IBM VisualAge C++ Professional for AIX

Getting Started

Version 4.0

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

First Edition (June 1998)

This edition applies to Version 4.0 of the VisualAge C++ product, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

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

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

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

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

Contents

Notices

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

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

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

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

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

Programming Interface Information

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

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

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

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

Trademarks and Service Marks

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

AIX
AS/400
DATABASE 2
DB2
IBM
Object Connection
Operating System/2
OS/2
OS/400
Presentation Manager
SAA
Systems Application Architecture
TeamConnection
VisualAge
Workplace Shell

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

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

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

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

PC Direct is a registered trademark of Ziff Communications Company and is used by IBM Corporation under license.

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

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

Industry Standards

VisualAge C++ supports the following standards:

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

About This Book

Welcome to the VisualAge C++ getting started. VisualAge C++ has been completely revamped for this release, to make your job of developing C++ applications easier. Before you jump right in, you may want some background information on the product's new features, or you might want to be guided through creating your first programs with the product. Whether you are new to VisualAge C++, or are a seasoned veteran of previous releases, you'll find the getting started useful.

VisualAge C++ is a complete, integrated environment for creating C++ applications.

VisualAge C++ gives you interactive, visual programming tools and an extensive library of classes. See the online help on the IBM Open Classes for more details on the classes.

Reference to VisualAge C++ in this book should be interpreted as VisualAge C++, Version 4.0.

This manual is divided into three sections:

- **What's Cool**

  This portion lets you know about some of the new features in VisualAge C++. It will take you 20 to 30 minutes to read all of the sections. Reading What's Cool will help you understand the changes in this release and how they will affect you.
- **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 exercise, or learn to navigate before starting your own projects.
- **Try VisualAge C++**

  Try VisualAge C++ will help you master the IDE quickly. It is an in-depth tutorial that covers developing two types of applications: a web-based application, and a graphical user interface application created with the Visual Builder. Try VisualAge C++ will take you anywhere from four to eight hours to complete, or you can try out only selected parts. Whatever time you spend on Try VisualAge C++, it will help you become more productive at coding in the new environment.

Data Access Tutorial

There is a separate tutorial for learning how to create a visual data access application. It takes you through the steps of creating an application that uses nonvisual, visual, and data access parts to work with a DB2 or ODBC database. The instructions are in Creating a Visual Data Access Application in the online documentation.

Who Should Use This Guide

This document is intended for application developers who are interested in developing applications in the new integrated development environment of VisualAge C++as well those programmers who want to learn about incremental compilation with VisualAge C++.

In addition, this manual introduces the concepts of programming with VisualAge C++ and explains the general process of editing, compiling, and debugging the new interface. You should have a general knowledge and experience in programming C++.

Conventions Used in this Guide

Commands appear **Like this**.

Coding examples and text that you enter appear `Like this`.

User interface controls appear **Like this**.

New terms appear *like this.*

Note: This book does not contain a glossary. Terms are defined when they first appear.

Window titles, folder names, notebooks, notebook tabs, or keys (for example, the Enter key) have no special appearance.

Mouse Button Naming Conventions

The following mouse-button naming conventions are used in this guide:

- Button 1 is used to select an object, open a file, or start an application. It is the left button unless this button has been customized to do something else.
- Button 2 is used to move or manipulate an object. It is the right button unless this button has been customized to do something else.

You can define the function of each mouse button from the Style Manager. If you are not sure how your mouse is configured, check the properties of the mouse item in your Style Manager.

Chapter 1. What's Cool!

Welcome to VisualAge C++, the innovative way to create C++ applications. What's Cool! will give you an overview of how these innovations can benefit you:

- Revolutionize your programming by using the powerful integrated development environment.
- Increase your productivity with the exciting, new compiler technologies unveiled in this version of VisualAge C++:
  - Incremental compilation
  - Reduced drudgery
  - Elimination of makefiles
  - Automatic instantiation of templates
- Make your work easier and reduce errors through visual programming for rapid application development with reusable parts.
- Diminish your learning curve with the help of an all-new HTML information system and SmartGuides that lead you through common tasks.
- Use the open technologies of VisualAge C++ to develop cross-platform applications.
- Write C++ code that adheres to the latest C++ standard.

Easy-to-Use Environment

The new Integrated Development Environment (IDE) will drastically 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 customized 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

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 to create projects and targets, and specify parts for your programs.

Code Development

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

This version of VisualAge C++ supports *orderless programming:* Orderless programming means that you do not need to declare functions before they can be called, or define classes before they are used. In fact, even the concept that one declaration appears "before" or "after" another declaration is meaningless. You can organize source code into files in any order.

For example, if you want to add a new class called "Animal", you can create an Animal.cpp file, and then enter everything relevant about animal into the file. Then, you can search through your existing application source using the quick browsing abilities of the IDE to locate the places where you want to use "Animal". You can then enter your changes, and rebuild.

Orderless programming also means that there is no need to include header files.

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 will also display 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 the Classes page of the Project section of the workbook, selecting the class code, and selecting 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 the 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 will see four major sections:

Workbook
This section lets you examine and set options that control the IDE itself.

Host This section lets you browse through files on your system.

Project This section lets you work with your C++ code.

Configuration
This section is where you set options for compatibility, optimization, and other operations.

Each section of the project workbook has a number of pages associated with it. When you click on a section tab, you will 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.

SmartGuides

SmartGuides make complicated tasks easy. They take you through a series of questions about the tasks you want to perform. The SmartGuide then uses your input to perform the task according to your specifications.

What can the VisualAge C++ SmartGuides help you do?

Project SmartGuide
Helps you set up your project.

Target SmartGuide
Helps you add a build target for your project.

Help System

Our new online, HTML-based help is organized to help you quickly find the information you need. The navigation pane lets you see where you are in the information structure, and lets you 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 navigation bar that you can click to go from one type of information to another. You can get to the VisualAge C++ Web site by clicking on the VisualAge C++ icon (if you have a connection to the internet).
2. The left frame contains an expandable index of topics available within the current type of information.
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. You can use text processing, wildcards, and logical modifiers to find the information you need. You can also specify what types of information -- concepts, tasks, reference, interface help and examples -- you want to search.

You can browse the online help outside the tools, or from within the tools' windows. F1 always gives you contextual 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. From your browser, you can even launch an IDE session on the sample project, or copy that project to a working directory, where you then can modify it to meet your needs.

Code-sensitive help provides descriptions of a keyword, class or function in a code source. To access code-sensitive help, just select a keyword or class or function in the source.

New Compiler Technologies

With VisualAge C++, Version 4.0, you will no longer have to wait a long time to complete a build, or to start up tools.

Fast Incremental Builds

Every time you build, VisualAge C++ 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 VisualAge C++, changes to a source file do not require recompilation of the header files that the source file includes.

In some situations, after a simple source code change, Version 4.0 can rebuild a program more than 10 times faster than previous versions of VisualAge C++.

In general, the more C++ files you have, the faster VisualAge C++, Version 4.0 builds compared to conventional compilers.

Database of Program Information

Databases, or *codestores* of information about your program, make up the core of VisualAge C++.

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 VisualAge C++, Version 4.0 can be extremely fast. If you add one line of code to a function, the update will happen almost instantaneously. While a traditional makefile-based system will 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 will rebuild only those particular functions.

Early Error Feedback

Because of incremental compilation, the VisualAge C++ 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 will no longer have to worry about maintaining complex header files and makefiles.

Reduced Drudgery

With VisualAge C++, Version 4.0, 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 VisualAge C++, Version 4.0, you can organize source code into files in any order. VisualAge C++ has orderless parsing, which means that the computer (not you!) sorts the declarations. VisualAge C++ also takes care of header files. Typically, header files require you to use various mechanisms to sort declarations for the compiler. (For more information on *orderless* programming, see "Code Development" on page 1.)

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;
		}

The above 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 philosophy is that the computer should serve the programmer, not the other way around. When you tell VisualAge C++ 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 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.exe" // 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.cpp" // List of C++ sources
         source type(cpp) "car.hpp"
         source type(cpp) "car.cpp"
         source type(cpp) "truck.hpp"
         source type(cpp) "truck.cpp"
         source type(cpp) "vehicle.hpp"
         source type(cpp) "vehicle.cpp"
         source type(cpp) "vlist.hpp"
         source type(cpp) "vlist.cpp"
         }
       }
     }
```

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-C++ standard (for example, true and false are not treated as keywords). However, the configuration file does not indicate that vehicle.obj depends on vehicle.cpp, vehicle.hpp, and so forth. In fact, because the target is an executable file, no object files are even produced.

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

The IDE provides a simple checkbox-based method to create and maintain 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++ 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.

Open Technology

Open technologies enable you to produce applications built from reusable parts (the parts are not bound to any particular operating system or set of standards).

You can reuse code only if you have robust class libraries. When developing applications, it is common to store the fundamental components as reusable, extensible classes for future development. The IBM Open Class Library contains the building blocks you need to develop robust and complex C++ programs.

VisualAge C++ generated code works across platforms. The source generated by our builders contains no operating-system-specific language constructs, enabling you to develop on one platform, copy the code to another platform, and then rebuild on that platform.

VisualAge C++, in conforming to the latest C++ standards, allows you to develop cross-platform applications effectively.

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 a wide variety of reusable classes across Windows NT 4.0, OS/2 4.0, and AIX 4.1.5. When you combine these classes together, you can create powerful applications:

Improved for Speed and Portability:	**New Classes:**
• Classes to create user interface (User Interface Classes) • Classes to access relational databases (Data Access Classes) • Classes to simplify abstract data types (Collection Classes) • Classes to simplify string manipulations (Data Type and Exceptions Classes) • Classes for input and output (I/O Stream Classes)	• New C++ standard classes • Frameworks of classes to code for the following: – internationalization – file systems – new portable 2D graphics – testing

Cross-Platform Development

The IBM Open Class Library is portable from platform to platform. With VisualAge C++, you can directly port interface designs from the Visual Builder, and database designs from the Data Access Builder, to all the platforms that VisualAge products support.

You do not have to worry about how your interface designs will look on other platforms. Because the IBM Open Class Library is portable, your interface designs will have the native look and feel of the "original" operating system, without you having to code the differences yourself.

As well, you will find it easy to port your code because the tools are similar from platform to platform.

Support for Latest C++ Standard

After having evolved rapidly over the last decade, C++ is now becoming more stable as standards gain approval. Many new language features have appeared (namespaces, run-time type identification, exceptions, and a rich library), along with major changes in templates and overloading. VisualAge C++ supports the language and library specified in the ISO committee draft of November 1997, except for the changes needed to support VisualAge's orderless programming.

While adhering to the new C++ standard, VisualAge C++ allows you to specify how much of that standard you want to follow when coding. As a result, you can easily move your applications to the new standards.

Visual Programming

By 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.

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

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.

Creating Interfaces and Programs Visually

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.

Accessing Relational Databases Visually

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.

What's Cool! Wrap-Up

As you read through What's Cool!, you discovered the many innovative features of VisualAge C++.

You discovered:

- The ease of working on the VisualAge C++ environment, because of the Integrated Development Environment, the new help system, and SmartGuides.
- The new compiler technologies incorporated into VisualAge C++, resulting in fast incremental builds and efficient code development.
- The open technologies incorporated into the VisualAge product, allowing you to produce applications built from reusable parts.
- The visual programming environment of VisualAge C++, which allows you to create interfaces and programs, and access relational databases, visually.

The new Integrated Development Environment will help you develop and maintain code much more efficiently.

If you take time to walk through the tour, you will learn, at a much faster rate, how VisualAge C++ can benefit you.

The Tutorial

No matter how much experience you have had working on previous versions of the IDE, or how experienced you are as a programmer, *please take time to do the exercise.*

With this release of VisualAge C++, much has changed in the IDE. When you spend time to do the exercise, you will discover how you can benefit from the changes. The result will be greater comfort and productivity when working with the new IDE interface.

Chapter 2. Tour of VisualAge

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 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.

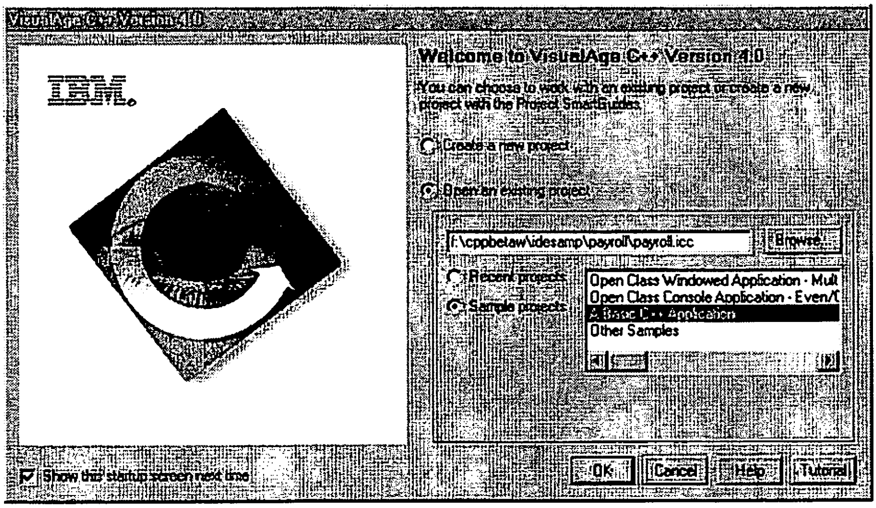
The second half of the tour, "More on the IDE" on page 26, invites you to explore further some of the concepts introduced in the first half, in any order, and suggests some references in the online help for more detailed information.

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 VisualAge folder.
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** radio button. From

the list of samples, select **A Basic C++ Application**:



VisualAge C++ Version 4.0

Welcome to VisualAge C++ Version 4.0

You can choose to work with an existing project or create a new project with the Project SmartGuides.

○ Create a new project

● Open an existing project

f:\cppbetaw\idesamp\payroll\payroll.icc

○ Recent projects

● Sample projects

Open Class Windowed Application - Mult
Open Class Console Application - Even/C
A Basic C++ Application
Other Samples

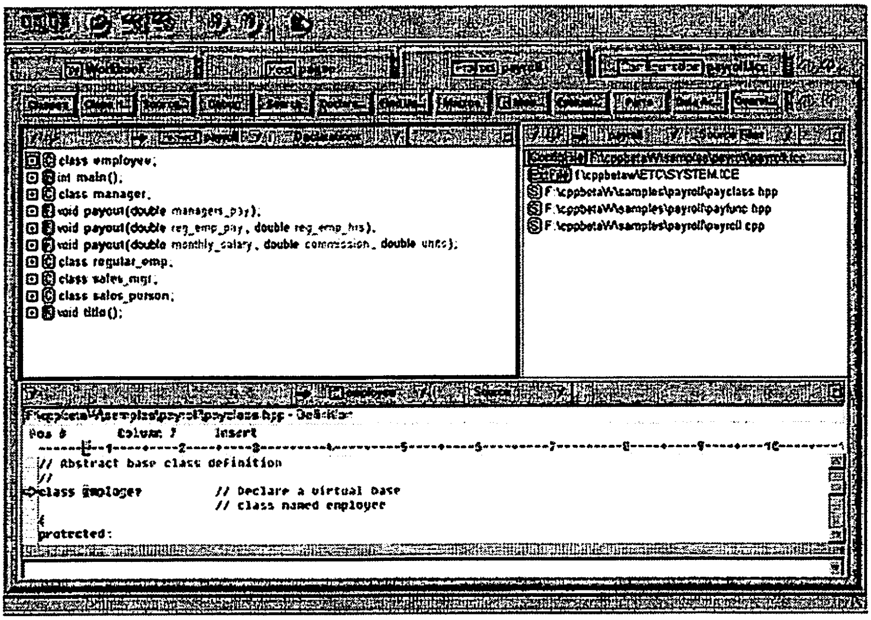
☑ Show this startup screen next time

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 Project** button
2. Browse through the **Directories** and **FoldersFiles** 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:



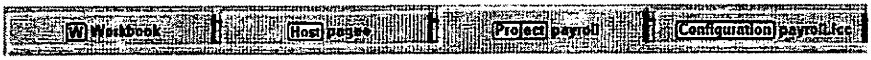
```
class employee;
int main();
class manager;
void payout(double managers_pay);
void payout(double reg_emp_pay, double reg_emp_hrs);
void payout(double monthly_salary, double commission, double units);
class regular_emp;
class sales_mgr;
class sales_person;
void title();
```

```
F:\cppbetaw\ETC\SYSTEM.ICE
F:\cppbetaV\samples\payroll\payclass.hpp
F:\cppbetaV\samples\payroll\payfunc.hpp
F:\cppbetaV\samples\payroll\payroll.cpp
```

```
Pos 8         Column 7      Insert
// Abstract base class definition
//
class employee          // Declare a virtual base
                        // class named employee
{
protected:
```

Workbook (IDE)

The workbook consists of four major sections:



Workbook
For examining and setting development environment options that control the IDE itself.

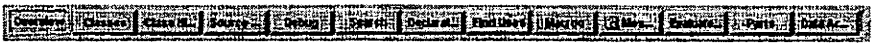
Host For browsing through files on your system.

Project For working with your C++ code. This tab only appears once you have opened a project.

Configuration
For setting options, or for adding and removing files from your project. This tab only appears once you have opened a project.

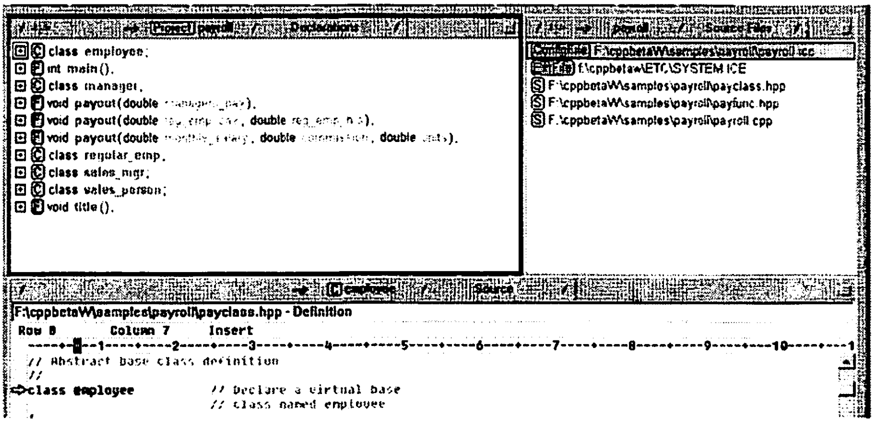
Click each tab to look at the different sections.

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**:



```
⊡ Ⓒ class employee;
⊡ Ⓕ int main().
⊡ Ⓒ class manager,
⊡ Ⓕ void payout(double manager_pay).
⊡ Ⓕ void payout(double reg_emp_rate, double reg_emp_hrs).
⊡ Ⓕ void payout(double monthly_salary, double commission, double units).
⊡ Ⓒ class regular_emp.
⊡ Ⓒ class sales_mgr;
⊡ Ⓒ class sales_person;
⊡ Ⓕ void title().
```

```
Config File F:\cppbetaW\samples\payroll\payroll.icc
Ext File f:\cppbetaw\ETC\SYSTEM.ICE
Ⓢ F:\cppbetaW\samples\payroll\payclass.hpp
Ⓢ F:\cppbetaW\samples\payroll\payfunc.hpp
Ⓢ F:\cppbetaW\samples\payroll\payroll.cpp
```

```
F:\cppbetaW\samples\payroll\payclass.hpp - Definition
 Row 8        Column 7     Insert
  -----+-█--1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----10----+----1
  // Abstract base class definition
  //
class employee            // Declare a virtual base
                          // class named employee
```

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

When you first open the IDE, you will see the Overview page in the Project section by default. If you are not there now, click the Project tab and the Overview button to get there.

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 right pane now.

Explore the Panes

There are three down-arrows (▼) across the top of each pane. You now have the focus in the upper right 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 right pane is 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 right pane you are seeing a Source Files view of the Payroll project 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 left pane, which shows a Declarations view of the Payroll project object. This view shows a list of all the declarations in the project. In this view the main function will be highlighted (selected) by default, if there is a main function. Otherwise, the first declaration is selected.

With your mouse, 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 upper right 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 or more source files (*.cpp, *.hpp, *.c, *.h, etc.)
These are the files you provide or create. Source files can contain more than C++.

one or more configuration files (*.icc)
.icc stands for Incremental C++ Configuration. A configuration file contains all the information about how the source files are processed. It is like an encapsulation of the project: 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 project.You do not have to create the configuration file, but you can edit it.

a codestore (*.ics)
The codestore is a database that contains the full information about a project. It is created the first time you build your project. 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.

Creating a Project

SmartGuides will step you through the process of creating a project.

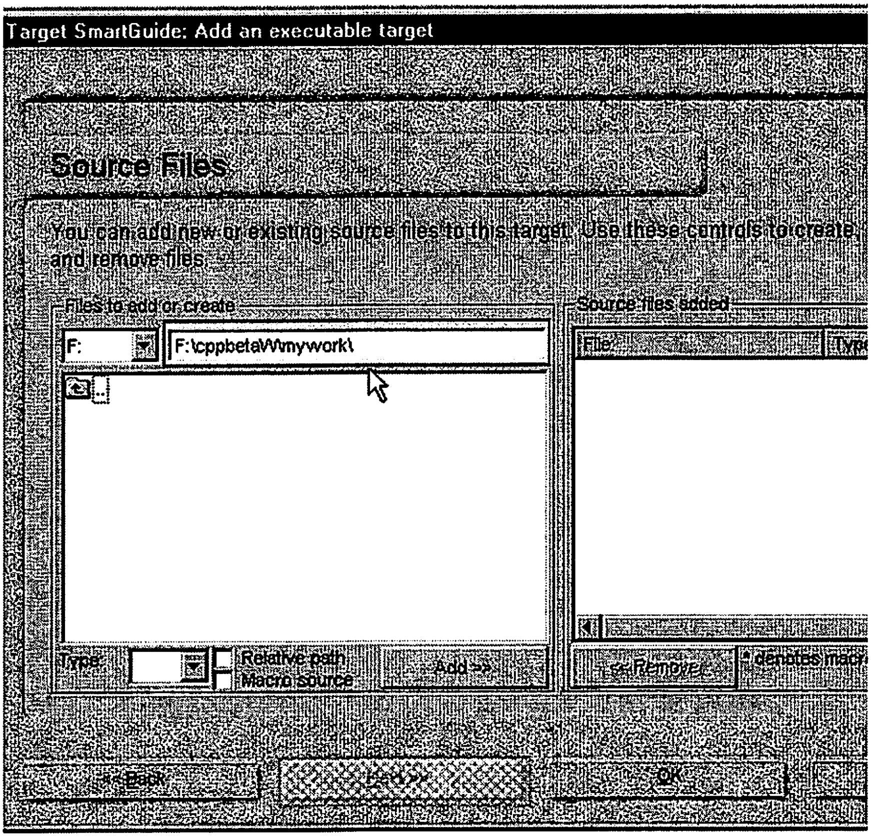
To create a project:

1. First, close the sample project. Pull down the **Project Workbook** menu and select **Close Project**.
2. From the **Project Workbook** menu, select **Create Project**. The Project SmartGuide opens.
3. Click **Next** until the Project Configuration page is displayed. First, you are prompted to create the configuration file. This is a plan for the new project. 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 the **Browse** button 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**:



Target SmartGuide: Add an executable target

Source Files

You can add new or existing source files to this target. Use these controls to create, and remove files

F:\cppbeta\Wmywork\

Source files can come from any directory.

10. Type a source file name, such as `helloworld`. 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: you will be prompted to confirm the name, and the SmartGuide will create the file for you. 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**. The Open Project 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 project. Since you haven't added any code yet, click **No**. The IDE display has now added two new tabs: a **Project** tab and a **Configuration** tab. The panes are mostly blank because no code has been entered and compiled yet.

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

Looking at Your Project

As you went through the steps to create your project, 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 project.

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

Look at the Project Overview

Make sure that you are in the Project section, with the Overview page selected. Click in the whitespace in the upper left pane. The project is identified by name, and the pane shows the Declarations view. There are no declarations yet.

In the upper right pane, your configuration file (helloworld.icc) is listed. This is the configuration file you named when you created the project using 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 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 left pane of this page, you can see:

- T a *target directive*, or a statement to specify the file that will be produced when you build, and
-

Ss a *source specification*, 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 Ss

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, 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 Project Options page by clicking the **Project Options** page button (also in the Configuration section).

The Project 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 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 **Drives** and **Folders** fields 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 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 project files are saved when you build your project.

Building 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 refresh 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 messages 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,

int main();

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 your 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:

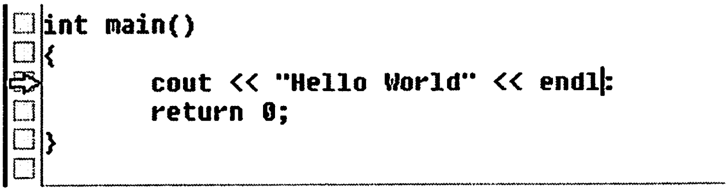
1. The status line at the bottom of the screen displays a message:

   ```
   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:



```
int main()
{
        cout << "Hello World" << endl:
        return 0;
}
```

Place the cursor into this line and replace the semicolon.

