** or **Target** list.
4. To undo the selections you have made, click **Reset**. Click **Cancel** to close the dialog box without moving the source file.

5. Click the **Apply** button to move the source file, and save the change to the configuration file. The views in the Configuration section are refreshed. To update the views in other sections, you must build the project.

Override the Default Type of a Source File

The type of a source determines what processes operate on it during a build. By default, the file extension determines the type of a source file. For example, a file with extension .c is by default a C source file, but you can use it as a C++ source.

In the **Configuration** section of the IDE, you can specify the type of a source file as well as the file name. Alternatively, you can edit your configuration file directly and override the default type of a source by including a type clause when you name the file in the source directive. For example, the following line from a configuration file identifies the file parker.c as an input file for a build and sets its type as C++ source.

```
source type(cpp) 'parker.c'
```

Use Resource Source Files as Sources for a Build

When you supply resource source files as source files for a build, the resource compiler runs automatically as a build extension during the build.

The best way to supply more than one resource source file is to combine them, using rcinclude resource compiler statements, into a single resource source file that you list in your configuration file. If you list more than one resource source file directly, the resource compiler build extension creates a temporary resource source file that includes your resource source files with rcinclude statements. However it does not check whether doing so will create problems such as conflicting macro definitions.

When you clean up the build environment by selecting Clean from the Project Workbook menu of the IDE, all temporary files created by the resource compiler build extension are deleted.

You can apply the following options to resource source files:

- res_compiler (*string*)
  The string contains the name of the resource compiler. By default the following resource compiler is used:
  AIX The **irc** tool on AIX
  OS/2 The **rc** tool on OS/2
  WIN The **irc** tool on Windows

- res_rc_options (*string*)
  The string contains command line options that you would supply if you were running the resource compiler from the command line. The options are described with the command line operation of the tool, and depend upon which resource compiler you are using. You can list several options in one string.

WIN For example, the following excerpt from a configuration file builds a resource DLL from the resource source file mystuff.rc on Windows platforms. The option the -D with the value PLAIN is passed to the **irc** resource compiler.

```
target 'mystuff.dll' {
   option res_rc_options( '-DPLAIN' ) {
      source 'mystuff.rc'
   }
}
```

When used as a build extension, the resource compiler runs only once for each target that is built from resource source files. It runs only if the resource binary file does not exist or has an earlier timestamp than the resource source file. It will also run if any #include files are updated. The following files are created by the resource compiler:

AIX On AIX, the **irc** tool creates a binary .res file.
OS/2 On OS/2, the **rc** tool binds resources directly into the target creates no resource binaries
WIN On Windows the **irc** tool creates binary .res and .rbj files.

Compile and Bind Resources

Building an application that uses resources is a simpler process than it was in earlier versions of VisualAge C++. Simply specify your resource files as source files for a build. You can specify resource files of the following forms:

- One or more resource source files (.rc) files.
  The best way to supply more than one resource file is to use rcinclude resource compiler statements in a single resource source file that you list in your configuration file. If you list more than one resource source file directly, they are merged into one temporary resource source file, but no check is made for potential problems such as conflicting macro definitions.
- AIX OS/2 One compiled resource file (.res).
- WIN One or more compiled resource file (.res) or resource object file (.rbj). Resource object files are produced by the Windows resource compiler and you do usually not work with them.

Here is an excerpt from a configuration file. These directives build myprog.exe from files mysource.c and myres.rc.

```
target 'myprog.exe' {
   source 'mysource.c'
   source 'myres.rc'
}
```

If you have shared or dynamic link libraries for resouces, you must load them explicitly at run time. You can use the following API to dynamically load a library:

- C++ The class IDynamicLinkLibrary from the IBM Open Class Library
- OS/2 The function DosLoadModule in the OS/2 Toolkit
- WIN The function loadlibrary in the Windows SKD

You can build shared or dynamic link libraries for resources in the same configuration file that builds an executable file that uses those libraries.

Compile IPF Help During a Build

You can use the Information Processing Facility (IPF) to create on line documentation. The IPF compiler, **ipfc** on Windows and OS/2, **ipf/x** on AIX, takes as input a file in IPF format, an .ipf file, and creates documentation in the form of an .inf or .hlp file. You can run **ipf** as a build extension during a build.

