

VisualAge[®] C++ Professional for AIX[®]



Try VisualAge C++

Version 5.0

Note!

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

First Edition (February 2000)

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v	Set up Dependencies Among Subprojects	15
Programming Interface Information	vii	Summing up Tutorial 2	16
Trademarks and Service Marks	vii	Chapter 4. Tutorial 3: Build a Subproject	17
Industry Standards	viii	Optimizing Subprojects	17
About This Book.	ix	Find Included Sources in the IDE	17
Chapter 1. Introduction to VisualAge C++ Configurations and Subprojects . . .	1	Organize the Project	19
Chapter 2. Tutorial 1: Configuration Files (.icc)	3	Create a New File	20
Copy Source Files for the Sample Project	4	Create a Custom Page	21
Create a Configuration File	4	Group Common Options in a Separate Configuration File	23
Configuration Directives	6	Summing up Tutorial 3	24
Setting Options	7	Chapter 5. Tutorial 4: Builds	25
Codestores	8	Create a Build	26
Summing Up Tutorial 1.	9	Add Conditional Processing to options.icc	26
Chapter 3. Tutorial 2: Introducing Projects	11	Set Build Variables in the Default Build	27
Create a New Project in the IDE	11	Build the Project.	28
Examine a Basic Project	13	Environments.	29
The Project File	14	Browse an Environment	30
A Closer Look at Subprojects	15	Summing up Tutorial 4	31
		Other Useful References	31
		Contacting IBM	33

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2000. All rights reserved.

Programming Interface Information

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

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

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

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

Trademarks and Service Marks

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

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architecture
TeamConnection
VisualAge
WebSphere
Workplace Shell

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

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

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

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

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

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

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

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

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

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

Industry Standards

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

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

About This Book

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

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

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

Chapter 1. Introduction to VisualAge C++ Configurations and Subprojects

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

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

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

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

The Sample Application

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

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

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

Chapter 2. Tutorial 1: Configuration Files (.icc)

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

The .icc file extension stands for Incremental C++ Configuration. A configuration file contains all of the information needed for processing source files. It holds all options applied to the source files and targets, lists all input libraries, and defines all targets (outputs). If a source file is not listed in a configuration file, or included by a file that is listed in the configuration file, it is not a part of the project.

Here is an example of a configuration file:

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

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

Option directive

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

Target directive

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

Source directive

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

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

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

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

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

Copy Source Files for the Sample Project

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

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

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

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

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

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

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

Create a Configuration File

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

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

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

Identifying your target

1. From the Project Workbook menu, select **Create configuration**.
2. Click **Next** on the first page of the SmartGuide. You can turn this page off by clicking the **Show this page next time** check box, to clear it.

3. On the Configuration page, enter:
 - a. util.icc, as the name for the configuration file. Whether or not you type the extension, the file will automatically be saved with the extension .icc.
 - b. a location for the configuration file. In the **Location** field, browse to or type the path for the directory to which you copied the sample source files, for example, /home/joe/sample.

Ignore the check box labelled **Disable created target**. This is used only when you are creating a configuration for migrating a project that has not been built in VisualAge C++ Version 4.0 or 5.0 before.

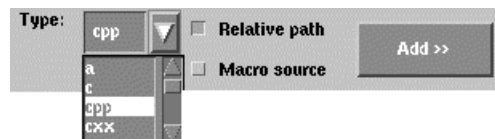
4. Click **Next**.
5. On the Target Type page, you can select whether the output will be a library or an executable. For this sample, select **Shared library (SHR)** from the pull-down list.
6. Click **Add target**.

Identifying options

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

Adding sources to the target

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



4. Click **Add>>**. All the file names should appear in the window on the right, with **cpp** appearing under **Type** for each file.
5. Click **Next**.
6. On the target type page, click **Finish** to complete the configuration and exit the SmartGuide.
7. When you are prompted to open the project in the IDE, click **Yes**.




8. When you are prompted to build the subproject, click **Yes**.
9. When you are prompted to optimize your configuration, click **No**.