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

Running 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, 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

This portion of the tour does not introduce any new tasks in the IDE, but contains more details on the following topics. For some of these 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 15.

- "About Incremental Compilation" on page 27
- "About Editing Source Files" on page 28
- "Searching a Project" on page 29
- "Using the Search Page" on page 30 (more advanced searches)
- "Object-based Searching with the Find Uses Page" on page 31
- "Configuration Files" on page 31
- "A Closer Look at the Configuration Section" on page 32
- "Setting Build Options" on page 34
- "Symbols used in the IDE" on page 34
- "Linking between Panes" on page 35

- "Some Useful Shortcut Keys" on page 36
- "Toolbar Buttons" on page 37
- "Menu Descriptions" on page 39

More detailed information is also available in the online documentation. Some suggested references you can search for in the online help:

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

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.

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 use the command-line interface to VisualAge C++, you still get full incrementality, but you must perform debugging through the IDE.

About Editing Source Files

The IDE maintains a single codestore for every project.

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. Syntax 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
- c
- rc
- lib
- vbf
- vbe
- dax
- mak
- ipf
- loc
- msg
- sqc

- sqx
- WIN hpj
- WIN mc
- OS/2 mkmsgf
- OS/2 msgbind
- OS/2 res

Searching a Project

There are three ways to search a project:

1. **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.
2. The **Search** page searches within any object, from a single class to the entire project.
3. 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 **Project** 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 C 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 `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+P** 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 pop-up menu.

Using the Search Page

The **Search** page button is in the **Project** section of the workbook.

By default, there are two panes: a Search view and a Source view. You can search all or part of your project for a pattern that you specify.

To search:

1. We will search for the string 'class'. Type `class` in the entry field:
2. Start the search by clicking the flashlight icon at the end of the entry field (🔦) or by hitting **Enter**. The icon then turns into a stop sign, which you can use to stop the search. Within one or two seconds, all lines containing a match for your pattern are listed.

Each numbered line returned in the top pane is a file location. The number in brackets is the line number in the source file.

The bottom pane is a Source view, which automatically displays the line selected in the top pane. As you can see, the search has returned all 92 occurrences of the string 'class', including occurrences where class is only part of a larger term, such as payclass, and all the different classes that have been defined in the project.

Clearly, this is not the most efficient way to find the class itself, but it is an exhaustive search that is useful if you want to include comments and variations on a string in your search. To find out how to search for the object, without including variations and comments, see "Object-based Searching with the Find Uses Page" on page 31.

Search options can be set by clicking the plus symbol next to the flashlight icon.

- The **Case-Sensitive** option restricts your search to exact case matches.
- The **Show Match** option shows matches only, instead of all lines.
- When the **Find All** option is selected, every match in each line is highlighted instead of only the first match in a line.

To repeat a previous search:

1. Click the drop-down arrow next to the flashlight icon. The drop-down menu contains previous patterns that you have searched.
2. Select a pattern to rerun a previous search.

Object-based Searching with the Find Uses Page

The Find Uses page is an even more compact and powerful method of searching than the Live Find and the Search page. Find Uses allows you to search by object, not just by text string.

Try a Find Uses Search

With the payroll sample project open, click the **Find Uses** button in the Project section.

The Find Uses page shows three panes:

- a Declarations view of the project (a list of all declarations in the project) in the upper left pane
- the Find Uses view of the declaration object selected in the left-hand pane
- the Source view, in the bottom pane.

In the Declarations view, select the first object (class employee). Two things happen:

1. The right-hand pane is updated with a report on the uses of the employee class (number and location). The number of uses found (4) includes the definition of employee class.
2. The bottom pane updates the source view to the first location in the source where class employee appears. If you select another occurrence from the reported list, the source view updates again.

Configuration Files

A VisualAge C++ project must have a configuration file.

When you build a project, 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++ creates 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.

A VisualAge C++ project can have more than one configuration file. You must have multiple configurations to build the following targets:

- More than one executable from the same set of source files
- A shared library and a static library from the same set of source files
- Different versions of the same executable, built with different options

For example, a project might have a debug configuration with debug options, as well as a production configuration with optimization options. (There will still be 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.

A Closer Look at the Configuration Section

The Configuration section appears only once you have opened a project.

To see the configuration file, click the **Configuration** tab:

There are many different ways you can look at the configuration. All of these can be customized to display the views you use most often. The pages already provided are:

There are two panes on the Options page: the Source and Targets view and the Change Options view. In the Change Options view you can set options for the files listed in the Source and Targets view.

The Source and Groups page shows three panes. In the Source Groups view, the components of your project are grouped together by file type. You can change the groupings of your files in the Change Source Group view, and you can edit the source files in the Source view.

The Targets page shows three panes: the Targets view, the Change Targets view, and the Source view.

In the Targets view, your project's source files are grouped according to the target or targets they build. You can move source files to different targets in the Targets view. To change the name or type of a target, use the Change Targets view.

The Project Options page allows you to choose options that will apply to the entire project, for example, whether optimization is on or off, and which type of processor you intend to run your application on.

The Options Groups page shows two panes: the Options Groups view and a Change Options Group view.

The Options Groups view shows the options you have assigned to groups. The options can be changed in the Change Options Group view.

The Advanced page offers two unique and powerful views of the configuration file: the Details view and the Interpreted view. The Details view presents the entire configuration file as structure. In Source view of the configuration file, you can only edit at the line level. The Details view allows you to perform many more object-level actions on the file. The Interpreted view shows you how the configuration file has been processed, for example, what path was taken through the source, and what values have been assigned to variables as a result. This is especially helpful when you need to debug your program.

The Source page is a full-screen Source view of the configuration file, in which you can edit the configuration file directly.

Together, the information in the pages in the Configuration section forms a complete description of the project. Everything you want to know about a project can be found by looking at the configuration file through one or another of these pages.

Setting Build Options

Build options are a part of the 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 project, 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 apply 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.

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

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:

- W Workbook
- Host Host
- Project Project
- Configuration Configuration

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

- Bp Breakpoint
- C Class
- F Function
- V Variable

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 descriptors that can be used 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.

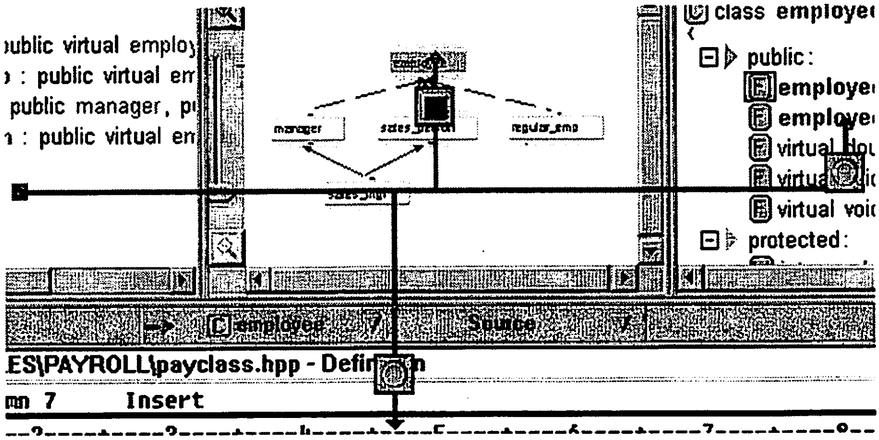
Linking between Panes

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:



```
ublic virtual employ
 : public virtual em
public manager, pu
 : public virtual em
```

```
class employe
{
⊟ ▷ public:
    employe
    employe
    virtual dou
    virtual ic
    virtual voi
⊟ ▷ protected:
```

```
ES\PAYROLL\payclass.hpp - Definition
mn 7      Insert
--2----+----3----+----4----+----5----+----6----+----7----+----8--
```

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.

Some Useful Shortcut Keys

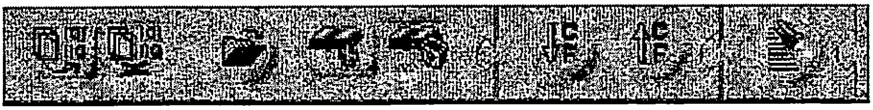
This section lists keys used to perform and manage Source view and the Editor window operations. Where two key names are joined by a plus sign (+), hold down the first key and press the second, or hold down the first two and press the third.

For a complete list of key commands you can use in the IDE, see **IDE Shortcut Keys** in the online help.

Navigating in the IDE	
Move between panes	F6
Exit Live Find	Esc
Commands	
Begin a build	Ctrl+Shift+B
Editor Shortcut Keys	
F9	Switch to the Command shell window; which gives you access to the command line prompt. (Note: the cursor must be located within the working area of the Source view.)
Esc	Moves the cursor to the command line.
Ctrl+Right	Move cursor to start of next word.
Crtl+Left	Move cursor to beginning of word or previous word.
Ctrl+C	Copy selected text to clipboard.
Ctrl+X	Cut selected text to the clipboard.
Ctrl+V	Paste text from the clipboard.
Ctrl+T	Select word/token.
Ctrl+Delete	Delete to end of line.
Ctrl+Backspace	Delete entire line.

Toolbar Buttons

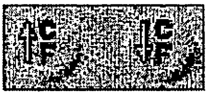
The IDE toolbar contains icons for frequently used actions. The default selection of icons is:



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.

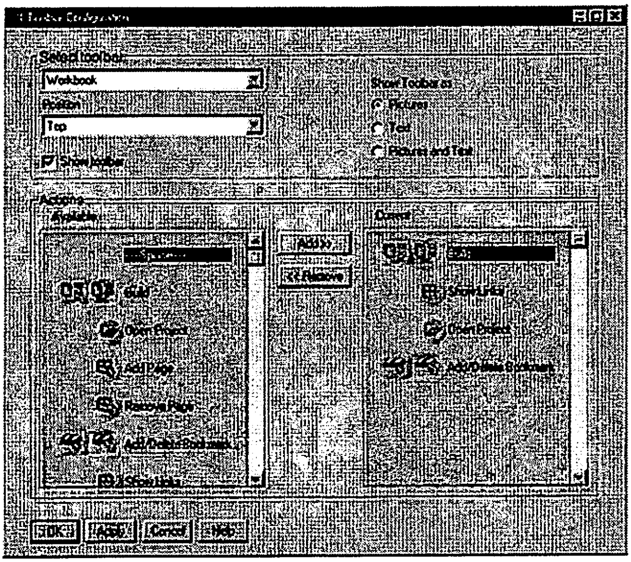
The last button starts a Live Find. It will open the Live Find window for the pane currently in focus.

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).



Toolbar Configuration

Select toolbar: Workbook

Position: Top

☑ Show toolbar

Show Toolbar as

● Pictures

○ Text

○ Pictures and Text

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++.

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 3. Try VisualAge C++

Try VisualAge C++ contains exercises you can use to gain a thorough understanding of the VisualAge C++ Integrated Development Environment (IDE).

Try VisualAge C++ is divided into two independent sections. In one, you will develop a web-based review tool. This web-based tool allows a group of reviewers to add comments to a set of HTML documents. This part covers the VisualAge C++ IDE in depth, and takes four to eight hours to complete, depending on which sections you complete, and how much experimenting you do with the IDE. It is structured so you can start or stop at the beginning of any part. If you start in the middle or skip a part, you may need to complete some simple prerequisites. The prerequisites are described at the beginning of each part.

In the other, you will develop a graphical user interface, using the Visual Builder, for administering the users of the review tool. The interface lets you administer user information for the web-based review tool. This section should take you three to six hours to complete. It does not explore IDE actions such as debugging, viewing class hierarchies, or navigating, although it does involve some use of the IDE. If you choose to complete this section of the exercise without doing the IDE section, you should at least do the Tour of the IDE.

Although the sections are related, neither is dependent on the other, so you can try just one, or both. However, if you are interested in having a useful web-based HTML review tool, you may want to complete both parts, then enhance the code to your liking to provide additional capabilities.

You can now proceed to either of the two main sections:

- "Develop a Web-Based Review Tool"
- "Develop a Graphical User Interface from a Visual Part" on page 103

Develop a Web-Based Review Tool

In this section, you will implement a web-based application that uses the Common Gateway Interface (CGI). The application interacts with users through the users' web browsers, although it runs on your own machine. This application allows reviewers to add comments to a set of HTML documents.

Here is an overview of this section:

Background Information
Before you start developing the review tool, you may be interested in some background information on how the tool will work, and on how CGI applications communicate with a web browser. Only read this information if you intend to do one of the following tasks:

- Develop other CGI applications.
- Add your own enhancements to the review tool.
- Use the tool in a production environment.

Part 1. Configuring a New Application
This section shows you how to create a simple web-based Hello-world application. You will use a series of SmartGuides to provide the VisualAge C++ IDE with information it uses to create a configuration file for your project. Then you will add a main function to your project. Finally, you will run the executable file that VisualAge C++ generated for your project.

Part 2. Modifying Configuration Options
This section shows you how to make simple changes in the options for your project's configuration. You will change the target CPU architecture to match that of CPU you are using.

Part 3. Declaring and Implementing a Class
In this section, you will create a Request class, which is a utility class for holding information about an incoming CGI request. You will learn how to create new source files in a project, add source code to them, and perform incremental builds of your project.

Part 4. Developing Classes In the IDE
In this section you will implement a hierarchy of user classes in which the base class, User, defines virtual methods for all available operations. You will learn how to use the Classes and Class Hierarchy pages of the Project section of the IDE, and you will verify that the classes and their members are properly organized so that access to restricted functions is limited to authorized users.

Part 5. Debugging and Revising Your Application
In this section you will add a method to your application, then use debugging features of the IDE to locate and correct a bug in the new method. You will learn how to set breakpoints, view variable contents, step through code, and perform other debugging tasks.

Part 6. Managing Configuration Files
In this section, you will add user variables to your project's configuration to customize how your project is built, and you will use the IDE to manage the settings and effects of those variables. You will learn how flexible and extensible the VisualAge C++ configuration language is.

Part 7. Optimizing Your Configuration
In this section, you will use the Configuration Optimizer, which analyzes your configuration and optimizes it to improve build performance.

Part 8. Defining the View Function for Reviewers
In this section you will flush out another member function. The section contains mainly source code. It offers you little additional guidance. Complete this and the following section if you want more practice in the IDE before you move on to your own development work, or if you intend to put the web-based review tool to use.

Part 9. Defining the Remaining User Functions

This section contains the source code you will need to complete the review tool. It advances the same learning objectives as Part 8.

Background Information

This section provides some background information about the exercise application and CGIs in general. It is divided into the following topics:

- How the Review Tool Will Work
- How CGI Applications Communicate with a Web Browser
- Running the CGI with a Web Server Other Than the VisualAge Help Server
- Determining Which Library Files Are Needed for a CGI

These topics may be helpful, but you do not need to read them to do the exercise. If you do not want to read them, proceed to "Part 1. Configuring a New Application" on page 46 .

How the Review Tool Will Work

The review tool reads HTML documents stored on your computer, and displays them in the Web browsers of users accessing the tool. Users access the tool using a unique key value that identifies them. The tool implements different levels of authority for different users based on their keys:

- A user with ***user*** authority cannot view your documents. If a key is unrecognized, the user's access defaults to this level and the user is prevented from viewing documents.
- A user with ***reader*** authority can view the HTML documents and navigate within them.
- A user with ***reviewer*** authority can do everything a reader can do, and can add comments to a document.
- A user with ***author*** authority can do everything a reviewer can do, and can mark comments as completed.

This organization lends itself well to an inheritance structure in which a user class with more authority inherits from one with less authority, and overrides certain member functions of its base classes. When a document is requested, the tool generates a view of the document based on the user's authority level:

- A ***user*** sees an error message stating that they are not authorized to view the document.
- A ***reader*** sees a document that looks identical to the original HTML document. The only difference is that hypertext links in the document are changed into calls to the CGI, so that when a link is followed the CGI still maintains control of the user's access.
- A ***reviewer*** sees a document with numbered markers inserted at each paragraph, preformatted text block, or list item. These numbered markers are hypertext links that

allow the reviewer to create comments at these locations. A reviewer also sees all comments already added to a document. Each existing comment also has a hypertext link that allows the reviewer to append another comment to it.

- An ***author*** sees a document similar to what a reviewer sees, with an additional hypertext link on each comment. This link allows the author to mark a comment as completed, meaning that the comment has been answered in some way.

The files being reviewed do not need to be stored in a publicly accessible location. Normally, HTML documents need to be stored in a web server directory for users to view them. The tool acts as an intermediary between your documents and web users. If you want, you can modify the tool later to provide access control to sensitive documents on your system.

You will need a web server on your workstation to use the tool. The VisualAge Help Server provided with VisualAge C++ is sufficient for the purpose of trying out the exercise, but should not be used for deploying a production-level web application. If you are using Windows NT, you can use Microsoft Peer Web Services instead, although additional setup may be required.

How CGI Applications Communicate with a Web Browser

CGI applications are short-lived: each time a user accesses the application from a browser, the application is loaded as a separate process, reads input, provides output, and terminates.

A CGI application receives requests from a user's browser in one of two ways: through an environment variable, or through the standard input device. Requests from an environment variable are called GET requests; requests through standard input are called POST requests. Normally, all requests are GET requests except for those submitted through an HTML form (such as the form for adding reviewers' comments).

When a request is sent to the server, the REQUEST_METHOD environment variable is set. Each time the CGI is invoked, it reads the REQUEST_METHOD environment variable to find out how the request was made. If the request method was GET, the CGI then reads the QUERY_STRING environment variable for the text of the request. If the request method was POST, the CGI determines the number of characters to read by reading the CONTENT_LENGTH environment variable, then reads that number of bytes of input from the standard input device.

The CGI sends output back to the user's browser by writing to standard output. First, the CGI writes a header to indicate what type of content is being returned (for example, plain text, HTML text, or a JPEG image), then the actual output is written.

Running the CGI with a Web Server Other Than the VisualAge Help Server

This manual assumes that you are using the VisualAge Help Server as your web server. If you choose to use a different web server, you will need to change your project configuration to reflect the path the web server uses to run CGIs; wherever the VisualAge Help Server directory is referred to, change this directory to your web server's CGI directory. You will also need to do some additional setup work to get the CGI working properly with dynamic link libraries (Windows NT, OS/2) or shared libraries (AIX) as explained below.

Normally, a CGI does not have access to environment variables other than those set by the web server. This means that any dynamic link libraries (OS/2 or Windows) or shared libraries (AIX) that are normally located through an environment variable, such as PATH or LIBPATH, are not accessible to the CGI. To make the CGI run under most web servers you must therefore either link the run-time libraries statically to the CGI, or else copy the necessary library files into the same directory as the CGI.

Static linking during development of a CGI increases build times significantly, because VisualAge C++ must locate the necessary code in the run-time libraries and link it statically to your application every time you build. You should use dynamic linking as you develop the CGI, and ensure that the CGI has access to the necessary run-time library files, either by copying them into the CGI directory, or by using the VisualAge Help Server as your web server during development. This help server preserves environment variables such as the PATH environment variable within the CGI environment.

If you are running Windows NT and you want to use Microsoft Peer Web Services instead of the VisualAge Help Server, copy the following dynamic link libraries from the VisualAge directories into the directory your CGI will run from. The CGI usually runs from X: \inetpub\scripts , where *X:* is a writable drive on your system. The DLLs to copy are in either the run-time or bin directories under the main VisualAge C++ directory. They should be copied into the CGI directory, rather than into a bin or run-time directory under the CGI directory:

- `runtime\cppobi36.dll`
- `bin\cppzm40i.dll`
- `bin\cpprmi40.dll`
- `bin\cpprbi40.dll`
- `runtime\cppoui36.dll`
- `runtime\cppogi36.dll`

Determining Which Library Files Are Needed for Any CGI

On any platform, for any CGI, you can determine what library files need to be copied to the CGI directory by following these steps:

1. Link the run-time libraries to the CGI dynamically, and place the target in the CGI directory.
2. For OS/2 only, make a backup of your CONFIG.SYS file, then comment out any setting of the LIBPATH environment variable in that file. Save CONFIG.SYS and reboot.
3. Open a command prompt or shell session and change to the CGI directory.

4. Change the PATH environment variable to ".". On OS/2, Windows, or C-Shell on AIX, type:

   ```
   SET PATH=.
   ```

   On AIX Bourne-Shell or Korn-Shell, type:

   ```
   EXPORT PATH=.
   ```

5. Type the CGI executable file name. As the CGI tries to load, the system will look for the necessary library files in the current directory only, because you have overridden the search paths normally used to find DLLs or shared libraries. At the first DLL or shared library the system cannot find, it displays an error message showing the name of that DLL.
6. Locate this DLL or shared library on your system and copy it to the CGI directory. Repeat steps 5 and 6 until you do not get any error message about missing library files.
7. For OS/2, restore the backup copy of CONFIG.SYS and reboot.

You can now proceed to "Part 1. Configuring a New Application".

Part 1. Configuring a New Application

Before you begin writing code, you need to provide some information so that VisualAge C++ can configure your program's build environment properly. In this section, you will use the VisualAge C++ Integrated Development Environment (IDE) to do the following:

- Create a *project* for your program. A VisualAge C++ project groups together the source files and actions needed to produce an application or part of an application. To work with your application in the IDE, it must be defined as a project.
- Add a target file, source files, and class library files to your project.
- Create a source file with a simple main function.
- Build and run your program.

This section should take you approximately 30 minutes to complete. You will learn to perform the following IDE tasks:

- Use the SmartGuides that lead you through creating the configuration for your project.
- Perform an initial build of your project.
- Find error information after an unsuccessful build.
- Open a view of an existing source file.
- Add a main function to your project.
- Rebuild your project.

If you are interested only in learning the IDE, and do not need to understand the details of the CGI application, you can skip passages titled *Implementation Details.*

Preparation for this Tutorial

To run the web-based portion of the tutorial, you will need to follow these steps for your application to work through a web browser. If you do not have root access to the machine, ask your system administrator to perform step 1 for you:

1. Create a symbolic link from `review.exe` in the directory from which your web server runs CGIs (/var/docsearch/cgi-bin if you are using the web server that is installed with VisualAge C++) to `review.exe` in a directory you have write access to. The file linked to does not have to exist; it will be created as you go through the exercise.

   Note: If you are one of several users on the system and another user has already created a symbolic link from /var/docsearch/cgi-bin/review.exe, you should replace `review.exe` with some other name (e.g. `review.cgi`, or `myreview.exe`) in the CGI directory. If you do this however, you will not be able to use some of the links in this exercise to test your code, because the links will point to the wrong file name.
2. Store `review.exe` in the directory you have write access to, rather than in `/var/docsearch/cgi-bin/`.
3. Change the permissions on the directory where you will be storing `review.exe` to be readable by all, so that the server process can read that directory and its contents. (Use **chmod 755 dirname** to do this.) Subdirectories you create in the exercise that are used for storing data (e.g. user information comments), should be writable by all, so that the server process running the CGI can write the necessary data. (Use **chmod 777 dirname** to make the directory **dirname** writable by all.)
4. After the first successful build, change the permissions on `review.exe` to be readable and executable by all, so that the server process can read it. (Use **chmod 755 review.exe** to do this.)

Create a Project and Add Files to It

Follow these steps to create your project:

1. Start the VisualAge C++ IDE.
2. Choose to create a new project in one of the following two ways:
   - From the VisualAge C++ startup screen, select the **Create a new project** radio button and click **OK**.
   - If no startup screen appears, or if you cancelled the startup screen, select **Project Workbook - Create Project** from the menu bar.
3. The Project SmartGuide will guide you through creating the project.
4. In the Project Configuration screen of the SmartGuide, specify `review.icc` as the project name. If you do not enter the file-name extension .icc, the SmartGuide will add it for you.
5. Specify a directory in which to store the project files. For example, if you are running OS/2 or Windows NT and you want to store the project files on drive F:, you might enter `F:\vatutor` as the directory. On AIX, you might enter `/u/myuserid/vatutor`. If the directory you specify does not exist, it will be created for you.

For the remainder of this exercise, wherever you see the path F:\vatutor, you can substitute the path you specify in this step.

6. Click **Next**. The Target Type screen appears. On this screen you add targets to your project. Make sure **Executable (EXE) file** is the current selection in the New target field, and click **Add target**.
7. The Target SmartGuide appears. The Target SmartGuide guides you through defining your target. Click the **Next** button until you get to the Target Name screen.
8. On the Target Name screen, enter the name and destination directory for your target. For the CGI example, name your target `review.exe` and specify the target directory as the directory from which CGIs run on your web server.
   - If you are using the VisualAge Help Server to try out the exercise, this directory is on the same drive where VisualAge was installed:
     - WIN \IMNNQ_NT
     - OS/2 :\NETQOS2
     - AIX the writable directory you specified in preparation for this Tutorial.
   - If you are using Peer Web Services on Windows NT, this directory is normally \inetpub\scripts.
9. You can also select a program type for your application, which predefines certain compiler and linker options. In the **Program Type** column, select **WIN IOC, OS/2 IOC**, or **AIX IOC** depending on your operating system. These two choices set project options so that your project can use IBM Open Class Library classes and member functions.
10. Click **Next** to go to the Source Files screen.
11. Enter the source file name `main.cpp` in the **Files to Add or Create** text entry field (remove any current path information from that field). Then click **Add**.
12. The SmartGuide asks if you want the file to be created. Click **Yes**. An empty main.cpp file is created.
13. Click **OK** to close the Target SmartGuide and return to the Project SmartGuide.
14. You will see the target you just defined in the list at the bottom of the Target Type screen. The last Project SmartGuide screen lets you specify help files. Since you will not add help files for this exercise you can close the Project SmartGuide by clicking the **Finish** button.
15. When the IDE asks if you want to open the project, click **Yes**.
16. When the IDE asks if you want to build the project, click **Yes**.

