VisualAge® C++ Professional for AIX®

Migration Guide

Version 5.0

Note!

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

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ 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.

Contents

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

About This Book. ix

Chapter 1. Prepare to Migrate to Version 5.0 1
Choosing the Right Compiler 2
  Features at a glance 2
  Recommendations based on code you have to maintain 3

Chapter 2. Migrate to Incremental Compilation 5
Create a Configuration File for Migration. 6
  A. Creating an Empty Configuration File with the IDE and SmartGuides 6
  B. Copying an Existing Configuration File from the Samples Provided with VisualAge C++ . . . 7
Preparation for Grouping Source Files for Migration 8
Create Source Group Directives for Migration . . . 9
Convert Compile and Link Options to Configuration Options. 10
Add Source Files to the Project for Migration . . . 11
  Adding Source Files Using the IDE 11
  Adding a Source File by Editing the Configuration File 12
  Example: Configuration File with Source File Added 12
  Build Errors You May Encounter After Adding Source Files 12
Promote Included System and User Header Files . . 13
Add Another Target to your Project for Migration 14
Migrate Other Commands 15
  Using the run Directive to Call Other Tools. . . 15
  Syntax of the run Directive 15
  Example: run Directive 16
  Notes on the Example 16
Arrange Options in Groups 17

Chapter 3. Migrate from Version 4.0 to Version 5.0 19
Combine Multiple Codestores in One Project . . . 19
Divide a Single-Codestore Project into Multiple Codestores 20

Chapter 4. Migrate to Version 5.0 Batch Compilation 21

Chapter 5. Troubleshooting References 23
Common Errors when Migrating 23
Errors Due to Changes in the C++ Language . . . 23
  bool, true, and false Keywords 23
  Changes to Digraphs in the C++ Language. . . 24
Errors Due to Changes in Compiler Behavior . . . 24
  Access-checking errors. 24
  typedefs 24
  Overloading ambiguities 25
  Syntax errors with new 26
Coding Adjustments for Orderless Programming . . 26
  Overload Resolution is Handled Differently . . 26
  Resolution to an Unexpected Declaration . . . 26
  Solution 27
  Unexpected Results With Macro Source Files . . 27
Common Template Problems 28
  Changes in Name Resolution 28
  Changes to friend Declarations 28
  Changes to the friend Declarator 29
Common Errors due to Duplicate Symbols 30
  Solution 31
Adjust for Using the IBM Open Class Library . . . 31

Chapter 6. Special Considerations for Migrating to Incremental Compilation . 33
Application Requires Multiple Targets 33
Application Uses Meta Source Files 34
Application Uses Meta Header Files 35
  Example: Meta Header Files. 36
Application Needs to Maintain Compatibility with a Batch Compiler 37
Application Uses Both C and C++ Source 38

Chapter 7. Migration Concepts 41
Promoting Source Files to the Configuration File . . 41
  Macro Source Files 42
  Using Macros Which Expand to Different Values in Different Source Files 42
Meta Source Files 43
Considerations When Using C++ I/O 43
  Example 44
Migration of Resource Files 44
  Resource Conversion Utilities for Cross-Platform Development. 44

Chapter 8. Other Migration References 47
Options for Compatibility with Previous Versions of VisualAge C++ 47
Compile Options from Earlier Versions of VisualAge C++ and Equivalent Configuration Options (AIX) . 53
Link Options from Earlier Versions of VisualAge C++ (AIX) 66
New Standard C++ Library Header Files 67
  The Language Support Library 67
  The Diagnostics Library 67
  The General Utilities Library 68
  The Standard String Templates 68

Localization Classes and Templates 68
The Containers, Iterators and Algorithms Libraries (the Standard Template Library) . . . 68
The Standard Numerics Library 69
The Standard Input/Output Library 69
Use the Standard Iostreams Library and the Compatibility Iostreams Library 71
Use the Standard Iostreams Library with the IBM Open Class Library 71
Makefile Rules Mapped to Configuration File Directives 72

Contacting IBM 73

Notices

Programming Interface Information

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

General-use programming interface 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
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architechture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

Chapter 1. Prepare to Migrate to Version 5.0

If you have not installed VisualAge C++, Version 5.0 yet, and you are not sure whether you should use the batch compiler or make the change to an incremental environment, please read "Choosing the Right Compiler" on page 2.

Migrating From a Batch Compiler Such as C Set ++® to Incremental Compilation

Lots of things have changed since the last release of C Set ++ and VisualAge C++ Version 3.5:

The Technology
: VisualAge features an incremental compiler. The implications for your code and your productivity are impressive, but if you are moving from a batch environment, you will need to spend some time to adapt your applications before you can take advantage of the changes. For example, makefiles cannot be processed by the incremental compiler. Following the migration process will reduce the amount of time and memory required to do each build, as well as the time you spend rebuilding when you make changes to source files.

The C++ Language
: VisualAge C++, Version 5.0 implements the 1998 approved C++ language standard. To help you make the transition, we have provided a number of compatibility options, but you still need to be aware of the many differences.

The VisualAge Development Environment
: The interface was redesigned in Version 4.0 with a host of helpful features. If you have not worked with VisualAge C++, Version 4.0, take time to familiarize yourself with the new Integrated Development Environment before you begin the process of migrating your applications. Several tutorials are included with the online help and in the *Getting Started* book.

You do not have to migrate fully in order to begin using VisualAge C++. You can migrate an application in stages, and take advantage of Version 5.0 features before your application is completely migrated. This approach makes sense for large applications.

In the initial phases of moving your applications to the new build paradigm, you may not see immediate improvement in build times, especially with newly created configuration files. Once you have invested the time to create an efficient build setup, however, you will find build performance increased, and the configuration file much easier to read than a traditional makefile.

The information in this document will help you make the transition and learn the new paradigm quickly.

Migrating From Version 4.0 to Version 5.0

While such features as incremental compilation and the Integrated Development Environment are familiar to users of Version 4.0, there are some changes you should be aware of. For example, VisualAge C++ now supports multiple codestores in a single project. We recommend you try the tutorials to learn about the new shape of projects. For a list of new features and capabilities, read What's New in the online help or in the *Getting Started* book.

Migrating to the New Batch Compiler

The new batch compiler included in VisualAge C++, Version 5.0 allows you to continue using makefiles. You may have to make some adjustments for changes in the C++ language, however. See "Chapter 4. Migrate to Version 5.0 Batch Compilation" on page 21.

Choosing the Right Compiler

VisualAge C++ Version 5.0 features both a fully incremental compiler and a new batch compiler. The Integrated Development Environment (IDE) operates with the incremental compiler. The batch compiler is run from the command line. Both compilers support the latest ANSI/ISO C++ language standard, and the latest version (Version 5) of the IBM® Open Class™ library.

Here are some considerations to help you choose the right setup for your needs.

Features at a glance

Features	**Benefits**	**Incremental Compiler**	**Batch Compiler**
Compilation	Supports the latest language standard	Uses configuration files	Uses makefiles
Ease of migration for makefile-based projects	Allow existing projects to take advantage of new standards and compiler technologies	Requires creation of configuration files	Little or no work required
IDE with real-time updates	Assists in rapid application development; integrated editing, browsing and debugging	Yes	No
Incremental compilation	Speeds compilation, especially with larger projects	Yes	No
Visual Builder	Speeds development with visual application assembly from predefined parts and automatically generated code.	Yes	Yes
	Compiles Visual Builder generated code	Yes	Yes

Data Access Builder	Speeds development with visual mapping from relational database tables into reusable parts, ready for use with the Visual Builder	Yes	Yes
	Compiles Data Access Builder generated code	Yes	Yes
Integrated debugger	Permits debugging without leaving the IDE	Yes	Yes[1]
Stand-alone debugger	Allows you to debug programs running a remote machine as well as any programs compiled with traditional debug information	Yes[2]	Yes
Libraries	ANSI/ISO Standard C++ library, including the Standard Template Library	Yes	Yes
IBM Open Class Library, V5.0	Compatible with Standard Template Library	Yes	Yes
Performance Analysis	Examine how your program uses system resources	Yes	Yes

[1]The IDE debugger allows you to debug programs that have been compiled with traditional debug information (TDI).

[2]Stand-alone debugging requires that you enable the export debugging information feature in your configuration file.

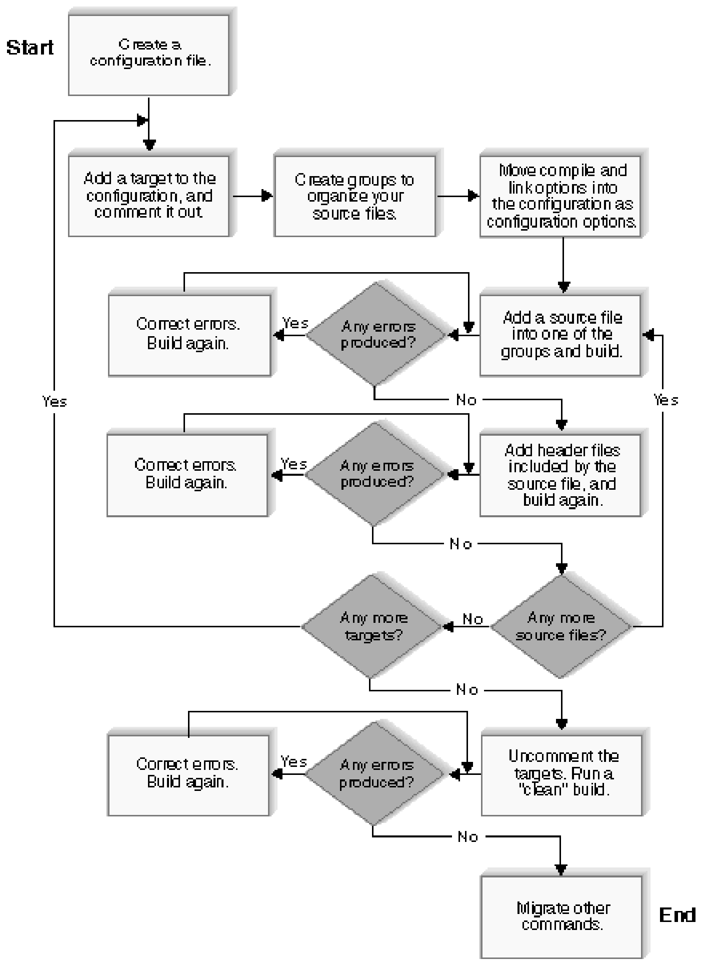
Recommendations based on code you have to maintain

If you are...	**IBM recommends:**
writing new code	incremental compiler
maintaining projects developed with VisualAge C++ Version 4.0	incremental compiler
maintaining existing code from a batch environment	batch compiler
porting existing code from another IBM platform	incremental compiler
porting existing code from a non-IBM platform	batch compiler
developing applications for deployment on multiple flavors of UNIX®	batch compiler

developing applications using C OpenMP support, SMP explicit directives or automatic parallelization	batch compiler

Chapter 2. Migrate to Incremental Compilation

For most projects, the task of migration follows this general pattern:


Start
Create a configuration file.
Add a target to the configuration, and comment it out.
Create groups to organize your source files.
Move compile and link options into the configuration as configuration options.
Add a source file into one of the groups and build.
Any errors produced?
Yes
Correct errors. Build again.
No
Add header files included by the source file, and build again.
Any errors produced?
Yes
Correct errors. Build again.
No
Any more source files?
Yes
No
Any more targets?
Yes
No
Uncomment the targets. Run a "clean" build.
Any errors produced?
Yes
Correct errors. Build again.
No
Migrate other commands.
End

Each of the following steps is explained in greater detail in this book:

1. "Create a Configuration File for Migration" on page 6, with one target, using the IDE's Project SmartGuide, or by copying a configuration file from one of the samples.
2. Ensure the target directive in your configuration file is disabled or commented, if you did not use the SmartGuide to do this automatically, so that the link phase is not invoked.
3. "Preparation for Grouping Source Files for Migration" on page 8 for your source files. In a later step, you will add source files to the configuration in these groups.
4. Using your current makefile and compiler, run a make and save the log to a text file. Open the file in the IDE. Locate the compile and link options, and "Convert Compile and Link Options to Configuration Options" on page 10.
5. Add a source file to the configuration. Choose one that includes the majority of your header files, if possible.

6. Build the project and correct any errors.
7. Promote included system and user header files and rebuild the project.
8. Add the next source file, and repeat steps 5, 6, and 7 until all of the source files for this target have been added. Once you have added several source files, you can make steps 5, 6, and 7 faster by adding a number of source files, rather than a single file, with each repetition of the steps.
9. Remove the comments from the target, and perform a *clean* build by selecting **Clean** from the **Project Workbook** menu, or by deleting the codestore (.ics file) and rebuilding. You may encounter build failures. You may need to add template source files or missing libraries, or address problems with virtual function tables not being defined. Do this before going to the next step.
10. For any pre-build and post-build targets your makefile called for, add the necessary run before or run after directives to invoke make. See "Migrate Other Commands" on page 15 for more details.
11. If your application requires more than one target, repeat steps 1 through 8 for each target. You can add several targets to a single configuration, or create a separate configuration for each target. The choices are described in "Application Requires Multiple Targets" on page 33.
12. When you have created a working configuration for each of your targets, you can combine all of the configuration files into one project.

Before you begin migrating, please read the Special Considerations to learn if there are any changes you should make to the process outlined here.

While migrating your project you may find that making significant changes to a configuration file results in a slow rebuild. You may achieve better performance by performing a clean build after you make extensive changes.

RELATED CONCEPTS

How Configuration Files Are Processed
"Migration of Resource Files" on page 44
"Promoting Source Files to the Configuration File" on page 41
Sources

RELATED REFERENCES

"Chapter 6. Special Considerations for Migrating to Incremental Compilation" on page 33

Create a Configuration File for Migration

Creating a basic, skeletal configuration file is the first step in migrating your application to the VisualAge C++ incremental compiler.

There are two ways to do this:

A. Creating an Empty Configuration File with the IDE and SmartGuides

1. Create a new configuration using the IDE's Configuration SmartGuide (select**Create Configuration** from the **Project Workbook** menu).
2. On the Configuration page of the SmartGuide (the title bar reads `Create a configuration`), give your configuration a name and choose a directory where the configuration file (.icc file) will be created.

