HP aC++/HP C A.06.28 Programmer's Guide

Integrity servers

HP Part Number: 769150-001
Published: March 2014
Edition: 13

Contents

HP secure development lifecycle

Starting with HP-UX 11i v3 March 2013 update release, HP secure development lifecycle provides the ability to authenticate HP-UX software. Software delivered through this release has been digitally signed using HP's private key. You can now verify the authenticity of the software before installing the products, delivered through this release.

To verify the software signatures in signed depot, the following products must be installed on your system:

- `B.11.31.1303` or later version of SD (Software Distributor)
- `A.01.01.07` or later version of HP-UX Whitelisting `(WhiteListInf)`

To verify the signatures, run: `/usr/sbin/swsign -v -s <depot_path>`. For more information, see *Software Distributor documentation* at http://www.hp.com/go/sd-docs.

NOTE: Ignite-UX software delivered with HP-UX 11i v3 March 2014 release or later supports verification of the software signatures in signed depot or media, during cold installation.

For more information, see *Ignite-UX documentation* at http://www.hp.com/go/ignite-ux-docs.

About This Document

This manual presents programming information on the C++ programming language, as implemented on Itanium®- based systems.

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. Contact your HP sales representative for details.

The latest version of this document is available on the web at `http://www.hp.com/go/hpux-C-Integrity-docs`.

Intended Audience

This manual is intended for experienced C and C++ programmers who are familiar with HP systems.

What's in This Document

HP aC++/HP C Programmer's Guide is divided into the following chapters:

Chapter 1	**Getting Started** Gives you an introduction to the HP aC++ product and its components. It also discusses the compiler command syntax and environment variables.
Chapter 2	**Command Line Options** Discusses command line options. Command line options are categorized into different sections based on how you can use them. This chapter also covers diagnostic messages and pragma directives.
Chapter 3	**Pragma Directives** Discusses pragmas supported in HP aC++. A pragma directive is an instruction to the compiler. Use a #pragma directive to control the actions of the compiler in a particular portion of a translation unit without affecting the translation unit as a whole.
Chapter 4	**Preprocessing Directives** Gives you an overview, the syntax, and usage guidelines of preprocessing directives. This chapter also includes a section on using HP aC++ templates.
Chapter 5	**Using HP aC++ Templates** Gives you an overview of template processing and describes the instantiation coding methods available in HP aC++.
Chapter 6	**Standardizing Your Code** Discusses HP aC++ keywords, Standard Exception Classes, and exceptions thrown by the Standard C++ library, and lists unsupported functionality.
Chapter 7	**Optimizing HP aC++ Programs** Gives you information about optimizing your programs.
Chapter 8	**Exception Handling** Discusses exception handling, and information on using threads and parallel programming.

Chapter 9	**Tools and Libraries** Discusses the tools and libraries bundled with HP aC++.
Chapter 10	**Mixing C++ with Other Languages** Provides guidelines for linking HP aC++ modules with modules written in HP C and HP FORTRAN 90 on HP systems.
Chapter 11	**Distributing Your C++ Products** Provides distribution-related information for C++ products. If you choose to distribute archive libraries or object files, your customer must have purchased HP aC++. Make sure that your customer has read this distribution information.
Chapter 12	**Migrating from HP C++ (cfront) to HP aC++** Discusses differences in syntax and functionality that you may need to consider, when migrating code from HP C++ (cfront) to HP aC++.
Appendix A	**Diagnostic Messages** Discusses the aCC message catalog and frequently encountered messages. The aC++ compiler can issue a large variety of diagnostics in response to unexpected situations or suspicious constructs.
Glossary	Contains definitions of terms used in this book, listed alphabetically.

