

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



Incremental Compilation

Version 5.0

Note!

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

Edition Notice

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

© 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
Programming Interface Information	vii
Trademarks and Service Marks	vii
Industry Standards	viii

About This Book.	ix
-----------------------------------	-----------

Chapter 1. Incremental Compilation Concepts	1
Incremental C++ Builds.	1
Incremental Configuration Files	2
Basic Configuration Files	3
More Complex Configuration Files	3
Configuration File Directives	4
Configuration File Syntax	6
Example: Configuration File	7
How Configuration Files are Processed	8
Sources	9
Primary vs Secondary Sources	11
Macro vs Non-macro Sources	11
Targets	12
Variables and Environments	14
Environment Variables in Configuration Files	15

Chapter 2. About Projects and Subprojects	17
--	-----------

Build in a Team Environment	17
Projects and Subprojects	17
Project Files	19
How Project Files Are Processed	19
Example: Project File	20
Related Configuration Files	20
Project File Syntax	21

Chapter 3. Using the Incremental Compiler	27
Build	27
Build from the Command Line	28
Build Executable Programs	29
Group Source Files in a Configuration	29
Compile and Bind Resources	30
Macros in C++ Source Files	30
Search Paths for Included Source Files (AIX)	32
Cleaning Up After Builds.	33
When to Use Makefiles	35

Chapter 4. Related References	37
Equivalent Batch Compile-Link and Incremental Build Options (AIX)	37
The One-Definition Rule (C++)	52

Contacting IBM	55
---------------------------------	-----------

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.

References to VisualAge or VisualAge C++ in this book should be interpreted as VisualAge C++, Version 5.0.

Chapter 1. Incremental Compilation Concepts

Incremental C++ Builds

VisualAge C++ gives you an incremental build paradigm that can greatly increase your productivity, by reducing the time required to repeatedly compile and link C++ code. When you perform an incremental build, only portions of code that are affected by changes in your source are recompiled and relinked.

After you write or modify your source code, you initiate a build to create a target module that is consistent with the current state of the C++ source code. VisualAge C++ performs one of the following types of builds:

- **Initial build**

The initial build has only source files as input. The first build you perform from a set of source files or any build that does not use an existing codestore is an initial build.

- **Incremental builds**

Subsequent builds use both the source code and the results of the previous build. Compile and link overhead is determined by how you changed the source code since the previous build. For example, if you change the body of a non-inlined function, only that function is recompiled and relinked. If you change a declaration in a header file, only functions and data that use the declared item are affected, rather than every file that includes the header.

To make incremental builds work, VisualAge C++ maintains a program database called the codestore. The codestore contains information about source files, stored so that source regions much smaller than the original source files can be recompiled independently. A project may consist of multiple subprojects, each with its own codestore. A codestore may represent more than one configuration file only when one configuration file explicitly includes another configuration file with include directives. The configuration file defines the source files, target files, and options used during a build. VisualAge C++ also stores this information in the codestore and uses it to determine how the configuration changes from one build to the next.

In addition to greatly increasing the speed of builds, incremental builds give you features that you can use to improve your productivity:

- **Orderless programming**

By default, the compiler processes all source files together, and declarations can appear in any order within a namespace scope, including global scope. The compiler finds declarations as it needs them, and you do not have to ensure that objects, functions, and classes are declared before they are used. You can change this default behavior by specifying the `lang(orderednamelookup)` which enforces order and allows for code portability. You can define global macros. If you specify which files contain global macros, those files are preprocessed first in the order in which you list them in the configuration file.

- **Header file flexibility**

You do not need traditional header files. You can continue to use them, but you can simplify them and supply them as ordinary source files, or include them using `#include` directives in the traditional manner.

- **The one-definition rule**

VisualAge C++ enforces the rule of the C++ standard that a declaration with external linkage cannot have more than one definition unless the definitions are identical.

There are useful differences between building a codestore, a subproject, and a project file.

- If you build a project object (`[Prj]`), the project file (.icp file) will be built. This initiates a build of all the subprojects associated with the project, in the order of their dependencies.
- If you build a subproject object (`[Sprj]`), the subproject *and* any other subprojects it depends on.
- If you build a codestore object (`[Cs]`), only the configuration file associated with the codestore will be built, regardless of dependencies. You can use your knowledge of what you have changed since your previous build to skip the expense of timestamp checking files that you know are unchanged.

RELATED CONCEPTS

“Projects and Subprojects” on page 17

“Incremental Configuration Files”

Codestore

C++ Source Files and Source Regions

“Macros in C++ Source Files” on page 30

“The One-Definition Rule (C++)” on page 52

RELATED TASKS

Set Build Options

Convert Included Source Files to Primary Source Files

Build

RELATED REFERENCES

Build Options

Incremental Configuration Files

A set of source files, input libraries, and processing options that are used to generate one or more targets is called a *configuration*. Each configuration is defined using a *configuration file*. A VisualAge C++ subproject must contain at least one configuration, but not necessarily a configuration file.

A configuration file is similar to a makefile, but with some important differences. Unlike a makefile, you do not need to specify any file-dependency or processing information. A configuration file simply contains definitions of targets, the source files used to build those targets, and the options used to process the source files and targets. Once you tell VisualAge C++ what to build, it figures out how to do the build.

A configuration file consists of a series of statements called *directives*. In its most basic form, a subproject consists of source files and a target (e.g., a C++ file and a header file are sources; they are used to build a target of an executable file).

Basic Configuration Files

This simple command invokes the compiler, and by default, the linker, to build an executable named `runme`, using the source `abc.cpp`:

```
xlc -o runme abc.cpp
```

The equivalent configuration file looks like this:

```
target "runme"  
{  
    source "abc.cpp"  
}
```

This example reveals some essential aspects of configuration files:

- There is no explicit command to start the compiler. The build command is not part of the configuration. The equivalent to `icc` is either to click the Build icon in the IDE, or to type `vacbld` on the command line, followed by the name of your configuration file.
- No command is needed to invoke the linking phase.
- The `.shr` suffix defines a target to be a shared library.
- The configuration file establishes a clear relationship between the source file and the target file by using `{` and `}` (open and close braces) to mark which sources are used to build a target.

More Complex Configuration Files

As the application grows more complex, you can add other directives to set options, group sources together, or set up conditional processing.

- The `target` directive specifies the build's output, such as an executable.
- The `source` directive specifies the source files for a project.
- You can use the `group` directive to define a group of source files. The `group` directive does not specify source files for a project, so the source group still needs to be listed in a `source` directive. The `group` directive makes dealing with source files that are processed in the same manner easier than dealing with the files individually.
- The `option` directive defines one or more build options. If you add a name to an option directive, you define an options group. An options group can be used like a variable to apply options that you commonly use together.
- `if` and `for` directives specify conditional processing.
- `run` directives launch processes other than the compiler as part of the build.

When you build an application, the configuration file is processed to create a new codestore or to update an existing one. The processing of each configuration file produces one updated codestore. The codestore is queried to determine changes from build to build.

The default extension for a configuration file is `.icc`, but you can use any extension. However, do not confuse these files with `.cfg` files used by the batch and incremental compilers. Also, be aware that VisualAge C++ may expect the default extension for configuration files, like in the Open dialog of the IDE. The configuration file can reside in any directory.

You can create a configuration file for a project in any text editor, or follow the Configuration SmartGuide to let the IDE create it for you. You can view, add, change and remove target, group, source and option directives:

- Edit the file directly in a source view in the IDE or in your editor

- Manipulate the objects representing them in other IDE views

Example: Option Group Directive

“Example: Configuration File” on page 7

RELATED CONCEPTS

Configurations

“Sources” on page 9

“Targets” on page 12

“Projects and Subprojects” on page 17

Codestore

“How Configuration Files are Processed” on page 8

“When to Use Makefiles” on page 35

“Incremental C++ Builds” on page 1

Editor Overview

RELATED TASKS

Create a Project

View a Build Setup (Configuration File)

Change a Build Setup (Configuration File)

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37

“Configuration File Syntax” on page 6

Configuration File Directives

Configuration files contain a sequence of directives that control the subproject build process. The way target files relate to source files, and what options apply to targets and sources, is determined by how directives are combined in the configuration file.

Directives can stand alone or have blocks of subdirectives enclosed in braces, { and }. The braces delimit a directive block and define the scope of a directive. Directive blocks can be nested. The following example contains two directive blocks. The source directive is a subdirective of the option directive and is the only directive in its block. Together, the inner block and the option directive make an outer block. Both the source directive and the option directive are subdirectives of the target directive.

```
target "day.exe" {
    option macros(global) {
        source "am.hpp", "am.cpp", "pm.cpp", "pm.hpp"
    }
}
```

In the following example, lang(oldDigraphs) is an attribute of the option directive. This attribute applies to source file day.cpp, but not to source file night.cpp

```
option lang(oldDigraphs) {
    source "day.cpp"
}
source "night.cpp"
```

Directives often contain names of options used by processes that run during builds. For example lang(oldDigraphs) in the code above is a language option that applies to C++ sources. Directives can also contain the following elements:

- Data
- Variables
- Environment Variables
- Expressions

The directives used in configuration files are listed below:

= (assign)	assigns values to variables
error	terminates the build with a message
for	processes directives repeatedly
group	assigns a list of source files to a group
if	conditionally processes directives
include	includes textual content of other files containing directives
option	set options that control how targets are built from sources
perform	controls scope of variables
run	runs programs during the build except for built-in processes such as compile and link.
source	names input files
target	names and specifies types of output files
tool	identifies a program that is to be loaded for use like a built-in process

Any directive can be preceded by the **disable** keyword. When applied to an option or target directive, this keyword disables the option or target part of the directive. Directives within the associated block are not affected. In other cases, the keyword **disable** deactivates the whole directive.

“Example: Configuration File” on page 7

RELATED CONCEPTS

“Incremental C++ Builds” on page 1

C Compilation

“Incremental Configuration Files” on page 2

“How Configuration Files are Processed” on page 8

RELATED TASKS

Change a Build Setup

Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37

“Configuration File Syntax” on page 6

Data Elements in Configuration Files

Configuration File Syntax

Configuration files contain text. You can modify configuration files from the Configuration (**Cfg**) section of the IDE, or using a text editor. You can give configuration files any name and extension you choose. By default, they have the extension `.icc`.

The configuration file syntax follows these C and C++ rules:

- Spaces, tabs, and new line characters serve as separators between tokens. Lines are free-format. There are no rules about indenting, but you should indent lines to improve readability.
- Comments can appear anywhere. They follow the same rules as C++ comments. Both the following forms are allowed:

```
// this comment runs to the end of the line
/* this comment is delimited at both ends
and can span lines */
```

The content of a configuration file is a sequence of directives and comments. For example, the following lines contain one comment and the target, option, and source directives.

```
// build an executable program
target "myprog" {
    option lang(compatMath) {
        source "mymain.cpp", "myclasses.hpp"
    }
}
```

Directives often contain names of options used by processes run during a build. In the code above, `lang(compatMath)` is a language option that applies to C++ sources.