You will encounter a build error, because you have not yet coded a main function for your program. A Help Tips error dialog will appear if you have not encountered a build error before. If the dialog appears, click **OK** to go to the Messages pane. Building the empty project now simply adds the sources and targets you specified to the codestore, so that you can work with them through the IDE.

Add a Simple main Function to main.cpp

1. Make sure that the Project section of the workbook is selected.
2. Select the Source Files page.
3. Select the S main.cpp object for the main.cpp source file in the Source Files view in the upper left pane. Notice that when you select this file, the source view underneath changes to display the content of main.cpp, which is currently empty.
4. Click anywhere inside the Source view. Enter the following code for main.cpp into it. If you are viewing this information online, copy the code from your browser window and paste it into the Source view.

   Note that all comments shown in this and other source code in this manual are for your information; you do not have to type them in.

   ```
   #include <fstream.h> // for use of cout statement (and file i/o later)
   int main() {
      cout << "Content-Type: text/plain\n\nHello world!" << endl;
      return 0;
   }
   ```

Implementation Details: The output begins with a header (`Content-Type: text/plain\n\n`), which is the CGI header that instructs a user's browser to display the subsequent text as plain unformatted text. This header will display when you run the program from a command shell, but will not display when you start the program from a browser.

Your project is now ready to build.

Build and Run Your Program

Now that you have added content to your source file, you can build and run the project.

1. To start the build, click the **Build** button ( ) on the toolbar or press **Ctrl+Shift+B**. The IDE automatically saves any files you have opened and added to the project, then starts building the project.
2. If any errors occurred during the build, the Messages page appears. (If it does not appear, but the status area indicates build errors, go to the Messages page in the Project section.) Select the error in the Messages pane; the view below this pane shows the source code for the object containing the error, with the cursor on the line containing the error. Correct any source code errors, then build again.
3. You can run your project from a command shell or from a web browser on your workstation:
   - In the command shell, change to the directory you specified as the target directory (the directory from which your CGIs run) and type the executable file name, review.exe.
   - From a browser, assuming you are using the VisualAge Help Server as your web server, enter the following location:

     http://localhost:49213/cgi-bin/review.exe

If there is no response, the help server may not be running. Select any of the choices from the **Help** menu in the IDE to start the VisualAge Help Server. Once the help server is running you can try the appropriate link above.

4. If you ran the program from the command shell, you will see:

   ```
   Content-Type: text/plain
   Hello world!
   ```

   If you ran the program from a web browser, the browser interprets the Content-Type header, and displays only the text `Hello world!`

You can now proceed to "Part 2. Modifying Configuration Options".

Part 2. Modifying Configuration Options

In "Part 1. Configuring a New Application" on page 46, you used the SmartGuides to create a simple project. You now have a configuration file for your project and one source file, main.cpp. In this part you will learn how to set or modify options for your project.

Because the SmartGuides did such a good job of helping you create your project, all the necessary options were set for you. The option you will change in this part will not have a great impact on this project. However, changing it will illustrate how easily you can change options for your project, target, or source files from within the VisualAge C++ IDE.

This section should take you approximately 15 minutes to complete. In it, you will learn to perform the following IDE tasks:

- View the options for your project's target and navigate the options hierarchy
- Use the Live Find feature to locate text
- Determine whether an option is the default or has been overridden
- Change the setting of an option
- View the source for your configuration file

Change Your Target CPU Architecture Option

By default, the target CPU architecture is a generic PowerPC processor. In all likelihood you know which specific processor you will be using, such as a Power PC or Power 2 processor. By changing the target CPU option, you allow VisualAge C++to generate an executable file containing instructions that are targeted to your processor. In a real application, making this change might improve performance, although in this exercise you will not notice any difference.

You can add, modify, or delete configuration options for your project in the Configuration section of the workbook. Follow these steps to specify your processor:

1. Select the Configuration section of the workbook.
2. Select the Options page of the Configuration section.

3. The left-hand pane shows a Source and Targets view of the configuration, while the right-hand pane shows a Change Options view. Any options you change in the Change Options view are applied to the object selected in the Source and Targets view. Select the Target object in the Source and Targets view.
4. Press F6 to move to the Change Options view.
5. There are two ways you can locate the option you want to change.
   - You could expand various entries in this view until you find the **Target CPU architecture for instruction set selection** option.
   - You can open a Live Find field by pressing **Ctrl+F**. In the **Change Options** view, the Live Find field is already displayed at the bottom of the pane, with the Live Find icon to its left. Click the entry field, and slowly type `Pow` . Notice that as you typed, the view changes to display the **Target CPU architecture...** option, and the first choice is displayed. You may also see that the light turns yellow on the flashlight icon as soon as your entry has a match. For more information on using this feature, see Search a View Using Live Find in the online help.
6. Notice that the current choice, **Common Power/Power PC/Power 2 processors**, is selected with a grayed radio button. This indicates that the grayed choice is the default and that the default is being used; no information for this option appears in your configuration. If you click this choice, the radio button changes to black, to indicate that this option is now explicitly set to the default choice.
7. Select the processor you are using. The text of the option you chose turns green. An option's text is displayed in green when its value, for the currently selected source or target, is set by your configuration. In other words, you have explicitly chosen a setting for this option, whether or not that setting is the default.
8. Click **Apply** at the bottom of the view. This saves any option changes you have made. The next time you build your project, these changes will take effect.
9. Select the **Source** page of your configuration file in the Configuration section (the rightmost button along the top of the Configuration section). You will see that gen(arch,"ppc") or (another processor type you choose) is now one of the target options for your project.

You can now proceed to "Part 3. Declaring and Implementing a Class".

Part 3. Declaring and Implementing a Class

In this section, you will create a Request class, which is a utility class for holding information about an incoming CGI request. You will modify the main function by replacing its current contents with a declaration of a Request object, and an output statement that prints a simple response to the request back to standard output.

In later sections, the Request object will determine the class of user accessing the tool (for example, Reviewer or Author), and will create an object of the appropriate user class, so that the object can perform actions based on the authorities assigned to that class.

This section should take you approximately 90 minutes to complete. In it you will learn to perform the following IDE tasks:

- Create a new file and associate it with your project
- Create a class
- Use the Classes view
- Make macros globally visible to your project
- Change a project option using the change options view
- Obtain various editable views of your source
- Use various editor features
- Locate a function in your project source using different techniques
- Maximize one view within a page to take up the entire page

You can cut sections of code from the online tutorial and paste them into the appropriate source views; this will let you concentrate on the concepts, and will save you time and potential typing errors.

Implementation Details

At the heart of the Request class is its constructor. The Request constructor does the following:

- Determines the type of request (GET or POST) for this and other CGI-specific information, see "How CGI Applications Communicate with a Web Browser" on page 44)
- Reads the request from the appropriate source
- Parses out certain fields in the request that will be common to most requests. For example, each CGI request should contain a key attribute (for example, key=abcde) which is used to uniquely identify each user by key. Each request should also contain a file attribute to identify which file to display.

Prerequisites If You Are Starting Here

You should have the following source files in a directory on your system:

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function

Start the VisualAge C++ IDE. Open the project identified by review.icc, then build the project when prompted.

Add the Request Class Declaration to the Project

You need to add the declaration for the Request class to the project. Follow these steps:

1. Create a file to contain the declaration. From the Project Workbook menu, select **Open or Create File**, or just press **Ctrl+O**.
2. In the Open File dialog, type the file name `request.hpp` in the File name entry field. Change the current directory in this dialog to the directory you defined earlier

as the directory in which to store the .icc file. The main.cpp file should already be listed in the directory along with other files of the form *review.**.

3. Under the **File Open Location** group box make sure that **Open as a Workbook Section** is selected. This will add a separate workbook section for this source file, to make it easy to access.
4. You also want this source file to be included in the current project and to be associated with the source directive containing main.cpp. In the **Project Options** box, select the **Add to Project** check box and select **Add Source to Source Directive**. This will add the new file to the main.cpp source directive.
5. Click **Open**.
6. An **Add to source** dialog appears. This lets you specify the source directive to which the source file will be added. Because your project currently has only one source directive (for main.cpp), you can simply click **Apply**.
7. The IDE creates a new workbook section that contains a source view of request.hpp, which is currently empty because you have not entered any code.
8. Declare the Request class in the new file as shown in the following example. Type the code into the Source view of request.hpp.

   ```
   #include <istring.hpp>       //Required for use of IString class
   // Class Definition
   class Request {
      IString ReqString,        //Request string from cgi call
              lReqString,       //Lowercase version for searches
              ReqMethod,        //How request was obtained
              Op,               //Operation requested
              Filename,         //File requested
              Filepath,         //File, with slashes corrected
              Key;              //User's key value
      int ReqLen;               //Length of request string
   public:
      //Constructor and get methods
      Request();
      IString GetReqString() {return ReqString; }
      IString GetOp()        {return Op;}
      IString GetFilename()  {return Filename;}
      IString GetFilepath()  {return Filepath;}
      IString GetKey()       {return Key;}
      int GetReqLen()        {return ReqLen;};
      //Get the value of an attribute=value pair from request string
      IString GetValue(IString Attribute);
   };
   ```

9. Build the application to ensure that you did not make any typing errors.
10. If you encounter an *error*, go to the Messages page in the Project section to read the message. Check that you entered the code exactly as shown above and correct any errors.

    If you encounter a *warning* but the status line at the bottom of the IDE window indicates the build completed successfully, you do not have to make any changes.

To see the result of the build, select the Classes page in the Project section. The Request class should appear in the upper left pane (a Classes view of the Project object), with its members in the right pane (a Members view of the selected Class object). Class members are grouped by access method and sorted alphabetically by name. As you select different class members in the right pane, the Source view underneath shows a yellow arrow beside the declaration of the member you select. For example, select the Request class constructor; its declaration appears in the Source view. Later, when you add definitions for some of the undefined functions, the Source view will change to show these definitions.

Add the Constructor for the Request Class

Because the Request class constructor does more than simply initialize its data members with default values, it is not defined inline in the class declaration. Instead, it is defined in a function definitions file, request.cpp.

To create request.cpp, follow the first four steps of the previous section (Add the Request Class Declaration to the Project). Add request.cpp to the Source directive and open it as a workbook section, just as you did for request.hpp. Then, in the Source view that opens for request.cpp, enter the following code:

```
Request::Request() {
   ReqString="";                             //Initialize string
   ReqMethod=getenv("REQUEST_METHOD");       //Determine how input provided
   if (ReqMethod=="GET") {                   //GET means it comes from
      ReqString=getenv("QUERY_STRING");      //QUERY_STRING variable
      ReqLen=ReqString.length();
   }
   else if (ReqMethod=="POST") {             //POST means read the right number
      ReqLen=atoi(getenv("CONTENT_LENGTH")); //of characters from standard input.
      for (int c=1;c<=ReqLen;c++)
         ReqString+=char(cin.get());
      ReqString.change("&"," ");             //In POST calls, an "&" separates fields
   }
   ReqString.change("+"," ");                //In both types, a "+" replaces a space
   //Special characters are escaped with % and two hex digits.
   //Convert these to their actual character values
   int p=ReqString.indexOf("%");
   while (p>0) {
      if (ReqString.subString(p+1,2).isHexDigits()) // If it's a hex pair after %
         ReqString.change(
            ReqString.subString(p,3),               // Change % and hex pair
            ReqString.subString(p+1,2).x2c(),p,1);  // to char value using IString::x2c
      p=ReqString.indexOf("%",p+1);                 // Search for next %
   }
   lReqString=IString::lowerCase(ReqString); // Create lowercase version for searches
   //Get the file attribute's value, then set Filepath to the value of
   //the WebRoot macro plus a slash plus the file name. WebRoot and slash will be
   //defined shortly.
   Filename=GetValue("file");
   Filepath=Filename;
   Filepath.change(slash,"/");
if (Filepath.subString(1,1)!=slash)
   Filepath=slash+Filepath;
Filepath=WebRoot+Filepath;
Key=GetValue("key");                         //Get user's key}
```

Build the project again. There should be two errors, stating that declarations for WebRoot and slash could not be found. You will define these next.

Define Macros for the Request Class

Create a macros file, macros.hpp, and add it to the Source directive in the same way you added request.cpp and request.hpp above, then add the following macros to it. Be sure to change the string literal in the WebRoot macro to point to the path where HTML documents will be stored for review.

```
#define WebRoot IString("f:\\vatutor\\htmldocs")
// Note - change above path to the correct path on your system
#define slash IString("/")
```

If you were to rebuild the project now, the same errors would occur, because VisualAge C++does not grant macros global scope by default. This differs from class, member, variable, and function definitions, which are C++ constructs and are accessible to any part of a VisualAge C++ program as long as the source files containing their declarations are part of a project's configuration.

Give Macros Global Scope

1. In the **Configuration** section, select the **Options** page.
2. In the **Source and Targets** view at the upper left, select the `macros.hpp` source file object.
3. In the **Change Options** view to the right, use the **Live Find** field to search for global.
4. The option **Set global scope for macros** should be visible.

   The option shows a hatched check box, which indicates that the option's default value is being used. In this case the default value is off. When you click the check box it changes to unchecked with no hatching, which sets the option to be off explicitly. This will cause the option to appear in the configuration file. Clicking the checkbox again will select it, which enables the option.

   Click the **Set Global Scope for Macros** option by clicking the checkbox until it is checked.
5. Save the change by clicking **Apply** at the bottom of the view. The macros contained in macros.hpp now have global scope, and so are accessible to other source files in your project.

Build the application again. You may see a help tips window advising you that you can optimize your configuration. If so, click **OK**. Go back to the Classes page in the Project section, and try selecting the Request class constructor from the Members view of the class. Notice that as you do so, the source view underneath changes to show the constructor's definition, rather than its declaration within the class declaration.

Define the Request::GetValue Function

Implementation Details: The GetValue function searches through the request string for the specified name followed by an equals sign (=), and, if the name is found, returns the value that immediately follows the equals sign, up to but excluding the first whitespace character or the end of the string.

You do not need to add a new source file to your project to define this function; instead, you can add it to request.cpp. There are two ways you can access the source file:

- Because you already have a workbook section open for this source file, you can go to the request.cpp section and enter the source code shown below.
- Follow these four steps:
  1. Select the Overview page of the Project section.
  2. In the Source Files view at the upper right, select the request.cpp source object.
  3. The Source view for this source file is displayed underneath.
  4. Click anywhere in the Source view to give it focus, then press **Ctrl+End** to scroll to the bottom of the source file.

  Enter the following source code for the GetValue function at the bottom of request.cpp:

```
IString Request::GetValue(IString Search) {
   Search.lowerCase();
   Search+="=*";
   for (int i=1; i<=ReqString.numWords();i++)
      if (lReqString.word(i).isLike(Search))
         return ReqString.word(i).subString(Search.length());
   return "";
}
```

This function folds the search string to lowercase, appends "=*" to the search string, and uses the IString::isLike member function, which allows wildcard comparisons to a string, to compare the search string to each whitespace-delimited word in the lowercase version of the request. If a match is found, the value of the attribute is returned; otherwise the function returns a blank string.

Revise Your main Function to Use the Request Class

Build your application again. If you have typed everything correctly, the project should build successfully. However, your main function still has not changed, so the Request class is not actually used in your program. You will now modify the main function to declare an object of the Request class, which will cause the Request constructor to read in the CGI request information and set certain class variables (such as *Filename* and *Filepath*); main will then do some simple processing of the request to determine the output to write back to the user's browser.

You can open a Source view for the main function in any of the following ways. Try them all so you will better understand how the VisualAge C++ IDE gives you flexibility in how you view your program. In the first two methods, the Source view takes up the entire workbook section; in the others, the Source view is displayed below the other views.

- Select the main.cpp file section of the workbook, if it is still available.
- If the main.cpp file section is *not* still available, select the Overview page or the Source Files page in the Project section of the workbook, and click mouse button 2 over main.cpp in the Source Files view. From the pop-up menu, select Open as a Workbook Section.
- Open the Overview page or the Source Files page of the Project section of the workbook, and select main.cpp in the Source Files view. The Source view below displays the contents of main.cpp.
- Open the Declarations page in the Project section, and select `int main()` in the Declarations view. The source view underneath shows the source for the main function (not the entire contents of main.cpp - in any source view, by default only the code for object selected is shown).

If the Source view you are using is one of several views on the current page, you can enlarge it, by doing one of the following:

- Click the maximize square at the top right of the view. The view will fill the screen space for this workbook section.
- Move the mouse until your pointer rests over the border between the Source view and an adjacent view. The arrow changes to a two-ended arrow. Press and hold the left mouse button and drag the border up until the Source view is as large as you want it.
- Place the cursor over the main object anywhere you see it on the current page and click mouse button 2. Select **Open as a Workbook Section** from the pop-up menu.

Change the source code for the main function to the following:

```
int main(int argc, char* argv[]) {
   // Declare a Request object; this reads the request into fields of object
   Request Req;
   // Output is plain text unless later shown otherwise
   Boolean HtmlOut=false;
   // Initialize the Result string to contain output
   IString Result="";
   // For now, require that key value be "secret"
   if (Req.GetKey()!="secret")
      Result="Sorry, you aren't authorized";
   else {
      // Handle request for file.  Display if found.
      if (Req.GetFilename()=="")
         Result="Sorry, you didn't specify a file";
      else {
         ifstream InFile(Req.GetFilepath());
         if (!InFile) Result="Sorry, file not found";
         else {
            // IString::lineFrom(ifstream,EOF) reads entire file
            Result=IString::lineFrom(InFile,EOF);
            // If the file contains "<html>" or "<HTML>" set HtmlOut to true
            if (Result.indexOf("<html>")>0 || Result.indexOf("<HTML>")>0)
               HtmlOut=true;
            else if (Result=="") Result="Sorry, file was empty";
         }
      }
   }
   // Write the Content-Type header of the appropriate type (html/plain)
   cout << "Content-Type: text/";
   if (HtmlOut) cout << "html";
   else         cout << "plain";
   // Write the result. The two \n's are a required part of the header.
   cout << "\n\n" << Result << flush;
   return 0;
}
```

Rebuild the project. If there are typing errors, go to the Messages page and correct them, then rebuild. If you get an error saying that the text "InFile" is unexpected, add the preprocessor directive `#include <fstream.h>` to the top of the main.cpp file. (This directive was in the file before; you may have accidentally deleted it when you replaced the **main** function.)

Run and Test the Program

You can now test this version of the CGI in your browser. Make sure the VisualAge Help Server is running:

- If you started this exercise from the Help menu of the VisualAge C++ IDE, or if your browser currently shows the host name as localhost:49213, the VisualAge Help Server is already running.
- Otherwise, select Help from the IDE titlebar menu and select any help topic to start the VisualAge Help Server.

Enter the following URLs in your browser to see how the CGI behaves. In both cases the CGI returns an error message; the first message states that the requested file was not found, because you have not yet created the index.html file; the second states that you are not authorized, because the key "unauthorized" is not a recognized key.

- http://localhost:49213/cgi-bin/review.exe?file=index.html+key=secret
- http://localhost:49213/cgi-bin/review.exe?file=index.html+key=unauthorized

Copy any existing HTML file to index.html in the directory you specified as WebRoot in macros.hpp, then try the two URLs again. The second URL's result is the same as before; the first URL should display the file you requested, as an HTML file. If it does not, check the following:

- Ensure that the path you specified for WebRoot points to a valid directory on your machine (check the contents of macros.hpp from the Macros page in the Project section).

- Ensure that the file index.html is stored in that directory.
- Ensure that any permission settings for the directory and file do not prevent the file from being read.

Now that you have defined the Request class and modified main to use it, you can proceed to the next part, "Part 4. Developing Classes in the IDE".

Part 4. Developing Classes in the IDE

In this web-based review tool, four classes of users will have access to different combinations of operations. For example, authors can mark comments as complete, but reviewers can only add comments. In this section, you will implement a hierarchy of user classes, in which the base class, User, defines virtual functions for all available operations. You will use the coding of these user classes to learn many new techniques for developing object-oriented programs within the IDE.

This section should take you approximately one to two hours to complete. In it you will learn about the following IDE tasks:

- Eliminate #include preprocessor directives by including header files in your project's configuration
- Control information flow between panes in the IDE
- Develop code in the orderless environment of VisualAge C++, in which any class, function, variable, or other declared construct is visible to all parts of the project as long as it is declared *somewhere* in the project
- Use the Search page to locate text in your project
- Use the Macros page to locate and change project macros

If you are interested only in learning the IDE, and do not need to understand the details of the CGI application, you can skip passages titled *Implementation Details.*

You should have the following source files in a directory on your system:

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program

Implementation Details

User class member functions are defined to produce an "Unauthorized access" error message. In derived classes, these functions are overridden to perform appropriate work for the method and the class.

One advantage of a hierarchical implementation of user classes is that authority levels are managed automatically by the hierarchy, instead of requiring you to perform authority checks throughout your code to determine if the current user is allowed to perform the current operation.

The User class and its subclasses implement the following functions: View, ShowAdd, Add, and Close. For the User class, the functions return an error message. Three subclasses of User are defined, which override selected User class member functions. These subclasses are Reader, Reviewer, and Author.

Create the Source Files for the User Class

To begin, you need to create three files and add them to your configuration.

- user.hpp
- user.cpp
- funcs.cpp

For each file, follow these steps, which you have already followed for other files in Part 3:

1. To open the file, select **Project Workbook - Open or Create File** from the title bar menu, or press **Ctrl+O**. (Note that if you current have a source view selected, you can also select **Open or Create File** from the **Source** title bar menu.)
2. Enter the source file name.
3. Select **Open as a Workbook Section** so that a new section for the file opens in the workbook.
4. To have the source added to your configuration, select **Add Source to Project**, and turn on the **Add Source to Source Directive** check box. Click **OK**.
5. Select the source directive: in the **Add to source** dialog, select the source directive containing main.cpp, then press **Apply**.
6. Enter the source code for that file, as shown below.
7. Rebuild the application after adding the source code, to ensure that you entered the source code correctly.

If you create all three source files before adding any code, put a space in each source file in its workbook section, so that the first time you rebuild, all three files are saved and recognized as source files. If you do not add anything to the files before you build, the build will fail because the files do not exist.

Source Code for Auxiliary Functions in funcs.cpp

The auxiliary functions you will need for this section both return an IString value. The first, Unauthorized, returns an error message stating that the user does not have authority to perform the requested function. The second, Error, is used to wrap its two arguments, a title and a message, in appropriate HTML markup so that it returns a valid HTML document.

Create the file funcs.cpp, add it to your project, and add the following two function definitions to it:

```
IString Unauthorized(IString Function) {
   return Error("Unauthorized Access",
      "You do not have sufficient authority to access the "
      +Function
      +" function.");
}
IString Error(IString Title, IString Text) {
   return "<html><head><title>"
          +Title
          +"</title></head><body bgcolor=\"#FFFFFF\"><p><h3>"
          +Title
          +"</h3><p>"
          +Text
          +"</p></body></html>";
}
```

Rebuild your project. Notice that the project built successfully even though you did not include istring.hpp as a header file for this source file. The IDE processes the istring.hpp header file because it is included by request.hpp; its class and members can therefore be accessed by any source file in the project. It is still a good practice, however, to include the necessary header files at the top of each source file that uses them, in case you later try to compile the code using a product other than VisualAge C++, Version 4.0. You can add the following directive to the top of this file if you like:

```
#include <istring.hpp>
```

You may also be interested to know that you could have reversed the order of definition above, and defined Error before Unauthorized, even though Unauthorized makes use of Error; VisualAge C++, Version 4 does not enforce the C++ requirement for forward declarations before first uses.

For more information on orderless programming, see “Reduced Drudgery” on page 6.

Source Code for the User Class in user.hpp and user.cpp

The source code to be added to user.hpp and user.cpp is given in this and the next three sections. Watch for comments indicating which file should contain each portion of code. The project will build successfully and produce the same program behavior, regardless of where you place a given declaration or definition. However, it is probably a good idea not to stray too much from the usual practice of placing class interfaces in .hpp files, and non-inline function definitions in .cpp files, in case you later want to build your program using a different compiler.

Implementation Details: The User class defines the functions View, ShowAdd, Add, and Complete, to return an error message, because none of these actions can be performed by a user who is not a Reader, Reviewer, or Author. These functions will be overridden in the Reader, Reviewer, and Author classes, to allow actions appropriate to the user's authority level to be dispatched.