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

Configuration Directives

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



In the IDE, three new workbook sections have appeared:

-  The Project section
-  The Configuration section
-  The Codestore section

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

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

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

1. Select the Configuration section by clicking on the  workbook tab.
2. Select the **Source** page button  to see the contents of the configuration file.

You should see the following:

```
option SubprojectOptions = gen(rtti, yes)
{
    options defaults("x1C_r"), link(linkwithsharedlib, yes), define("USE_IPF", "1")
    {
        target type(shr) "util.so"
        {
            source type(cpp) "matrix.cpp"
            source type(cpp) "notifier.cpp"
            source type(cpp) "vector.cpp"
            source type(cpp) "vector.hpp"
            source type(cpp) "matrix.hpp"
            source type(cpp) "except.hpp"
            source type(cpp) "notifier.hpp"
            source type(cpp) "observer.hpp"
            option macros(global, yes)
            {
                source "pthread.h"
            }
        }
    }
}
```


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

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

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

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

Setting Options

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

In this part you will:

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


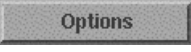

There are several types of options for compiling and linking. They are all set in configuration files. If you know the name and syntax of an option, you can type it directly in the configuration file to set it. However, there are hundreds of options available. The technique below is the fastest way to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

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

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

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

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

1. Select the Configuration section .
2. Select the Options page .
3. In the left pane, select the target object ().
4. Click into the Change Options view (the right pane).
5. Click into the Live Find field at the bottom of the pane.
6. Type export all. The view scrolls to the first match. Press Enter until you see **Export all defined external symbols** :

```
[-] C++ Only Linker Options
    [+  Export all defined external symbols
    [+ Initialization priority for multiple shared libraries:  *
[-] Optimization Options
[-] C Only List Options
[-] Miscellaneous Options
```

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

C++ Only Linker Options
 Export all defined external symbols
Keyword: link(exportsall)
Batch keyword: -bexportsall
Kind: Boolean
Default Value: false


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

C++ Only Linker Options
 Export all defined external symbols
Keyword: link(exportsall)
Batch keyword: -bexportsall
Kind: Boolean
Default Value: false

After you do this, and the option is explicitly set (turned on), the option description is displayed in green, and a check mark appears in the box next to it. The option also appears in your configuration file.

10. At the bottom of the pane, click **Apply**. This copies the option to your configuration file.

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

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


Codestores

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


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

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

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

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

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

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

Summing Up Tutorial 1

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

What have you learned so far?

This tutorial introduced the concepts of:

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

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

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

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

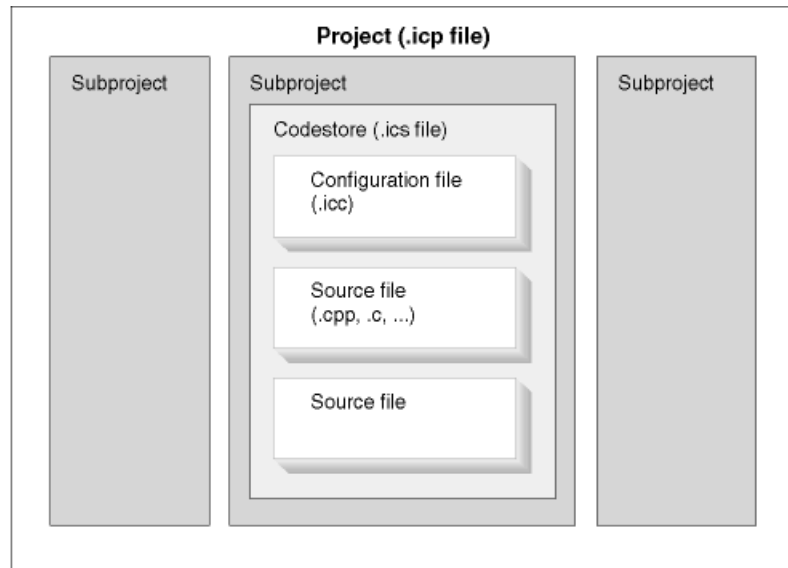
If you are interested in learning more about the concepts introduced in this tutorial, try reading some of the following pages in the online help:

- Codestore
- Configuration Files
- Configuraion File Directives

Chapter 3. Tutorial 2: Introducing Projects

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

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



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

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

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

Create a New Project in the IDE

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

Before you start these steps, ensure you have write access to a directory, so that you can copy the sample files you will be using. If you have not done Tutorial 1, please follow the steps in "Copy Source Files for the Sample Project" on page 4 before you continue with this part.

