VisualAge® C++ Professional for AIX®

Getting Started

Version 5.0

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

First Edition (February 2000)

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

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

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.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

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

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

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered tradmark of Ziff Communicatoins Company and is used by IBM Corporation under license

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

Industry Standards

VisualAge C++ 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

Much of the information in this book is also available in the online help.

To find this information, or any topics listed in this book 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 book. In such cases, there is a cross-reference to the page on which the related topic appears.

VisualAge C++, Version 5.0 Getting Started

Chapter 1. Getting Started with VisualAge C++

Welcome to *Getting Started with VisualAge C++*. This book is meant to help you get up and running with VisualAge C++ and give you an idea of what VisualAge C++ can do.

This book is divided into five sections:

- **What's New in VisualAge C++**

  This portion outlines changes made to VisualAge C++ since the previous release. See "Chapter 2. What's New in VisualAge C++?" on page 5.

- **What Is VisualAge C++?**

  What is VisualAge C++? introduces the overall capabilities of the VisualAge C++. It tells you about the tools available to you and some of their features. It will take you 20 to 30 minutes to read all of the sections. See "Chapter 3. What Is VisualAge C++?" on page 11

- **Installing VisualAge C++**

  This section will guide you through installing VisualAge C++. It includes information about system requirements, installing and licensing VisualAge C++. This section starts with "Introduction to Installing VisualAge C++" on page 27.

- **Tour VisualAge C++**

  Tour VisualAge C++ gets you working with the new integrated development environment (IDE). In 20 to 30 minutes, the tour will guide you step-by-step through some basic tasks to create and run a small program. This is a good way to prepare for the more in-depth tutorial, or learn to navigate before starting your own projects. This section starts with "About the Tour" on page 35.

- **Try VisualAge C++**

  This is an introduction to configuration files and features of the new Integrated Development Environment. In four modules of about 20 minutes each, you will create a configuration and a project, learn about configuration language and project language, and optimize a subproject. The tutorials start with "Introduction to VisualAge C++ Configurations and Subprojects" on page 59.

This book also includes the Migration Guide for VisualAge C++ Professional for AIX, Version 5.0. The Migration Guide helps you to move projects from your present compiler to either the incremental compiler or the batch compiler that are part of the VisualAge C++ suite of tools. The Migration Guide starts with "Chapter 7. Prepare to Migrate to Version 5.0" on page 97.

Chapter 2. What's New in VisualAge C++?

This release of VisualAge C++ Professional Version 5.0 includes:

New Tools and Features

A Batch Compiler

VisualAge C++ now includes a traditional, makefile-based compiler that is fully ANSI/ISO compliant. The new batch compiler supports both 32- and 64-bit architecture and optimization technology. SMP architecture is supported through C OpenMP support, SMP explicit directives and automatic parallelization.

64-Bit Support

Compilers
: VisualAge C++ now offers full support for creating 64-bit programs. Both the batch and incremental compilers each support both 32- and 64-bit optimization technology.

Debugging
: Both the stand-alone debugger and the IDE debugger have seamless 64-bit debugging. Debugging your 64-bit application is the same as debugging a 32-bit application. No special steps are needed to start either debugger. You can also debug a 32-bit process and start debugging another 64-bit debug process at the same time. Views will show 32- or 64-bit addresses as appropriate.

IBM Open Class
: IBM® Open Class™ applications that do not use user interface or 2D graphics classes can now be built in 64-bit mode as well.

Detailed Help for Upgrading to VisualAge C++ Version 5.0

The Migration Guide for VisualAge C++ Version 5.0 provides help on moving to either the batch compiler or the incremental compiler that are part of VisualAge C++ Professional Version 5.0. It provides information on migrating your existing programs to VisualAge C++ Professional Version 5.0 from any of the following compilers:

- VisualAge C++ Professional Version 4.0 (incremental compiler)
- IBM C and C++ Compilers Version 3.6 (batch compiler)
- C Set ++® for AIX Version 3.1 (batch compiler)

We strongly encourage you to read the migration guide before trying to migrate your existing projects to VisualAge C++ Professional Version 5.0.

For your convenience, we have included a copy of the Migration Guide for VisualAge C++ Version 5.0 in this book. See "Chapter 7. Prepare to Migrate to Version 5.0" on page 97.

A Stand-Alone Debugger

A stand-alone debugger, the Distributed Debugger, is shipped with VisualAge C++. The Distributed Debugger lets you debug applications containing traditional debug information. You can debug programs running on your local machine or a remote machine accessible from a TCP/IP network.

A Performance Analyzer

Now you can use the VisualAge C++ IDE to analyze the performance of programs by tracing the execution of a program and creating a trace file. The trace file contains trace analysis data that can be examined in several views. Using these views, you can improve the performance of a program, examine a sequence of calls leading up to an exception, and in general, understand what happens when a program runs. The IDE can trace and analyze applications generated by the VisualAge C++ Version 5.0 compiler. You cannot use the IDE to trace files produced by other compilers.

Improved Run-Time Performance

A new C++ object model improves run-time performance. You save space with virtual bases and virtual calls are faster when using the IBM object model.

Extension APIs

The incremental compiler internals are now available to users via a full featured set of interfaces. Customized tools such as a *lint*-like style checker can be easily written to your own specifications and integrated with the compilation process as extensions or made as stand-alone tools.

Four types of extension interfaces are available:

- *Observer extensions* let you register observers that are notified when specific events in the compile cycle are encountered

- *Incorporation extensions* let you add to or modify compilation phases
- *Dependency graph extensions* let you add data to the codestore, with incremental update
- *User interface extensions* let you add views, buttons, and menu items to the VisualAge C++ IDE

Improvements to the Incremental Compiler

Easier Maintenance of Source Code Compatibility

VisualAge C++ now allows you to take advantage of the productivity gains of using the incremental compiler while maintaining code compatibility with other traditional C++ compilers. The ordered name lookup option makes the compiler try to resolve all names to the same declarations that a traditional compiler would. In other words, the option helps you write portable code by maintaining all the #include directives and forward declarations necessary to compile your source with a compiler other than VisualAge C++.

Multiple Codestores In A Single Project

The new structure of projects allows you to group many codestores in a single project. You can debug, browse, and build code for your executables and shared libraries at the same time. This allows you better scaling, so you can manage programs in smaller chunks. Rebuilds are quicker since dependencies between codestores are tracked at a different level than dependencies within codestores. The debugger for executables is very useful, because when your executable calls for a function in another shared library, you can access another codestore for that shared library.

Enhancements to the Integrated Development Environment

Easier Transition from Makefiles

Configuration Smart Guide

The Configurations page has a check box to disable configuration directives, which makes migration easier. Migration help is always close at hand, thanks to the new Browse Migration Guide button. Even when you are in the middle of the SmartGuide, you can jump directly to the most relevant information on the task in the online version of the Migration Guide for VisualAge C++ Version 5.0. For your convenience, the Migration Guide is also included in this book. It starts with “Chapter 7. Prepare to Migrate to Version 5.0” on page 97.

Change Configuration Options View

You can now easily locate the batch equivalents of configuration options and vice versa through the VisualAge C++ IDE.

Configuration Views

Configuration views now allow you to select directives to enable or disable. For example, if you are having migration problems with particular source files, you can select the source directives in a configuration view and disable them until you are ready to include the files in the build.

Convert Batch Options View

Use the Convert Batch Options View to convert or display equivalent batch and incremental options. You no longer need to look through lengthy reference tables to find equivalent options.

Source Code Control

It's now easy to check files into or out of virtually any source control system. Both IBM CMVC and IBM VisualAge TeamConnection™ are supported with custom interfaces within the VisualAge C++ IDE, but you can issue commands to any other version control software from within the development environment. Updated file management views let you see which files are locked for easy, safe collaboration with other members of your development team.

Expanded Selection of Editors

In addition to the default LPEX editor, you can customize Source views in the IDE to adopt the style and command set of your favorite keyboard mapping editors. The two new keyboard mappings are vi and emacs, which are widely used editors, common for AIX.

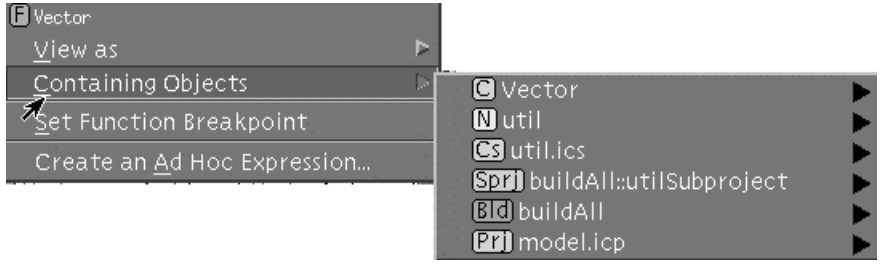
If you prefer to work with a stand-alone editor, rather than using the Source views in the IDE, you can set the IDE to update its views from your editor. You can also choose to launch the external editor whenever a page containing a Source view is displayed or a Source view is updated.

Better File Management Support

Now you can use the Host section of the IDE workbook to take care of file-oriented tasks without leaving the development environment: create, delete, copy or rename files and directories and preview the contents of your files in a simple, graphical interface.

Quick Context Finder

The **Containing Objects** menu item makes it very easy to see at any time where any function, object, or file fits into the bigger picture. For example, for a function, you can see what class it is in, the namespace, the codestore, the subproject, the build and the project.



Improved Support for Creating Classes

The new Class SmartGuide enables you to create a skeleton for a new class and use the information that already exists in the codestores. This saves you from having to create declarations and remember the names for this section. For example, you can pick the base classes for your new class from a list generated from information in the codestore and not have to worry about remembering their exact spellings. You can even tell the SmartGuide to imitate your personal coding style.

Chapter 3. What Is VisualAge C++?

VisualAge C++ is a flexible, innovative suite of tools designed to offer a rich environment for object-oriented application development. It is especially valuable for high-performance and heavily computational applications.

The VisualAge C++ suite of tools consists of:

ANSI/ISO Compliant Batch and Incremental Compilers

Two compilers offer you incredible flexibility. You can use the batch compiler to move projects up to the ANSI/ISO 1998 C++ standard, including STL, with no project delays or disruptions. The incremental compiler offers you the ability to move up to the new standards while taking advantage of the productivity gains from compiling only parts of program affected by changes made since the last compile and the tight integration between the incremental compiler and the IDE. The incremental compiler lets you do this while still maintaining source code compatibility with your makefile-based projects.

A Highly Customizable Integrated Development Environment

The VisualAge C++ IDE is a powerful development environment that takes advantage of a database of program information, the codestore, to give you greater understanding of your programs. This architecture allows you to set breakpoints from within the editor, instantly browse the class hierarchy, modify the source code as you debug, and easily find references for a method inside a complete project as you edit, compile and debug your code in one spot.

A Stand-Alone Debugger

Both the debugger in the VisualAge C++ IDE and the stand-alone debugger allow you to debug programs compiled with traditional debug information. The stand-alone debugger, the Distributed Debugger, also lets you debug programs running on remote machines accessible through a TCP/IP network. You can use the Distributed Debugger when you want to debug a program, but do not need access to the full range of information the VisualAge C++ IDE debugger provides.

Visual Programming Environments

The VisualAge C++ visual programming environments, Visual Builder and Data Access Builder, let you build programs from parts. Using Visual Builder,

you simply drag and drop, and connect visual parts. Visual Builder creates the code for you. With Data Access Builder, you can use the drag-and-drop interface to create your database mappings, and Data Access Builder creates the source code for you. Any parts you create in Data Access Builder can then be used in Visual Builder. Source code generated by Visual Builder and Data Access Builder can be compiled by both the incremental and the batch compilers.

Class Libraries

The IBM Open Class Library provides advanced class libraries and frameworks to build rich and robust applications. It is fully ANSI compliant and you can use the ANSI Standard Template Library (STL) containers in an IBM Open Class application.

A Comprehensive Help System

Using the VisualAge C++ comprehensive, online help system can significantly reduce your learning curve when starting to use VisualAge C++. Our online, HTML-based help provides you with information on every component in VisualAge C++ and includes tutorials on how to use many of the tools. It is optimized for fast access to information, and provides fuzzy and precise full-text search. Most information is also provided in the Adobe Acrobat PDF format.

For more information on these tools, see:

- **"Compilers"**
- **"Integrated Development Environment" on page 21**
- **"Stand-Alone Debugger" on page 23**
- **"Visual Programming Environments" on page 24**
- **"Class Libraries" on page 25**
- **"Help System" on page 26**

Compilers

VisualAge C++ gives you two compilers: a batch compiler and an incremental compiler. This offers you the flexibility to maintain legacy projects with the batch compiler, while taking advantages of the productivity gains offered by the incremental compiler for new projects.

Both VisualAge C++ compilers support the ANSI/ISO 1998 C++ standard.

While adhering to the new C++ standard, the VisualAge C++ compilers provide options that allow for non-standard legacy C++ features. As a result, you can easily move your applications to the new standards.

Batch Compiler

The VisualAge C++ batch compiler is a traditional, makefile-based compiler that is fully ANSI/ISO 1998 compliant. The batch compiler is compatible with previous IBM C/C++ batch compilers, making migration and porting to the VisualAge C++ batch compiler easy.

The batch compiler also supports both 32 and 64 bit optimization technology. SMP architecture is supported through C OpenMP support, SMP explicit directives and automatic parallelization.

Incremental Compiler

The VisualAge C++ incremental compiler uses new program understanding technology to increase your productivity. Using the incremental compiler offers you the following advantages:

- Fast Incremental compilation
- Reduced drudgery
- Elimination of makefiles
- Automatic instantiation of templates

While taking advantage of the productivity gains of using the incremental compiler, the VisualAge C++ incremental compiler allows you to maintain code compatibility with other traditional C++ compilers. The ordered name lookup option makes the incremental compiler resolve all names to the same declarations that a traditional, makefile-based compiler would. In other words, the option helps you write portable code by maintaining all the #include directives and forward declarations necessary to compile your source with a compiler other than VisualAge C++.

The VisualAge C++ incremental compiler is compliant to ANSI/ISO 1998 standard including the most complete implementation of STL. It also supports both 32 and 64 bit optimization technology.

Fast Incremental Builds

With the VisualAge C++ incremental compiler you no longer have to wait for an entire project rebuild each time you make a change. You only have to wait as long as necessary for the parts of your program affected by the change to be rebuilt.

Incremental Builds
Every time you build, the VisualAge C++ incremental compiler rebuilds only what it needs to rebuild. With traditional compilers, header files are recompiled every time a source file that includes them is recompiled. With the

VisualAge C++ incremental compiler, changes to a source file do not require recompilation of the header files that the source file includes. It does not even require the whole changed file to be recompiled, only the changed part of the file and the parts of the program that depend on that changed section.

Database of Program Information
Databases, or *codestores* of information about your program, make up the core of the VisualAge C++ incremental compiler.

When you first build your program, a codestore is created. A codestore contains information consisting of, for example, the signatures and contents of functions, and the names and types of variables.

Builds using the VisualAge C++ incremental compiler can be extremely fast. If you add one line of code to a function, the update happens almost instantaneously. While a traditional makefile-based system has to rebuild all source files that have changed, along with all object files that depend on those source files, the VisualAge C++ codestore records exactly which functions need rebuilding, and rebuilds only those particular functions.

Early Error Feedback
Because of incremental compilation, the VisualAge C++ incremental compiler can provide rapid feedback. The compiler checks all interfaces before compiling function bodies and variable definitions. The result is a much faster error-reporting process than that of conventional compilers.

Freedom from Dependencies
All compilation dependencies are maintained automatically in the codestore. You no longer have to worry about maintaining complex header files and makefiles.

Reduced Drudgery

With the VisualAge C++ incremental compiler you can avoid mundane programming tasks such as:

- Typing and maintaining multiple copies of essentially the same declarations in multiple places
- Organizing header files
- Having to avoid circularities in header file inclusions, especially when using inline functions
- Organizing header files to minimize recompiling when header files are changed

With the VisualAge C++ incremental compiler, you can organize source code into files in any order. By default, the VisualAge C++ incremental compiler uses orderless parsing, which means that the computer sorts the declarations,

not you. You can disable this feature to maintain code compatibility with conventional, batch compilers by using the ordered name lookup option.

The VisualAge C++ incremental compiler also takes care of header files. Typically, header files require you to use various mechanisms to sort declarations for the compiler.

To realize how much productivity you will gain by letting VisualAge C++ do the work for you, take a look at a simple example that has a main() function and two classes, A and B, organized into files in this manner:

File A.h	File B.h	File main.cpp

`#ifndef A_H`
`#define A_H`
`#include "B.h"`
`class A : public B`
`{`
`public:`
`B* f() {return new B;}`
`};`
`#endif`
	`#ifndef B_H`
	`#define B_H`
	`#include "A.h"`
	`class B`
	`{`
	`public:`
	`A* g() {return new A;}`
	`};`
	`#endif`
		`#include "B.h"`
		`int main()`
		`{`
		`B b;`
		`return 0;`
		`}`

This code follows the standard method for organizing header files. Even this simple code will not compile when organized in this way. (Trace by hand the macro processing to see why.) To fix the code, you have to add forward declarations (for example, class A in B.h), and move the inline function definitions into separate files. But this is a tedious process, even for this simple example, and becomes a coding nightmare in a large project.