Typographical Conventions

This document uses the following conventions.

audit(5)	An HP-UX manpage. In this example, *audit* is the name and *5* is the section in the *HP-UX Reference* respectively. On the Web and on the Instant Information CD, it may be a hot link to the manpage itself. From the HP-UX command line, you can enter “`man audit`” or “`man 5 audit`” to view the manpage. See *man*(1).
Book Title	The title of a book. On the Web and on the Instant Information CD, it may be a hot link to the book itself.
KeyCap	The name of a keyboard key.
Emphasis	Emphasized text.
Bold	Strongly emphasized text.
Bold	The defined use of an important word or phrase.
`ComputerOut`	Text displayed by the computer.
`UserInput`	Commands and other text that you type.
`Command`	A command name or qualified command phrase.
`Variable`	The name of a variable that you may replace in a command or function or information in a display that represents several possible values.
[]	The contents are optional in syntax. If the contents are a list separated by |, you must choose one of the items.
{}	The contents are required in syntax. If the contents are a list separated by |, you must choose one of the items.
...	The preceding element may be repeated an arbitrary number of times.
|	Separates items in a list of choices.

HP-UX Release Name and Release Identifier

Each HP-UX 11i release has an associated release name and release identifier. The*uname*(1) command with the `-r` option returns the release identifier. This table shows the releases available for HP-UX 11i.

Table 1 HP-UX 11i Releases

Release Identifier	Release Name	Supported Processor Architecture
B.11.31	HP-UX 11i v3.0	Intel® Itanium®
B.11.11	HP-UX 11i v1	PA-RISC
B.11.23	HP-UX 11i v2.0	PA-RISC
B.11.31	HP-UX 11i v3.0	PA-RISC
B.11.20	HP-UX 11i v1.5	Intel® Itanium®
B.11.22	HP-UX 11i v1.6	Intel® Itanium®
B.11.23	HP-UX 11i v2.0	Intel® Itanium®
B.11.31	HP-UX 11i v3.0	Intel® Itanium®

Publishing History

Edition	Release Date	Product Version
13	March 2014	HP aC++ v A.06.28
12	September 2012	HP aC++ v A.06.27
11	September 2011	HP aC++ v A.06.26
10	March 2010	HP aC++ v A.06.25
9	September 2009	HP aC++ v A.06.20
8	September 2007	HP aC++ v A.06.15
7	November 2006	HP aC++ v A.06.12
6	May 2006	HP aC++ v A.06.10
5	September 2005	HP aC++ v A.06.05
4	December 2004	HP aC++ v A.06.00/A.05.60
3	September 2004	HP aC++ v A.05.55.02
2	March 2004	HP aC++ v A.05.55
1	August 2003	HP aC++ v A.05.50

Related Documents

You can fine additional information about the HP aC++/HP C compiler on the web at http://www.hp.com/go/hpux-C-Integrity-docs.

The following is a list of documents available with this release:

- *HP aC++/HP ANSI C Release Notes*

  This document gives an overview of new command-line options and features in HP aC++ and HP C compilers for Itanium®-based systems.

- *HP C/HP-UX Reference Manual*

  This manual presents reference information on the C and C++ programming languages.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are truly committed to providing documentation that meets your needs.

Please send comments to: `c++-editor@cup.hp.com`

Please include document title, manufacturing part number, and any comment, error found, or suggestion for improvement that you have about this document.

1 Getting Started with HP aC++

The information in this document applies to the release of HP aC++ and HP ANSI C compilers version A.06.28 for the HP-UX 11i v3 operating system.

The HP ANSI C compiler supports ANSI programming language C standard ISO 9899:1999. HP aC++ compiler supports the ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++).

Version A.06.28 of the HP aC++/HP C compiler provides leading edge support for C++11 standard language features, with complete binary compatibility with earlier releases and -AA compilation mode.