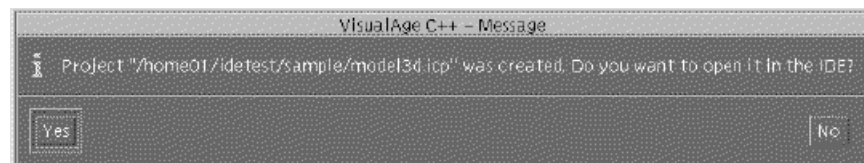
When you have copied the source files, create your project:

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

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



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



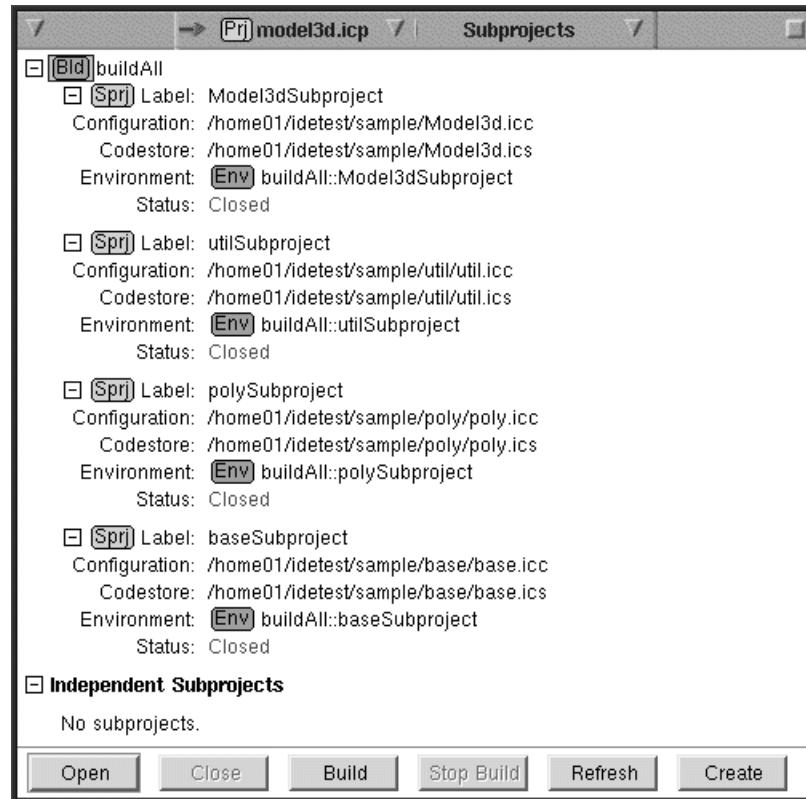
10. In IDE, the **Overview** page of the Project section is displayed by default. You have created a simple project consisting of four subprojects.




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

Examine a Basic Project

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

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




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

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

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

The Project File

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

The contents of the file are as follows:

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

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

Subproject directive

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

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

Build directive

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

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

Use directive

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

In the next step, you will examine the subproject.

A Closer Look at Subprojects

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

Each subproject is essentially a combination of:

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

A subproject is represented in the IDE as `Spri`.

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

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

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

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

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

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

Set up Dependencies Among Subprojects

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

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

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

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

```
subproject Model3dSubproject icc "Model3d.icc", ics "Model3d.ics"
{
depends on polySubproject
```

```

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

```

or:

```

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

```

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

Summing up Tutorial 2

What have you learned so far?