With VisualAge C++, you can organize your code in this way:

File A.h	File B.h	File main.cpp

`class A : public B`
`{`
`public:`
`B* f() {return new B;}`
`};`
	`class B`
	`{`
	`public:`
	`A* g() {return new A;}`
	`};`
		`int main()`
		`{`
		`B b;`
		`return 0;`
		`}`

Or, if you prefer, you can put all of the code in a single file in whatever order makes sense to you:

File main.cpp

```
class A : public B
{
public:
B* f() {return new B;}
};
class B
{
public:
A* g() {return new A;}
};
int main()
{
B b;
return 0;
}
```

Notice that there are no forward declarations, macro guards, or #include statements. VisualAge C++ eliminates the need for such tedious programming practices by maintaining a codestore database for your program, and by replacing makefiles with a configuration file that defines, among other things, the source and header files needed to compile your project.

Eliminate Makefiles

Makefiles are another source of drudgery for C++ programmers. On large projects, makefiles are difficult to manage and frequently become out-of-date, so you often need to resort to automatic dependency generators.

The VisualAge C++ incremental compiler makes the computer serve the programmer, not the other way around. When you tell VisualAge C++ incremental compiler what to build, the compiler figures out how to build it.

A project consists of one or more related C++ object files, libraries, and executables, together with the corresponding source files, processing rules, and processing options. The output object files, libraries, and executables are called *targets*, and the source files are called *sources*. A project will often contain different configurations, which are different versions of the same project, but with different sets of options.

A configuration file defines a project's configurations. The configuration file is different from a makefile in that you do not need to specify processing and inter-file dependencies for C++ files. You only need to specify source-to-target dependencies. Therefore, where a makefile says:

> "Call the C++ compiler with this source, these options, producing this target, when these header files change."

a configuration file says:

> "This is a C++ source and here are its options; this is a C++ target and here are its options; here is the list of which sources map to which targets."

Here is a simple example of a configuration file annotated with comments:

```
option
link(linkwithmultithreadlib), // Use multi-threaded library
link(linkwithsharedlib) // Use shared library
   {
   target "carlot" // Produce an executable
      {
      option
      lang(offsetofnonpodclasses), // Backward compatibility for old code
      lang(digraphs, no),
      incl(searchpath, "."),
      lang(nokeyword, "true"),
      lang(nokeyword, "false")
         {
         source type(cpp) "carlot.C" // List of C++ sources
         source type(cpp) "car.h"
         source type(cpp) "car.C"
         source type(cpp) "truck.h"
         source type(cpp) "truck.C"
         source type(cpp) "vehicle.h"
         source type(cpp) "vehicle.C"
         source type(cpp) "vlist.h"
         source type(cpp) "vlist.C"
         }
       }
     }
```

This configuration file specifies that a batch of C++ source is to be compiled and linked to produce an executable file. Some options are specified for compatibility with pre-ANSI/ISO 1998 C++ standard (for example, true and false are not treated as keywords). However, the configuration file does not indicate that vehicle.o depends on vehicle.C, vehicle.h, and so forth. In fact, because the target is an executable file, no object files are even produced.

You can build a project using different configuration files, or by processing configuration files conditionally.

The IDE provides a simple interface for creating and maintaining configuration files, so that you do not have to write or edit the files yourself. The IDE can handle any configuration file, whether it is coded by hand or generated by the IDE.

Automatic Instantiations of Templates

VisualAge C++ automatically instantiates templates, without creating the excessive code characteristic of current automatic schemes.

The VisualAge C++ incremental compiler method of instantiating templates has the following advantages:

- It stores the program in the codestore, which contains both templates and their instantiations, and holds them accessible to the compiler.
- It instantiates only the templates that are needed.
- You can browse the templates and their instantiations as a compiler would browse regular declarations, because the codestore is available to the user interface.

Many compilers support explicit instantiation under programmer control and automatic schemes. However, because of their effect on build times, code size, and dependency management with makefiles, these automatic schemes are not practical for programming projects that rely heavily on templates. You can make the manual scheme work, but the process is inconvenient. If you have tried to browse through the C++ standard template library with any development system, you probably discovered that the systems you were using could not handle the uninstantiated templates that make up most of the library. VisualAge C++, however, can.

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

Features	Benefits	Incremental Compiler	Batch Compiler
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

Features	Benefits	Incremental Compiler	Batch Compiler
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

Integrated Development Environment

The VisualAge C++ Integrated Development Environment (IDE) will improve the way you program. The design of the IDE allows you to focus on the programming task, rather than on trying to get around the development environment.

The IDE is tightly integrated and customizable to your needs. The IDE, organized in a workbook style with tabs, will make it easy for you to:

- Navigate through your code
- Find all the uses of any declaration
- See how C++ resolves overloading
- Debug at the source or at the annotated assembly level
- Set breakpoints on classes
- Add new views and tools
- View class and include hierarchies graphically
- Customize the way you build and run your projects

The new help system has been categorized into easily accessible categories of information. A powerful search engine helps you to quickly locate the information you need.

For setting up a build, connecting to a database, or performing a number of other routine tasks, VisualAge provides SmartGuides to lead you through the process. SmartGuides help you create projects, configurations, targets, and classes.

Code Development

Now, with the new Integrated Development Environment (IDE), you can develop and maintain code much more easily than ever before.

In the IDE, you can navigate through your program structure looking for type definitions, function locations, calling relationships, declaration usage, and other program elements. The IDE displays your program from any viewpoint you choose. Using the quick-search and filtering capabilities of the panes, you can quickly narrow in on the information you need.

When you want to add a member to a class or add code to a function, you do not have to search a directory of your file system to find the file containing the class declaration or function. Instead, you can select the appropriate class or function object to see the source code in a linked pane. You can choose to work with files or objects, or both.

When you build, the views of the IDE are updated to reflect the structure and relationships of your program.

You can debug multiple processes concurrently in the IDE. In other words, you can have two processes running at the same time, and debug the way they interact with each other.

In the IDE, you can fix errors quickly. When you click on an error message, the IDE displays a view of the source code causing the error. You can correct the error immediately, and then rebuild your application.

Tightly Integrated and Object-Oriented

Now, with the editor and debugger integrated, you can set breakpoints as you edit your code, and you can modify your source as you debug. In the tightly integrated IDE, access to files on your system is simply a click away.

The IDE's object-based environment allows you to view and manipulate objects, such as classes, source files, and functions, while you work on your application. As an object-based environment, the IDE forces information to be grouped in a useful, meaningful manner, and hides information that is not immediately important; the visual setup of the IDE allows you to access information, and interact with data quickly.

The IDE treats program entities as objects, and keeps information about them, so that when you select a source file, function, class, method, object, or variable, information specific to that object is available. The IDE also displays other objects related to that object, along with information specific to them.

Program Understanding Made Easy

The views of the IDE are updated to reflect the structure and relationships of your program, as you build. Do you need to know how a type is defined, or how a function is implemented? You will discover that the answers are only a click or two away.

You can easily navigate through your program structure to look for type definitions, function locations, calling relationships, declaration usage, and other program elements.

For example, you can navigate to the source for a particular method by selecting first the Classes page of the Project section of the workbook, then the class code, and then the method from the view that is displayed for that class. The source code for the selected method is displayed in the Source view.

The IDE displays your program from any viewpoint you choose. Using the quick-search and filtering capabilities of the panes, you can quickly narrow in on the information you need.

Highly Customizable

After extensive testing with C++ programmers, we designed the IDE based on programmers' feedback. You will find the new IDE an extremely powerful environment to work in. And you can modify the IDE to make it work according to your personal preferences.

When you open the IDE, you see five major sections:

- **Workbook:** Examine and set options that control the IDE itself.
- **Host:** Browse through files on your system.
- **Project:** Define your project file and control the build behavior for your project.
- **Configuration:** Define a configuration file. A configuration file is used to specify options, targets and source files.
- **Codestore:** Work with code at the source level.

Each section of the project workbook has a number of pages associated with it. When you click on a section tab, you see a page showing a "snapshot" of your system, of a project, or of a component of that project (such as a source file or a process running under debug control).

Each page in the project workbook is divided into panes. You can divide a page into any number of panes, add or remove panes, change which objects are viewed in a pane, link the panes together so that objects flow the way you want them to flow, and select among the views for any kind of object. You can also add your own pages.

Stand-Alone Debugger

VisualAge C++ includes a stand-alone debugger, the IBM Distributed Debugger, that can be used debug programs compiled with traditional debug information outside of the VisualAge C++ IDE. Use the Distributed Debugger when you do not need access to all of the views the IDE debugger supplies.

The Distributed Debugger is especially useful for remote debugging of your applications. You can debug your program running on a remote machine accessible through a TCP/IP network from your machine.

The Distributed Debugger offers an improved user interface over stand-alone debuggers previously available for AIX. This improved user interface lets you debug multiple applications from a single debugger session. Each program you debug is shown on a separate program page with a tab on each page displaying program identification information such as the name of the program being debugged.

Visual Programming Environments

Using the Visual Builder, you can visually create object-oriented programs using C++, often without having to write a single line of source code. You simply drag and drop, and connect the visual parts. Visual Builder creates the code for you.

With the Data Access Builder, you can create database access classes customized for your existing relational database tables. You use the drag and drop interface to create your database mappings, and the Data Access Builder creates the source code for you.

Through easy-to-use drag and drop interfaces, visual programming reduces your programming time and improves your code quality.

Visual Builder

Visual Builder provides an extensive library of prefabricated GUI parts that you can use to build your applications. However, you are not limited to these parts; you can extend the Visual Builder by creating and adding your own reusable parts to the parts palette. You can even import or export parts from other applications. Visual Builder parts represent actual classes. When you generate code through the Visual Builder, the parts are created as classes. You can then add your own class definitions or import them from other applications.

Visual Builder works well not only for creating interfaces, but for prototyping designs. Whether prototyping or creating an interface for your application, the process is the same: you arrange the parts, make the necessary connections, and let the Visual Builder generate the C++ code for you.

With a library of reusable components, the Visual Builder helps you to reduce programming time and improve code quality.

Data Access Builder

You can visually create mappings of database tables to C++ classes, and then let the Data Access Builder generate the C++ code for you. A quickmap feature allows you to do a column-to-attribute direct mapping. You can customize your classes to suit your needs.

You can use the classes, generated by the Data Access Builder, directly in your C++ programs. Besides generating the C++ source for your mappings, you can generate parts for your mappings, and use the mapping parts in the Visual Builder. The Visual Builder allows you to wire together database applications quickly, and efficiently.

The Data Access Builder also provides the following:

- Separate services for connecting and disconnecting from your databases

- Commit and rollback operations to handle transaction services
- Selection and retrieval of a group of objects from a datastore, which you can manipulate using the IBM Open Class Library collection classes
- Classes that support multiple connections to the datastores
- Direct support for DB2® using embedded SQL or the DB2 Call Level Interface
- Open Database Connectivity support for access to many database products using the ODBC CLI and appropriate database driver.

Because the Data Access Builder generates classes customized to your data, you can perform common database tasks, such as adding, retrieving, updating, and deleting data.

Class Libraries

Whether you are a novice programmer or an experienced developer, the IBM Open Class Library can help reduce your programming effort. The IBM Open Class Library offers you a comprehensive set of classes, ranging from basic input/output operations and string handling to user interface support.

VisualAge C++ offers you a wide variety of reusable classes to combine to create powerful applications:

- User Interface classes that provide support for building the graphical user interface of an application.
- Application Control classes that provide support for multi-threaded execution environments.
- Collection Classes that provide a set of commonly used abstract data types used to build collections.
- Streams and Storage classes that implement persistent storage mechanisms for Open Class components.
- Error Handling and Testing classes that provide support for building robust and well-behaved applications.
- 2D Graphics Framework that supports the platform-independent creation, manipulation, and rendering of 2D graphic objects.
- Text and International Frameworks that provide support for Unicode text strings and for building easily localizable components.

All IBM Open Classes are 64-bit compatible, except for the user interface and 2D graphics classes.

To find out about changes in the IBM Open Class Library, refer to the following pages in the online help:

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

Help System

Our online, HTML-based help is organized to help you quickly find the information you need. Use the navigation pane to see where you are in the information structure, and move easily from topic to topic.

The interface for the main online help uses three frames for fast and easy navigation:

1. The upper frame contains a search bar that lets you search the online help with a full-text search engine. You can get to the VisualAge C++ website by clicking on the VisualAge C++ icon (if you are connected to the internet).
2. The left frame contains an expandable index of topics.
3. The right frame displays the information for the topic you have selected.

If you know exactly what you are looking for, use the full-text search engine in the upper frame. You can use text processing, wildcards, and logical modifiers to find the information you need. You can also specify what types of information you want to search: concepts, tasks, reference, interface help, or examples.

You can browse the online help outside the tools, or from within the tools' windows. F1 always gives you context-sensitive help. All of the product's information is available from the Help menus located within the tools' windows.

Navigation through our samples has never been easier or better organized. We have provided information with each sample to help you determine which sample will best suit your needs. You can navigate to the samples with two or three quick clicks of your mouse button.

You will also find the "Chapter 7. Prepare to Migrate to Version 5.0" on page 97 in our online help and in this book. This guide provides you with help moving your projects from your present compiler to either the incremental or the batch compiler, as well as detailed reference information on changes you should be aware of.

Much of the online documentation is also available in Adobe Portable Document Format (PDF) format. You can view and print this information using the Adobe Acrobat** reader. If you do not already have the Acrobat reader program installed, you can download it free from Adobe's website at http://www.adobe.com. The Adobe PDF files are in the /usr/vacpp/pdf directory after you install VisualAge C++ Professional Version 5.0.

Chapter 4. Installing VisualAge C++

System Requirements

Hardware

Display
SVGA 800x600 (1024x764 recommended)

CD-ROM drive
Required

Mouse or pointing device
Required

Memory (RAM)
96MB minimum (128MB or higher recommended)

Disk space
Up to 750MB

Software

Operating System
IBM AIX Version 4.2 or higher

On AIX 4.3 or higher, you must have the Motif 1.2 compatibility libraries installed. They can be found in the `X11.compat.adt.Motif12` fileset.

Prerequisites for the Data Access Class Builder
A database with the appropriate ODBC database driver, or DB2 V2.1.2, or later.

To access the online help
A frames-capable browser such as Netscape Communicator Version 4.04 (or later).

Introduction to Installing VisualAge C++

VisualAge C++ Professional for AIX Version 5.0 uses License Use Management to control the license for the product. You will need to install and configure License Use Management before installing VisualAge C++. After installing VisualAge C++, you will need to enroll your license with License Use Management.

Follow these steps to install VisualAge C++ Professional for AIX Version 5.0:

1. “Installing and Configuring License Use Management”
2. “Installing VisualAge C++” on page 29
3. “Enrolling Licenses with LUM” on page 30

Installing and Configuring License Use Management

VisualAge C++ Professional for AIX Version 5.0 uses License Use Management (LUM) to control the license for the product. LUM is supported on AIX Version 4.1 and higher.

Note: VisualAge C++ Professional Version 5.0 requires the you are using AIX Version 4.2 or higher.

LUM is a component of the Base Operating System (BOS) in AIX Version 4.3 or higher. For systems running AIX Version 4.1 or AIX Version 4.2, LUM must be installed manually.

LUM only needs to be installed and configured once. If your system already has LUM installed and configured, you can skip to “Installing VisualAge C++” on page 29.

If you need to install LUM manually, you can obtain the installation images from the following sources:

- On the web at http://www.ibm.com/software/is/lum. This is the official web site and preferred source. It contains the latest version of the LUM Administration Runtime Kit (ARK) along with the documentation.
- Anonymous FTP from ftp.software.ibm.com. Log in with userid `anonymous` and enter your e-mail address as the password. Change the directory to `software/lum/aix`. The LUM ARK installation images are contained in the sub-directory `ark`. The LUM documentation is contained in the sub-directory `doc`.
- A version of LUM is available on the CD-ROM for VisualAge C++ for AIX, Version 5. This version consists of the smallest number of filesets required to install and run LUM, but may not be the most current supported version available. We recommend obtaining a complete installation of LUM from the official web site.

Once you have the filesets, install and configure LUM according to the instructions in the LUM documentation. The documentation for LUM, *Using License Use Management Runtime for AIX*, SH19-4346, is available in Adobe PDF form at http://www.ibm.com/software/is/lum/library.html. After you have installed VisualAge C++, you can find simplified instructions for installing and configuring LUM in the README.password file in the /usr/vacpp directory.

Installing VisualAge C++

You can install VisualAge C++ Professional Version 5.0 in one of two ways:

- Select the individual filesets you wish to install on your machine. This allows you to control which components of VisualAge C++ are installed.
- Use the `_all_latest` install option. This option installs all the components found on a CD, giving you the most complete installation of VisualAge C++ Version 5.0 possible.

If you are upgrading an existing installation of VisualAge C++, we recommend that you uninstall your existing version of VisualAge C++ before installing VisualAge C++ Professional Version 5.0.

Note: If you are installing VisualAge C++ Professional Version 5.0 on a machine running AIX Version 4.3, ensure that the Motif 1.2 compatibility fileset, X11.compat.adt.Motif12, is installed on that system. The Motif 1.2 compatibility library is part of the AIX Version 4.3 Base Operating System (BOS). It is a prerequisite for VisualAge C++ Professional Version 5.0.