To run **ipf** during a build, simply list the IPF files as source files for the build. If these files have the file extension .ipf, the build process recognizes them as IPF source files. If they have any other file extensions, you must explicitly specify that their type is ipf. The build process runs **ipf** for each IPF source file for which the resulting help file does not exist or has an earlier date than the IPF source file.

WIN Error messages can be found in the ipfc.log file.

You can apply the following options to IPF source files.

- ipf_preopt (*string*)
  The string may contain any of the command line options that you can specify before the file name when you run **ipf** from the command line. You can list several options in one string.
- ipf_output (*string*)
  The string identifies the type of the output file and can be 'inf' or 'hlp'. If such a file does not exist or has an earlier timestamp than the IPF source file, **ipf** is run. The *string* is not passed to the **ipf** command. For example, if

ipf_output = 'inf' and filename.inf does not exist, then **ipf** is run. If filename.inf does exist and is up to date, **ipf** is not run.

- ipf_postopt (*string*)
  The string may contain any of the command line options that you can specify after the file name when you run **ipf** from the command line. You can list several options in one string.

OS/2 For example, the following lines from a configuration file build an .inf file on OS/2:

```
option ipf_preopt( '/w:2' ),
       ipf_output( 'inf' ),
       ipf_postopt( '/i' ) {
   source 'wpayroll.ipf'
}
```

The lines above have the same effect as entering the following on the command line:

```
ipfc /w:2 wpayroll.ipf /i
```

WIN For example, the following lines from a configuration file build an .hlp file on Windows:

```
option ipf_preopt( '/q /w2' ),
       ipf_output( 'hlp' ){
   source 'wpayroll.ipf'
}
```

The lines above have the same effect as entering the following on the command line:

```
ipfc /q /w2 wpayroll.ipf
```

Use Makefiles as Sources for a Build

You can process a makefile during a VisualAge C++ build, by supplying a makefile as a source in your configuration file. The build process runs the make utility implicitly to process the makefile **before** the compile and link is performed. Typically, you do this to set up the environment at the start of the build process or to run tools that are not integrated into the IDE.

Supply a makefile as a source for a build to run the make utility as a build extension of the VisualAge C++ build process. A source file is recognized as a makefile if it has the file extension .mak or is qualified by a type(mak) clause. Because it is a build extension, the way you specify options for the make utility is different from other processes that run during a build. You can use the following configuration file options to control how the make utility is run when processing makefiles during a build:

- make_name (*string*)
  The string contains the make utility executable's name.
  AIX The default executable name for the make utility on AIX is **make**.
  OS/2 WIN The default executable name for the make utility on OS/2 and Windows NT is **nmake**.
- make_preopt (*string*)
  The string contains any command line options that you would place before the makefile name if you were running the make utility from the command line. Do not include the /f option, because it is added automatically before the makefile name when the make utility is invoked. The make utility will fail if this option is specified twice.
- make_postopt (*string*)
  The string contains any command line options that you would place after the makefile name if you were running the make utility from the command line.

The makefile itself is exactly like a makefile you use when you run the make utility from the command line.

Processing makefiles during a build has some limitations:

- Since the /f option is added automatically to the invocation command to specify the name of the makefile to the make utility, any make utility that does not support this option will not work with this build extension.
- You cannot run the make utility as a build extension after or at the same time as the compile and link. However you can use the run directive to start the make utility as a command line program.
- The build process cannot test time stamps on files named inside the makefile. It always runs the make utility, regardless whether all files are up to date.
- Do not use the /q option. When run with the /q option, the make utility returns zero only if all targets are up to date and returns a nonzero value otherwise. The build process interprets a nonzero return value as a failure code and terminates your build.
- Output that displays to the standard output stream, stdout, is not visible. You can redirect the streams by adding the redirection characters to the end of the string you supply for option make_postopt, as in the example below. Errors will be available, however.
- Any settings made to environment variables from the makefile are lost when the make utility ends. If you want to use makefiles to set up your environment, you should start the build from the makefile, by running the **vacbld** command from the makefile.

Here is an excerpt from a configuration file that runs the make utility on a file called makefile:

```
option make_name( 'mymake' )
       make_preopt( '/a'),
       make_postopt('/nologo > make.out') {
   source type(mak) makefile
}
```

In this example, the type clause is required in the source directive because the makefile does not have an extension. You can omit this clause if the source file has the file extension .mak. Two command line arguments, /a and /nologo, are specified and output to stdout is redirected to the file make.out.