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

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

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

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


- Projects
- How Project Files are Processed

Chapter 4. Tutorial 3: Build a Subproject

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

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

First, build the subproject `utilSubproject`:

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

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

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

Optimizing Subprojects

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

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

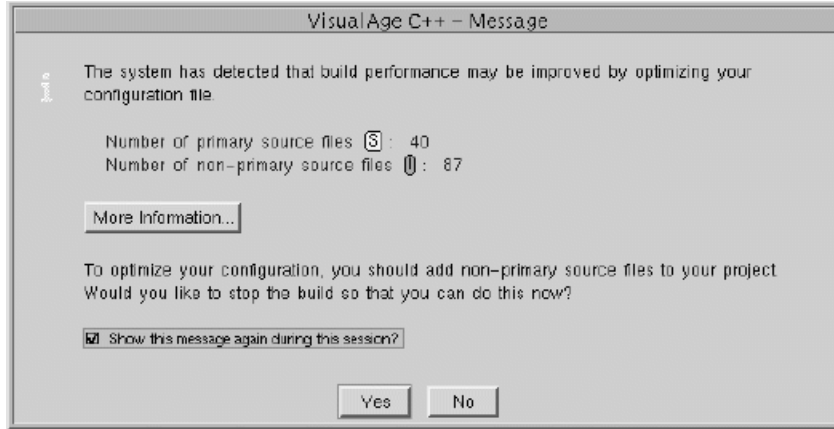
With the incremental compiler, there is no need to repeatedly include header files. Each time you launch a build, the compiler first checks the configuration file to see which sources are listed there. Sources listed in the configuration file are called *primary* sources. *Secondary* sources are those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

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

The IDE provides some handy features for identifying secondary sources.

Find Included Sources in the IDE

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



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

Locate the Included Sources

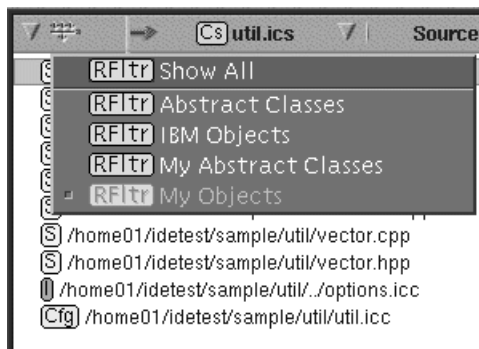
Go to the Codestore section for utilSubproject. Select the Source Files view on the Source Files page.

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

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

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

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

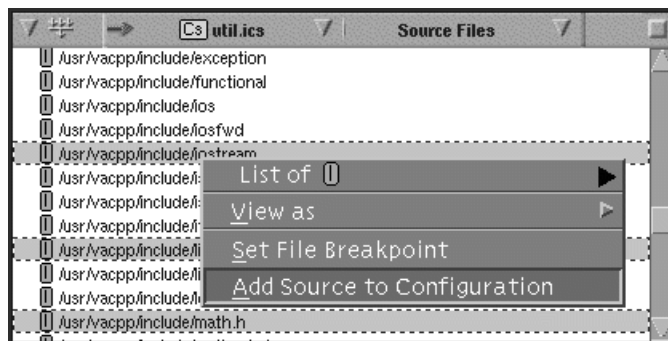



Now scroll to the bottom of the pane: you should see a much longer list of included files.



Promote the Included Files

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

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



3. In the **Add to Source** dialog, select the **Add Source to Source Directive** radio button. Select the directive that currently includes `pthread.h`.
4. Click **Apply**.
5. To update the view, select **Refresh Configuration Views** for the `util.icc`  object from the **Project Workbook** menu.

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

Promoting included sources is the most important way to improve build efficiency. If you are interested in exploring other ways, you can bookmark this page, and read the online migration information before continuing the tutorial.

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

Organize the Project

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

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

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

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

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

Why Separate the Options?

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

Create a New File

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

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

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

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




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

Create a New File

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

4. In the **File open location** part of this dialog, ensure that the **Current pane** radio button is selected.
5. Click **Open**.

