

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

Migration Concepts and Tasks

Version 4.0



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

Migration Concepts and Tasks

Version 4.0

First Edition (September 1998)

This edition applies to Release 4.0 of IBM VisualAge C++ Professional 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.

If you have comments about this document, address them to:

IBM Canada Ltd. Laboratory
Information Development
2G / 345 / 1150 / TOR
1150 Eglinton Avenue East
North York, Ontario, Canada, M3C 1H7
FAX: (416) 448-6161
torrcf@ca.ibm.com

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 . . . vi
Trademarks and Service Marks vi

Migration **1**
Migration Tools 1
Convert Makefiles to Configuration Files 2
MAKE2CFG Output. 4
The MAKE2CFG Conversion Aid and Recursive Makes 4
Makefile Conversion Aid Considerations 5
MAKE2CFG Restrictions and Limitations 5
Multiple Configuration Files 6
Configuration File Names Generated by MAKE2CFG 7
Configuration File Contents 9
Command Grouping 10
Mapping Makefile Commands to Configuration File Directives 10
Migration Build 11
Migrate Source Files. 12
After a Successful Migration Build . . . 13
Limitations. 14
Migration Build Considerations 14
Optimize Configuration Files after Migration 15
Using the Configuration Optimizer. . . 15
Error Messages 16
Generate Makefiles from Configuration Files 16
The Format of the Generated Makefile 18
Restrictions and Limitations 19
Configuration File Directives Mapped to Makefile Rules 20
Using Parts from Previous Versions of VisualAge 21
Using Handwritten Code from Previous Versions of VisualAge 22
Migrate Existing Code 23
Migration Considerations 23
Build with Existing Object Files and Libraries 24
Object Files Created in Previous VisualAge C++ Releases 25
Use DLLs and Executable Files Created in Previous VisualAge C++ Releases 25
DLLs and Executable Files Created in Previous VisualAge C++ Releases 26
Create Module Definition Files (OS/2, Windows) 26
Convert Module Definition Files (OS/2, Windows) 27
Compile Options from Earlier Versions of VisualAge C++ (OS/2, Windows) 33
Link Options from Earlier Versions of VisualAge C++ (OS/2, Windows) 44
C and C++ Binary and Run-Time Interoperability with Previous Releases 46
Object Files and Interoperability. . . . 46
DLLs, Executable Files, and Interoperability 47
Special Considerations When Using C++ I/O 47
Previous VisualAge Run-Time Libraries and Version 4.0 Equivalents. 47
VisualAge C++ for OS/2, Version 3.0 48
VisualAge for C++ for Windows, Version 3.5 49
C and C++ Compilers for OS/2, AIX and for Windows NT, Version 3.6. 50
Convert Included Sources to Primary Sources (C++). 52
EXCNV (Windows) 53

© Copyright IBM Corp. 1998

Notices

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. 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:

Director of Licensing
Intellectual Property and & Licensing
International Business Machines Corporation
North Castle Drive, MD - NC119
Armonk, New York 10504-1785
U.S.A.

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.

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

© Copyright IBM Corp. 1998

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.

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.

Warning: 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:

AIX
AS/400
DB2
CICS
IBM
OS/2
OS/390
OS/400
RS/6000
San Francisco
VisualAge
Visual Servlet
WebSphere

Lotus, Lotus Notes eSuite, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the U.S. and/or other countries.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems, Inc.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the U.S. and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and/or other countries.

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the U.S. and/or other countries.

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

Migration

Migration Tools

With this release, VisualAge C++ introduces a number of changes to improve your application development process. These changes include full ANSI C++ language support, incremental C++ compilation, the use of configuration files to control builds, and an integrated development environment directly connected to the compilation process.

Attempts to build existing applications in this new development environment may generate errors. The tools and processes described in this document will help you migrate applications developed in the following environments:

- AIX C Set++ for AIX, Version 3.1.4
- OS/2 VisualAge C++ for OS/2, Version 3.0
- WIN VisualAge for C++ for Windows, Version 3.5
- AIX OS/2 WIN IBM C and C++ Compilers for OS/2, AIX and for Windows NT, Version 3.6

Makefile Conversion Aid
Use **make2cfg** as an aid in initially converting your makefiles to configuration files. A different configuration file is created from the groups of targets specified in your makefiles, and each configuration file, after some modification, can be used to run a build process.

Migrate Source Files
Migrate your source files one file at a time with the migration build process. You can resolve any errors in your files as you progress through the migration build process. This approach allows you to maintain close control of your source files.

Configuration File Optimizer
While not strictly part of a migration process (all configuration files can be optimized periodically), you should use the configuration file optimizer after an initial project migration to get the best performance out of the new build process.

WIN Resource File Migrator (EXCNV)
Use this tool to migrate your VisualAge for C++ for Windows, Version 3.5 resource source compiler files to the correct resource file format for this release. This ensures that extended menus are handled correctly.

© Copyright IBM Corp. 1998

Configuration Exporter (CFGEXPRT)
Use **cfgexprtas an aid in generating makefiles from configuration files. After modification, the makefiles can be used to build applications on other platforms. This approach allows you to maintain a common source base for applications that are to be compiled on VisualAge C++ Version 4.0, and on other platforms or with other compilers.**

Convert Makefiles to Configuration Files

The command-line makefile conversion aid, **make2cfg**, helps you convert your existing makefiles into configuration files, which are required for the build process. **make2cfg** attempts to reproduce as closely as possible the general behavior of the original makefile. In many cases the generated configuration file needs modification to handle situations specific to the build process being converted. **make2cfg** does not support the cross-platform conversion of makefiles. It must be run on the platform for which the makefile was written.

Determine whether the makefile maps to a single build configuration or multiple build configurations by examining the makefile variables and how the make utility is invoked. A makefile represents more than one build configuration if it can be run with different sets of options and environment variables, for example, if the same makefile could be used to build either a debug or a non-debug version of an executable.

Repeat the following steps for each build configuration represented in the makefile:

1. Ensure that the make utility is visible in your system path.
2. Change to the directory that contains the makefile for your application.
3. Ensure that the makefile you intend to convert is valid by executing the make utility with the `-n -p` option. This generates a stream of commands without executing them.
4. Create any files that the makefile would normally depend on and generate. This ensures that the make utility used by **make2cfg** runs successfully because the files on which it has dependencies do exist. An error message is generated if one of these files is missing.
5. Set any environment variables needed for the makefile to run properly. For example, if you intend to build a debug version of the project, and there is a TYPE environment variable that is set to `/Ti+ /O-` for debug builds, and to `/Ti- /O+` for production builds, ensure that TYPE is set for a debug build before running **make2cfg**.
6. Invoke **make2cfg** by typing the following command at the VisualAge C++ command prompt:

`make2cfg [`*options*`]`
Where *options* are:

-f *filename*
: Specifies the name of the makefile to be converted, otherwise **make2cfg** assumes *makefile* as the name. For example, specify `make2cfg -f one.mak` at the command prompt if you had previously specified `nmake -f one.mak`.

-qna[meprefix]=*prefix*
: Sets the prefix for the name of the configuration file. This is useful if multiple sets of configuration files are to be created from the same makefile. If you do not specify this option, the name of the configuration file created by **make2cfg** is based on the target being produced.

-qov[erwrite]
: Overwrites a configuration file if it has the same file name as the one being created. If you do not specify this option, **make2cfg** ends with an error if the file name for the new configuration file already exists.

-qto[ol]
: Specifies the executable name of the make utility that runs on the makefile. The default make utility on OS/2 and Windows NT is `nmake`, and on AIX it is `make`. The make utilities supported by the conversion aid are:

  `nmake` on OS/2 Versions 3 through 5
  `nmake` on VisualAge for C++ for Windows, Version 3.5
  `make` on AIX Version 4.1.4 (with PTF U440378) and on AIX Versions 4.2 and 4.3
  `gnumake` Version 3.74 on AIX, OS/2, and Windows NT.

-qru[n]
: Initiates a migration build on the first configuration file generated by **make2cfg** by executing the command `vacbld` *`config.icc`* `-migrationbuild`. This first configuration file generally represents the first target in the makefile, or the closest to the ALL target in the tree of dependencies in the makefile. For complicated makefiles producing multiple .exe, .dll, or .lib targets, multiple configuration files are generated by **make2cfg**. When this occurs you should not use `-qrun` because not all of the configuration files are used in the build process.

make utility options
: Are the options you use when you run the make utility. Use the same options when you run **make2cfg**.

-? or -h[elp]

Displays the command line help that lists the format for running **make2cfg** and identifies the options you can use.

7. Compare the output created for each build configuration. If differences exist, you can do one of the following actions:
   - Manually combine the output to create a single configuration file that can handle all build configurations
   - Keep separate configuration files, each representing a different build configuration

MAKE2CFG Output

Two forms of output result from running the conversion aid:

- One or more configuration files. See Makefile Conversion Aid Considerations for a description of the guidelines that determine when **make2cfg** creates multiple configuration files.
- A message displayed on the screen indicating that the conversion is successful. Any errors generated by the conversion aid are displayed on the screen. If errors occur in the make utility run by the conversion aid, the make utility errors are listed in the error log, `make2cfg.err`.

You may need to modify the configuration file generated by **make2cfg** before you perform a migration build or a normal build. For example, you may need to change some of the options specified in the configuration file to ensure compatibility between your code and the new language level supported in this product. You may also want to change the options to allow you to take advantage of the new level of IBM Open Class in the product. In some cases, build steps in the makefile, such as the creation of import libraries, are no longer required in the configuration file and are removed by the conversion aid. In other cases, these may not be removed by the **make2cfg** process and you need to edit the configuration file to remove them. The error messages generated during your first few builds will also help you determine the changes you need to make in your configuration file.

The MAKE2CFG Conversion Aid and Recursive Makes

The **make2cfg** conversion aid does not handle recursive make invocations (the make utility invoked within a makefile). For situations where recursive makes occur, take the following actions:

- Run **make2cfg** in each directory that has a makefile that is executed as part of a build.
- **make2cfg** converts make invocation lines into run directives. Change the run directives so that the **vacbld** command is used to invoke the newly generated configuration files. For example, assume `make1.mak` invokes

make2.mak, and that the new configuration files created for each are main.icc and general.icc, respectively. The following run directive is created in main.icc:

```
// nmake /f make2.mak
run targets ('help')
   'nmake /f make2.mak'
```

Change the run directive to use the **vacbld** command in place of **nmake**:

```
// nmake /f make2.mak
run targets ('help')
   'vacbld general.icc -m'
```

Note that the -m option has been added so that a migration build is run rather than a normal build. Remove the -m option when you are ready to run a normal build.