The effect of the example is equivalent to entering the following line on the command line:

```
mymake /a /f makefile /nologo > make.out
```

Build with Existing Object Files and Libraries

You can build new modules that link to existing object modules and libraries. Simply add the .obj or .lib files to the list of sources in your configuration file. Name the target output module in the configuration file. The build process runs the linker to produce a target linked to the specified object files or libraries.

The libraries may be static libraries, or be may import libraries if the target module loads shared libraries at run time.

The VisualAge C++ run-time library is linked by default, and you do not need to mention it in your configuration file, unless you choose not to link with the single-thread static version. Use build options to link to a different version of the run-time library or not to link to the run-time library.

Sources can be object files and libraries that were created in previous builds.

You can also use object files and libraries that were built by earlier versions of VisualAge C++ and predecessor products. There are some migration considerations:

- If the objects and libraries were created from C++ sources that do not comply with the C++ standard, you may have to set build options for compatibility with C++ sources from earlier versions of VisualAge C++ and predecessor products.
- OS/2 WIN DLLs that were built with the run-time libraries of VisualAge C++ 3.0 or 3.5, CSet++ 2.0 or 2.1, and IBM C and C++ Compilers 3.6 on OS/2 and Windows NT platforms must be migrated to use the new run-time libraries.

Here is an excerpt from a configuration file that builds a program in file saturn.exe from a C source file, an existing library, and an object file:

```
/* ... set build options ... */
option planet6 {
   target 'saturn.exe' {
      source 'saturn.cpp', 'rings.obj', 'planet.lib'
   }
}
```

Add Part Source to the Configuration File (AIX)

In VisualAge C++, the Visual Builder and Data Access Builder components use parts to support application development. Visual Builder visual or nonvisual parts are files with the extension .vbf or .vbe. Data access parts are composed of a data access session file, a source file with the extension .dax, and one or more Visual Builder parts.

You can open the Visual Builder or the Data Access Builder from the command line or an icon to edit a part. The resulting part file is saved by the builder.

From the Builder you can generate source files. A message appears after the file generation; take note of the generated file names. The generated source files (for example .hpp, .cpp, and .rc files) are used during the build process.

If you need to check the names of the generated files at a later time, you can. For example, with Visual Builder parts, you can browse the .vbf file using a standard editor; ensure you do not change the file. The line 'VBGeneratedFiles:' shows any files which might have been generated. Depending on the way your part is constructed, the files may not actually exist.

Add only the required source files to your project. Any files (like .hpp or .cpv files) that are normally included with a preprocessor directive do not need to be added to the configuration file.

You can use a vbload.dat file to initialize the Visual Builder with required base parts. This file must appear in the directory from which you launch the Visual Builder (with the ivb command). Use the vbload.dat file found in the samples directories as a starting point. The .vbe files listed from the INCLUDE directory are the base Visual Builder parts required by Data Access parts, while the rest of the file can be replaced with your part's own Builder file dependencies.

Precompile Embedded SQL During a Build

Source files that contain embedded SQL statements must be precompiled before they can be used as ordinary C or C++ source files. If you are using DB2, you can run the DB2 SQL precompiler as a build extension during a build.

DB2 must be running on your workstation when you use the DB2 SQL precompiler in a build. If DB2 is not active:

- Make sure DB2 is installed.
- In a command line window, enter the **db2start** command to start DB2. The **db2start** command is described in the DB2 documentation.

To build your project, you can open the VisualAge C++ IDE or enter the **vacbld** command in the VisualAge C++ command line window, in the usual manner.

To precompile embedded SQL during a build, specify the C or C++ source files that contain the embedded SQL as source files for the build:

- C C source files with embedded SQL should have the file extension .sqc.
- C++ C++ source files with embedded SQL should have the file extension .sqx.

If your source files have different file extensions, you must explicitly specify that the type is sqc for C and sqx for C++. When the build process encounters a file of type sqc or sqx, and DB2 is installed, the DB2 SQL precompile function is run automatically.

You must apply the following option to source files with embedded SQL:

- SQLTargetDB ('*string*')
  The string contains the name of the database against which the sqc or sqx file is precompiled.