3. On the same page, check the box labelled **Disable created target and source directives**.
4. Click **Next**.
5. On the Target Type page, select the type of target your configuration will build (an executable, shared library or static library), and click the **Add Target** button.
6. Click **Next** on the first page of the Target SmartGuide.
7. On the Target Name page, specify a name for your executable, library or shared library, and select a program type from the list in the lower half of the page.
8. Click **OK**, rather than **Next** (if you click **Next** accidentally, simply click **Back** on the next page).
9. Click **Finish**.
10. When you are prompted to open the project in the IDE, click **Yes**.
11. When you are asked if you wish to begin a build, click **No**.

Once you have created a configuration file, you can begin creating groups in the configuration. Later, when you add source files, they will be placed in the groups you create. Proceed to “Preparation for Grouping Source Files for Migration” on page 8.

B. Copying an Existing Configuration File from the Samples Provided with VisualAge C++

1. Browse the Samples section of the online help. Choose a sample project that resembles your application in function. There are also samples to illustrate changes to the language standard, using database access, and using the Standard Template Libraries.
2. Locate the sample you want in the `idesamp` or `samples` directory, copy the .icc file to your working directory, and rename it.
3. Start the IDE and open this project by selecting **Open an Existing Project** and clicking the **Browse** button to locate the renamed .icc file.
4. When you are prompted to begin a build, click **No**.
5. Switch to a Source view in your Configuration section.
6. Remove the file name listed in the target directive and replace it with the name of your target (.exe, .so or .a file).
7. Comment out the target directive, using C++-style double slashes (//), so that the link phase is not invoked (otherwise there may be many unresolved reference errors after each build). Ensure the open brace (’{’) following the target directive is *not* commented out (i.e., it should be on a separate line).
8. Delete any source files listed for the sample project.
9. Save and rename the updated .icc file by selecting **Save As** from the Source menu.

Once you have created a configuration file, you can begin creating groups in the configuration. Later, when you add source files, they will be placed in the groups you create. Proceed to “Preparation for Grouping Source Files for Migration” on page 8.

RELATED CONCEPTS

Configuration Files
Incremental C++ Builds
Macros in C++ Source Files

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

Configuration File Syntax

Preparation for Grouping Source Files for Migration

If you have just finished using the Configuration SmartGuide to create a configuration, you will have a configuration file that resembles the following:

```
disable target type(exe) "test2.exe"
{
  disable source "afile.cpp"
  disable source "main.cpp"
}
```

Now that you have created a basic configuration file for your first target, you will create groups in order to organize the source files used as input. At this stage, you have not yet added any source files. There are several reasons for creating groups for your source files at this stage, rather than adding all the source files in one group:

- When you add source files later, you will be able to add them in a methodical fashion.
- You will be able to easily maintain the configuration if you need to make changes after migration.
- When you have to apply options later, it will be easier to apply the options to a group of source files than to list the files inside each option directive.

You will create empty groups like the following, where applicable to your application:

- A group called PRIMARY_SOURCES, which will list your own .cpp, .cxx or .C files.
- A group called MACRO_SOURCES, which will list any of your own .cpp, .cxx or .C files which contain macros that need to be visible outside the file. Macro sources can be your own files, or system include files. If you wish to keep the different types of files separate, you could create two groups, MACRO_SOURCES and SYSTEM_INCLUDES, instead of one.
- If necessary, a group to collect any source files that do not fall into the previous three categories, for example, PRIMARY_SPECIAL_GROUP.

These group names are given as examples. You can use any descriptive name that helps you to organize your source files.

When you are ready to start creating groups for your sources, proceed to "Create Source Group Directives for Migration" on page 9.

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

Create Source Group Directives for Migration

You can add a group directive by using objects in the IDE or by editing the configuration file.

Creating Source Group Directives in the IDE

1. Switch to the **Configuration** section of the IDE and select the **Source and Groups** page.
2. Select the **Source and Groups** view on this page (the upper left pane). Right-click on the background (white space) of the view.
3. Select **Add Directive -> Group** from the pop-up menu.
4. In the **Group** field of the dialog box, type the name you want to give the group (PRIMARY_SOURCES, MACRO_SOURCES, or SYSTEM_INCLUDES. These example names are explained in the previous step, "Preparation for Grouping Source Files for Migration" on page 8).
5. Click **Apply**. The new group directive is added to the configuration file. The views in the Configuration section are refreshed to show the new directive. Currently the directive is empty. It contains no files (source specifiers). Views in other sections will not be updated until you build the project.
6. To make sure the group is considered a source, you must add the new group directive into a source directive.

Creating Source Group Directives in the Configuration File

1. Switch to the **Configuration** section of the IDE and select a **Source** view.
2. Move the insertion point to the location in the file where you want to insert a group. A group should be defined before it is called by your target and source directives.
3. Type `group` *`groupname`*`=` '' (empty string enclosed in single quotes), to create an empty group. Replace *`groupname`* with the names of each group you want to create, for example, PRIMARY_SOURCES, MACRO_SOURCES, or SYSTEM_INCLUDES (These example names are explained in the previous step, "Preparation for Grouping Source Files for Migration" on page 8).
4. Make sure that each group name you create also appears later in a source directive, for example:
   `source` *`type(cpp) PRIMARY_SOURCES`*
   If the group name does not appear in a source directive, the files in the group will not be considered part of the build.

When you have created the groups for your source files, your configuration file should resemble the following example :

```
group SYSTEM_INCLUDES = ''
group MACRO_SOURCES = ''
group PRIMARY_SOURCES = ''
disable target targetname
{
 option macros(global)
 {
  source type(cpp) SYSTEM_INCLUDES, MACRO_SOURCES
 }
 source type(cpp) PRIMARY_SOURCES
}
```

The next step in the migration process is to "Convert Compile and Link Options to Configuration Options" on page 10.

RELATED CONCEPTS

Configuration Files

RELATED TASKS

"Preparation for Grouping Source Files for Migration" on page 8
"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

group Configuration File Directive
macros global Miscellaneous Build Option (C++)

Convert Compile and Link Options to Configuration Options

Follow these steps in order to identify the options that were set for your application in your makefile:

1. Run a make and generate a make log file, using your current batch compiler.
2. Save the log as a text file.
3. Open the log file as a section in the IDE. To do this, locate the file name in the **Host** section of the Workbook, right-click over it, and select **View as -> Section** from the pop-up menu.
4. For now, disregard any options and commands that are not related to compilation (for example, options and commands for tools or preprocessors). You will need to refer to these commands later. Now you need to convert only the remaining compile and link options to configuration options.
5. In the remaining options, check to see if any sets of options are applied only to specific targets, or only to specific groups of source files. If there are sets of options like this, you may want to create option groups later, in order to make it easier to apply multiple options, and to maintain the setup after migration. Details are provided in the Related Tasks below.

Once you have identified which compile and link options applied to your project from your make log, you can use the **Convert Batch Options** view in the IDE to determine their equivalent configuration options and insert them into your configuration file. Option tables are also provided in the online documentation and in this book.

To convert the options in the IDE

1. Select the **Configuration** section of the IDE.
2. In the Configuration section, select the **Options** page.
3. In the upper left-hand pane (**Source and Targets** view), highlight the source file (Ss), or target (T) to which you want to apply the compile and link options.
4. Change the view in the right-hand pane. In the **View Types** menu, select **Convert Batch Options**.
5. In the right-hand pane (now the **Convert Batch Options** view), click in the **Batch Options** entry field.
6. Type in the first compile or link option you identified in your make log, in the format it appears (for example, `-O`).
7. Click **Add**. If the option has a valid equivalent in VisualAge C++, Version 5, the equivalent configuration option is displayed under **Converted options**.

8. When you have finished converting all the options identified in your makefile log, click **Apply**. The options are applied to the object you selected in the **Source and Targets** view.

When you have converted all the compile and link options and applied them to the correct sources, groups, or targets, you can proceed to “Add Source Files to the Project for Migration”. For further optimization of your options, you may wish to create groups for the options in the same way you created groups of sources. See Arrange Options in Groups for more information on doing this.

RELATED TASKS

“Arrange Options in Groups” on page 17
“Chapter 2. Migrate to Incremental Compilation” on page 5
“Migrate Other Commands” on page 15
Setting Build Options

RELATED REFERENCES

AIX Compile Options from Earlier Versions of VisualAge C++

Add Source Files to the Project for Migration

Once you have created a configuration file and set up source groups, you can insert source files into your source groups.

Even if your application is very large, do not add a long list of source files immediately. Adding one source file that includes many header files early in the migration process will make the rest of the process much faster. In most applications, a few source files will include the majority of header files that are needed by the application. It is best to avoid adding multiple files at once until you have done this part of the process several times, without encountering new errors.

Start by adding one of your .cpp, .cxx, or .C files to the PRIMARY_SOURCES group you created in Step 3 of the migration process.

You can add source files using views in the IDE, or by editing the configuration file.

Adding Source Files Using the IDE

1. In the IDE, select the **Configuration** tab, and the **Sources and Groups** page button. This view allows you to add source specifiers to configuration source and group directives.
2. Select the PRIMARY_SOURCES source group (from the Source and Groups pane), which you created in “Create Source Group Directives for Migration” on page 9.
3. To add the source to your group directive, select **Browse** from the **Change Group Directives** pane and select a source file in the **Select Files Dialog**.
4. To confirm the selected file, click **OK**.
5. Click **Add**.
6. Click **Apply**.

7. Build the project by clicking the Build button or by pressing **Ctrl+Shift+B**.

If there are build errors, use the **Messages** page of the **Project** section in the IDE to locate and resolve them.

Adding a Source File by Editing the Configuration File

1. In the IDE, select the **Configuration** tab, and the **Source** page button.
2. Add a source file to the PRIMARY_SOURCES group directive, by typing the filename inside the empty string (’ ’)you inserted when you created the groups:

   ```
   group PRIMARY_SOURCES ='filename'
   ```

   where *filename* is the name of the source file. When you add more files to the same group, separate them with commas.
3. Build the project by clicking the Build button or by pressing **Ctrl+Shift+B**.

If there are build errors, use the **Messages** page of the**Project** section in the IDE to locate and resolve them.

Example: Configuration File with Source File Added

If you created the groups used as examples in the previous step, your configuration file would now resemble the following:

```
group SYSTEM_INCLUDES = 'firstfile.cpp'
group MACRO_SOURCES = ''
group PRIMARY_SOURCES = ''
disable target targetname
{
 option macros(global)
 {
  source type(cpp) SYSTEM_INCLUDES, MACRO_SOURCES
 }
 source type(cpp) PRIMARY_SOURCES
}
```

When you build this configuration file, firstfile.cpp is compiled, but as the target is currently disabled, it will not be linked.

Build Errors You May Encounter After Adding Source Files

Some errors may be due to changes in the C++ language, which were not enforced by older compilers. You may need to either modify your source to conform to the new standard, or use compatibility options to mimic old C++ language semantics. See the related references for compatibility options and information about common errors.

Other errors may occur because a header file which was previously promoted to be a PRIMARY_SOURCE file contains a macro which was not used by other source files, but is used in this newly added source file. You may need to convert the header file from a primary source to a macro source by adding the header file name to the MACRO_SOURCES group and removing it from the PRIMARY_SOURCES group. More information on macro and non-macro sources is provided in Migration Concepts.

Once you have completed a build with no errors, the next step is to promote the files that were included by this file to your configuration, rebuild, and correct any further errors, before you add any more source files. Proceed to Promote Included System and User Header Files.

RELATED CONCEPTS

Sources
"Promoting Source Files to the Configuration File" on page 41

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

Options for Compatibility with Previous Versions of VisualAge C++
"Common Errors when Migrating" on page 23

Promote Included System and User Header Files

Each time you add a source file to your PRIMARY_SOURCES group, the next step is to promote any files that were included by that file.

To locate the included files in the IDE and add them to the configuration file:

1. Go to the **Codestore** section of the workbook. Select the **Source Files** page, and the **Source Files** view on that page.
2. In the pane showing the Source Files view, pull down the **Filter** menu ( ) and select **Show All** from the list of filters.
3. The view will now display all the files used to build the codestore: those that are listed in the configuration file are displayed as , and those that are included are displayed as .
4. Highlight one or more system files identified by the glyph. As a general rule, it is better to avoid promoting system files from the /usr/include directory until after you have promoted other system files, such as those in /usr/vacpp/include.
5. Right-click on the selection, and select **Add Source to Configuration** from the pop-up menu.
6. Select **Add Source to Source Directive** in the dialog box.
7. In the **Add to Source Directive** dialog box, select one of the groups you created in an earlier stage of the Migration Process. For example, if you created only one group for macro sources, select it. Otherwise, select SYSTEM_INCLUDES.
8. Build the project, and correct any errors.
9. Return to the **Source Files** view. You should no longer see the system file you promoted displayed as an object.
10. If there are still included files to promote in this view, repeat step 4 to step 7. Add your application's included files to the PRIMARY_SOURCES group, system files to the SYSTEM_INCLUDES group, and so on.

11. Build again. You may encounter some build errors that are due to macros in your application include files. If this is the case, move the files containing macros to the MACRO_SOURCES group, instead of the PRIMARY_SOURCES group.
12. Return to the Source Files view again. This time, you should see no more application header files displayed as secondary sources. If there are any, add them to the PRIMARY_SOURCES or the MACRO_SOURCES group.

Once you have corrected any errors, you can return to Add Source Files to the Project for Migration, to add the next source file. If you have already added a number of source files, you can modify Step 5 of the migration process by adding a number of files in each iteration. The steps are described in "Chapter 2. Migrate to Incremental Compilation" on page 5.

When you have no more source files to add for this target, you have finished creating the configuration. You have several choices now:

- If you have not already done so, you can refine your configuration by creating groups for your options, as you did for your source files. This will not impact build times, but will make your configuration file easier to read. The steps are described in Arrange Options in Groups.
- If your project contains other targets, you can add the remaining targets. See Add Another Target to your Project for Migration for details.
- If you have finished creating configurations for all the targets in your project, remove the **disable** keyword (or comments, //) from the configuration file for each target, to enable the linking phase. Optionally, you can also create a project and add each configuration to the project as a subproject. This is described in Combine Multiple Codestores in One Project.
- Finally, when all of your targets have been migrated, you can migrate any parts of your former make process that were not related to compiling and linking (for example, code generation phases). See "Migrate Other Commands" on page 15 for information on doing this.