Makefile Conversion Aid Considerations

The makefile conversion aid, **make2cfg**, helps you convert your existing makefiles into configuration files. Configuration files are required for the build process in the IDE. The conversion attempts to reproduce as closely as possible the general behavior of the makefile. (Information about using the **make2cfg** aid is described in Convert Makefiles to Configuration Files.)

MAKE2CFG Restrictions and Limitations

- **make2cfg** performs the conversion on the expanded output from the make utility (that is, using options -n -p), instead of on the makefile itself. As a result, some abstraction and some of the structure of the makefile may be lost during the conversion.
- Since **make2cfg** does not execute the commands within the makefile, you need to ensure that all referenced source files are present. If files are created in a rule, and those files are not listed as the targets, then use of them elsewhere in a dependency list will result in **make2cfg** failures. Therefore, any files listed as a dependency must either be present or must be the target of another rule. If an intermediate file is missing, **make2cfg** will display an error message indicating that the make utility did not run successfully, and the make2cfg.err file will contain the make utility error indicating which file is missing.
- The order in which commands are executed may differ between the makefile and the configuration file. With a configuration file:
  - The incremental build process determines the order in which actions are performed and you have a limited ability to control this ordering.
- More than one configuration file may be required because:

  - The ordering of the commands cannot be represented when a file is both an input and an output for the build.
  - More than one executable file cannot be represented in the same codestore because each executable defines a main() function, and this does not meet the requirements of the C++ one-definition rule.
  - The same source base is used to create both a dynamic library and a static library as targets. Since a target built for use in a dynamic library (DLL, or shared library) is compiled differently than when it is used in a static library (LIB, or non-shared library), a separate configuration file is required.
- OS/2 WIN Run directives are created in the configuration file for each step in the makefile that specifies the dynamic creation of .def files. However, you should use the link(export) option to handle exports outside of .def files.
- Options are converted for the following IBM tools: the C++ compiler, the C compiler, the linker, the resource compiler, and (for OS/2 and Windows NT), ILIB . Other tools will be invoked using a run directive in the configuration file.
- AIX Linker options are converted for the `ld` command, but not for the `makeC++SharedLib` or `ar` commands.
- **make2cfg** does not support cross-platform conversion of makefiles. It must be run on the platform for which the makefile was written.

Multiple Configuration Files

make2cfg attempts to put all targets specified in the makefile into a single configuration file. However, there are configuration file restrictions that prevent **make2cfg** from doing so in some cases. When those conditions occur, **make2cfg** creates multiple configuration files based on the following guidelines:

- Each configuration file may contain only one executable.
- A source file may only be specified in a configuration file once.
- More than one configuration file may be required to represent a target if it is built using a sequence of commands whose ordering cannot be represented. This condition occurs when entries result in such things as run directives or tool extensions that are executed between build commands.

OS/2 WIN Import library generation is handled as part of the build of a DLL. Makefile rules that build an import library, by using ILIB for example, will not be propagated to the configuration file if the following conditions occur:

- A DLL is being built
- All of the dependencies of the import library target are also dependencies of the DLL

If an executable file is also being built and it has a dependency on the import library, the dependency will be removed because a DLL and an executable file built in the same configuration file is all that is needed.

Configuration File Names Generated by MAKE2CFG

The names of configuration file created by **make2cfg** are based on the target being produced plus the .icc extension. If `main.exe` is the target, the configuration file name created by **make2cfg** is `main.icc`.

There is the possibility that multiple targets in a makefile could result in configuration files with the same name. To avoid this, the file extension of the target is appended to the configuration file name with an underscore. If `main.exe` and `main.dll` are the targets, the configuration file names will be `main_exe.icc` and `main_dll.icc`, respectively.

Since FAT file systems on OS/2 limit the number of characters in a file name to 8 characters for the name and 3 characters for the extension, the possibility exists for configuration file names to occur more than once. Duplicate names can also occur on non-FAT file systems when directories are specified. To avoid duplicate names, characters are replaced with numbers, starting with the 8th character.

OS/2 WIN The following table summarizes naming alternatives for configuration files for OS/2 and Windows NT:

Makefile target	**8.3 file names (OS/2 FAT file system)**	**Long file names (OS/2 HPFS and Windows NT)**
main.exe	main.icc	main.icc
themainfile.exe	themainf.icc	themainfile.icc
themainfile.exe themainfile.dll	them_exe.icc them_dll.icc	themainfile_exe.icc themainfile_dll.icc
themainfile.exe themainfile.dll themainframe.dll	them_exe.icc them_dll.icc them_dl1.icc	themainfile_exe.icc themainfile_dll.icc themainframe.icc
main.exe mymain.dll	(when qnameprefix=theprefix) theprefi.icc thepref1.icc	(when qnameprefix=theprefix) theprefix_main.icc theprefix_mymain.icc
themainfile.exe themainfile.dll themainframe.dll	(when qnameprefix=my) my_thema.icc my_them1.icc my_them2.icc	(when qnameprefix=my) my_themainfile_exe.icc my_themainfile_dll.icc my_themainframe.icc

sub1\hi.dll sub2\hi.dll ... sub11\hi.dll	(when qnameprefix=a) a_hi_dll.icc a_hi_dl1.icc ... a_hi_d10.icc	(when qnameprefix=a) a_hi_dll.icc a_hi_dll1.icc ... a_hi_dll10.icc
sub1\hi.dll sub2\hi.dll ... sub11\hi.dll	hi_dll.icc hi_dll1.icc ... hi_dll10.icc	hi_dll.icc hi_dll1.icc ... hi_dll10.icc

Note that the examples in the table assume that a separate configuration file is generated for each target shown. However, a configuration file may contain more than one DLL, in which case one of the DLLs is chosen for use as the configuration file name.

AIX The following table summarizes naming alternatives for configuration files for AIX:

Makefile target	**Long file names (AIX)**
main	main.icc
themainfile	themainfile.icc
themainfile themainfile.a	themainfile.icc themainfile_a.icc
themainfile themainfile.a themainframe.a	themainfile.icc themainfile_a.icc themainframe.icc
main mymain.a	(when qnameprefix=theprefix) theprefix_main.icc theprefix_mymain.icc
themainfile themainfile.a themainframe.a	(when qnameprefix=my) my_themainfile.icc my_themainfile_a.icc my_themainframe.icc
sub1/hi.a sub2/hi.a ... sub11/hi.a	(when qnameprefix=a) a_hi_a.icc a_hi_a1.icc ... a_hi_a10.icc
sub1/hi.a sub2/hi.a ... sub11/hi.a	hi_a.icc hi_a1.icc ... hi_a10.icc

When the **make2cfg** process ends, it displays a list of the configuration files created and the corresponding targets that the names were generated from. For example:

```
Makefile Target           Configuration File
themainfile.exe ......... them_exe.icc
themainfile.dll ......... them_dll.icc
themainframe.dll ........ them_dl1.icc
```

Configuration File Contents

Configuration files created by **make2cfg** contain the following:

- A comment stating that the configuration file was generated by **make2cfg**.
- The command line options that were used with the makefile.
- A warning that **make2cfg** only processes the conditional path that is in effect, based on the environment at the time **make2cfg** runs.
- A tool directive to launch the configuration optimizer, `tool 'config_opt'`. Although the tool directive is ignored during a migration build, it does take effect during the first regular build following file migration. Note that the tool directive is no longer specified in the optimized configuration file that has been generated.
- An option group definition that contains options for compatibility with previous compilers.

  For OS/2 and Windows NT, the options in the group are:

  ```
  lang(allowTypedefAsClassName,yes),
  gen(mangleLinkageKeywordOnlyFuncs,no),
  lang(compatMath,yes),
  lang(ansiForStatementScopes,no),
  lang(nokeyword,'bool'),
  lang(nokeyword,'export'),
  lang(nokeyword,'false'),
  lang(nokeyword,'mutable'),
  lang(nokeyword,'true'),
  lang(templateDefImpls,dontparse),
  macros(redefine,yes),
  link(nameMangling,compat)
  ```

  For AIX, the options in the group are:

  ```
  lang(allowTypedefAsClassName,yes),
  gen(compatNestedTemplateAlignmentRule,yes),
  lang(compatMath,yes),
  lang(ansiForStatementScopes,no),
  lang(nokeyword,'bool'),
  lang(nokeyword,'export'),
  lang(nokeyword,'false'),
  lang(nokeyword,'mutable'),
  lang(nokeyword,'true'),
  lang(templateDefImpls,dontparse),
  macros(redefine,yes),
  link(nameMangling,compat)
  ```

- A separate section for each rule in the makefile. Source groups and option groups are defined for each command.
- A separate options group for each file extension that results in the invocation of a command.
- D_ and R_ variables identifying dependency and rule source groups, respectively, that may be used in a run directive.
- O_ variables identifying option groups.
- Regular source groups beginning with an underscore (_).

Note that you need to remove source type(obj) 'cppwpa3.obj' from the .icc file if cppwpa3.obj was present in the original Windows NT makefile. You also need to add option gen(rtti) to the configuration file if you plan to use the new version of IBM Open Class included in the VisualAge C++, Version 4.0.