Unlike C and C++ statements, directives do not require a semicolon or other termination character. Directive names are case-sensitive and must be in lowercase. Option names are not case-sensitive.

Configuration files can also contain the following elements:

- Data
- Variables
- “Environment Variables in Configuration Files” on page 15
- Expressions

“Example: Configuration File” on page 7

RELATED CONCEPTS

“Incremental Configuration Files” on page 2

“How Configuration Files are Processed” on page 8

RELATED TASKS

Change a Build Setup
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37
“Configuration File Directives” on page 4
Build Options

Example: Configuration File

The following example builds a small application called styleChecker.

```
/*AIX version*/
if $__TOS_AIX__ {
    targetName = "styleChecker"
/* compiler options */
option generalOpts = incl(searchPath, $VACPPMAIN "/extensionAPI/include/"),
                        link(libsearchpath,
$VACPPMAIN "/extensionAPI/lib/"),
                        link(heap,
0x60000000),
                        defaults(x1C128_r)
/* API libraries */
group libSrc = "libCS_common.a",
              "libCS_factory.a",\
              "libCS_tobeyIlGenerator.a",
              "tobeyData.exp",
              "tobeyData.o"
/* Windows version */
} else {
    targetName = "styleChecker.exe"
/* compiler options */
option generalOpts = incl(searchPath, $VACPPMAIN "/private/include/"),
                        link(libsearchpath,
$VACPPMAIN "/private/lib/")
/* API libraries */
group libSrc = "csccommon.lib",
              "csfactory.lib"
}
/* The rest is common to both Windows and Aix */
option generalOpts {
target targetName {
```

```
source type(cpp) "styleChecker.C"

source libSrc

}

}
```

RELATED CONCEPTS

“Incremental Configuration Files” on page 2
“How Configuration Files are Processed”

RELATED TASKS

Change a Build Setup
Set Build Options
Build

RELATED REFERENCES

“Configuration File Syntax” on page 6
Build Options

How Configuration Files are Processed

A configuration file defines a configuration for a subproject, rather than specifying an ordered set of steps for a build. Unlike makefiles, which drive a make process using ordered dependencies based on targets, most configuration file directives establish a relationship between source files and target files.

When you build, VisualAge C++ processes directives in a configuration file. Targets are built incrementally, and building targets drives the build process. Target directives name the output files. Source directives supplied as subdirectives of a target directive list the source files to use to create the target file. These files determine whether C++ incremental compile, C compile, link, resource bind, or other actions are performed during the build process.

Option directives apply options to the source and target files. The set of sources and options can be built up before and within the target directive. Options that are in effect when the target directive is processed, and that apply to the files used to produce the target file, are passed to the processes that operate on those files.

For example, the directives below forward options that apply to C source files and to the target executable to the C compiler and the linker. The group directive creates a group name for the source files. When the source and target directives are processed, the options defined as sourceopts and targetopts are applied to the source and target files. Options that have no meaning for the target and source file types are ignored. The following example ignores any source options that are specific to C++ source files, resources, or any types of files other than C source files. Any target options that do not apply to executable targets are also ignored.

```
option sourceopts = /* ... */
option targetopts = /* ... */
group MySourceList = myprog.c, myfunc1.c, myfunc2.c
option sourceopts, targetopts {
    target myprog.exe { source MySourceList }
}
```

If an option is specified more than once for the same files, second and subsequent uses of the option either append to the value of the setting or reset the option, depending on the option. Similarly, some options implicitly set or override others.

Some steps in the build process do not require a target directive to be activated. Build extensions add capabilities to the build process that are not necessarily driven by target directives. To use a build extension specify a source file that the build extension processes in a source directive, or use a tool directive. The description of each build extension tells you what types of source files they are associated with, and how to prepare your configuration file to use them.

Other factors can alter the order in which directives in configuration files are handled:

- The if directive allows conditional processing of blocks of directives.
- The perform directive can limit the scope of variables to within blocks of directives.
- The include directive textually embeds directives from another configuration file.
- The run directive schedules execution of command line programs. The targets, sources, and other parameters of the run directive control whether execution occurs before or after the compile stage, the link stage, and other run directives.
- A configuration file can start another build as a synchronous child process, using a run directive.
- A non-zero return code from a run directive can terminate the build.
- Build errors suppress the processing of subsequent directives that depend on targets of the build.
- The error directive stops processing of the configuration file.

“Example: Configuration File” on page 7

RELATED CONCEPTS

“Projects and Subprojects” on page 17

“Incremental Configuration Files” on page 2

Codestore

Build Extensions and Extension Specification Files

RELATED TASKS

Add or Change Source Files

Change a Build Setup

Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37

“Configuration File Syntax” on page 6

“Configuration File Directives” on page 4

Build Options

Build Extensions Supplied with VisualAge C++

Sources

The source files for a build are specified in your configuration file. The source directive controls the name and type of the sources for a build.

The default types recognized by VisualAge C++ are listed in the following table. If a type is not recognized in the type clause on all platforms, the platforms to which it applies are shown in the first column. The file extensions associated with each source type are shown in the second column. Source files are recognized *only* by their extension, unless you specify their type in a configuration file. To make your configuration file compatible with many operating systems, types specified in the type clause are mapped onto different file extensions if required by your operating system.

A configuration file can list source files other than the standard .c, .cpp, .h and .hpp file types. Examples of other source types are:

type Clause	File Extension	Type of Source
> AIX a	a	static library
> AIX C	C	C++ source
c	c	C source
cpp	cpp	C++ source
cxx	cxx	C++ source
> AIX exp	exp	export file
h	h	C++ source
hpj	hpj	Windows® help project file
hpp	hpp	C++ source
> AIX imp	imp	import file for shared library
ipf	ipf	source for Information Processing Facility compiler
lib	> AIX a lib	> AIX static library > OS/2 > WIN static or import library
mak	mak	makefile
> OS/2 msgbind		message file
> OS/2 mkmsgf		text file in OS/2® message format
> AIX o	o	object file for linking
obj	> AIX o > OS/2 > WIN obj	object file for linking
> WIN rbj	rbj	resource object file
rc	rc	resource source
res	res	resource binary for linking
> AIX so	so	shared object file for linking
sqc	sqc	C source with embedded SQL
sqx	sqx	C++ source with embedded SQL

When you list source files directly in the configuration file, you can view and manipulate them as objects in the IDE.

Primary vs Secondary Sources

Source files that are listed directly in the configuration file, no matter what file type they are, are considered *primary* sources. *Secondary* sources are any files that are included by any other file in the project, but are not listed in the configuration file. For example:

```
target "t.exe"
{
    option macros(global)
    {
        source "stdio.h", "string.h", "other.h"
    }
    option macros(global), define("MAIN", "")
    {
        source "myglobal.h",
    }
    source "one.cpp" , "theRest.cpp",
}

```

Any files which might be included by the .cpp or .h files in this configuration file are considered *secondary* sources. When a secondary source file is added directly to the configuration file (in other words, made primary), it is *promoted*.

Wherever possible, promote your secondary sources to primary ones because of the following characteristics of secondary sources:

- The text of a secondary source file is included at each #include statement in any other source, resulting in a larger codestore and potentially longer preprocessing and compilation times.
- Options settings in effect at the point of the inclusion apply to the included file. This means that the same secondary source may be compiled multiple times with different or even unintended options settings. For example, if macros are defined in the primary source file or the configuration file for the primary source that includes the secondary source, they are also defined in the secondary source. This may not have been your intent.
- Any macros defined in a secondary source file become global macros if a macro source file includes the secondary source.

Macro vs Non-macro Sources

A macro source file defines macros used by other source files. C++ constructs such as functions, classes, and variables do not need to be forward-declared before use in the orderless programming environment of VisualAge C++. However, the order in which the compiler encounters macros is important: a macro must be encountered before it is used. If you promote a macro source file to your configuration, you need to indicate to the compiler that the source file contains macros used by other source files, and therefore should be processed before non-macro source files.

To instruct the compiler to treat a file as a macro source, enclose the source name in the option macros(global). For example:

```
//1. primary, macro source:
option macros(global)
{

```

```

source "tree.hpp", "btree.hpp"
}
//2. primary, non-macro source:
source "one.cpp", "two.cpp"

```

If your configuration includes more than one macro source file, pay attention to the order in which these files are listed. The compiler follows that order to preprocess macro source files and the macros within them. For example, if header file `btree.hpp` depends on macros defined in header file `tree.hpp`, and both files are identified as macro source files as in the above example, `tree.hpp` should precede `btree.hpp` in the configuration file.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2

“Targets”

C++ Source Files and Source Regions

RELATED TASKS

Add or Change Source Files

Convert Included Sources to Primary Sources

“Group Source Files in a Configuration” on page 29

RELATED REFERENCES




source Configuration File Directive

group Configuration File Directive

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37

Targets

In a configuration file, a target can be one of the following:

- executable file
- object file
-   dynamic link library file
-  shared library
- static library


The files produced by a VisualAge C++ build are specified in the configuration file. The target directive controls the name and type of the target of a build.

A target is described with the target keyword followed by open and close braces, usually listing the sources that must be used to create it.

```

target "test.exe"
{
// options and sources are usually listed here
}

```

A target is represented in the IDE with the  symbol.

The following table shows the common target types that are recognized by VisualAge C++ builds. If a type is not recognized in the type clause on all platforms, the platforms to which it applies are shown in the first column. The file extensions associated with each target type are shown in the second column. To

make your configuration file compatible with many platforms, types specified in the type clause are mapped to different file extensions if required by the target platform.

type Clause	File Extension	Type of Target
> AIX a	a	static library
> OS/2 dld	dld	dynamic loadable driver
dll	> OS/2 > WIN dll > AIX shr	> OS/2 > WIN dynamic link library > AIX shared library
exe	> OS/2 > WIN exe > AIX no extension or exe.	executable
lib	> OS/2 > WIN lib > AIX a	static library
> AIX o	o	object file
obj	> OS/2 > WIN obj > AIX o	object file
> OS/2 pdd	pdd	physical device driver
> AIX shr	> AIX shr	shared library
> AIX so	> AIX so	shared object
> OS/2 vdd	vdd	virtual device driver

> OS/2

> WIN OS/2 and Windows NT[®] operating systems use file extension .dll for DLLs and file extension .exe for executable programs. You should not override these types or use their file extensions for other kinds of targets.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2
“Sources” on page 9

RELATED TASKS

Change a Build Setup
Build
Produce Object Files from a Build
Produce Multiple Targets from One Build

RELATED REFERENCES

target Configuration File Directive

Variables and Environments

Variables can be set at any level of your working environment:

- The system level (environment variables)
- The build level (in the project file)
- The subproject level (in the configuration file)

The variables discussed on this page are all set in the project file. See the Related Concepts below for information on variables set in a configuration file.

There are various ways of representing environments and variables in the IDE:

Asn

an *assignment* is a single variable, which you create. In the project file, it takes the form *assignment=value*, for example:

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

You can assign any kind of value (for example, a string or a number) to assignment variables you create. When you want to refer to an assignment in a configuration file, you must precede the assignment name with a '\$'.