RELATED CONCEPTS

"Promoting Source Files to the Configuration File" on page 41

RELATED TASKS

"Add Source Files to the Project for Migration" on page 11

Add Another Target to your Project for Migration

If you have already gone through the migration process described in "Chapter 2. Migrate to Incremental Compilation" on page 5, you have created a configuration file and added a number of sources to it.

If you have more than one target in your project (for example, multiple executable files, or an executable file and several libraries), you do not have to repeat the first three steps of the migration process to add another target. You can add another target to the same configuration file, or create a separate configuration for each target. Considerations for either option are described in "Application Requires Multiple Targets" on page 33.

To create a new configuration for each target, simply repeat the migration process for each target.

To add a target to an existing configuration, follow the steps described in Add or Remove a Target.

When you have created a configuration for every target, you can remove the **disable** keyword (or comments, //) from the target directive in each configuration file, to enable the linking phase. Optionally, you can also create a project and add each configuration to the project as a subproject. This is described in "Combine Multiple Codestores in One Project" on page 19.

Finally, when all of your targets have been migrated, you can migrate any parts of your former make process that were not related to compiling and linking (for example, code generation phases). See "Migrate Other Commands" for information on doing this.

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

Migrate Other Commands

In general, when you migrate an application, the only portions of your makefile that will have an equivalent in a configuration file are the commands directly related to compilation. Tools and preprocessors or post-processors can still be called using make. The command to launch the make process can be inserted in your configuration file with run directives (`run before`, `run after` and `run cleanup`, described below).

If your makefile contains commands which call other tools, a roughly equivalent configuration file would follow this format:

- `run before` *`make setup`*
- target and source directives (i.e. equivalent of everything in the makefile that has to do with compilation)
- `run after` *`make post_compile_commands`*

Grouping common commands in the run before and run after directives will help minimize the dual maintenance of the makefile and configuration file.

Using the run Directive to Call Other Tools

Use the run directive to execute system commands at different times during a build. Typical uses of the run directive include:

- Invoking other compilers such as an assembler or previous version of VisualAge C++.
- Invoking external utilities such as gencat, perl, or cppfilt.
- Setting file permissions of generated files.
- Invoking VisualAge C++ recursively to perform preprocessing.
- Calling make to perform portions of a build that have not yet been converted to VisualAge C++.

Syntax of the run Directive

```
run [ before | after | cleanup ] [ sources (string_list) ] [ targets
(string_list) ] string_list
```

The parameters of the run directive are system commands that are run synchronously as child processes at one of three points in the build. `run before` and `run after` directives execute commands either before or after the compile and link phase of the build is performed. `run cleanup` executes the commands only when **Clean** is selected from the IDE **Project Workbook** menu. Run directives normally execute during every build, in the order in which they appear in the configuration file. The exception to this is when a list of file sources and targets are specified. In this case, the sources and targets determine the order in which the directives are executed so that dependencies between directives are satisfied. The timestamps of the sources and targets also determine whether or not the commands are run. The rules are as follows:

- If any sources are missing or have timestamps that are later than the previous build then the commands are run.
- If any targets are missing or have timestamps that are later than the previous build then the commands are run.
- If any sources have timestamps that are later than any targets then the commands are run.
- If no sources or targets are specified then the commands are run.

Example: run Directive

The following configuration file copies a text file from another directory and then uses it to generate a message catalog using the gencat utility:

```
//run directive #1:
run before
sources("/home/project/messages/catalog.txt")
targets("mycatalog.txt")
"cp /home/project/messages/catalog.txt /home/build/mycatalog.txt"

//run directive #2:
run before
sources("mycatalog.txt")
targets("mycatalog.cat")
"cd /home/build",
"-chmod -f +w mycatalog.cat",
"usr/bin/gencat mycatalog.cat mycatalog.txt",
"-chmod -f -w mycatalog.cat"
```

Notes on the Example

1. Run directive#1 will execute before run directive#2 because run directive#2 lists “mycatalog.txt” as one of its targets and run directive#1 lists “mycatalog.txt” as one of its sources. That is, the order of appearance alone does not determine the order in which the commands are executed.
2. The use of the “-” preceding the chmod commands indicates that a nonzero return code from the command should not matter. If you omit the “-” then the build will fail if the command returns a nonzero return code.

If you have completed the migration process described in “Chapter 2. Migrate to Incremental Compilation” on page 5, you may wish to further optimize your code for the new environment. See the Related References below for more details.

RELATED CONCEPTS

run Configuration File Directive

RELATED REFERENCES

"Coding Adjustments for Orderless Programming" on page 26
"Makefile Rules Mapped to Configuration File Directives" on page 72

Arrange Options in Groups

If you are applying the same set of options to more than one source file, source group, or target, you may find it easier to create groups of options than to repeatedly list the same options in your configuration file.

For example, if you have an ungrouped set of options that appear in several places in your configuration file, like this:

```
target "mytarget"
  {
  option lang(signedbitfields), lang(digraphs,no), alloc(debug)
    {
    source mysource1.C
    }
  }
```

you can create a group for the options, and use only the group name, instead of the list of options, in every part of the configuration where you need to repeat the same options.

Here is an example of defining an options group, using the options from the example above:

```
option common_options = lang(signedbitfields), lang(digraphs,no), alloc(debug)
```

Using the example of the configuration file above, the file would now resemble this:

```
target "mytarget"
{
 option common_options
  {
   source mysource1.C
  }
}
```

RELATED CONCEPTS

option Configuration File Directive

RELATED TASKS

Migrate to Incremental Compilation

Chapter 3. Migrate from Version 4.0 to Version 5.0

If you are already using VisualAge C++ Version 4.0 incremental compilation and have set up your application's configuration file or files, migrating to Version 5.0 is relatively simple. You have three possible approaches to choose from:

1. Keep the divisions in your application as they are currently. In this case, you do not necessarily need to create project files to organize your configuration files, but you may wish to do so in order to make it easier to define multiple build scenarios.
2. Create a new project, using a project file, in order to bring together several different codestores into one project. See "Combine Multiple Codestores in One Project" for details on doing this.
3. Create a new project, using a project file with several subprojects, in order to split up a large application that contains multiple targets currently in one configuration file. For information on doing this, read "Divide a Single-Codestore Project into Multiple Codestores" on page 20.

Some considerations when deciding whether to combine multiple codestores in one project are listed in "Application Requires Multiple Targets" on page 33.

RELATED CONCEPTS

"Application Requires Multiple Targets" on page 33

Combine Multiple Codestores in One Project

To create a VisualAge C++, Version 5.0 project that includes several projects created in Version 4.0, or several configurations created in the migration process, do the following:

1. Start the Project SmartGuide, either by selecting **Create a new project** from the Welcome dialog, or selecting **Create project** from the Project Workbook menu in the IDE.
2. On the first page of the SmartGuide, the Project page, enter a name and working directory for the project. The name you enter here will be used to name the project file (an .icp file) which will contain the definition of your project, in much the same way the configuration file contains a definition of your targets and sources.
3. On the same page, in the **Project setup** area, select the second radio button, **Create a new project using new or existing configuration files**. (If you have already advanced to the next page, click **Back** to return).
4. Click **Next**.
5. On the next page, Configurations, browse to find each configuration file (.icc file) that you want to bring into the new project. Highlight each file or several files, and click **Add**. If you also want to create a new configuration, click the Create configuration button. This will launch the Configuration SmartGuide.
6. When you have located or created all the configuration files you need, click the **Finish** button. An .icp file will be created, listing all the configurations you selected as individual subprojects, and containing the outline of a default build similar to a make all.

7. You will be prompted to open the project in the IDE. To see how the new project was set up, click **Yes** to open it.

If your Version 4.0 project made use of IBM Open Class, read "Adjust for Using the IBM Open Class Library" on page 31 for information on other changes you may have to make.

RELATED TASKS

"Adjust for Using the IBM Open Class Library" on page 31

Divide a Single-Codestore Project into Multiple Codestores

If your project was developed in VisualAge C++ Version 4.0 and contained multiple targets in a single configuration, you can divide this configuration into several subprojects in Version 5.0. Considerations for doing this are listed in "Application Requires Multiple Targets" on page 33.

If you choose to do this, the following steps can help maintain readability and reusability of your project setup as you create subprojects:

1. Create a configuration file (.icc file) that contains only the options you previously had listed in your Version 4.0 configuration.
2. In this new file, create a named options group to list the options.
3. In your original configuration file, remove the options, include the new .icc file, and name the option group contained in the new file in place of the options you removed.
4. Cut and paste each target out of the original file, into a new configuration file (use a separate file for each target). Make sure to include all the source directives for each target.
5. Cut and paste the include directive for the options file and the option group name into each new file you create.

Once you have created a separate configuration for each target, follow the steps in Combine Multiple Codestores in One Project to create a project that consists of a subproject for each configuration.

RELATED TASKS

"Chapter 3. Migrate from Version 4.0 to Version 5.0" on page 19

Chapter 4. Migrate to Version 5.0 Batch Compilation

You can use IBM VisualAge C++ in batch mode as a C compiler for files with a .c (small c) suffix, or as a C++ compiler for files with a .C (capital C), .cc, .cpp, or .cxx suffix. The compiler processes your text-based program source files to create an executable object module, in the same manner as most traditional batch compilers.

However, the Version 5.0 batch compiler supports the new C++ language standard (ISO/IEC 14882:1998). You may have to make some changes in your code to reflect this. See “Errors Due to Changes in the C++ Language” on page 23 for more information on source-related changes you may have to make.

RELATED CONCEPTS

VisualAge C++ Batch Compilers

RELATED REFERENCES

Common Errors When Migrating
Common Template Problems

AIX Equivalent Batch Compile and Incremental Build Options
List of Batch Compiler Options and Their Defaults
New Standard C++ Library Header Files

Chapter 5. Troubleshooting References

Common Errors when Migrating

You may encounter situations in which code that compiles without errors in versions of VisualAge C++ prior to Version 4.0 produces warnings or error messages in VisualAge C++ Version 4.0 and 5.0. This can be due to changes in the language, or due to differences in the compiler behavior.

If you are migrating from any version of VisualAge C++ prior to Version 4.0, you should be aware of potential syntax errors with new. Versions of VisualAge C++ prior to Version 4.0 treated the following two statements as semantically equivalent:

```
new (int *) [1];
new int* [1];
```

The first statement is syntactically incorrect even in older versions of the C++ standard. However, previous versions of VisualAge C++ accepted it. This inconsistency with the language standard was corrected in VisualAge C++ Version 4.0. The first statement will produce a compilation error in Version 4.0 and 5.0.

Examples of changes to the language that may affect your code are provided in "Errors Due to Changes in the C++ Language".

If you are migrating from a batch environment to the incremental compiler, some of the examples in "Errors Due to Changes in Compiler Behavior" on page 24 may be applicable to you.

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

"Coding Adjustments for Orderless Programming" on page 26
"Common Template Problems" on page 28
"Errors Due to Changes in Compiler Behavior" on page 24
Errors Due to Changes in the C++ Language.

Errors Due to Changes in the C++ Language

bool, true, and false Keywords

The C++ standard now defines as keywords the tokens bool, true, and false. In versions of VisualAge C++ prior to Version 4.0, these keywords were not defined. When you migrate programs that define these keywords, you will encounter compilation errors. You can either remove your definitions, or use the 'lang(nokeyword)' option for each of these keywords that you want to undefine for compatibility purposes. For example, with the incremental compiler, to disable all three keywords, add the following to your configuration:

```
option lang(nokeyword, "bool"),
       lang(nokeyword, "true"),
       lang(nokeyword, "false")
```

To make the same change with the batch compiler, use the -qnokeyword=true|false|bool option.

Changes to Digraphs in the C++ Language

The C++ standard now defines and, bitor, or, xor, compl, bitand, and_eq, or_eq, xor_eq, not and not_eq as alternate tokens for &&, |, ||, ˆ, ˜, &, &=, |=, ˆ=, ! and !=. If any of these alternate tokens are used as variable, function or type names then you can set lang(digraph, no), or -qnodigraph to suppress the parsing of these tokens as digraphs.

Note: the lang(nokeyword) option cannot be used to disable the digraph.

RELATED REFERENCES

"Common Errors when Migrating" on page 23
"Common Template Problems" on page 28

Errors Due to Changes in Compiler Behavior

The following are examples of code which compiles without errors in versions of VisualAge C++ prior to Version 4.0, but which will produce errors or warnings with the incremental compiler.