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

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

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

Create a Custom Page

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

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

Create a New Pane

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

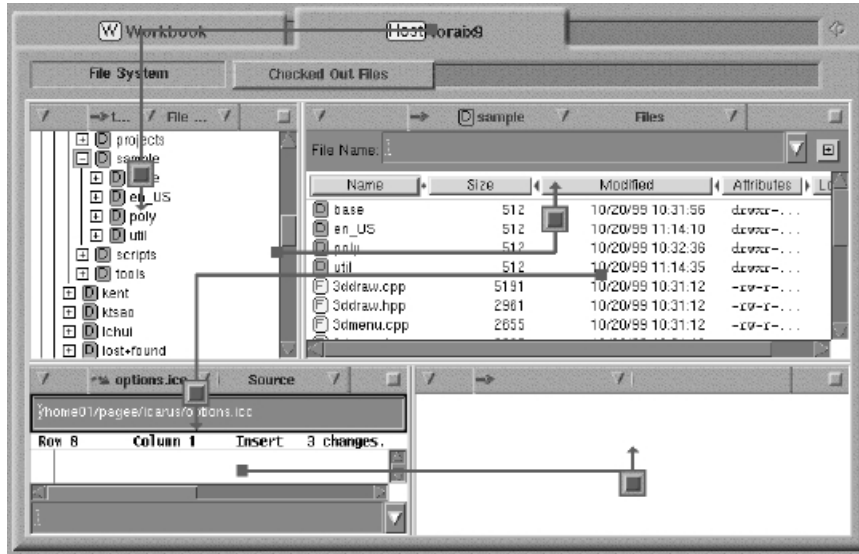
You have just created a new pane. It is empty, and has no view type. By default, whenever you create a new pane, the new pane is linked to the last pane that had focus.

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

Change the Input to the New Pane



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

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

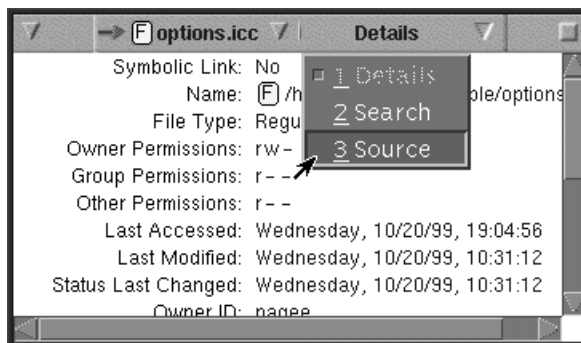


The link for the new pane originates in the lower left pane.

Moving the Link

1. Clear the link diagram by pressing Esc. (You can also move links in the link diagram, but the method you will use here saves time).
2. Hold the mouse pointer over the linking arrow in the title bar of the new pane (), click, and begin dragging it toward the pane directly above. The mouse pointer becomes a chain link  . (If it does not, release the mouse and try again: dragging by any other part of the title bar will move the pane).
3. Release the mouse when the link is anywhere inside the upper right pane.

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



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

Save the New Page Layout

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

1. Select **Save Page Description as** from the **Page** menu.

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

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

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

Group Common Options in a Separate Configuration File

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

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

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


```
option SubprojectOptions = gen(rtti, yes)
{
    option defaults(x1C_r), link(linkwithsharedlib, yes),
    define("USE_IPF", "1")
```
2. Cut this from util.icc, and paste it into options.icc.
3. In options.icc, create a group for these options, named ProjectOptions, as follows:


```
option ProjectOptions = gen(rtti, yes), defaults(x1C_r),
    link(linkwithsharedlib, yes), define("UES_IPF", "1")
```
4. Now, in **util.icc**, add ProjectOptions into the current option directive applied to the target, as follows:


```
option ProjectOptions,link(exportAll)
{
    target type(shr) "until.a"
    {....
```
5. At the top of util.icc, add an include directive to include the file containing the definition of the options group ProjectOptions:


```
include "../options.icc"
```
6. Now remove the same options from each of the remaining three configuration files, replace them with the name of the ProjectOptions group in each file, and add the include directive to each file. The remaining configuration files are:
 - poly/poly.icc
 - base/base.icc
 - model3d.icc
 (note that in model3d.icc, the relative path will be different: "options.icc")

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

Why is the New File a .icc File?

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

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

Summing up Tutorial 3

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

You learned that:

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

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

To learn more about optimizing configuration files, try reading the process for migrating to incremental compilation. The complete *Migration Guide* is also available in PDF form, in the `/pdf/` subdirectory of the VisualAge installation directory. Look for `migrate.pdf`.