Var

a *variables object* represents a group of assignments. Like a group of options or a group of source files, it must have a name, and can contain any number of assignments.

The definition of a variables object takes the general form:

```
variables group_name
{
assignment1 = x
assignment2 = y
}
```

In the project file, variables objects are associated with builds or with subprojects, by use directives. For example, here is a build that makes use of the variables group called `opt_settings`:

```
build optimized
{
  use opt_settings
  use subproject_1
  use subproject_2
}
```



Similarly, the definition of a subproject can include a use directive to associate it with a variables object:

```
subproject sample1 icc "sample1.icc" ics "sample1.ics"
{
  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 defined in the project file, however. You can create your own environments on the Environments page of the Project section of the workbook. They can then be associated with run specifications, when you run or debug your application.

Individual variables are set in the project file. These appear in the IDE as assignment objects (), and you can create and modify them in the IDE just as you can create and modify any other objects. A named group of these assignments appears as a variables object: . A variables object behaves in the same way as a named group of sources in a configuration file or a named group of subprojects in a project file.

You can view the system environment variables in the Change Environment view. The values set in the system are listed under the heading **Evaluation context**.

RELATED CONCEPTS

Variables in Configuration Files

RELATED REFERENCES

“Environment Variables in Configuration Files”

Environment Variables in Configuration Files

In the project file, use environment variable names as they exist at the system level. In the configuration file, put the character \$ at the beginning of the names of environment variables. For example, the if directive in the following line checks whether an environment variable named TEST has been defined:

```
if defined($TEST) { /* ... */}
```

The value and type of environment variables are set by the environment and cannot be changed within the configuration file. You can use various types of environmental variables:

- Your own environment variables
- Those that are predefined by VisualAge C++
- Any that are set by other processes.

Use the operating system to define your own environment variables before starting the build.

The most efficient way to use environment variables whose values are needed more than once is to read them into a project variable, then use the project variable from then on.

Environment variables can appear in expressions or in “Configuration File Directives” on page 4 where string values are expected. You can test whether they are defined and use them in a conditional expression to control the logic of the configuration file. For example, the following if directive sets options only when an environment variable named OS2 is set:

```
if $OS2 {  
    option generalopt = incl(searchPath, ".")  
    option targetops = link(linkWithMultiThreadLib), opt(size)  
}
```

Environment settings are represented in the IDE with the  symbol.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2

“How Configuration Files are Processed” on page 8
Conditional Processing in Builds

RELATED TASKS

Change a Build Setup
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37
Build Options
Variables in Configuration Files
Predefined Build Environment Variables

Chapter 2. About Projects and Subprojects

Build in a Team Environment

When several developers are working on the same project, their individual configurations can be assembled into the overall application using *project files* (.icp). With a project file, you can split up an application development effort into a number of subprojects, each containing their own configuration file (.icc), codestore (.ics), and environment. The project file is used by the IDE and the command line driver to coordinate the building of the various configurations.

Your team may be able to take advantage of features for file change control that allow the following actions on a central repository of source files:

- Check files in
- Check files out
- Extract files

The host section shows which files are currently checked out by other users.

RELATED CONCEPTS

“Projects and Subprojects”

“Incremental Configuration Files” on page 2
Codestore

RELATED TASKS

Build

Check In, Check Out, or Extract a Source File

RELATED REFERENCES

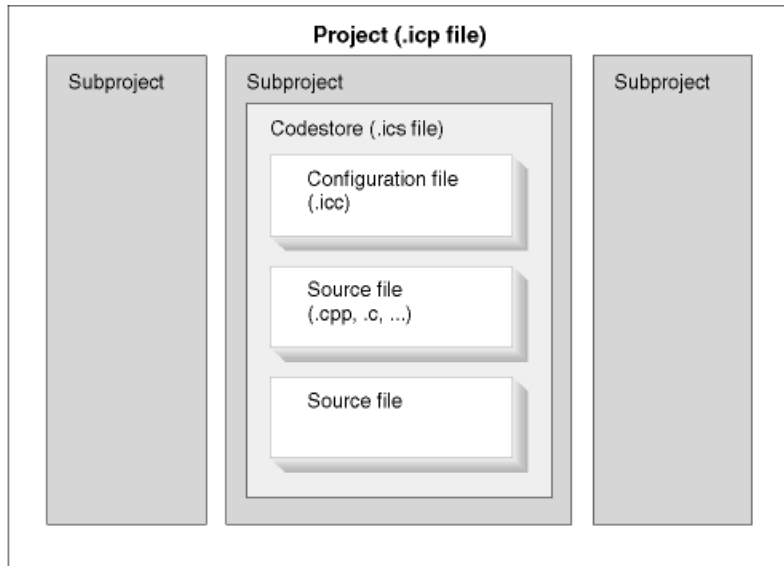
“Project File Syntax” on page 21

Build Options


Projects and Subprojects

A project is a collection of work done by a group of application developers. In VisualAge C++, you can build entire complex applications by bringing together various components or modules in a project file.

In previous versions of VisualAge C++, a project was a set of source files which made up an application. Now, projects can include subprojects called configurations, which can be built separately. Basically, what was a project in previous versions is now a subproject. Subprojects can be specified as configuration(.icc) files or as codestore (.ics) files.




Integral to any project is the environment you specify. A project file can include environment variables, which set conditions that determine how the application is built.

A project is represented in the IDE with the  symbol.

A subproject consists of one to three elements. These elements are specified in the project file to bring together multiple parts of an application and create a single cohesive application:

- A configuration file (required)
- A codestore (which is a file if you are using a persistent codestore)
- A related environment (optional).

The most logical way to subdivide a project is to create subprojects that correspond to the working groups that are assigned to the different parts of the project. For example, a team working on the user interface might have one subproject and the team working on the accounting logic might have a separate subproject. A project file would be created to bring the two elements together in the finished application.

A subproject is represented in the IDE with the  symbol.

RELATED CONCEPTS

“Project Files” on page 19

Configurations

Integrated Development Environment (IDE) Workbook

Codestore

RELATED TASKS

Create a Project

Create a Configuration

“Build in a Team Environment” on page 17

Project Files

The purpose of a project file is to describe one or more builds for your project. A build generally contains one or more subprojects and some variables to control how the processing of each subproject will proceed.

Project (.icp) files are text files that describe builds consisting of one or more subprojects. A subproject consists of a configuration (.icc) file, a codestore, and an environment. The IDE and command line driver use the project file to coordinate the builds of different configurations. Aside from naming the configuration file and codestore, each configuration names the following:

- Antecedents (those configurations that must be completed first, whose targets become source for the current configuration).
- Environment variable settings that are passed into the IDE and are used to process the individual configurations.

The .icp files are similar to the .icc files in a number of ways:

- Some directives in .icp files contain block-structured, named stanzas, of the form *directive-keyword stanza-name { directives }*.
- The .icp files can be edited in both the IDE and as text files.
- Processing of .icp files is incremental. In this way builds can be modified and the incorporations can be restarted directly from the IDE.

The syntax of project files is similar to that of configuration files. However, their purpose is quite different, and so their semantics are different as well. In particular, names are scoped so environment variables can be easily overridden.

How Project Files Are Processed

A project file brings together subprojects and describes how to build them. Environments can be different, even on different builds of the same subproject. Subprojects can also be mixed and matched in different build directives.

A project file is processed in the following steps:

1. The project file is parsed, each token interpreted by the VisualAge C++ builder.
2. Subprojects are ordered by dependencies.
3. Independent subproject builds are started, with a separate environment (options, environment variables, and so on) for each subproject.
4. If two subprojects are interdependent, one is picked and built as far as possible, then the other is built.
5. Steps 2 to 4 are repeated until all the subprojects are complete or until no further progress can be made.

“Example: Project File” on page 20

RELATED CONCEPTS

“Projects and Subprojects” on page 17
Codestore

RELATED TASKS

Create a Project
View a Build Setup (Configuration File)
Change a Build Setup (Configuration File)

Example: Project File

```

// Builds even.so
subproject even icc "even.icc"{
depends on odd
}
// Builds odd.shr
subproject odd icc "odd.icc"{
depends on even
}
// builds console.o
subproject console icc "console.icc"{
depends on odd
depends on even
}
// builds proj (executable)
subproject all icc "all.icc"{
depends on odd
depends on even
depends on console
}
group pieces = even, odd, console, all
// Use this to build everything with no library
subproject simple icc "proj.icc"
build proj{
use pieces
}
build together{
use simple
}

```

Related Configuration Files

even.icc

```

option link(force),link(runtime)linking){
    target type(so) "even.so" {
        source type(cpp) "odd.h"
        source type(cpp) "even.cpp"
        source type(shr) "odd.shr"
    }
}

```

odd.icc

```

option link(force),link(runtime)linking){
    target type(shr) "odd.shr" {
        source type(cpp) "odd.cpp"
        source type(cpp) "even.h"
    }
}

```

console.icc

```
target "console.o" {
    source type(cpp) "console.cpp"
    source type(cpp) "decl.h"
    source type(cpp) "iostream"
}
```

all.icc