Access-checking errors

```
class A {
  class B {
        void f(A::B);
        // A::B is private and can not be accessed from B
        // void f(B); <-- this is the appropriate change which
        // works for both compilers.
  };
};
```

The following code would result in the error `1540-0413: "A::B" is already declared with a different access`:

```
class A {
 public:
  class B;
  const B& foo();
 private:
  class B {};
};
```

This can be solved by either moving the definition of class B to the public part of class A (before the declaration of `foo()`) or moving the declaration of the member function `foo` to the private of class A (after the class B definition)

typedefs

This code will generate error `1540-0193: A typedef name cannot be used in this context.` Do not use the typedef-name: instead, use the name of the class:

```
class A { };
typedef A B;
class C {
 friend class B; // Should be friend class A;
};
```

Overloading ambiguities

There are now floating point and long double overloads of the standard math functions. For example, the following, which would generate no errors in a previous version of VisualAge C++, will produce the error message `1540-0219: The call to "pow" has no best match` on Version 5.0:

```
#include <math.h>
int main()
{
  float a = 137;
  float b;
  b = pow(a, 2.0);  //The call to "pow" has no best match.
  return 0;
}
```

The solution is to cast the arguments to pow, or use the compatMath language option, which removes the float and long double overloads. In this example casting 2.0 to be of type float solves the problem:

```
b = pow(a, (float)2.0);
```

The following generates a number of errors:

```
1540-0216: An expression of type "B" cannot be converted to "A".
1540-0219: The call to "A::A" has no best match.
1540-1228: Argument number 1 is an lvalue of type "B".
1540-1202: No candidate is better than "A::A(const A&)".
1540-1231: The conversion from argument number 1 to "const A &" uses the
user-defined conversion "B::operator A() const" followed by an
lvalue-to-rvalue transformation.
1540-1202: No candidate is better than "A::A(const C &)".
1540-1231: The conversion from argument number 1 to "const C &" uses the
user-defined conversion "B::operator C() const".
```

```
e.C
--
struct C {};
struct A {
  A();
  A(const C &);
  A(const A &);
};
struct B {
  operator A() const { A a ; return a;};
  operator C() const { C c ; return c;};
};
void f(A x) {};
int main(){
  B b;
  f((A)b);

  // The call matches two constructors for A instead of calling "operator A()
  return 0;
}
```

Solutions include (depending on your access to classes A, B, and C):

- changing `f((A)b)` to the explicit call `f(b.operator A())`
- removing the constructor `A(const C &)`
- adding a constructor `A(B)`
- removing either `operator A()` or `operator C()`

Syntax errors with new

Versions of VisualAge C++ prior to Version 4.0 treated the following two statements as semantically equivalent:

```
new (int *) [1];
new int* [1];
```

The first statement is syntactically incorrect even in older versions of the C++ standard. However, previous versions of VisualAge C++ accepted it. This inconsistency with the language standard was corrected in VisualAge C++ Version 4.0. The first statement will produce a compilation error in Version 4.0 and 5.0.

RELATED REFERENCES

"Coding Adjustments for Orderless Programming"
"Common Errors when Migrating" on page 23
"Errors Due to Changes in Compiler Behavior" on page 24

Coding Adjustments for Orderless Programming

Orderless programming may introduce changes to the semantics of a program migrated into the VisualAge C++, Version 5.0 incremental compilation environment. You should be aware of the effects of orderless programming described below, and make the necessary adjustments to your code.

If you are concerned about maintaining compatibility with batch compilation environments, consider using the lang(orderednamelookup) configuration option. This option can force builds to use an ordered programming model so that potential errors relating to orderless programming are detected and diagnosed within VisualAge C++, Version 5.0. However, builds may be slower when you use this option.

Overload Resolution is Handled Differently

VisualAge C++, Version 5.0 resolves an overloaded function call to the best available match, based on all known declarations of the function. In a traditional compiler, only those declarations that have been encountered at the point where the call occurs are used to determine the best match. In the following example, traditional compilers would resolve the call of foo to foo(int), since foo(double) has not yet been seen. The VisualAge C++ incremental compiler, however, knows about both declarations of foo, and will resolve the call to foo(double).

```
void foo (int);
void bar() { foo(1.2); } // call to foo matches which overload?
void foo(double);
```

Resolution to an Unexpected Declaration

A variable of class or struct type whose name is the same as that of the class or struct may result in a different interpretation in VisualAge C++, Version 5.0. In the following example, `stat` is both a type of struct, and an instance variable of that struct. A traditional compiler treats the `int f(stat);` declaration as a declaration of a function `f` that takes an argument of type `stat`. In VisualAgeC ++ Version 5.0, the declaration is treated as that of an integer variable that is initialized by the int operator of struct stat:

```
struct stat {
    operator int();
    };
int f(stat); // function declaration or variable initializer?
struct stat stat;
```

Solution

```
struct stat {
    operator int();
    };
int f(struct stat);
 // the elaborated type (with the keyword struct) ensures that
 // the struct stat is introduced as a declaration and so it can
 // not be an expression.
struct stat stat;
```

Unexpected Results With Macro Source Files

Do not rely on the order of listing of macro source files in the configuration file when any of the files have include directives. If a macro source includes another macro source that is listed after it in the configuration file, or includes a non-macro source that is included in the configuration file, the included file will be preprocessed and compiled before any non-macro source files (in other words, it will be treated as part of the macro source that includes it). This may cause unexpected results if there are dependencies between the macro source files. For example, consider the following source files and macro source statement in a configuration file:

```
// a.hpp
#include "c.hpp"
//...

// b.hpp
#define DEFINED_IN_B

// c.hpp
#ifdef DEFINED_IN_B
//...
#endif

// Configuration file
//...
option macros(global) {
   "a.hpp",
   "b.hpp",
   "c.hpp"
}
//...
```

The macro source a.hpp includes c.hpp, however c.hpp contains a conditional preprocessor directive based on a macro defined in b.hpp. Because neither a.hpp nor c.hpp includes b.hpp, the macro DEFINED_IN_B will be treated as false in c.hpp, because c.hpp is preprocessed before b.hpp. If you add #include "b.hpp" to either a.hpp or c.hpp, the problem is solved.

The best way to prevent this from happening is to ensure that macro source files only define macros, and do not include any non-macro source files.A file can become a macro source even if you do not list it in a macros(global) option, if it is included by another file that is listed in a macros(global) option. The only way to force a macro to not be global is to make it a secondary source file (a file not listed in the configuration file), and to ensure that no macro source files include the file. This can be a difficult task, requiring many primary source files to be changed to secondary source files, and potentially resulting in slower rebuild times.

RELATED REFERENCES

"Common Errors when Migrating" on page 23

Common Template Problems

If your code makes use of templates, you will be affected by various changes to the way the C++ language handles templates.

Changes in Name Resolution

- A template definition must be parseable (previous compilers would skip the definition and only parse the template as it was instantiated). To control the way the compiler handles template definitions, use the lang(templatedefimpls) option. To make the compiler follow the behavior of previous versions, set this option to `dontparse`.
- A name not found by name lookup and not indicated to be a type by the `typename` keyword, is assumed to not name a type.
- Name lookup does not consider template-dependent base classes.
- The keyword `typename` must be used to mark a dependent name as a type. The following example illustrates this:

```
template <class T> struct A
{
typedef int X;
};
template  <class T> struct B:A <T>
{
T::Y b1; // error Y is not found
A <T>::X b2; // error X is not found
void foo(X); // error X is not found
};
```

The errors can be fixed by changing the definition of `B` to:

```
template <class T> struct B : A <T>
{
typename T::Y b1;
        // keyword "typename" tells parser Y is a type
typename A<T>::X b2;
        // keyword "typename" tells parser X is a type
void foo(typename A<T>::X);
        // keyword "typename" tells parser X is a type
};
```

Changes to friend Declarations

With VisualAge C++, Version 4.0 and 5.0, friend declarations in templates may not have the same meaning they did with previous versions. For example, the following code will generate a warning message:

```
struct A {} a;

template <class T> struct S;

template <class T> void f(T&, S<T>&) {}

template <class T> A& operator << (A&, S<T>&) { return a;}
```

```
    template <class T> struct S
{
friend void f (T&, S&); // no explicit arguments
friend A& operator << (A&, S&); // no explicit arguments
};
```

To migrate this code, the friend declarations should be changed to include explicit template arguments:

```
template <class T> struct S
{
friend void f<T> (T&, S&); // explicit argument T
friend A& operator << <T> (A&, S&); // explicit argument T
};
```

Without the explicit arguments, the friend declarations will introduce nontemplate functions `f(int&, S&)`' and '`operator <<(A&, S&)`' into global scope and these nontemplate functions (which have no corresponding definition) will be the friends of `S`.

With the template argument added explicitly, an instantiation of `S`, such as `S<int>`, will make the template instantiations `f<int>(int&, S<int>&)` and `operator << <int> (A&, S<int>&)` friends of `S`.

Previous versions of VisualAge C++ would not accept explicit template arguments on friend declarations. If you wish to maintain compatibility with previous versions of the compiler, the explicit template arguments should be added with the use of a macro.

Changes to the friend Declarator

`friend` no longer prototypes the function. It simply implies that a function of the specified signature is granted friend access:

```
g.C
--
class A {
friend int lib_func1(int); // This function is from a library.
};
int f(){
  return lib_func1(1);
}
```

The solution is to add the following line:

```
int lib_func1(int);
```

RELATED TASKS

"Chapter 4. Migrate to Version 5.0 Batch Compilation" on page 21
"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

"Common Errors when Migrating" on page 23
templateDefImpls Language Option

Common Errors due to Duplicate Symbols

The VisualAge C++, Version 5.0 incremental compiler enforces the *one-definition rule* across all the source files listed in a configuration file. Traditional compilers are not able to enforce this rule, so it will not be uncommon to encounter some error messages associated with duplicate symbols while migrating to the incremental compiler, even if you did not encounter such errors with any other or previous compilers.

You may encounter unexpected error messages if:

- Two different classes are defined with the same name at global scope in two different source files.
- A function is defined in a header file which is processed more than once.

Usually the messages associated with a duplicate-definition error will indicate where the original and conflicting definitions appear. Wherever possible, it is best to rename one of the declarations. When the error is the result of header files which are processed more than once, it is best to promote the header files to the configuration file.

In the case of a class with multiple definitions, if you cannot rename either of the definitions, you can use namespaces to overcome the problem, as follows:

1. If the source file defining the class appears in the configuration file, remove it.
2. For each source file you remove, add a new source file to the configuration. This new source file should define a unique namespace, and should include the original source file between the opening and closing braces of the namespace.
3. Each time you reference the class, qualify the class name with the enclosing namespace, or precede the reference with a `using namespace` statement.

Example: Multiple Definitions of a Class

```
// one.hpp
class TooMany {
   public:
      int x,y;
};
```

```
// two.hpp
class TooMany {
   public:
      int a,b;
};
```

```
// main.cpp
int main(void) {
   TooMany One;
   TooMany Two;
   One.x=3;
   Two.a=4;
}
```

```
// toomany.icc
   target "problem.exe"
   {
      source "one.hpp",
             "two.hpp",
             "main.cpp"
   }
```

A build of this project fails with two messages:

```
x is not a member of class "TooMany"
"TooMany" is already defined.
```

Solution

To solve the problem, make one.hpp and two.hpp into secondary sources, included by the new primary sources ns_one.hpp and ns_two.hpp. (Do this by changing the sources “one.hpp” and “two.hpp” in the configuration to “ns_one.hpp” and “ns_two.hpp”.) Each of these new files defines a namespace within which it includes its near-namesake. In `main`, explicitly reference the appropriate namespace. The solution is shown below (the contents of one.hpp and two.hpp do not change):

```
// ns_one.hpp
namespace ns_One {
   #include "one.hpp"
}
// ns_two.hpp
namespace ns_Two {
   #include "two.hpp"
}
// main.cpp
int main(void) {
   ns_One::TooMany One;
   using namespace ns_Two;
   TooMany Two;
   One.x=3;
   Two.a=4;
}
// toomany.icc
   target "problem.exe"
   {
      source "ns_one.hpp",
             "ns_two.hpp",
             "main.cpp"
   }
```

RELATED TASKS

“Promote Included System and User Header Files” on page 13

RELATED REFERENCES

The One-Definition Rule

Adjust for Using the IBM Open Class Library

If your project was built in VisualAge C++ Version 4.0, you needed to set several language options so that bool, false and true would not be treated as keywords.

A different version of the IBM Open Class is included in VisualAge C++, Version 5.0, and this version does not require those options to be set. To migrate your project to Version 5.0, you should remove the following options from your configuration files:

- lang(nokeyword, bool)
- lang(nokeyword, false)
- lang(nokeyword, true)

More detailed information on changes to the IBM Open Class is included in Changes in Version 5 of the IBM Open Class in the online help.

For examples of configuration files for applications using IBM Open Class, browse the `ioc/samples` subdirectory in the VisualAge installation directory.

RELATED CONCEPTS

Changes in Version 5 of the IBM Open Class
Changes in Version 4 of the IBM Open Class

RELATED TASKS

“Chapter 2. Migrate to Incremental Compilation” on page 5
“Chapter 4. Migrate to Version 5.0 Batch Compilation” on page 21

Chapter 6. Special Considerations for Migrating to Incremental Compilation

If VisualAge C++ incremental compilation is to become the only development environment for your application, the migration path will resemble closely the steps outlined in "Chapter 2. Migrate to Incremental Compilation" on page 5. The following describe special situations that may require you to change the steps described there.

Application Requires Multiple Targets
: The resulting project may have to consist of multiple smaller projects.

Application Uses Meta Source Files or Meta Header Files
: The meta source model reduces duplicate preprocessing and compilation in a traditional environment, and is easily translated into an efficient configuration for your VisualAge C++, Version 5.0 project.

Application Needs to Maintain Compatibility With a Batch Compiler
: You must avoid using some Version 5.0 features and features of the new C++ language standard.

Application Uses both C and C++ Source
: C and C++ can be combined in the incremental compilation environment with little effort. There are some debugging considerations, however.

When you have determined the considerations for your application, you are ready to begin the migration process.

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5Migrate a Simple Application to VisualAge C++, Version 5.0 Incremental Compilation

Application Requires Multiple Targets

In VisualAge C++ Version 4.0, the only way to debug across targets was to combine them in a single codestore. This is no longer necessary. With VisualAge C++, Version 5.0, you can create a project to produce multiple executables or shared libraries. Typically, a project like this will consist of a subproject for each executable or library. You can also combine several targets in a single subproject. Considerations for both approaches are outlined here.

Multiple Targets in a Single Codestore

Here are some reasons for choosing to produce all targets from a single configuration file with a single codestore:

- Build times for changes made to shared interfaces will be reduced. For example, if three libraries and two executables make use of the same class, changes to the class interface need only be rebuilt once. If the same libraries and executables were in separate codestores, changes to the class interface would need to be rebuilt five times.
- Less disk space will be required to save the codestore.

Within a single codestore, you cannot break the one-definition rule (defining the same function or type more than once). You can, however, define the function main() more than once. This support for multiple main functions allows you to develop and debug multiple executables using the same libraries in the same codestore.

A Separate Codestore for Each Target

There are several reasons to distribute your targets among several subprojects. If your application meets any of the following conditions, you should set up your project to consist of several subprojects, one for each target:

- Producing all targets from a single codestore may result in too large a codestore, or excessive build times or memory usage.
- You have multiple definitions of a name. However, if you can rename one of the conflicting definitions, or make use of namespaces to separate the conflicting definitions, and recode, you can produce all targets from a single codestore.
- You can confine code changes to implementations, not interfaces. If you are only making changes to the internal implementations of individual targets, having targets in different codestores may result in faster build times and less memory usage.
- If you expect to make frequent changes to a specific target, you will not have to rebuild any other targets when you make updates.