You can optionally apply the following options to source files with embedded SQL:

- SQLConnOptions ('*string*')
  In the string, you can specify any of the options that you can append to the command 'db2 connect to *dbname*' when you are precompiling from the command line. These options are described in your DB2 documentation.
- SQLPrepOptions ('*string*')
  In the string, you can specify any of the options you can append to the command 'db2 precompile *fileName*' when you are precompiling from the command line. These options are described in your DB2 documentation.

- SQLPromptEnabled ('*flag*')
  When the flag is set to 'Yes', the build extension prompts the user for a user ID and password if a connection fails to be established due to an incorrect ID or password. The default is 'Yes'. Set the flag to 'No' to suppress the prompt.

The DB2 precompile build extension produces the following files:

- C source files with file extension .c from .sqc files
- C++ source files with file extension .cxx from .sqx files

You can list the files produced by the build extension as sources in your configuration file. For example, you can perform a DB2 SQL precompile to create .c or .cxx files from .sqc or .sqx source files, and use the .c or .cxx files as sources for the build.You can even specify the .sqc or .sqx files directly as sources for the executable file or libraries that are targets of the build.

For example, the following lines from a configuration file build an executable target, MyTown.exe, from one import library and two C++ source files that contain embedded SQL.

```
option SQLTargetDB('towndb1') {
   target 'MyTown.exe' {
      option SQLTargetDB('towndb2') {
         source 'downTown.sqx'
      }
      source 'upTown.sqx', 'db2api.lib'
   }
}
```

In the lines above, the file downTown.sqx is precompiled with option SQLTargetDB('towndb2'). The file upTown.sqx is precompiled with option SQLTargetDB('towndb1'). To create MyTown.exe, files downTown.cxx, upTown.cxx, and db2api.lib are used.

Add or Change Targets Overview

A target is an output file from a build. A target directive in a configuration file defines a target. A target directive usually contains one or more source or group directives that specify the source files used to build the target. In the IDE, the Targets view on the Targets page lists all of target directives in the configuration file. The target's source directives and any group directives defined in those source directives are also listed.

The Configuration section of the IDE provides views and actions to:

- Add or remove target directives
- Change the name or type of a target

- Add , change, and remove source files from a target directive
- Move a source file defined in a target to another target

Add or Remove a Target

The Configuration section of the IDE workbook provides views and actions to add and remove target directives from your configuration file.

To add a target directive:

1. Select the directive that you want to either add the new target directive after or into from whatever Configuration view it is displayed in. Click mouse button 2 on the directive to display a menu. To add the target directive to the top of the configuration file, click mouse button 2 on the background of a Source and Groups, Targets, Source and Targets, or Options Groups view to display a menu.
2. Select the action that adds the target directive to where you want it in the configuration file:
   - Select **Add Target after...** to add a target directive after the directive or select **Add Target into...** to add a target directive within a directive that has a directive block.
   - Select **Add Target...** to add a target directive to the top of the configuration file. If the ProjectOptions directive exists, the Add target action will add the appropriate directive inside the ProjectOptions block, as the first item in that block.

   A dialog box is displayed.
3. Type a name in the **Target** field. If the file name is not enclosed in quotes or does not have a file extension, it is assumed to be a variable which must be already defined in the configuration file.
4. Click the arrow beside the **Type** field to display the menu. Select a target type.
5. If you want to undo the selections you have made, click **Reset**. Click **Cancel** to exit the dialog box without adding the target directive.
6. Click **Apply**. The new target directive is added to the configuration file. To update the codestore and the other views in the IDE, you must build the project.

To remove a target directive, click mouse button 2 on the target directive, and select **Remove** from the pop-up menu. The selected target directive is deleted from the configuration file, and the views in the Configuration section are refreshed. If there are source and group directives defined in the target, they are also removed from the configuration file.

Change a Target's Source Files

You can add, change, and remove source files from a target directive using the Change Source and Groups view on the Targets page of the Configuration section. The Change Source and Groups view is displayed when you select a source directive in the Targets view.

To add a source file to a target directive:

1. Select a source or group directive in the Targets view.
2. In the Change Source and Groups view, type the file name of the source file in the **Source** entry field, or click **Find** to select files from your file system. In the**Find** dialog, you can select multiple files by holding down the **Ctrl** key and selecting the files with mouse button 1.
3. Ensure **Use Relative Path** is selected to add the source file with a path relative to configuration file directory. If you do not select this option, the source will be added with a full path name.
4. Click **Add** to add the file to the **Files** list. The **Files** list is updated.
5. To change or remove a file from the **Files** list, select the file and click **Change** or **Remove**. If you want to undo the selections you have made, click **Reset**.
6. Click **Apply** to save the change to the configuration file. To update the other views in the IDE, you must build the project.