```
option link(runtimeLinking),
    link(libSearchPath, ".") {
    target "proj" {
        source type(o) "console.o"
        source type(so) "even.so"
        source type(shr) "odd.shr"
    }
}
```

RELATED CONCEPTS

“Project Files” on page 19

“Incremental Configuration Files” on page 2

RELATED TASKS

Change a Build Setup

Set Build Options

Build

RELATED REFERENCES

“Project File Syntax”

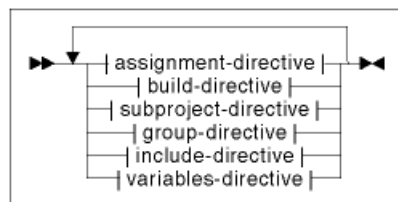
“Configuration File Syntax” on page 6

Build Options

Project File Syntax

Project files can have comments in them. You can use the standard C++ comment style: `//..newline`, or `/*..*/` pairs, just like you can in configuration files.

Project File Program

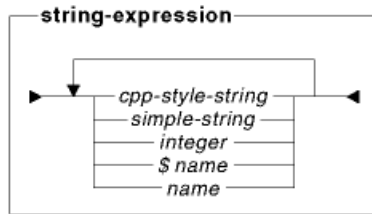


String Expressions

Many directives use string expressions. A string expression is a list of string components. When a string expression is to be evaluated, all components are evaluated and concatenated.

String Component

There are five types of string component:



Cpp-style-string is string specified in double quotes. Escape sequences within this type of string are transformed into single-byte values.

Simple-string is specified in single quotes. The contents between the quotes is the value of the string. There is no escape sequence transformation.

Integer is string containing one or more numerical characters 0 - 9.

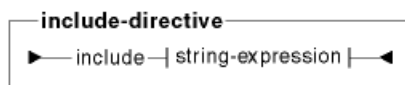
\$name is an environment variable. This string component evaluates to the value of environment variable 'name'. If environment variable is not defined, it evaluates to an empty string.

The last type of string component is a local variable. The value of a local variable depends on the directives evaluated prior to variable evaluation. If the variable is not known when its value is used in a string expression, an error message is generated.

Local variables can be set individually, or they can be set in named stanzas, which can then be reused. Environment variables can be set and reset.

Include Directive

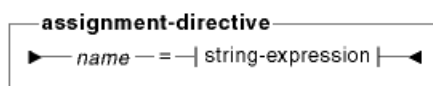
Project files can be split into separate files, and combined with an include-directive.



An include directive is evaluated in two steps: the string expression is evaluated first, then the file named by the expression value is opened and its content is evaluated.

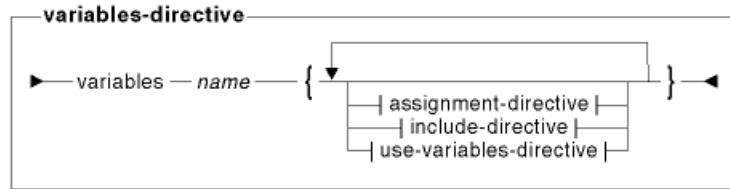
Assignment Directive

This directive assigns the value of a string expression to a local variable:



Variables Directive

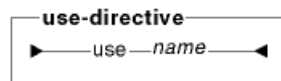
Variables directive sets values to a set of local variables in a named stanza:



The name specified after keyword `variables` is the name of this stanza type directive. Files included by an `include-directive` in this stanza may only contain `assignment-directives`, `use-variables-directives` and other `include-directives`.

Use Directives

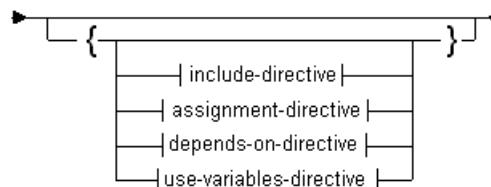
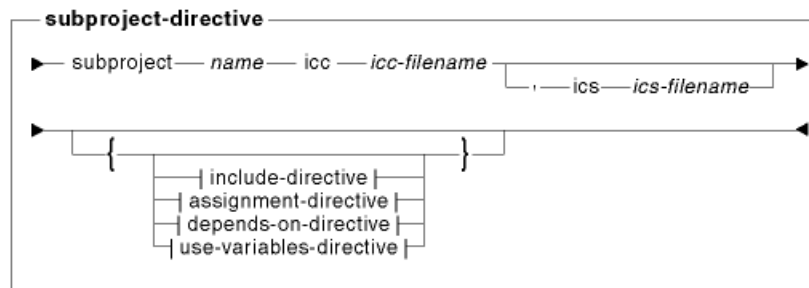
The `use-directive` provides a way to reuse the contents of a stanza type directive.



name refers to a variable or subproject directive. In this grammar we use `'use-variable-directive'` and `'use-subproject-directive'` to indicate what type of `'use'` directive we are dealing with.

Subproject Directive

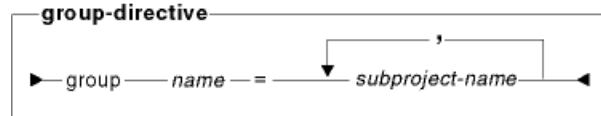
The `subproject-directive` describes a single configuration. It contains `.icc` and optionally `.ics` filenames. Additionally, environment variables can be defined and `variables-directives` used. Files containing directives can be textually included, and `depends-on-directives` can be supplied to order this configuration in the build.



`icc-filename` and `ics-filename` are string expressions. Note that the stanza is optional.

Subproject Group Directive

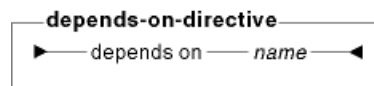
The group-directive defines a named group of configurations. The group name can be used in a use directive. The directive simplifies usage of configurations within builds: a single use-group-directive replaces a list of use-subproject-directives.



subproject-name-list is a list of comma - separated configuration or group names.

Depends On Directive

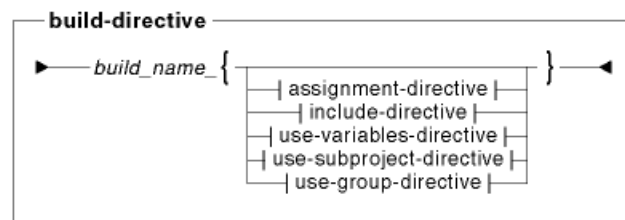
The depends-on-directive is used to declare a dependency between configurations. If a configuration, named Y, uses some results from configuration X , then the depends-on-directive is used to describe this relation. Ideally, it would be nice to have X completed before Y is started. In the most general case, however, two or more configurations can be mutually dependent and simple ordering of configurations is not possible. Dependency relationship makes possible planning of configuration processing order, identification of configuration that may be executed in parallel etc.



A depends-on directive may only appear in a 'subproject' stanza. The specified name is the name of another configuration.

Build Directive

The build-directive names all of the configurations that, taken together, comprise the project itself. You must list all subprojects;when a subproject depends on another subproject both subproject names must be in the build stanza.



"Example: Project File" on page 20

RELATED CONCEPTS

"Project Files" on page 19

Codestore

"Incremental Configuration Files" on page 2

"How Configuration Files are Processed" on page 8

RELATED TASKS

Change a Build Setup
Build


RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37
Project File Directives
Build Options

Chapter 3. Using the Incremental Compiler

Build

There are several ways to start a build in the IDE:

- Click the Build button  on the Workbook toolbar.
- Select **Build** from the **Project Workbook** menu, and select one of the objects from the cascading menu.
- Press **Ctrl+Shift+B** to build the object currently identified in the information area at the bottom of the IDE window.
- Click the **Build** button in the Overview page of the Project section to build the object that is currently selected in the view.

The differences between building a complete project, building a subproject, and building only a codestore are described in *Incremental C++ Build*, in the Related Concepts below.

The information area at the bottom of the IDE window displays the object that is being built:



You can change the object to be built with the selector arrow next to the object name. If you are not sure which object will be built, hold the mouse pointer over the Build toolbar button. The hover help shows the name of the object.

All of your source file changes are saved automatically before the build starts. It does not matter which page is displayed when you start the build.

For performance reasons, the codestore is not saved to disk on every build by default. You can adjust how often the codestore is automatically saved on the Settings page in the Workbook section.

Progress messages and the results of your build are displayed in the information area at the bottom of the IDE window. Build messages are displayed in a Messages view.

Rebuilding After Moving Source Files

If you attempt to rebuild a subproject after deleting a source file or included source file (e.g. a .cpp or .hpp file) from its original path, the build will fail. The build also fails if you move a source file or included source file from its original location to a path not searched elsewhere in the project. If you move a source file from its original path to a path searched for any other source file in the project, the file will be found and the subproject will build successfully.

RELATED CONCEPTS

Configuration Files
Codestore
Incremental C++ Build

RELATED TASKS

View Build Messages
Control Build Messages
Build from the Command Line

RELATED REFERENCES

Dynamic Default Build Setting

Build from the Command Line

Before you build with VisualAge C++, you must have a project file or configuration that defines the sources or input files, targets or output files, and options that apply to the build. To build, open the VisualAge C++ command line window and issue the **vacbld** command.

C++

The **vacbld** tool performs incremental builds for C++ sources. If you have an existing codestore, **vacbld** updates it. If you do not have a codestore, **vacbld** creates one.

You can also use **vacbld** to compile C code.

Specify the same project file, configuration file, or codestore for command line builds that you use for builds started from the IDE. You can make changes to your project file, configuration file, or source code with any text editor, even when the IDE is open to that particular project. When you return to the IDE, you are asked if you wish to reload the changed file. The next build, regardless of whether it is launched by the **Build** button in the IDE or the **vacbld** command, synchronizes the codestore with C++ source files and the configuration file.

To monitor the progress of a command line build, see all warning messages, or select the level of error messages to detect, set options of the **vacbld** command. Error messages are sent to the standard error stream, `stderr`.

RELATED CONCEPTS

“Project Files” on page 19
“Incremental Configuration Files” on page 2
“When to Use Makefiles” on page 35

RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

Build Options
List of Batch Compiler Options and Their Defaults

Build Executable Programs

To build an executable program, make sure your target type is `exe`. Executable files, by default, have no extension.

For example, this excerpt from a configuration file builds a program called 'myprog' from three source files, including a library. The program is built with the multithread version of the run-time library.

```
option link(linkWithMultiThreadLib) {
  target "myprog" {
    source "myprog.cpp", "myprog.hpp", "mylib.a"
  }
}
```

RELATED CONCEPTS

"Incremental Configuration Files" on page 2

RELATED TASKS

Change a Build Setup
Build

RELATED REFERENCES

Build Options
Target Directive

Group Source Files in a Configuration


Grouping individual files makes it easier to deal with multiple source files that are processed in the same manner. To make your configuration file easier to read, you can group source files together and give the group a descriptive name. For example, if your configuration includes a set of header files that contain functions for data processing, you can group them under one group directive called `data_files`, and reference `data_files` in the source directive later.

A group directive takes the general form:

```
group group_name = "source1", "source2", "source 3"... "source $x$ "
```

To reference a group directive as a source, simply list `group_name` in place of the names of the files:

```
source group_name
```

A group directive is represented in the IDE with the  symbol.

RELATED TASKS




Add a Group Directive

RELATED REFERENCES

group Configuration File Directive

Compile and Bind Resources




When using the incremental compiler, building an application that uses resources is a simpler process than it was in earlier versions of VisualAge C++. Simply specify your resource files as source files for a build. You can specify resource files of the following forms:

- One or more resource source files (.rc) files.
The best way to supply more than one resource file is to use `rcinclude` resource compiler statements in a single resource source file that you list in your configuration file. If you list more than one resource source file directly, they are merged into one temporary resource source file, but no check is made for potential problems such as conflicting macro definitions.
-   One compiled resource file (.res).
-  One or more compiled resource file (.res) or resource object file (.rbj).
Resource object files are produced by the Windows resource compiler and you do usually not work with them.

Here is an excerpt from a configuration file. These directives build `myprog` from files `mymyprog.c` and `myres.rc`.