Command Grouping

In a makefile, the commands listed within a description block will either all be executed or not. To reproduce this behavior for run directives in a configuration file, **make2cfg** adds a group directive for each description block in the makefile. Each group directive contains a list of all of the dependency files that could trigger the description block. By using the group directive as the source clause for all run directives under a single description block, the run directives are grouped together for common execution. The following is an example of a group directive created by **make2cfg**:

```
group R_ahellow4_rc_ = 'ahellow4.rcx'
run sources(R_ahellow4_rc_) targets('ahellow4.rc')
  'IRCCNV ahellow4.rcx ahellow4.rc'
group D_ahellow4_rc_ = 'ahellow4.rc', R_ahellow4_rc_
```

Mapping Makefile Commands to Configuration File Directives

Makefile commands for tools not directly supported by the **make2cfg** conversion aid will be converted to run directives in the configuration file. **make2cfg** will not produce build extensions in the configuration file for these tools. For example, the IPF command in a Windows NT makefile is converted to the following run directive in the configuration file:

```
group R_general_hlp_ = 'general.ipf'
run sources(R_general_hlp_) targets('general.hlp')
  'ipfc /q /hlp general.ipf'
```

Options for the following tools will be converted to their configuration file equivalents:

- C compiler
- C++ compiler
- OS/2 WIN ILIB (only for options that relate to the linker)

- Linker
- Resource compiler

However, not every option in the makefile has a corresponding option in the configuration file. The makefile conversion proceeds on the assumption that the makefile worked prior to conversion and therefore all options should be valid.

AIX On an AIX operating system, the compiler has a number of command variations with different default options that can be used directly from the command line or can be specified with the `-F` option. When the `-F` option is used with a recognized compiler command, **make2cfg** will map it to an option, `defaults(`*option*`)`, in the configuration file, where *option* is one of the following. The 'i' prefix on the command variation is removed when it is inserted in the option.

xlC	xlC128	xlC128_r	xlC128_r7	xlC_r	xlC_r4	xlC_r7
xlc ixlc	xlc128 ixlc128	xlc128_r	-	xlc_r ixlc_r	xlc_r4	xlc_r7
cc icc	cc128 icc128	-	-	cc_r icc_r	cc_r4	cc_r7
-	-	-	-	-	CC_r4	-
c89 ic89	-	-	-	-	-	-

Migration Build

The migration build process helps you migrate your existing source files. This step in your migration tasks follows the creation of a configuration file (through the **make2cfg** conversion aid).

A migration build compiles and links one file at a time, and gives you direct control of any corrections that need to be made in your source files.

The migration build process uses a configuration file created by the makefile-to-configuration-file conversion aid, **make2cfg**. You can also use a configuration file that you created manually.

The migration build attempts to compile each file specified in the configuration file, one file at a time. If errors are detected in a file, the migration build stops. This allows you to fix the errors before restarting the migration build process.

Migrate Source Files

The migration build process allows you to migrate existing source files developed in the following environments:

- AIX C Set++ for AIX, Version 3.1.4
- OS/2 VisualAge C++ for OS/2, Version 3.0
- WIN VisualAge for C++ for Windows, Version 3.5
- AIX OS/2 WIN IBM C and C++ Compilers for OS/2, AIX and for Windows NT, Version 3.6

A migration build fully compiles one source file at a time, compared to normal builds which perform incremental compilation only on changed files. The migration build allows you to fix any errors in the file before it proceeds to the next file. The migration build process does not make any changes to your source files.

Before using the migration build process, create a configuration file from an existing makefile using the makefile-to-configuration-file conversion aid, **make2cfg**. This ensures that the configuration file matches the requirements for the migration build process, and that the file has not been optimized.

To migrate your source files:

1. Invoke the migration build process in one of the following ways:

   - Open the project in the IDE using the configuration file created with **make2cfg**. Click **No** in response to the message dialog: 'Project has not been built yet. Would you like to begin a build?' Select **Migration Build** from the **Tools** menu. You may be asked whether you want to continue with the migration build. This occurs because the IDE normally opens with a set of options that permit a normal build. Click **Yes** to continue the migration build. The project closes and reopens with options that are set for a migration build, and the build continues. If you click **No**, the migration build does not proceed.
   - Type the following command at the VisualAge C++ command prompt to open the IDE with migration build options in effect:
     `vacide -nob -noc`
     Open the project using the configuration file created with **make2cfg** and select **Migration Build** from the **Tools** menu in the IDE.
   - Type the following command at the VisualAge C++ command prompt:
     `vacbld` *`config.icc`* `-migrationbuild`
     where *config.icc* is the name of your configuration file.
2. If the configuration file contains macro source files, the migration build process is likely to fail. If you are running the migration build process

through the IDE, the migration build issues a warning and gives you the option of continuing the build or of stopping it. If you are running the migration build process with the **vacbld** command at the VisualAge C++ command prompt, the warning message appears and you can stop the process with the Ctrl+C key combination.

If you stop the build, you can take one of the following actions:

- Re-create the configuration file by running the **make2cfg** conversion aid on your makefile, and then repeat the migration build process.
- Remove the source files (from the configuration file) on which the other source files are dependent, and include them as header files in the dependent source files in the traditional manner, that is, as #include statements. (This ensures that functions, classes, and variables are declared and that macros are defined in each file before it is processed.) Restart the migration build process.
- Stop using the migration build process and attempt a normal build.

3. Correct any errors identified in the single file, and run the migration build again.
   - If you invoked the migration build from the IDE, click the Buildbutton  on the toolbar, and the build proceeds and attempts to build the next source file.
   - If you invoked the migration build through the **vacbld** command, you need to invoke the command again. In this case the migration build starts from the beginning and all source files are processed again, one file at a time.

4. Continue to run the migration build until you correct the errors in each of the files and the migration build completes successfully. Even though the tool directive (`tool 'config_opt'`) to invoke the configuration optimizer during a normal build may have been added to the configuration file, the tool directive is ignored during the migration build.

After a Successful Migration Build

- After completing a successful migration build, you need to run a normal build to generate a persistent codestore (an .ics file).
  - To run a normal build in the IDE, close the project and the IDE, and then restart the IDE so that the options required for a normal build are set. Run a normal build on your migrated project to generate a persistent codestore (an .ics file).
  - If you used the **vacbld** command with the -migrationbuild option, perform a normal build using the **vacbld** command to generate a persistent codestore (an .ics file).

Since the **make2cfg** process added the tool directive `tool 'config_opt'` to the configuration file, this normal build will also result in an optimized configuration file.

- If you used #include directives as a workaround for source file interdependencies, you can now restore the source files to the configuration file and run a normal build. If the build is successful, you may remove the #include statements from the files, but this action is not mandatory.

Notes:

- When you run a migration build with the IDE, you have turned off the options that generate a persistent codestore during a build. Any subsequent projects opened in the IDE during the same session will not generate a persistent codestore with either a normal build or a migration build. To ensure that the options for a persistent codestore are reset, close the IDE and reopen it before attempting a normal build.

- The migration build process is only intended for the migration of an application from a previous release. You should not use the migration build process once you have completed the migration. If you do reuse it, the results may be unpredictable.

Limitations

The migration build process should not be used with either a persistent codestore or a configuration file that has been optimized in *any* way.

If a persistent codestore (an .ics file), exists from an earlier build of the configuration file, it will be ignored by the migration build process. If you specify a codestore on the command line when you invoke the migration build through **vacbld**, it will also be ignored.

Use the **make2cfg** makefile conversion aid to convert your makefile to a configuration file to avoid any optimization of the configuration file.

Migration Build Considerations

The migration build process handles files differently than a normal build does. Here are some of the differences:

Migration Build

- Ensures that each file is fully compiled before making any attempt to compile another file.
- Does not necessarily process files in the order in which they are specified in the configuration file.

- Does not alter the way the compiler compiles a source file, but it prevents the compiler from compiling more than one source file during each migration build iteration.

Normal Build

- Runs the build process on one group of files at a time.
- Ensures that macro source files are processed in order and before any primary source files. Other than this ordering, there is no guarantee which parts of any of the files are incorporated at any one time. The unit of compilation (a non-included source file) that exists for conventional compilers has no meaning for the incremental compiler.
- Decomposes groups of files into source regions and places the regions into various work queues. Items on the work queue are processed by their priority. There is no guarantee of the order of processing for items in the same work queue which have the same priority.

Optimize Configuration Files after Migration

The configuration optimizer changes your configuration file to make builds more efficient. It does this by converting included sources to primary sources whenever possible. This reduces the amount of preprocessing and reprocessing of identical declarations and also reduces the amount of work to rebuild your project after one of the sources is changed.

Using the Configuration Optimizer

Since the **make2cfg** process adds the tool directive `tool 'config_opt'` to the configuration file, once you complete a migration build and run a normal build, the configuration optimizer automatically creates an optimized configuration file for you. The tool directive does not appear in the new configuration file produced by the configuration optimizer.

In addition to this initial configuration file optimization, the IDE keeps track of the number of changes you make to your project and will prompt you to run the configuration optimizer.

To run the configuration optimizer in the IDE, select **Optimize Configuration** in the **Tools** menu. Or you can add the tool directive `tool 'config_opt'` to your configuration file and run a normal build. Your configuration file is optimized during this build, and the tool directive does not appear in the new configuration file that is produced.

The configuration optimizer creates a backup of the configuration file if any changes are made to it. The first backup has the suffix BK0, the second, BK1,

and so on. After the BK9 backup is created, the sequence starts at BK0 again, and any previous backup files are overwritten.

Error Messages

The configuration optimizer will generate warning messages indicating which headers could not be promoted to sources and why.

Generate Makefiles from Configuration Files

The configuration exporter (**cfgexprt**) generates a makefile from a configuration file. The generated makefile needs to be modified before it can be used on other platforms or with other compilers to build the project as it was described in a project configuration file.

The configuration file lists source file names and the targets these source files are translated into. The exact list of source files and targets can be altered by the values of environment variables in the configuration file. The configuration exporter creates a makefile for one set of values of the environment variables.

Ensure that the configuration file builds successfully and has been optimized before you run **cfgexprt**. To generate a makefile from a configuration file, type the following command at the VisualAge C++ command prompt:
`cfgexprt [options] configuration_file`

where *options* are:

-? or -Help
Displays the command line help that lists the format for running **cfgexprt** and identifies the options you can use.

-C *codestore_file*
The name of the codestore file. This option is required if the `-qdependend` option is also specified; otherwise, it is optional.

-D*name***[**=*value***]**
The values of environment variables referenced in the configuration file. By default the configuration exporter ignores the values of environment variables defined in the shell in which the configuration exporter runs. (See the the `-qcheckshell` option below.) If the `-D`*name* option is used more than once for the same option, the last occurrence takes precedence. The value of the environment variable from the `-D`*name* option overrides the value from the command shell.