If the target directive has no source directives, then you need to create one. Right-click on the target and select **Add Source into...**. The **Add Source into...** dialog creates the source directive and allows you to specify source files for it.

To change or remove a file from a target directive:

1. Select the source or group directive in the Targets view that contains the file you want to change or remove.
2. From the **Files** list, select a file.
3. If you are changing the file name, make the changes in the **Source** field. You can also change whether or not a relative path is used to specify the file. If the file name is not enclosed in quotes or does not have a file extension, then it is assumed to be a variable which must be already defined in the configuration file.
4. Click **Change** or **Remove** to change or remove the file. The **Files** list is updated.
5. Click **Apply** to save the change to the configuration file. To update the other views in the IDE, you must build the project.

Build Executable Programs

To build an executable program, make sure your target type is exe. You can do this by specifying a target file with file extension .exe. If you omit the extension from the file name in the target directive, and do not explicitly specify a target type, VisualAge C++ creates an executable file with no extension.

For example, this excerpt from a configuration file builds a program called myprog.exe from three source files, including a library. The program is built with the multithread version of the run-time library.

```
option link(linkWithMultiThreadLib) {
   target 'myprog.exe' {
      source 'myprog.cpp', 'myprog.hpp', 'mylib.lib'
   }
}
```

Produce Object Files from a Build

C When you build from C sources, object files are produced automatically. VisualAge C++ provides a C compiler that creates an object file from each non-included C source file in the traditional manner.

C++ To output object files when you build from C++ sources, specify an object file as the type of the target. When you build from C++ sources, VisualAge C++ performs incremental builds and does not produce object files unless you ask for them explicitly.

C++ All targets built from C++ sources are specified in the configuration file. Unlike the C compiler, the C++ compiler can combine the code generated from more than one C++ source file into one object file. The mapping of source files to target files in the configuration file determines how many C++ sources are used to build each target object file. For example, this excerpt from a configuration file builds an object file that contains the code generated from two C++ source files:

```
target 'myobject.obj' {
  source 'a.cpp', 'b.cpp'
}
```

C++ Specifying the same source file for more than one target can result in violations of the one-definition rule, so you usually do not attempt to create an object file as well as another target from the same source file. You can get around the one-definition rule with a two-step build. For example, to build both an executable file and an object file from at least one common C++ source file, you can follow these steps:

1. Specify the object file as a target and the common C++ source files as a sources.
2. Specify the executable file as a target and the object file produced by the previous step as a source.

OS/2 You can use the object files produced by VisualAge C++ as input to linkers other than the VisualAge C++ linker. If you use the **ilink** linker supplied with VisualAge C++ for OS/2 3.0 and IBM C and C++ Compilers 3.6 to build a DLL from object modules created by VisualAge C++, make sure you specify the **ilink**option /include:_dllentry. The **ilink** linker does not automatically generate a reference to _dllentry and the DLL is not initialized properly at run-time without this entry point.

Produce Multiple Targets from One Build

You can specify any number of targets for a build. For example, you can build several libraries and the executable file that uses those libraries in one build. Simply specify each library and the executable as target files.

C If your sources are C source files or object files, you can reuse them within one build as often as you want, to build any number of targets.

C++ The one-definition rule limits the way you can reuse C++ source files in one codestore. You you cannot refine external functions or variables in the codestore. Therefore, you sometimes need more than one codestore to build more than one target. If two or more of the targets contain common external functions or variables, you must have different codestores for the targets that conflict. Here are the most common combinations of targets, sources, and options for which you need separate codestores:

- More than one executable, because the external function main is used by each executable.
- More than one target from at least one common source file.
- Different versions of the same target, built with different options. For example, you may want to create debug and optimized versions the same executable file in one build. You can use conditional processing and one configuration file, but the configuration file must be processed more than once to produce the different versions of the target and each version has its own codestore.

There are two approaches you can take for builds:

- Use separate configuration files
  If the targets are to be used together in the same application, it is often better to have a separate configuration file for each target.

The IDE works with a single configuration file and you must have a separate project for each configuration file. However, You can close one project and open another in a single IDE session.