If you are ready to begin migrating your application to the incremental compilation environment, proceed to “Chapter 2. Migrate to Incremental Compilation” on page 5.

RELATED CONCEPTS

Configurations
The One-Definition Rule

RELATED TASKS

Build in a Team Environment
“Divide a Single-Codestore Project into Multiple Codestores” on page 20
Produce Multiple Targets from One Build

RELATED REFERENCES

“Chapter 6. Special Considerations for Migrating to Incremental Compilation” on page 33

Application Uses Meta Source Files

The meta source model eliminates much of the duplicate preprocessing and compilation that the migration process attempts to reduce. Therefore, if your application already makes use of meta source with a batch compiler, you may see only a small improvement when you initially migrate to the incremental compiler. However, with this model, every change requires the entire meta source file to be recompiled. With the incremental compiler, subsequent builds should be significantly faster.

To migrate an application that uses a meta source file, follow these steps in place of the steps outlined in Migrate to VisualAge C++, Version 5.0 Incremental Compilation:

1. “Create a Configuration File for Migration” on page 6, with one target, using the IDE’s Project SmartGuide, or by copying a configuration file from one of the samples directories.
2. Comment out the “target” directive in your configuration file, if you did not already do so when creating the configuration file, so that the link phase is not invoked.
3. Add each file listed in the meta-source file to your configuration, inside a macros(global) option. Use the same order in the configuration file as in the meta source file. Do not include any header files in the configuration unless they are *only* included by the meta source file itself. For example:
```
//target "prog.exe"
    {
        option macros(global)
        {
            source "src1.cpp"
            source "src2.cpp"
            ...
            source "srcN.cpp"
        }
    }
```
4. Build the application and fix any errors. Repeat this step until all errors are fixed.
5. If your application requires more than one target, “Add Another Target to your Project for Migration” on page 14 to your configuration file, and repeat Steps 1 through 4.
6. Remove the comments from the targets, and perform a *clean* build by selecting **Clean** from the **Project Workbook** menu, or by deleting the codestore (.ics file) and building.
7. Add the necessary run before or run after directives to invoke make for any pre-build and post-build targets your makefile called for. See “Migrate Other Commands” on page 15 for more details on doing this.

RELATED CONCEPTS

“Application Uses Meta Header Files”
“Meta Source Files” on page 43

RELATED REFERENCES

“Chapter 6. Special Considerations for Migrating to Incremental Compilation” on page 33

Application Uses Meta Header Files

It is not uncommon for an application to have one header file, for example “glob.h”, that includes all or most of the application’s header files. If your application fits this description, you should follow these steps to migrate the application:

1. “Create a Configuration File for Migration” on page 6, with one target, using the IDE’s Project SmartGuide, or by copying a configuration file from one of the samples directories.
2. Comment out the “target” directive in your configuration file, if you did not already do so when creating the configuration file, so that the link phase is not invoked.

3. Using your current, makefile-based setup, run a make and save the log to a text file. Open the file as a section in the IDE. Separate the options for tools other than the compiler, and “Convert Compile and Link Options to Configuration Options” on page 10.
4. Add the meta header file to the configuration, and apply the macros(global) option to it. For more information on adding source files, see the Related Task below.
5. Build the project and correct any errors.
6. Add other source files, several at a time, or all at once.
7. Build and correct any errors. Repeat this step until all errors have been fixed.
8. If your application requires more than one target, “Add Another Target to your Project for Migration” on page 14 to your configuration file, and repeat Steps 1 through 7.
9. Remove the comments from the targets, and perform a *clean* build by selecting **Clean** from the **Project Workbook** menu, or by deleting the codestore (.ics file) and building.
10. Add the necessary run before or run after directives to invoke make for any pre-build and post-build targets your original makefile called for. See “Migrate Other Commands” on page 15 for more details on doing this.

Example: Meta Header Files

```
target "test.exe"
    {
     option macros(global), define("MAIN", "")  //or define(MAIN, "1"),
//depending on how MAIN
//is defined
{
// macros(global) is used to make macros defined in glob.h
// visible in each "srcX.cpp" file.
// MAIN is an example of a macro that might be used to
// conditionally define variable and functions declared in
// the header file.
  source "glob.h"
}
  source "src1.cpp"
  source "src2.cpp"
  ...
  source "srcN.cpp"
    }
```

RELATED CONCEPTS

“Application Uses Meta Source Files” on page 34
“Meta Source Files” on page 43

RELATED TASKS

“Add Source Files to the Project for Migration” on page 11

RELATED REFERENCES

“Chapter 6. Special Considerations for Migrating to Incremental Compilation” on page 33

Application Needs to Maintain Compatibility with a Batch Compiler

The following migration considerations apply if the target platforms for your application do not all support a version of the VisualAge C++, Version 5.0 incremental compiler:

- You may have to maintain a list of your sources in both your makefiles and your configuration files. Other commands, however, can be kept in their current state in a makefile. For more information on which types of commands to migrate, read "Migrate Other Commands" on page 15.
- You will need to maintain #include directives and forward declarations. Otherwise you may encounter preprocessor errors, or syntactic or semantic errors, when compiling changed code in the old environment.

  The lang(orderednamelookup) configuration option can force builds to use an ordered programming model so that potential errors relating to orderless programming are detected and diagnosed within VisualAge C++, Version 5.0.
- If you are using a version of Visual Age C++ or C Set ++ prior to Version 4.0, or if your compiler does not support the new (1998) language standard, you should use appropriate compatibility options and minimize use of language features such as:
  - for-loop counter variables scoped to the for statement. Set the lang(ansiForStatementScopes) option to "no" if you need to build your code with a compiler that does not support this change to the language.
  - New keywords such as true, false, bool. You can use the lang(nokeyword,"*name*") option for each such keyword so that a compile error occurs when you try to use such a keyword and you have not defined it somewhere in your program.
  - Enhancements to templates in the new standard. Template partial specializations and explicit instantiations are new. Avoid partial specializations, and use #pragma define instead of explicit instantiations. You can use macros for handling typename and template name qualifiers and for the header of a template specialization.
  - Namespaces
  - The Standard C++ Library, as declared in the namespace std, and which includes header files listed in "New Standard C++ Library Header Files" on page 67.

  See the information in Options for Compatibility with Previous Versions of VisualAge C++ for options that can help you maintain compatibility with earlier releases.
- In the case where new constructs must be used (for example, you may need to conditionally use "typename" to allow your template code to compile in both environments), you can use a set of preprocessor directives to define macros such as the following, and make the source file containing this code a macro source file in your configuration file. The macro used in this example, __IBMCPP__, is a predefined preprocessor macro that can be used to manipulate source based on the level of IBM compiler that is being used. __IBMCPP__ is set to the value of 500 for IBM VisualAge Version 5.0 and later, and can be used as follows:

```
#if __IBMCPP__ >= 500
 #define TYPENAME typename
#else
  #define TYPENAME
#endif
```

This provides backwards compatibility while allowing you to use new keywords as keywords. Where possible, use compatibility options instead.

RELATED TASKS

"Chapter 2. Migrate to Incremental Compilation" on page 5

RELATED REFERENCES

nokeyword Language Option
Options for Compatibility with Previous Releases
orderednamelookup Language Option
"Chapter 6. Special Considerations for Migrating to Incremental Compilation" on page 33

Application Uses Both C and C++ Source

You can use the IDE to develop both C code and C++ code. However, there are some special considerations when migrating an application using both C and C++ source files.

Features Supporting C Development

- The incremental compiler does not generate object files unless you specifically request them. If you need to generate object files (.a or .obj files), you can use the file(genObject, *path*) option to generate C object files to a specified directory. This can be useful for developing projects that use multiple directories.
- In a configuration file, you can explicitly specify whether each source file contains C or C++ source code, or accept the default type set by the file extension. VisualAge C++ runs the C compiler for C source files (i.e., for files with the extension .c)
- If you build with mixed source in the IDE, messages generated from compilation of both types of source are displayed in the Messages view.

Limitations with C Development

- You will need to specify the link(debug) option for targets containing C and you will need to specify gen(debug) for C source files. If all of your source is C++, and you are using the IDE is for debugging, these options should not be used.
- The C compiler is not incremental. A VisualAge C++ build from C source files records the include hierarchy and information from the configuration file in the codestore. It stores no other information about your C source code. Each time you rebuild, every C source file is recompiled if it or a file that it includes has a later timestamp than on the previous build.
- You cannot use incremental compilation for C source files or use incremental linking on code generated from C source code. If you have both C and C++ source files in one build, the linker links the object files compiled from C source files with the code in the codestore generated from C++ source files.
- You must include header files in every C source file that requires them, and ensure that functions, variables, etc., are declared before they are used. You do not have to take these precautions with C++ source code.
- C symbols are not exported by the option link(exportAll). C symbols need to be explicitly exported using the link(export, *symbol*) option.

If you are ready to begin migrating your application to the incremental compilation environment, proceed to "Chapter 2. Migrate to Incremental Compilation" on page 5.

RELATED CONCEPTS

Sources

RELATED REFERENCES

debug Link Option
exportAll Link Option
export Link Option
Special Considerations for Migrating to Incremental Compilation

Chapter 7. Migration Concepts

Promoting Source Files to the Configuration File

When a secondary source file is added directly to the configuration file (i.e., made primary), it is *promoted.* Promoting a source file can significantly improve build performance, although not every source file can safely be promoted.

The compiler only compiles source files that are listed in the configuration file once; any #include preprocessor directives in source files that include the primary source file are ignored. You can safely promote a source file to the configuration file provided the promotion does not change how the promoted file or other source files are transformed during preprocessing. For example, a source file containing conditional preprocessor directives based on a macro definition should not be promoted, if the source file is included by several other source files that define the macro differently. Consider the following three files, one.cpp, two.cpp, and condsrc.hpp:

one.cpp

```
#define CONDITION1
#include "condsrc.hpp"
```

two.cpp

```
#define CONDITION2
#include "condsrc.hpp"
```

condsrc.hpp

```
#ifdef CONDITION1
// Source code...
#endif
#ifdef CONDITION2
// Different source code...
#endif
```

Given these three files, it would probably not be safe to promote condsrc.hpp to your configuration, because the files one.cpp and two.cpp include differently preprocessed versions of the file. Making condsrc.hpp a primary source would be an error because the preprocessed code seen by the compiler would be different than if condsrc.hpp were not promoted. However, if the two sections of code in condsrc.hpp are not mutually exclusive (that is, there is no harm if both sections are compiled for both .cpp files) the best approach would be to define both macros within the configuration file, and to promote condsrc.hpp to the configuration file as a macro source file. The configuration file would contain the following macro definitions:

```
option define("CONDITION1"),
       define("CONDITION2"),
       macros("global") {
   source "condsrc.hpp"
}
```

Another situation where promoting a source file may not be safe is one in which the source file defines a macro whose name is also used in a different way elsewhere in your program. For example, consider the following two files, three.cpp and three.hpp:

three.cpp

```
int A=4; // Line 1
int x=A; // Line 2
#include "three.hpp"
x++;
```

three.hpp

```
#define A 3
```

If **three.hpp** is made a macro source in the configuration (so that it is visible to all non-macro sources in the project), then **A** will be defined as the integer literal 3, resulting in two problems:

- A compilation error at line1 of **three.cpp** (because the statement `int A=4;` becomes `int 3=4;`)
- The literal value 3 assigned to **x** at line 2, instead of the variable A, whose value would have been 4.

This situation is usually easy to detect during migration, as compile errors often result. In this example the best solution would be to remove the ambiguity by renaming either the variable A or the macro A, and promoting **three.hpp** to the configuration file as a macro source file.

Macro Source Files

When you promote a source file to the configuration file, you should determine whether the source file should be a macro source file. Macro source files are treated differently from other primary source files in two ways:

- Macro source files are preprocessed before compilation begins, and any macros defined in them are visible to all primary source files
- Macro source files are guaranteed to be preprocessed in the order in which they appear in the configuration file, so macros defined by one configuration file are defined in all subsequently listed macro source files unless redefined or undefined later.

If a header file defines macros used by other sources, and is listed as a source in a configuration file, it should be enclosed in the macros(global) option.

Using Macros Which Expand to Different Values in Different Source Files

Here is an example,

```
--global.hpp--
....
extern int i
#ifdef MAIN
= 1
#endif
;
```

```
—-t.icc—
target "t.exe"
   {
       option define("MAIN", "") //or define(MAIN, "1"), depending
                                 //on how MAIN is defined
    {
         source "global.hpp", "other.hpp"
        }
source "t.cpp"
...
...
}
```

In this application, global.hpp is included by many .cpp files but the macro MAIN is only defined when compiling one of the .cpp files. global.hpp can be manually promoted by enclosing it with option define(MAIN, “”) { global.hpp }.

RELATED CONCEPTS

Sources

RELATED TASKS

“Promote Included System and User Header Files” on page 13

Meta Source Files

A meta source file is a file that includes other source files to improve the performance of full builds in a traditional compiler environment. For example, if you have ten source files that all make use of a group of common header files, you can reduce the time taken for a full build of your application by compiling a single meta file that includes these ten source files, instead of compiling each source file separately, as follows:

```
// meta.cpp
#include "src1.cpp"
#include "src2.cpp"
...
#include "srcN.cpp"
```

Headers used by more than one source file in the group are compiled only once when the meta source file is compiled (provided the header files use macro guards), instead of being compiled up to ten times. This build improvement in full builds is offset by slower builds when a single source file included by the meta source file is changed, because the meta source file itself is the compilation unit that must be recompiled.

When you migrate meta source files to configuration files for the VisualAge C++, Version 5.0 incremental compiler, you can gain some performance benefits (because, as with a traditional compiler, your included files will be compiled fewer times). However, the difference in build times may not be as significant as the difference achieved when migrating from individual source files to a configuration file. The meta source model has already reduced the build time.