You must have root user access to install VisualAge C++.

To install VisualAge C++ by selecting filesets:

1. Insert the first CD into your CD-ROM device.
2. At a command prompt, enter `smit install_latest`.
3. Press PF4 or click **List** to display a list of devices.
4. Select the CD-ROM device, then click **OK**.
5. Press PF4 or click **List** to select the filesets you wish to install.

   Some of the VisualAge C++ filesets are specific to the AIX version and language environments on your machine. If you select filesets that do not match your version of AIX or your language environments, you will receive a failed install message. We recommend that you read the fileset descriptions closely.

   Important: When fileset names differ only by the AIX version that supports them, only select the fileset supported by the AIX version equal to or lower than the version of AIX running on your machine. For example, if you are running AIX Version 4.2, and both *thisfileset.aix41* and *thisfileset.aix43* are available from the list of filesets, choose *thisfileset.aix41*.
6. Follow the on-screen instructions to complete this installation.
7. If you want to install the source code for the IBM Open Class libraries or the documentation for the Japanese or Simplified Chinese language environments, repeat the above steps using the second CD.

To install all the components of VisualAge C++:

1. Insert the first CD into your CD-ROM device.

2. At a command prompt, enter `smit install_latest`.
3. Press PF4 or click **List** to display a list of devices.
4. Select the CD-ROM device, then click **OK**.
5. Click **OK**
6. Follow the on-screen instructions to complete this installation.

   Some of the VisualAge C++ filesets are specific to the AIX version and language environments on your machine. You may receive an error message at the end of the install process.

   If you receive an error message at the end of the install process, check the names and descriptions of the filesets that did not install. Filesets not intended for your version of AIX are expected failures. In addition, filesets which require language environments not available on your system are also expected failures.
7. If you want to install the source code for the IBM Open Class libraries or the documentation for the Japanese or Simplified Chinese language environments, repeat the above steps using the second CD.

Once you have installed VisualAge C++ Professional Version 5.0, you will need to enroll your licence for the product before using it. For help on enrolling your licence see "Enrolling Licenses with LUM".

Enrolling Licenses with LUM

Before starting to use VisualAge C++, you must enroll your VisualAge C++ license with the License Use Management software you installed and configured earlier. Two LUM license certificates are provided with this product: a concurrent nodelock license certificate and a concurrent network license certificate. You should enroll the appropriate certificate for the type of license server you have configured.

Before enrolling a license certificate, ensure that you have LUM installed and configured on your machine. If you have not yet installed and configured LUM, refer to "Installing and Configuring License Use Management" on page 28.

The license certificates for VisualAge C++ for AIX, Version 5.0 can be found in the /usr/vacpp directory after you have installed VisualAge C++:

- The concurrent nodelock license certificate file name is **vac50_cn.lic**.
- The concurrent network license certificate file name is **vac50_c.lic**.

The LUM Basic License Tool runs either from a GUI or a command line interface. You must have root user access to enroll your VisualAge C++ license with LUM.

To enroll a license certificate using the LUM Basic License Tool GUI:

1. At a command prompt, enter `/var/ifor/i4blt` to invoke the LUM Basic License Tool.
2. Select **Products** > **Enroll Product** from the main menu. The Enroll Product dialog box appears.
3. Click **Import**. The Import dialog box opens.
4. In the **Filter** field, enter `/usr/vacpp/*.lic`.
5. Select either **vac50_cn.lic** or **vac50_c.lic** from the **Files** field, then click **OK**. Information about your VisualAge C++ license should now be displayed in the Enroll Product window.
6. Click **OK**. The Enroll Licenses window opens.
7. (Optional) Fill in the Administrator Information part of the Enroll Licenses window.
8. Fill in the number of valid purchased licenses of the product in the Product Information part of the Enroll License dialog box.
9. Click **OK**. The product should be successfully enrolled.
10. Exit the LUM Basic License Tool by selecting **Products** > **Exit**.
11. If you have enrolled concurrent network licenses, you must distribute the licenses before starting to use VisualAge C++ Version 5.0. For instructions on how to distribute licenses, see *Using License Use Management Runtime for AIX*, SH19-4346. Simplified instructions for distributing licenses can also be found in the `README.password` file which can be found in `/usr/vacpp`.

To enroll a license certificate using the LUM Basic License Tool command line interface:

1. Extract the **i4blt** command from the top of the product license file:
   - If you are enrolling a concurrent nodelock license, extract the command from **vac50_cn.lic.**
   - If you are enrolling a concurrent network license, extract the command from **vac50_c.lic**.
2. Replace `number_of_lics` in the command with the number of valid purchased licenses of the product.
3. (Optional) Replace `admin_name` in the command with the name of the administrator.
4. Invoke the updated command from the `/var/ifor` directory. The product should be successfully enrolled.
5. If you have enrolled concurrent network licenses, you must distribute the licenses before starting to use VisualAge C++ Version 5.0. For instructions on how to distribute licenses, see *Using License Use Management Runtime for AIX*, SH19-4346. Simplified instructions for distributing licenses can also be found in the `README.password` file which can be found in `/usr/vacpp`.

Accessing the Online Documentation

VisualAge C++ Version 5.0 provides you with extensive online documentation. To view the online help you need to have a frames-capable browser such as Netscape Communicator Version 4.04 (or later) installed on your system.

Much of the online documentation is also available in Adobe Portable Document Format (PDF) format. You can view and print this information using the Adobe Acrobat** reader. If you do not already have the Acrobat reader program installed, you can download it free from Adobe's website at http://www.adobe.com. The Adobe PDF files are in the /usr/vacpp/pdf files after installing VisualAge C++ Professional Version 5.0.

If you are using AIX CDE as your window manager, you can access the VisualAge C++ online documentation by selecting **Help Home Page** from the VisualAge C++ Professional 5.0 group in the Application Manager.

If you are new to VisualAge C++, we recommend that you take the "Tour the Integrated Development Environment" on page 35 and try some of the tutorials.

To find out about new features and improvements to VisualAge C++ since the last release, see "Chapter 2. What's New in VisualAge C++?" on page 5.

If you are planning to migrate existing projects to VisualAge C++ Professional Version 5.0, we strongly recommend that you read the "Chapter 7. Prepare to Migrate to Version 5.0" on page 97. It contains information to help you:

- Migrate to the VisualAge C++, Version 5.0 batch compiler from your presenet batch compiler.
- Migrate to the VisualAge C++, Version 5.0 incremental compiler from your present batch compiler.
- Migrate from VisualAge C++, Version 4.0 to VisualAge C++, Version 5.0.

Installing Fixes for VisualAge C++

You can install any available PTF fixes that apply to VisualAge C++ Professional Version 5.0 by installing over your existing program.

You must have root user access to install fixes for VisualAge C++.

1. Insert the latest IBM VisualAge C++ AIX CD-ROM into your CD-ROM device.
2. On the command line, type `smit install_latest` and press **Enter**.
3. Press **PF4** to display a list of devices.
4. Select the CD-ROM device, then press **Enter**.

5. To install the full product, press **Enter**. You can also press **PF4** to select the filesets you wish to install.
6. Follow the instructions to complete the installation.

Uninstall VisualAge C++

You must have root user access to uninstall this product.

1. At the command line, type `smit install_remove` and press **Enter**. The Remove Installed Software window opens.
2. On the **SOFTWARE Name** line, press **PF4**. A list of the available software filesets appears.
3. Select all VisualAge C++ filesets and VisualAge tools, then press **Enter**.

Some filesets may not uninstall if they are required by other, installed products.

Chapter 5. Tour VisualAge C++

About the Tour

This tour is designed to give you an introduction to the development environment.

In the first half of the tour, we suggest you read the pages in sequence. Follow the links at the bottom of each page to learn how to perform a series of basic tasks:

- Open and close projects
- Create and edit source files
- Run your program
- Respond to error messages

This portion of the tour should take about 20 minutes.

The second half of the tour, "More on the IDE", invites you to explore further some of the concepts introduced in the first half, in any order.

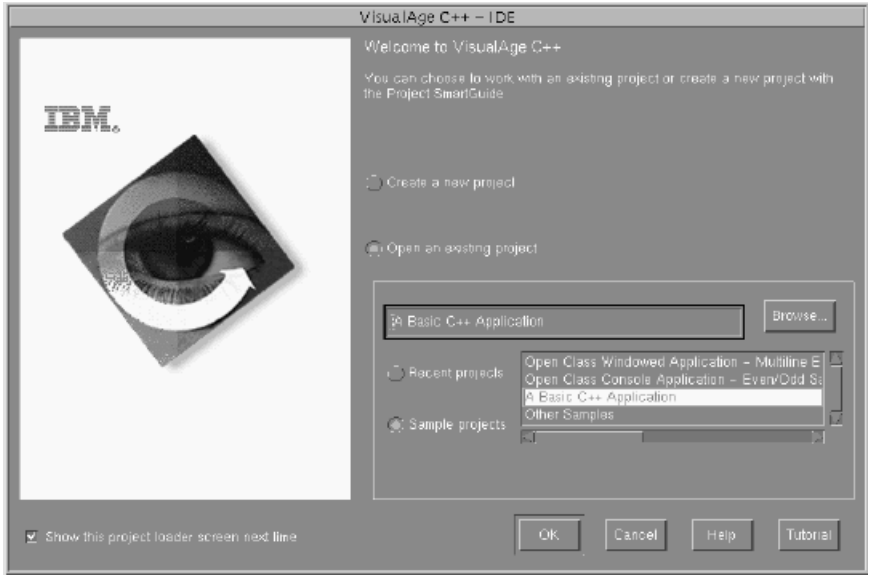
In the second half, you can also follow links to some more detailed online reference information.

Tour the Integrated Development Environment

This tour uses a sample program to introduce you to the VisualAge C++ Integrated Development Environment (called the IDE).

1. Start the IDE by double-clicking on the VisualAge IDE icon  in the Application Manager.
2. The first screen offers you a choice of creating a new project or working on an existing one. Select **Open an Existing Project**, and the **Sample projects** radio button. From the list of samples, select **A Basic C++**

Application:

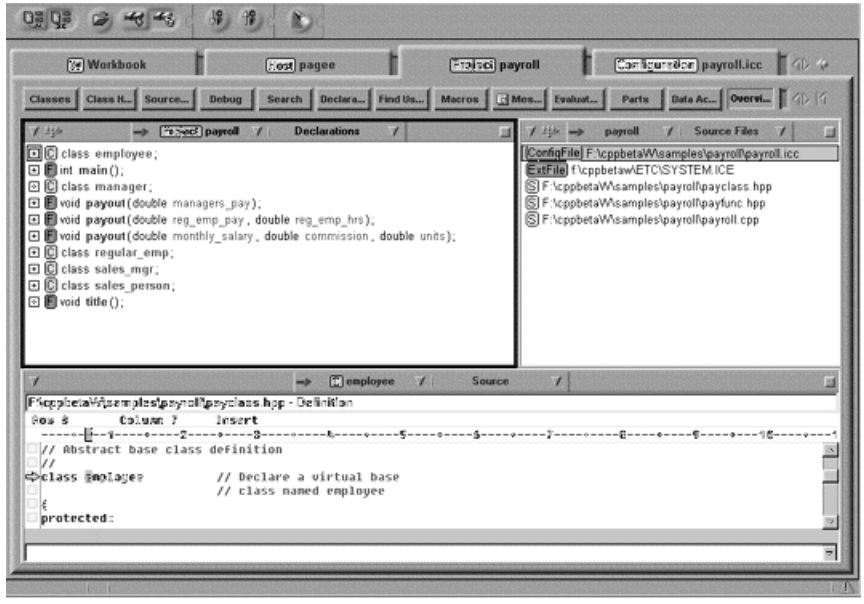


3. Click **OK**.
4. You are asked if you want to build the project. Click **Yes**.

If you already have the IDE open, follow these steps:

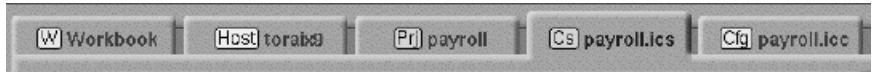
1. Click the Open Projectbutton  .
2. Browse through the **Drives** and **Folders** fields to find the `idesamp/payroll` directory in the main folder where VisualAge C++ is installed.
3. Select **payroll.icc** and click **Open**.
4. You are asked if you want to build the project. Click **Yes**.

The payroll project is displayed as a workbook with four sections. Each section is represented as a tab. Your screen should look like this:



Workbook (IDE)

The workbook consists of five major sections:



- **Workbook:** For examining and setting development environment options that control the IDE itself.
- **Host:** For browsing through files on your system.
- **Project**: For browsing the subprojects that make up your project and the project file (.icp) in which builds, subprojects, and variables are defined.
- **Codestore:** For working with your C++ code. This tab only appears once you have opened a subproject.
- **Configuration:** For setting options, or for adding and removing files from a subproject. This tab only appears once you have opened a subproject.

Click each tab to look at the different sections.

There is a row of buttons below each tab:

For each tab, you will see a different selection of buttons. Each button displays a different **page**.

Each page is further divided into mini-windows, or **panes**:

When you are finished looking through the different tabs and pages, click the **Project** tab again, and make sure the **Overview** page button is selected.

A Closer Look at Panes

To see a configuration overview, click the Codestore tab and the Overview page button.

On the Overview page there are three panes.

You use panes to look at *objects.* You can look at objects in different ways. Each different way of looking at an object is referred to as a *view*. For example, the object in the upper left pane is the Payroll project. The view in the upper left pane is a Declarations view.

You can change the focus to different panes with the mouse or by pressing F6. Change the focus to the upper left pane now.

Explore the Panes

There are three down-arrows () across the top of each pane. You now have the focus in the upper left pane. Click on the left-most arrow with the mouse pointer.

The menu that appears when you click this arrow controls the pane. You can move or resize the pane, or change the way in which it is connected (linked) to other panes.

The middle arrow displays the object menu. With this menu you can select the object you want to view. For example, the object being viewed in the upper left pane is the codestore for the Payroll project.

The right-most arrow allows you to select a view. There are different sets of views available for the different objects you select in the object menu. For example, in the upper left pane you are seeing a Source Files view of the Payroll codestore object.

Changing the views in the panes will not affect the contents of your project. Views are simply tools to help you browse through the project.

Pressing F1 takes you to the Help for more detailed information about the view for the pane in focus.

See How the Panes Work Together

The linking between the panes allows for powerful browsing and easy editing.

Change the focus back to the upper right pane, which shows a Declarations view of the file object selected in the upper left pane (or a Details view, if you have selected the configuration file). This view shows a list of all the declarations in the file. Select payroll.cpp in left pane. In the right pane, the main function will be highlighted by default. (If your file has no main function, the first declaration is selected.)

Now, select payclass.hpp in the left pane. In the right pane, select the manager class. Notice that the source view at the bottom changes as you do this.

When you select a class in the Declarations view with your mouse, two things happen:

1. If you had any other pane selected before doing this, the upper left pane now becomes the pane in focus (the border is darker).
2. The lower pane now displays the source code for the class object you selected.

Now, try the same action in the left pane. If you select any of the source file objects listed, the source view in the lower pane will update to show the file object selected. The immediate updating is made possible by links between the panes.

When you are more familiar with the IDE, you can customize the way the panes are linked to select the objects, views, and linking patterns that are most useful to you. The linking between panes is explained in greater detail later in this tour.

What's in a Project?

Each application you create in VisualAge C++ must be set up as a *project*.

Every project consists of the following files:

- one project file (*.icp)
  This controls your project at the highest level. In it, you define subprojects and builds.
- one or more configuration files (*.icc)
  .icc stands for Incremental C++ Configuration. Each subproject consists of a configuration and a codestore. A configuration file contains all the information about how the source files are processed. It is like an encapsulation of the subproject: it holds all options, lists all input libraries and other source files, and defines your targets. The compiler uses the configuration file to generate the codestore. If a source file is not listed in the configuration file or #included in another source file, then it is not part of the subproject or the project. You do not have to create the configuration file, but you can edit it.
- one or more source files (*.cpp, *.hpp, *.c, *.h, etc.)
  These are the files you provide or create. Source files can contain more than C++.
- a codestore (*.ics)
  The codestore is a database that contains the full information about a configuration. It is created the first time you build your subproject. You do not have to write this file; it is created for you, and updated automatically every time you build.