-O *file* The name of the makefile generated by the configuration exporter. The default name is determined by the value specified with the `-qtargetos` option.

When `-QTargetos=`	The name of the makefile for `-O` is
AIX	Makefile
NT	makefile
OS2	makefile
WIN95	makefile

-QDependinc=*file*

Enables you to include a makefile that contains the list of dependencies between source files and header files. The include statement is only generated if this option is used. You would typically use a tool such as **makedep** on the target platform to build the correct dependency list.

-Q[NO]Gendepend

Indicates that the configuration exporter should add the dependencies found in the configuration file specified in the `-c` *codestore_file* option to the generated makefile. The dependency information from the last successful build is extracted from the codestore file and appended to the appropriate rule with `{%INCLUDE%}` in front of each file name, if the target operating system is not AIX. Another way to handle dependencies is to specify `-qdependinc` and generate a dependency file using a tool such a **makedep**.

-Q[NO]Checkshell

Causes the configuration exporter to check the command shell for the value of an environment variable if the variable is not defined with the `-D`*name* option. The default `-qnocheckshell` does not check the command shell for a value.

-QTargetos=(AIX | NT | OS2 | WIN95)
Specifies which operating system the generated makefile can be used on. This option defaults to the operating system that the project currently resides on.

-QTargetos=	To produce
AIX	UNIX and GNUmake-compatible makefile
NT	nmake for Windows NT-compatible makefile
OS2	nmake for OS/2-compatible makefile
WIN95	nmake for Windows 95-compatible makefile

configuration_file
The name of the configuration file.

Note that the minimum number of characters required to specify the options are shown in uppercase. The options, however, are not case sensitive. Some options are incompatible, in which case the first option specified takes precedence.

The Format of the Generated Makefile

The following list delineates the structure of the makefile generated by the configuration exporter:

```
all:run_before_rules \
  run_target_rules \
  run_after_rules

run_before_rules: <list of targets of the run_before directive>
```

run_target_rules: <list of all targets and default run directives>

run_after_rules: <list of targets of the run_after directive>

clean:
    <erase all files that appear as targets and intermediate files>
<**include** <dependency file>>

The makefile has the standard `all` target. Building the `all` target builds the project in a similar manner as it would be built in VisualAge C++ Version 4.0 with a configuration file. The following order is preserved:

- All run-before directives are built before the targets, the default run directive, or run-after directives in the order they are seen
- All targets and default run directives are built in the order they are seen
- The run-after directives are the last to be built (in the order they are seen)
- The order of execution of run directives is determined by their order of appearance in the configuration file

Restrictions and Limitations

The makefile created by cfgexprt does not reproduce all of the specifications defined in the configuration file used as the source for the conversion process. The following list gives you an indication of some of the items that need to be addressed manually in the makefile:

- Command variants based on .cfg files are not supported by **cfgexprt**. Any options specified in variants are individually converted.
- Option relationships, where two options translate into one option in the makefile, are not supported. Options are mapped on a one to one basis between the configuration file and the makefile.
- Macros (environment variables) specified in the configuration file are not converted. Specify the values of any macros in the makefile either with `-D`, or in the local environment and by specifying `-qcheckshell`.
- `before` or `after` must be specified for all run directives if sequencing is required. Otherwise, run directives are converted in their order of appearance in the configuration file and they are run in their order of appearance in the makefile.
- Source-target sequencing of run directives is not supported in the conversion.
- The literals `run_before_rules`, `run_target_rules`, and `run_after_rules` cannot be specified as targets in the configuration file or the makefile will not be generated.
-

cfgexprt may generate excessive warnings if your option group does not apply to a specific source, even though it applies to other sources. For example:
`option link(debug) {source 'a.cpp', 'a.rc'}`
generates a warning about not being able to find a mapping for `link(debug)` for source `a.rc`.

Configuration File Directives Mapped to Makefile Rules

When you run the configuration exporter (**cfgexprt**), target, source, and run directives in the configuration file are mapped to targets, dependencies, and commands in the generated makefile. The following table shows configuration file directives and equivalent makefile rules.

Configuration file directive	Equivalent makefile rule
run (before|after) command	run_before_rules: <previous run before targets> run_before__X run_before__X: command
run command	run_target_rules: <previous targets> run__X run__X: command
run source('gram.y') command	run_target_rules: <previous targets> run__X run__X: gram.y command
run target('ytab.h') source('gram.y') command	run_target_rules: <previous targets> run__X run__X: ytab.h ytab.h: gram.y command

target 'a.out' {
source 'a.C'
}
	run_target_rules: <previous targets> a.out
	a.out: a.o
	link command
	a.o: a.C
	compile command

target 'a.out' {
source 'b.o'
}
	run_target_rules: <previous targets> a.out
	a.out: b.o
	link command

target 'a.out' {
source 'a.C', 'b.o'
}
	run_target_rules: <previous targets> a.out
	a.out: a.o b.o
	link command
	a.o: a.C
	compile command

target 'a.o' {
source 'a.C'
}
	run_target_rules: <previous targets> a.o
	a.o: a.C
	compile command

source 'a.rc'	run_target_rules: <previous targets> a.res a.res:a.rc irc command

Using Parts from Previous Versions of VisualAge

To continue to use parts from Versions 3.0 and 3.5, perform these steps as required by your application:

- Make sure any member function signatures with input arguments specify both a type and a name for each argument. (Previously, you only had to specify a type.) If you run the migration utility on a part file in which only types are specified, VisualAge defines the name and type to be the same (for example, IDate and IDate).
- Run the migration utility on your part files (.vbb and .vbe). To do this, type the following, substituting the names of your files for the placeholders below:

  ```
  ivb -vbmigrate mypart1.vbb mypart2.vbe @mypart.lst
  ```

  As shown in the example, you can simply list the files separated by a space or create a list file. If you use the list-file method, the file name must be preceded by an at-sign (@). If you do not specify a file name, a file selection window appears.
- Clean up migrated parts that contain connections between the actionResult feature and a connection parameter. For more information, see online help for CPP.IMP.501.w.
- Any connections from promoted or embedded *ready* events are deleted by the migration utility, because they are no longer permitted. Add alternative connections as necessary.
- Edit handwritten code for updated features and changes in generated code.

Using Handwritten Code from Previous Versions of VisualAge

If your application contains user-written code from Versions 3.0 or 3.5, you might have to update it so that it meshes properly with generated code. In addition, several key member functions are being marked obsolete, so although you are not required to update your code in this release, you might want to do it anyway. A general list of changes that affect user-written code follows.

- Several IBM Open Class part features have been changed, removed, or renamed. Consider reviewing part documentation for changes that affect your application. In particular, template support has been added, affecting the interfaces of the ICollectionViewComboBox, ICollectionViewListBox, and IVBContainerControl parts.
- Most embedded parts are now instantiated in an implementation class separate from the composite class. As a result, any user-written code referring to these embedded parts must be updated.

  For example, suppose member function or custom logic code sets the label of an IPushButton* part, named MyButton, that is embedded into a part called MyPart. The current code looks something like this:

  ```
  iMyButton -> setText('Update');
  ```

  Generated code for the push button now appears in an implementation class associated with MyPart called fImpl, so your code needs to look like this:

  ```
  fImpl -> iMyButton() -> setText('Update');
  ```
- If your handwritten code calls initializePart( ), remove the call. This function is no longer used to enable notification and signal the *ready* event.
- If the name of a composite is the same as that of its primary part, VisualAge changes the name of the primary part to force uniqueness. It does this by appending the word *Part* to the end of the name. For example, a primary part called MyWindow is renamed automatically to MyWindowPart. However, any user-written code referring to this part must be updated manually.
- Use of the INotificationEvent class for passing event data in the notification signal is being phased out. Its replacement, INotificationEventFor, is a template, enabling support for event data other than pointers. Instances of INotificationEvent commonly appear in generated feature code for attributes; if you have modified the code to pass event data, consider updating the call to use the new class.

  The current event signal looks something like this:

  ```
  notifyObservers(INotificationEvent(SomeClass::someId, *this, true, (void*)&someEventData));
  ```

  The preferred form for an event signal now looks something like this:

```
notifyObservers(INotificationEventFor<someEventDataType>(IInterest(*this, someId), someEventData) );
```

Migrate Existing Code

Binary compatibility of this release of Data Access class library and generated code with previous releases is not guaranteed. If you want to use Data Access applications developed with previous releases with this release's runtime classes and generated Data Access classes, you must do the following things:

1. Generate new Data Access classes by importing the Data Access Builder session used for the original application's Data Access classes and regenerating. This release of VisualAge runtime classes will not work with Data Access classes from previous releases.
2. Re-compile and link the old application source with the new Data Access classes. The old application executable will not work with the new class library.

Since your old application will not have taken advantage of any of Data Access Builders new features, you will have to add code to the application if you want to use functionality such as stored procedure calls and large object access.

Migration Considerations

Saved sessions from the previous releases of Data Access Builder can be loaded into the new release. However, there are differences that you should be aware of:

1. Visual Builder (.vbe) files are no longer generated. When Data Access Builder generates code, it will always generate parts that are usable for visual composition. These parts are loaded as part of the project.
2. SOM (.idl) classes are no longer generated or supported with this release of the product.
3. You must create a project in the IDE to compile and run the application that uses the Data Access Builder generated code.

You do not have to modify existing C++ applications that use the generated code from the last release of our product. However, to use the new Data Access Class Library, you have to import the session (.dax files) used by your C++ applications and regenerate the source code. Then you have to recompile and bind these applications before using them with the new class library.

By following the instructions on importing Data Access Builder sessions, you can import .dax files from the following VisualAge Products:

- VisualAge C++ for OS/2 v. 3.0

- VisualAge C++ for Windows v. 3.5
- VisualAge for Java for OS/2 or Windows v.1.0; **Note:** Only schema information (that is, the database, tables, and views used) is carried over. Information related to the mapping (for example, user-defined methods and attribute type mapping) is language-specific and must be re-created once the .dax file is imported to this C++ version.

AIX Data Access Builder cannot import .dax files from VisualAge for C++ for OS/2 v. 3.0 because they are binary encoded. Only ASCII text .dax files can be imported.