Chapter 5. Tutorial 4: Builds

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

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


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

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

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

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

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

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

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

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

Create a Build

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

To create a new build and create variables for it:

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

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

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

Add Conditional Processing to options.icc

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

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

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

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

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

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

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

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

```
option ProjectOptions=opt_build,
  gen(rtti, yes),
  defaults(x1C_r),
  link(linkwithsharedlib, yes), define("USE_IPF", "1")
```

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

Set Build Variables in the Default Build

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

To do this:

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

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

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



- create a new build directive
- select the subprojects you want to use
- create a variable to identify the new scenario


- add conditional processing to the common configuration file, if new options are needed for this scenario

Build the Project

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

Build the Optimized Scenario

Before starting to compile the build you set up earlier in this tutorial, you must close any subprojects that were opened in memory in the default build. Select each open subproject and click Close. When the  and  graphics disappear from the subproject description, and the status reads Closed, the subproject is closed. If the subprojects are not closed in this build, you will get an error message when you try to build the optimized build.

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

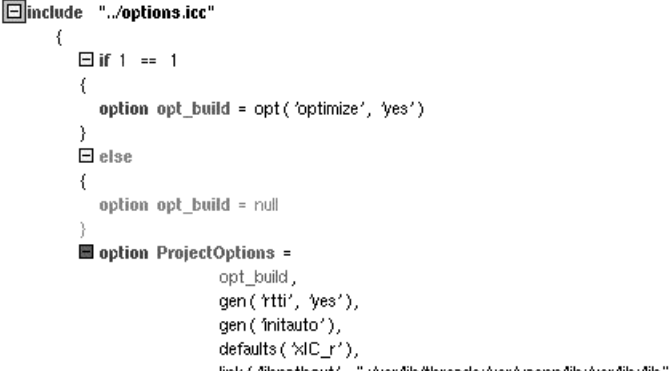



When the build is complete, all the subprojects are open in memory. To see how the configuration file for any subproject was interpreted, open a configuration file as a section:

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

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

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

```


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

```

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

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

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

Environments

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

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

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

Asn

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

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

Var

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

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


Env

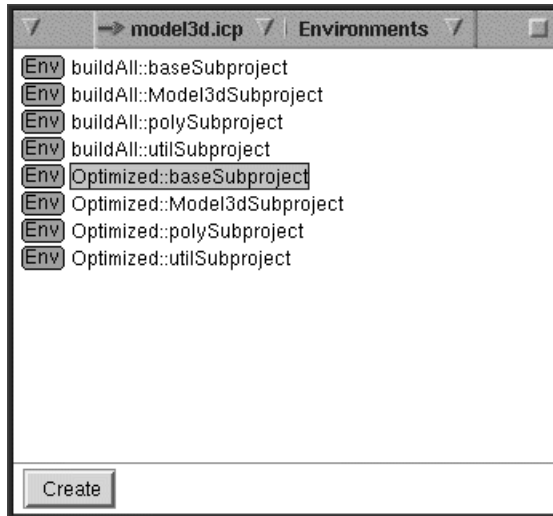
an *environment* is the total collection of the values inherited from the system environment, the values set or overridden by assignments you created, and values contained in variables groups. Like a variables group, environment objects are named. Environments are not selected in the project file, however. They are associated with run specifications, when you run or debug your application.

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

Browse an Environment

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

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




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

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


- The variables and values under **Evaluation context** are those in the system
- The variables under **Environment variables** are those you have set within the project, in the Optimized build:

```
☐ Environment Variables
  Name *Value (  evaluated )*
  "optimize" "1"
```

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

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

Now you can easily associate this with a run specification:

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

Summing up Tutorial 4

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

You learned how to:

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

Other Useful References

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

In the online help, you can find the following:

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

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

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

Projects and Subprojects
Incremental Configuration Files
Targets
Types of Source Files

Contacting IBM

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

Comments on This Help

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

Fee Support

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

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

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

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

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

Phone numbers for information on Support Line offerings are:

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

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

- Elsewhere, please contact your local IBM office.

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

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

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

Consulting Services

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

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

AIM_SERVICES@us.ibm.com