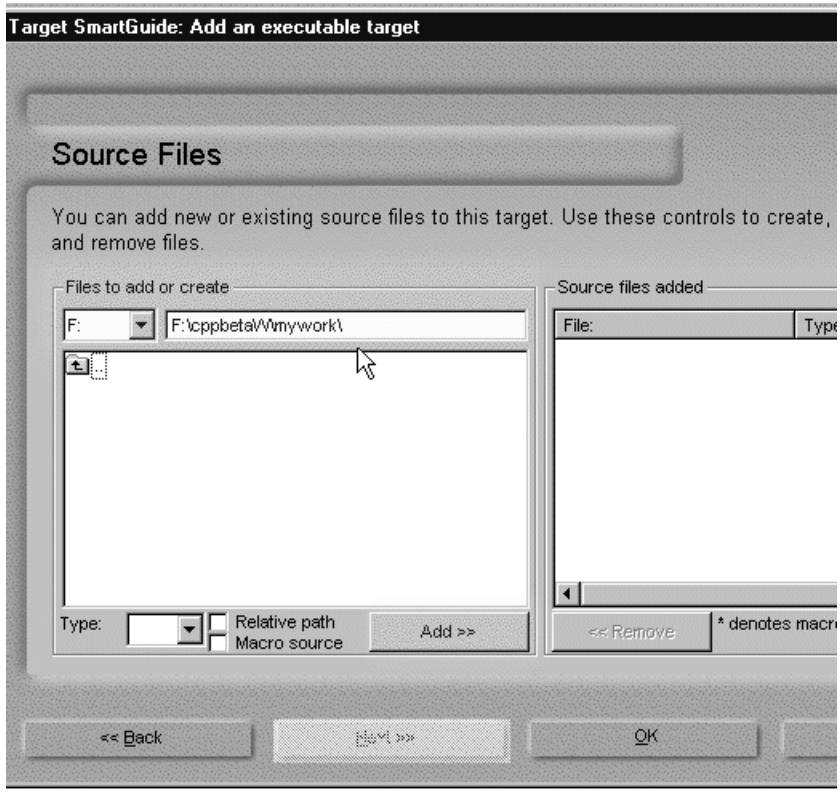
The tabs, pages, and panes you see in the workbook offer various ways to view and work with all of these components.

Create a Configuration

SmartGuides will step you through the process of creating a subproject:

1. First, close the sample project. Pull down the **Project Workbook** menu and select **Close Project**.

2. From the **Project Workbook** menu, select **Create Configuration**.
   The Configuration SmartGuide opens.
3. Click **Next** until the Configuration page is displayed.
   First, you are prompted to create the configuration file. This is a plan for the new subproject. You do not need to decide all of the details yet. You can always modify the configuration file later.
   Type a name, such as `helloworld`, to name your configuration file. The default suffix is .icc. It will be added by the SmartGuide if you do not type it.
   Choose a directory where this file and the codestore (.ics) will be located for this project. This is not necessarily where your source files (.cpp, .hpp, .c, .h, etc.) will be located. If you choose a directory that does not currently exist, the SmartGuide will create it for you. You can type a specific path, or click **Browse** to select a directory.
4. Click **Next**.
   The Target Type page is displayed.
5. From the pull-down list on this page, choose the type of target you want to create. Select **Executable**, and click **Add Target** (rather than **Next**).
   The Target SmartGuide opens.
6. Click **Next** until the Target Name page is displayed.
7. On the Target Name page, type `helloworld` to name your target file and select a directory where it will be stored. (The program type, in the lower part of the window, will remain "Default").
8. Click **Next** to get to the Source Files page.
9. On the Source Files page, you can specify new or existing source files to include in your project in the field **Files to add or create**:



Source files can come from any directory.

10. Type a source file name, such as `helloworld.cpp`. At the bottom of the window, pull down the **Type** menu and select **cpp** as your source type.
11. Click **Add** to add it to the project. It doesn't matter if the file already exists or not. The file name appears in the **Source files added** list, on the right side of the window.
12. For now, we are only creating one source file, so click **OK**.
    The dialog closes, and you are taken back to the Target Type SmartGuide. Now the target you defined is listed in the **Current targets** list.
13. Click **Finish**.
    A message window appears, and asks if you want to open your project in the IDE. Click **Yes**.
14. Finally, you are asked whether you would like to build the new configuration. Since you haven't added any code yet, click **No**.
    The IDE display has now added two new tabs: a **Codestore** tab and a **Configuration** tab. The panes are mostly blank because no code has been entered and compiled yet.

You have created a configuration, and you are ready to add content to your source file. Now, we will look at what you just created.

Looking at Your Subproject

As you went through the steps to create your configuration, you set no options, and you included no header files. You supplied only two file names (the target and the source), but that information was enough to create the basic outline for a subproject. We only need one subproject for this exercise, because we are creating only one executable. We do not need to create a project file (.icp) because this executable will have no dependencies on other targets.

Before you add any content to your source files, look at what you have so far.

Look at the Overview of the Configuration

Make sure that you are in the Codestore section, with the Overview page selected. Click in the white space in the upper right pane. The pane shows the Declarations view. There are no declarations yet.

In the upper left pane, your configuration file (helloworld.icc) is listed. This is the configuration file you named when you used the SmartGuides. Your source file is not listed, because you have not yet built the project. Select the configuration file by clicking it.

In the lower pane, you can see the contents of the configuration file. The Source view shows that you have defined a target and a source of type **cpp**. In this case, the type listed matches the suffix you have added to your source file name, but you could also have assigned a source file type of cpp to your file even if you named it `helloworld.c`, for example, or `helloworld` without any suffix. The file type can even be one you define yourself.

Click the **Configuration** tab to see more detailed information about helloworld.icc.

In the Configuration section, click the **Options** button. In the upper left pane of this page, you can see:

-  , a *target directive*, or a statement to specify the file that will be produced when you build, and
-  , a *source specifier*, or a statement to direct the IDE to use helloworld.cpp as input to a build. With your cursor, select the helloworld.cpp source specifier  .

The right pane shows a list of options. You did not set any particular options when you created the project, and you used the default program type in the Target SmartGuide, so only the option defaults apply. You can look at them here.

Click the plus sign (+) to expand **Optimization Options**, then expand **Common C/C++ Optimization Options**. A list of options appears, and you can see that they are in the default state. For example, optimization is turned off (the 'no' radio button is greyed) and the code you generate from this source file will be compiled for the most generic level of the PowerPC processor.

All the settings you see in this pane on the Options page are options applied *only to the source or target file you have highlighted in the left pane on this page.*

Now, go to the Subroject Options page by clicking the **Subproject Options** page button (also in the Configuration section).

The Subproject Options page looks very similar to the Options page, but the options here will be applied to *all* files in the project.

As you click through the other pages, you will see that each one presents a different emphasis on the configuration file.

Now that you have an overview of the framework of your project, you can add some content.

Adding Content to a Source File

VisualAge C++ has a built-in editor that you can use from any page in the workbook.

You can work on the same file simultaneously on one or more pages. You do not need to worry about your file relationships becoming outdated: changes made in source code on one page will immediately be reflected in every other page displaying the same section of code.

You have just finished creating a new project, but you have no content in your source files to edit yet.

To open and edit a source file:

1. From any tab, pull down the **Project Workbook** menu, and select **Open or Create File.**
2. In the Open or Create File dialog box, the file you specified when you created the project (helloworld.cpp) is selected in the **File name** field. (If it is not, browse through the **Directories** and **Files** areas to find it, and select it.)
3. Click **Open** to open helloworld.cpp. A source view of your empty file is opened.

4. Click anywhere inside the editor pane, and type or paste this short sample:

```
#include <iostream.h>
int main()
{
cout << "Hello World" << endl;
return 0;
}
```

You have just edited a source file. You do not have to explicitly save this file because all modified files in the subproject are saved when you start to build.

Build Your Project

In a typical development environment, some actions, such as changes to global header files, result in a complete rebuild. VisualAge C++ changes only what has been updated.

Usually, your first build will be the longest and every subsequent build will be shorter. Build time can be affected by the options you set, the number and complexity of your source files, and the type of linking you have chosen.

The first build is called the *initial* build. Every build after the initial build is an *incremental* build. The initial build for your sample will be very quick, because you have only one source file, and it is very small.

Click the Build button  .

If there were no errors in your Helloworld program, you should see the build result displayed in the message field at the bottom of your screen:

`Last Compile completed successfully on [`*`date`*`] in [`*`time`*`]`

If you see this message, congratulations! You have successfully created, edited, and compiled your project. Click the **Overview** button on the **Project** tab again. The Declarations view in the upper left pane now shows the main function,

and the Source view below shows the source code for this object, with `int main` highlighted.

If your compilation was not successful, the IDE can make the process of correcting your errors simple and fast.

Addressing Compilation Errors

If your Helloworld program was error-free, you saw how the IDE informed you of the status of your successful compilation: a message appeared in the status bar at the bottom of the screen.

Let's introduce an error to see how the IDE will handle it:

Place the cursor in the editor, or Source pane, and click into this line:

```
cout << "Hello World" << endl;
```

Change the semicolon (;) at the end of the line to a colon (:).

Now try rebuilding. Click the Build button  .

This time, two things happen:

1. The status line at the bottom of the screen displays a message:

   ```
   Last Compile terminated with errors on [date] in [time]
   ```

2. The page displayed has changed.

You now have the Messages page displayed.

On this page, the top pane lists the errors that were encountered during the rebuild, with a short description of each. The bottom pane shows your source code and has highlighted the point where the error occurred:

Place the cursor into this line and replace the semicolon.

Rebuild by clicking the build button again. It's that simple!

Run Your Program

Once you have successfully compiled your program, running it is simple.

You do not need to go to any particular page, tab, or view. You do not need to save the results of your build. From any page, pull down the **Project Workbook** menu. Select **Run**.

A command-line window appears briefly, displaying your output ("Hello World", or whatever words you used in the sample).

You have completed the tour of the IDE.

By now, you already know how to perform several important tasks in the IDE:

- Open or close a project
- Create a project
- View and set options for your project
- Edit and compile code
- Run your program

You can return to any part of the tour and try the tasks again if you're unsure about any of the steps.

If you're ready to learn more, you can continue to explore the IDE in a little more detail. The next portion of the tour offers more detail on some of the concepts you have already learned. You can explore them in any order.

More on the IDE

For some of the following topics, you'll need to have the payroll sample open. If you're not sure how to open it, read "Tour the Integrated Development Environment" on page 35.

Choose a topic:

• "Menu Descriptions" on page 56

More detailed information is available in the online documentation. Some suggested references:

- How Configuration Files are Processed
- Codestore
- Build Options
- Links Between Panes
- Editor Shortcut Keys

About Incremental Compilation

With VisualAge C++, the method of separate compilation managed through makefiles is no longer necessary.

When you make a change anywhere, only the affected functions are recompiled and linked, *not* the included header files, *not* the entire file where the functions are located. Build time is significantly reduced, and you are free from managing dependencies. You no longer have to maintain and sort complex header files and makefiles.

With the codestore, the IDE can provide information about your objects that other compilers cannot, and it provides the information to the various views of an object quickly. It also can give you early error feedback by checking all interfaces before compiling function bodies and variable definitions. If an error is found, you will be notified immediately.

How does Incremental Compilation Impact Build Times?

In general, the more files you have, the greater the improvement you notice over conventional compilers, within certain guidelines:

- Build time should be proportional to the changes made in the source code since the last build.
  - Changing a comment requires no recompiling
  - Changing the body of a non-inlined function only requires recompiling that function
  - Changing the body of an inline function requires recompiling all of the function's callers
  - Changing a declaration in a header file only requires recompilation of affected functions (instead of all functions in all source files that include the header file). This is a major advantage in moderate and large projects, where all source files tend to include most header files.

- Linking should take time proportional to the size of functions recompiled, not proportional to the program's size. For moderate-sized programs (a few tens-of-thousands of lines of code), VisualAge C++ takes a few seconds to do incremental builds that involve recompiling a few functions.

Incremental Compiling from the VisualAge C++ Command Line

VisualAge C++ is incremental all the time; no options to set, no trade-offs to make. If you invoke the incremental compiler from the command line, you still get full incrementality, but you must perform debugging through the IDE.

More on Editing Source Files

The IDE maintains a single codestore for each subproject. Even if you have several subprojects open, whenever you open a view of a source file, only one codestore will be updated with your changes.

This means that no matter where or when you choose to edit your source file, or how many views you use, there will always be only one version of that code.

Errors in Your Source

The live parsing editor is active in all source views, whenever you are working on a file with a .c, .cpp, .h, or .hpp extension. Simple lexical errors will be detected before you have to rebuild your source.

The Messages page displays any errors that occur. By default, there are two panes: a Messages view and a Source view. If an error occurs during a build, you will automatically be taken to the Messages page. Each time you select an error message in the Messages view, the Source view will be updated with the location where the error occurred. You can edit the file here, and then build again.

Types of Source You Can Use
Source files can consist of more than C++. VisualAge C++ supports the following types of files:

- cpp
- cxx
- hpp
- h
- c
- rc
- lib

- vbf
- vbe
- dax
- mak
- ipf
- loc
- msg
- sqc
- sqx

Searching a Project

There are three ways to search a project:

- **Live Find** is a dynamic search available in most views. It can be accessed with shortcut keys, and works like most other dynamic searches by finding text strings to match your criteria in the body of your code. It is also useful for locating strings that are not necessarily in your source code, for example, to search for an option in the Options page, you can type a part of the option name or category in the **Live Find** entry field and avoid scrolling many long lists of options.
- The **Search** page searches within any object, from a single class to the entire project.
- You can search semantically using the **Find Uses** page. Searching semantically means searching for an *object*, such as a class, rather than a text string.

Try performing a Live Find using the payroll project.

1. Select the **Codestore** tab.
2. Select the **Classes** page button.
3. Click the Live Find toolbar button  or:
   Click mouse button 2 on any part of the background (white space) in the view.
   Select **Find (Live)** from the pop-up menu.
   A text entry field appears at the bottom of the view.
4. Select the  next to the employee class to start the search at the top of the view's contents.
5. In the text entry box, slowly type 'pa':
   - As you type the 'p', the employee class is highlighted. The Source view also updates with the corresponding code.

- As you type 'a', the virtual function pay() is highlighted, and the function definition is displayed in the Source view.

6. In the text entry box, remove the 'a'. The results are updated again, but there is now a pull-down key  to the right of the text entry box that will take you back to previous searches without retyping your search string.
7. You can move through the list of strings that contain the letter p by pressing **Enter** or **Ctrl-N** to move forward to the next match and **Ctrl-U** to move backwards to the previous match.
8. Press **Esc**, or click the small flashlight icon next to the entry field and select **Close** from the popup menu.

Configuration Files

A VisualAge C++ project must have at least one configuration file. Each subproject contains one configuration file, and often represents one target.

When you build a subproject, VisualAge C++ uses the configuration file to figure out how to do the build. A configuration file is similar to a makefile, but with some important differences:

- Configuration files are easier to create and maintain than makefiles. VisualAge C++ can create the configuration file for you.
- Configuration files do not require any C++ file dependency information or processing commands.

Using the SmartGuides and views in the Configuration section, you can create and edit configuration files in the IDE without necessarily learning any syntax.

While it is possible to build several targets in a single configuration file, this is not always the best approach. You must have multiple configurations to build the following targets:

- More than one executable from the same set of source files
- A DLL and a static library from the same set of source files

For example, one subproject might have a debug configuration with debug options, and another may have a production configuration with optimization options. Each configuration will produce only one codestore.

The default extension for a configuration file is .icc, but any extension can be used. The configuration file can reside in any directory.

Setting Build Options

Build options are a part of each configuration file (.icc). You can set build options through the views in the **Configuration** section of the workbook. You do not need to know configuration file syntax to edit these views.

Options can be set globally for your entire subproject, or applied only to a list of files.

To set options for an entire project:

1. Go to the Configuration section.
2. Click the **Project Options** page button.
3. Set the options for the project. Click the **Apply** button.

Project options applied to all files that are part of the project when the options are set.

To set build options on a file or group of files:

1. Go to the Configuration tab.
2. Click the **Options** page button.
   There are two panes on this page: a Sources and Targets view and a Change Options view.
3. Select a source file or target from the Sources and Targets view.
4. Set the options for that source file or target in the Change Options view.
5. Click the **Apply** button.

Some options are set automatically when you define your application type on the Target Name page of the Target SmartGuide.

Symbols Used in the IDE

You may have seen that some views in the IDE contain colored circles with letters. These are some of the symbols used to represent the various objects displayed in the IDE.

If you look at the tabs in the workbook, you will see four of the symbols:

-  Workbook
-  Host
-  Project
-  Configuration
-  Codestore

There are many other symbols also used. Some common examples are:

- Included file (one that is #included in a source file)
- Class
- Function
- Asn Variable assignment

To see the various types of symbols and the relationships among the objects they represent:

1. Select the **Workbook** tab.
2. Select the **Schema Overview** page button.
   Three panes are displayed:
   - A Descriptors view, which lists all the types of objects that can be represented in the IDE. Expand the descriptor to see how the selected descriptor relates to others.
   - A View Types view, which lists the types of views available. Expand the views to see which descriptors are allowed in each type of view.
   - A Page Types view, which lists the types of pages available. Expand the page types to see the types of descriptors available on a page.

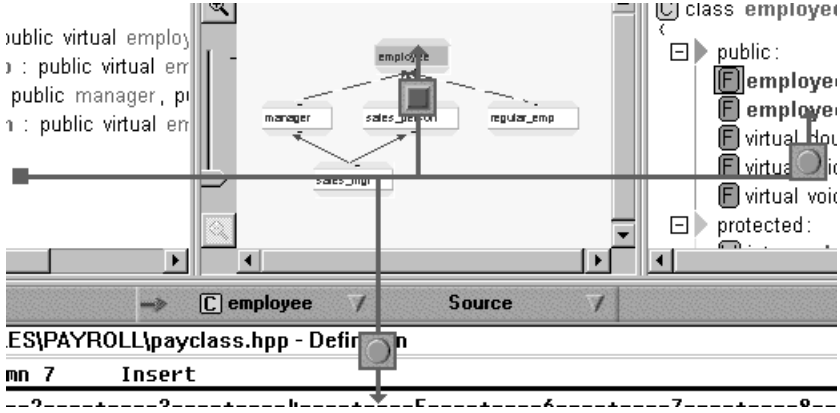
Linking Between Panes

One of the most powerful features of the VisualAge IDE is the linking between the panes.