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'
   }
}
```

Object Files Created in Previous VisualAge C++ Releases

You can link object files created with VisualAge C++ Version 4.0 with those created in previous releases of VisualAge C++. However, old and new object files must be compiled with the same options:

- If you use the link option `link(linkwithsharedlib,yes)` in the configuration file, to link with the shared run-time library, you need to have compiled objects with the `-Gd+` compiler option in the previous releases, to use a shared run-time library.
- If you use the link option `link(linkwithsharedlib,no)` in the configuration file, to statically link the run-time library, you need to have compiled objects with the `-Gd-` compiler option in the previous releases, to not use a shared run-time library.

Use DLLs and Executable Files Created in Previous VisualAge C++ Releases

You can mix DLLs and executable files created in previous releases of VisualAge C++, with DLLs and executable files created in this release. All of the DLLs and executable files need to be created with a dynamically linked run time, but they cannot use IBM Open Class libraries. Since there is no one-to-one mapping of run-time libraries from previous releases to this release, you need to use the DLLRNAME command to change to the run-time libraries available in this release.

A set of C and C++ run-time migration DLLs have been provided for this purpose. They provide the same exports as the run-time libraries in the previous releases. For a given DLL, for example appl.dll, use DLLRNAME to convert to the appropriate C and C++ migration DLLs:

For Windows NT:

Single Thread	dllrname appl.dll cppws35i=cppzsi40
Multi-Thread	dllrname appl.dll cppwm35i=cppzmi40
Subsystem	dllrname appl.dll cppwn35i=cppzni40

For OS/2:

Single Thread	dllrname appl.dll cppos30=cppzs40
Multi-Thread	dllrname appl.dll cppom30=cppzm40
Subsystem	dllrname appl.dll cppon30=cppzn40

DLLs and Executable Files Created in Previous VisualAge C++ Releases

If you built DLLs with VisualAge for C++ for Windows or VisualAge C++ for OS/2, and you did not use IBM Open Class Libraries, then it is possible to reuse the DLLs. To do so, you must use the C and C++ run-time libraries in your application.

The change is required for two reasons:

- There have been subtle changes in the C++ run-time function related to atexit() processing (to conform to the new C++ ANSI specification).
- Two USL Iostream libraries are no longer combined with the C and C++ run-time libraries. The two flavors of the Stream Library are the old USL library, and the new ANSI library.

A set of C and C++ run-time migration DLLs are available that provide all the entry points that were present in the previous Windows and OS/2 versions of the product. The migration DLLs maintain the pre-ANSI atexit() processing behavior, they handle the split of the USL Iostream library function, and they call the same C and C++ run-time libraries as code that is generated when you use this product.

Create Module Definition Files (OS/2, Windows)

VisualAge C++ does not use module definition (.def) files in the same way that earlier versions did.
OS/2 To build a DLL with previous versions of VisualAge C++ for OS/2 3.0 and IBM C and C++ Compilers 3.6, you had to create a module definition file and supply it to the linker.
WIN In previous versions of VisualAge for C++ for Windows version 3.5 and IBM C and C++ Compilers 3.6, supplying a module definition file was optional.

The primary purpose of a .def file was to list the symbols imported or exported by the module being built. You could also specify other options that affected how the linker created the target module. Module definition files are no longer required. Generally, you should have no reason to create one.

However, you may want a .def file as an aid in migrating DLLs from earlier versions of VisualAge C++, or to extract from an existing DLL a list of the exported symbols.
C++ These files may be useful because they show mangled C++ symbol names.

WIN The names are also decorated on Windows. The **ilib** utility cannot determine the calling convention of functions in a library from by looking at the library file alone, because the linkage is specified only in the source code or in a .def file. Therefore, it assumes all exported functions have the default calling convention, _Optlink.

You can use the **ilib** command to create a module definition file. Follow the steps listed below:

1. Have an existing DLL or object modules that were generated from C or C++ sources that export symbols explicitly, or a combination of objects modules and a DLL.
2. In a VisualAge C++ command line window, open a working directory.
3. Issue the **ilib** command. Use the /gendef:*fileName* option. If you omit the file name, the module definition file has the same base name as the first input file and the extension .def. The short form /gd is equivalent to /gendef. Also list the input DLL or object modules on the command line.

For example, the following command creates the module definition file winner.def from the DLL winner.dll. If a file named winner.def already exists, it is replaced.