This chapter discusses the following topics:

- "Components of the Compilation System" (page 22)
- "Compiler Command Syntax and Environmental Variables" (page 24)
- "Files on the aCC Command Line" (page 25)
- "Environment Variables" (page 26)
- "Floating Installation" (page 29)

Components of the Compilation System

The HP aC++ compiling system consists of the following components:

`aCC`	The aCC driver is the only supported interface to HP aC++ and to the linker for HP aC++ object files.
`cc`	`cc` is the HP-UX C compiler.
`c89`	`c89` is the HP-UX ANSI-conforming C89 compiler.
`c99`	`c99` is the HP-UX ANSI-conforming C99 compiler.
`ecom`	The `ecom` compiler (for A.06.*) compiles the C++ source statements. Preprocessing is incorporated into the compiler.
`ctcom`	The `ctcom` compiler (for A.05.*) compiles the C++ source statements. Preprocessing is incorporated into the compiler.

The other HP aC++ executable files are:

`c++filt`	`c++filt` is the name demangler. It implements the name demangling algorithm which encodes function name, class name, and parameter number and name.
`ld`	`ld` is the linker. It links executables and builds shared libraries.

HP aC++ Runtime Libraries and Header Files:

```
Standard C++ Library
       /usr/lib/hpux32/libstd.so (32-bit shared version)
       /usr/lib/hpux32/libstd.a  (32-bit archive version)
       /usr/lib/hpux64/libstd.so (64-bit shared version)
       /usr/lib/hpux64/libstd.a  (64-bit archive version)

HP aC++ Runtime Support Library
       /usr/lib/hpux##/libCsup.so
       /usr/lib/hpux##/libCsup11.so – ISO C++11 standard compliant
       /usr/lib/hpux##/libstd.so and libstd_v2.so
       /usr/lib/hpux##/libstd_v2.so and librwtool_v2.so
       /usr/lib/hpux##/libstream.so
       Libraries in /usr/include/hpux##

       (where ## is 32 or 64 provided as part of the HP-UX core system)
```

```
Standard C++ Library
       /usr/lib/hpux32/libstream.so (32-bit shared version)
       /usr/lib/hpux32/libstream.a  (32-bit archive version)
       /usr/lib/hpux64/libstream.so (64-bit shared version)
       /usr/lib/hpux64/libstream.a  (64-bit archive version)

       Header files for these libraries are located at /opt/aCC/include/.
```

Using the aCC Command

To invoke the HP aC++ compiling system, use the `aCC` command at the shell prompt. The`aCC` command invokes a driver program that runs the compiling system according to the filenames and command line options that you specify.

Compiling a Simple Program

The best way to get started with HP aC++ is to write, compile, and execute a simple program, as shown in the following example:

```
#include <iostream.h>
int main()
{
      int x,y;
      cout << "Enter an integer: ";
      cin >> x;
      y = x * 2;
     cout << "\n" << y <<" is twice " << x <<".\n";
}
```

If this program is in the file `getting_started.C`, compiling and linking the program with the `aCC` command produces an executable file named `a.out`, by default:

```
$ aCC getting_started.C
```

Executing the Program

To run this executable file, just enter the name of the file. The following summarizes this process with the file `getting_started.C`:

```
$ a.out
  Enter an integer: 7
  14 is twice 7.
```

Debugging Programs

You can use programming and debugging aides.

HP Code Advisor

HP Code Advisor is a code checking tool that can be used to detect programming errors in C/C++ source code. Use "`/opt/cadvise/bin/cadvise`" to invoke the tool. A brief description is available with the `-help` option.

```
$ /opt/cadvise/bin/cadvise -help
```

Additional information is available at: `http://www.hp.com/go/cadvise/`.

HP WDB Debugger

You can also use the HP WDB debugger to debug your C++ programs after compiling your program with either the `-g`, the `-g0`, or the `-g1` option.

Example:

The `-g0` option enables generation of debug information:

```
$ aCC -g0 program.C
```

The `gdb` command runs the HP WDB debugger:

```
$ gdb a.out
```

For more information on the HP WDB debugger, refer to "Debugging Options" (page 35).

Accessing Online Example Source Files

Online example source files are located in the directory `/opt/aCC/contrib/Examples/RogueWave`. These include examples for the Standard C++ Library and for the Tools.h++ Library.