To see how the panes on a page are linked:

Pull down the **Page** menu, and select **Show Link Diagram**. A Help Tips window may appear: click **OK** to continue to the linking diagram.

An example of a linking diagram:



You can see there are two types of symbols on the link diagram: 

(automatic link) and  (manual link).

- If a pane has an automatic link, the input varies with the pane that has focus. In other words, in the example above, the bottom pane will take input from whichever pane in the top row is active.
- With a manual link, the input comes from the same pane regardless of which pane has focus. For example, the centre pane in the top row in the example above will not change when a different pane becomes active.

You can customize linking by clicking on the link symbol to change the link type. The online help includes more information on the links and how you can work with them.

You do not need to change the linking diagram in order to work with the IDE, but understanding it will help you to customize your work environment.

While the linking diagram is displayed, all other functions within the IDE are suspended.

To go back to the IDE, select **Hide Link Diagram** from the **Page** menu, or press **Esc**.

Toolbar Buttons

The IDE toolbar contains icons for frequently used actions. The default selection of icons is:

-  The first two buttons are for saving and building the currently loaded project.

-  The third button is for loading a new project, and closing the existing project.

-  The next two buttons are for adding and removing bookmarks on the pages in the IDE.

-  The next two buttons are pane-specific. The pane with the current focus is the recipient of these two actions. Use the first icon to choose the next object in the pane history, and the second icon to choose the previous object in the pane history.

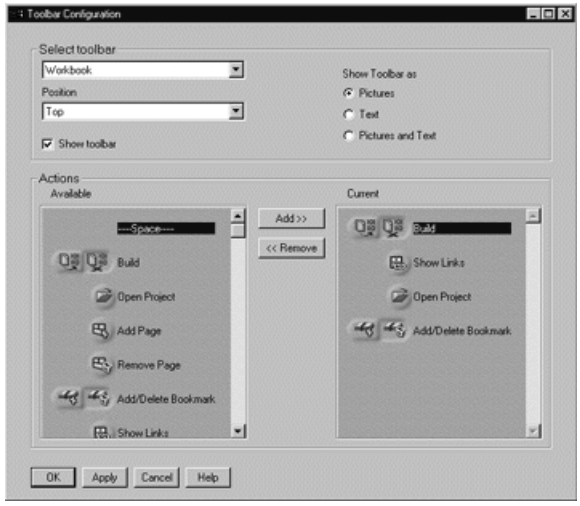
-  The last button starts a Live Find. It will open the Live Find window for the pane currently in focus.

Other buttons that appear are view-specific. For example, when a source view is active, buttons for recording macro keystrokes or printing will also appear on the toolbar.

To find out what any button does, place your cursor over it without clicking. A flyover label will appear.

To customize your toolbar:

1. Select the **Workbook** tab.
2. Select the **Settings** page.
3. Select the **Toolbar Configuration** push button in the **Settings** view. The Toolbar Configuration window opens (shown below).



To add and remove buttons from the toolbar, select them from the scrolling lists and click on the **Add** or **Remove** buttons.

Menu Descriptions

Project Workbook
Use this menu to perform actions on the project as a whole, such as starting a build, removing sections, and opening new projects and files.

Page Use this menu to perform actions on a page, such as adding and removing pages, viewing a linking diagram, saving and removing page descriptions, and quickly accessing other pages in the workbook.

Pane Use this menu to perform actions on a selected pane, such as changing the object displayed, changing the view of the object in the pane, adding and removing panes, maximizing a pane, changing settings, and setting filters.

Selected
Use this menu to perform actions on the selected object in a view.

Debug
Use this menu to initiate debug actions, such as debugging, running, stopping, stepping and terminating.

Bookmarks
Use this menu to set a bookmark on any page in the IDE when you want to be able to quickly flip between commonly used pages.

Help Use this menu to access the online documentation provided for VisualAge C++, start a tutorial, or browse the samples.

Dynamic Menus
These menus are associated with a particular tab and a particular view. As you change the pane focus, the fourth menu changes to reflect your selection. Use these menus to perform actions relating to the view and object selected.

Chapter 6. Try VisualAge C++

Introduction to VisualAge C++ Configurations and Subprojects

These short tutorials will introduce you to the concepts of projects and subprojects in VisualAge C++ Version 5.0. You will use the code for a sample 3-D modelling application to construct a project, make some modifications to the build setup, and build the application.

The tutorials will each take approximately 20 minutes to complete. You will learn:

- The basic setup of projects and subprojects
- The language of project files and configuration files
- How to set variables and use directives to control the build environment
- How to improve subproject build times
- How to organize projects for easy maintenance

As you go through the pages, you can bookmark a page in your browser at any time, and return to it later. To do this, right-click on the background of the page and select **Add Bookmark for frame** from the pop-up menu.

The Sample Application

The sample project, model3d, allows you to create and manipulate three-dimensional objects. The project consists of four subprojects:

1. A library for processing basic operations such as transformations from three dimensions to two dimensions, shape creation, and object hierarchies
2. A library for displaying the models in the user interface using the IBM OpenClass 2d graphics classes
3. A user interface
4. A specific implementation of a polygonal, three-dimensional object

If you are already familiar with VisualAge C++ Version 4.0 configuration files, you can go directly to Tutorial 2.

Tutorial 1: Configuration Files (.icc)

Configuration files are the key to using the incremental compiler, in the same way that makefiles are essential to using any other compiler.

The .icc file extension stands for Incremental C++ Configuration. A configuration file contains all of the information needed for processing source

files. It holds all options applied to the source files and targets, lists all input libraries, and defines all targets (outputs). If a source file is not listed in a configuration file, or included by a file that is listed in the configuration file, it is not a part of the project.

Here is an example of a configuration file:

```
option link(export,all)
{
    target type(exe) "Helloworld"
    {
        source "Helloworld.cpp"
    }
}
```

The file consists of *directives* and *specifiers*. A directive declares an object of a certain type, and the specifier names it.

Option directive
The word `option` is an option directive. It introduces one or more options and can also name a group of options. The option applies to everything within the opening brace and its corresponding closing brace.

Target directive
The word `target` is a target directive. It indicates the object file to be created when the sources are built. The opening brace following the target directive signals that everything between this brace and the corresponding closing brace is used to build the target.

Source directive
The word `source` is the source directive. It introduces a source file, a list of source files, or the name of a source group. The name Helloworld.cpp is the source specifier.

This basic configuration file means: Use Helloworld.cpp to build Helloworld.exe, and apply the export linking option to all symbols when building.

Together, this information is one configuration. When you build this configuration, there are two results:

1. The output, the 'Helloworld' executable, is created.
2. A database of information about the output, the input, and the options is created. This database is called a *codestore*. The file name of the codestore is the same as the file name of the configuration file, with the extension .ics.

In the next part you will create a directory to contain copies of the sample code used in the tutorials, and create a configuration file with the help of the VisualAge C++ SmartGuides.

Copy Source Files for the Sample Project

The source files for the 3D modeler sample are located in the following directories:

- /usr/vacpp/tutorial/
- /usr/vacpp/tutorial/util/
- /usr/vacpp/tutorial/poly/
- /usr/vacpp/tutorial/base/

To ensure that you will be able to make code changes and create files as necessary in the tutorial, you should copy the files in these directories to a directory of your own, maintaining the same directory structure. That is, within your writeable directory, you should also create sudirectories called /poly, /base, and /util.

For Tutorial 1, you will use only the files from the /util subdirectory. To copy the files for the /util subdirectory:

1. Go to the Host section (Host)of the workbook.
2. In the File Tree view, locate a directory to which you have write access.
3. Right-click on the directory, and select **Create directory** from the pop-up menu.
4. In the Create Directory dialog, enter `util` as the name for the new directory.
5. Click **OK**.
6. In the File Tree view, locate /usr/vacpp/tutorial.
7. In the Files view (to the right of the File Tree view), click on the first file in the list, hold down the Shift key, and click on the last file. This selects all of the files.
8. Right-click on the selected files, and select **Copy** from the pop-up menu.
9. In the File Tree view, select your new directory, right-click on it, and paste the sample files you copied.

Repeat these steps for each of the directories listed at the top of this page.

Now you are ready to do the first tutorial, in which you will create a configuration.

Create a Configuration File

A configuration file loosely replaces a makefile, in the sense that it organizes dependencies between build outputs (targets) and sources. It has advantages over a makefile, however:

- You do not have to explicitly state or maintain dependencies: you merely have to list your sources for each target. VisualAge C++ will automatically track the relationships between sources and targets.
- A configuration file is much easier to read. In addition, you do not need to learn the syntax in order to create configuration files: the SmartGuides provided with VisualAge C++ create the files for you.

To create a configuration file, you will use the source files in the /util directory. These files provide the low-level function (the utilities) for the 3D modeler.

Identifying your target

1. From the Project Workbook menu, select **Create configuration**.
2. Click **Next** on the first page of the SmartGuide. You can turn this page off by clicking the **Show this page next time** check box, to clear it.
3. On the Configuration page, enter:
   a. util.icc, as the name for the configuration file. Whether or not you type the extension, the file will automatically be saved with the extension .icc.
   b. a location for the configuration file. In the **Location** field, browse to or type the path for the directory to which you copied the sample source files, for example, /home/joe/sample.

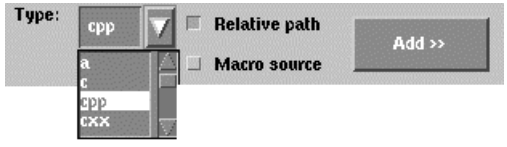
   Ignore the check box labelled **Disable created target**. This is used only when you are creating a configuration for migrating a project that has not been built in VisualAge C++ Version 4.0 or 5.0 before.
4. Click **Next**.
5. On the Target Type page, you can select whether the output will be a library or an executable. For this sample, select **Shared library (SHR)** from the pull-down list.
6. Click **Add target**.

Identifying options

1. The introductory page for the Target SmartGuide appears. Again, you can turn this page off by clicking the **Show this page next time** check box, to clear it. Click **Next**.
2. On the Target Name page, `util.a` appears in the first field.
3. Make sure that the target directory listed is one to which you have write access.
4. For this sample, you will make use of the IBM Open Class library. Select **IOC** in the **Program type** list.
5. Click **Next**.

Adding sources to the target

1. On the Source Files page, use the directory tree to locate the following source files:
   - vector.hpp
   - vector.cpp
   - matrix.hpp
   - matrix.cpp
   - except.hpp
   - notifier.hpp
   - notifier.cpp
   - observer.hpp
2. Select all the files in the list above. Click on the first file and hold down the **Ctrl** key to select the rest.
3. Select **cpp** from the pull-down **Type** menu, below the browse window:



4. Click **Add>>**. All the file names should appear in the window on the right, with `cpp` appearing under **Type** for each file.
5. Click **Next**.
6. On the target type page, click **Finish** to complete the configuration and exit the SmartGuide.
7. When you are prompted to open the project in the IDE, click **Yes**.
8. When you are prompted to build the subproject, click **Yes**.
9. When you are prompted to optimize your configuration, click **No**.

In the next step, we will examine the configuration file that the SmartGuide created.

Configuration Directives

In the previous step, you defined a configuration by telling the SmartGuide what kind of target you wanted to produce, and which source files to use. In this step, we will examine what the SmartGuide created with your input.

In the IDE, three new workbook sections have appeared:

- Prj The Project section
- Cfg The Configuration section
- Cs The Codestore section

In creating a configuration, you did not create a *project*. In VisualAge C++ Version 4.0, a project consisted of only one configuration. Therefore, only the configuration file was needed to control a project. However, in Version 5.0, your project is controlled with a project file (.icp file), which must list the configurations that make up the project. Because you have not yet created a project, the Project section is empty.

The Codestore section displays the results of the build in various different views.

The Configuration section displays all the information about the configuration file. To see the contents of the configuration file:

1. Select the Configuration section by clicking on the Cfg workbook tab.
2. Select the **Source** page button 

to see the contents of the configuration file.

You should see the following:

```
option SubprojectOptions = gen(rtti, yes)
{
   options defaults("xlC_r"), link(linkwithsharedlib, yes), define("USE_IPF", "1")
   {
      target type(shr) "util.so"
      {
       source type(cpp) "matrix.cpp"
       source type(cpp) "notifier.cpp"
       source type(cpp) "vector.cpp"
       source type(cpp) "vector.hpp"
       source type(cpp) "matrix.hpp"
       source type(cpp) "except.hpp"
       source type(cpp) "notifier.hpp"
       source type(cpp) "observer.hpp"
       option macros(global, yes)
       {
          source "pthread.h"
       }
     }
   }
 }
```

The *target directive* declares the target to be a shared library, and `"util.so"` specifies its name.

The *source directives* all declare sources that are C++ files, and name the source files individually. One directive can also name several sources. For example, the source directives above have the same meaning as the following:

```
source type(cpp) "matrix.cpp", "notifier.cpp", "vector.cpp"...etc
```

In the next part of this tutorial, you will also add an *option directive* to set build options for this shared library.

Setting Options

In the previous steps, you created a basic configuration, and learned that a configuration can consist of sources, targets, and options.

In this part you will:

- Use the Change Options view to set an option
- Use the Live Find feature to quickly search through the available options for one specific option
- Learn how to read the option display to determine if a setting is turned on or off by default, or explicitly set in the configuration file

There are several types of options for compiling and linking. They are all set in configuration files. If you know the name and syntax of an option, you can type it directly in the configuration file to set it. However, there are hundreds of options available. The technique below is the fastest way to find and set options when you do not know their spelling.

The syntax of an option directive in a configuration file is:

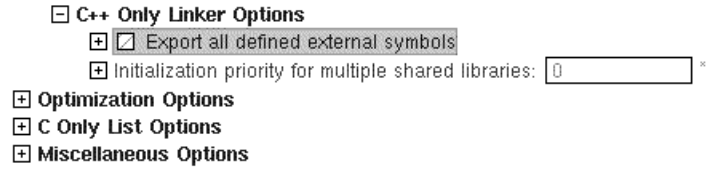
```
option option_type(option_name, setting)
```

In this example, the option is a link type of option, named 'exportall', set to 'yes'.

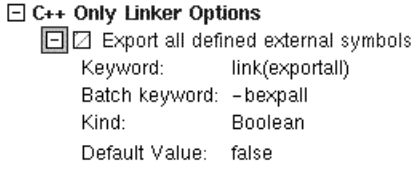
The link(exportall) option is used to export all defined, external symbols in C++ source from the target being built. We will use this option because it is the simplest way of making symbols externally visible so that other modules can access them.

1. Select the Configuration section  .
2. Select the Options page  .
3. In the left pane, select the target object ().
4. Click into the Change Options view (the right pane).
5. Click into the Live Find field at the bottom of the pane.

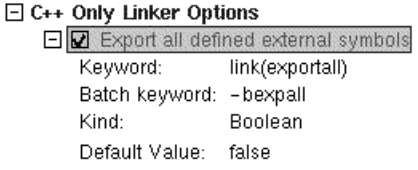
6. Type `export all`. The view scrolls to the first match. Press Enter until you see **Export all defined external symbols** :



7. Currently, the check box next to **Export all defined external symbols** is hashed with grey lines. This means it is in the default state. Click on the plus symbol next to the checkbox to see more details about the option:



8. From this, you can see that:
   - The full spelling of this option is link(exportall).
   - The equivalent option for the batch compiler is -bexpall.
   - The default state is false (i.e., by default, the option is off).
9. Click the hashed check box until a check mark appears:



   After you do this, and the option is explicitly set (turned on), the option description is displayed in green, and a check mark appears in the box next to it. The option also appears in your configuration file.
10. At the bottom of the pane, click **Apply**. This copies the option to your configuration file.

The text `option link(exportall, "yes")` is added to your configuration file. This has the same meaning as link(exportall). By default, if no setting appears after a Boolean option, and the option appears in the configuration file, the setting is assumed to be yes.

Check the Source page in the Configuration section (Cfg) to see how your configuration file was updated.

Codestores

The codestore (.ics file) is a database that contains the full information about a configuration. It holds information about:

- Your sources (the filenames, locations, and types)
- Your targets (the filenames, locations, and types)
- All settings used to build the target (options you have set, variables you have defined, and their values, if calculated)

The codestore is created the first time you build a configuration, and is updated automatically every time you rebuild. This is what makes incremental compilation faster: since all of the information is saved when each source file is compiled, only the parts that have changed need to be recompiled.

There is another benefit to having a codestore: it means that you can browse all of this information as you maintain and develop your code. You can instantly see the relationships that exist among source files, and even among functions and declarations.

While you cannot view the contents of a codestore in the same way you can view the source text of a configuration file, you can see what it contains in the Codestore section () of the IDE. Every view in this section is a view into the codestore.

For example, go the Codestore section of the configuration you built in the previous pages, and try these features:

- Select the Source Files page and look at the Source Files view. You can see all of the sources you listed in your configuration file.
- Select the Classes page and look at the Classes view. You can see all the classes defined in all of the source files. Select any class object () to see its source.
- Select the Declarations page and look at the Declarations view. If the declaration of the namespace util is not expanded, click the plus sign next to it. You can see the declarations of all the classes and functions in all of the sources.