To locate a view of a source file, select its workbook section, or select the file name in the Source Files view of the Source Files page in the Project section. If no section tabs are currently visible, click the Restore icon at the extreme right end of the Title Bar.

```
// In user.hpp:
class User {
   Request* Req;
   IString UserId;
   virtual IString View()     {return Unauthorized("View");}
   virtual IString ShowAdd()  {return Unauthorized("ShowAdd");}
   virtual IString Add()      {return Unauthorized("Add");}
   virtual IString Complete() {return Unauthorized("Complete");}
   public:
      User(Request* req) : Req(req),UserId("") {}
      User(): Req(0), UserId("") {}
      IString Action();
};
// In user.cpp:
IString User::Action() {
   IString Action=IString::lowerCase(Req->GetValue("op"));
   if (Action=="") return Error("No Action Specified", "The CGI call did not specify an op= argument.")
   switch (Action[1]) {
      case 'a':
         if (Action=="add") return Add();
         break;
      case 'c':
         if (Action=="complete") return Complete();
         break;
      case 's':
         if (Action=="showadd") return ShowAdd();
         break;
      case 'v':
         if (Action=="view") return View();
   }
   return Error("Unknown Action Specified","The CGI call requested an unknown action, "+Action);
}
```

Source Code for the Reader Class in user.hpp and user.cpp

Implementation Details: The Reader class only overrides one member function of the User class, the View function. For now, this function, and all other methods overridden by derived classes of User, will simply display a message saying the function is not yet implemented. This will allow you to test whether the program is correctly dispatching based on the authority level of the user and the type of action requested, before you actually add the code for each action.

```
// In user.hpp:
class Reader : public User {
      virtual IString View();
   public:
      Reader(Request* req) : User(req) {}
};
// In user.cpp:
IString Reader::View() {
   return Error("Unimplemented","Reader::View is not yet implemented");
}
```

Source Code for the Reviewer Class in user.hpp and user.cpp

> ***Implementation Details:*** The Reviewer class inherits directly from User, rather than through Reader, because the Reviewer::View function is different from the Reader::View function, and therefore Reviewer has nothing to inherit from Reader.

```
// In user.hpp:
class Reviewer: public User {
      virtual IString Add(),
                      ShowAdd(),
                      View();
public:
   Reviewer(Request* req) : User(req) {}
};
// In user.cpp:
IString Reviewer::Add() {
   return Error("Unimplemented","Reviewer::Add is not yet implemented");
}
IString Reviewer::ShowAdd() {
   return Error("Unimplemented","Reviewer::ShowAdd is not yet implemented");
}
IString Reviewer::View() {
   return Error("Unimplemented","Reviewer::View is not yet implemented");
}
```

Rebuild the project now, and correct any typing errors. Once the project builds successfully, you can look at the Classes page in the Project section to see the classes you have defined so far.

Four classes should be displayed in the Classes view of the project (the top left pane on the Classes page). Try selecting different classes; then, for each class, select different members in the Details view to the right. Whereas in Part 3, the Source view for each member function only showed its in-class declaration, the Source view now shows the full function definition for any member function you have implemented.

Source Code for the Author Class in user.hpp and user.cpp

Implementation Details: The Author class inherits from Reviewer because the Add and ShowAdd functions are the same. However, the View function is implemented differently, because for an author, each comment must be displayed with a Complete link so that the author can mark the comment as completed. As well, the Complete function must be implemented to allow the author to complete the comment.

```
// In user.hpp:
class Author : public Reviewer {
      virtual IString View(),
                      Complete();
   public:
      Author(Request* req) : Reviewer (req) {}
};
// In user.cpp:
IString Author::View() {
   return Error("Unimplemented","Author::View is not yet implemented");
}
IString Author::Complete() {
   return Error("Unimplemented","Author::Complete is not yet implemented");
}
```

Rebuild the project to ensure that you have entered the above code correctly, and make any required corrections.

Update the Main Function to Dispatch the Appropriate Action

Implementation Details

The last piece of coding work for this part is to change the contents of the main function to do the following:

- Create a Request object, as before, to read the request
- Determine what user type the user's key represents, based on a list of keys stored in a file
- Declare a pointer to User and assign the appropriate type of user to it
- Call User::Action for that pointer to User, and write out the result of this function, as plain or HTML text depending on whether the result contains HTML markup or not

Locate a Source view of the main function. You can select the workbook section for main.cpp, or use one of the methods described in "Part 3. Declaring and Implementing a Class" on page 51. For a new way of locating a function, try the following steps from the Search page of the Project section:

1. In the **Live Find** entry field at the top of the Search view, type `main` and press **Enter**. A list of matches is displayed in the view.
2. Select the match that shows the function declaration for main. Its source code is displayed in the bottom pane of the Search page.

Change the contents of the main function to the following (do not remove the #include directive for fstream.h):

```
int main (int argc, char* argv[]) {
   // Create a request object to parse out the input
   Request Req;
   // Define four user types:
   enum {tUser=0, tReader=1, tReviewer=2, tAuthor=3 };
   // Determine the user's type, based on the user's key
   int uType=tUser;
   ifstream UFile(UserFilepath);
   IString UserLine="";
   Boolean KeyFound=false;
   while (UFile && uType==tUser) {
      UserLine=IString::lineFrom(UFile);
      if (UserLine.word(1)==Req.GetKey())
         uType=UserLine.word(2).asInt();
   }
   User* U;
   switch (uType) {
      case tReader:
         U=new Reader(&Req);
         break;
      case tReviewer:
         U=new Reviewer(&Req);
         break;
      case tAuthor:
         U=new Author(&Req);
         break;
      default:
         U=new User(&Req);
   }
   IString Result=U->Action();
   if (Result=="") Result=Error("No Output","The CGI did not return any output.");
   cout << "Content-Type: text/";
   if (Result.indexOf("<html>")>0 || Result.indexOf("<HTML>")>0)
      cout << "html";
   else
      cout << "plain";
   cout << "\n\n" << Result << flush;
   return 0;
}
```

If you rebuild now, you will find that UserFilepath is not defined. You can define this string in your macros file (macros.hpp) to the path and filename of the file that will contain user information. There are two ways you can find macros.hpp:

- Select its section in the workbook
- Go to the Macros page in the Project section. In the Macros view on this page, the first is selected by default. Because macros.hpp is the first object in the Macros view in this case, the contents of macros.hpp are shown in the Source view below.

Add a definition of UserFilePath to macros.hpp. Change the text shown in the example to a path that is valid on your computer:

```
#define UserFilepath IString("F:\\vatutor\\users.dat")
```

For testing purposes, create this file in the directory you chose and add the following lines:

```
reader 1
reviewer 2
author 3
```

If you create the file within the IDE, note the following points:

- When you open the file, do not select Add Source to Project.
- You must explicitly save the file by pressing **Ctrl+S**, selecting the 💾 toolbar button, or selecting **Source - Save File**. Files that are not part of your project are not automatically saved when you rebuild.

Once UserFilepath is defined, rebuild the project. You can use the links in the following section to test out your code.

Test the CGI User Creation and Action Dispatching

Ensure that the VisualAge Help Server is running on your machine, then try some of the commands in the table below to test out the CGI and see how it performs for users of different authority levels making different requests.

Add the instruction under **Command to Try** at the end of the URL for the executable. For example, to try the first command, type the following in the **URL** field:

```
http://localhost:49213/cgi-bin/review.exe?op=Add+key=unknown
```

User Type	Request	Command to Try	Message Type Returned
User	Add	op=Add+key=unknown	Unauthorized access
	Complete	op=Complete+key=unknown	
	Show Add	op=ShowAdd+key=unknown	
	View	op=View +key=unknown	
	Unknown request	op=Anything +key=unknown	Unknown action specified
Reader	Add	op=Add+key=reader	Unauthorized access
	Complete	op=Complete+key=reader	
	Show Add	op=ShowAdd+key=reader	
	View	op=View+key=reader	Reader::View unimplemented
	Unknown request	op=Anything+key=reader	Unknown action specified

Reviewer	Add	op=Add+key=reviewer	Reviewer::Add unimplemented
	Complete	op=Complete+key=reviewer	Unauthorized access
	ShowAdd	op=ShowAdd+key=reviewer	Reviewer::ShowAdd unimplemented
	View	op=View+key=reviewer	Reviewer::View unimplemented
	Unknown request	op=Anything+key=reviewer	Unknown action specified
Author	Add	op=Add+key=author	Reviewer::Add unimplemented
	Complete	op=Complete+key=author	Author::Complete unimplemented
	ShowAdd	op=ShowAdd+key=author	Reviewer::ShowAdd unimplemented
	View	op=View+key=author	Author::View unimplemented
	Unknown request	op=Anything+key=author	Unknown action specified

If the output you see in your browser after trying these links does not reflect what is indicated under "Message Type Returned" in the table, you may have entered some of the code incorrectly. See below for possible causes and corrective actions. If the links worked as expected, you can proceed to "Part 5. Debugging and Revising Your Application" on page 68.

Correct Output Errors

The following are some errors that may occur as you try out the links in the table above. Possible causes and corrective actions are given.

- **Message indicating that the server is not responding**

  You have not started the VisualAge Help Server. Start the IDE and select one of the items under the Help menu. This should start the VisualAge Help Server. If no help displays in a browser, you may need to reinstall the product and ensure that the Help components are installed.

- **Message indicating that the object could not be found or the script request is not valid**

  Check for the following possible problems:

  - Ensure that you have set the read/write permissions correctly for the HTML file you created
  - If permissions are set correctly, or if you have changed them, make sure you have cleared your browser's chached memory by reloading or refreshing the view.
  - The VisualAge Help Server is running, but the CGI application could not be found in the directory where the help server expects to find CGIs. Your executable files must be in the correct directory. In addition to your executables, the directory

should also contain other files that are not part of your project. If it does not, you may have accidentally created a new directory instead of choosing the existing one.

- If only your executable file is stored in that directory, you have installed the VisualAge Help Server in a different directory. Locate that directory by searching for the file vacwebx on your file system. When you have found the directory where this file is stored, do the following:
  1. Select the Targets page in the Configuration section.
  2. Select the target object in the Targets view at left.
  3. Click the **Change Target** entry field in the Change Targets view at the right.
  4. Correct the path of the executable file to include the appropriate directory.
  5. Click **Apply.**
  6. Rebuild the application.

If other files are in the directory but your executable file is not, follow the six steps shown above, but for step 4 use the directory where the VisualAge Help Server is installed. Make sure the file name is spelled correctly (for example, you may have accidentally specified `reveiw.exe` as the target when you set up your project, instead of `review.exe`).

- **Dialog asking where you want to save the file**

  The VisualAge Help Server is not configured properly, or the CGI is stored in a document directory instead of a CGI directory, or the CGI executable terminated abnormally.

  - If the Help Server is incorrectly configured, it treats the executable file as a downloadable file rather than a CGI (for example, the directory the file is stored in may not be the directory where the Help Server expects to find CGIs). You may need to reinstall VisualAge C++ to solve this problem.
  - If the executable terminated abnormally, you should try invoking it from a command line in the directory to which the target was written, to ensure that it built successfully. If it fails when invoked from the command line, rebuild the application and try again.

- **CGI appeared to work, but not all actions yielded the expected results**

  You may not have entered the source code correctly. Go to the online version of this exercise (the Tutorial) and replace the source code in your project with the source files provided at the start of Part 5. Debugging and Revising Your Application, by copying and pasting from your web browser into the IDE. Rebuild the application and try again.

Once you have corrected any problems, proceed to “Part 5. Debugging and Revising Your Application”.

Part 5. Debugging and Revising Your Application

Your CGI now handles GET and POST requests, the two types of requests that would be received from a user's browser calling the CGI. In this section, you will add a third request method that uses command line parameters as the source of the CGI arguments, so that you can start a debug session for the project within the IDE, and supply the request you want to debug as the program's command line parameters. This change will allow you to debug new methods as you implement them later, using the debugging features of the IDE.

You will also implement the Reader::View function and an auxiliary function, ReadFile. The Reader::View function will have a bug in it that you will try to find and fix.

This section should take you approximately 45 minutes to complete. In it you will learn to perform the following IDE tasks:

- Remove multiple workbook sections from the Workbook section
- Mark and copy text in a source view using the line-marking and copying macros of the source editor
- Set breakpoints in your code, and disable default breakpoints, before you start a debug session
- Start a debug session from the Debug page of the Project section
- Step through your code, run to a breakpoint, and run to a selected statement in a debug session
- View the threads, call stack, stack frames, and local variables for a process running under debugger control (a *debug process*)
- Add a new pane in the IDE
- Resize a pane in the IDE
- Change or set the object type for a pane
- Change the view type for a pane
- Create and save a file that is not part of the project
- View a region of storage used by a debug process
- Correct a bug in your code while debugging

If you are interested only in learning the IDE, and do not need to understand the details of the CGI application, you can skip paragraphs or sections titled Implementation Details.

You should have the following source files in a directory on your system.

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program
- funcs.cpp - Auxiliary functions used by user classes
- user.hpp - The class declarations for the user classes
- user.cpp - The function implementations for the user classes

Remove Unneeded Workbook Sections

If you have not yet exited the IDE, started it up again, and reloaded your project, the workbook sections may appear very crowded, because you have a workbook section for each of your source files.

Once your project has built successfully, the contents of all the source files will be represented in the codestore. You no longer need to have separate workbook sections for each file. You may find it easier to navigate the source files, functions, classes, declarations etc. through the Project section of the workbook You can remove each source file workbook section by clicking mouse button 2 over the section's tab and choosing **Remove This Object's Workbook Section**. However, if you want to remove a large number in one action, you can follow these steps instead:

1. Select the Workbook section, Overview page. This page shows a Table of Contents view, which is a list of the current sections and pages of the workbook.
2. Scroll down the view to the first File section.
3. Select that section by clicking mouse button 1 on the section entry.
4. Hold down the **Ctrl** key, and click mouse button 1 on each other workbook section you want to remove.
5. Click mouse button 2 while over one of the selected entries, and select **Remove This Object's Workbook Section.**

You can open a series of workbook sections for different files by selecting each desired file in a Source Files view (for example, on the Overview page of the Project), clicking mouse button 2, and choosing **Open as a Workbook Section.**

Change the main Function to Accept Command Line Arguments

Implementation Details: A CGI is never invoked with command line arguments from a user's web browser, but for debugging purposes it is useful to allow the arguments to be specified by the command line instead of environment variables. The lines you add below set the necessary environment variables so that the arguments provided at the command line can be treated as a regular GET request coming from a web browser.

Add the following lines to the start of the main function, right after the opening brace:

```
if (argc>1) {
   IString Args="";
   for (int i=1;i<argc;i++) Args+=IString(argv[i])+"+";
   Args.stripTrailing('+');
   putenv("REQUEST_METHOD=GET");
   char*TArgs=strdup("QUERY_STRING="+Args);
   putenv (TArgs);
   }
```

Define the ReadFile Function in funcs.cpp

The ReadFile function returns the contents of a specified file. This is an auxiliary function, so you should add it to the source file funcs.cpp. You also need to include the fstream.h header file at the top of funcs.cpp. Although the file already has access to the fstream class (because fstream.h is included elsewhere in the program), it does not have access to the EOF macro that is defined when fstream.h is included, because macros do not have global scope by default.

To the top of funcs.cpp, add:

```
#include <fstream.h>
```

At the bottom of funcs.cpp, type the following:

```
//... Previously defined functions
IString ReadFile(IString Path) {
   ifstream In(Path);
   if (!In) return "";
   return IString::lineFrom(In,EOF);
}
```

Redefine the Reader::View Function

Implementation Details

Currently, this function displays a message indicating that it is still unimplemented. You will change it to a function that does the following:

- Retrieves the contents of the requested file, using the ReadFile function you just defined.
- Changes all <a href=...> links in the document to calls to the CGI with the correct op=, file=, and key= values. When the link is followed, the CGI is invoked to display the new file.

Before you start, you should first define the macro EXE in macros.hpp, so that it contains the full filename (but not the path) of the CGI executable file:

```
#define EXE IString("review.exe")
```

Next, redefine Reader::View by selecting the current contents of the function in one of the source views that display it, and typing the code below.

Implementation Details: Note that a char* array is used to store the modified version of the requested file's contents; although the IString class supports a concatenation operator (+=), it reallocates storage for the target string after each allocation, which is less efficient than anticipating the total length required, and declaring a char* array sufficiently large to hold the final output string.

```
// In user.cpp
IString Reader::View() {
   // Call ReadFile to read the file pointed to by Request::FilePath
   IString Result=ReadFile(Req->GetFilepath());
   if (Result=="")
      return Error("Could Not Read File",
         "The file you requested, "
         +Req->GetFilename()
         +", was empty, or could not be read.");
   // Find the first "href=" attribute.
   int href=Result.indexOf("href=");
   // Return if there are no href= tags to fix
   if (href==0) return Result;
   // Substitute the executable name and necessary arguments
   // in all href= calls.  EXE must be defined in macros.hpp
   IString LinkInsert=EXE+"?Op=View+Key="+Req->GetKey()+"+file=";
   // Allocate a new IString large enough to contain the new string,
   // by determining the number of inserts to do.
   int NewLength=Result.occurrencesOf("href=")
       *LinkInsert.length()
       +Result.length()+1;
   char* NewResult=new char[NewLength];
   NewResult[0]='\0';
   int last=0;
   while (href>0) {
      // Write from last position written up to past the href=,
      // and add the link insertion text
      strcat(NewResult,
             Result.subString(last+1,href+5-last)+
             LinkInsert);
      last=href+5;
      href=Result.indexOf("href=",last);
   }
   return NewResult; // Automatically converted to an IString
}
```

Rebuild the project. You should see three identical errors, stating that the private member User::Req cannot be accessed. Because the Reader, Reviewer, and Author classes will often need to access information from the Request object, this member should be declared protected.

1. Obtain a Source view for the declaration for the User class.
2. Select the line in the Source view containing `Request* Req;`.
3. Press **Alt+L**. This keystroke shortcut highlights a line for moving, copying, or deleting.
4. Move the cursor down to the line before the public access specifier.
5. Press **Alt+M**. This keystroke shortcut moves the currently selected text below the current line.
6. Insert the protected access specifier above the Request declaration. The class declaration should now look like this:

```
class User {
   IString UserId;
   virtual IString View()     {return Unauthorized("View");}
   virtual IString ShowAdd()  {return Unauthorized("ShowAdd");}
   virtual IString Add()      {return Unauthorized("Add");}
   virtual IString Complete() {return Unauthorized("Complete");}
   protected:
      Request* Req;
   public:
      User(Request* req) : Req(req),UserId("") {}
      User(): Req(0), UserId("") {}
      IString Action();
};
```

Rebuild the project. Before you try the Reader::View function, create a file with some links in it and save it as index.html in the directory to which the WebRoot macro points. (Use the file contents shown below even if you already copied a file to index.html in a previous part of the exercise.)

You can create a file in the IDE that is not associated with the project by following these steps:

1. Select **Project Workbook - Open File** from the titlebar menu or press **Ctrl+O**.
2. Enter the new file name, `index.html`.
3. Select **Open as a Workbook Section**
4. Make sure that the **Add to Project** check box is *not* checked. Click **Open**.
5. After you enter the text, remember to save the file. (Files that are not part of your project are not automatically saved when you build.) Press **Ctrl+S**, click the **Save File** button from the toolbar, or select **Source - Save File** from the titlebar menu.

The index.html file should contain the following:

```
<html><head><title>Problems with Links</title></head>
<body><p>Here are some common problems:</p>
<ul>
   <li><a href="http://elsewhere.com/other.htm">Some links are offsite</a></li>
   <li><a href="nothere.htm">Some links are to nonexistent files</a></li>
   <li><a href="here.htm" target="elsewhere">Some links are to different frames</a></li>
   <li><a HREF="upper.htm">Some links have uppercase attributes</a></li>
   <li><a href=" there.htm">Some links are syntactically invalid</a></li>
   <li><a href="there.htm">Some links are fine but fail in the CGI</a></li>
</ul>
</body></html>
```

Save this file, then try out the following link to see how the Reader::View function works.

http://localhost:49213/cgi-bin/review.exe?op=View+key=reader+file=index.html

If you get an "unauthorised" error message, you did not create the user file defining access levels for Reader, which the UserFilepath macro in macros.hpp will point to. This is described in Source for macros.hpp . Once you have done this, try the link again.

When the index.html file is displayed, it should look something like the following:

Here are some common problems:

- Some links are offsite
- Some links are to nonexistent files
- Some links are to different frames
- Some links have uppercase attributes
- Some links are syntactically invalid
- <a href="review.exe?Op=View+Key=reader+file=

The last link got messed up by the CGI. You will correct this problem by debugging your program. If another browser window opened when you tried the third link, close the extra browser window now.

Implementation Details

None of the other links actually work, because:

- The first link, to an offsite location (one starting with http://) should not have been modified by the CGI.
- The second and third links point to files not found in the WebRoot directory. These two problems are easily solved by creating the necessary files. The second link points to a frame that does not exist, so a new browser window opens to display the error message.
- The fourth link has its href= attribute in uppercase (HREF=) and so this was not found by the CGI. However, because the browser considers the referring document (your CGI) to be stored in the cgi-bin directory, it adds "cgi-bin" to the path of the link.
- The fifth link was syntactically invalid (the href= attribute began with a space).

If you intend to use this exercise to create a working review tool, you can fix problems with these links yourself later, by adding additional code to the View function to anticipate offsite locations, href attributes in uppercase, and syntactically invalid links.

Set and Disable Some Breakpoints before You Start Debugging

The VisualAge IDE allows you to set breakpoints before you start a debugging session, and supports a wide range of breakpoints, including these breakpoint types:

- Statement breakpoint - Execution stops immediately before executing the statement
- Function breakpoint - Execution stops at the first statement in the function when it is called
- Class breakpoint - Sets a function breakpoint in every function of the class
- File breakpoint - Sets a breakpoint in every function defined in the source file
- Virtual function breakpoint - Sets a function breakpoint on the selected function, and on all overrides of that function in derived classes

Before you start debugging, it would be useful to set a function breakpoint in Reader::View, so that execution stops at the first statement in the Reader::View function.

1. Locate the Reader::View function in any view that displays the function and shows an F object to its left, for example the Class Details view of the Reader class (from the Classes page).
2. Position the mouse pointer over this symbol, and click the right mouse button. From the pop-up menu, select **Set function breakpoint** or **Set virtual function breakpoint** (since the Reader class is the class whose View function you want to debug, either breakpoint type will do).

Next, select the Debug page of the Project section. This page, which you will use to start a debugging session, is divided into the following panes:

- A Run Specifications pane, which you will use to start debugging and specify command-line arguments.
- Two panes for viewing and managing breakpoints. Notice that the breakpoint you set for the View function is shown in the first of these views, as well as a breakpoint for the entry point of the main function, which will cause execution to stop on entry to main.
- A process log, in which the IDE tracks debugging-related events, such as signals and process termination.
- A list of debuggable processes (currently empty, because you have not yet started debugging within the IDE).
- A list of monitored expressions, also currently empty.

Before you start the debug session, you need to provide the request to the CGI. Enter the following string in the **Arguments** field of the run specification:

```
Op=View+Key=reader+File=index.html
```

If you were to start debugging now, execution would stop on entry to main, because of the entry point breakpoint. You have already set a function breakpoint for the function you want to start debugging in, so you should disable the entry point breakpoint. Click the ⯃ red octagon beside the **Entry Point main of *** breakpoint; the red octagon turns green, indicating that the breakpoint is disabled.

If you try expanding different entries in the **Breakpoints** view of the project, you will see that all but the **All Breakpoints** breakpoint are childless. Each time you start a debugging session for the project, a child breakpoint specific to the new debugging process is added to each existing breakpoint, and that child breakpoint's state (enabled or disabled) is inherited from its parent. The IDE allows you to debug several copies of the project executable file as separate processes at the same time; you can disable or enable individual breakpoints at a project level by clicking the octagon beside a parent breakpoint, or at a process level by expanding the parent breakpoint and clicking the octagon beside the appropriate child. If you want to disable or enable all breakpoints, you can click the octagon beside **All Breakpoints**, until the breakpoint's state has overridden all child breakpoint states.

You are now ready to start the debugging session.

Start the Debug Session

From the Debug page of the Project section, click the **Debug** button to start the debugging session.

When you start debugging a program, the IDE creates a process for that program, and opens a Process section in the workbook corresponding to that process. It also adds buttons to the toolbar; these buttons allow you to step through the process and perform other debugging actions. If the program is a console (non-windowed) program, such as this CGI, a console window opens so that console input can be obtained and output can be displayed. The IDE automatically regains focus after the console window is opened.

Normally you do your debugging from pages within a Process section. You can debug several versions of the same program by starting debug processes several times for the same executable file, using the same or different sets of arguments. To switch from debugging one process to another, select the Process section for the process you want to debug next.

The program should run until the first executable statement of Reader::View. (If execution stops at the start of the main function you did not disable the entry point breakpoint; press **Ctrl+Shift+U** to run to the next breakpoint.) The current page of the Process section is the Source page, which is divided into four panes:

- The Threads pane displays an expandable object for each thread in your program. The program has only one thread, so this view may not seem useful here, but in fact it illustrates an important IDE concept. Expand the thread entry; you will see a list of five subentries. Each of these subentries can also be expanded to reveal details about the process being debugged. If you expand the stack entry, for example, you will see something that looks very much like the Stack view in the pane below the Threads view. If you then expand any Stack Frame within that stack entry, you will see a list of the local variables for that stack frame, very much like the Local Variables view in the right-hand pane.
- The Stack view displays entries for each allocated stack frame, with the topmost frame corresponding to the current function. As you select different stack frames, the remaining two views are updated.
- The Source view displays the source code for the function currently selected in the Stack view. As you step through this code, the blue arrow to the left of the current line, and the blue rectangle shading the start of the next statement to be executed, move through the source code to indicate the current execution point.
- The Local Variables view displays all variables that are local to the function or a block within it. You can expand a variable to see its contents or subparts.