Compiler Command Syntax and Environmental Variables

The `aCC` command (the driver) invokes the HP aC++ compiling system. The aCC command is followed by options and files that need to be compiled.

```
aCC [options] [files]
```

You must use the `aCC` command to link your C++ programs and libraries. This ensures that all libraries and other files needed by the linker are available.

Example:

```
aCC prog.C
```

This command compiles the source file `prog.C` and puts the executable code in the file `a.out`.

For a complete list of command line options, see Chapter 2: "Command-Line Options" (page 31).

Examples of the aCC Command

Following are some examples of the `aCC` command:

Compiling and Renaming an Output File

```
aCC -o prog prog.C
```

This command compiles `prog.C` and puts the executable code in the file `prog`, rather than in the default file `a.out`.

Compiling and Debugging

```
aCC -g prog.C
```

This command compiles `prog.C` and includes information that allows you to debug the program with the HP WDB debugger, `wdb`.

Compiling Without Linking

```
aCC -c prog.C
```

This command compiles `prog.C` and puts the object code in the file `prog.o`. It neither links the object file nor creates an executable file.

Linking Object Files

```
aCC file1.o file2.o file3.o
```

This command links the listed object files (`file1.o`, `file2.o`, and `file3.o`) and puts the executable code in the file `a.out`.

NOTE: You must use the `aCC` command to link your HP aC++ programs and libraries. This ensures that all libraries and other files needed by the linker are available.

Compiling, Optimizing, and Getting Verbose Information

```
aCC -O -v prog.C
```

This command compiles and optimizes `prog.C`, gives verbose progress reports, and creates an executable file `a.out`.

Compiling and Creating a Shared Library

```
aCC +z -c prog.C
```

```
aCC -b -o mylib.sl prog.o
```

The first line compiles `prog.C`, creates the object file `prog.o`, and puts the position-independent code (PIC) into the object file. The second line creates the shared library `mylib.sl`, and puts the executable code into the shared library.

Files on the aCC Command Line

Files containing source or object code to be compiled or linked by HP aC++ can be any of these files:

- A C++ Source File (.C file)
- Preprocessed Source Files (.i Files)
- Assembly Language Source Files (.s Files)
- Object Files (.o Files)
- Library Files (.a and .so Files)
- “Configuration Files (.conf Files)” (page 26)

Unless you use the `-o` option to specify otherwise, all files that the aCC compiling system generates are put in the working directory, even if the source files are from other directories.

C++ Source File (.C file)

You must name the HP aC++ source files with extensions beginning with either `.c` or `.C`, possibly followed by additional characters. If you compile only, for example by using `-c`, each C++ source file produces an object file with the same file name prefix as the source file and a `.o` file name suffix.

However, if you compile and link a single source file into an executable program in one step, the `.o` file is automatically deleted, unless `-g` is used without `+noobjdebug`.

NOTE: HP recommends that your source files have `.c` or `.C` extensions only, without any additional characters. While extensions other these are permitted for portability from other systems, they may not be supported by HP tools and environments.

Preprocessed Source Files (.i Files)

Files with `.i` extensions are assumed to be preprocessor output files. These files are processed in the same way as `.c` or `.C` files, except that the preprocessor is not run on the `.i` file before the file is compiled.

Use the `-P` or the `-E` compiler option to preprocess a C++ source file without compiling it.

Assembly Language Source Files (.s Files)

Files with names ending in `.s` are assumed to be assembly source files. The compiler invokes the assembler through `cc` to create `.o` files from these.

Use the `-S` option to compile a C++ source file to assembly code and put the assembly code into a `.s` file.

Object Files (.o Files)

Files with `.o` extensions are assumed to be relocatable object files that have to be included in the linking. The compiler invokes the linker to link the object files and create an executable file.

Use the `-c` option to compile a C++ source file into a `.o` file.

Library Files (.a and .so Files)

Files ending with `.a` are assumed to be archive libraries. Files ending with `.so` are assumed to be shared libraries.