RELATED CONCEPTS

“Application Uses Meta Source Files” on page 34
“Application Uses Meta Header Files” on page 35

Considerations When Using C++ I/O

Two implementations of the iostreams input/output library are included in VisualAge C++, Version 5.0:

- The first implementation conforms to the specification of the iostreams library as it appears in the ISO C++ standard. Types and functions associated with this implementation are declared in the standard header files <iosfwd>, <iostream>, <ios>, <streambuf>, <istream>, <fstream>, <ostream>, <iomanip>, <sstream>

and <strstream>. Use this implementation of iostreams for conformance to the C++ standard, or when using other components of the Standard C++ Library.

- The second implementation is provided for compatibility with versions of VisualAge C++ prior to Version 4.0. Types and functions associated with this implementation are declared in the non-standard header files <iostream.h>, <fstream.h>, <iomanip.h>, <strstream.h>, <stream.h> and <stdiostr.h>.

The standard iostreams library declares all names in the standard namespace std, while the compatibility iostreams library declares all names in the global namespace. In a project that uses both iostreams implementations, if the using-directive `using namespace std;` appears in any scope that encloses the use of a name which is declared by both implementations of iostreams, a compile-time error will result.

Example

```
#include <iostream>
#include <iostream.h>
using namespace std; // Line 1
int main(void) {
   cout << "I can hear music\n"; // Line 2 - which cout?
}
```

The header file <iostream> declares the static object std::cout and the header file <iostream.h> declares the static object ::cout. Because of the using-directive at line 1, both of these objects will be considered when name lookup attempts to resolve the identity of the cout named at line 2. The compiler will halt with a diagnostic message. If the using-declaration at line 1 is deleted, the source of ambiguity is removed and the example will compile successfully. The cout named at line 2 will then resolve to the compatibility iostream library's cout, which is declared in the global scope in the header file <iostream.h>.

It is possible to use the two implementations of iostreams together.

RELATED TASKS

"Use the Standard Iostreams Library and the Compatibility Iostreams Library" on page 71
"Use the Standard Iostreams Library with the IBM Open Class Library" on page 71

Migration of Resource Files

Resource Conversion Utilities for Cross-Platform Development

Because resource files are unique to the environment they were developed in, resources cannot simply be inherited by similar applications developed under different environments. To create the same application and use the same resources for a different platform, the format of the resource must be converted to suit the operating environment. The resource conversion utilities enable resources to be ported to other platforms without having to re-create the objects for similar applications.

AIX

Resource Conversion Utility

The Resource Conversion Utility converts Windows-specific resource files for use with AIX. VisualAge C++, Version 5.0 for AIX can accept Windows-format icons, bitmaps, and cursors without conversion.

OS/2

WIN Resource Conversion Utility

The Resource Conversion Utility converts OS/2-specific resources to Windows-compatible resources and vice versa. Resources compiled with an OS/2® resource compiler, will enable only OS/2 applications to access resources in the .RC files.

RELATED TASKS

"Chapter 4. Migrate to Version 5.0 Batch Compilation" on page 21
"Chapter 2. Migrate to Incremental Compilation" on page 5

Chapter 8. Other Migration References

Options for Compatibility with Previous Versions of VisualAge C++

Three predefined options groups are provided with VisualAge C++, Version 5.0 to allow for greater compatibility with IBM C and C++ Compilers, Version 3.6 and later, and with the C++ language standard.

The options groups can be set in a configuration file by including the option group name in an option directive. The available groups are lang_compat366, lang_strict98 or lang_ansi, and lang_extended. The options and settings included in each group are listed in the tables below.

To use these option groups with the batch compiler, use the -qlanglvl option, with the settings -qlanglvl=compat366, -qlanglvl=strict98 or -qlanglvl=ansi, and -qlanglvl=extended, respectively.

Note: the lang_strict98 and lang_ansi groups are identical.

Options		**Group names**		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
gen (rtti)	-qrtti	no	yes	no

Options		**Group names**		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
gen (compatNestedTemplateAlignmentRule)	-qlanglvl=oldtmplalign	yes	no	no

Options		**Group names**		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (allowTypedefAsClassName)	-qlanglvl=typedefclass	yes	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (anonymousStructs)	-qlanglvl=anonstruct	no	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (ansiForStatementScopes)	-qlanglvl=ansifor	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (compatFriendDeclarations)	-qlanglvl=oldfriend	yes	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (compatMath)	-qlanglvl=oldmath	yes	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (compatTempAccessChecking)	-qlanglvl=oldtempacc	yes	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (compatTemplateSpecialization)	-qlanglvl=oldtmplspec	yes	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (extendedAnonymousUnions)	-qlanglvl=anonunion	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (illformedPointerToMember)	-qlanglvl=illptom	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (implicitInt)	-qlanglvl=implicitint	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (longlong)	-qlonglong	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (offsetOfNonPODClasses))	-qlanglvl=offsetnonpod	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (oldDigraphs)	-qlanglvl=olddigraph	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (staticConstLinkage)	N/A	yes	yes	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (templateDefImpl, dontparse | parsewithwarnings)		dontparse	parsewithwarnings	parsewithwarnings
	-qtmplparse= no | parse | warn	no	warn	warn

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (trailingEnumCommas)	-qlanglvl=trailenum	yes	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (zeroExtentArrays)	-qlanglvl=zeroextarray	no	no	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
link (nameMangling)	-qnamemangling=compat|ansi	compat	ansi	ansi

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
macros (redefine)	-qlanglvl=redefine	no	no	no

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,bool)	-qkeyword=bool	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,explicit)	-qkeyword= explicit	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,export)	-qkeyword=export	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option>**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,false)	-qkeyword=false	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option>**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,mutable)	-qkeyword= mutable	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option>**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,namespace)	-qkeyword= namespace	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,true)	-qkeyword=true	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,typename)	-qkeyword= typename	no	yes	yes

Options		Group names		
Configuration Option	**Batch Option**	**compat366**	**strict98/ ansi**	**extended**
lang (keyword,using)	-qkeyword=using	no	yes	yes

RELATED REFERENCES

AIX Equivalent Batch Compile and Incremental Build Options (AIX)

Compile Options from Earlier Versions of VisualAge C++ and Equivalent Configuration Options (AIX)

With the VisualAge C++ incremental 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 of C Set ++, Version 3.1.4. 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. Whether the new option applies only to C or only to C++ code in this version of VisualAge C++ is indicated in the description column.

In the option syntax, the characters [and] enclose optional items, and the character | indicates alternatives.

Old Option	Description	Configuration File Option
-#	Display language commands but do not invoke them; output goes to stdout.	N/A
-+	Treat .c files as C++ source code.	N/A
C -a	Produce test coverage information; output goes to .d file.	N/A

Old Option	Description	Configuration File Option
-ae	Produce test coverage information for C++ exception handling; output goes to .d file.	N/A
-B	Construct alternate compiler/assembler/linkage editor program names.	N/A

C++ -bbigtoc
C++ -bnobigtoc
	Generates extra code when the size of the table of contents (TOC) grows to a size greater than 64KB.	link(bigTOC, yes)
		link(bigTOC, no)

-bD:*number* -bmaxdata:*number*	Set the size of the heap in bytes. The first number specifies the total virtual address space to reserve. The second number the amount of physical memory to commit initially.	link(heap, *number*)

-bdynamic
-bstatic
	Determines which types of library files are searched by linkage editor.	link(linkWithSharedLib [,yes])
		link(linkWithSharedLib,no)

C++ -bI:*filename*		supported as a source in the configuration: source type(imp) "*filename*" or source "*filename*.imp"
-blibpath:*string*	Override search path.	link(libPathOut, *string*)
-bnoentry	Specify that the address of the symbol is the start address of the executable module being built.	link(entry,no)

-bnso or -bnautoimp
-bso or -bautoimp
	Specifies if unstripped, shared objects are statically linked as ordinary object files.	link(static, yes)
		link(static, no)

C -brtl	Tells the linkage editor to accept both .so and .a library file types.	N/A
-c	Do not send object files to the linkage editor.	N/A

-C	Preserve comments in preprocessed output.	pp(preprocessOnly)
		pp(preserveComments)

C -D<*name*>[=<*def*>]	Define <name> as in #define directive. If <def> is not specified, 1 is assumed.	define(...)
C++ -D*name*[=[*value*]]	Define preprocessor macro.	define(*macro_name*[, *string*])
-Wl,-D*number*	Set the starting address for the data section of the output file to *number*.	link(dataStart, *number*)

C++ -E	Preprocess but do not compile. Output goes to stdout.	pp(preprocessOnly)
		pp(stdout)

-e*name*	Specify that the address of the symbol is the start address of the executable module being built.	link(entry, *string*)
C++ -F	Use alternate configuration file, etc.	supported as a vacbld command option

Old Option	**Description**	**Configuration File Option**
▶ C -F<*x*>[:<*stanza*>]	Use alternate configuration file, etc.	supported as a vacbld command option stanzas also supported by defaults(*stanza*)

-g	Include traditional debug option.	link(debug[, yes])
		link(debug, no)

-H*number*	Set the address alignment for all loadable segments so that the start of each segment is aligned at a multiple of the specified number of bytes.	link(alignAddr, *number*)
▶ C++ -I	Specify #include search paths.	incl(searchpath, *path*)
▶ C -I<*dir*>	Search in directory <dir> for include files that do not start with an absolut path.	include(searchPath,string)
▶ C -I<*key*>	Search the specified library file, where <key> selects the file lib<key>.a.	N/A
▶ C++ -K ▶ C++ -z		N/A
-l*name*		N/A
-L*string*	Specify search path for library files.	link(libSearchPath,*string*)
-ma	Generate inline calls to the “alloca” function as if “#pragma alloca” directives were in the source file.	N/A
-M	Generate information to be included in a “make” description file; output goes to .u file.	file(makeDep[,yes])
▶ C++ -o	Name generated executable or object file.	N/A

▶ C -o<*name*>	Name generated executable or object file.	file(genObject[,yes])
		file(genObject,*string*)
		file(genObject,no)

-O -O2 -O3	Optimize code.	opt(level, *number*)
▶ C -O4	Equivalent to -O3 -qipa with automatic detection of architecture and tuning option.	N/A
-p ▶ C++ -pq	Enable code for performance analysis.	gen(profile[, yes])

▶ C -pg	Generate profiling support code including BSD profiling support.	gen(profile, bsd)
		link(debug[,yes])

-P	Preprocess but do not compile. Output goes to .i file.	pp(preprocessOnly[,yes])

-Q	Inline specified user functions.	opt(inline[, yes])
		opt(inline, no)
		opt(autoInline[, yes])
		opt(autoInline, no)
		opt(autoInlineLimit)

Old Option	Description	Configuration File Option

-Q!
-qnoinline
	Do not inline any function.	opt(inline, no)
		opt(autoInline, no)

C -Q=<*lc*>
C -qinline=<*lc*>
	Inline if number of source statement in function is less than the number specified in <lc>.	opt(inline)
		opt(autoInline)
		opt(autoInlineLimit)

C -Q-<*nm*>
C -qinline-<*nm*>
	Not inline function listed by names in <nm>.	opt(inline[, yes])
		opt(autoInline)
		opt(noInlineFunc,name)

C -Q+<*nm*>
C -qinline+<*nm*>
	Attemp to inline funtion listed by names in <nm>.	opt(inline[, yes])
		opt(inlineFunc,name)

C -qalias=<*typ* | *allp* | *addr* | *ansi*>	Specifies the aliasing assertion to be applied to your compilation unit. The available options are: typ= Pointers to different types are never aliased. allp= Pointers are never aliased. addr= Variables are disjoint from pointers unless their address is taken. ansi= Pointers can only point to an object of the same type. Require -O option.	opt(alias,...)
-qalign=*option*	Specify alignment of data items.	gen(align, *option*)
-qansialias C++ -qnoansialias	Specify which aliasing rules can be used during optimization.	opt(alias, ...)

C++ -qarch=*name*
C -qarch=<*com* | *pwr* | *pwr2* | *pwrx* | *ppc* | *ppcgr*>
	Specifies the architecture on which the executable program will be run. The available options are:
	com= Produce an object that contains instructions that will run on all the POWER and PowerPC hardware platforms.
	pwr= Produce an object that contains instructions that will run on the POWER hardware platform.
	pwr2= Produce an object that contains instructions that will run on the POWER2 hardware platform.
	pwrx= Same as pwr2.
	ppc= Produce an object that contains instructions that will run on any of the 32-bit PowerPC hardware platforms.
	ppcgr= Produce an object that contains optional graphics instructions for PowerPC processors.
	The default is -qarch=com. If the -qarch option is specified without the -qtune=<option>, the compiler uses -qtune=pwr.
		gen(arch, *name*)
		gen(arch,...)