```
ilib /gd:winner.def winner.dll
```

Convert Module Definition Files (OS/2, Windows)

If you are migrating projects from earlier versions of VisualAge C++ and predecessor products, you should eliminate any module definition files that you previously used during builds. Instead, specify the options that were set in .def files by setting build options or specifying your target in your configuration file.

The major effort of eliminating your old module definition files is likely to be in specifying exports and imports for DLLs. VisualAge C++ provides alternative ways to export and import symbols. Usually the process of building, creating, or using DLLs is simpler than it was before.

Here is a summary of the statements that made up module definition files in VisualAge C++ 3.0 or 3.5, CSet++ 2.0 or 2.1, and IBM C and C++ Compilers 3.6. The statements are listed in alphabetic order and each is followed by the

statement syntax, a brief description, and how to convert the statement for the VisualAge C++ build process. If the statement no longer applies to builds, the conversion is specified as 'Do not convert'. Optional parameters of the statements are enclosed in brackets, [and], and alternative items are separated with a vertical line, |.

BASE

syntax **BASE =** *address*
description Sets preferred load address.
conversion Use build option link(base, *address*).

OS/2 CODE

syntax **CODE [***attributes***]**
description Sets default attributes for code sections.
conversion Use build option link(code, *attributes*).

OS/2 DATA

syntax **DATA [***attributes***]**
description Sets default attributes for data sections.
conversion Use build option link(data, *attributes*).

DESCRIPTION

syntax **DESCRIPTION '***string*'
description Describes the target module.
conversion Use build option link(versionString, *string*)

OS/2 EXETYPE

syntax **EXETYPE [OS2 | WINDOWS | UNKNOWN]**
description Specifies target operating system.
conversion Do not convert.

EXPORTS

syntax

EXPORTS

OS/2

symbol **[=** *internal_name***] [@***ordinal***]**
[RESIDENTNAME | NONAME]]
[*numwords***]**

WIN

symbol **[=** *internal_name***] [@***ordinal***]**
[DECORATED | CONSTANT]]
[*numwords***]**

description

Lists symbols exported from DLL, each on a separate line.

syntax

EXPORTS

OS/2

symbol **[=** *internal_name***] [@***ordinal***]**
[RESIDENTNAME | NONAME]]
[*numwords***]**

WIN

symbol **[=** *internal_name***] [@***ordinal***]**
[DECORATED | CONSTANT]]
[*numwords***]**

conversion

Use one of the following techniques:

- Modify your source to export symbols explicitly:
- Use the following options either alone or in combination:
  - Use link(export, ...) to export specific symbols.
  - Create a file containing export specifications and build with link(exportList, *fileName*) .
- C++

  Build C++ modules with link(exportAll) to export all external symbols.
- C++

  Build C++ modules with link(export_NameSpace, *name*) to export all external symbols in the named namespace.
- C++

  Build C++ modules with link(export_Prefix, *string*) to export all external symbols that begin with the specified string.

This last three link options give you less control than link(export, ...) or link (exportList, ...) but are easier to use.

You can easily create the file of export specifications for use with link(exportList,*fileName*) by copying the export descriptions from the lines that follow the EXPORTS keyword in your module definition file into a new file.

WIN

There is no equivalent for the

DECORATED keyword in VisualAge C++.

HEAPSIZE

syntax OS/2

HEAPSIZE = [MAXVAL | *number*]
WIN

HEAPSIZE = *reserve [, commit]*

description Specifies the size of the local heap.

conversion OS/2

These is no need to covert. The equivalent build option is ignored.
WIN

Use build option link(heap, *reserve*[, *commit*])

OS/2 IMPORTS

syntax **IMPORTS**
[*internal_name* **=]** *DLLname.entrypoint*

description Lists imported functions by name or ordinal, each on a separate line.

conversion Supply an import library with a source directive or use build option link(import, ...).

LIBRARY

syntax OS/2

LIBRARY *fileName* **[INITGLOBAL | INITINSTANCE] [TERMGLOBAL | TERMINSTANCE]**
WIN

LIBRARY *fileName [* **BASE =** *address]*

description Names and sets type of target DLL.

conversion Use the directive target [dll] *fileName*.
Optionally also set build option link(base, *address*).
OS/2

Optionally also set OS/2 build options link(dllInitInstance) and link(dllTermInstance).

NAME

syntax OS/2

NAME *fileName* **[WINDOWAPI | WINDOWCOMPAT | NOTWINDOWCOMPAT]**
WIN

NAME *fileName [* **BASE =** *address]*

description Names and sets type of target executable file

conversion Use the directive target [exe] *fileName*.
OS/2

Optionally also set build option link(pmType, *type*).
WIN

Optionally also set build option link(base, *address*).

OS/2 OLD

syntax **OLD '[***directory***] DLL***name'*

description Preserves compatibility with old version of a DLL.

conversion Do not convert.

OS/2 PHYSICAL DEVICE

syntax **PHYSICAL DEVICE** *fileName* **[INITGLOBAL | INITINSTANCE] [TERMGLOBAL | TERMINSTANCE]**

description Names and sets type of target physical device driver.

conversion Use the directive target [pdd] *fileName*.
Optionally also set OS/2 build options link(dllInitInstance) and link(dllTermInstance).

OS/2 ROBASE

syntax **ROBASE =** *address*

description Sets start address for read-only segments.

conversion Obsolete, do not convert.

OS/2 SEGMENTS

syntax **SEGMENTS ['***segname***'] [CLASS '***class***'] [***attributes***]**

description Sets attributes for the named segment.

conversion Use build option link(section, *name*, *attributes*).

STACKSIZE

syntax **STACKSIZE size**

description Sets size of the local stack.

conversion Use build option link(stack, *reserve*, *commit*).

STUB

syntax OS/2

STUB [NONE | '*fileName*']
WIN

STUB *fileName*

description Adds the named DOS stub.

conversion Use build option link(stub[, *fileName*]) or link(stub, no).

OS/2 VIRTUAL DEVICE

syntax **VIRTUAL DEVICE** *fileName* **[INITGLOBAL | INITINSTANCE] [TERMGLOBAL | TERMINSTANCE]**

description Names and sets type of target physical device driver.

conversion Use the directive target [vdd] *fileName*. Optionally also set OS/2 build options link(dllInitInstance) and link(dllTermInstance).

VERSION

syntax *VERSION 'file number'*

description Adds version string to executable.

conversion Use build option link(version, *major*[, *minor*])

Compile Options from Earlier Versions of VisualAge C++ (OS/2, Windows)

VisualAge C++ includes both an incremental compiler that replaces the traditional C++ compiler supplied in earlier versions of the product, and a C compiler. The old form of options are no longer used. For both C and C++, you control the compile and link stages of build by specifying options in a configuration file.

The syntax of configuration file options indicates whether the option relates to code generation, optimization, the C or C++ language, the link stage, or other features of builds.

The table below gives the configuration file option equivalent to compile options from earlier versions of VisualAge C++ and predecessor products. In an incremental C++ compile, not all the old options are meaningful, some are automatic, and the functionality of others is provided in a different way. An entry of N/A in the table indicates that an equivalent new option is not

available. In this version of VisualAge C++, some options may apply to only one of C or C++ code, but not both. Option applicability is indicated in the description column.

In the option syntax, the characters [and] enclose optional items, and the character | indicates alternatives. For example, /Fm[+|-] is equivalent to /Fm+, /Fm-, or /Fm.

Old Option	Description	Configuration File Option
/?	Display help	N/A
/B'*options*'	Pass options to linker	link options
/C[+|-]	Perform compile without linking	N/A
/D*name* D*name*=[*value*]	Define preprocessor macro.	define(*macro_name*[, *string*])
/Fa[+|-][*dir*][*name*]	C Produce assembler listing	file(genAsm[, yes]) file(genAsm, no) file(asmFileName, *String*)
/Fb[+|-|*]	Produce a browser file	N/A
/Fc[+|-]	C Perform syntax check only, instead of a full compile.	file(syntaxOnly[, yes]) file(syntaxOnly, no)
/Fe*name*	Specify name of executable output file	target [exe] *fileName*
/Fi[+|-][*dir*][*name*]	C Produce, name, and direct precompiled header file	file(genPCH[, yes]) file(genPCH, *fileName*) file(genPCH, no)
/Fl[+|-][*dir*][*name*]	C Produce, name, and direct listing file	list(listing[, yes]) list(listing, no) list(fileName, *string*)
/Fm[+|-] /Fm*name*	Produce a linker map file	link(map[, yes]) link(map, no) link(map, *name*)
/Fn[+|-] /Fn[dir][name]	C Produce assembler listing with code	file(genAsmWithCode[,yes]) file(genAsmWithCode, *fileName*) file(genAsmWithCode, no)

Old Option	Description	Configuration File Option
/Fo[+|-][*dir*][*name*]	C Control and name object file	file(genObject[, yes]) file(genObject, *fileName*) file(genObject, no)
/Ft[+|-][*dir*]	Control and direct files for template resolution	N/A
/Fw[+|-][*dir*][*name*]	Create intermediate code files only	N/A
/G3 /G4 /G5	Specify the processor for which the code is optimized	opt(tune, *processor*)
/Ga	Turn on implicit SOM mode	N/A
/Gb	Disable direct access to attributes for DSOM	N/A
/Gd[+|-]	Dynamically link to the runtime library, instead of linking statically	link(linkWithSharedLib[, yes]) link(linkWithSharedLib, no)
/Ge[+|-]	Build .exe or .dll file	target [*type*] *fileName*
/Gf[+|-]	Use fast floating-point execution.	opt(float[, yes]) opt(float, no)
/Gh[+|-]	Enable code for performance analysis	gen(profile[, yes]) gen(profile, no)
/Gi[+|-]	Use fast integer execution	opt(integer[, yes]) opt(integer, no)
/Gk[+|-]	Support objects generated with C Set ++ and containing templates	N/A
/Gl[+|-]	Remove unreferenced functions	gen(unreferenced[, yes]) gen(unreferenced, no)
/Gm[+|-]	Link with the multithread library, instead of the single-thread library	link(linkWithMultiThreadLib[, yes]) link(linkWithMultiThreadLib, no)
/Gn[+|-]	Hide default library information from linker	link(defaultLibs[, yes] link(defaultLibs, no)
/Gp[+|-]	Support parmdwords	gen(parmDWord[, yes] gen(parmDWord, no)
/Gr[+|-]	Generate code to run on ring 0	gen(ring0[, yes] gen(ring0, no)

Old Option	Description	Configuration File Option
/Gs[+|-]	Support stack probes	gen(probe[, yes] gen(probe, no)
/Gt[+|-]	Support tiled memory for 16-bit code	gen(tiledMemory[, yes] gen(tiledMemory, no)
/Gu[+|-]	Stop external functions from using data defined in intermediate files	N/A
/Gv[+|-]	C Perform special handling of ES and DS registers	gen(loadDS[, yes] gen(loadDS, no)
/Gw[+|-]	C Generate FWAIT instruction after each floating-point load instruction	N/A
/Gx[+|-]	C++ Enable C++ exception handling	gen(eh[, yes] gen(eh, no)
/Gy[+|-]	Decorate all external data names with leading underscore	gen(decorateExternalNames[, yes] gen(decorateExternalNames, no)
/Gz[+|-]	Prebuild SOM classes	N/A