Use the `-c` and `+z` options to create object files of Position-Independent Code (PIC) and the `-b` option to create a shared library.

Use the `-c` option to create object files and the `ar` command to combine the object files into an archive library.

Configuration Files (.conf Files)

You can configure compiler options on a system-wide basis. The compiler reads the configuration files:

`/var/aCC/share/aCC.conf` (aC++), or

`/var/ansic/share/cc.conf`(ANSI C), if present.

In C-mode, the configuration file defaults to`/var/ansic/share/cc.conf`, unless overridden by the environment variable `CC_CONFIG`..

In C++ mode, the config file defaults to `/var/aCC/share/aCC.conf`, unless overriden by the environment variable `CXX_CONFIG`.

The options in the configuration file can be specified in the same manner as that for `CCOPTS` and `CXXOPTS`, namely:

```
[options-list-1] [|[options-list-2]]
```

where options in `options-list-1` are applied before the options in the command line, and options in `options-list-2` are applied after the options in the command line.

The final option ordering would be:

```
<file-options-1><envvar-options-1><command-line-options>
<envvar-options-2><file-options-2>
```

NOTE: No configuration files are shipped along with aC++, but can be installed by the system administrator, if required.

The config file options before the "`|`" character set the defaults for compilations, and the options after the character override the user's command line settings.

Environment Variables

This section describes the following environment variables that you can use to control the HP aC++ or HP C compiler:

- "aCC_FULL_PATHNAMES Environment Variable" (page 27)
- "aCC_MAXERR Environment Variable" (page 27)
- "CXXOPTS Environment Variable" (page 27)
- "CCLIBDIR Environment Variable" (page 27)
- "CCROOTDIR Environment Variable" (page 28)

- “CXX_MAP_FILE Environment Variable” (page 29)
- “TMPDIR Environment Variable” (page 29)

aCC_FULL_PATHNAMES Environment Variable

Exporting the `aCC_FULL_PATHNAMES` variable causes the compiler to include full path names for files in compiler messages. This feature is useful in debugging.

aCC_MAXERR Environment Variable

The `aCC_MAXERR` environment variable allows you to set the maximum number of errors you want the compiler to report before it terminates compilation. The default is 100.

CXXOPTS Environment Variable

The `CXXOPTS` environment variable provides a convenient way to include frequently used command-line options automatically.

Options before the vertical bar (|) are placed before command-line options to `aCC`. The options after the vertical bar are placed after any command-line option. Note that the vertical bar must be delimited by white space.

If you do not use the vertical bar, all options are placed before the command line parameters. Set the environment variable and the options you want are automatically included each time you execute the `aCC` command.

Syntax:

```
export CXXOPTS="options | options" ksh/sh notation
setenv CXXOPTS "options | options" csh notation
```

Usage:

For quick or temporary changes to your build environment, use `CXXOPTS` instead of editing your makefiles.

Example:

```
export CXXOPTS="-v | -lm" ksh/sh notation
setenv CXXOPTS "-v | -lm" csh notation
```

The above command causes the `-v` and `-l` options to be passed to the `aCC` command each time you execute it.

When `CXXOPTS` is set as above, the following two commands are equivalent:

```
aCC -g prog.C
aCC -v -g prog.C -lm
```

CCLIBDIR Environment Variable

The `CCLIBDIR` environment variable causes the `aCC` command to search for libraries in an alternate directory before searching in the default directories.

Syntax:

```
export CCLIBDIR=directory   ksh/sh notation
setenv CCLIBDIR directory   csh notation
```

`directory` is an HP-UX directory where you want HP aC++ to look for libraries.

Example:

```
export CCLIBDIR=/mnt/proj/lib
```

In this example HP aC++ searches the directory `/mnt/proj/lib` for libraries before searching the directory `/opt/aCC/lib`.