```
target "myprog" {
    source "mymyprog.c"
    source "myres.rc"
}
```

If you have shared or dynamic link libraries for resources, you must load them explicitly at run time. You can use the following API to dynamically load a library:

-  The class `IDynamicLinkLibrary` from the IBM Open Class™ Library
-  The function `DosLoadModule` in the OS/2 Toolkit
-  The function `loadlibrary` in the Windows SKD

You can build shared or dynamic link libraries for resources in the same configuration file that builds an executable file that uses those libraries.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2

RELATED TASKS

Use Resource Source Files as Sources for a Build
Build
“Build Executable Programs” on page 29

RELATED REFERENCES

Build Extensions Supplied with VisualAge C++

Macros in C++ Source Files

You can include preprocessor macros in C++ source files, exactly like you do in source files that you prepare for conventional C++ compilers.

Unlike conventional compilers, VisualAge C++ performs an orderless compile and you can specify many source files in any order. Functions, classes, and variables do not have to be declared before they are referenced, because the build process finds the declarations as it needs them. However, preprocessor macros do have intrinsic position dependencies. The order in which they are defined, undefined, or redefined affects your code. For example, constructs such as the following are common, especially in header files:

```
#ifndef MY_GUARD
#define MY_GUARD 1
...
#endif
```

Because of the ordering requirements of macros, changes to macros cause more recompilation than changes to C++ code.

How VisualAge C++ treats a macro depends on whether the source file in which it is defined is a macro source. A macro source is a C++ source file that is listed in the configuration file and to which the option `macros(global, yes)` applies.

If a macro is defined in a source that is not a macro source, it is local to the source file in which it is defined. The macro is visible only to code that appears after the macro definition in the same file. The macro definition is not visible from other source files.

Macros in macro sources are global. They are visible from the point of definition to the end of the file and continue to be visible in all macro source files that are listed subsequently in your configuration file. Global macros are visible to all sources that are not macro sources.

When you convert a header file to a primary source file, you should also make it a macro source if the header defines macros that are used by other source files. Apply `macros(global)` to the header file, so that macros defined in it are visible in other sources that previously included the header file.

You should specify source files that are not header files as primary sources and either accept the default setting of `macros(global, no)` or explicitly apply `macros(global, no)` to them. Do this to give code in these files the same behavior as in regular C++ compilers.

The only difference between macro sources and other primary sources is in the way macros are preprocessed. The order of macro sources in the configuration file controls the scope of macros defined within them. For processing C++ declarations, macro source files are treated exactly like other source files.

The following build options can affect how macros are processed:

Option	Definition
<code>macros(global[, yes])</code> <code>macros(global, no)</code>	Determines whether source files are macro sources. By default, source files are not macro sources.
<code>macros(redefine[, yes])</code> <code>macros(redefine, no)</code>	Controls whether macros can be redefined without using the <code>#undef</code> preprocessor directive first.
<code>define(macro_name[, string])</code>	Defines macros that can be used in source files to which this option applies.

Option	Definition
<code>undefine(<i>macro_name</i>)</code>	Undefines a macro in the of source files to which this option applies.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2
 Codestore
 C++ Source Files and Source Regions
 “The One-Definition Rule (C++)” on page 52

RELATED TASKS

Convert Included Source Files to Primary Source Files

Search Paths for Included Source Files (AIX)

Included sources are C and C++ source files that are listed in `#include` preprocessor directives, but not listed in a configuration file. If a file is named as a primary source in your configuration file, as well as being included with a preprocessor directive, VisualAge C++ treats it as a primary source and not as an included source.

The search rules depend on whether the included files are system or user included files.

- System include files are included when the `#include` directive takes the form `#include <fileName>`.
- User include files are included when the `#include` directive takes the form `#include "fileName"`.

Searches For System Included Files

For all system included files, VisualAge C++ searches in the following order:

1. The path or paths specified with `incl(searchPath, path)` are searched first.
2. **C++** If you build from C++ sources the directory `/usr/vacpp/include` is searched next.
3. The directory `/usr/include` is searched last.

The last two directories are searched only if you build with the default setting of `incl(standardInclude)`. You can set `incl(standardinclude, no)` to search only the path specified with `incl(searchPath, path)`.

Searches for User Included Files

For user included files, the search path includes the directory in which the file that contains the `#include` directive resides.

C

For C, the file containing the `#include` directive refers to the primary source that directly or indirectly includes other files.

C++

For C++, the file containing the `#include` directive refers to the primary source that directly or indirectly includes other files only when you build with `incl(userStrategy, primary)`.

C++

When you build with `incl(userStrategy, including)`, the file containing the `#include` directive is any C++ source file that directly includes other C++

sources. It may be an included file rather than a primary file, and may be in a different directory from the primary file that included it.

VisualAge C++ follows this order when you build with the default setting of `incl(searchPathFirst, no)`:

1. The directory that contains the source file with the `#include` directive is searched first.
2. The path or paths specified with `incl(searchPath, path)` is searched next.
3. **C++** If you build from C++ sources with the default setting of `incl(standardInclude)`, the directory `/usr/vacpp/include` is searched next.
4. If you build with the default setting of `incl(standardInclude)`, the directory `/usr/include` is searched last.

VisualAge C++ follows the order below when you build with `incl(searchPathFirst, yes)`:

1. The path or paths specified with `incl(searchPath, path)` is searched first.
2. The directory that contains the source file with the `#include` directive is searched next.
3. **C++** If you build from C++ sources with the default setting of `incl(standardInclude)`, the directory `/usr/vacpp/include` is searched next.
4. If you build with the default setting of `incl(standardInclude)`, the directory `/usr/include` is searched last.

RELATED CONCEPTS

“Incremental C++ Builds” on page 1
C Compilation
C++ Source Files and Source Regions
Search Paths for Primary Source Files
Linker Searches

RELATED TASKS

Add or Change Source Files
Set Build Options
Build

RELATED REFERENCES

Miscellaneous Build Options

Cleaning Up After Builds

A clean build is a build that starts with no input other than source files, including the configuration file. The first time you build a new project, VisualAge C++ performs a scratch build. Cleaning up the build environment after builds involves removing all files produced by previous builds so that the next build is the equivalent of the first build for the project.

By default, the build process creates a codestore. If the build is successful, the target files specified in your configuration file are produced. Other intermediate or temporary files may also be produced by the build process, build extensions that you use, or by command line programs that you start with the `run` directive in your configuration file.

You can clean up the build environment after you have performed a number of builds. This is analogous to including a “clean” directive block in makefiles in traditional build environments. You may want clean up the build environment for a number of reasons:

- To package the source files for archiving or distribution.
- During development, to eliminate any problems that you suspect may relate to old information in the codestore or intermediate files.
- To determine how long it takes to rebuild your project from scratch, using the final versions of the source files.

You can always delete the codestore file directly, but cleaning up the build environment is preferable because it ensures that all files produced by a build are deleted.



To clean up the environment, select **Clean** from the **Project Workbook** menu of the IDE and choose the desired codestore from the cascaded menu.. The cleanup flag is passed on to all steps in the build process, including build extensions. For example the resource compiler removes old temporary resource source files during a cleanup. You should take the following additional steps to make sure the environment is completely reset:

- If your configuration file includes run directives that create files, move files, or do other tasks from which you must recover, include one or more run directives that take the form:

```
run cleanup command_string_list
```

The `command_string_list` contains commands that perform the recovery.

- If your build procedure includes steps that happen outside the VisualAge C++ build process, check whether these steps must be repeated or altered after the cleanup.

  For example, on OS/2 and Windows NT platforms you can use the `ilib` command to add individual object modules to a library. You may need to take some action if the files involved are affected by the cleanup.

When the cleanup is complete, the IDE displays a box that asks you whether you want to build your project. If you reply yes, VisualAge C++ performs an initial build as though you are building this project for the first time. If you reply no, a small new codestore is created for the project because it remains open in the IDE. A similar option called `-clean` is available when using the `vacbld` command.

RELATED CONCEPTS

“Projects and Subprojects” on page 17

“Incremental Configuration Files” on page 2

Codestore

Build Extensions and Extension Specification Files

RELATED TASKS

Build

“Build from the Command Line” on page 28

Generate a New Codestore

Run Command Line Programs During Builds

RELATED REFERENCES

When to Use Makefiles

You never have to use a makefile when you develop an application exclusively with the incremental compiler in VisualAge C++. However, the batch compiler, C for AIX, and OpenMP C require the use of makefiles. Therefore, the make utility is available to process makefiles.

In VisualAge C++ configuration files perform much of the role played by makefiles in traditional C and C++ development environments. The only way to perform an incremental C++ build is to process a configuration file.

You may find that not only is it possible to do everything you require from configuration files, but also that using configuration files is simpler and less error-prone than maintaining makefiles. For example, the VisualAge C++ build process implicitly runs the processes required to build the targets you specify in your configuration file from sources you list in your configuration file.

There are two ways to combine processing makefiles with VisualAge C++ builds:

1. You can issue the **vacbld** command inside a makefile to perform a command line build that processes a configuration file.
2. You can specify your makefile as a source in your configuration file. The VisualAge C++ build process can run the make utility implicitly before the compile and link step.

You may want to use makefiles in the following circumstances:

- To set up your build environment, you can set environment variables and run programs from a makefile before processing your configuration file. Using a makefile to set environment variables for the build works only if you start the **vacbld** command from your makefile. When you process a makefile as a source in your configuration file, the environment variable settings made from the makefile are lost when you return to the main build process.
- If your build is complex and requires more than one configuration file, you can issue a sequence of **vacbld** commands from your makefile.
- If you must create directories or move files at specific times during the build process, or run tools that monitor the build, you can encode the necessary commands in a makefile.
- For builds that do not involve any C or C++ code, you can eliminate the overhead of loading the VisualAge C++ kernel by using the command line utilities. For example, you can run the resource compiler directly from the command line, from a makefile, or as part of a build.

If you use a configuration file for some steps in a build and a makefile for other steps, the program database for your build, the codestore, may not accurately reflect the state of your source files. This may cause problems for future builds.

OS/2 **WIN** For example, on OS/2 and Windows NT platforms you can run the library manager from the command line or from a makefile to update libraries. If the library was originally created by a build, the codestore is not updated to incorporate changes made to the library outside the build process.

RELATED CONCEPTS

“Incremental Configuration Files” on page 2
“Incremental C++ Builds” on page 1
VisualAge C++ Batch Compilers
C Compilation
Codestore

RELATED TASKS

Change a Build Setup
Build
“Build from the Command Line” on page 28
Produce Multiple Targets from One Build
Run Command Line Programs During Builds
Use Makefiles as Sources for a Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37
Build Extensions Supplied with VisualAge C++

Chapter 4. Related References

Equivalent Batch Compile-Link and Incremental Build Options (AIX)

VisualAge C++ includes an incremental C++ compiler, with a C compiler extension, and traditional C and C++ compilers. For both C and C++, you can control the compile and link stages of a build by specifying options in a configuration file or in a traditional makefile.

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





The tables below give the configuration file option equivalents to compile options of the traditional batch compiler. In an incremental C++ compile, not all the batch options are meaningful. Some are automatic, and the functionality of others is provided in a different way. An entry of N/A indicates that an equivalent incremental option is not available. Whether the incremental option applies only to C or only to C++ code in this version of VisualAge C++ is indicated in the description line.









In the option syntax, the following rules apply:

- The characters [and] enclose optional items.
- The characters (and) enclose a list of alternatives.
- The character | indicates alternatives.
- The character * denotes zero or more items.









For example `-qkeyword=string[:string]*` is equivalent to `-qkeyword=string`, `-qkeyword=string1:string2`, `-qkeyword=string1:string2:string3` and so on. Another example: `-qtbtable=(none|small|full)` is equivalent to `-qtbtable=none`, or `-qtbtable=small`, or `-qtbtable=full`.







Batch Option	Incremental Option
<code>-#</code>	N/A
Display language commands but do not invoke them; output goes to stdout.	
<code>--+</code>	Use the <code>c++</code> source type parameter on a source directive.
Treat all source files as C++ source code.	
<code>-Bprefix</code>	N/A
Construct alternate program names.. The <code>-t</code> option determines which programs have alternate names.	
<code>-bbigtoc</code> <code>-bnobigtoc</code>	<code>link(bigTOC, yes)</code> <code>link(bigTOC, no)</code>
Generates extra code to allow the table of contents (TOC) to grow to a size greater than 64KB.	
<code>-bD:number</code> <code>-bmaxdata:number</code>	<code>link(heap, number)</code>
Set the size of the heap in bytes. The first number specifies the total virtual address space to reserve. The second number the amount of physical memory to commit initially.	







Batch Option	Incremental Option
-bdynamic -bstatic	N/A
Determines which types of library files are searched by the linkage editor and whether a library is linked dynamically or statically.	
-bl: <i>filename</i>	supported as a source in the configuration: source type(imp) " <i>filename</i> " or source " <i>filename.imp</i> "
Names a file containing a list of imported symbols.	
-bllibpath: <i>string</i>	link(libPathOut, <i>string</i>)
Override search path used at run time.	
-bnoentry	link(entry,no)
Specify that the address of the symbol is the start address of the executable module being built.	
-bnso or -bnautoimp -bso or -bautoimp	link(static, yes) link(static, no)
Specifies if unstripped, shared objects are statically linked as ordinary object files.	
-brtl	link(runtimeLinking)
Tells the linkage editor to search for both .so and .a library file types and to resolve undefined symbols at run time.	
-c	Specify a target with type object file
Do not send object files to the linkage editor.	
-C	 C pp(preprocessOnly)  C pp(preserveComments)
Preserve comments in preprocessed output.	
-Dname[= <i>def</i>]	define(<i>name</i> [, <i>def</i>])
Define preprocessor macro <i>name</i> as in #define directive. If <i>def</i> is not specified, 1 is assumed.	
-Wl,-D <i>number</i> ,-bPD: <i>number</i>	link(dataStart, <i>number</i>)
Set the starting address for the data section of the output file to <i>number</i> .	
-E	 C pp(preprocessOnly)  C pp(stdout)
Preprocess but do not compile. Output goes to stdout.	
-ename	link(entry, <i>string</i>)
Specify that the address of the symbol is the start address of the executable module being built.	
-Fx[: <i>stanza</i>]	supported as a vacbld command option stanzas also supported by defaults(<i>stanza</i>)
Use alternate configuration file or stanza, or both.	

Batch Option	Incremental Option
-g	link(debug[, yes]) link(debug, no)
Include traditional debug information in the target.	
-H <i>number</i>	link(alignAddr, <i>number</i>)
Set the address alignment for all loadable segments so that the start of each segment is aligned at a multiple of the specified number of bytes.	
-I <i>dir</i>	incl(searchpath, <i>path</i>)
Search in directory <i>dir</i> for include files that do not start with an absolute path.	
-K -z	link(alignPage, yes)
Aligns the header, text, data, and loader sections of the output file so that each section begins on a page boundary.	
-l <i>key</i>	N/A
Search the specified library file, where <i>key</i> selects the file <i>libkey.a</i> , or <i>libkey.so</i> if the -rtl option is used.	
-L <i>string</i>	link(libSearchPath, <i>string</i>)
Specify search path for library files.	
-ma	N/A
Generate inline calls to the "alloca" function as if "#pragma alloca" directives were in the source file.	
-M	 file(makeDep[,yes])
Generate information to be included in a "make" description file; output goes to .u file.	
-o <i>name</i>	 file(genObject[,yes])  file(genObject, <i>name</i>)  file(genObject, no) filename specified on target directive
Name generated executable or object file.	
-O -O2 -O3	opt(level, <i>number</i>)
Optimize code.	
 -O4  -O5	 opt(level, <i>number</i>)
Level 4 is equivalent to -O3 -qipa with automatic detection of architecture and tuning option. At level 5, interprocedural calls are also optimized.	
-P	gen(profile[, yes])
Enable code for performance analysis. Enables profiling with prof .	
-pg	gen(profile, bsd)
Generate profiling support code including BSD profiling support. Enables profiling with gprof .	
-P	 pp(preprocessOnly[,yes])

Batch Option	Incremental Option
Preprocess but do not compile. Output goes to .i file.	
-Q -qinline	opt(inline[, yes]) opt(autoInline[, yes])
Consider any function for inlining. The default is to inline only those functions declared inline.	
-Q! -qnoinline	opt(inline, no) opt(autoInline, no)
Do not inline any function.	
 -Q= <i>lc</i>  -qinline= <i>lc</i>	opt(inline) opt(autoInline) opt(autoInline, <i>lc</i>)
Inline if number of source statements in function is less than the number specified in <i>lc</i> .	
 -Q- <i>name[:name]*</i>  -qinline- <i>name[:name]*</i>	opt(inline[, yes]) opt(noInlineFunc, <i>name</i>)
Do not inline function listed by names.	
 -Q+ <i>name[:name]*</i>  -qinline+ <i>name[:name]*</i>	opt(inline[, yes]) opt(inlineFunc, <i>name</i>)
Attempt to inline functions listed by names.	
-qaggrcopy=[no]overlap	N/A
Destructive aggregate copy.	
-qalias=(typ allp addr ansi)	opt(alias,...)
Specifies the aliasing assertion to be applied to your compilation unit. The available options are: typ= Pointers to different types are never aliased. allp= Pointers are never aliased. addr= Variables are disjoint from pointers unless their address is taken. ansi= Pointers can only point to an object of the same type. Requires -O option.	
-qalign= <i>name</i>	gen(align, <i>name</i>)
Specify alignment of data items.	
-qansialias -qnoansialias	opt(alias, ...)
Specify which aliasing rules can be used during optimization.	
-qarch= <i>name</i>	gen(arch, <i>name</i>)
Specifies the architecture on which the executable program will be run. See the AIX processor types page for a list of valid names. The default is -qarch=com. If the -qarch option is specified without the -qtune=<option>, the compiler uses -qtune=pwr.	
 -qassert= <i>option</i>  -qassert=(typ allp addr)	opt(alias, <i>option</i>) opt(aliasAssert,typ[,yes]) opt(aliasAssert,typ,no) opt(aliasAssert,allp[,yes]) opt(aliasAssert,allp,no) opt(aliasAssert,addr[,yes]) opt(aliasAssert,addr,no)





















Batch Option	Incremental Option
Specifies the aliasing assertion to be applied to your compiler unit. The available options are: typ= Pointers to different types are never aliased. allp= Pointers are never aliased. addr= Variables are disjoint from pointers unless their address is taken.	
-qattr	 list(attr[,yes])
Produce an attribute listing; output goes to.lst file.	
-qattr=full	 list(attr, full)
Produce an attribute listing containing all names, whether referenced or not; output goes to .lst file.	
-qbitfields=signed -qbitfields=unsigned	lang(signedBitfields[,yes]) lang(signedBitfields,no)
Specify whether bitfields will be signed or unsigned.	
-qcache	