/H*num*	Set maximum length of external names	C link(nameLength, *bytes*) C++ link(nameMangling, *ansi*, *size*) C++ link(nameMangling, *compat*, *size*)
/I*path*[;*path*]	Specify #include search paths	incl(searchPath, *path*)
/J[+|-]	Treat unspecified char variables as signed char	lang(signedChars[, yes] lang(signedChars, no)

Old Option	Description	Configuration File Option
/L[+|-]	C Produce a minimal listing file	list(listing[, yes] list(listing, no)
/La[+|-]	C Include a minimal layout in the listing file	list(refStructVar[, yes] list(refStructVar, no)
/Lb[+|-]	C Include a layout in the listing file	list(allStructVar[, yes] list(allStructVar, no)
/Le[+|-]	C Expand macros in the listing file	list(expMacros[, yes] list(expMacros, no)
/Lf[+|-]	C Set all listing options on.	list(full[, yes] list(full, no)
/Li[+|-]	C Expand user #include files in the listing file	list(expusrinc[, yes] list(expusrinc, no)
/Lj[+|-]	C Expand user and system #include files in the listing file	list(expAllInc[, yes] list(expAllInc, no)
/Lp*number*	C Set page length of listing file	list(pageLen, *number*)
/Ls[+|-]	C Include the source code in the listing file	list(incSource[, yes] list(incSource, no)
/Lt'*string*'	C Set title string for the listing file	list(title, *string*)

Old Option	Description	Configuration File Option
/Lu'*string*'	C Set subtitle string in the listing file	list(subTitle, *string*)
/Lx[+|-]	C Generate a minimal cross-reference table in the listing file	list(minXRef[, yes] list(minXRef, no)
/Ly[+|-]	C Generate a cross-reference table in the listing file	list(fullXRef[, yes] list(fullXRef, no)
/Mp /Ms /Mc /Mt	Set default calling convention	gen(call, optlink) gen(call, system) gen(call, cdecl) gen(call, stdcall)
/N*number*	C End compilation when error count reaches *number*	debug(maxerror, *number*)
/Nd*name*	Set names of default data, uninitialized data, and constant segments	gen(dataSeg, *name*) gen(uninitSeg, *name*) gen(constSeg, *name*)
/Nt*name*	Set names of default code or data segment	gen(codeSeg, *name*)
/Nx*name*	C++ Set names of exception-handling code or data segments	gen(ehCodeSeg, *name*) gen(ehDataSeg, *name*)
/O[+|-|2|3|4]	Optimize code for speed.	opt(level, *level*)
/Oc[+|-]	Optimize code for size.	opt(size[, yes]) opt(size, no)
/Oi[+|-]	Inline specified user functions	opt(inline[, yes]) opt(inline, no) opt(autoInline[, yes]) opt(autoInline, no)
/Ol[+|-]	Use intermediate code linker	N/A

Old Option	Description	Configuration File Option
/Om[+|-]	C Limit working set size	opt(limitWorkingSet[, yes]) opt(limitWorkingSet, no)
/Op[+|-]	Perform optimizations that involve the stack pointer	opt(stackPointer[, yes]) opt(stackPointer, no)
/Oq[+|-]	Control stack chaining.	opt(stackChaining[, yes]) opt(stackChaining, no)
/Os[+|-]	Invoke the instruction scheduler	opt(schedule[, yes]) opt(schedule, no)
/P[+|-]	C Run the preprocessor only	pp(preprocessOnly[, yes]) pp(preprocessOnly, no)
/Pc[+|-]	C Preserve source code comments in preprocessor output.	pp(preserveComments[, yes]) pp(preserveComments, no)
/Pd[+|-]	C Redirect preprocessor output to stdout	pp(stdout[, yes]) pp(stdout, no)
/Pe[+|-]	C Suppress #line directives in preprocessor output	pp(suppressLine[, yes]) pp(suppressLine, no)
/Q[+|-]	C Suppress the compiler logo when invoking the compiler	N/A
/qalias	Specify which aliasing rules can be used during optimization	opt(alias, allp) opt(alias, typ) opt(alias, addr) opt(alias, ansi)
/qarch	Specify target CPU architecture	opt(arch, *CPUtype*)
/qautoimported /qnoautoimported	WIN Import all external references	gen(import[, yes]) gen(import, no)

Old Option	Description	Configuration File Option
/qautothread /qnoautothread	WIN C Turn static, nonconstant data into thread-local data	N/A
/qbitfields=signed /qbitfields =unsigned	Specify default sign of bitfields	gen(signedBitfields[, yes]) gen(signedBitfields, no)
/qdbgunref /qnodbgunref	Use referenced and unreferenced symbols during debugging	gen(debugUnreferenced[, yes]) gen(debugUnreferenced, no)
/qdigraph /qnodigraph	Permit standard digraph and keyword operators	C lang(cDigraphs[, yes]) C lang(cDigraphs, no) C++ lang(digraphs[, yes]) C++ lang(digraphs, no)
/qignerrno	Tell the optimizer that the program will never refer to or set errno	lang(ignerrno[, yes]) lang(ignerrno, no)
/qignprag=disjoint /qignprag=isolated /qignprag=all	Ignore references to #pragma disjoint and #pragma isolated_call	opt(pragmaDisjoint[, yes]) opt(pragmaDisjoint, no) opt(pragmaIsolatedCall[, yes]) opt(pragmaIsolatedCall, no)
/qinitauto	Initialize automatic storage	gen(initAuto[, yes]) gen(initAuto, *number*) gen(initAuto, no)
/qipa	C Use interprocedure analysis for optimization	N/A
/qisolated_call =*fname*[:*fname*]...	List functions that do not change data objects at time of function call	opt(isolatedCall, *string*)

Old Option	Description	Configuration File Option
/qlibansi /qnolibansi	Process standard C library names as system functions	gen(libANSI[, yes]) gen(libANSI, no)
/qlonglong /qnolonglong	Process or disallow type long long int	lang(longLong[, yes]) lang(longLong, no)
/qmakedep	C Create dependency files to include in makefile	file(makeDep[, yes])
/qro /qnoro	Use read-only, or read-write storage for string literals	gen(readOnly[, yes]) gen(readOnly, no)
/qrtti=*rtoption* /qnortti	Generate information for run-time type information	N/A
/qsomvolattr /qnosomvolattr	Use volatile on attribute prototypes	N/A
/qtune=*option*	Specify the processor for which code is optimized	opt(tune, *processor*)
/qwin32s /qnowin32s	WIN C Help port your program to the Win32s runtime environment.	N/A
/R[e|n]	Generate code that can be used as a subsystem without a runtime environment.	link(standAlone[, yes]) link(standAlone, no)
/Sa /Se /S2	C Set language level.	lang(level, ansi) lang(level, extended) lang(level, saa12)
/Sc	Set C++ language level.	N/A
/Sd[+|-]	Set the default file extension for source files to .c, instead of .obj	N/A
/Sg[*left*][,*right*|*] /Sg[-]	C Set left and right margins for the input file	misc(margins[, yes]) misc(margins, no) misc(marginValues, left[, right])
/Sh[+|-]	C Allow use of ddNames	lang(ddNames[, yes]) lang(ddNames, no)

Old Option	Description	Configuration File Option
/Si[+|-] /Si[*dir*][file*name*]	C Use precompiled header files, if they exist and are current	file(usePCH[, yes]) file(usePCH, *fileName*) file(usePCHl, no)
/Sm[+|-]	C Ignore obsolete keywords, instead of treating them like any other identifier	lang(ignObsoleteKeywords[, yes]) lang(ignObsoletekeywords, no)
/Sn[+|-]	C Allow use of DBCS	lang(dbcs[, yes]) lang(dbcs, no)
/Sp[1|2|4|8|16]	Specify alignment or packing of data items within structures and unions	gen(pack, *number*)
/Sq[*left*][, *right*] /Sq[*left*][, *] /Sq[-]	C Ignore sequence numbers in source columns	misc(sequenceNumbers[, yes]) misc(sequenceNumbers, no) misc(sequenceNumberValues, *left*[, *right*])
/Sr[+|-]	C Use old-style rules for type conversion	lang(preserveUnsignedPromotion[, yes]) lang(preserveUnsignedPromotion, no)
/Ss[+|-]	C Allow double-slash comments	lang(slashSlashComments[, yes]) lang(slashSlashComments, no)
/Su[+|-|1|2|4]	Control size of enum variables	gen(enumSize, int) gen(enumSize, small) gen(enumSize, *bytes*)
/Sv[+|-]	C Allow use of memory files	lang(memoryFiles[, yes]) lang(memoryFiles, no)
/Tc*filename*	Compile as a C source file, regardless of extension	N/A
/Td[c|p]	Specify the default language for sourc files, overriding extension	N/A

Old Option	Description	Configuration File Option
/Ti[+|-]	C Generate debugger information.	gen(debug[, yes]) gen(debug, no)
/Tm[+|-]	Enable debug version of memory management functions	alloc(debug[, yes]) alloc(debug, no)
/Tn[+|-]	C Generate line number information for debugging	gen(lineOnlyDebug[, yes]) gen(lineOnlyDebug, no)
/Tp*filename*	Compile as a C++ source file, regardless extension	N/A
/Tx[+|-]	Provide a complete machine-state dump when an exception occurs, instead of providing only the exception message and address	gen(dump[, yes]) gen(dump, no)
/U*name* /U*	Undefine macros	undefine(*macro_name*)
/V'*string*'	Include a version string in the object and executable files	link(versionString, *string*)
/W0 /W1 /W2 /W3	Set minimum severity level of messages the compiler produces and counts	report(level, E) report(level, W) report(level, I)
/W*group*[+|-][*group*]	C Generate or suppress messages in a group	info(group[, yes]) info(group, no) info(all[, yes]) info(all, no)
/Xc[+|-]	C Do not search paths specified using /I	file(ignoreusrincpaths[, yes]) file(ignoreusrincpaths, no)
/Xi[+|-]	C Do not search paths specified in INCLUDE	file(ignoreEnvIncPaths[, yes]) file(ignoreEnvIncPaths, no)

Link Options from Earlier Versions of VisualAge C++ (OS/2, Windows)

VisualAge C++ includes an incremental linker that replaces the traditional linker supplied in earlier versions of the product. Much of the same link functionality is provided, but you specify options in the configuration file for your project rather than as command line options for the linker.

The table below gives the configuration file option equivalent to link options from earlier versions of VisualAge C++ and predecessor products. In an incremental build, not all the old options are meaningful, some are automatic, and the functionality of others is provided in a different way. An entry of N/A in the table indicates that an equivalent new linker option is not available.

In the option syntax, the characters [and] enclose optional items, and the character | indicates alternatives. For example, /E[:1|:2] is equivalent to /E:1, /E:2, or /E.

Old Syntax	Description	Configuration File Option
/?	Display help	N/A
/ALIGNA[DDR]:*number*	Set address alignment for segments	link(alignAddr, *number*)
/A[LIGNFILE]:*number*	Set file alignment for segments	link(alignFile, *number*)
/A[LIGNMENT]:*factor*	Set alignment factor	link(alignFile, *number*)
/BAS[E]: <*address*|*@filename*, *key*>	Set preferred loading address	link(base, *address*)
/BR[OWSE] /NOBR[OWSE]	Add browse information	N/A
/CODE: *attributes*	Set section attributes for code	link(code, *attributes*)
/C[ODEVIEW] /NOC[ODEVIEW]	Include traditional debug information	link(debug,[yes]) link (debug, no)
/DATA: *attributes*	Set section attributes for data	link(data, *attributes*)
/DB[GPACK] /NODB[GPACK]	Pack debug information	N/A
/D[EBUG] /NODEB[UG]	Support traditional debugger	link(debug,[yes]) link (debug, no)

Old Syntax	Description	Configuration File Option
/DEF[AULTLIBRARYSEARCH][:*library*] /NOD[EFAULTLIBRARYSEARCH][:*lib*]	Search default libraries	link(defaultLibs[, yes]) link(defaultLibs, no) link(defaultLibsName, *library*) link(noDefaultLibsName, *library*)
/DLL	Generate DLL	target *fileName*.dll target [dll] *fileName*
/EN[TRY]:*symbol*	Specify an entry point in an executable file	link(entry, *symbol*)
/EXEC[UTABLE]	Generate .exe file	target *fileName*.exe target [exe] *fileName*
/E[XEPACK][:1|:2]	Compress exectuable	link(exePack[, yes]) link(exePack, 1) link(exePack, 2) link(exePack, no)
/EXT[DICTIONARY] /NOE[XTDICTIONARY]	Use extended dictionary to search libraries	link(extDictionary[, yes]) link(extDictionary, no)