- Use one configuration file with conditional logic
  You can use conditional processing in builds to determine which targets to generate each time the configuration file is processed. This is often preferable when the targets are alternate forms of the same code, such as optimized and non-optimized versions of the same executable or dynamic and static versions of the same library.

  The IDE builds a single codestore. Every time you build from the same configuration file in one IDE session, the codestore is overridden. Also, the value of environment variables is set when the IDE opens and cannot be changed in an IDE session. Therefore, the conditional logic in your configuration file should depend on local variables, which you can change between builds, rather than on environment variables.

You can automate a build that requires more than one codestore the following way:

1. Create a separate configuration file for each target that you must build with a separate codestore. You can reduce duplication by moving lines that are common to more than one configuration file into other files that you can use as included configuration files during the build. Replace the common lines with include directives to process the included configuration files at build time.
2. Create a makefile that runs the **vacbld**command once for each configuration file. You can also set up environment variables in the makefile for use during each run of **vacbld**.

   or

   Create another configuration file that contains a sequence of run directives, each of which issues a **vacbld** command. This method removes the need for a makefile. However, it may be less convenient if you use environment variables, because you cannot assign values to environment variables in a configuration file.

View Build Messages

Build messages are displayed in a Messages view. By default, the Messages page in the Project section contains a Messages view linked to a Source view. When you select a message, the source code related to the message is displayed for you to edit.

If the automatic messages page setting is enabled, messages generated from a build cause the page designated as the messages page to be displayed. By default, the Messages page is the designated messages page.

Control Build Messages

You control the output of build messages by specifying build options in the configuration file. The IDE provides actions to disable a message or to treat an information or warning message as an error message. These actions add the appropriate build options to the configuration file. In a Messages view, click mouse button 2 on a message object to display the pop-up menu containing these actions.

To disable a message, select **Disable Message** from the pop-up menu of the message. The report message option is added to the configuration file as a project option.

To treat an information or warning message as an error message, select **Treat Message as Error** from the pop-up menu of the message. The report treataserror option is added to the configuration file as a project option.

To reset the output behavior of a build message, use the Project Options page in the Configuration section. This page also contains other options for controlling build messages. If you want to control build messages for a specific file, use the Change Options view on the Options page.

Clean Up the Build Environment

Select **Clean** from the **Project Workbook** menu to erase the codestore and intermediate files.

When you use the **Clean** action, the pages in the Project and Configuration sections are erased and recreated since the codestore that they rely on has been erased. If you opened sections for any project or configuration objects, they will also be erased. The Workbook and Host sections remain.

Build in a Team Environment

When several developers are working on the same project, they can share files in two ways:

- Share source files as they would in traditional build environment, but keep separate copies of compiled code
- Share source files, compiled code, and possibly targets

Your team may also be able to use a shared configuration file, if no drive letters or absolute paths are included with file names in the configuration file. All relative paths in a configuration file specify directories relative to the location of the configuration file. To make a configuration file that your team

can share, put it at the root of the directory tree for source files and specify relative paths for all source and target files.

If you cannot use only relative file names, your team can start with a shared configuration file. The developers may be able to share the configuration file by using variables for the drive letters and paths.

Sharing Source Files Only
The simpler approach is to share source files only.

C++ Each developer maintains a personal copy of the codestore, and can take advantage of local incremental C++ builds. Your team can share all source files or extract local copies from the central repository. After refreshing the local sources or accessing the shared source files, each developer must incrementally build in order to incorporate changes made to all sources.

Sharing Sources and the Codestore
If your team runs centralized builds, you can share the codestore and targets, as well as the source files. In this approach, each developer takes a copy of the codestore and sources that may be changed locally. For example, if the centralized build occurs overnight, each developer can update the local cache in the morning. You can make local builds during the day, and check the updated sources back into the central repository at the end of the day for the next overnight build.

For this approach to work, your team must use a common configuration file. You must also build to keep the codestore up to date with new paths and timestamps for the source and target files. These synchronizing builds should be relatively quick.

C++ An advantage of sharing the codestore is that the amount of work to be done by local incremental builds is relatively small, because each developer must incorporate the changes made by only one person to the sources each day. If your team does not share the codestore, each developer must incorporate the changes to all sources into the local codestore. On the other hand, the approach of sharing source files only is simpler and gives individual developers more freedom to modify the personal copies of the configuration file.