Specifies the hardware cache characteristics.	
-qchars=signed -qchars=unsigned	lang(signedChars[, yes]) lang(signedChars, no)
Treat plain <i>char</i> variables as signed or unsigned.	
-qcheck -qnocheck -qcheck= <i>suboptions</i>	gen(check, zeroDivide[, yes]) gen(check, zeroDivide, no) gen(check, nullPointer[,yes]) gen(check, nullPointer,no) gen(check, bounds[,yes]) gen(check, bounds,no)
The suboptions are: all= Switches on all of the following suboptions. nullptr= Performs run-time checking of addresses contained in pointer variables used to reference storage. bounds= Performs run-time checking of addresses when subscripting within an object of known size. divzero= Performs run-time checking of integer division.	
 -qcinc= <i>prefix</i>  -qnocinc= <i>prefix</i>	incl(externC, <i>prefix</i>) incl(noExternC, <i>prefix</i>)
Includes files from the subdirectory <i>prefix</i> and inserts extern "C" { at the beginning of the file and } at the end.	
-qcompact -qnocompact	opt(size[, yes]) opt(size, no)
Optimize code for size.	
 -qplusplusmt  -qnoplusplusmt	 lang(slashSlashComment[,yes])  lang(slashSlashComment,no)
Permit <code>"/ /"</code> to introduce comment that lasts until the end of the current source line, as in C++.	






Batch Option	Incremental Option
-qdataimported -qdataimported= <i>name[:name]*</i>	link(dataImported, none) link(dataImported, all) link(dataImported, names) link(dataImportedNames, <i>name</i>)
Specify which data items are imported. If no names are specified, all data items are assumed to be imported. This is the default.	
-qdatalocal -qdatalocal= <i>name[:name]*</i>	link(dataLocal, none) link(dataLocal, all) link(dataLocal, names) link(dataLocalNames, <i>name</i>)
Specify which data items are local. If no names are specified, all data items are assumed to be local.	
-qdbcs -qnodbcs	lang(dbcs[,yes]) lang(dbcs,no)
Allow use of DBCS.	
 -qdbxextra  -qnodbxextra	gen(debugunreferenced[, yes]) gen(debugunreferenced,no)
Produce a symbol table for unreferenced variables.	
-qdigraph -qnodigraph	lang(digraphs[, yes]) lang(digraphs, no)
Permit ANSI digraph and keyword operators.	
-qdollar -qnodollar	lang(allowDollarInNames[, yes]) lang(allowDollarInNames, no)
Allow the dollar sign ('\$') in identifier names.	
-qdpcl -qnodpcl	link(dpcl[, yes]) link(dpcl, no)
 -qeh  -qnoeh	 gen(eh[, yes])  gen(eh, no)
Controls whether C++ exception handling is enabled in the module being built.	
-qenum=(small int intlong 1 2 4 8)	gen(enumSize, ...)
<p>Small denotes that either one, two or four bytes of storage will be allocated for enum variables based on the range of the enum constants. int is the default, and causes enum variables to be treated as though they were of type signed int for C and either signed or unsigned int for C++ depending on the range of the enum constants.</p> <p>intlong causes enum variables to be treated as though they were of type int or type long depending on the range of enum constants. The sign of the type is determined in the same way as for int.</p> <p>1 will cause enum variables to be packed into 1 byte. 2 will cause enum variables to be packed into 2 bytes. 4 will cause enum variables to be packed into 4 bytes. 8 will cause enum variables to be packed into 8 bytes.</p>	

Batch Option	Incremental Option
-qextchk -qnoextchk	link(typeCheck[, yes]) link(typeCheck, no)
Perform external name type-checking and function call checking.	
-qfdpr -qnofdpr	gen(fdpr, yes)
Collect information about programs for use with the AIX fdpr (Feedback Directed Program Restructuring) performance-tuning utility.	
-qflag= <i>sev1[:sev2]</i>	list(reportLevel, sev1) report(level, sev2)
Specify severity level of diagnostics to be reported in source listing <i>sev1</i> , and stderr <i>sev2</i> ; the severity levels are one of i,w,e,s.	
-qfloat= <i>opt[:opt]*</i>	gen(float, ...) gen(float, <i>string</i> [,yes]) gen(float, <i>string</i> ,no)
Specifies various floating-point options. These options provide different strategies for speeding up or improving the accuracy of floating-point calculations. They affect the generated code, but not operations performed in library functions.	
-qflttrap= <i>opt[:opt]*</i> -qnoflttrap	gen(floatTrap, <i>option</i> [, yes]) gen(floatTrap, <i>option</i> , no)
Generate calls to detect and trap floating point exceptions. The available options are: overflow, underflow, zerodivide, invalid, inexact, enable, imprecise.	
-qfold -qnofold	gen(float, fold[, yes]) gen(float, fold, no)
Specifies that constant floating-point expressions are to be evaluated at compile time.	
-qfullpath -qnofullpath	link(debugFullPath[, yes]) link(debugFullPath, no)
Specify path information.	
-qfuncsect -qnofuncsect	gen(funcSect[, yes]) gen(funcSect, no)
Place instructions for each function in a separate csect.	
 -qgenpcomp[= <i>filename</i>]	N/A
Generate a precompiled header file; <i>name</i> can either be a directory containing the precompiled header (named csetc.pch) or the name of the precompiled header.	
 -qgenproto  -qnogenproto	 file(genProto[, yes])  file(genProto, no)
Generate ANSI prototypes from K&R function definitions.	
 -qgenproto= <i>parmnames</i>	 file(genProto, withParameterNames)
Produces ANSI prototypes from K&R function definitions with the names of parameters included.	
-qhalt=(i w e s)	N/A
Stop the compiler after the first phase if the severity of errors detected equals or exceeds <sev>; the severity level is one of i, w, e, s, which stand for information, warning, error, and severe.	
-qhaltmsg= <i>message[:message]*</i>	report(treatAsError, <i>message</i>)









Batch Option	Incremental Option
Treats the specified message like an error message (preventing the creation of the target).	
-qheapdebug -qnoheapdebug	alloc(debug[,yes]) alloc(debug,no)
Enables debug versions of memory management functions.	
-qidirfirst -qnoidirfirst	incl(searchpathfirst[, yes]) incl(searchpathfirst, no)
Specify search order for files.	
-qignerrno -qnoignerrno	opt(ignErrno) opt(ignErrno, no)
Tell the optimizer that the program will never refer to or set errno.	
-qignprag=disjoint -qignprag=isolated -qignprag=all	opt(pragmaDisjoint[, yes]) opt(pragmaDisjoint, no) opt(pragmaIsolatedCall[, yes]) opt(pragmaIsolatedCall, no)
Honor or ignore references to #pragma disjoint and/or isolated_call.	
-qinfo=class	info(class)
Produce additional lint-like messages based on class.	
-qinitauto=number	gen(initAuto[, yes]) gen(initAuto, number) gen(initAuto, no)
Initialize automatic storage to hh, where hh is a hexadecimal value. A number is a hexadecimal number if it begins with 0x, an octal number if it begins with 0 or a decimal number if it begins with 1-9. This option generates extra code and should only be used for error determination	
-qinlglue -qnoinlglue	gen(inlinePointerGlue[, yes]) gen(inlinePointerGlue, no)
Generate fast external linkage by inlining the code (pointer glue code) necessary for calls via a function pointer and calls to external procedures.	
-qinline -qnoinline	opt(autoInline[, yes]) opt(autoInline, no) opt(inline[, yes]) opt(inline, no)
Determine whether functions in your code are candidates for inlining. See the various -Q options.	
<pre> > C -qipa > C -qnoipa > C -qipa=option </pre>	N/A
Turn the IPA optimizer on or off.	
-qisolated_call=name[:name]*	opt(isolatedCall,...)
Specify that the calls to the function name listed have no side effects.	
<pre> > C++ -qkeyword=string[:string]* > C++ -qnokeyword=string[:string]* </pre>	lang(keyword, string)
Controls whether the specified string is treated as a keyword or an identifier whenever it appears in your C++ source.	

Batch Option		Incremental Option
-qlanglvl	<ul style="list-style-type: none"> ➤ C++ = [no]anonstruct ➤ C++ = [no]anonunion ➤ C++ = [no]ansifor ➤ C++ = [no]illptom ➤ C++ = [no]implicitint ➤ C++ = [no]newexcp ➤ C++ = [no]offsetnonprod ➤ C++ = [no]olddigraphs ➤ C++ = [no]oldfriend ➤ C++ = [no]oldmath ➤ C++ = [no]oldtmplacc ➤ C++ = [no]oldtmplalign ➤ C++ = [no]oldtmplspec ➤ C++ = [no]trailenum ➤ C++ = [no]typedefclass = [no]ucs ➤ C++ = [no]zeroextarray 	lang(anonymousStructs[, yes]) lang(extendedAnonymousUnions[, yes]) lang(ansiForStatementScopes[, yes]) lang(illformedPointerToMember[, yes]) lang(implicitInt[, yes]) lang(newThrowsException[, yes]) lang(offsetOfNonPODClasses[, yes]) lang(oldDigraphs[, yes]) lang(compatFriendDeclarations[, yes]) lang(compatMath[, yes]) lang(compatTempAccessChecking[, yes]) gen(compatNestedTemplateAlignmentRule) lang(compatTemplateSpecialization[, yes]) lang(trailingEnumCommas[, yes]) lang(allowTypedefAsClassName[, yes]) lang(universalCharacterNames[, yes]) lang(zeroExternArrays[, yes])
Set the language level option.		
➤ C++ -qlanglvl=<compat366 strict98 ansi extended>		options for compatibility with older C++ compilers options for compatibility with the C++ standard
Specify the language level to be used during compilation.		
➤ C -qlanglvl=<ansi saa12 saa extended classic>		lang(level,...)
Specify language level to be used during compilation.		
-qdbl128 -qnoldbl128 -qlongdouble -qnolongdouble		gen(longDouble[, yes]) gen(longDouble, no)
Represent long doubles as 128 bit or 64 bit values.		
-qlibansi -qnolibansi		gen(libansi) gen(libansi, no)
Process functions with names that match ANSI C library names as being the ANSI C functions. This allows the compiler to make certain assumptions about the behavior of these functions.		
-qlinedebug -qnolinedebug		gen(lineOnlyDebug[, yes]) gen(lineOnlyDebug, no)
Generates abbreviated line number and source file name information for the debugger.		
-qlist -qnolist		➤ C list(listing[, yes]) ➤ C list(listing, no)
Produce an object listing; output goes to .lst file.		

Batch Option	Incremental Option
-qlistopt -qnolistopt	 list(options[, yes])  list(options, no)
Display the settings of all options; output goes to .lst file.	
-qlonglong -qnolonglong	lang(longlong) lang(longlong, no)
Build process disallows long long int.	
-qlonglit -qnolonglit	N/A
Makes unsuffixed literals into the long type in 64-bit mode.	
 -qmacpstr  -qnomacpstr	 lang(macPStr[, yes])  lang(macPStr, no)
Allow '\p' as the first character of a string constant.	
-qmakedep -qnomakedep	 file(makeDep[,yes])  file(makeDep,no)
Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.	
-qmaxerr= <i>number</i> [: <i>severity_level</i>]	 debug(maxErrors, <i>number</i>)
Instructs the compiler to halt compilation when a specified number of errors is reached.	
-qmaxmem= <i>num</i>	opt(maxMem, <i>number</i>)
Limit the amount of memory used by space intensive optimization to <i>num</i> . <i>num</i> is specified in kilobytes.	
-qmbcs -qnombcs	lang(dbcs[, yes]) lang(dbcs, no)
Allow use of DBCS.	
-qnamemangling= <i>option</i>	link(nameMangling, ansi) link(nameMangling, compat)
Chooses the name mangling scheme for external symbol names generated from C++ source code.	
-qnoprint	 list(listing, no)
Direct listing to /dev/null.	
 -qonce  -qnoonce	 file(once[, yes])  file(once, no)
Process #include files only once.	
 -qobjmodel=(ibm compat)	 gen(objectModel, <i>model</i>)
Select the default C++ object model for the compilation unit.	
 -qpascal  -nopascal	 lang(pascal[, yes])  lang(pascal, no)

Batch Option	Incremental Option
Accept and ignore the keyword 'pascal' as a type modifier.	
-qpath= <i>components:path</i>	N/A
Construct alternate program names. The programs in the directory <i>path</i> and specified by <i>components</i> are used instead of the regular ones. <i>components</i> can a combination of any of the following: p= preprocessor c= compiler frontend b= compiler backend i= compiler inliner a= assembler l= linkage editor m= 'munch' utility E= create export list I= inter procedural analysis	
-qpdf1 -qpdf2	opt(profileDirectedFeedback[,yes])
Perform aggressive optimizations with profile directed feedback.	
-qphsinfo	N/A
Produce compiler phase information	
 -qpriority= <i>num</i>	 link(sharedLibPriority, <i>number</i>)
Specifies the priority level for initialization of static constructors.	
-qproclocal -qproclocal= <i>name[:name]*</i>	link(procLocal,< <i>none</i> <i>all</i> <i>names</i> >) link(procLocalNames, <i>string</i>)
Specify which functions are local. If no filenames are specified, all invoked functions are assumed to be defined within the current file. The last explicit specification for a function takes precedence.	
-qprocimported -qprocimported= <i>name[:name]*</i>	link(procImported,< <i>none</i> <i>all</i> <i>names</i> >) link(procImportedNames, <i>string</i>)
Specify which functions are imported. If no filenames are specified, all invoked functions are assumed to be defined outside the current file. The last explicit specification for a function takes precedence.	
-qprocunknown -qprocunknown= <i>name[:name]*</i>	link(procUnknown,< <i>none</i> <i>all</i> <i>names</i> >) link(procUnknownNames, <i>string</i>)
Specify which functions are unknown to be local or imported. If no filenames are specified, all functions called are assumed to be unknown. This is the default when no user options are specified. The last explicit specification for a function takes precedence.	
-qprofile=ibm	gen(profile, ibm)
 Enables profiling with IBM's Performance Analyzer.	