In the Local Variables view, expand the Result variable, then the pBuffer member of Result. The data member pBuffer points to Istring's storage. The value of pBuffer may be null, or the string may contain random data, indicating that no buffer has been allocated. Step over the call to ReadFile by clicking the toolbar button or by pressing **Ctrl+Shift+O**. The contents of the buffer should change to show the start of an HTML file. Expand NewResult to show its contents; this char array also appears to contain random data, as it has not yet had storage allocated to it.

Debug the String Concatenation

Since you are trying to debug the eventual contents of NewResult, you should run the program up to the point where NewResult has a first substring concatenated to it.

1. Find the statement `last=href+5` (use theLive Find entry field, or scroll down the Source view until you find it).
2. Position the mouse pointer over this statement in the source view and click the right mouse button.
3. Select the Stmt object, and from the cascading menu select **Run to This Statement**.

You can also click the gray square in the left margin beside the statement to set a breakpoint on the first statement of the line, then click the run button in the toolbar to run to the first breakpoint.

It would be easier to debug the concatenations of source string parts to the target string if the target string were displayed in a flowed view, so that you did not have to keep scrolling to the right. Follow these steps to obtain a Storage view of the target string; this will help you learn to use the Storage view of the debugger portion of the IDE.

1. Select the Storage page of the Process section.
2. Increase the height of the Threads view by moving the mouse over the line dividing the threads pane from the panes below it, then dragging the line down until you can view at least five lines of the Threads view.
3. Expand the Thread object, and the Stack object of the thread. The current stack frame for the thread is displayed in an expanded state. Select this stack frame; it should be the stack frame for Reader::View.
4. Scroll down through the local variables of the stack frame to the Result variable. Expand this variable as you did earlier, and expand its pBuffer member to display the contents of the Result string. Select the contents; notice that the Storage view at bottom right updates to show the start of storage for that string.
5. Scroll down further to the NewResult variable and expand it, then click the text contents shown for it; the Storage view now shows the contents for that string.

The default storage view is not suitable for viewing the contents of long text strings. To change this view to something more suitable, follow these steps:

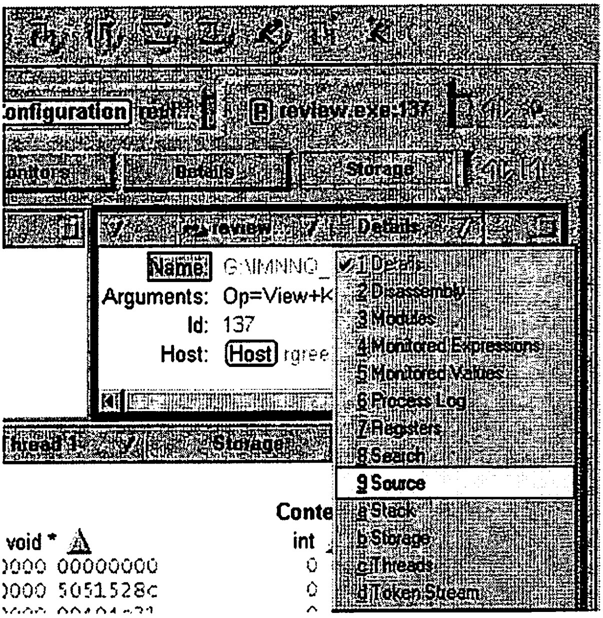
1. Click the arrow to the right of the int column header, then select **Destroy** from the pop-up menu. This removes the column from the Storage view.
2. Remove the Single byte character column as well. You now see only the void* column and the Addresses column.
3. Click the arrow to the right of the void* column, and expand the **Choose a built-in type** choice. Change the built-in type to **Single byte character**. When you have made this change, click the arrow to the right of the void* column again to minimize the pop-up list.
4. Expand the **Storage Options** entry at the top of the storage view.
5. Expand the **Number of bytes per line** entry beneath this, and change the number of bytes to 32.
6. If you want, you can also increase the number of lines of storage displayed.
7. Collapse the storage options. If you cannot see all 32 characters of each line of text, drag the left edge of the pane farther to the left, or change the number of bytes per line to a smaller number.

It would also be useful if the start of the string were at the top of the Storage view. You can move it up by clicking the single down-arrow (▼) to the left of the Addresses column.

The Source view of the process is not visible on the Storage page, but this is the view you need, so you can step through your code while watching how each statement affects the contents of the **NewResult** character array. Follow these steps to add a Source view:

1. Hold down the Ctrl key and position the mouse pointer over the right edge of the Threads view.
2. When the pointer icon changes to an arrow ⇤ drag the new pane left until it takes up half the top portion of the window.
3. Click the arrow to the right of the Thread object shown on the title bar of the **Details**view.
4. Select **Opened Objects** from the menu.
5. Select the P object for the current process from the menu below that to set the object type of the new pane to the process object for this process.
6. A Help Tip appears warning you that changing the object type of the pane will detach the pane from other panes that link to it. Click **OK**.

7. Change the View type to a Source view, by clicking the arrow to the right of the **Details** entry in the view's title bar, and choosing **Source**.



Name: G:\MMNQ_
Arguments: Op=View+K
Id: 137
Host: Host rgree

```
)000 00000000    0
)000 5051528c    0
```

8. Another Help Tip appears; click **OK** again.
9. If you cannot see enough of the Source view (at least four or five lines of source code), slide the bottom edge of the Source view's frame down. This also resizes the adjacent view. You can obtain even more screen space for the views with the **Maximize** button above the project toolbar, on the far right. This hides the workbook sections. To display them again, click the restore button that has replaced the maximize button.

You can now step through the source code and watch the NewResult string grow. The debugger provides several types of step commands. The Step Over command (or **Ctrl+Shift+O**) lets you step over source code statements without stepping through any code that may be called by them. Other step commands include Step Debug, Step Into, and Step Return. If you are interested in the types of breakpoints available, see Types of Breakpoints in the online Concepts help.

Step over the statements in the **while** block one by one. Notice, each time the `NewResult` string is concatenated to, that the storage view updates to show the changed contents in red. On the next step command, the red changes to black. Continue stepping until the **while** loop ends. Suddenly the bug becomes clear: you have not concatenated the remainder of the file after the last reference you processed.

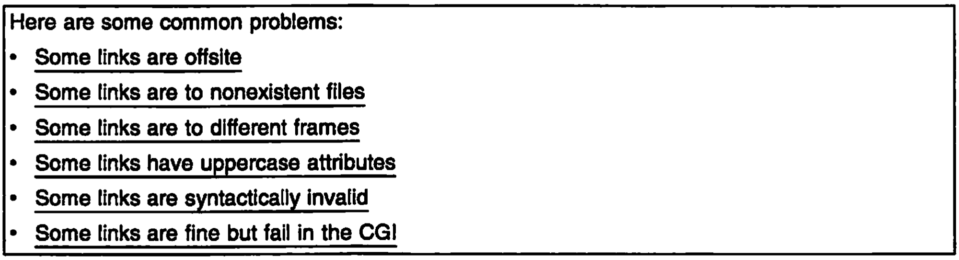
Add the following code in the current source view, before the **return** statement:

```
if (last>0) strcat(NewResult,Result.subString(last+1));
```

Before you rebuild the project, click the **Run toolbar** button to run to the end of the program. If you do not run to the end of the program, when you rebuild you will be warned that all currently running processes will be terminated. The IDE prompts you for this because you cannot change the object code in an executable file while that executable is running, and you may need to run your executable to the end if you have any writable files open, otherwise their data may be lost. Since you are not accessing any files in write mode, click **OK**. Once the program is rebuilt, try running it again from your web browser, at:

http://localhost:49213/cgi-bin/review.exe?op=View+key=reader+file=index.html

The last link in the HTML document should display correctly:



Here are some common problems:

- Some links are offsite
- Some links are to nonexistent files
- Some links are to different frames
- Some links have uppercase attributes
- Some links are syntactically invalid
- Some links are fine but fail in the CGI

Notice that you have just made a source code correction in the middle of a debugging session. The source code change does not affect the current process being debugged; you could have continued stepping through the old version of your executable file, and the source code change would be remembered. Even if you terminate a debuggee process after changing code in a Source view from the Process section, your changes are remembered.

You can now go straight to "Part 8. Defining the View Function for Reviewers" on page 87 , if you want to continue coding the CGI without making configuration changes. This and the following section will yield a functional web-based review tool, and will take about two hours. Or you can work through "Part 6. Managing Configuration Files" on page 81 , if you want to learn more about project configuration and customizing your configuration file (45 minutes), and "Part 7. Optimizing Your Configuration" on page 86, if you want to use the IDE's built-in configuration optimizer, which can improve build performance (15 minutes).

If you feel you have already learned enough about the VisualAge IDE to start doing your own coding, and you do not plan to use the web-based review tool, you can stop doing the CGI exercise now, and proceed to the next section on creating a GUI.

The visual component "Develop a Graphical User Interface from a Visual Part" on page 103 , shows you how to quickly build windowing interfaces and integrate them with your own source code in a VisualAge C++project.

Part 6. Managing Configuration Files

By now, you have probably worked enough with your project to understand that it is controlled by a configuration file, and that you can edit that file to modify the source files, targets, and options for your project. You may also have noticed that the configuration file, or .icc file, is coded in a structured language similar to C or C++.

In this section, you will add two user variables to the configuration, one for a working version of your target and one for a production version. You will apply options to each version. You will then create a third user variable and add an if-block to your code, so that, depending on that third variable's setting, either the working version or the production version will be generated when you build.

This section should take you approximately 45 minutes to complete. In working through it, you will learn to perform the following IDE tasks:

- Obtain different views of options
- Modify your project configuration
- Create your own options groups
- Create your own configuration variables
- Associate options with specific objects in your project, such as source directives and targets
- Apply options to a target

You should have the following source files in a directory on your system.

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program
- funcs.cpp - Auxiliary functions used by user classes
- user.hpp - The class declarations for the user classes
- user.cpp - The function implementations for the user classes.

Review of Configuration Pages and Views

There are two principal purposes for your project configuration:

- To specify the source files and target that make up your project
- To specify what options apply to the target or to source files

Select the Configuration section, then the Options page. You can use the Options page to set options for specific target or source objects in your project. To do this, select a source or target object in the Source and Targets view on the left, then apply options specifically to the selected source or target by changing options in the Change Options view to the right. See "Setting Build Options" on page 34.

Next, select the Source and Groups page. The Source and Groups view shows that your source files are grouped into two source directives, the second group containing only macros.hpp, the first group containing all other source files mentioned in the configuration file. macros.hpp is in its own group because earlier in this exercise, you applied the global macros option to that file only. If you select the source directive object Sd for macros.hpp, the current line of the source view underneath changes to show the source specifier for this file, and you can see the global macros option enclosing the source directive.

If you wanted to add more source files whose macros should be treated as having global scope, you could select this source directive in the Source and Groups view, enter the new file names in the Source entry field of the Change Source and Groups view to the right, and click **Add**, then **Apply**. Note that this action does not create the source file for you; if you want to create a source file and add it to an existing project, the best way is to use the **Open File** dialog, and check the box to add the file to a source directive. A subsequent dialog asks you which source directive to add the new file to, and a source view then lets you create the contents of the new file.

You can apply actions to most objects in the configuration views by clicking mouse button 2 over the object. A pop-up menu appears, showing which actions you can perform on the object. You can select multiple objects and click mouse button 2 over any one of them. Any actions that apply to all the objects selected are displayed in a pop-up menu. For example, you can select several source files in a Source and Targets view, click mouse button 2, and select Remove from the pop-up menu to remove all the files from your configuration, or you can change the paths of all selected files to relative paths (relative to the project working directory), or to full paths.

The next page is the Targets page, where you will see a list of the targets for your project in the Targets view, with a list of the associated source directives for each target in the view beneath. If you select the Target object in the Targets view on this page, you can then modify the target path in the Change Targets view to the right by editing the entry and clicking **Apply**.

The Project Options page is useful for setting options that apply to the entire project. When you first set options from this page, the IDE creates a named option group called ProjectOptions that sets the options you choose, and encloses the entire configuration (source directives, target directives, other options) within its braces. By enclosing the rest of the configuration within the ProjectOptions group, the IDE enables future options applied to this group to apply to the entire project. Note, however, that if you edit your configuration file later, and add configuration information outside the ProjectOptions group, options you set from the Project Options page will not affect these additions.

The Options Groups page shows groups of options for your project. By default, options that all apply to the same part of your configuration (for example, a source directive) are gathered into an unnamed group. This page is a good starting point if you are trying to locate an option you know you have set, when you can't remember what files the option applies to in your configuration. Option check boxes and radio buttons appear shaded or hashed if the default value is currently being used. Descriptions are shown in green for those options you have explicitly changed.

In the Options Groups view at the left you should see two unnamed groups, one for global macros and the other for all other options. You can select any of these groups of options and modify the options settings in the Change Options Groups view to the right. The options you add are added to the group, while the options whose default values you restore are removed from the group. From the Change Options Groups view you can not only change option settings, you can also give an option directive a name or remove a name.

You will use the Options Groups page to make the first changes to your configuration. The two remaining pages are Advanced and Source.

The Source page shows an editable Source view of the configuration file.

The Advanced page shows two views of the configuration, a Details view and an Interpreted view. Both views are hierarchically organized, collapsible views of the contents of your configuration. The Details view includes any changes you have made to the configuration since the last build, other than direct editing changes. (For example, changes you made by adding source files through a dialog or pop-up menu are displayed in this view). The Interpreted view reflects the configuration file as currently interpreted by VisualAge C++. Where a variable is used in an expression, it displays the current value of the variable rather than its name; it displays in grey text those sections of the configuration that were not interpreted (for example, those contained in **If** blocks whose conditions evaluated to false). The Interpreted view does not reflect changes you make in the Details view until you do one of the following:

- Rebuild the project
- Refresh the Configuration views, which causes the configuration to be reinterpreted, and all configuration views to be updated, without starting a build. Select **Project Workbook - Refresh Configuration Views** from the menu bar
- Press the **Interpret** button at the bottom of the Interpreted view, which changes the Interpreted view to match the current contents of the configuration, without changing other configuration views

The **Advanced** page is very useful for ensuring that the changes you make have the intended effect while modifying your configuration.

Add Options Groups to the Configuration

You will start changing the configuration by creating two simple, named options groups, one for the working version of the program, one for the production version. The working version uses the shared versions of the run-time libraries; the production version is statically linked to the run-time libraries, and is also optimized. Follow these steps to create these two groups:

1. In the Configuration workbook section, select the Options Groups page.

2. In the Options Groups view of the Options Groups page, move the mouse pointer over a blank area of the view and raise a pop-up menu (click the right mouse button). Select **Add Option...**
3. In the Add Option dialog, enter the name `WorkingVersion` in the Group entry field at the top of the dialog, and click **Apply**. If you wanted to you could set the applicable options within this dialog; however, because the dialog does not support Live Find, it is easier to save the options group as is, then apply the changes from within the Options Group page.
4. Repeat steps 1 and 2 for another new option, and name this one `ProductionVersion`.
5. Select `WorkingVersion` in the left view, and in the Change Option Groups view to the right, turn on the option to link with the shared libraries (use Live Find and type `shared` to locate this option). Click **Apply** to save this change.
6. Select **ProductionVersion** in the left view, and turn *off* the option to link with the shared libraries. (You may need to click this box twice; it is a three-state checkbox, whose values are on, off, and default. Turn the option off.). As well, turn *on* optimization by using Live Find to locate `Optim`, and pressing **Enter** or **Ctrl+N** until the current option selected is "Optimization", with two choices below it, "Yes" and "No". Change the choice to "Yes". Click Apply to save these changes.

   Go to the Advanced page. Notice that these options groups appear with no source or targets within them. You will not add any source or targets to them; instead, you will assign one or the other option to a third variable, depending on whether you are still developing the program or are ready to ship it.

Create a Variable in the Configuration

Configuration files support untyped user variables (variables in which the type is determined by the assignment, not by a type declaration). You can add the definition `abc="def"` to your configuration file, and `abc` will be treated as a string containing the characters `def`. You can then use comparison operators to compare variables of compatible types, or to compare a variable to a constant.

For this configuration file, you need to define a variable `ReadyToShip`, with a value of either `true` or `false`, and a variable `LinkOptions`, whose value is determined by the value of `ReadyToShip`. You need to make these changes in a source view of the configuration.

1. Select a page of the Configuration section that has a source view, for example the Source page.
2. In the source view, you can, if you like, delete the empty braces after `ProductionVersion` and `WorkingVersion`, although the braces do not affect the syntax. You can highlight and delete a range of lines by moving the cursor, pressing **Alt+L** on the first and last lines you want to select, then pressing **Alt+D**.
3. At the top of the configuration file, insert the following:

   ```
   ReadyToShip="false"
   ```
4. Below the ProductionVersion and WorkingVersion options you recently defined, add the following configuration code:

```
if (ReadyToShip=="true")
   option LinkOptions=ProductionVersion
else
   option LinkOptions=WorkingVersion
```

When you make changes to your configuration, you need to either rebuild your project, or refresh the configuration views, to see how your changes affect the configuration. Rebuilding the project is time-consuming, because when you change your configuration even slightly, the entire codestore may need to be recompiled. This differs from source code changes, where small changes usually result in short rebuild times.

The IDE lets you view configuration file changes without rebuilding. Select **Refresh Configuration Views** either from the **Project Workbook** menu, or from the pop-up menu that appears when you click mouse button 2 over the tab for the **Configuration** section. Then go to the Advanced page. Notice that, in the Interpreted Configuration view, the statement `option LinkOptions = ProductionVersion` is grayed out, because the **if**-block containing this statement did not get evaluated. Also notice that the value of the variable `ReadyToShip` is shown in the **if** statement, instead of the variable name.

Add the LinkOptions Option to the Target Statement

Now you will make use of the `LinkOptions` variable you defined above in determining which link options are applied to the target executable.

1. If you are not already on the **Advanced** page of the Configuration section, switch to that page.
2. In the **Details** view, position the mouse pointer over the shared library option that currently precedes the target object.
3. Click mouse button 2, and select **Remove**. This removes the shared library option from the target specifier.
4. Select the **Options** page of the Configuration section.
5. Select the target object in the **Source and Targets** view.
6. In the **Change Options** view, look for the **Option Variables** entry field near the bottom. In this field, enter `LinkOptions`, the variable you previously defined. Click **Add** below this field, then on **Apply** at the bottom of the view.

Rebuild the project once again. Now, whenever you want to switch between a shared-library development version, and a nonshared-library production version, you can go to a source view of your configuration, change the value of `ReadyToShip` from `false` to `true` or from `true` to `false`, and rebuild.

This part of the exercise described ways to customize your configuration file. If you use the SmartGuides to build your project, you will probably not need to do this kind of customization for simple projects such as the one in this exercise. Simply changing the appropriate link option in a Change Options view of the target object would accomplish the same result. However, for larger projects, the ability to customize your configuration is important. See the Configuration Files section in the online Concepts help for further details.

You can proceed to:

- "Part 7. Optimizing Your Configuration", if you want to use the Configuration Optimizer to improve the organization of your configuration

  or"Part 8. Defining the View Function for Reviewers" on page 87, if you want to finish coding the CGI portion.

- "Develop a Graphical User Interface from a Visual Part" on page 103, if you want to try developing an application using the Visual Builder and the IDE.

Part 7. Optimizing Your Configuration

You can use the VisualAge C++ configuration optimizer to analyze your configuration and modify it so that build performance is improved.

When VisualAge C++ optimizes your configuration, it scans the entire configuration and the codestore database, so that it can add included files directly into your configuration, when this action would not modify the meaning of your program. It can also change or add options for particular files. Although it may make your configuration file seem larger and more complex, optimization should speed later rebuilds by better managing dependencies between files.

This section should take you approximately 10 minutes to complete. In it, you will learn to use the Configuration Optimizer.

You should have the following source files in a directory on your system.

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program
- funcs.cpp - Auxiliary functions used by user classes
- user.hpp - The class declarations for the user classes
- user.cpp - The function implementations for the user classes.

Optimize Your Configuration

To optimize, select **Tools - Optimize Configuration** from the title bar menu.

The IDE inserts the statement `tool"config_opt"` in your .icc file, then rebuilds your project (because the configuration has changed). This statement causes the configuration optimizer extensions to be loaded and added to the work queue for the rebuild operation. The configuration optimizer gets control only if the build completes successfully. After the project has rebuilt, the configuration optimizer analyzes the codestore for the following information:

- Source files that are included in your project but that are not mentioned in the .icc file (for example, files included using the #include preprocessor directive within a source file)
- .inl files that define inline member functions outside their class declaration
- Options that were applied to all files in an include hierarchy, but need not be

The configuration optimizer adds the required files to your configuration, removes files that are not needed, and changes the scope of certain options. If the optimization results in changes to the configuration, the optimizer initiates a second rebuild operation.

For a small project, optimizing your configuration may not have a noticeable impact on build performance. However, for a production application, particularly one using IBM Open Class Library classes extensively, build times may be reduced if you optimize your configuration periodically.

The configuration file shown in the **Prerequisites** sections of the following parts has not been optimized, for simplicity. You can, however, optimize your configuration at any time later in this exercise.

You can proceed to:

- "Part 8. Defining the View Function for Reviewers", if you want to finish coding the CGI portion

  or

- "Develop a Graphical User Interface from a Visual Part" on page 103, if you want to try developing an application using the Visual Builder and the IDE.

Part 8. Defining the View Function for Reviewers

In this part, you will build on your existing knowledge of the VisualAge C++ IDE by repeating many of the editing, building, and navigation tasks you already learned in previous parts. You will add a new member function to the Reviewer class to support viewing documents that contain comments. You will also create two classes to support adding comments to documents. Finally, you will modify the Reviewer::View function definition so that it shows a version of an HTML file containing reviewers' comments.

This section should take you between 30 minutes and 2 hours to complete. If you are simply interested in using the review tool, the section will take little time. If you want to understand CGI programming, and in particular the design of the review tool, the section will take longer.

Because you already have considerable experience adding code to functions, rebuilding your project, and recovering from build errors, this section contains less guidance than prior sections have done.

Remember, where necessary, to change any path or executable file names to the names required for your system.

You should have the following source files in a directory on your system.

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program
- funcs.cpp - Auxiliary functions used by user classes
- user.hpp - The class declarations for the user classes
- user.cpp - The function implementations for the user classes

Redefine Functions

The next few sections provide code that you can add to your project. Under the heading for each class or member function to be added or changed in this section, you will find a brief description, followed by the required code. To replace a function definition in your source code with a new one from this part, follow these steps:

1. In the online version of this section, select the replacement text for the function in your browser and copy it to the system clipboard.
2. In a source view in the IDE, select the original function definition (*not* its in-class declaration), and paste the clipboard contents so that they overwrite the prior definition. (If the original function is defined inline within its class, change the definition to a declaration, and add the new definition.)
3. After several operations, rebuild your project to ensure that you copied and pasted correctly.

Move Common Code from Reader::View

Implementation Details

Currently, the Reader::View member function only adds the necessary CGI call information to all href links in a viewed document. This code is useful not only for allowing readers to view a file, but for allowing reviewers and authors to view or add comments to a file. Therefore it would be a good idea to move this code into a separate function.You will move the code from Reader::View into a new User class member function, LinkFix, and call that function from Reader::View.

In a Source view, add a public declaration for LinkFix to the User class declaration in user.hpp:

```
IString LinkFix();
```

Also, in user.cpp, change the function name of Reader::View to User::LinkFix. Do this by deleting the old function signature and typing in the new one. Then add the following new definition for Reader::View:

```
IString Reader::View() {
   return LinkFix();
}
```

CommentType and CommentSection Classes

Implementation Details

The first user function you will implement is User::View. However, before you begin, you need to implement some support classes for viewing, because a reviewer's view of a document is more complex than a reader's view. Unlike a reader, a reviewer can view comments, and can add new comments at specific points. The CommentSection class is used to hold sections of the document in which comments can be added, while the CommentType class is used to hold the comments. In this exercise, you will add support for comments only at the start of each paragraph and list item. If you want, you can later add code of your own to handle other insertion points.

Once these classes are defined, you will declare an array of CommentSection elements in Reviewer::View. Element zero will contain all text up to the first paragraph or list item tag, and each subsequent element will contain text following the tag of the prior element, up to and including the next paragraph or list item tag. The CommentSection class will define a Print() operator that returns an IString consisting of the text up to the tag, the numbered hypertext link that allows comment entry, any comments that may exist for that entry, and the text up to the next tag.