Summing Up Tutorial 1

In this tutorial you learned that a configuration file, which loosely corresponds to a makefile, is a text file that defines targets (output), source files (input), and options.

What have you learned so far?

This tutorial introduced the concepts of:

- Configurations, and how they replace makefiles
- Codestores, and how incremental compilation is made possible
- Configuration options, and how to set them
- Views in the IDE, and how information from the codestore is displayed

In VisualAge C++ Version 4.0, a project consisted only of a single configuration, that is, one configuration file and the associated codestore.

In Version 5.0, many configurations can be grouped in each project. The project is divided into subprojects in order to make this possible.

In the next tutorial you will get a closer look at how subprojects fit together to form a project.

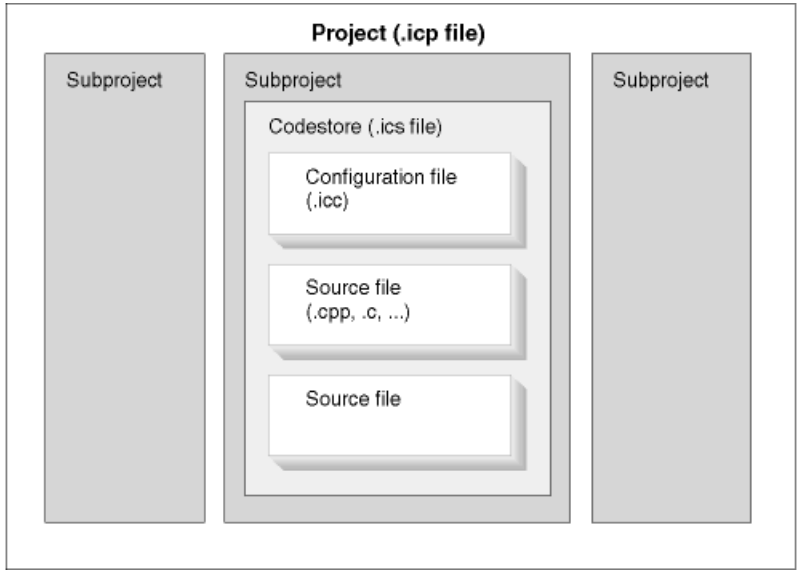
If you are interested in learning more about the concepts introduced in this tutorial, try reading some of the following pages in the online help:

- Codestore
- Configuration Files
- Configuraion File Directives

Tutorial 2: Introducing Projects

In this tutorial, you will use the Project SmartGuide to help you set up a small project, then examine the project to understand its structure.

Each application you create in VisualAge C++ must be set up as a *project*:



Every project is managed from a single project file (.icp file), where you control your project at the most general level. At the project level, you can organize the *subprojects*, arrange them in different combinations for different types of *builds*, and set build variables.

At the subproject level, you define source files, outputs, and the options for compiling and linking specific targets. A project can contain any number of subprojects.

In the next step, you will use the Project SmartGuide to help you set up a project.

Create a New Project in the IDE

In this tutorial, you will launch the Project SmartGuide, which will take you through the steps necessary to combine four existing pieces of the sample application into one project.

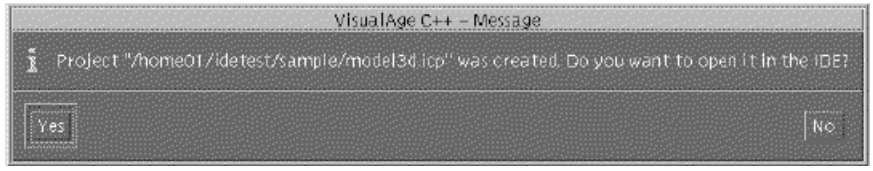
Before you start these steps, ensure you have write access to a directory, so that you can copy the sample files you will be using. If you have not done Tutorial 1, please follow the steps in "Copy Source Files for the Sample Project" on page 61 before you continue with this part.

When you have copied the source files, create your project:

1. Select **Create Project** from the **Project Workbook** menu. This will launch a SmartGuide to step you through the process of creating a new project.

2. When the introductory page of the Project SmartGuide appears, click **Next>>** to begin creating the project. If you do not want to see this page each time you start the SmartGuide, clear the radio button labelled **Show this screen next time**, in the lower left corner of the dialog box.
3. The Project page asks you to name the project, and choose the directory where the .icp file (project file) will be stored. Enter a name for the project, for example, Model3d. Choose the directory you created as a location for the file.
4. Click the **Browse** button to locate a directory for which you have write access.
5. When you have selected a directory, select the second radio button below the **Project directory** field, labelled **Create a new project using new and existing configuration files**.
6. Click **Next>>**.
7. On the Configurations page, browse through the file system to the directory and subdirectories you created (for example, /home/joe/sample):
   - in the root of the directory, select Model3d.icc, and click **Add>>.**
   - in the /util subdirectory, select util.icc, and click **Add>>** .
   - in the /base subdirectory, select base.icc and click **Add>>** .
   - in the /poly subdirectory, select poly.icc and click **Add>>.**
8. When these four files appear in the **Added Configurations** list box, click **Finish**.

9. To confirm, click **Yes**.



10. In IDE, the **Overview** page of the Project section is displayed by default.

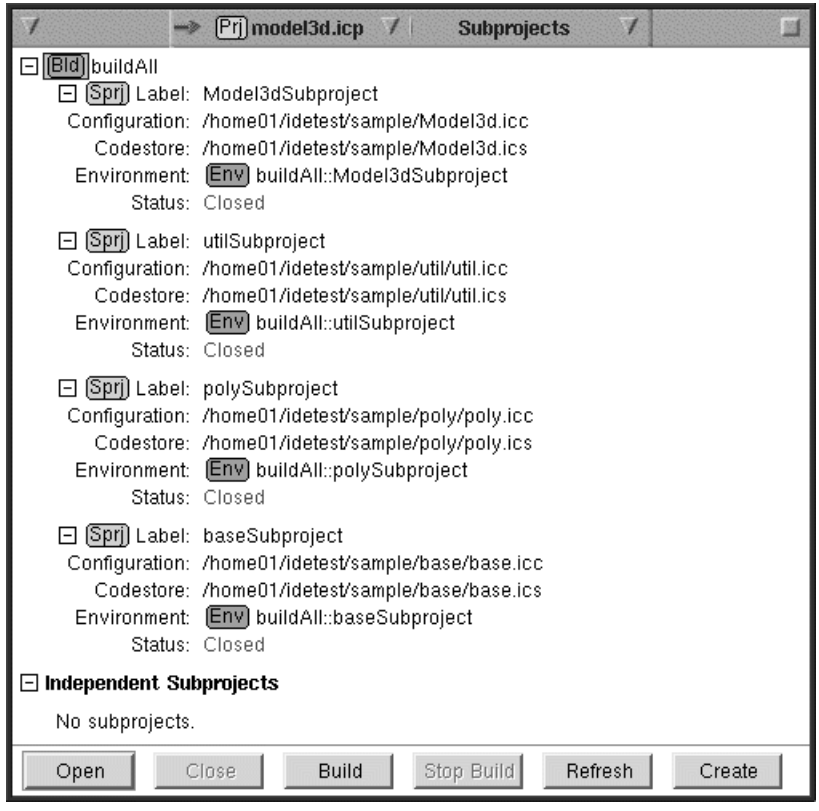
You have created a simple project consisting of four subprojects.

In the next section, you will examine the project file that was generated by the Project SmartGuide.

Examine a Basic Project

In the previous step, you created a project. Now the Project section (Prj) of the IDE workbook contains information about the project.

The Overview page of the Project section describes how the project is divided into subprojects. The description looks like this:



This summary shows that the project consists of four subprojects (Sprj), which were given the names of the configuration files that were used. The subprojects each contain a configuration file (Cfg or .icc file) and a codestore (Cs or .ics file), however, you cannot see the details of these files. Although the subprojects were created, they are not currently open in memory: the Status line for each subproject shows it is `Closed`.

When you clicked **Yes** to open the project in the IDE, only the project file (.icp file) was opened in memory. To load each of the subprojects listed in this project file, you must select them in this view and click **Open**. A subproject is only open in memory when the **Status** line indicates a status other than `Closed`, such as `Building`, `Built`, `Unbuilt`, or `Built with errors`.

In the next step you will examine how the project file language works.

The Project File

To see the contents of the project file, from the Project page click on the **Source** page button 

.

The contents of the file are as follows:

```
subproject Model3dSubproject icc "model.icc", ics "model.ics"
{
}
subproject utilSubproject icc "util/util.icc", ics "util/util.ics"
{
}
subproject baseSubproject icc "base/base.icc", ics "base/base.ics"
{
}
subproject polySubproject icc "poly/poly.icc", ics "poly/poly.ics"
{
}
build buildAll
{
use Model3dSubproject
use utilSubproject
use baseSubproject
use polySubproject
}
```

Project file language is similar to configuration file language. The project is described using *directives* and *specifiers*. A directive declares an object to be of a certain type, and the specifier names it.

Subproject directive

The word `subproject` is the subproject directive. In effect, it declares that a subproject object exists. The directive's `icc` and `ics` files must follow it: they declare that the subproject consists of a configuration file and a corresponding codestore. The specifier, or name, of the first subproject is `modelSubproject`.

The opening and closing braces ({ and }) are intended to enclose environment variables, if any are set, or to list dependencies on other subprojects. They are not necessary if you do not intend to set any project variables, or if the subproject has no dependencies.

Build directive

The word `build` is the build directive. It begins a description of a build, which can combine any number of subprojects to be compiled together. The name of the default build created by the SmartGuide is buildAll.

Between the { and } for the build directive, the components of a build are listed, within use directives. The default build includes all of the subprojects.

Use directive

The word `use` is a directive to include the subproject in a build. If the subproject is not identified with a use directive, inside the { and the } of a build directive, it will not be processed when the build is launched.

In the next step, you will examine the subproject.

A Closer Look at Subprojects

A project consists of one or more subprojects. The relationships among subprojects are defined in the project file.

Each subproject is essentially a combination of:

- A configuration, defined by a configuration file (.icc)
- A codestore (.ics), which is like a database containing the compiled code from all sources in the configuration, as well as all options and other settings
- An environment (), which is the sum of the environment variables set at the system level and any variables you have defined to supplement or replace these

A subproject is represented in the IDE as  .

As you saw in the previous part of the tutorial, each subproject has a definition, such as:

```
subproject Model3dSubproject icc "Model3d.icc" ics "model3d.ics"
```

You currently have four subprojects in your project, but there is not yet a way to determine the order in which they should be built. While VisualAge C++ automatically tracks dependencies between sources and targets within a configuration, you still need to specify relationships between subprojects.To specify order, you need to set up *dependencies*. Dependencies are defined using **depends on** directives. For example, the following subproject definition states that Model3dSubproject is dependent on the other three subprojects:

```
subproject Model3dSubproject icc "Model3d.icc" ics "model3d.ics"
{
            depends on polySubproject
            depends on baseSubproject
            depends on utilSubproject
}
```

This definition ensures that the libraries needed to build the executable model3d are available before the compiler attempts to build Model3dSubproject.

In the next part, you will set up dependencies in your project file.

Set up Dependencies Among Subprojects

As you saw in the previous part, dependencies between subprojects need to be defined in the project file.

The main part of the modeler sample, Model3dSubproject, uses the targets of the other three subprojects as sources for its own target. It therefore depends on the other three subprojects. Similarly, each of the other subprojects has a dependency on one or more subprojects, as follows:

- the target libbase.so requires libutil.so as a source
- the target libpoly.so requires libbase.so as a source
- the target model3d requires libbase.so, libutil.so and libpoly.so as sources

The dependency relationships among the subprojects can be described in the project file in two ways. Either:

```
subproject Model3dSubproject icc "Model3d.icc", ics "Model3d.ics"
{
             depends on polySubproject
}
subproject polySubproject icc "poly/poly.icc", ics "poly/poly.ics"
{
             depends on baseSubproject
}
subproject baseSubproject icc "base/base.icc", ics "base/base.ics"
{
             depends on utilSubproject
}
subproject utilSubproject icc "util/util.icc", ics "util/util.ics"
{
}
```

or:

```
subproject Model3dSubproject icc "Model3d.icc", ics "Model3d.ics"
{
             depends on polySubproject
             depends on baseSubproject
             depends on utilSubproject
}
subproject polySubproject icc "poly/poly.icc", ics "poly/poly.ics"
{
             depends on baseSubproject
             depends on utilSubproject
}
subproject baseSubproject icc "base/base.icc", ics "base/base.ics"
{
```

```
            depends on utilSubproject
}
subproject utilSubproject icc "util/util.icc", ics "util/util.ics"
{
}
```

Go to the Source page in the Project section now to type or paste in these directives, before advancing to the next tutorial.

Summing up Tutorial 2

What have you learned so far?

In this tutorial you were introduced to the basic concepts of projects in VisualAge C++ Version 5.0. You learned that:

- Configurations and codestores form subprojects
- The subprojects are organized in a project file
- Project file language is descriptive, easy to read, and similar to configuration file language

In the next tutorial you will get a closer look at how subprojects are grouped into builds, to give you flexibility in choosing compiling and linking scenarios.

If you want to learn more about project file directives, read some of these references in the online help:

- Projects
- How Project Files are Processed

Tutorial 3: Build a Subproject

In this tutorial you will try building the subproject utilSubproject, then optimize it and structure the rest of the project to make it easier to maintain.

Along the way, you will also be introduced to the basics of customizing pages in the IDE, creating files, and editing.

First, build the subproject utilSubproject:

1. Go to the Overview page in the Project section.
2. Select the subproject graphic () for utilSubproject.
3. Click **Build** (at the bottom of the page).

You may see a message warning you that the subproject needs to be optimized, and listing the numbers of primary and non-primary sources. The message asks if you want to stop the current build. For this tutorial, click **No**. We will address this message in the next step.

The status line at the bottom of the window shows the progress and the outcome of the build.

Optimizing Subprojects

In the previous part of the tutorial, you built the subproject utilSubproject. At this or other times, a message may have appeared when you started the build, suggesting that you *optimize* the configuration. In the next part you will learn how to do that.

Optimizing a configuration means reducing the amount of superfluous processing that takes place when header files are included by more than one source file.

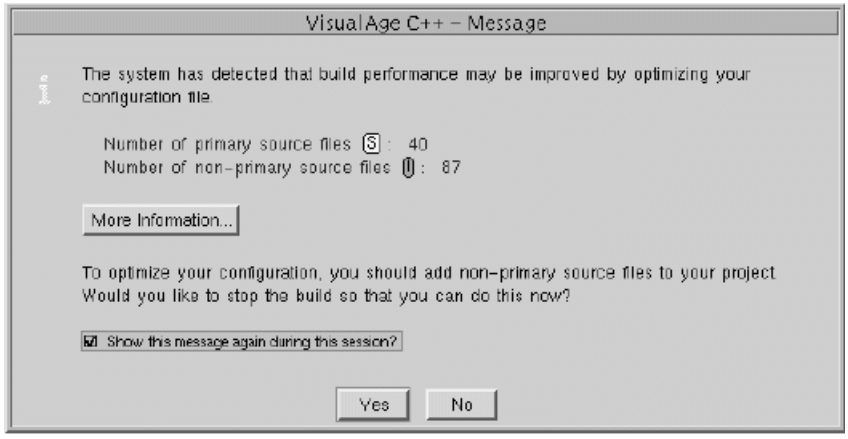
With the incremental compiler, there is no need to repeatedly include header files. Each time you launch a build, the compiler first checks the configuration file to see which sources are listed there. Sources listed in the configuration file are called *primary* sources. *Secondary* sources are those files that are used in the build, but do not appear in the configuration file. That is, they are included by other files. When you optimize, you reduce the number of secondary sources and increase the number of primary sources.

Why are primary sources more efficient? Since the compiler checks for these first, it "knows" about them. That is, it processes them once and saves the information in the codestore. If it encounters these files again, even in an #include directive, it does not bother processing them again. However, if it finds an #include directive for a file that is not already "known" to be a primary source, it follows the instruction, even if the file has already been processed several times.

The IDE provides some handy features for identifying secondary sources.

Find Included Sources in the IDE

When you built utilSubproject you may have seen a message like the following:



The message indicated that your subproject should be optimized. In this step we will do as the message suggests, and *promote* the secondary sources in utilSubproject to the configuration file.

Locate the Included Sources

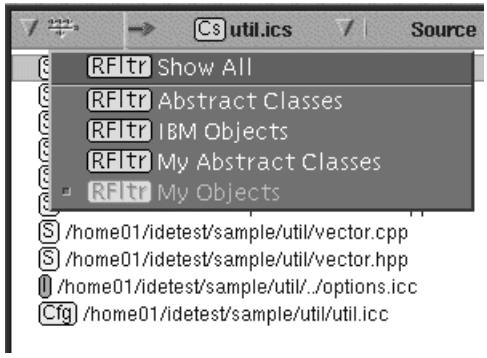
Go to the Codestore section for utilSubproject. Select the Source Files view on the Source Files page.

This view shows a list of sources (Ⓢ). Files listed with this glyph are those that appear in a source directive in the configuration file. In other words, they are primary sources.

All other files that have been used in a build are usually displayed as 0, or included files. The configuration file is displayed as Cfg .

However, the Source Files view does not currently show the included files. By default, a *filter* suppresses the system files to simplify the view.