Your team may be able to take advantage of features for file change control that allow the following actions on a central repository of source files:

- Check files out
- Check files in
- Extract files

Build from the VisualAge C++ Command Line Window

Before you build with VisualAge C++, you must have a configuration file that defines the sources or input files, targets or output files, and options that apply to the build. To build, open the VisualAge C++ command line window and issue the **vacbld** command.

C++ The **vacbld** tool performs incremental builds for C++ sources. If you have an existing codestore, **vacbld** updates it. If you do not have a codestore, **vacbld** creates one.

You can use **vacbld** to compile C code, or to process any valid configuration file.

Specify the same configuration file and codestore for command line builds that you use for builds started from the IDE. When you are not using the IDE, you can make changes to your configuration file or source code with any text editor. The next build, regardless of whether is it launched by the **Build** button in the IDE or the **vacbld** command, synchronizes the codestore with C++ source files and the configuration file.

To monitor the progress of a command line build, see all warning messages, or select the level of error messages to detect, set options of the **vacbld** command. Error messages are sent to the standard error stream, stderr.

Redirect Standard Streams

In C and C++ programs, the standard streams read from the keyboard and write to the screen. Many utilities included in VisualAge C++ and command line programs use the standard streams. When you launch a program from the operating system GUI, or want input and output operations on these streams to read from and write to files, you can redirect the standard streams.

OS/2 The Presentation Manager (PM) interface uses the stdout and stderr streams somewhat differently than non-Presentation Manager programs. Strings written to stdout or stderr do not show up on the screen unless redirected.

There are two ways you can redirect the standard streams:

- **From Within an Application**
  To redirect C standard streams to a file from within your application, use the freopen library function. For example, to send your output to a file called pia.out instead of sending it to stdout, code the following statement in your program:

```
freopen('pia.out', 'w', stdout);
```

- **From the Command Line**
  To redirect C or C++ standard streams to a file from the command line, use the standard redirection symbols > and < with the file handles for standard streams. For example, to run the program bill.exe, which has two required parameters *XYZ* and *123*, and redirect the output from stdout to a file called bill.out, use the following command:

  ```
  bill XYZ 123 > bill.out
  ```

You can also use the file handles to redirect one standard stream to another. For example, to redirect stderr to stdout, you use the command:

```
2> &1
```

C You cannot use redirection from the command line for memory files.

WIN On Windows 95, you cannot use redirection from the command line for exceptions. stderr always goes to the screen.

Run Command Line Programs During Builds

During a build, you can run command line programs. Do this to perform activities during a build that are not integrated into the VisualAge C++ kernel. For example, you can create directories and move files, perform some network activities, or run your own programs that have some other role in your build procedure.

You can start programs from the configuration files in one of the following ways:

- Take advantage of build extensions
- Use the run directive
- Process a makefile

Take advantage of build extensions
VisualAge C++ builds recognize certain command line tools and utilities as build extensions. The documentation for each command line program supplied with VisualAge C++ tells you if it is available as a build extension and, if it is an extension, how to run it from your configuration file.

Some build extensions run automatically when you specify a source file that is of the type that the build extension processes. For example, the resource compiler is run implicitly if you specify a resource source file as a source for a build. To run other build extensions, such as the configuration optimizer, you must use the tool directive in your configuration file.

Not all build extensions have command line equivalents. For example, the only way to run the C compiler is as a build extension.

Use the run directive to start any command line program
You can supply any sequence of command line strings to the run directive, and specify whether the commands are to be run before or after the compile and link steps of the build. The commands run synchronously as a child process of the build.

Process a makefile
The make utility is itself a build extension. If you supply a file of type .mak as a source to a VisualAge C++ build, the make utility processes that makefile. Typically, you do this to set up the environment at the start of the build process or to run tools that are not integrated into the IDE. You can use makefiles much like you do in traditional programming environments, including earlier versions of VisualAge C++ and predecessor products, except that you cannot compile or link directly from a makefile.

Nominate Inline Functions

Inlining is a C and C++ feature that you can exploit to improve the performance of your code. To reduce the overhead of function calls, you can request that the code for some functions be inserted inline, in place of the call statement, and that no actual call occurs.

You cannot specify exactly which function calls are replaced with inlined copies of functions, but you can make suggestions that compiler may act upon. For functions in the VisualAge C++ run-time library, you have limited control. For functions that you supply, you have greater flexibility.