Old Option	**Description**	**Configuration File Option**

C++ -qassert=*option*
C -qassert=<*typ* | *allp* | *addr*>
	Specifies the aliasing assertion to be applied to your compilaton unit. The available options are:
	typ= Pointers to different types are never aliased.
	allp= Pointers are never aliased.
	addr= Variables are disjoint from pointers unless their address is taken.
		opt(alias, *option*)
		opt(aliasAssert,typ[,yes])
		opt(aliasAssert,typ,no)
		opt(aliasAssert,allp[,yes])
		opt(aliasAssert,allp,no)
		opt(aliasAssert,addr[,yes])
		opt(aliasAssert,addr,no)

-qattr	Produce an attribute listing; output goes to.lst file.	list(attr[,yes])
-qattr=full	Produce an attribute listing containing all names, whether referenced or not; output goes to .lst file.	list(attr, full)

-qbitfields=signed
-qbitfields=unsigned
	Specify whether bitfields will be signed or unsigned.	lang(signedBitfields[,yes])
		lang(singedBitfields,no)

-qbrowse	Produce information for the source code browser; output goes to .brs file.	N/A

-qchars=*signed*
-qchars=*unsigned*
	Treat plain *char* variables as signed or unsigned.	lang(signedChars[, yes])
		lang(signedChars, no)

-qcheck
-qnocheck
C -qcheck=suboptions
	The suboptions are:
	all= Switches on all of the following suboptions.
	nullptr= Performs run-time checking of addresses contained in pointer variables used to reference storage.
	bounds= Performs run-time checking of addresses when subscipting within an object of known size.
	divzero= Performs run-time checking of integer division.
		gen(check, zeroDivide[, yes])
		gen(check, zeroDivide, no)
		gen(check, nullPointer[,yes])
		gen(check, nullPointer,no)
		gen(check, bounds[,yes])
		gen(check, bounds,no)

-qcinc=<*prefix*>
C++ -qnocinc=*prefix*
	Includes files form/user/include by inserting
	extern "C" {
	before each <prefix> and inserting }
	after it.
		incl(externC, *prefix*)
		incl(noExternC, *prefix*)

-qcompact
-qnocompact
	Optimize code for size.	opt(size[, yes])
		opt(size, no)

-qcpluscmt C -qnocpluscmt	Permit "//" to introduce comment that lasts until the end of the current source line, as in C++.	lang(slashSlashComment [,yes]) lang(slashSlashComment,no)

C++ -qdataimported
-qdataimported=*string*
C -qdataimported=
<*name1*>:<*name2*>:...
	Specify which data items are imported. If no names are specified, all data items are assumed to be imported. This is the default.	link(dataImported, none)
		link(dataImported, all)
		link(dataImported, names)
		link(dataImportedNames, *string*)

-qdatalocal
C++ -qdatalocal=*string*
C -qdatalocal=
<*name1*>:<*name2*>:...
	Specify which data items are local. If no names are specified, all data items are assumed to be local.	link(dataLocal, none)
		link(dataLocal, all)
		link(dataLocal, names)
		link(dataLocalNames, *string*)

-qdbcs
C -qnodbcs
	Allow use of DBCS.	lang(dbcs[,yes])
		lang(dbcs,no)

Old Option	Description	Configuration File Option
-qdbxextra C -nodbxextra	Produce symbol table for unreferenced variables.	gen(debugunreferenced[, yes])

-qdigraph
C -qnodigraph
	Permit ANSI digraph and keyword operators.	lang(digraphs[, yes])
		lang(digraphs, no)

C -qdisjoint=<*disjointSpec*>	Tell the optimizer that certain aliases are impossible.	N/A

-qdollar
C -qnodollar
	Allow user to specify dollar sign ('$') in identifier names.	lang(allowDollarInNames[, yes])
		lang(allowDollarInNames, no)

-qenum=*enumopt* C -qenum=<*small* | *int* | *1* | *2* | *4*>	Specify whether minimum-sized enumerated types will be produced or not. <enumopt> can be either small, int, 1, 2 or 4. small denotes that either one, two or four bytes of storage will be allocated for enum variables based on the range of the enum constants. int is the default, and causes enum variables to be treated as though they were of type signed int for C or either signed or unsigned int for C++ depending on the range of the enum constants. 1 will cause enum variables to be packed into 1 byte. 2 will cause enum variables to be packed into 2 bytes. 4 will cause enum variables to be packed into 4 bytes.	gen(enumSize, ...)

-qextchk
C -qnoextchk
	Perform external name type-checking and function call checking.	link(typeCheck[, yes])
		link(typeCheck, no)

-qfdpr C -qnofdpr	Collect information about programs for use with the AIX fdpr (Feedback Directed Program Restructuring) performance-tuning utility.	N/A
C++ -qflag	Set minimum severity level.	report(level, ...)

C -qflag=<*sev*>	Specify severity level of diagnostics to be reported in source listing adn stderr; the severity level is one of <i,w,e,s,u>.	list(reportLevel, sev)
		report(level,sev)

C -qflag=<*sev1*>:<*sev2*>	Specify severity level of diagnostics to be reported in source listing (<sev1>), adn stderr (<sev2>); the severity levels are one of <i,w,e,s,u>.	list(reportLevel, sev1)
		report(level, sev2)

Old Option	Description	Configuration File Option

▶ C++ -qfloat
▶ C -qfloat=*<opt1>:<opt2>: ...:<optN>*
	The available options are:
	emulate= Function calls are emitted in place of PowerPC floating-point instructions.
	
	fltint= Do range checking of floating point to integer conversions.
	hsflt= Never round single-precision expressions, and do not perform range checking for floating-point to integer conversions.
	hssngl= Round single-precision expressions only when the results are stored into REAL*4 memory locations.
	nans= Detect conversion of single-precision NaNS to double precision call checking.
	nomaf= Suppress generation of multiply-add instructions.
	nofold= Suppress compile-time evaluation of constant floating-point expressions.
	rndsngl= Ensure strict adherence to IEEE standard. Results of operations on single-precision values remain in single precision.
	rrm= Specify run-time rounding mode.
	
	rsqrt= Specify whether a division by the result of a square root can be replaced with a multiple by the reciprocal of the square root.
	
	spnans= Generate extra instructions to detect signalling NaN of conversion from single precision to double precision. (Obsolete; use the equivalent -qfloat=nans option.)
		gen(float, ...)
		gen(float, *string*[,yes])
		gen(float, *string*,no)

-qflttrap
-qflttrap=*option*
▶ C -qflttrap= *<opt1>:<opt2>:...:<optN>*
▶ C -qnoflttrap
	Generate calls to detect and trap floating point. The available options are: overflow, underflow, zerodivide, invalid, inexact, enable, imprecise.	gen(floatTrap, *option*[, yes])
		gen(floatTrap,*option*, no)

-qfold
-qnofold
	Specifies that constant floating-point expressions are to be evaluated at compile time.	gen(float, fold[, yes])
		gen(float, fold, no)

-qfullpath
-qnofullpath
	Specify path information.	link(debugFullPath[, yes])
		link(debugFullPath, no)

-qfuncsect
▶ C -qnofuncsect
	Place instructions for each function in a separate cset.	gen(funcSect[, yes])
		gen(funcSect, no)

▶ C++ -qgenpcomp	Generate pretokenized include files.	N/A

Old Option	**Description**	**Configuration File Option**

C -qgenpcomp[=dirname]	Generate pretokenized include files.	file(genPCH[, yes])
		file(genPCH, *string*)
		file(genPCH, no)

C++ -qgenproto	Generate ANSI prototypes form K&R function definitions.	N/A

C -qgenproto=<*parmnames*>	Produces ANSI prototypes from K&R function definitions.	file(genProto[, yes])
		file(genProto, no)

C++ -qhalt C -qhalt=<*w* | *e* | *s* | *u*>	Stop compiler after first phase if severity of errors detected equals or exceeds <sev>; the severity lievel is one of w, e, s, u.	N/A
-qhd	Generate additional information for use with the Heapview debugger.	N/A

C -qheapdebug
C -qnoheapdebug
	Enables debug versions of memory management functions.	alloc(debug[,yes])
		alloc(debug,no)

-qidirfirst
-qnoidirfirst
	Specify search order for files.	incl(searchpathfirst[, yes])
		incl(searchpathfirst, no)

-qignerrno
-qnoignerrno
	Tell optimizer that program will never refer to or set errno.	opt(ignErrno)
		opt(ignErrno, no)

C++ -qignprag
C -qignprag=disjoint
C -qignprag=isolated
C -qignprag=all
	Honour or ignore references to #pragma disjoint and/or isolated_call.	opt(pragmaDisjoint[, yes])
		opt(pragmaDisjoint, no)
		opt(pragmaIsolatedCall[, yes])
		opt(pragmaIsolatedCall, no)

C++ -qinfo=*infoopt* -qinfo=<*class*>	Produce additional lint-like messages based on <class>.	info()

C++ -qinitauto=*value*
C -qinitauto=<*hh*>
	Initialialize automatic storage to <hh>. <hh> is a hexadecimal value. This generates extra code and should only be used for error determination.	gen(initAuto[, yes])
		gen(initAuto, *number*)
		gen(initAuto, no)

-qinlglue
-qnoinlglue
	Generate fast external linkage by inlining the code (pointer glue code) necessary at calls via a function pointer and calls to external procedures.	gen(inlinePointerGlue[, yes])
		gen(inlinePointerGlue, no)

C++ -qinline
C++ -qnoinline
	Determine whether functions in your code qualified with the _Inline or inline keywords are candidates for inlining.	opt(inline[, yes])
		opt(inline, no)

C++ -qipa
C++ -qnoipa
C++ -qipa=*option*
	Turn the IPA optimizer on or off.	opt(ipa)
		opt(ipa, no)
		opt(ipaoption, *option*)

C -qipa[=<*ipa_options*>]	Turn on inter-procedural analysis.	N/A
C -qisolated_call= <*name1*>:<*name2*>:...	Specify that the calls to the functios listed have no side effcts. <name1> and <name2> are function names. The user may specify as many function names as necessary.	opt(isolatedCall,...)

Old Option	Description	Configuration File Option
C++ -qlanglvl	Set language level.	N/A
C -qlanglvl=<*ansi* | *saa12* | *saa* | *extended* | *classic*>	Specify language level to be used during compilation. <langlvl> can be ansi, saa12, saa, extended, or classic.	lang(level,...)

-qldbl128
-qnoldbl128
-qlongdouble
-qnolongdouble
	Represent long doubles as 80 bit values on OS/2 and Windows®, or as 128 bit values on AIX.	gen(longDouble[, yes])
		gen(longDouble, no)

-qlibansi
-qnolibansi
	Process ANSI C library names as system functions.	gen(libansi)
		gen(libansi, no)

C -qlinedebug
C -qnolinedebug
	Generates abbreviated line number and source file name information for the debugger.	gen(lineOnlyDebug[,yes])
		gen(lineOnlyDebug,no)

-qlist
C -qnolist
	Produce an object listing; output goes to .lst file.	list(listing[,yes])
		list(listing,no)

-qlistopt C -qnolistopt	Display the settings of all options; output goes to .lst file.	list(options[, yes])

-qlonglong
-qnolonglong
	Processor disallow long long int.	lang(longlong)
		lang(longlong, no)

-qmacpstr
C -qnomacpstr
	Allow '\p' as the first character of a string constant.	lang(macPStr[, yes])
		lang(macPStr, no)

-qmakedep
C -qnomakedep
	Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.	file(makeDep[,yes])
		file(makeDep,no)

C -qmaxerr	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.	debug(maxErrors, *number*)
-qmaxmen=<*num*>	Limit the amount of memory used by space intensive ooptimization to <num>. <num> specified in kilobytes. priority=NNN	opt(maxMem, *number*)

-qmbcs
C -qnombcs
	Allow use of DBCS.	lang(dbcs[, yes])
		lang(dbcs, no)

-qnoprint	Direct listing to /dev/null.	list(listing, no)
-qnostdinc C -qstdinc	Specify which files are included with the #include "file_name" and #include <file_name> directives. If -qnostdinc is specified, the /usr/include directory is not searched.	incl(standardInclude, no)

-qonce
C -qnoonce
	Process #include files only once.	file(once[, yes])
		file(once, no)

-qpascal
C -nopascal
	Accept and ignore the keyword 'pascal' as a type modifier.	lang(pascal[, yes])
		lang(pascal, no)

Old Option	Description	Configuration File Option
C -qpdf1 C -qpdf2	Perform aggressive optimizations with profile directed feedback.	N/A
-qphsinfo	Produce compiler phase information	N/A
C++ -qpriority=<*num*>	Specifies the priority level for initialization of static constructors.	link(sharedLibPriority, *number*)

C -qproclocal
C -qproclocal= <*name1*>:<*name2*>:...
	Specify which functions are local. If no filenames are specified, all invoked functions are assumed to be defined within the current file. The last explicit specification for a function takes precedence.	link(procLocal,<*none* | *all* | *names*>)
		link(procLocalName, *string*)

C -qprocimported
C -qprocimported= <*name1*>:<*name2*>:...
	Specify which functions are imported. If no filenames are specified, all invoked functions are assumed to be defined outside the current file. The last explicit specification for a function takes precedence.	link(procImport,<*nome* | *all* | *names*>)
		link(procImportName,*string*)

C -qprocunknown
C -qprocunknown= <*name1*>:<*name2*>:...
	Specify which functions are unknown to be local or imported. If no filenames are specified, all functions called are assumed to be unknown. This is the default when no user options are specified. The last explicit specification for a function takes precedence.	link(procUnknown,<*none* | *all* | *names*>)
		link(procUnknownName, *string*)

-qproto
C -qnoproto
	Assert that procedure call points agree with their declarations even if the procedure has not been prototyped. This allows the caller to pass floating point arguments floating point registers instead of general purpose registers.	lang(checkNonProto[,yes])
		lang(checkNonProto,no)

-qro
-qnoro
	Put string literals in read only area.	gen(readonly[, yes])
		gen(readonly, no)

-qroconst
-qnoroconst
	Put constant values in read only area.	gen(readonlyconst)
		gen(readonlyconst, no)

C -qrwvftable C -qnorwvftable	Place virtual function tables into read/write memory.	N/A

-qshowinc
C -qnoshowinc
	Include the source for all included files in the source listing, if the -qsource option is in effect.	list(expAllInc[,yes])
		list(expAllInc,no)

C++ -qsom	Turn on implicit SOM mode.	N/A
C++ -qsomgs	Disable direct access to SOM attributes.	N/A
C++ -qsominc	Include or exclude files in implicit SOM mode.	N/A
C++ -qsomro	Write the release order of the specified class to stdout.	N/A

Old Option	**Description**	**Configuration File Option**
-qsource C -qnosource	Produce a source listing; output goes to.lst file.	list(incSource[,yes]) list(incSource,no)
C++ -qspill=*number* -qspill=<*size*>	Specify the size of the register allocation spill area.	opt(registerSpillSize, *number*)
-qsrcmsg C -qnosrcmsg	Reconstruct source lines in error along with the diagnostic messages.	report(srcMsg[, yes]) report(srcMsg, no)
-qstatsym -qnostatsym	Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of xcoff objects).	link(staticSymbols) link(staticSymbols, no)
-qstrict -qnostrict	Valid only at -03. This option turns off aggresive optimizations which have the potential to alter the semantics of a user's program. This option also sets -qfloat=nofltint:norsqrt.	opt(strict) opt(strict, no)
C -qsyntaxonly C -qnosyntaxonly	Causes the compiler to perform syntax checking without generating an object file.	file(syntaxOnly[,yes]) file(syntaxOnly,no)
-qtabsize=<*size*>	Change the length of tabs in your source file.	file(tabSize, *number*)
C++ -qtbtable=*option* -qtbtable=<*option*>	Generate a traceback table for each function and place it in the text segment at the end of the function code. The available options are: none= No traceback table is generated. full= A full traceback table is generated. This is the default with -g specified. small= A traceback table is generated with no name or parameter information. This is the default when -g is NOT specified.	gen(traceBackTable,option) link(traceBackTable, <*none*|*full*|*small*>)
-qtempinc -qtempinc=<*directory*> C -qnotempinc	Automatically generate template functions in the specified directory. Use the "notempinc" option to suppress this behaviour.	N/A
C++ -qtempmax	Specify the maximum number of files to be created in the tempinc directory for each template file.	N/A

Old Option	Description	Configuration File Option

C++ -qtune=*name*
C -qtune=<*403 | 601 | 603 | 604 | pwr | pwr2 | pwr2s*>
	Specifies the architecture system for which the executable program is optimized. The available options are:
	403= Produce an object optimized for all the PowerPC 403™ processors. Use -qfloat=emulate to emulate the floating-pointing instructions omitted by the 403 processor.
	601= Produce an object optimized for all the PowerPC 601® processors.
	603= Produce an object optimized for all the Power PC 603® processors.
	604= Produce an object optimized for all the PowerPC 604™ processors.
	pwr= Produce an object optimized for the POWER hardware platform.
	pwr2= Produce an object optimized for the POWER2 hardware platform.
	pwr2s= Produce an object optimized for the POWER2 hardware platform, avoiding certain quadruple-precision instructions that would slow program performance.
	pwrx= Same as pwr2.
		opt(tune, *name*)
		opt(tune,...)