Change the filter by pulling down the filter menu (the ▦ in the title bar) and selecting **Show All** from the list of filters:

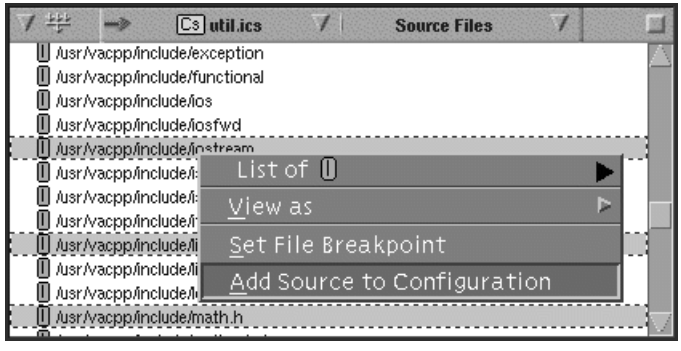


Now scroll to the bottom of the pane: you should see a much longer list of included files.

Promote the Included Files

Promoting a file means adding it to the configuration file in a source directive. To do this:

1. Select the following included files. Select the first one, then hold down the **Ctrl** key, and select the others. (Note: you will have to release the **Ctrl** key in order to scroll up or down.)
   - usr/vacpp/include/iostream
   - usr/vacpp/include/list
   - usr/vacpp/include/math.h
   - usr/vacpp/include/stdlib.h
   - usr/vacpp/include/string
1. Right-click on the selected files.
2. Select **Add Source to Configuration** from the pop-up menu:



3. In the **Add to Source** dialog, select the **Add Source to Source Directive** radio button. Select the directive that currently includes pthread.h.

4. Click **Apply**.

5. To update the view, select Refresh Configuration Viewsfor the util.icc  object from the **Project Workbook** menu.

To see the changes, switch to the Source page in the Configuration section. Now, in addition to the original sources (.hpp and .cpp files), you will also see source directives for the system header files you specified. The Source files view should also list more objects as  , and fewer objects as  .

Promoting included sources is the most important way to improve build efficiency. If you are interested in exploring other ways, you can bookmark this page, and read "Chapter 8. Migrate to Incremental Compilation" on page 99 before continuing the tutorial.

In the next part of this tutorial, you will learn how to organize the subprojects within a project to improve maintainability.

Organize the Project

In the previous section, you improved the performance of a single configuration file by reducing the amount of reprocessing of included files. In this section, you will organize your project file by identifying common options that are processed repeatedly in a project, and grouping them in a separate configuration.

The configuration files for the four subprojects in your project all require some of the same options:

- gen(rtti, yes)
- link(linkwithsharedlib, yes)
- defaults(xlC_r)
- define("USE_IPF", "1")

We will remove these options from each configuration file, paste them into a new configuration file, and replace them with an include directive and the name of a new options group in the original four configuration files.

To make this process easier, and to introduce some of the customization features of the IDE, you will also rearrange one of the views in the Host section so that you can view two different sources in one page, and cut and paste between them.

Why Separate the Options?

Grouping the common options in a single file will make your project easier to maintain. With the options in one place, you will only have to make changes once if you later decide to change the options, or apply conditional processing to the options.

Create a New File

Before you can remove the common options, you will need a new file to paste them into.

Go to the Host section, and click on the File System page. There are currently three views:

- The upper left pane, the File Tree view shows the directory structure of the host machine.
- The upper right pane, the Files view, is linked to the first pane, and shows the files contained in each directory as it is selected in the File Tree view.
- The lower pane is linked to the Files view; as soon as a file is selected in the Files view, the lower view defaults to a Source view and displays the file contents. If you select a directory in the Files view, the lower view defaults to a Details view.

Before starting the steps on this page and the next page, it is a good idea to maximize the screen space, so that there is room to see the contents in your Source views clearly. To do this, click the Maximize button in the upper right corner of the IDE (it is located on the same bar as the menus, at the far right end). The section tabs and page buttons will temporarily disappear, leaving more space for the panes.

Later, if you want to return the tabs, you can click the same button again. To temporarily see the tabs, hold the mouse pointer over the area where they are normally displayed.

Create a New File

1. First, select the lower pane on the File System page, even if it is currently empty. The file you create will be opened in the pane that is currently selected.
2. From the Project Workbook menu, select **Open or Create File**.
3. In the Open or Create File dialog, browse to the directory you created for your copy of the tutorial files. In the **Selection** field, type the name of the new file: options.icc. The full path of the file should appear in the **Selection** field.
4. In the **File open location** part of this dialog, ensure that the **Current pane** radio button is selected.
5. Click **Open**.

The new, empty file appears in the lower pane. Notice that:

- The arrow in the pane title bar appears broken. This is because the pane is no longer displaying an object from a linked view: it is displaying an object you created, so the link is temporarily disconnected. As a result, the contents of this pane will not change when you select other objects in the Files view.
- The object representing the new file does not appear as an or object.

  It is an (file) object. This is because this new file has not been added to any configuration. It is not yet recognized as a source.

In the next section, you will create a page layout that makes it easy to browse through the other configuration files and paste into this one.

Create a Custom Page

You have now created a new file, in which you will collect common options from the four subprojects to make the project easier to maintain.

To cut and paste the options, it would be handy to have two Source views open. However, no default pages in the IDE offer two Source views. In this section, you will customize a page in the Host section and give the new page a new name.

Create a New Pane

1. Select the lower pane, which is currently displaying your new file.
2. Hold down the **Ctrl** key, and move the mouse pointer to the right edge of the view.
3. When the pointer changes to a large arrow, drag the arrow toward the centre of the view and release the mouse.

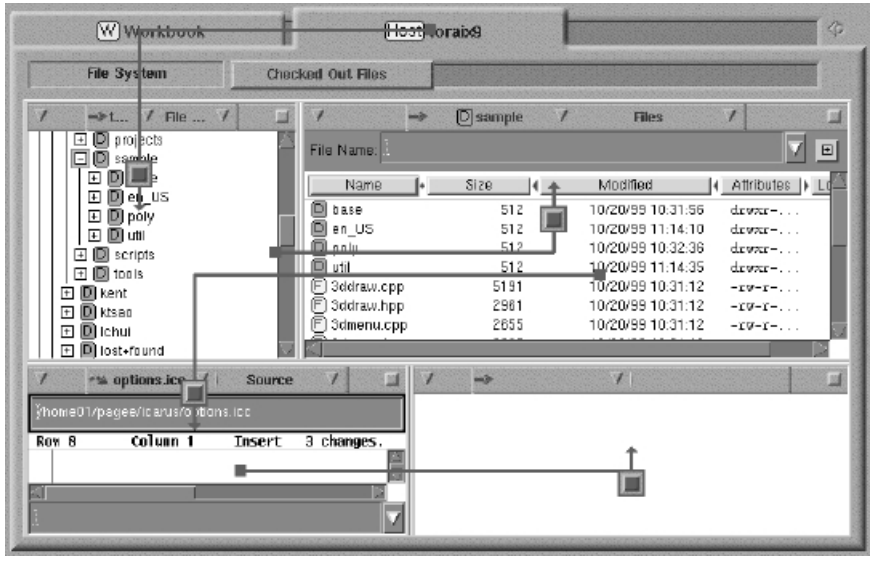
You have just created a new pane. It is empty, and has no view type. By default, whenever you create a new pane, the new pane is linked to the last pane that had focus.

In this case, the Source view was selected when you created the new pane. However, you want this new view to show the source for a different file.

Change the Input to the New Pane

As you browse for the configuration file for each subproject in the File Tree view, the files in each directory are listed in the Files view. In order for the new pane to show a source view of a file, it must be linked to a pane in which you select individual files. We will move the link for this pane so that it connects the new pane to the pane containing the Files view.

First, take a look at the current layout of the page. Select **Show Link Diagram** from the **Page** menu. The links between the panes look like this:



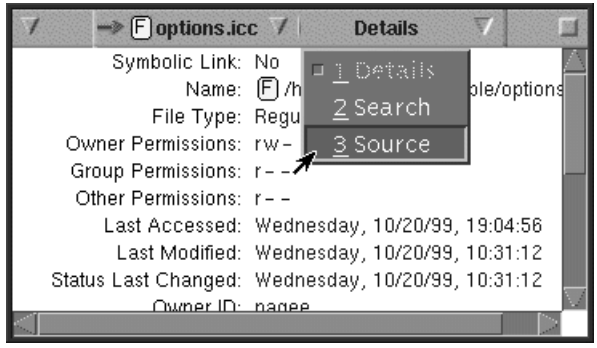
The link for the new pane originates in the lower left pane.

Moving the Link

1. Clear the link diagram by pressing Esc. (You can also move links in the link diagram, but the method you will use here saves time).
2. Hold the mouse pointer over the linking arrow in the title bar of the new pane (➡), click, and begin dragging it toward the pane directly above.

   The mouse pointer becomes a chain link  . (If it does not, release the mouse and try again: dragging by any other part of the title bar will move the pane).
3. Release the mouse when the link is anywhere inside the upper right pane.

That's all! Your new pane is now linked to the Files view. If the view type is not Source, change it now by selecting **Source** from the View Type menu:



Now you can edit each configuration file by simply selecting the file in the Files view.

Save the New Page Layout

You can easily save this new page layout for future use:

1. Select **Save Page Description as** from the **Page** menu.
2. Type a name for the new layout in the **Page Description Name** field. The name can be several words if necessary, such as `Browse and Edit`.
3. Make sure the first radio button, **Add as New Page Description**, is selected.
4. Click **OK**.

A new page button with the name you typed appears. The original page layout for the File System page remains intact, and you can switch between them easily. If you want to change the name later, simply repeat the steps above, but select the second radio button, **Replace Existing Page Description**, instead.

In the next section, you will use the new page layout to find each of the four configuration files and move the common options into the new file.

Group Common Options in a Separate Configuration File

Now that you have set up a handy layout for moving content across two open files, you can use it to cut the common options from the four configuration files in your subprojects, and paste them into the new file, options.icc.

In your new page, Edit and Browse (or the name you chose), locate the first configuration file, util.icc, in the /util subdirectory. The contents are shown in the lower right pane.

1. Highlight the following information from util.icc, with the mouse, the Shift + arrow keys, or editor shortcuts:

```
option SubprojectOptions = gen(rtti, yes)
{
   option defaults(xlC_r), link(linkwithsharedlib, yes),
   define("USE_IPF", "1")
```

2. Cut this from util.icc, and paste it into options.icc.
3. In options.icc, create a group for these options, named ProjectOptions, as follows:
```
option ProjectOptions = gen(rtti, yes), defaults(xlC_r),
   link(linkwithsharedlib, yes), define("UES_IPF", "1")
```
4. Now, in **util.icc**, add ProjectOptions into the current option directive applied to the target, as follows:
```
option ProjectOptions,link(exportAll)
  {
    target type(shr) "until.a"
       {....
```
5. At the top of util.icc, add an include directive to include the file containing the definition of the options group ProjectOptions:
```
include "../options.icc"
```
6. Now remove the same options from each of the remaining three configuration files, replace them with the name of the ProjectOptions group in each file, and add the include directive to each file. The remaining configuration files are:
   - poly/poly.icc
   - base/base.icc
   - model3d.icc
     (note that in model3d.icc, the relative path will be different: `"options.icc"`)

To save the changes, launch a new build. Press **Ctrl+Shift+B** or click **Build**.

Why is the New File a .icc File?

You could just as easily save the new file as a text file or any other kind of header file. As long as the contents of the file can be recognized and interpreted as configuration directives when the include directive is processed, you can give the file any extension that is convenient for you. In this scenario, we have chosen to call the new options file a configuration file for two reasons:

- When the file is opened in a Source view in the IDE, it is parsed as a configuration file. That is, any errors in syntax or spelling of directives will be caught immediately.
- Future maintenance will be easy: it will be obvious that the contents are configuration directives, even if the file is not opened.

Summing up Tutorial 3

In this tutorial you were introduced to the concepts of build optimization, and project organization.
You learned that:

- Promoting secondary source files optimizes your build's performance
- Grouping common options in a single file makes your project easier to maintain
- The IDE can be easily customized to suite your needs

In the next tutorial you will set up different build scenarios, create a new build and adjust a configuration file to handle conditional processing.

To learn more about optimizing configuration files, try reading "Chapter 8. Migrate to Incremental Compilation" on page 99. The complete *Migration Guide* is also available in PDF form, in the /pdf/ subdirectory of the VisualAge installation directory. Look for migrate.pdf.

Tutorial 4: Builds

So far in these tutorials, you have created a project that consists of four subprojects, but you only have one build scenario defined for these subprojects. That scenario, the default BuildAll object created by the SmartGuides, sets only the normal options required for development.

A *build* is a group of one or more subprojects, environments and variables, associated by **use** directives in a project file.

A subproject is only compiled and linked if it appears in a build. A build takes the general form:

```
build name_of_build
{
use name_of_subproject1
use name_of_subproject2
.
.
}
```

When you set up a project using a SmartGuide, unless you use the Advanced setup to define your own builds, the SmartGuide creates a default build that assumes all the subprojects will be compiled and linked. Builds can also use variables. For example, if you want to define two scenarios (such as an optimized build and a normal build) for building the model3d subproject, you can set up two different builds this way:

```
build optimized
{
  use Model3d
  opt=1
}
build default
{
  use Model3d
  opt=0
}
```

Each scenario can make use of a different set of options. When you want to build for one scenario, all you have to do is:

1. Select a Bld object, either the default build or the optimized build, from the Overview page.
2. Click **Build** at the bottom of the view.

That's it! From one easy-to-read text file, you can now set up a complete range of testing, development, and production scenarios.

In the next part, you will create a build for the optimization scenario, and create build variables to control which options are used with each scenario.

Create a Build

In this section, you will set up a new build scenario, or create a new build, and adjust the options.icc configuration file to handle conditional processing for choosing one build or the other.

To create a new build and create variables for it:

1. Go to the Project File page in the Project section. This page provides a variety of views for editing your project file.
2. In the upper left pane, the Builds view shows a schematic diagram of your project file, organized by builds. The object at the highest level is the build object (Bld). Select this and right-click on it.
3. From the pop-up menus, select **Add directive after** and **Build**, to add a new build directive after this build directive in the project file.
4. In the **Add Build after** dialog, type `optimized` in the**Name** field.
5. In the **Subprojects** area of the dialog, select each one of the four subprojects in turn from the pull-down list. Click **Add** after selecting each subproject.
6. In the **Variables** area, select the **Create assignment** radio button. In the **Name** and **Value** fields, type `optimize` as the name of the variable, and assign it the value of 1.
7. Click **Add**.

8. Click **Apply** to update the project file.

You now have a second build scenario defined for the project. However, at this point there is no effective difference between this scenario and the default scenario, BuildAll.

Next, you will edit the options.icc configuration file to select different options based on the value of the optimize variable.

Add Conditional Processing to options.icc

Now that you have the facility to choose one of two builds, you need to ensure that the options applied for each scenario are appropriate for each build.

If you recently finished the previous tutorial, you should still have options.icc displayed in a Source view either in the File System page or the custom page you created in the Host section. However, this view is small, and it would be difficult to edit the file in it.

To open the file in the largest possible viewing area, you can do either of the following:

- Maximize the Source view in the File System page or in your custom page by clicking the maximize control (), a small raised square at the far right end of the pane title bar.
- Open the file into its own workbook section. Select the file in the Files view or File Tree view, right-click, and select **View as** and **Section** from the pop-up menus.

First, you will define which options will be applied when the optimized build is selected. Type or paste the following at the top of options.icc:

```
if $optimize==1
  option opt_build=opt(level,2)
else
  option opt_build=null
```

The value of `opt` will be evaluated each time a build object is selected and a build is begun. However, we have still not connected the outcome of this processing to the targets of the subprojects.

Currently, all four subprojects include the configuration file options.icc, which in turn contains the definition of the options group `ProjectOptions`. To make sure this group includes the optimize option group, you must add it to the defintion, as follows:

```
option ProjectOptions=opt_build,
        gen(rtti, yes),
        defaults(xlC_r),
        link(linkwithsharedlib, yes), define("USE_IPF", "1")
```

In the next part of the tutorial, you will set the opt variable in your original build, to ensure your builds are consistent.

Set Build Variables in the Default Build

You have now defined a new build, and created a mechanism for passing information about the build to the subprojects, using conditional processing in the options.icc file. However, one last task remains: you must set the value of the build variable in the original build.

To do this:

1. Go to the Project section, and select the Project File page.
2. In the Builds view, select the original build (BuildAll).
3. When you select this object, the linked view on the right side of the page becomes a Change Build view.
4. In the **Variables** area of this view, create the same variable you created in the new build, but with a zero value:
   a. Select the **Create assignment** radio button.
   b. In the **Name** field, type `optimize`.
   c. In the **Value** field, type `0`.
   d. Click **Add**.
   e. Click **Apply**.

The project file is now updated, and both builds contain a value for the build variable, optimize, that controls whether the build makes use of optimization options.

Now, no matter which build object you select, the appropriate options will be applied to the build. You will not have to change the options again, and adding a new scenario is easy:

- create a new build directive
- select the subprojects you want to use
- create a variable to identify the new scenario
- add conditional processing to the common configuration file, if new options are needed for this scenario

Build the Project

Now that you have created two builds in your project, you can test the build process and see how easy it is to launch compilation and linking for two scenarios from one view, and then to check to ensure the options you expected to use were applied.