/FI[xed] /NOFI[xed]	Do not relocate the file in memory	link(fixed[, yes]) link(fixed, no)
/FO[RCE] /NOFO[RCE]	Create target even if errors are detected	link(force[, yes]) link(force, no)
/HEA[P]: *reserve* [,*commit*]	Set size of program heap	link(heap, *number* [, *number*])
/H[ELP]	Display help	N/A
/IG[NORECASE] /NOI[GNORECASE]	Set case sensitivity	link(ignoreCase[, yes]) link(ignoreCase, no)
/INC[lude]	Force a reference to a symbol	link(include, *symbol*)
/I[NFORMATION] /NOIN[FORMATION]	Display status of linking process	N/A
/L[INENUMBERS] /NOLI[NENUMBERS]	Include line numbers in map file	link(lineNum[, yes]) link(lineNum, no)
/LO[GO] /NOL[OGO]	Display logo and echo response file	N/A
/M[AP][:*name*] /NOM[AP]	Generate map file	link(map[, yes]) link(map, *fileName*) link(map, no)
/OLD[CPP] /NOLD[CPP]	Support templates from previous versions	N/A

Old Syntax	Description	Configuration File Option
/OPTF[UNC] /NOOPTF[UNC]	Remove unreferenced functions	link(optFunc[, yes]) link(optFunc, no)
/O[UT][:*name*]	Name output file	target [*type*] *fileName*
/PACKC[ODE][:*number*] /NOP[ACKCODE]	Pack code segments	N/A
/PACKD[ATA][:*number*] /NOPACKD[ATA]	Pack data segments	N/A
/PDD	Generate a .pdd file	target *fileName*.pdd target [pdd] *fileName*
/PM[TYPE]:*type*	Specify OS/2 application type	link(pmType, *type*)
/SEC[TION]:*name*, *attributes*	Set attributes for section	link(section, *name*, *attributes*)
/SE[GMENTS]:*number*	Set maximum number of segments	N/A
/ST[ACK]:*reserve* [,*commit*]	Set stack size of application	link(stack, *reserve* [, *commit*])
/STU[B]:*filename*	Specify the name of the DOS stub file	link(stub, *string*) link(stub, no)
/SU[BSYSTEM]:*subsystem* [,major[.minor]]	Specify the required subsystem and version	link(subsystem, *string*[, *number*[, *number*]])
/VERB[OSE] /NOV[ERBOSE]	Display status of linking process	N/A
/VDD	Generate a .vdd file	target *fileName*.vdd target [vdd] *fileName*
/VER[SION]:*major* [.*minor*]]	Write a version number in the run file	link(version, *number*[, *number*])

C and C++ Binary and Run-Time Interoperability with Previous Releases

Object Files and Interoperability

The build process uses the new run-time libraries for all object files that are being linked, if the entry points and interfaces have not changed between releases. However, you should not mix object files that use a shared run-time library, with object files that do not. See 'Object Files Created in Previous VisualAge C++ Releases' for information about handling object files.

DLLs, Executable Files, and Interoperability

You can use DLLs and executable files created with previous VisualAge C++ releases, with DLLs and executable files created in this release. All of the DLLs and executable files need to be compiled with a shared run-time library. However, the run-time libraries used by DLLs and executable files compiled in a previous release need to use the run-time libraries in this release. See 'Use DLLs and Executable Files Created in Previous VisualAge C++ Releases' for information about using DLLRNAME to forward references to the new run-time libraries.

Some restrictions apply when you use two run-time libraries. The restrictions are the same as in previous releases:

- You cannot use memory management across run-time environments. For example, if you call malloc to allocate storage in one run-time environment, you cannot call free or realloc in a different run-time environment.
- You cannot use exception handling across run-time environments. For example, you cannot call throw in one environment and call catch in another.
- You cannot use I/O across run-time environments. For example, if you open a file or a stream in one run-time environment, you cannot close it in another.
- You cannot use putenv or setenv across run-time environments.

Special Considerations When Using C++ I/O

The ANSI C++ I/O library provided in this release, and the AT&T I/O library do not share streams and therefore cannot be used together on the same stream. However, they can be used in the same application on different streams. Also, the ANSI library and the C++ library can coexist in the same run-time DLL because entry points in the ANSI library are all in the std namespace. If you use streams, note that the IBM Open Class library also uses ANSI C++ I/O. Therefore you cannot use the IBM Open Class library if you need to coexist with objects or libraries used in previous releases that use iostream.

Previous VisualAge Run-Time Libraries and Version 4.0 Equivalents

The following tables show the names of the libraries used in previous releases of VisualAge C++ and the equivalent libraries used in VisualAge C++ Version 4.0.

VisualAge C++ for OS/2, Version 3.0

Single thread library	CPPOS30.DLL	CPPRSI40.DLL (Single thread C library) CPPRAI40.DLL (Single thread C++ library)
Multi-thread library	CPPOM30.DLL	CPPRMI40.DLL (Multi-thread C library) CPPRBI40.DLL (Multi-thread C++ library)
Subsystem library	CPPON30.DLL	CPPRNI40.DLL (C subsystem library) CPPREI40.DLL (C++ subsystem library)
Single thread import library	CPPOS30I.LIB	CPPRSI40.LIB (Single thread C import library) CPPRAI40.LIB (Single thread C++ import library)
Multi-thread import library	CPPOM30I.LIB	CPPRMI40.LIB (Multi-thread C import library) CPPRBI40.LIB (Multi-thread C++ import library)
Subsystem import library	CPPON30I.LIB	CPPRNI40.LIB (C Subsystem import library) CPPREI40.LIB (C++ Subsystem import library)
Single thread static library	CPPOS30.LIB	CPPRSS40.LIB (Single thread C static library) CPPRAS40.LIB (Single thread C++ static library)
Multi-thread static library	CPPOM30.LIB	CPPRMS40.LIB (Multi-thread C static library) CPPRBS40.LIB (Multi-thread C++ static library)
Subsystem static library	CPPON30O.LIB	CPPRNS40.LIB (C Subsystem static library) CPPRES40.LIB (C++ Subsystem static library)
Single thread object library	CPPOS30O.LIB	CPPRSO40.LIB (Single thread C object library) CPPRAO40.LIB (Single thread C++ object library)

Multi-thread object library	CPPOM30O.LIB	CPPRMO40.LIB (Multi-thread C object library) CPPRBO40.LIB (Multi-thread C++ object library)
Subsystem object library	CPPON30O.LIB	CPPRNO40.LIB (C Subsystem object library) CPPREO40.LIB (C++ Subsystem object library)
Single thread complex number library	CPPOOX3.LIB	CPPRSY40.LIB (C++)
Multi-thread complex number library	CPPOOY3.LIB	CPPRMY40.LIB (C++)

VisualAge for C++ for Windows, Version 3.5

Single thread library	CPPWS35I.DLL	CPPRSI40.DLL (Single thread C library) CPPRAI40.DLL (Single thread C++ library)
Multi-thread library	CPPWM35I.DLL	CPPRMI40.DLL (Multi-thread C library) CPPRBI40.DLL (Multi-thread C++ library)
Subsystem library	CPPWN35I.DLL	CPPRNI40.DLL (C subsystem library) CPPREI40.DLL (C++ subsystem library)
Single thread import library	CPPWS35I.LIB	CPPRSI40.LIB (Single thread C import library) CPPRAI40.LIB (Single thread C++ import library)
Multi-thread import library	CPPWM35I.LIB	CPPRMI40.LIB (Multi-thread C import library) CPPRBI40.LIB (Multi-thread C++ import library)
Subsystem import library	CPPWN35I.LIB	CPPRNI40.LIB (C Subsystem import library) CPPREI40.LIB (C++ Subsystem import library)

Single thread static library	CPPWS35.LIB	CPPRSS40.LIB (Single thread C static library) CPPRAS40.LIB (Single thread C++ static library)
Multi-thread static library	CPPWM35.LIB	CPPRMS40.LIB (Multi-thread C static library) CPPRBS40.LIB (Multi-thread C++ static library)
Subsystem static library	CPPWN35.LIB	CPPRNS40.LIB (C Subsystem static library) CPPRES40.LIB (C++ Subsystem static library)
Single thread object library	CPPWS35O.LIB	CPPRSO40.LIB (Single thread C object library) CPPRAO40.LIB (Single thread C++ object library)
Multi-thread object library	CPPWM35O.LIB	CPPRMO40.LIB (Multi-thread C object library) CPPRBO40.LIB (Multi-thread C++ object library)
Subsystem object library	CPPWN35O.LIB	CPPRNO40.LIB (C Subsystem object library) CPPREO40.LIB (C++ Subsystem object library)
Single thread complex number library	CPPWOX3.LIB	CPPRSY40.LIB (C++)
Multi-thread complex number library	CPPWOY3.LIB	CPPRMY40.LIB (C++)

C and C++ Compilers for OS/2, AIX and for Windows NT, Version 3.6

Single thread library	CPPRSI36.DLL	CPPRSI40.DLL (Single thread C library) CPPRAI40.DLL (Single thread C++ library)
Multi-thread library	CPPRMI36.DLL	CPPRMI40.DLL (Multi-thread C library) CPPRBI40.DLL (Multi-thread C++ library)

Subsystem library	CPPRNI36.DLL	CPPRNI40.DLL (C subsystem library) CPPREI40.DLL (C++ subsystem library)
Single thread import library	CPPRSI36.LIB	CPPRSI40.LIB (Single thread C import library) CPPRAI40.LIB (Single thread C++ import library)
Multi-thread import library	CPPRMI36.LIB	CPPRMI40.LIB (Multi-thread C import library) CPPRBI40.LIB (Multi-thread C++ import library)
Subsystem import library	CPPRNI36.LIB	CPPRNI40.LIB (C Subsystem import library) CPPREI40.LIB (C++ Subsystem import library)
Single thread static library	CPPRSS36.LIB	CPPRSS40.LIB (Single thread C static library) CPPRAS40.LIB (Single thread C++ static library)
Multi-thread static library	CPPRMS36.LIB	CPPRMS40.LIB (Multi-thread C static library) CPPRBS40.LIB (Multi-thread C++ static library)
Subsystem static library	CPPRNS36.LIB	CPPRNS40.LIB (C Subsystem static library) CPPRES40.LIB (C++ Subsystem static library)
Single thread object library	CPPRSO36.LIB	CPPRSO40.LIB (Single thread C object library) CPPRAO40.LIB (Single thread C++ object library)
Multi-thread object library	CPPRMO36.LIB	CPPRMO40.LIB (Multi-thread C object library) CPPRBO40.LIB (Multi-thread C++ object library)

Subsystem object library	CPPRNO36.LIB	CPPRNO40.LIB (C Subsystem object library) CPPREO40.LIB (C++ Subsystem object library)
Single thread complex number library	CPPRSY36.LIB	CPPRSY40.LIB (C++)
Multi-thread complex number library	CPPRMY36.LIB	CPPRMY40.LIB (C++)

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.

EXCNV (Windows)

The EXCNV tool migrates Version 3.5-level resource source compiler files to the VisualAge C++ Version 4.0-level resource source compiler files.

In Version 3.5, the MENUEX statements used the same short form of the type and state flags that the MENU statement used.

To be in sync with the MENUEX specifications as defined in the Microsoft SDK, the type and state flags have changed for the MENUEX statements. The EXCNV tool adds an MFT_ tag to the type flags:

Version 3.5 MENUEX Type Flag	Version 4.0 MENUEX Type Flag
STRING	MFT_STRING
BITMAP	MFT_BITMAP
MENUBARBREAK	MFT_MENUBARBREAK
MENUBREAK	MFT_MENUBREAK
OWNERDRAW	MFT_OWNERDRAW
RADIOCHECK	MFT_RADIOCHECK
RIGHTORDER	MFT_RIGHTORDER
RIGHT JUSTIFY	MFT_RIGHT JUSTIFY
SEPARATOR	MFT_SEPARATOR

and adds an MFS_ tag to the state flags:

Version 3.5 MENUEX State Flag	Version 4.0 MENUEX State Flag
GRAYED	MFS_GRAYED
DISABLED	MFS_DISABLED
CHECKED	MFS_CHECKED
HILITE	MFS_HILITE
ENABLED	MFS_ENABLED
UNCHECKED	MFS_UNCHECKED
UNHILITED	MFS_UNHILITED
DEFAULT	MFS_DEFAULT

EXCNV also changes MENUITEM SEPARATOR in a MENUEX statement to MENUITEM -1,'',MFT_SEPARATOR.

It also adds WINDOWS.H, if the header file is not already present.



Printed in the USA