Inlining Library Functions
In the VisualAge C++ runtime library, intrinsic functions are the functions that may be inlined. Some intrinsic functions belong to the set of built-in functions that are declared in <builtin.h>. You cannot turn off inlining for the built-in functions because they have no bodies in the runtime library. The compiler must generate the code for them. For other intrinsic functions, the compiler can either insert the code for the function inline or create a call to the function in the runtime library.

Some functions are inlined only when you build your module with optimization on. To inline these functions build with opt(level, *level*) when *level* has a non-zero value.

When optimization is on, the intrinsic functions are not inlined in the following circumstances:

- If you build with opt(size), these functions are not inlined to minimize the size of your executable module.
- You can selectively disable inlining of library functions by enclosing the name of the function in parenthesis when you call the function. For example, in intrinsic function strcpy is not inlined as a result of the following statement.

```
(strcpy) (str1, str2);
```

Inlining User Code

You can also request that the compiler inline the code for your own functions.

C A C function is a candidate for inlining if it meets one of the following criteria.

- It is declared with the C keyword _Inline. The function must be in the same source file as the call, or in an included header.
- OS/2 WIN It is smaller than the size specified with opt(autoInlineThreshold, *number*).
- It is called from a small number of locations, is small, and has storage class static.

C++ A C++ function is a candidate for inlining if it meets one of the following criteria.

- It is declared with the C++ specifier inline.
- It is a member function defined in a class declaration.
- OS/2 WIN It is smaller than the size specified with opt(autoInlineThreshold, *number*).
- It is called from a small number of locations, and is itself a small function.

Function calls cannot be inlined if they meet one of the following criteria:

- The function has a variable number of arguments.
- A function is called from within inline expansion of itself. For a function that is directly recursive, the call to the function from within itself is not inlined. For example, the following table shows possible inlining of three functions, A, B and C, that call each other.

function call	**Can the call be inlined?**
A calls B	yes
B calls C	yes
C calls B	no, because it is made from within an inline expansion of B itself.

To enable inlining of candidate functions qualified with inline or _Inline, you must build with opt(inline) and turn on optimization by building with opt(level, *level*) when *level* has a non-zero value.

To enable automatic inlining of candidate C and C++ functions, build with opt(autoInline, yes) and turn on optimization by building with opt(level, *level*) where level has a non-zero value.
OS/2 WIN Use opt(autoInlineThreshold, *number*) to control the amount of inlining that will occur.

You cannot completely control what functions are inlined. In all cases, the compiler ultimately decides, based on internal algorithms.

Check Out, Check In or Extract a Source File

The IDE provides an interface to access a change control system. You can check out, check in, or extract source files from a change control system, such as IBM TeamConnection, using IDE actions. These actions are only available if the change control support has been enabled. To enable and configure change control support, click **Change Control Configuration** on the Settings page of the Workbook section.

To check out, check in or extract a source file from a change control system:

1. Click mouse button 2 on a file, primary source, or included source object to display a menu. Select **Checkout**, **Checkin,** or **Extract**. Multiple file selection is supported for these actions.
2. If you have not suppressed the change control prompt, a dialog box is displayed. Confirm the change control command and click **OK**.
3. The request is submitted to the change control system. The status of the command is displayed in the Change Control Status dialog box.

Get Mangled Names for C++ Symbols

You may need to specify C++ symbols in mangled form as input to a VisualAge C++ build.

If you are migrating an existing DLL from earlier versions of VisualAge C++ and predecessor products, you may already have mangled names in the module definition (.def) file for the DLL. Module definition files are no longer required, but you may find it convenient to extract the list of mangled symbol names from them.

You can get the mangled names from an existing object file or static library, by using one of the following commands.

The **cppfilt** Command
Run **cppfilt** in binary mode. Specify option /B for binary mode. Usually, you also want to set /P to include public symbols, /R to include referenced symbols, or /X to include exported symbols. Specify the object module or library as the input files.

The **ilib** Command
You can use ilib to get mangled names from a static library. Follow the instructions to list the contents of a static library. The listing file includes the decorated and mangled version of names.

You must edit the output produced by running these commands to get a text file that lists only the names you want.

Example Output from **cppfilt** in Binary Mode
Example Listing Produced by the **ilib** Command



Created in Canada