-qtwolink C -qnotwolink	Link twice in order to minimize the number of static constructors included form libraries (default with langlvl=compat).	N/A

-qunique
-qnounique
	Generates unique names for static constuctor/descructor file ocmpilation units.	link(uniqueNames[, yes])
		link(uniqueNames, no)

C++ -qunroll
C++ -qunroll=*number*
-qnounroll
C -qunroll[=n]
	Allow the optimizer to unroll loops.	opt(loopUnroll[, yes])
		opt(loopUnroll, *number*)
		opt(loopUnroll, no)

-qupconv
-qnoupconv
	Preserves the unsigned specification when performing integral promotions.	lang(preserveUnsigned Promotion[, yes])
		lang(preserveUnsigned Promotion, no)

C++ -qusepcomp	Use pretokenized include files.	N/A

C -qusepcomp[=dirname]	Use pretokenized include files.	file(usePCH[,yes])
		file(usePCH,*string*)
		file(usePCH,no)

-qvftable
-qnovftable
	Determine whether the virtual function table is included in the module that is the target of the build.	gen(vft[, yes])
		gen(vft, no)

C++ -qwait C -qwait[=<seconds>]	Specify the number of seconds to wait for a NetLS license to become available. The default is to wait forever.	N/A

C -qxcall
C -qnoxcall
	Generate code to static routines within a compilation unit as if they were external routines.	gen(externStaticLinkage[, yes])
		gen(externStaticLinkage, no)

Old Option	Description	Configuration File Option

-qxref
C -noxref
	Produce a cross-reference listing; output goes to .lst file.	list(minXRef[, yes])
		list(minXRef, no)

C -qxref=full	Produce a cross-reference listing containing all names, whether referenced or not; output goes to .lst file.	list(fullXRef[, yes])
		list(fullXRef, no)

-r	Permits the output file to be produced even if it contains unresolved symbols.	link(force)

C++ -s		link(strip[, yes])
		link(strip, no)

-S	Generate an assembler language (.s) file	file(genAsm[,yes])
C++ -S*number*	Set the total size of the program stack in bytes.	Windows NT® and OS/2 only: link(stack, *number*)
C++ -t	Apply prefix from the -B option to the specified program.	N/A

C -t<*p/c/b/i/a/l/m/f/I*>	Apply prefix form the -B option to the specified program <x>, where x can be on or more of the following:
	p= preprocessor
	c= compiler frontend
	b= compiler backend
	i= compiler inliner
	a= assembler
	l= linkage editor
	m= ’munch’ utility
	I= inter procedural analysis
		N/A

C++ -T*number*	Sets the start address of the text section of the output file to *number*.	link(textStart, *number*)
-U<*name*>	Undefine name as in #undef directive.	undefine(*name*)
-v	Display language processing commands as they are invoked by xIC; output goes to stdout.	N/A
-w	Suppress information, language-level, and warning messages.	report(level, ...)
C++ -W	Give specified options to specidied compiler program.	N/A
C -W<*x,y*>	Give the options <y> to the compiler program <c> where x can be one or more of the following: <p,c,b,i,a,l,I,m>	N/A
C++ -y<*option*>	Specifies compile-time rounding of constant floating-point expressions. (is this the right description?)	gen(roundConstFp, *option*)

Old Option	Description	Configuration File Option
C -Y<*x*>	Specify compile-time rounding of constant floating-point expressions, where <x> can be one of the following: n= round to nearest m= round toward minus infinity p= round toward positive infinity z= round toward zero.	gen(roundConstFp, *option*)
C++ -Z*string*	Prefix the names of the library search paths with the *string*.	link(libsearchpathprefix, *string*)

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Configuration Files
Setting Build Options

RELATED TASKS

Build

RELATED REFERENCES

Build Options

C++ Build Options for ANSI C++ Compliance

C++ Build Options for Compatibitility with Old C++ Compilers

Link Options from Earlier Versions of VisualAge C++ (AIX)

VisualAge C++ includes an incremental linker that replaces the traditional AIX linker (ld). 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.

In previous versions of VisualAge C++, we recommended that you issue all link edit commands to ld through the xlC command. Therefore, the table in Compile Options from Earlier Versions of VisualAge C++(AIX) provides the new equivalents to the earlier link options.

RELATED CONCEPTS

Incremental C++ Build
C Compilation
Set Build Options

RELATED TASKS

Build

RELATED REFERENCES

Compile Options from Earlier Versions of VisualAge C++ (AIX)

New Standard C++ Library Header Files

The Standard C++ Library is composed of eight special-purpose libraries:

- The Language Support Library
- "The Diagnostics Library"
- "The General Utilities Library" on page 68
- "The Standard String Templates" on page 68
- "Localization Classes and Templates" on page 68
- "The Containers, Iterators and Algorithms Libraries (the Standard Template Library)" on page 68
- "The Standard Numerics Library" on page 69
- "The Standard Input/Output Library" on page 69
- C++ Headers for the Standard C Library (page 70)

C++ implementations that do not conform to the C++ International Standard, including versions of VisualAge C++ prior to Version 4.0, may not provide support for the Standard C++ Library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use the Standard C++ Library.

The Language Support Library

The Language Support Library defines types and functions that will be used implicitly by C++ programs that employ such C++ language features as operators new and delete, exception handling and runtime type information (RTTI). To maintain compatibility with versions of this product prior to Version 4.0, do not use the standard headers in the left column of the table below. Instead, use the non-standard headers in the right column, which have been provided with VisualAge C++, Version 5.0. for compatibility with previous releases.

Standard C++ header	**Equivalent in previous versions**
<exception>	AIX <stdexcept.h> WIN OS/2 <stdexcpt.h>
<limits>	no equivalent
<new>	<new.h>
<typeinfo>	<typeinfo.h>

The Diagnostics Library

The Diagnostics Library is used to detect and report error conditions in C++ programs. To maintain compatibility with versions of this product prior to Version 4.0, do not use the standard headers in the left column of the table below. Instead, use the non-standard headers in the right column, which have been provided with VisualAge C++, Version 5.0. for compatibility with previous releases.

Standard C++ header	**Equivalent in previous versions**
<stdexcept>	AIX <stdexcept.h> WIN OS/2 <stdexcpt.h>

The General Utilities Library

The General Utilities Library is used by other components of the Standard C++ Library, especially the Containers, Iterators and Algorithms Libraries (the Standard Template Library). C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the General Utilities Library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use this library.

Standard C++ header	Equivalent in previous versions
<utility>	no equivalent
<functional>	no equivalent
<memory>	no equivalent

The Standard String Templates

The Strings Library is a facility for the manipulation of character sequences. C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the Strings Library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use this library.

Standard C++ header	Equivalent in previous versions
<string>	no equivalent

Localization Classes and Templates

The Localization Library permits a C++ program to address the cultural differences of its various users. C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the Localization Library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use this library.

Standard C++ header	Equivalent in previous versions
<locale>	no equivalent

The Containers, Iterators and Algorithms Libraries (the Standard Template Library)

The Standard Template Library (STL) is a facility for the management and manipulation of collections of objects. C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the STL. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use this library.

Standard C++ header	Equivalent in previous versions
<algorithm>	no equivalent
<bitset>	no equivalent
<deque>	no equivalent

Standard C++ header	Equivalent in previous versions
<iterator>	no equivalent
<list>	no equivalent
<map>	no equivalent
<queue>	no equivalent
<set>	no equivalent
<stack>	no equivalent
<vector>	no equivalent

The Standard Numerics Library

The Numerics Library is a facility for performing seminumerical operations. C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the Numerics Library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, do not use this library.

Users who require library facilities for complex arithmetic but wish to maintain compatibility with older compilers may use the compatibility complex numbers library whose types are defined in the non-standard header file <complex.h>. Although the header files <complex> and <complex.h> are similar in purpose, they are mutually incompatible.

Standard C++ header	Equivalent in previous versions
<complex>	no equivalent
<numeric>	no equivalent
<valarray>	no equivalent

The Standard Input/Output Library

The standard iostreams library differs from the compatibility iostreams in a number of important respects. C++ implementations that do not conform to the C++ International Standard, including versions of this product prior to Version 4.0, may not provide support for the standard iostreams library. To maintain compatibility between such a product and VisualAge C++, Version 5.0, use instead the compatibility iostreams library. More information on iostreams is provided in Special Considerations When Using C++ I/O.

Standard C++ header	Equivalent in previous versions
<fstream>	no equivalent
<iomanip>	no equivalent
<ios>	no equivalent
<iosfwd>	no equivalent
<iostream>	no equivalent
<istream>	no equivalent

Standard C++ header	Equivalent in previous versions
<ostream>	no equivalent
<streambuf>	no equivalent
<sstream>	no equivalent

C++ Headers for the Standard C Library

The C International Standard specifies 18 headers which must be provided by a conforming hosted implementation. The name of each of these headers is of the form *name*.h. The C++ Standard Library includes the C Standard Library and, hence, includes these 18 headers. Additionally, for each of the 18 headers specified by the C International Standard, the C++ standard specifies a corresponding header that is functionally equivalent to its C library counterpart, but which locates all of the declarations that it contains within the std namespace. The name of each of these C++ headers is of the form c*name*, where *name* is the string that results when the .h extension is removed from the name of the equivalent C Standard Library header. For example, the header files <stdlib.h> and <cstdlib> are both provided by the C++ Standard Library and are equivalent in function, with the exception that all declarations in <cstdlib> are located within the `std` namespace.

C++ implementations which do not conform to the C++ International Standard may not support the "c*name*" headers. To maintain source code compatibility between such an implementation and VisualAge C++, Version 5.0, do not use these headers. Instead, use the corresponding C Standard Library header whose name is of the form "*name*.h."

Standard C++ Header	Corresponding Standard C & C++ Header
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>
<cwchar>	<wchar.h>
<cwctype>	<wctype>

RELATED CONCEPTS

Special Considerations When Using C/C++ I/O

Use the Standard Iostreams Library and the Compatibility Iostreams Library

It is possible to use the standard iostreams library and the compatibility iostreams library together. To do so, it is better to fully qualify the standard iostreams library names that are needed, than to use C++ `using` directives to access the standard iostreams library. For example:

```
#include <iostream>
#include <iostream.h>
int main(void) {
   cout << "I can hear music\n"; // Line 1
   std::cout << "Sweet, sweet music\n"; // Line 2
}
```

In this example, line 1 will use the compatibility iostreams library, which is declared in the global namespace, while line 2 will use the standard iostreams library, which is declared in the std namespace.

RELATED CONCEPTS

Special Considerations When Using C/C++ I/O

Use the Standard Iostreams Library with the IBM Open Class Library

Version 5 of the IBM Open Class library includes a new header, istream.hpp. This header allows you to choose the iostream library you want to use through a macro, __IOC_ANSI_STREAM. For example, a truly portable IBM Open Class application is written this way:

```
#include <iostream>
#include <istring.hpp>
#ifdef __IOC_ANSI_STREAM
using namespace std;
#endif
IString is("I can hear music");
cout << is << endl;
```

In your makefile or configuration file, simply define __IOC_ANSI_STREAM if you want to use the ANSI stream library. By default, the IBM Open Class library uses the USL library to preserve existing behavior.
IOC libraries are iostream independent. This means that you can use either iostream library, with one exception: the File stream library. Version 5 of the IBM Open Class library includes two File stream libraries: one compiled with the ANSI stream library (libvacfastrm.a on AIX platforms, or cpposi50.dll on OS/2 or Windows) and the other compiled with the USL stream library (libvacfstrm.a or cppofi50.dll).

RELATED CONCEPTS

Special Considerations when Using C/C++ I/O

RELATED TASKS

"Use the Standard Iostreams Library and the Compatibility Iostreams Library"

Makefile Rules Mapped to Configuration File Directives

The following table shows configuration file directives and equivalent makefile rules.

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

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

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

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

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

RELATED CONCEPTS

Configuration Files

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Getting Started book, or our PDF documents. Send your e-mail to: torrcf@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

http://www.ibm.com/support/ describes IBM Support Offerings on all platforms, worldwide.

http://www.ibm.com/rs6000/support/ describes support offerings on the RS/6000® platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

http://www.lotus.com/passport describes the IBM and Lotus® Passport Advantage™ contracting option.

The IBM Software Support Handbook, accessible from http://www.ibm.com/software/support, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- **United States**: 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- **Canada**: 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

- Elsewhere, please contact your local IBM office.

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred

- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere™ Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit http://www.ibm.com/software/ad/vaws-services/ or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com