Create the files comment.hpp and comment.cpp and add them to the part of your configuration that contains other source files that you have defined. Then add the following code to them:

Class declarations and inline members (comment.hpp):

```
class CommentType {
   IString UserId,      // ID of comment creator
           Date,        // Date comment was created
           Time,        // Time comment was created
           Text;        // Text of comment
   int LineNum,         // Line to which comment applies
       CommentNum;      // Comment number for this line
   Request* Req;        // Pointer to the current request
   Boolean Completed;   // Whether the author has completed the comment
   public:
      // Set methods
      void SetUser(IString U) {UserId=U;}
      void SetDate(IString D) {Date=D;}
      void SetTime(IString U) {Time=U;}
      void SetText(IString U) {Text=U;}
      void SetReq(Request* R) {Req=R;}
      void SetLineNum(int L) {LineNum=L;}
      void SetCommentNum(int L) {CommentNum=L;}
      void SetCompleted(Boolean T) {Completed=T;}
      // Constructors and print function
      CommentType() : LineNum(0), CommentNum(0), Req(0) {}
      CommentType(int Ln, int Ct, Request* R) :
         LineNum(Ln), CommentNum(Ct), Req(R) {}
      IString Print(Boolean CompleteLink);
   };
class CommentSection {
   protected:
      IString Tag;             // Tag for this section (eg. <p>, <li>)
      IString Text;            // HTML Text following tag
      Boolean ShowInsert;      // Whether to show the numbered link for adding comments
      Boolean CompleteLink;    // Whether to show the "Complete" link (used for authors)
      CommentType* Comment;    // Each section can have up to 16 comments
      int Count;               // Count of comments
      Request* Req;            // Pointer to the request
      int Number;              // Number of this comment section (ie. array index)
   public:
      // Constructor
      CommentSection(): Tag(""),
                        Comment(new CommentType[16]),
                        Text(""),
                        Count(-1),
                        Number(0),
                        Req(0),
                        CompleteLink(false),
                        ShowInsert(false) {}
      // Set methods
      void SetTag(IString Tg) {Tag=Tg; }
      void SetComment(IString Text, IString User,
                      IString Date, IString Time,
                      Boolean Completed);
      void SetText(IString Tx) { Text=Tx; }
      void SetReq(Request* Rq) {Req=Rq;}
      void SetNumber(int Num) {Number=Num;}
      void SetCompleteLink(Boolean tf) {CompleteLink=tf;}
      void SetShowInsert() {ShowInsert=true;}
      Boolean IncrementCount() {
         if (Count<16) {Count++; return true; }
         return false;
      }
      IString Print();
};
```

Non-Inline Function Definitions (comment.cpp):

```
// Print this comment. Each comment is contained in a one-cell
// table (so that a border is shown). If the comment is completed,
// show that it is completed. Otherwise, if CompleteLink is true
// (and this is set only in the Author override of View), show
// a link allowing the comment to be completed.
IString CommentType::Print(Boolean CompleteLink) {
   // Print the start of the table
   IString Result=
      "<br><table border=1><tr><td>";
   // Print the User ID of the user who created the comment
   // Underscores in User ID should become spaces
   Result+="Created by: <font color=\"#005555\">"
         +UserId.change("_"," ")
         +"</font>" ;
   // Print the Date and time of comment creation
   Result+=" on: <font color=\"#005555\">"
         +Date
         +"</font> at <font color=\"#005555\">"
         +Time
         +"</font></td></tr>";
   // On a new row, print the comment text
   Result+="<tr><td>"
         +Text
         +"</td></tr>\n";
   // Indicate that comment is complete
      if (Completed)
         Result+="<tr><td><font color=\"#FF0033\">Completed</font></td></tr>";
      // Otherwise, add link to complete it if requested by caller
      else if (CompleteLink)
         Result+="<tr><td><a href=\"review.exe?Op=Complete+key="
               +Req->GetKey()
               +"+file="+Req->GetFilename()
               +"+line="+IString(LineNum)
               +"+cmt="+IString(CommentNum)
               +"\">Mark as Completed</a></td></tr>";
   Result+="</table>\n";
   return Result;
}
// Set method for creating a comment within this comment section
// Only creates the comment if it is one of the first 16 comments
// as only 16 comments are allowed per paragraph or list item.
void CommentSection::SetComment(
       IString Text, IString User,
       IString Date, IString Time,
       Boolean Completed) {
   if (Count>=0 && Count<16) {
       Comment[Count].SetText(Text);
       Comment[Count].SetUser(User);
       Comment[Count].SetDate(Date);
       Comment[Count].SetTime(Time);
       Comment[Count].SetReq(Req);
       Comment[Count].SetCompleted(Completed);
       Comment[Count].SetLineNum(Number);
       Comment[Count].SetCommentNum(Count+1);
   }
}
// CommentSection::Print() prints the tag for the section;
// if the section is not section 0 (everything up to the first
// paragraph or list item tag) it prints a hyperlinked number
// that allows comment insertion;
// if there are any comments for this section, it calls
// CommentType::Print() for each of them;
// finally, it prints its text, that is, the original HTML up
// to the next paragraph or list item tag.
IString CommentSection::Print() {
   IString SNumber=Number;
   IString Result=Tag;
```

```
if (Number>0)
   Result+="<a href=\""+EXE+"?Op=ShowAdd+key="+Req->GetKey()
         +"+file="+Req->GetFilename()
         +"+line="+SNumber+"#Ct"+SNumber+"\">"+SNumber+"</a> ";
for (int cxc=0;cxc<=Count;cxc++)
   Result+=Comment[cxc].Print(CompleteLink);
Result+="<a name=\"Ct"+SNumber+"\">";
Result+=Text;
return Result;
```

Reviewer::CreateSections

Implementation Details

The CommentSection and CommentType classes do almost all of the work of handling the sections of each HTML file for you. The View function only needs to read the HTML file requested, break it up into sections, and print the sections.

Comments are stored in a file with the same name as the HTML file but in a separate directory, defined by the macro CommentRoot.

Each comment in a comment file starts with the line "<cmt>", followed by five words:

Word	Meaning	Example
1	Line comment applies to	"4"
2	Completed or not	"u" or "c"
3	User ID of creator	"RGREEN" or "SBEHM"
4	Date of creation	"12/04/1997"
5	Time of creation	"12:23"

Subsequent lines are text of the comment. The comment ends with the line "<cmt>".

Place a declaration for this macro in your macros.hpp source file. Change the path shown below to the path where you want to store comment files on your system.

```
#define CommentRoot IString("g:\\vatutor\\comments")
```

Remember, you can edit macros from the Macros page of the Project.

The code you used here for creating sections will be used later by the Author version of View and by other member functions of the Reviewer and Author classes. To allow common access to this code, you should define a separate member function of Reviewer, which breaks up the HTML file into these sections. Place the function declaration in user.hpp and the implementation in user.cpp. As well, in user.hpp add a declaration for an array of comment sections, and a counter, CurCs, to keep track of the count of these sections, as shown below.

Declaration (user.hpp)

```
// Place these declarations inside the Reviewer class declaration,
// with protected access:
protected:
   void CreateSections(IString& Content, Boolean CompleteLink=false);
   // Declare an array of comment sections
   CommentSection* Section;
   int CurCs;
```

Definition (user.cpp)

```
void Reviewer::CreateSections(IString& Result, Boolean CompleteLink) {
    // Result contains the HTML file with href links already fixed
    // The CompleteLink argument tells CreateSections whether to show
    // a link authorizing completion of the comment (Author::View sets
    // CompleteLink, while Reviewer::View does not)
    // Create a lowercase copy for tag searching
    IString result=IString::lowerCase(Result);
    // Create a target array of 256 CommentSection objects.
    // Track total used so array can be expanded later if more than
    // 256 sections are found in document.
    // CurCs is declared in class.
    CurCs=0;
    int MaxCs=256;
    // Section is a pointer to an array of CommentSection elements;
    // initialize it now.
    Section=new CommentSection[MaxCs];
    // Search for next paragraph or list item
    // MinNonZero() will be defined in func.cpp
    int Next=MinNonZero(
        NextTag(result,"<p"),
        NextTag(result,"<li"));
    // If no such tag is found, set next to the end of the file,
    // so that the first section becomes the entire file.
    // Return when done, as no more sections need be created.
    if (Next==0) {
       Section[CurCs].SetTag(Result);
       Section[CurCs].SetReq(Req);
       CurCs++;
       return;
    } else {
    // Otherwise, set tag and comment number, and continue
       Section[CurCs].SetTag(Result.subString(1,Next-1));
       Section[CurCs].SetReq(Req);
       Section[CurCs].SetNumber(CurCs);
       CurCs++;
    }
    // Keep adding more sections
    while (Next>0) {
       // Find the end of the tag and set Tag to it
       int eTag=result.indexOf(">",Next);
       IString Tag=Result.subString(Next,eTag-Next+1);
       // Look for next paragraph or list item
       int NextNext=MinNonZero(
          NextTag(result,"<p",Next+1),
          NextTag(result,"<li",Next+1));
       IString Text="";
       // Set section to rest of text if no next tag found,
       // otherwise set text to rest of this section
       if (NextNext==0)
          Text=Result.subString(eTag+1);
       else
          Text=Result.subString(eTag+1,NextNext-eTag-1);
       // Expand the Section array if it is too small
       if (CurCs>=MaxCs) {
          MaxCs+=1;
          MaxCs*=1.25;
          CommentSection* NewCs=new CommentSection[MaxCs];
          for (int cs=0;cs<=CurCs;cs++)
             NewCs[cs]=Section[cs];
          delete[] Section;
          Section=NewCs;
          }
       // Create the CommentSection
       Section[CurCs].SetNumber(CurCs);
       Section[CurCs].SetTag(Tag);
       Section[CurCs].SetReq(Req);
       Section[CurCs].SetText(Text);
       Section[CurCs].SetCompleteLink(CompleteLink);
       CurCs++;
       Next=NextNext;
    } // End of While
    // Open the comment file and read in any comments
    // Comments are delimited by <cmt> and </cmt>
    ifstream CommentFile(Req->GetCommentFn());
    while (CommentFile) {
       IString CommentLine=IString::lineFrom(CommentFile);
       Boolean Completed=false;
       if (CommentLine.isLike("<cmt>*")) {
          int CmtNum=CommentLine.word(2).asInt();
          if (CommentLine.word(3)=="c") Completed=true;
          IString User=CommentLine.word(4);
          IString Date=CommentLine.word(5);
          IString Time=CommentLine.word(6);
          IString Text=IString::lineFrom(CommentFile);
          IString NextLine=IString::lineFrom(CommentFile);
          while (CommentFile && NextLine!="</cmt>") {
             Text+=NextLine+"\n";
             NextLine=IString::lineFrom(CommentFile);
          }
          // Add a comment type to this comment section if the comment is valid
          if (Text!="" && CmtNum>0 && CmtNum<Cur
Cs && Section[CmtNum].IncrementCount())
             Section[CmtNum].SetComment(Text,User,Date,Time,Completed);
       } // end outer if
    } // end while
 } // end function
```

Reviewer::View

The View function contains very little code of its own; most of this code has been moved out to the CommentType and CommentSection function and to the Reviewer::CreateSections function. Change the definition of Reviewer::View to the following:

```
IString Reviewer::View() {
    // 1. Get the file contents
    IString Result=LinkFix();
    if (Result=="")
      return Error("Could Not Read File",
         "The file you requested, "+Req->GetFilename()
        +", could not be read, or was empty.");
    // 2. Initialize the CommentSection Array.
    CreateSections(Result, false);
    // 3. Print the CommentSections
    Result="";
    for (int cs=0;cs<CurCs;cs++) Result+=Section[cs].Print();
    return Result;
    }
```

NextTag and MinNonZero (funcs.cpp)

Implementation Details: The NextTag function searches for the next occurrence of a tag in a string. Because a tag can have attributes, NextTag looks for the first match that contains the opening brace and name of the tag followed by a closing brace, space, or new line. The MinNonZero function, which is overloaded for two and three arguments, returns the lowest nonzero value of a set of nonnegative values.

Add the following definition for these two functions to funcs.cpp:

```
int NextTag(IString String, IString Tag, int StartPos=1) {
   return MinNonZero(
      String.indexOf(Tag+">",StartPos),
      String.indexOf(Tag+" ",StartPos),
      String.indexOf(Tag+"\n",StartPos));
   }
int MinNonZero(int a, int b, int c=0) {
   if (c>0) return MinNonZero(a,MinNonZero(b,c));
   if (a==0) return b;
   if (b==0) return a;
   if (a>b) return b;
   return a;
   }
```

Request::GetCommentFn

This function returns the name of the comment file, which is determined from the main HTML file name and the comment directory. Add the function's declaration as a public member of Request, in request.hpp; add its definition in request.cpp.

```
// In request.hpp, within public part of Request class declaration:
IString GetCommentFn();
// In request.cpp
IString Request::GetCommentFn() {
    IString CommentFn=Filename;
    CommentFn.change("/",slash);
    if (CommentFn.subString(1,1)!=slash)       // Add a leading slash
       CommentFn=slash+CommentFn;
    // Add path to start of file.
    // CommentRoot is defined in macros.hpp
    CommentFn=CommentRoot+CommentFn;
    return CommentFn;
 }
```

Test the New Implementation of Reviewer::View

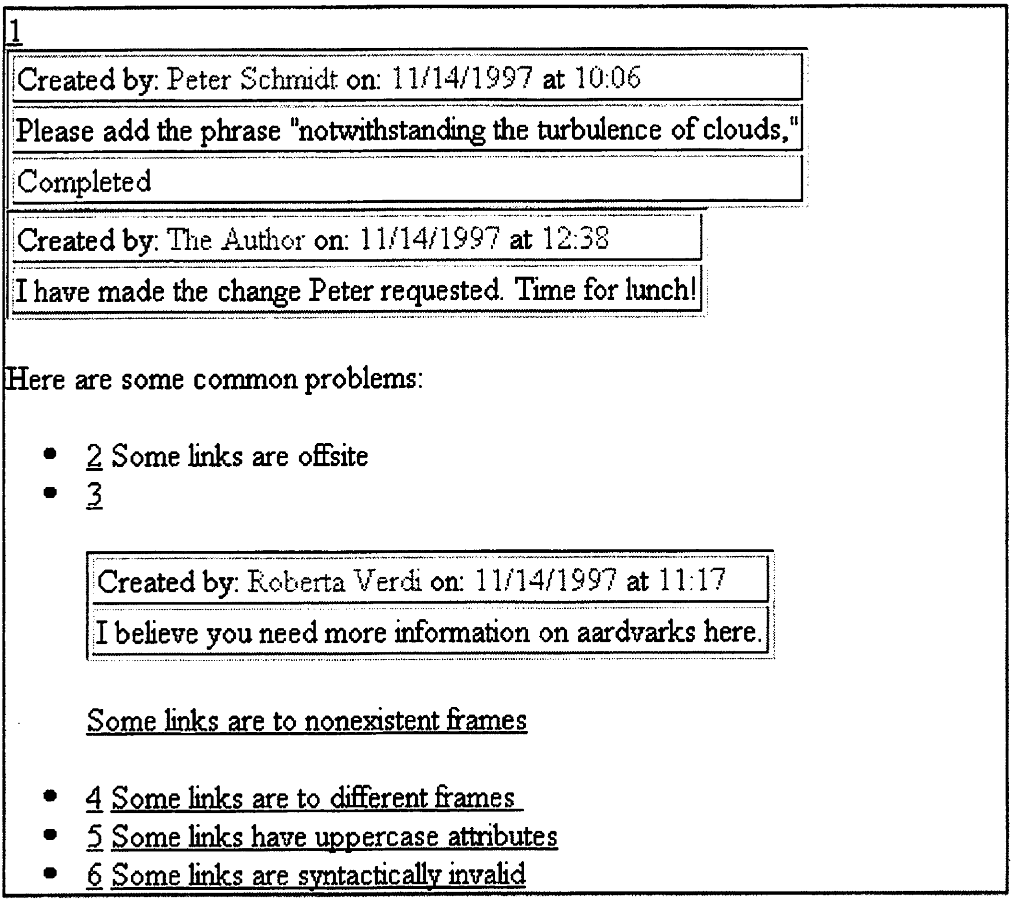
You have not yet implemented the ShowAdd or Add functions, so you cannot add comments to test whether the View function is working properly. You can, however, test View by copying the following text and placing it in a file in your comment root directory. Give that file the same name as an HTML file in your web root directory, for example, index.html:

```
<cmt> 1 c Peter_Schmidt 11/14/1997  10:06
Please add the phrase "notwithstanding the turbulence of clouds,"
</cmt>
<cmt> 3 u Roberta_Verdi 11/14/1997  11:17
I believe you need more information on aardvarks here.
</cmt>
<cmt> 1 u The_Author 11/14/1997  12:38
I have made the change Peter requested. Time for lunch!
</cmt>
```

Try viewing the HTML file using your CGI. Enter this URL in your browser:

http://localhost:49213/cgi-bin/review.exe?op=View+key=reviewer+file=index.html

The text should look something like the following:



1

Created by: Peter Schmidt on: 11/14/1997 at 10:06
Please add the phrase "notwithstanding the turbulence of clouds,"
Completed

Created by: The Author on: 11/14/1997 at 12:38
I have made the change Peter requested. Time for lunch!

Here are some common problems:

- 2 Some links are offsite
- 3

Created by: Roberta Verdi on: 11/14/1997 at 11:17
I believe you need more information on aardvarks here.

Some links are to nonexistent frames

- 4 Some links are to different frames
- 5 Some links have uppercase attributes
- 6 Some links are syntactically invalid

Now that you have implemented Reviewer::View, the remaining functions are relatively simple.

You can proceed to "Part 9. Defining the Remaining User Functions", if you want to finish coding the CGI portion. Or "Develop a Graphical User Interface from a Visual Part" on page 103, if you want to try developing an application using the Visual Builder and the IDE.

Part 9. Defining the Remaining User Functions

In this section you will implement the ShowAdd and Add functions of the Reviewer class, which respectively display a form for adding a comment, and add that comment. You will also implement the Complete function of the Author class, which marks a comment as completed.

This section should take you approximately 30 minutes to complete. It does not introduce any new IDE concepts or tasks. You can use it to strengthen your understanding of the IDE, and you can complete it if you want to have a working web-based review tool which you can later refine and enhance to suit your needs. If you do not need such a tool, you can proceed directly to Develop a Graphical User Interface from a Visual Part.

You should have the following source files in a directory on your system.

- review.icc - The configuration file for your project
- main.cpp - The source code for your main function
- request.hpp - The class declaration for the Request class
- request.cpp - The function implementations for the Request class
- macros.hpp - Macros defined for your program
- funcs.cpp - Auxiliary functions used by user classes
- user.hpp - The class declarations for the user classes
- user.cpp - The function implementations for the user classes
- comment.cpp - The class declarations and incline members
- comment.hpp - Non-inline function definitions
- users.dat - A plain text file listing three levels of users.

Complete the Function Implementations

Implementation Details: The Reviewer::View function already does most of what you need for the ShowAdd function. You will begin by copying code from that function into Reviewer::ShowAdd. The only change you will make will be to insert the comment form at the appropriate location in the file. The Author::Complete function is also very similar to Reviewer::View; it finds the incomplete comment in the comment file, changes its status to completed, then shows a new view of the requested file with its comments.

Follow these steps to complete the function implementations, all within user.cpp:

1. Type the body of the Reviewer::View function into Reviewer::ShowAdd. The Reviewer::ShowAdd function should now read as follows:

```
IString Reviewer::ShowAdd() {
    // 1. Get the file contents
    IString Result=LinkFix();
    if (Result=="")
      return Error("Could Not Read File",
         "The file you requested, "+Req->GetFilename()
        +", could not be read, or was empty.");
    // 2. Initialize the CommentSection Array.
    CreateSections(Result, false);
    // 3. Print the CommentSections
    Result="";
    for (int cs=0;cs<CurCs;cs++) Result+=Section[cs].Print();
    return Result;
 }
```

2. Add a statement before the call to LinkFix to determine the line number where the user wants to add a comment. Use the Request::GetValue function to determine this value:

```
int LineNum=Req->GetValue("line").asInt();
```

3. Add the following after the call to CreateSections:

```
if (CurCs<LineNum)
   return Error("Cannot Insert Comment",
      "A section for inserting comment "
      +IString(LineNum)
      +" cannot be found.");
Section[LineNum].SetShowInsert();
```

4. To insert the Add Comment form, add the following to CommentSection::Print() immediately before "Result+=Text;"

```
   if (ShowInsert) {
       Result+="<table border=1><tr><td>Add a Comment<br>\n";
       Result+="<form method=post action=\""+EXE+"\">\n";
       Result+="<input type=hidden name=key value="+Req->GetKey()+">\n";
       Result+="<input type=hidden name=file value="+Req->GetFilename()+">\n";
       Result+="<input type=hidden name=op value=add>\n";
       Result+="<input type=hidden name=line value="+SNumber+">\n";
       Result+="<textarea name=text rows=10 cols=60>\n</textarea><br>\n";
       Result+="<input type=submit name=Submit value=Submit></form></td></tr></table>";
   }
```

5. Rebuild your project. Access the HTML file by entering the following text in your browser's URL field:

   http://localhost:49213/cgi-bin/review.exe?op=View+key=reviewer+file=index.html

   Try adding a comment in this file, by clicking on a numbered hypertext link. A form for adding your comment should appear. Enter some text in the comment form and click **Submit**. You should get an error message saying that the Reviewer version of the Add function is not yet implemented.

6. Next, redefine the Add function by deleting the existing definition from user.cpp and entering the definition below.

Implementation Details: The Add function first tries to open the comment file in append mode. It tries up to 50 times to do so, waiting 50 milliseconds after each try, in case another process is also trying to modify the comment file at the same time (for example, two users adding comments at once). If the comment file still cannot be opened, an error is returned. If it can be opened, the required markup and information for the comment are appended, including the current date (created from the IDate static member function today()) and time (created from ITime::now).

```
IString Reviewer::Add() {
   // Open file in append mode
   ofstream CmtFile(Req->GetCommentFn(),ios::app);
   int tries=50;
   while (!CmtFile && tries>0) { // Try up to 50 times
      CmtFile.close();
      IThread::current().sleep(50);
      tries--;
      CmtFile.open(Req->GetCommentFn(),ios::app);
   }
   if (!CmtFile)
      return Error ("Comment File Locked",
   "The comment could not be added. The comment file may be locked by another process.");
   // Write out the markup and comment information
   CmtFile << "<cmt> "
           << Req->GetValue("line")
           << " u "
           << GetUserId() << " "
           << IDate::today().asString("%m/%d/%Y ") << " "
           << ITime::now().asString("%H:%M\n");
   // Read the entire request string, then cut it down to contain only the text
   // of the comment, which spans from "text=" to "Submit=Submit"
   // Submit=Submit comes from the user pressing the Submit button.
   IString Text=Req->GetReqString();
   int texteq=Text.indexOf("text=");
   // Exclude up to and including "text="
   Text=Text.subString(texteq+5);
   int sub=Text.indexOf("Submit=Submit");
   // Exclude from Submit=Submit to end
   if (sub>0) Text=Text.subString(1,sub-1);
   if (Text=="") Text="No text in comment";
   CmtFile << Text << "\n" << "</cmt>" << endl;
   CmtFile.close();
   // Finally, call View.
   return View();
}
```

7. Add the following three header files, either to a source directive in your configuration, or using #include <*iheader* .hpp> directives at the top of user.cpp:

   - ithread.hpp
   - itime.hpp
   - idate.hpp

8. Rebuild the project. You will probably encounter one error: a reference to an undefined function, GetUserId(). Add the following as a public function inside the declaration for the Reviewer class:

   ```
   IString GetUserId() { return UserId; }
   ```

   Currently, UserId is a private member of the User class, so if you rebuilt now you would get an access error. Move the declaration for UserId from the private to the protected section of the User class declaration (right after the Request* object).

   You also need to set the value of UserId, which is currently not set anywhere. Add a SetUserId() function to the public section of the User class:

   ```
   void SetUserId(IString u) {UserId=u;}
   ```

   Then set the user ID by adding the following in your main function (in main.cpp), after the switch block:

   ```
   U->SetUserId(UserLine.word(3));
   ```

The GetUserId function is defined in Reviewer, because it is only called from Reviewer::Add (a default user does not have a user ID). The SetUserId function is defined in User, because it is called by the polymorphic User* object in main.

9. Rebuild the project. Before you try out the CGI, you need to modify the users.dat file by adding a user ID or name after each user's key and authority level. Edit the file and change it, for example, to the following. The file is located in the path specified by the UserFilepath macro.

   ```
   reader 1 Arthur_Green
   reviewer 2 Roland_Burgess
   author 3 Kay_Smith
   ```

10. Try adding a comment again. This time, you should see the text of your comment along with a user name, the date, and time, imbedded in the text of the document. The URL to view your file is:

    http://localhost:49213/cgi-bin/review.exe?op=View+key=reviewer+file=index.html

11. Implement the Author::View function. It is almost identical to Reviewer::View. The only difference is that the second argument to the call to CreateSections is set to **true** instead of **false**. This argument to CreateSections is then passed on to each CommentSection and CommentType object, so that the CommentType::Print function prints a link to allow the author to complete the comment. Mark the body of the Reviewer::View function (use Alt+L at the top and bottom lines of the body) and copy it into Author::View (use Alt+C); change the call to CreateSections as noted.

12. The final function to implement is Author::Complete.

Implementation Details: A request to complete a comment should contain the file, line, and comment number of the comment to complete. These arguments can be parsed out by the Request::GetValue function. The Complete function then opens the file in binary mode, reads in characters until it finds the byte whose value of 'u' (for "Unhandled") needs to be changed to 'c' (for "Completed"), writes out only that byte, and closes the file. This approach minimizes file I/O by reading only the bytes than required to locate the comment to complete, and by writing only a single byte of output.