 -qproto  -qnoproto	 lang(checkNonProto[,yes])  lang(checkNonProto,no)
Assert that procedure call points agree with their declarations even if the procedure has not been prototyped. This allows the caller to pass floating point arguments in floating point registers instead of in general purpose registers.	

Batch Option	Incremental Option
-qro -qnoro	gen(readonly[, yes]) gen(readonly, no)
Put string literals in read only area.	
-qroconst -qnoroconst	gen(readonlyconst) gen(readonlyconst, no)
Put constant values in read only area.	
<pre>> C++ -qrtti > C++ -qnortti</pre>	<pre>> C++ gen(rtti[, yes]) > C++ gen(rtti, all) > C++ gen(rtti, typeInfo) > C++ gen(rtti, dynamicCast) > C++ gen(rtti, no)</pre>
Generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.	
-qshowinc -qnoshowinc	<pre>> C list(expAllInc[,yes]) > C list(expAllInc,no)</pre>
Include the source for all included files in the source listing, if the -qsource option is in effect.	
-qsource -qnosource	list(incSource[,yes]) list(incSource,no)
Produce a source listing; output goes to.lst file.	
-qspill= <i>number</i>	opt(registerSpillSize, <i>number</i>)
Specify the size of the register allocation spill area.	
<pre>> C -qsrcmsg > C -qnosrcmsg</pre>	<pre>> C report(srcMsg[, yes]) > C report(srcMsg, no)</pre>
Reconstruct source lines in error along with the diagnostic messages.	
-qstaticinline -qnostaticinline	lang(staticInlineLinkage[, yes]) lang(staticInlineLinkage, no)
Controls whether inline functions are treated as static or extern.	
-qstatsym -qnostatsym	link(staticSymbols) link(staticSymbols, no)
Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of xcoff objects).	
-qstdinc -qnostdinc	incl(standardInclude[,yes]) incl(standardInclude, no)
If -qnostdinc is specified, the /usr/vacpp/include and /usr/include directories are not searched for include files.	
-qstrict -qnostrict	opt(strict) opt(strict, no)

Batch Option	Incremental Option
Valid only at -O3. This option turns off aggressive optimizations which have the potential to alter the semantics of a user's program. This option also sets -qfloat=nofltn:norsqrt.	
-qstrict_induction -qnostrict_induction	gen(strictinduction[, yes]) gen(strictinduction, no)
Disable loop induction variable optimizations that have the potential to alter the semantics of the program.	
 -qsymtab=unref -qsymtab=static	gen(debugUnreferenced[,yes]) gen(debugUnreferenced,no) link(staticSymbols) link(staticSymbols,no)
Set certain back end options.	
 -qsyntaxonly  -qnosyntaxonly	 file(syntaxOnly[,yes])  file(syntaxOnly,no)
Causes the compiler to perform syntax checking without generating an object file.	
-qtabsize= <i>size</i>	file(tabSize, <i>number</i>)
Change the length of tabs in your source file.	
-qtbtable(non small full)	link(traceBackTable,<none full small>)
Generate a traceback table for each function and place it in the text segment at the end of the function code. The available options are: none= No traceback table is generated. full= A full traceback table is generated. This is the default with -g specified. small= A traceback table is generated with no name or parameter information. This is the default when -g is NOT specified.	
 -qtempinc[= <i>directory</i>]  -qnotempinc	N/A
Automatically generate template functions in the specified directory. Use the "notempinc" option to suppress this behavior.	
 -qtempmax= <i>number</i>	N/A
Specify the maximum number of files to be created in the tempinc directory for each template file.	
-qthreaded -qnothreaded	N/A
Make sure only thread safe optimizations thread safe optimizations are performed.	
-qtmplparse= no -qtmplparse= -qtmplparse=	lang(templateDefImpls, parseWithWarnings) lang(templateDefImpls, parseWithErrors) lang(templateDefImpls, dontParse)
Controls whether parsing and semantic checking are applied to template definition implementations or only to template instantiations.	
-qtune= <i>name</i>	opt(tune, <i>name</i>) opt(tune,...)
Specifies the system architecture for which the executable program is optimized.	

Batch Option	Incremental Option
<pre>> C++ -qtwolink > C++ -qnotwolink</pre>	N/A
Link twice in order to minimize the number of static constructors included from libraries.	
<pre>> C++ -qunique > C++ -qnounique</pre>	<pre>> C++ link(uniqueNames[, yes]) > C++ link(uniqueNames, no)</pre>
Generates unique names for static constructor or destructor compilation units.	
<pre>-qunroll -qunroll=<i>number</i> -qnounroll</pre>	<pre>opt(loopUnroll[, yes]) opt(loopUnroll, <i>number</i>) opt(loopUnroll, no)</pre>
Allow the optimizer to unroll loops.	
<pre>> C -qupconv > C -qnoupconv</pre>	<pre>> C lang(preserveUnsignedPromotion[, yes]) > C lang(preserveUnsignedPromotion, no)</pre>
Preserves the unsigned specification when performing integral promotions.	
<pre>> C -qusepcomp[=<i>name</i>]</pre>	N/A
Use precompiled header files. <i>name</i> can either be a directory containing then precompiled header (named csetc.pch) or the name of the precompiled header.	
<pre>-qvftable -qnovftable</pre>	<pre>gen(vft[, yes]) gen(vft, no)</pre>
Determine whether the virtual function table is included in the module that is the target of the build.	
<pre>-qxcall -qnoxcall</pre>	<pre>gen(externStaticLinkage[, yes]) gen(externStaticLinkage, no)</pre>
Generate code to static routines within a compilation unit as if they were external routines.	
<pre>-qxref -noxref</pre>	<pre>> C list(minXRef[, yes]) > C list(minXRef, no)</pre>
Produce a cross-reference listing; output goes to .lst file.	
<pre>-qxref=full</pre>	<pre>> C list(fullXRef[, yes]) > C list(fullXRef, no)</pre>
Produce a cross-reference listing containing all names, whether referenced or not; output goes to .lst file.	
<pre>-r</pre>	link(force)
Permits the output file to be produced even if it contains unresolved symbols.	
<pre>-s</pre>	<pre>link(strip[, yes]) link(strip, no)</pre>
Strip the symbol table to save space.	

Batch Option	Incremental Option
-S	<div style="background-color: #f0f0f0; padding: 2px; margin-bottom: 2px;">> C file(genAsm[,yes])</div> <div style="background-color: #f0f0f0; padding: 2px;">> C file(genAsm, no)</div>
Generate an assembler language (.s) file	
-Wl, -Snumber	link(stack,number)link(stack, number)
Set the total size of the program stack in bytes.	
-t(p c b i a l m E I)	N/A
Apply the prefix from the -B option to the specified program, where the program can be one or more of the following: p= preprocessor c= compiler frontend b= compiler backend i= compiler inliner a= assembler l= linkage editor m= 'munch' utility E= create export list I= inter procedural analysis	
-Tnumber	link(textStart, number)
Sets the start address of the text section of the output file to <i>number</i> .	
-Uname	undefine(name)
Undefine name as in #undef directive.	
-v	N/A
Display language processing commands as they are invoked by xIC; output goes to stdout.	
-V	N/A
Same as -v except format is similar to shell commands.	
-w	report(level, ...)
Suppress information, language-level, and warning messages.	
<div style="background-color: #f0f0f0; padding: 2px; margin-bottom: 2px;">> C++ -W</div>	N/A
Give specified options to specified compiler program.	
-Wcomponents, option	N/A
Give the <i>option</i> to the compiler program where <i>components</i> can be one or more of the following: p,c,b,i,a,l,m,E. See -t for <i>component</i> meanings.	
-ymode	gen(roundConstFp, mode)
Specify compile-time rounding of constant floating-point expressions, where <i>mode</i> can be one of the following: n= round to nearest m= round toward minus infinity p= round toward positive infinity z= round toward zero.	
-Zstring	link(libsearchpathprefix, string)
Prefix the names of the library search paths with the <i>string</i> .	

RELATED CONCEPTS


“Incremental C++ Builds” on page 1
C Compilation
“Incremental Configuration Files” on page 2


RELATED TASKS

Set Build Options
Build

RELATED REFERENCES

“Equivalent Batch Compile-Link and Incremental Build Options (AIX)” on page 37
Link Options from Earlier Versions of VisualAge C++
Build Options
Batch Compiler Options

 Build Options for ANSI C++ Compliance

 Build Options for Compatibility with Old C++ Compilers

The One-Definition Rule (C++)

The one-definition rule is specified in the C++ standard. The one-definition rule has two aspects:

1. A translation unit must not contain more than one definition of any variable, function, class type, enumeration type, or template.
2. External definitions must not be redefined within a program.

Traditional C++ compilers enforce only the first part of the one-definition rule. The incremental compiler enforces the full rule, to conform to the C++ standard and to remove the possibility of two or more conflicting definitions being visible at the same time during a build. As implemented in the incremental compiler, the one-definition rule means that declarations with external linkage cannot be defined in more than one source file. The one-definition rule allows some declarations, such as class and enum definitions, templates, and inline functions, to be defined in different source files as long as they consist of the same sequence of tokens. The incremental compiler accepts these multiple definitions.

Because the incremental compiler performs C++ builds in which all source files are considered at once, the one-definition rule applies to the set of C++ source files that are input to one build. As a result, the incremental compiler may not accept code that other compilers accept in some circumstances. The circumstances relate to following constructs and are described in detail below:

- Classes, enums, Templates, and Inline Functions
- Enumerations and extern Variables
- typedef Statements

If your C++ sources are rejected by the incremental compiler due to the one-definition rule, you should consider using one of the following methods to correct the problem:

1. Change the name of one of the conflicting definitions.
2. Enclose the code that depends on each of the conflicting definitions in different namespaces.
3. Split your source files so that each of the conflicting definitions is in a separate codestore. Do this by creating a separate configuration file in the same project. For example, you can build DLLs and your executable program from different configuration files.

If you specify the same source file more than once in your configuration file, the incremental compiler treats each occurrence of the file as a separate and unrelated source file. The one-definition rule limits the way you can reuse C++ source files in one codestore.

Classes, enums, Templates, and Inline Functions

The C++ language does not allow classes, enums, templates, and inline functions to be defined more than once in a program, unless all three of the following conditions are satisfied:

- The definitions appear in different source files.
- The definitions consist of the same sequence of identical tokens.
- The definitions are semantically identical.

Traditional compilers test for the first condition. For example, they make sure that classes and enums are defined only once within a source file. The incremental compiler tests for the first two conditions and lets you redefine classes and enums only with definitions that are identical to the original definitions. Therefore, if a class or enum name is used with different definitions in different source files, the incremental compiler reports an error that traditional compilers do not find. While traditional compilers only check this rule within a source file, the incremental compiler checks it across all source files.

Enumerators and extern Variables

The C++ language allows only one definition of a given enumerator or variable name with external linkage in namespace scope, which includes global scope, and insists that multiple declarations of an external variable match. While traditional compilers only check this rule within a source file, the incremental compiler checks it across all source files.

typedef Statements

Standard C++ does not allow definitions with external linkage to be defined differently in different translation units. The incremental compiler applies this rule to types defined in typedef statements, which have no linkage according to standard C++. Unlike traditional compilers, the incremental compiler does not let you redefine a type with typedef statements in more than one source file unless the typedef statements are identical.

If the incremental compiler rejects a typedef statement in your C++ source because of the one-definition rule, you have an additional recovery technique to the three methods listed above. You can tell the incremental compiler to treat all typedef statements as local to the source file in which the type is defined, by building with option `lang(localTypedefs, yes)`.

RELATED CONCEPTS

“Incremental C++ Builds” on page 1

Codestore

“Incremental Configuration Files” on page 2

C++ Source Files and Source Regions

RELATED TASKS

Add or Change Source Files

Convert Included Source Files to Primary Source Files

Produce Multiple Targets from One Build

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