Build the Optimized Scenario

Before starting to compile the build you set up earlier in this tutorial, you must close any subprojects that were opened in memory in the default build.

Select each open subproject and click Close. When the Cfg and Cs graphics disappear from the subproject description, and the status reads Closed, the subproject is closed. If the subprojects are not closed in this build, you will get an error message when you try to build the optimized build.

1. On the Overview page of the Project section, select the build object (Bld) for the optimized build you created.
2. Click the **Build** button at the bottom of the view.

When the build is complete, all the subprojects are open in memory. To see how the configuration file for any subproject was interpreted, open a configuration file as a section:

1. Select any of the open configurations.
2. Right-click on the configuration, and select View as - Section from the pop-up menus.

A workbook section is opened. In this section, select the **Advanced** page.

The right pane of the Advanced page shows the Interpreted Configuration view. This view shows you exactly how the file was interpreted, and shows how conditional processing was evaluated. For example, your view of the utilSubproject configuration file looks like this:

```
include "../options.icc"
    {
      if 1 == 1
      {
        option opt_build = opt('optimize', 'yes')
      }
      else
      {
        option opt_build = null
      }
      option ProjectOptions =
                    opt_build,
                    gen('rtti', 'yes'),
                    gen('initauto'),
                    defaults('xlC_r'),
                    link('libpathout', ".:/usr/lib/threads:/usr/vacpp/lib:/usr/lib:/lib");
```

You can see that the directive “if $optimize == 1” has been evaluated to “if 1 == 1”. As a result, the definition of the option group opt_build is opt(optimize, yes), and the text is displayed in green. The definition “option opt_build=null” is gray, indicating that it was not chosen.

The Interpreted Configuration view is a simple but effective way to trace suspected errors in a configuration file: it is easy to see exactly how the logic was followed in a build.

Sometimes, your build scenarios require no different options than a normal build, but may need a different set of environment variable settings. This can also be controlled easily through **use** directives in the project file. The next part of the tutorial shows you how to see which build variables have been assigned to your subprojects in each of the builds you have set up.

Environments

In the previous part of the tutorial, you learned how to create build variables and control the options used for individual build scenarios.

When you set a variable at the project file level of a project, VisualAge C++ understands the variable as part of the total *environment* for each subproject.

There are various ways of representing environments and parts of environments in the IDE and in project file language:

an *assignment* is a single variable, which you create. The variable you created in this tutorial is an example of an assignment. In the project file, it takes the form `assignment=value`. It can have a numeric or string value, for example:

```
opt=1
opt='yes'
```

a *variables object* represents a group of assignments. Like a group of options or a group of source files, it can have a name, and can contain any number of assignments. Variables objects are associated with builds in the project file by use directives. For example, here is a build that makes use of the variables group called opt_settings:

```
build optimized
  {
  use subproject 1
  use subproject 2
  use opt_settings
  }
```

an *environment* is the total collection of the values inherited from the system environment, the values set or overridden by assignments you created,

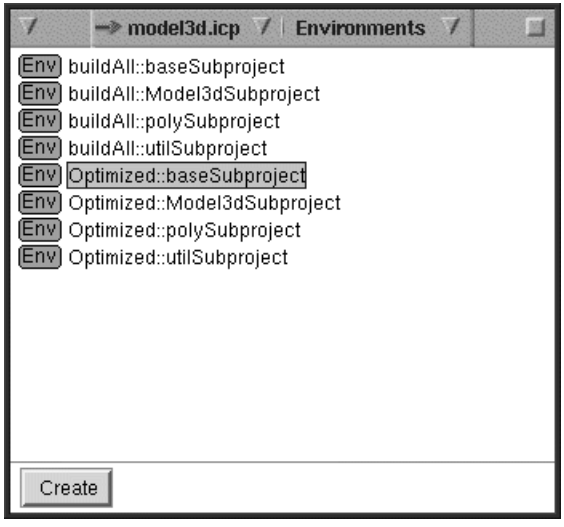
and values contained in variables groups. Like a variables group, environment objects are named. Environments are not selected in the project file, however. They are associated with run specifications, when you run or debug your application.

In the next part of the tutorial, you will use the Environments view to get a quick overview of what information about environment variables and build variables is applied to each subproject.

Browse an Environment

Go to the Project section, and select the Environments page.

Currently, the Environments view shows eight environment objects (Env):

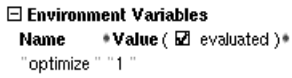


Each environment object represents the environment — the sum of the system environment plus any variables set within the project — as it is interpreted for each subproject, each time a subproject is used in a build. In your project, there is a default build that uses all four subprojects, and a build for optimization, which you created.

Select any one of the environments for the Optimized build. On the right, the Change Environments view shows all the values that are interpreted:

- The variables and values under **Evaluation context** are those in the system

- The variables under **Environment variables** are those you have set within the project, in the Optimized build:



An environment can also be explicitly created. When you create an environment, you can use it to set up the conditions for running an executable, and then apply it to a run specification when you are ready to run or debug. Creating an environment is simple:

1. In the Environments View, click **Create**. An object named `Unnamed Environment` *x* is added to the list of environments.
2. In the Change Environments view, edit the Env field to name the environment.
3. If you want your environment to use the values of another predefined environment, select the environment from the list under **Inherited Environments**.
4. To add additional assignment variables, enter them in the fields under **Name** and **Value**.
5. Click **Change**.

Now you can easily associate this with a run specification:

1. Switch to the Project section.
2. On the Overview page, open the Model3dSubproject codestore. Select the codestore object ( ) and double-click or click **Open**.
3. In the Codestore section, select the Debug page.
4. In the Run Specifications view, pull down the list next to **Environment**. Select the new environment from the pull-down list. It will be interpreted when you click **Debug** or **Run**.

Summing up Tutorial 4

In this tutorial you were introduced to the concepts of builds, build variables, and environments.
You learned how to:

- Set up different build scenarios
- Edit a configuration file to handle conditional processing for choosing between build scenarios
- Quickly see how a configuration file was interpreted in a build
- Quickly see a list of all the build variables applied to any subproject

Other Useful References

Here are some recommendations if you are interested in learning more, or looking for more detailed information.

In the online help, you can find the following:

Equivalent Batch Compile-Link and Incremental Build Options can be found in the Reference section, under Build References.

"Chapter 7. Prepare to Migrate to Version 5.0" on page 97 is also available in PDF format, and is included in this book. Online, this page can be found in the Concepts section, under Build References. This can be a useful reference whether you are moving from VisualAge C++ Version 4.0, or from a batch compiler.

Other tutorials are available in the online documentation. Look for The VisualAge C++ Version 5.0 Tutorials in the navigation. For more detailed information on some of the concepts introduced, try reading some of these topics in the online Concepts section:

Projects and Subprojects
Incremental Configuration Files
Targets
Types of Source Files

VisualAge C++, Version 5.0 Migration Guide

Chapter 7. 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 18.

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 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

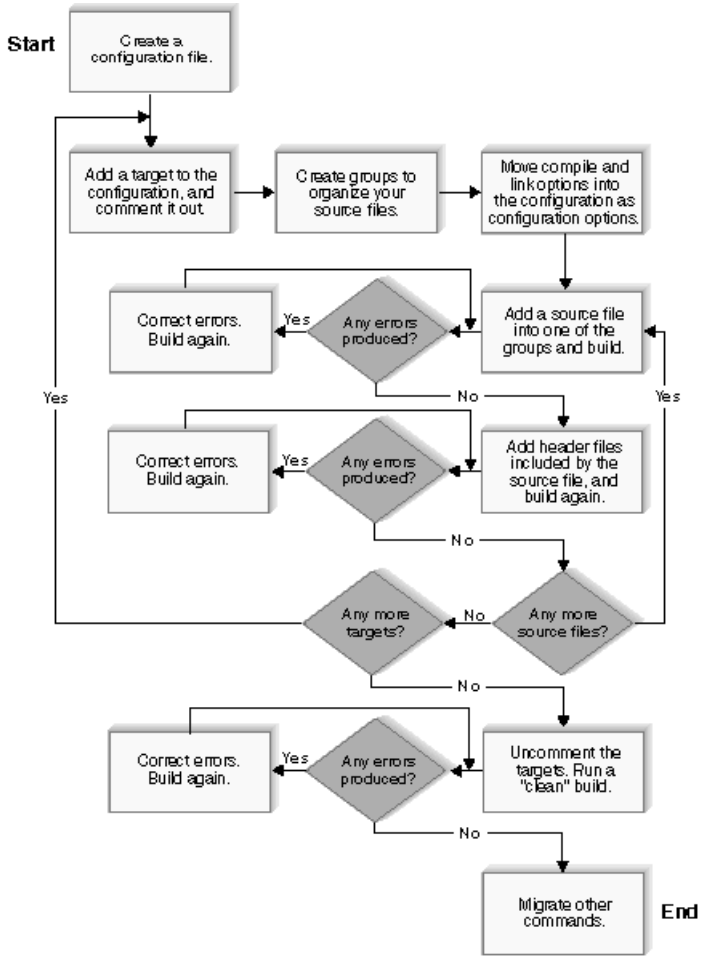
While such features as incremental compilation and the Integrated Development Environment are familiar to users of Version 4.0, there are some changes you should be aware of. For example, VisualAge C++ now supports multiple codestores in a single project. We recommend you try the tutorials to learn about 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 10. Migrate to Version 5.0 Batch Compilation" on page 119.

Chapter 8. Migrate to Incremental Compilation

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



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

1. Create a configuration file, 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. Create groups 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 them to configuration options.
5. Add source files 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 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 111 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 133.
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 145
"Promoting Source Files to the Configuration File" on page 141
Sources

RELATED REFERENCES

"Chapter 12. Special Considerations for Migrating to Incremental Compilation" on page 133

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 102.

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".

RELATED CONCEPTS

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

RELATED TASKS

"Chapter 8. Migrate to Incremental Compilation" on page 99

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".

RELATED TASKS

"Chapter 8. Migrate to Incremental Compilation" on page 99

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 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 102).
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 102).
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 105.

RELATED CONCEPTS

Configuration Files

RELATED TASKS

"Preparation for Grouping Source Files for Migration" on page 102
"Chapter 8. Migrate to Incremental Compilation" on page 99

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 **Open as a Workbook 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. Change the view in the right-hand pane. In the **View Types** menu, select **Convert Batch Options**.
4. In the upper left-hand pane (**Source and Targets** view), highlight the source file (), or target () to which you want to apply the compile and link 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 113
"Chapter 8. Migrate to Incremental Compilation" on page 99
"Migrate Other Commands" on page 111
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 **Host** tab, and the **File System** page button. This view allows you to browse your system.
2. Locate a source file from your application and right-click on it.
3. To add the source to your configuration file, select **Add Source to Source Directive** from the pop-up menu.
4. In the **Add to Source Directive** dialog, select the PRIMARY_SOURCES source group, which you created in "Create Source Group Directives for Migration" on page 103.
5. Click **Apply**.
6. 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 Troubleshooting 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 141

RELATED TASKS

"Chapter 8. Migrate to Incremental Compilation" on page 99

RELATED REFERENCES

Options for Compatibility with Previous Versions of VisualAge C++

"Common Errors when Migrating" on page 121

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 Source Directive** from the pop-up menu.
6. 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.
7. Build the project, and correct any errors.
8. Return to the **Source Files** view. You should no longer see the system file you promoted displayed as an object.
9. 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.
10. 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.
11. 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 8. Migrate to Incremental Compilation" on page 99.

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 111 for information on doing this.

RELATED CONCEPTS

"Promoting Source Files to the Configuration File" on page 141

RELATED TASKS

"Add Source Files to the Project for Migration" on page 106

Add Another Target to your Project for Migration

If you have already gone through the migration process described in "Chapter 8. Migrate to Incremental Compilation" on page 99, 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 133.

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 in the online help.

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 115.

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 8. Migrate to Incremental Compilation" on page 99

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 8. Migrate to Incremental Compilation” on page 99, 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 125
“Makefile Rules Mapped to Configuration File Directives” on page 178

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 9. 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 116.

Some considerations when deciding whether to combine multiple codestores in one project are listed in "Application Requires Multiple Targets" on page 133.

RELATED CONCEPTS

"Application Requires Multiple Targets" on page 133

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 131 for information on other changes you may have to make.

RELATED TASKS

"Adjust for Using the IBM Open Class Library" on page 131

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 133.

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 9. Migrate from Version 4.0 to Version 5.0" on page 115

Chapter 10. 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 122 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 11. 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” on page 122.

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 122 may be applicable to you.

RELATED TASKS

“Chapter 8. Migrate to Incremental Compilation” on page 99

RELATED REFERENCES

“Coding Adjustments for Orderless Programming” on page 125“Common Template Problems” on page 127
“Errors Due to Changes in Compiler Behavior” on page 122
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 &&, |, ||, ^, ~, &, &=, |=, ^=, ! 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 121
"Common Template Problems" on page 127

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 121
"Errors Due to Changes in Compiler Behavior" on page 122

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 initalizer?
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 121

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 10. Migrate to Version 5.0 Batch Compilation" on page 119
"Chapter 8. Migrate to Incremental Compilation" on page 99

RELATED REFERENCES

"Common Errors when Migrating" on page 121
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 109

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 8. Migrate to Incremental Compilation" on page 99
"Chapter 10. Migrate to Version 5.0 Batch Compilation" on page 119

Chapter 12. 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 8. Migrate to Incremental Compilation" on page 99. 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 Application Uses 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 8. Migrate to Incremental Compilation" on page 99
Migrate 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 8. Migrate to Incremental Compilation” on page 99.

RELATED CONCEPTS

Configurations
The One-Definition Rule

RELATED TASKS

Build in a Team Environment
"Divide a Single-Codestore Project into Multiple Codestores" on page 116
Produce Multiple Targets from One Build

RELATED REFERENCES

"Chapter 12. Special Considerations for Migrating to Incremental Compilation" on page 133

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 101, 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 110 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 any pre-build and post-build targets your makefile called for. See "Migrate Other Commands" on page 111 for more details on doing this.

RELATED CONCEPTS

"Application Uses Meta Header Files"
"Meta Source Files" on page 143

RELATED REFERENCES

"Chapter 12. Special Considerations for Migrating to Incremental Compilation" on page 133

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 101, 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 105.
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 110, 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 any pre-build and post-build targets your original makefile called for. See "Migrate Other Commands" on page 111 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 135
"Meta Source Files" on page 143

RELATED TASKS

"Add Source Files to the Project for Migration" on page 106

RELATED REFERENCES

"Chapter 12. Special Considerations for Migrating to Incremental Compilation" on page 133

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 informations on which types of commands to migrate, read "Migrate Other Commands" on page 111.
- 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 172.

  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 400 for IBM VisualAge Version 4.0 and later, and can be used as follows:

```
#if __IBMCPP__ >= 400
 #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 8. Migrate to Incremental Compilation" on page 99

RELATED REFERENCES

nokeyword Language Option
Options for Compatibility with Previous Releases
orderednamelookup Language Option
"Chapter 12. Special Considerations for Migrating to Incremental Compilation" on page 133

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 8. Migrate to Incremental Compilation" on page 99.

RELATED CONCEPTS

Sources

RELATED REFERENCES

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

Chapter 13. 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 109

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 135
"Application Uses Meta Header Files" on page 136

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 177
"Use the Standard Iostreams Library with the IBM Open Class Library" on page 177

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.

< 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.

RELATED TASKS

"Chapter 10. Migrate to Version 5.0 Batch Compilation" on page 119
"Chapter 8. Migrate to Incremental Compilation" on page 99

Chapter 14. Other Migration References

Options for Compatibility with Previous Versions of VisualAge C++

Three predefined option 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 option 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**
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

Equivalent Batch Compile and Incremental Build Options

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
-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

Old Option	Description	Configuration File Option
-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
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*)

Old Option	Description	Configuration File Option
-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)
-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*>	Attempt to inline funtion listed by names in <nm>.	opt(inline[, yes]) opt(inlineFunc,name)

Old Option	Description	Configuration File Option
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)

Old Option	Description	Configuration File Option
-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)
-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)

Old Option	Description	Configuration File Option
-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)

Old Option	Description	Configuration File Option
-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
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	Honor 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()

Old Option	Description	Configuration File Option
C++ -qinitauto=*value* C -qinitauto=<*hh*>	Initialize 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,...)
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)

Old Option	Description	Configuration File Option
-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)
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*)

Old Option	Description	Configuration File Option
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

Old Option	Description	Configuration File Option
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
-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*)

Old Option	Description	Configuration File Option
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)

Old Option	Description	Configuration File Option
-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)
-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

Old Option	Description	Configuration File Option
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*)
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 173
- "The Standard String Templates" on page 173
- "Localization Classes and Templates" on page 173
- "The Containers, Iterators and Algorithms Libraries (the Standard Template Library)" on page 174
- "The Standard Numerics Library" on page 174
- "The Standard Input/Output Library" on page 175
- C++ Headers for the Standard C Library (page 175)

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
<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
<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" on page 177

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"
	}

Makefile rule	Equivalent Configuration file directive
run_target_rules: <previous targets> a.res a.res:a.rc irc command	source “a.rc”

RELATED CONCEPTS

Configuration Files

Contact Us

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