Change the definition of Author::Complete in user.cpp to the following:

```
IString Author::Complete() {
// Open the file in binary read-write mode. If the file doesn't exist,
// there are no comments.
   FILE* stream=fopen(Req->GetCommentFn(),"rb+");
   if (!stream)
      return Error("Could Not Complete Comment",
        "The comment could not be completed because the comment file, "
        +Req->GetCommentFn()
        +", could not be found.");
   char* Buffer=new char[256];
   IString CommentFile="";
   int CommentNum=Req->GetValue("cmt").asInt();
   int LineNum=Req->GetValue("line").asInt();
   IString ThisCmt="<cmt> "+IString(LineNum)+" ";
   int bytesread=fread(Buffer,sizeof(char),256,stream);
   CommentFile+=Buffer;
```

```
   // Read in the file 256 bytes at a time.
   // Continue reading until the number of comments for this line
   // is at least equal to the comment number the Complete applies to
   while (bytesread>0 && CommentFile.occurrencesOf(ThisCmt)<CommentNum) {
      bytesread=fread(Buffer,sizeof(char),256,stream);
      CommentFile+=Buffer;
   }
   // Find the comment position of the nth comment for this line
   int lastpos=0;
   int thispos=CommentFile.indexOf(ThisCmt);
   for (int i=1;i<CommentNum;i++) {
      lastpos=thispos;
      thispos=CommentFile.indexOf(ThisCmt,thispos+1);
   }
   // If that comment's completion code is 'u' change it to 'c'
   if (CommentFile[thispos+ThisCmt.length()]=='u') {
      fseek(stream, thispos+ThisCmt.length()-1,SEEK_SET);
      fputc('c',stream);
      fclose(stream);
   }
   // If you like, add an error return here that returns a
   // message stating that the comment was already completed.
   return View();
}
```

13. Rebuild the application. You may find that SEEK_SET is undefined. If so, add the directive #include <stdio.h> at the top of user.cpp and rebuild again.
14. Try completing a comment. You will need to use an author user ID to do so; try accessing the file by entering the following text in your browser's URL field:

    http://localhost:49213/cgi-bin/review.exe?op=View+key=author+file=index.html

The CGI portion of your program is now complete. You can now review your own web documents by adding the documents to your WebRoot directory tree; you can let others review them by adding their user IDs, passwords, and authority levels to your users file. This program is still only a prototype. You may want to enhance the error handling and the types of HTML markup to which it allows comments to be added, and you may want to enable it to work with first and last names.

If you wanted to ship this program to customers now, you would first need to change the configuration so that the version produced used the static version of the run-time libraries. If you completed Part 6. Managing Configuration Files, you have an easy way of doing so - change the value of ReadyToShip to "true" and rebuild. However, there is still some work to do before you have a program you can present to users who will be installing the CGI to run from their web servers. Your users probably do not want to edit a user management file directly; you need to create a graphical user interface (GUI) program that runs locally, not through a browser, so that the administrator of the machine running the CGI can create and manage user IDs. In the next sections, you will learn to use the Visual Builder component of VisualAge C++, along with the IDE, to create the GUI interface and the user management code.

You have now completed the CGI portion of the VisualAge C++ tutorial.

You can proceed from here to the the integrated Visual Builder component of VisualAge C++, in which you will create a graphical user interface to administer the users for your review tool, or browse the online help from the VisualAge C++ Help Home Page, or the VisualAge C++ Samples home page, if you want to browse the samples, copy a sample directory to your own working directory, or try launching a sample in the IDE.

Develop a Graphical User Interface from a Visual Part

In this section of the tutorial, you will use VisualAge C++ to create a user interface and the underlying code to administer a list of users. This will give you practice with the Integrated Development Environment and its Visual Builder component. If you already implemented the review tool in the previous sections, you can use this user interface to administer the users for that web. If not, this section at the least will help you learn how to create visual parts and integrate them with your own code within the IDE.

Here is an overview of this section:

Part 1. Creating a Project for a Visual Part
Starting from the VisualAge C++ IDE, you will create a new project.

Part 2. Creating Parts in the Visual Builder
Without writing any code, you will create two parts: a nonvisual Main part, used for implementing your main function, and a visual part for the user interface for your program.

Part 3.Connecting the Main Part to the User Interface Part
You will make a connection in the Visual Builder from the Main part to the User Interface part, so that the frame window of your user interface is loaded and displayed when your main function runs, and add both parts to the configuration of your project in the IDE.

Part 4. Adding User Interface Controls to a Visual Part
You will improve the appearance of your user interface by aligning and resizing the various controls.

Part 5. Manipulating the Appearance of User Interface Controls
You will improve the appearance of your user interface by aligning and resizing the various controls, using the Visual Builder's alignment features.

Part 6. Making Simple Connections
You will make your first connection of a visual part to an action, so that selecting the part causes the action to be performed. With a few clicks of your mouse and without typing in any code, you will cause the **Exit** button to close your program.

Part 7. Adding Help Text to User Interface Controls
In this section you will add hover and status-line help for your controls.

Part 8. Making Connections to User-Defined Functions
Some of your push buttons perform actions too complex to implement through a connection to a predefined action. You will connect these buttons to user-defined functions.

Part 9. Defining Functions for Custom Connections
Finally, you will add the necessary code to the user-defined functions you created in Part 8, so that your user interface is fully functional.

Part 1. Creating a Project for a Visual Part

You will create the initial versions of your user interface without writing a single line of code.

Before working with parts in the Visual Builder, we will create an empty project in the IDE. After creating the parts, we will return to the IDE and import the parts.

Create a Project in the IDE

1. Start the IDE. Select the **Create a new project** radio button from the Welcome screen, or pull down the **Project Workbook** menu and select **Create project...**
2. The Project SmartGuide opens.
3. Click **Next** to move to the **Project Configuration** screen, if this is not the first displayed screen.
4. In the **Project Configuration** screen, enter the project's name as `gui.icc`.
5. Specify a directory in which to store the project files. For example, you might enter `/u/myuserid/vatutor`.

   For the remainder of this exercise, wherever you see the path `/u/myuserid/vatutor`, you can substitute the path you specify in this step.

   Click **Next**.
6. In the **Target Type** screen, the default selection is `Executable(EXE)` file. Click **Add Target** to accept this selection.
7. The Target SmartGuide opens. In this SmartGuide you will add a target to your configuration. Click **Next** to get past the opening screen.
8. In the **Target Name** screen enter `gui` as the target name, and select **AIX IOC** from the **Program Type** column below the entry fields. We will not be entering any source files yet, so click **OK** (rather than **Next**). You are returned to the Target Type screen, and gui now appears in the **Current targets** list.
9. Click **Finish** to finish creating the configuration for the new project.
10. A message window will appear and you are asked if you want to open the project in the IDE. Click **Yes**.
11. When you are prompted to build the project, click **No**.

You now have a project in the IDE and you are ready to begin creating parts in the Visual Builder. You can proceed to “Part 2. Creating Parts in the Visual Builder” on page 105 .

Part 2. Creating Parts in the Visual Builder

You have set up an empty project in the IDE which will make use of visual and non-visual parts. You will create the parts in the Visual Builder: one non-visual part will implement your main function, and one visual part will implement the frame window in which users interact with your program.

Create myGui, a Visual Part

1. Start the Visual Builder. If you are using the Common Desktop Environment desktop manager, you can start the Visual Builder by double-clicking on the Visual Builder icon in the VisualAge C++ Application group. You can also type `ivb` on the command line if you have modified your path to include the VisualAge C++directory `/usr/vacpp/bin`.The main Visual Builder window opens.
2. From the **Part** pull-down menu, select **New** and **Visual Part**.
3. In the **Part - New** dialog, type myGui in the **Class name** field. You do not have to enter a file name or a description. Make sure the default base class, `IFrameWindow`, is selected in the **Base class** field.
4. Click **Open**. The Composition Editor opens, and an empty frame window is shown.
5. To produce the source file for this part:
   - click the **Generate Part Source** icon or
   - press **Ctrl+G** or
   - pull down the **File** menu and select **Save and generate** and **Part source**.
6. A message window entitled **Generation results - myGui** opens, and three files are listed:
   - myGui.h
   - myGui.hpp
   - myGui.cpp
7. Close this dialog by pulling down the dialog **File** menu and selecting **Exit**. (You can also press **Ctrl+F3** to close both this window and the Visual Composition Editor window.)

Change the Canvas Style

There is one modification to make to the myGui part you just created. By default, the empty frame window has a 'canvas', or working area inside the window, that is a *multi-cell canvas*. For the purposes of this simple application, it will be easier to work with an ordinary canvas. To make sure you have the right style of canvas, follow these steps:

1. Click inside the frame window. Four square handles appear inside the frame window.
2. Click and hold mouse button 2 and select **Open settings** from the pop-up window.

3. Check the title bar of the Properties dialog: it should say Canvas1 (Canvas) Properties. If it says Canvas1 (Multicell canvas) Properties, you have a multi-cell canvas in the frame window.
4. Click **Cancel**.
   - If you did not have a multi-cell canvas, you can go to "Part 2. Creating Parts in the Visual Builder" on page 105.
   - If you have a multi-cell canvas, press the **Delete** key now, while the canvas is still selected.
5. The inside area of the frame window should now show the message No Client Currently Assigned in the whitespace.
6. Click the **Composers** button in the left column of the Parts Palette.
7. Click the **ICanvas*** part type button in the column on the right.
8. Click anywhere in the whitespace inside the frame window to place the canvas.

Create Non-visual Part, myMain

1. In the Visual Builder main window, pull down the **Part** menu again, and select **New** and **NonVisual Part or Class**.
2. In the **Class name** field, type myMain. You do not have to enter a file name or description, but in the **Base class** field, make sure IVBmain is selected.
3. Click **Open**. The Visual Composition Editor opens again, this time showing a part called VBMain. Save and generate the source for this part as you did in Step 5 (**Ctrl+G**). Again, a message window will list the files that were created.
4. Close only the Generation results message window, leaving the Composition Editor window, with the myMain part open. Now you will open the myGui part into the same workspace.
5. In the Composition Editor, pull down the **Options** menu and select **Add Part** (or press **Ctrl+P**).
6. In the **Add part** dialog, click on the **List...** button to see a list of all the available parts.
7. The **List parts** dialog appears. Type an 'm' in the Search String field to jump to the parts starting with m. Two parts appear: myGui and myMain.
8. Select **myGui** and click **Ok**.
9. In the **Add part** dialog, with myGui* now selected in the **Class name** field, click **OK** again.
10. You are back at the Visual Composition Editor window, and the mouse pointer has become a '+' sign. Click anywhere in the free-form surface (the white space) to place the myGui part. An empty frame window appears on the canvas. The mouse pointer becomes an arrow again.

Now you are ready to connect the interface part (myGui) to the main part (myMain). Go to "Part 2. Creating Parts in the Visual Builder" on page 105.

Part 3. Connect the Main Part to the User Interface Part

You have just finished creating two new parts in the Visual Composition Editor. However, the main part still needs to be connected to the visual part so that the frame window displays when you load the program.

Connect the Parts

Now you will associate the myGui part with the nonvisual main part IVBMain1, so that your user interface is launched when main is called.

1. Select the myGui object, by clicking once on it, then click and hold down mouse button 2.
2. From the pop-up menu select **Connect**.
3. From the cascading menu select **show**. This action will cause the myGui object to be shown when main is called.
4. A dotted line appears; move the mouse until the "star" end of the line is over the free-form surface, outside the myGui framewindow. Click mouse button 1, then select **ready** from the pop-up menu.
5. A green arrow appears, indicating that a connection has been made. To see what the arrow connects, click over it with mouse button 2 and select **Open settings**. A dialog with the title **Event-to-action connection - Settings** appears. In the **Event** list, `ready` is selected, and under the **Action** list, `show` is selected. If your connection does not have these properties, you can change the settings here. Otherwise, click **Cancel** to close the dialog.
6. Now you can generate the source code. Press **Ctrl+G** (or use one of the other methods for generating source code, listed in "Part 2. Creating Parts in the Visual Builder" on page 105).
7. A message window appears, entitled **Generation results for myMain**. Close the message window.

Add New Parts to your VisualAge C++ Project

Before you can test the parts you have just created in the Visual Builder, you will have to add them to the configuration of your project in the IDE.

Switch to the IDE, if you have it open, or open the IDE and open the project you created in "Part 1. Creating a Project for a Visual Part" on page 104, `gui.icc`. Do not build the project yet, as there are no source files. To add the parts:

1. Click on the **Host** tab to browse the files on your system.
2. On the **File System** page, the upper left pane, a **File Tree** view, shows a view of the directory tree. Find the directory you selected as your working directory in the

Visual Builder. In the right pane, the **File** view, you should see a set of files with the names you assigned your parts: *myMain.** and *myGui.**.

3. Of this group of files, which includes files of type .cpp, .vbf, .hpp, and .h, you only need to select the .cpp files. All the necessary header files are #included in the .cpp files, and do not need to be explicitly added here.
4. Hold down the **Ctrl** button and select `myMain.cpp` and `myGui.cpp`.
5. With both of these files highlighted, place the pointer over either file and click on mouse button 2. A pop-up menu appears.
6. From the pop-up menu, select **Add Source to Target Directive**.
7. The **Add to Target** dialog appears, showing that you will be adding these two source files to the instructions for building the target you defined in Part 1, `gui`. Make sure the **Use Relative Path** box is checked, and click **Apply**.
8. To see the change made to your configuration file, switch to the Configuration section by clicking on the **Configuration** tab. Select the Source page to see the contents of `gui.icc`.
9. Under the target portion of the configuration file, you should now see a section of code like this:

   ```
   target "gui"
   {
   source "myMain.cpp",
   "myGui.cpp"
   }
   ```

   This means that the source files `myMain.cpp` and `myGui.cpp` will be used to build the target `gui`. (A number of other files are also listed; these were automatically added by the Targer SmartGuide when you specified the program type `AIX IOC` for your project.) You have added the two parts to your project.

Test the Parts

To test the parts in the IDE, build the project (click the Build button). When the build completes, go to the **Debug** page of the **Project** section. In the **Run Specifications** view in the upper left pane, click the **Run** button. An empty frame window appears in the foreground. Close the frame window.

Now that you have successfully created a simple visual project, it's time to start adding user interface components to it. Proceed to "Part 4. Adding User Interface Controls to a Visual Part".

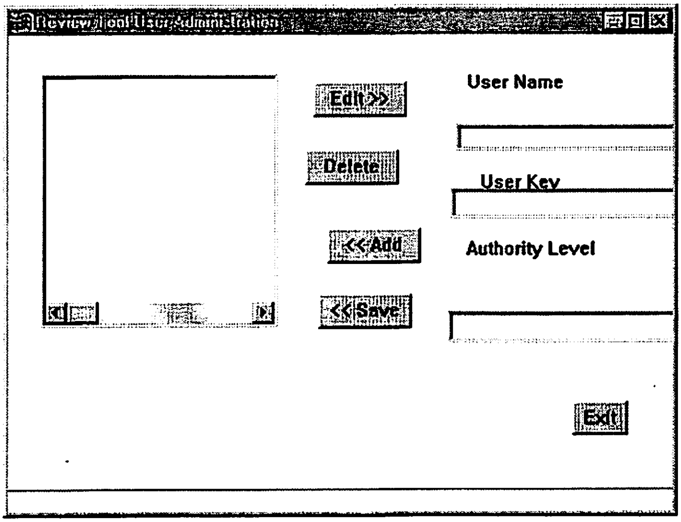
Part 4. Adding User Interface Controls to a Visual Part

Your visual part so far displays only a frame window with the title `FrameWindow`. In this section you will:

- change the title displayed for the visual part
- add the required user interface controls to the frame window:
  - list box

- buttons
- text fields
- entry fields
- spin buttons

When you are finished with this section, your application will look similar to this:



Open the myGui Part in the Visual Composition Editor

You may still have a Composition Editor window open for myMain. You can close it if you wish; it is not needed for this section. From the Visual Builder main window, double-click on myGui in the Visual Parts list. A Composition Editor window opens.

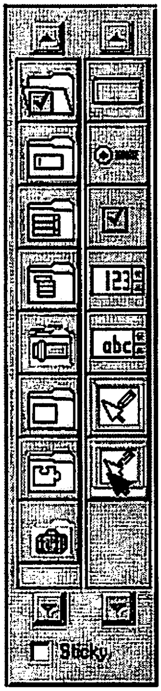
Change the Title of the Frame Window

To change the title that appears in the frame window to something more meaningful than `FrameWindow`:

1. In the Composition Editor, hold down mouse button 2 on the title bar of the frame window. From the pop-up menu, select **Open settings.** A list of properties you can edit apppears. At the bottom of the list, click in the entry field next to **Title**, delete `Frame window`, and type in a name, for example, `User Administration`.
2. To finish the input, press **Shift-Enter**. Click **OK**. You have changed the title of your frame window. Now we will add some user interface controls.

Add Parts

Part are added to your workspace with the parts palette. The parts palette looks like this:



The Sticky checkbox is used whenever you want to drop several types of the same type onto the canvas in succession. This saves you from having to repeatedly click the part type. To turn off sticky parts, deselect the **Sticky** check box.

Generally, you add parts to a canvas in three steps:

1. Select the parts category from the left column of the parts palette.
2. Select the type of part you want from the right column of the parts palette.
3. Click anywhere on the canvas to place the selected part.

Note that if you accidentally add a part you did not want, you can remove it by selecting the part and pressing the Delete key.

(These steps will be listed again). If you accidentally add a part you did not want, you can remove it by selecting the part and pressing the **Delete** key.

Add a List Box

Now you will add a list box to the left side of your window to display the names of existing users.

1. Click the **Lists** category , then click the **IListBox*** part type .
2. Move the mouse pointer just inside the top left corner of the canvas inside the frame window. Click mouse button 1. A list box is added to the canvas.
3. The Composition Editor gives each new object a default name of *ObjectType* plus a number. Because you will be writing code that uses this object, you should rename it from `ListBox1` to `UserList`. Click mouse button 2 to raise a pop-up menu over the list box, and select **Open settings**. In the entry field, change the name to `UserList`.
4. Click **OK** to close the Properties dialog.

Add Buttons

Next, you'll add five buttons:

1. Click the **Buttons** category . Click the **IPushButton*** part type , then click the **Sticky** checkbox at the bottom left.
2. Click four times in a vertical line down the center of the canvas to drop four buttons. These will be the buttons used to edit, delete, add, and save user entries.
3. Click a fifth time just inside the bottom right corner of the canvas. This will be your Exit button.
4. Turn off the **Sticky** check box.

Now change the displayed text for the five buttons to indicate their intended functions.

Now change the displayed text for the five buttons to indicate their intended functions. Double-click on the button, or hold down mouse button 2 and select **Open settings** from the pop-up menu. If a new part is added when you click the mouse button, the Sticky check box needs to be deselected

Change the button text in the **text** entry field as follows, from top to bottom:

- **Edit >>**
- **Delete**
- **<< Add**
- **<< Save**

- **Exit**

The last controls you need are two text input fields for the user's name and key, a spin button field for the user's authority level, and three static text fields describing those input fields.

Add Text Fields

Begin with the text fields. Click the **Data Entry** parts category , and click the **IStaticText*** part type . Turn on the **Sticky** check box if it is turned off. Add three static text objects to the right of each of the top three push buttons. Do not worry about exact placement yet.

Turn off Sticky parts to stop adding static text objects.

Select each of the static text objects you just created, by double-clicking on each object, or by clicking and holding mouse button 2, and selecting **Open settings**. Edit the **text** field in the **StaticText Properties** dialog.

Change the text for the three static text objects as follows:

- **User Name**
- **User Key**
- **Authority Level**

Add Entry Fields

Next, add a text entry field under **User Name** and **User Key**. Click the **IEntryField*** part type (also in the Data Entry parts category), turn on the **Sticky** check box, and click in the space underneath each of these text objects. Turn off the **Sticky** parts check box.

Rename the two entry field parts to UserName and UserKey by clicking mouse button 2 on each object, selecting **Change Name**, and entering the new part name in the **Name Change Request** dialog that appears (you can also do this by editing the **Part name** field from the **Properties** dialog.

Add Spin Buttons

Add one last part, a text spin button for the authority level. Click the **Buttons** category , and the **ITextSpinButton*** type . Drop an ITextSpinButton object under the **Authority Level** text. Rename the spin button to `AuthorityLevel`.

You need to add a set of three text choices to the spin button, and change some settings on it to make the spin button display properly:

1. Click mouse button 2 over the spin button and select **Open settings**.
2. In the **Properties** dialog, scroll down to the **InitialContents** field. Select the entry field beside this field, and a ... button appears. Click this button.
3. Enter the following text on separate lines in the **Array of strings prompter** dialog:

```
Reader
Reviewer
Author
```

4. Click **OK** twice to close the **Properties** dialog.

Try Your Program

The layout of your user interface is a bit messy, but it would still be nice to try out the interface.

Press **Ctrl+G** to generate part source for the part. The Generation results message window appears, showing the myGui.* files that were updated. Close the message window.

Switch to the VisualAge C++ IDE. Open the gui.icc project if you do not already have it open.

If you had the IDE open already, you may see a message window advising you that some files have been changed outside the IDE, and prompting you to reload the files. Click **OK** to reload the files.

You do not need to make any changes because no new files have been created. The list of source files in the configuration should be the same as it was after "Part 3. Connect the Main Part to the User Interface Part" on page 107. Simply click the **Build** button. Once the build has completed, switch to the **Debug** page in the **Project** section, and click the **Run** button, or press **Shift+Ctrl+R**.

If you try clicking a button in the application, nothing happens, because you have not yet made connections between the buttons and actions. An example of an action related to a button would be the closing of the window when the **Exit** button is pressed. You will start making these connections in "Part 6. Making Simple Connections" on page 117 .

You can proceed to "Part 5. Manipulating the Appearance of User Interface Controls" if you want to improve the user interface by properly aligning, sizing, and distributing the objects in it, using the Composition Editor's tools. Or you can proceed to "Part 6. Making Simple Connections" on page 117 if you are already familiar with the Composition Editor's alignment and sizing tools from a previous VisualAge product.

Part 5. Manipulating the Appearance of User Interface Controls

In your program's current interface, the UI controls may be sized, aligned and distributed unevenly, because you dropped them into the canvas by hand, and the push buttons were sized according to their text content rather than a common width.

In this section, you will use some of the Composition Editor's toolbar buttons to make the following changes:

- align objects
- change the width of a set object to match each other
- spread the objects so that they are evenly distributed
- move and resize controls

Resize the UserList List Box

The list box for user names is not currently large enough to display any names. We will make it wider and deeper so that you can view available user names later. There are two ways you can resize any object:

- you can select the object, then drag any corner of it with the mouse
- you can open the settings for the object and enter new dimensions for it.

Select the UserList box, and click on the handle on the bottom right corner. Drag it down until it fills most of the left side of the canvas. You can drag other corners if you like, until you are satisfied with the shape and size of the list box.

Grab the bottom right corner of the UserList list box and drag it down until it fills most of the left side of the canvas. You can drag other corners as well if you like, until you are satisfied with the size and shape of the list box.

Align the Main Push Buttons

You can align several objects so that the left, right, top, or bottom edges of all the objects are aligned. The last selected object stays put, and the others are aligned to it. Before you align the four push buttons, drag the last one (**<<Save**) until it is positioned just to the right of the list box. Do not worry about its vertical placement yet.

Select the first object by clicking mouse button 1. You can use the **Ctrl** key and click moust button 1 the remaining three buttons, ending with the **<<Save** button.

Click the Align Left button from the toolbar. This aligns the left edges of the selected objects to match the left edge of the last selected object, **<<Save**. Do not deselect the buttons yet. You will need to have them all selected in the same order for the next step.

Resize Push Button Widths

Since these four push buttons are all fairly close in width, it would be nice if they were all *exactly* the same width. Click the **Match Width** button while the objects are still selected. This makes the other three buttons the same width as **<< Save**. Like aligning, size-matching actions also use the last selected object as the reference point, or anchor. Do not deselect the buttons yet.

Distribute Push Buttons

The push buttons might look better if the spaces between them were equal. They are already selected in top-to-bottom order; click the **Distribute Vertically** button to make the spaces between them the same. If you changed the selection order when resizing the button widths, the vertical order of the buttons would change to match the order in which you selected them.

The distribution made the gaps between the push buttons equal, but the gaps between them may be a bit wide. This is because distribution spaces the selected objects evenly across the entire height of the canvas (or its width, when you distribute horizontally). You can decrease this vertical spacing by selecting the **Exit** button as well as the others, and distributing vertically again. Notice that when you redistribute the buttons vertically, their horizontal placement does not change.

Distribute the Right-hand Section of the Canvas

Next, you can distribute the GUI elements on the right-hand section of the canvas. Starting at the top of the canvas (with User Name), select the three static text fields, the two entry fields, the spin button field, and the **Exit** push button. Then click **Distribute Vertically**.

Now, align the left edges of all these items except the **Exit** push button. Deselect that push button by holding down **Ctrl** and clicking mouse button 1; then click **Align Left** .

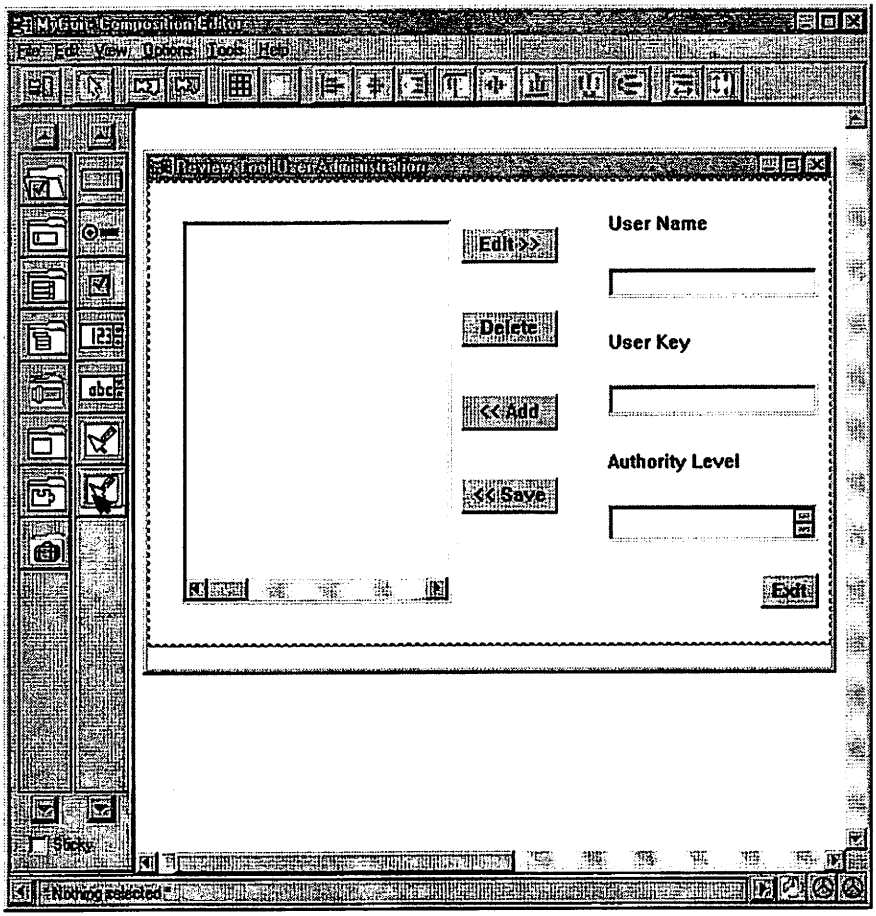
Resize Entry Fields and Spin Button

These controls may run off the right edge of the screen. To resize them, select only one of them, for example the top one, drag its top right corner to the left until it fits within the canvas. Then, select all three controls, *ending* with the one you just resized, and click Match Width.

Align Exit Button with the Other Fields on the Right

Finally, align the right edge of the **Exit** button to the right edge of another right-hand entry field. Select the **Exit** button, then the entry field, then click **Align Right**.

Your user interface should now look something like the following:



Press **Ctrl+G** to regenerate the source code for the parts and save your changes.

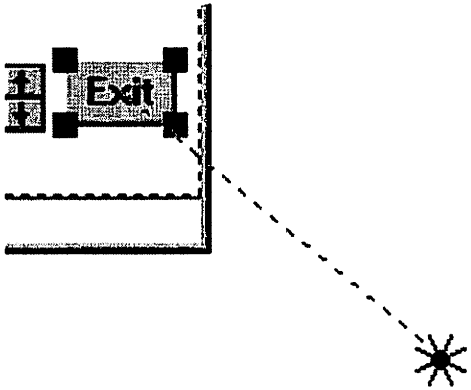
You can now proceed to “Part 6. Making Simple Connections”, in which you will implement some of the connections that will determine the behavior of the user interface.

Part 6. Making Simple Connections

A connection in the Visual Builder is used to associate an event with an action. For example, you can connect the buttonClick event for a push button to a built-in action such as closing the window, or to a member function you define for the visual part. In this section, you will create a simple connection to close the program when the user clicks **Exit**.

1. Select the **Exit** push button and click mouse button 1.

2. From the pop-up menu, select **Connect**.
3. From the cascading menu, select **buttonClickEvent**.
4. A dashed line appears, with one end attached to the **Exit** push button, and a star at the other end:

   This line indicates that you are making a connection associated with the user clicking the **Exit** button. The kind of connection you can create depends on where you drop the star end of the line. Move the star up to the title bar of the frame window and click mouse button 1.
5. A menu appears with the choices available when the buttonClickEvent action for a button is associated with the FrameWindow object. Select **close** so that when the button is clicked, the program closes.
6. A green arrow appears, pointing from the **Exit** button to the titlebar. This means the connection has been made.
7. Try running the project: switch to the IDE, (you will again see a message window prompting you to reload the files: click **OK**) rebuild, and click the **Run** button in the **Run Specifications** view of the project's **Debug** page. The window for your user interface should open, and when you click **Exit**, it should close. The other buttons still do not have any effect because you have not yet associated any actions with them.

You can proceed to:

- "Part 7. Adding Help Text to User Interface Controls" on page 119 if you want to learn how to add hover and info area help to your push buttons and list boxes

  or

- "Part 8. Making Connections to User-Defined Functions" on page 120 if you don't want to add help text to your controls, or already know how.

Part 7. Adding Help Text to User Interface Controls

You can easily add two kinds of help text to a user interface control that supports such help, by entering that text in appropriate fields of the control's Settings dialog. Both kinds of help are displayed when the user passes the mouse pointer over the control for which the help is provided. The two kinds of help are:

- short fly-over help, or hover help, which appears directly over the control
- long fly-over help, or hover help, which appears in the frame window's IInfoArea object (if one is provided).

If you want, you can enable only one or the other, or both.

This section is optional, and will probably take you 10 minutes.

Enabling the Fly-Over Help

Before you add the text of this help, you need to enable the two types of help. The remaining tasks you need to complete to enable hover help are to add an IVBFlyText object, and to connect the IInfoArea object to the IVBFlyText object. Once these objects are properly set up, you will add the help text to each control.

1. Click the **Other** parts category from the left column of the Parts palette.
2. Click the IVBFlyText* part from the right column of the Parts palette.
3. Click an area of the composition editor workspace outside the FrameWindow object. An IVBFlyText* object appears on the workspace. Hover help for short text help is now enabled.
4. Click the **InfoArea1** object at the bottom of the frame window.
5. Click mouse button 1.
6. Select **Connect**.
7. From the cascading menu select **this**.

   Note: This appears at the *bottom* of the cascading menu. Do not select ***this,** which appears at the top) to the IVBFlyText* object's **longTextControl** attribute.
8. Move the star-shaped icon over the IVBFlyText object and click mouse button 1.
9. From the menu select **longTextControl**.

Both long and short hover help are now enabled.

Add Help to Each Push Button

This section describes the steps for adding short and long hover help text to the **Edit** push button. The long version appears in the info area at the bottom of the window, while the short version appears over the object itself. Follow these steps to add help text to the **Edit** push button:

1. In the Composition Editor, select the **Edit** button, click mouse button 2 over it, and select **Open settings** from the pop-up menu.
2. In the list of properties look for **Fly-over long text**.
3. Enter the following text in the entry field to the right of **Fly-over long text**:

   ```
   Copy the entry selected in the list to the editable fields
   ```
4. Enter the following text in the entry field to the right of **Fly-over short text**:

   ```
   Copy entry to edit fields
   ```
5. Press Enter or click **OK** to close the **Properties** dialog.

You can follow these same steps to add hover help to the other buttons, if you like. Here is the text to add:

Button	Long Text	Short Text
Delete	`Delete the user whose name is selected in the list`	`Delete selected user from list`
<< Add	`Add the user information at right to the list`	`Add to list of users`
<<Save	`Save the changed user's information to the list`	`Save edited user`

If you want, you can save the part (press **Ctrl+G**), then switch to the IDE, rebuild the project, and try running the application. When you move the mouse over one of these push buttons, you should see a short bubble of hover help text near the button, and a longer text string in the information area at the bottom of the window.

You can proceed to "Part 8. Making Connections to User-Defined Functions" so that the push buttons in your user interface perform some useful work.

Part 8. Making Connections to User-Defined Functions

Switch to the myGui part in the Composition Editor.

In the Composition Editor, you can make connections from controls to predefined functions, such as removing the selected entry from a list or adding the value of an entry field to a list. For the Review Tool User Administration program, these simple connections are not sufficient, because additional processing needs to take place (for example, the user you delete needs to be deleted from the file containing the list of users).

In this section you will make connections in the Composition Editor from the Add, Delete, Edit and Save buttons to member functions of the myGui class. You will then code these functions in the VisualAge C++ IDE.

Create Multiple Connections for the Delete Push Button

The **Delete** push button will remove the selected entry from the list of users, and should remove that user's information from the user information file. You could implement this as a single connection to a single member function. This would make the connections in the Composition Editor simpler, but would involve more user-written code. Instead, you will make two connections for this button. The first will call a member function you will code, to remove the entry from the user file. The second will use a built-in action, the Remove action, to remove the entry from the displayed list.

Because both connections involve the selected item in a list of items, the order in which the connections are called is important. If the Remove action occurs before your member function is called, the selected entry in the list will no longer exist, so you will not be able to determine which element to remove from your array of users. If you create the custom connection first, it will be called first in the executable file, so you will avoid this problem. (You can re-order connections later if you need to; see Reordering Connections in the VisualAge C++ online Tasks help.)

Follow these steps to create the user-defined member function connection:

1. Click mouse button 2 over the **Delete** push button and select **Connect** from the pop-up menu.
2. Select the **buttonClickEvent** action from the cascading menu.
3. Move the mouse until the star end of the connection line is outside the area for the FrameWindow, over the free-form surface (the white space outside the canvas area).
4. Select the **Member Function...** action from the pop-up menu.
5. In the **Member Function Connection** dialog, enter `void DeleteUser()` as the member function signature (the return type is optional). Notice that the **OK** button is grayed out until you have added the braces to identify the text as a function. Click **OK** to finish the connection.

Follow these steps to create the Remove action that will delete the entry from the displayed list:

1. Click mouse button 2 over the **Delete** push button and select **Connect.**
2. Select the **buttonClickEvent** action.
3. Move the star over the list box and click mouse button 1.
4. Select the **remove** action.

You should now see two green arrows, both starting at the **Delete** button.

Select the second connection line (pointing to the list box) so that it shows three small squares at the start, midpoint, and end of the line. Select the middle square and drag it away from the other two squares, to make the two parts of the line longer. You will notice that this connection line, unlike the others created so far, appears dashed. The Composition Editor displays a dashed line for any connection that is incomplete. The connection is incomplete because you have not indicated what to remove from the list. The Remove action requires an index parameter to tell it what to remove; you need to set this index parameter to the item selected in the listbox.

Follow these steps to complete the connection:

1. Select the incomplete connection line.
2. Click mouse button 2 over it and select **Connect**.
3. Select **index**.
4. Move the mouse until the star is over the list box.
5. Click mouse button 1.
6. Select **selection**.

The connection is now complete. However, the work for the **Delete** button is not finished. You still need to define the DeleteUser function so that the selected item is removed from the user information file before it is removed from the list box. You also need to make connections for the other buttons. You will do this work in "Part 9. Defining Functions for Custom Connections".

Part 9. Defining Functions for Custom Connections

To define the DeleteUser member function, you need to instruct the Visual Builder to include user files for the myGui class in the project configuration, so that you can declare and define the function. These files have the extensions .hpv and .cpv. Then, within the IDE, you can create these files and add the declaration and definition to them.

Follow these steps to define the DeleteUser function:

1. Switch from the Composition Editor view of the myGui part to the Class Editor view. You can do this in two ways:
   - Select the Class Editor icon at the bottom right of the Composition Editor window.
   - Select **View - Class Editor** from the titlebar menu.
2. Select the **Generation options** section of the Class Editor notebook.
3. The lower left portion of this notebook section shows a group box entitled **User files included in generation**. In this group box, turn on the first two check boxes, for C++ header file (.hpv) and C++ code file (.cpv).
4. Press **Ctrl+G** to generate part source again.

When you do this, #include directives for myGui.hpv and myGui.cpv are added to the corresponding myGui.hpp and myGui.cpp files that the Visual Builder generates. You can then declare and define your own member functions of the myGui class in these files from within the IDE.

Define Member Functions for myGui

1. Switch to the IDE and reload the part files when prompted.
2. Press **Ctrl+O** or select **Open or Create File** from the **Project Workbook** menu.
3. Enter a new file name, `myGui.hpv`, in the **Selection** entry field at the bottom of the dialog. Check the **Open as a Workbook Section** radio button. Make sure you have *not* checked the **Add to Project** option, because the generated myGui.hpp file already includes this file with an #include directive. Click **Open** to open the file.
4. Add the declaration `void DeleteUser();` to myGui.hpv. Notice that you do not need to scope the declaration with its class name, because the generated source includes this file from within the body of the class declaration in myGui.hpp.
5. Open another new file, myGui.cpv, as you did for myGui.hpv. For now, define the DeleteUser function with an empty body:

   ```
   void myGui::DeleteUser() {
       ;
   }
   ```

   Rebuild the project. The build should complete successfully, and you can run the executable file, however, if you try the Remove action you will not see any change, because:

   - The list is empty, so there is nothing to remove from it.
   - The `DeleteUser()` function does not do anything as yet.
6. Open another new file, `myGui.cpv`, as you did fof `myGui.hpv`. for now, define the DeleteUser function with an empty body:

   ```
   void myGui::DeleteUser() {
       ;
   }
   ```
7. Save this file. **(Ctrl+S)**.

Change the myGui Constructor

In order to have something to remove from the list, you first need to populate the list. The sensible place to do this is during construction of the myGui object. From the Class Editor you can add code to the start or end of the class constructor; when you generate source code, this code gets inserted at the appropriate location.

You can either enter the complete code within the Class Editor, or you can code a call to a separate user-defined function so that you can then edit your custom code from within the IDE. Since you have already set up your environment to use the .hpv and .cpv files, it makes sense to add the code as a separate member function in these files. It also makes sense to add the code as a separate member function because you may want to populate the list at other times than when the user interface is first loaded. Here are the steps to add a call to the new function:

1. In the Class Editor view, select the **User Code** tab.
2. Select **Ending constructor code** from the upper left pane.
3. In the bottom pane, enter `PopulateList();`. This is a simple function call to the member function, which you will code in the next section.
4. Press **Ctrl+G** to generate part source.
5. Switch to the IDE.
6. If you already have workbook sections for myGui.hpv and myGui.cpv, select the myGui.hpv section. Otherwise, obtain a source view of this file in the Source Files page of the project section by selecting the source file name in the Source Files view.
7. Enter the declaration for the constructor add-on code in the source view:

   ```
   void PopulateList();
   ```

Define the PopulateList Function

1. Change to a source view of myGui.cpv as was described for myGui.hpv in Step 6 above, and add a new class definition for a `UserInfo` class. You will use this class in the PopulateList function, as well as in other functions you define in myGui.cpv. Add the following class declaration, which includes inline constructors and member functions:

   ```
   class UserInfo {
      public:
         IString Name, Key;
         int Authority;
         UserInfo() : Name(""),Key(""),Authority(0) {}
         UserInfo(IString Line) {
            Key=Line.word(1);
            Authority=Line.word(2).asInt();
            Name=Line.word(3).change("_"," ");
            }
         IString Out() {
            if (Name+Key!="")
               return Key+" "
                      +IString(Authority)+" "
                      +IString::change(Name," ","_")
                      +"\n";
            return "";
         }
      };
   ```

2. Next, add some static variables used by several of the functions you will define. Add these at the top of the myGui.cpv file:

   ```
   UserInfo* UserData=0;
   int UserCount=0;
   int UserMax=32;
   ```

3. Next, define the PopulateList function. This function reads lines from the file containing a list of users, populates an array of user records with the users in the file, and adds the name of each user record as the last line in the list box. Here is the definition:

```
void myGui::PopulateList() {
   // 1. Define array for holding user information
   if (UserCount==0) UserData=new UserInfo[UserMax];
   // 2. Read in user file information. UserFilepath
   // will be picked up from the macros.hpp file defined
   // for the CGI part. As each valid
   // user is read, add their name to the UserList object.
   ifstream UserFile(UserFilepath);
   while (UserFile) {
      IString UserLine=IString::lineFrom(UserFile);
      if (UserLine.numWords()>=3) {
         // If the array size is reached, double its size
         if (UserCount==UserMax) {
            UserMax*=2;
            UserInfo* tUserData=new UserInfo[UserMax];
            for (int i=0;i<UserCount;i++)
               tUserData[i]=UserData[i];
            delete[] UserData;
            }
         UserData[UserCount]=UserInfo(UserLine);
         fImpl->iUserList()->addAsLast(UserData[UserCount].Name);
         UserCount++;
      } // end if
   } // end while
   UserFile.close();
} // end function
```

4. The function makes use of the fstream class, and of a preprocessor macro that you defined in the CGI portion of the exercise in the file macros.hpp. You need to include the fstream.h header file at the top of myGui.cpv. If you completed the CGI portion, you can include macros.hpp there as well. Otherwise, add a macro definition for UserFilepath. Add parts of the following excerpt to the top of myGui.cpv depending on your setup. The comments indicate which parts to include in your situation:

```
#include <fstream.h>
// If you did not complete the CGI portion,
// or if it is stored in a different directory than this project,
// add a macro definition like the following:
#define UserFilepath IString("f:\\vatutor\\users.dat")
// Otherwise, include the macros.hpp file:
#include "macros.hpp"
```

   Do not add a trailing semicolon to the UserFilepath macro definition.

You can now build your project again, and try out the user interface. Before you run the program, however, try creating the file pointed to by the UserFilepath macro, if that file does not already exist; create it with the following contents:

```
reader 1 John_Borge
author 3 Ralph_Heinho
reviewer 2 Rachel_Kruger
abcdef 2 Jennifer_Sceeles
```

If you create it within a Source view in the IDE, remember to save the file.

Run your program; you should see the names of these users displayed in the user list. When you select a name and click **Delete**, the name is deleted from the list. However, if you exit the program and start it again, all names are still there, because your DeleteUser function does not yet do anything.

Define the DeleteUser Function

Replace the empty DeleteUser() body in myGui.cpv with the following code:

```
void myGui::DeleteUser() {
   // Determine what item in list is selected
   unsigned long Index=
    fImpl->iUserList()->selection();
   // If an item was selected...
   if (Index!=IBaseListBox::notFound) {
      // Overwrite existing user file
      ofstream UserFile(UserFilepath);
      // Write out information for all users except this one
      for (int i=0;i<UserCount;i++)
         if (i!=Index) UserFile << UserData[i].Out();
      UserFile.close();
      // Remove all items from list
      fImpl->iUserList()->removeAll();
      // Set UserCount to 0 to force reloading of users
      UserCount=0;
      // Repopulate list
      PopulateList();
   }
}
```

This function determines which item in the UserList object is selected with the call:

```
fImpl->iUserList()->selection();
```

The three parts of this call are:

fImpl A member of the myGui class that points to an object of the myGuiImpl class. myGuiImpl is an implementation class generated by the Visual Builder, which you use to access the member functions of the user interface controls for the visual part.

iUserList() A member function of the myGuiImpl class to access the pointer to the UserList object. The Visual Builder creates member functions of the implementation class to access each user interface part. Each function name consists of a lowercase "i", followed by the name of the part.

selection() A member of the IBaseListBox class, from which IListBox inherits. This funciton returns the index value (0 to n) of the first selected entry in the list. The IBaseListBox::notFound static value is returned if no entry is selected.

If an item is selected, DeleteUser opens the user data file for output, overwriting its existing content, and writes all user records except the one selected for deletion to the file. It then removes all entries from the UserList object, sets the UserCount variable to zero, and calls PopulateList to repopulate the list with the remaining entries.

It may seem simpler to use the remove function to remove only the selected entry from the list box. However, that entry would remain in the array of UserInfo objects. Resetting the UserCount variable to zero causes PopulateList to overwrite existing array elements when it loads all remaining entries from the user file.

Define Functions for Adding, Editing, and Saving Entries

You still need to add the following user functions:

- AddUser - This function adds the user whose information is shown in the right-hand fields
- EditUser - This method retrieves the selected user's information and places it in the right-hand fields
- SaveUser - This function saves the information in the right-hand fields to the record for the user whose information was most recently selected by EditUser.

For each of these functions, you should add a simple connection in the Composition Editor, starting with the **buttonClickEvent** for the appropriate button, and ending with a Member function connection to a member function whose name matches the appropriate name above. Don't forget to save and generate part source once you have made these connections.

Add the following declarations to myGui.hpv for these functions:

```
void AddUser(),
     EditUser(),
     SaveUser();
```

Define the functions as shown in the sections below. Before you add them, you should add the following to the top of myGui.cpv, just below the #include directives. The LastEntryEdited variable is used to keep track of which item in the user list was most recently moved to the edit fields, while the AuthLevel and AuthString functions are used to translate either way between integer authority levels and their corresponding strings.

```
unsigned long LastEntryEdited=IBaseListBox::notFound;
int AuthLevel(IString Auth) {
   if (Auth=="Reviewer") return 2;
   else if (Auth=="Author") return 3;
   return 1;
}
IString AuthString(int level) {
  switch (level) {
     case 1: return "Reader";
     case 2: return "Reviewer";
     case 3: return "Author";
     }
   return "";
}
```

Define AddUser (myGui.cpv)

The AddUser function adds the information for the user identified by the two entry fields and the text spin button, provided neither the user's name nor their key has already been added. It then resets the UserCount variable to zero, clears the list box of displayed users, and updates the list. It also resets the LastEntryEdited variable, to prevent you from accidentally overwriting the information for the user you most recently asked to edit with the new user's information.

```
void myGui::AddEntry() {
// Get the input information for the user
   IString UserName=fImpl->iUserName()->text();
   IString UserKey=fImpl->iUserKey()->text();
   int Authority=AuthLevel(fImpl->iAuthorityLevel()->text());
// Check that this user isn't already in file; return if it is.
   for (int i=0;i<UserCount;i++) {
      if (UserData[i].Key==UserKey) return;
      if (UserData[i].Name==UserName) return;
   }
// Add to the file.
   ofstream UFile(UserFilepath,ios::app);
   UFile << UserKey << " "
         << Authority  << " "
         << UserName.change(" ","_") << endl;
   UFile.close();
// Clear list and force repopulation
   UserCount=0;
   LastEntryEdited=IBaseListBox::notFound;
   fImpl->iUserList()->removeAll();
   fImpl->iUserName()->setText("");
   fImpl->iUserKey()->setText("");
   PopulateList();
}
```

Define EditEntry (myGui.cpv)

The EditEntry function determines what user is selected in the user list, and copies the information for that user to the name, key, and authority fields at the right. Because the index of the selected user is stored in the LastEntryEdited variable, this variable is also accessible to the SaveEntry function defined below.

```
void myGui::EditEntry() {
   // Determine the item selected in the list, and save for SaveEntry
   LastEntryEdited=
    fImpl->iUserList()->selection();
   // Copy this user's information to the input fields at right
   if (LastEntryEdited!=IBaseListBox::notFound) {
      UserInfo* up=&UserData[LastEntryEdited];
      fImpl->iUserName()->setText(up->Name);
      fImpl->iUserKey()->setText(up->Key);
      fImpl->iAuthorityLevel()->setText(AuthString(up->Authority));
   }
}
```

Define SaveEntry (myGui.cpv)

The SaveEntry function uses the value of the LastEntryEdited variable to ensure that the information being saved was actually requested from a call to EditEntry, and to determine which element of the UserData array should be changed. It writes all user information out to the file, then forces the user list to be repopulated.

```
void myGui::SaveEntry() {
   // Make sure this is the last entry requested for editing
   // and that it's within the current array size
   if (LastEntryEdited<UserCount &&
      LastEntryEdited!=IBaseListBox::notFound) {
   // "up" is used as a shorthand for the element we want
      UserInfo* up=&UserData[LastEntryEdited];
   // Get the information from the input fields
      up->Name=fImpl->iUserName()->text();
      up->Key=fImpl->iUserKey()->text();
      up->Authority=AuthLevel(fImpl->iAuthorityLevel()->text());
   // Write the entire file out, replacing old contents
      ofstream UserFile(UserFilepath);
      for (int i=0;i<UserCount;i++)
         UserFile << UserData[i].Out();
      UserFile.close();
   // Update the user list.
      fImpl->iUserList()->removeAll();
      UserCount=0;
      PopulateList();
   }
}
```

Once you have added this code to myGui.cpv, rebuild the project (make sure you have done a Save and Generate in the Visual Builder, since the last Visual Builder changes you made). Run the program; you should be able to do the following with the tool:

- Add new users, specifying a name, key value, and authority level
- Copy the information for existing users to the input fields so you can change their information
- Edit a user's information, and save the changes back to the file
- Delete a user from the list
- Exit the program

Congratulations!

You have now completed the Visual Builder portion. You learned most of what you will need to know in order to create simple visual user interfaces and to implement C++ code for them. For more in-depth information see the online Tasks information in the VisualAge C++ help system. Look under **Develop an Application Using Parts** for common Visual Builder tasks.

If you have already done the other parts of this Getting Started guide, and you still feel you need more practice, try loading one of the sample projects, either from the online Samples navigation in the help system, or from the welcome screen of the IDE.

Communicating Your Comments to IBM

VisualAge C++ Professional for AIX
Getting Started
Version 4.0

Part Number 30L8557

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@ca.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)