When `CCLIBDIR` is set a in the above example, the following two commands are equivalent:

```
aCC -L/mnt/proj/lib file.o
aCC file.o
```

NOTE: You can use the `-Ldirectory` option to specify additional directories for the linker to search for libraries.

CCROOTDIR Environment Variable

The `CCROOTDIR` environment variable causes `aCC` to invoke all subprocesses from an alternate `aCC` directory, rather than from their default directory. The default aCC root directory is `/opt/aCC`.

Syntax:

```
export CCROOTDIR=directory   ksh/sh notation
setenv CCROOTDIR directory   csh notation
```

`directory` is an aCC root directory where you want the HP aC++ driver to search for subprocesses.

Example:

```
export CCROOTDIR=/mnt/CXX2.1
```

In this example, HP aC++ searches the directories under `/mnt/CXX2.1` (`/mnt/CXX2.1/bin` and `/mnt/CXX2.1/lbin`) for subprocesses rather than their respective default directories.

CXX_MAP_FILE Environment Variable

To facilitate easy migration of build environment from a different compiler to HP aC++, an option mapping support is provided. You can use the option mapping files to map the options in the third party compilers to HP aC++ equivalents. The mapping file is a text file that defines the mapping rules. The compiler reads the mapping file and applies the specified replacements to the options on the command line. This minimizes the need to make Makefile or script changes. The CXX_MAP_FILE environment variable allows you to change the location of the mapping file.

Syntax:

```
export CXX_MAP_FILE=file path
```

Example:

```
export CXX_MAP_FILE=/home/src/my_option.map
```

The example specifies that HP aC++ should use mapping file from file path specified using CXX_MAP_FILE.

Defining the Mapping Rules:

Following is the syntax for defining the rules in the mapping file:

LHS => RHS

where:

- Left Hand Side (LHS) is the third party compiler option.
- Right Hand Side (RHS) is the HP aC++ compiler option

NOTE: Ensure to use a space before and after "=>".

To define rules for options that have arguments, use the $<number> wildcard.

For Example:

$1 for the first argument, and $2 for the second. If the third party compiler option (LHS) does not match with any HP aC++option, leave the RHS blank.

TMPDIR Environment Variable

The `TMPDIR` environment variable allows you to change the location of temporary files created by the compiler. The default directory is `/var/tmp`.

Syntax:

```
export TMPDIR=directory   ksh/sh notation
setenv TMPDIR directory   csh notation
```

`directory` is the name of an HP-UX directory where you want HP aC++ to put temporary files during compilation.

Example:

```
export TMPDIR=/mnt/temp   ksh notation
setenv TMPDIR /mnt/temp   csh notation
```

The above example specifies that HP aC++ should put all temporary files in `/mnt/temp`.

Floating Installation

More than one version of the HP aC++ compiler can be installed on one system at the same time. The floating installation feature allows you to install the compiler in any location. You can install as many compiler versions as required, depending on your system’s resources.

HP aC++

By default, HP aC++ is installed under the `/opt/aCC` directory. In earlier releases, the compiler driver (`aCC`) looked for related files in subdirectories of the `/opt/aCC` directory. This prevented installation of more than one version of HP aC++ on the same system at the same time.

Only files in `/opt/aCC` are affected by floating installation. Regardless of the HP aC++ driver you use, the compiler still uses the libraries, linker, and other files located in `/usr/lib` and `/usr/ccs`.

Floating installation is designed to help facilitate in-house development. You must not ship libraries in non-standard places, because explicit runtime library specifications and linker options are required.

You can use the `__HP_aCC` predefined macro to determine which version is being run.

HP C

You can use the `__HP_cc` predefined macro to determine which version is being run.

NOTE: Do not use floating installation with the following:

- `CCROOTDIR` environment variable
- `-tc,name` command line option

Setting up Floating Installation

You may want to install the most recent compiler version and keep the prior version on one system. If there are problems with the most recent version, you can easily switch to the prior one. Following is an example of how to set up the floating installation feature for this purpose. Assume that your system will have two versions of the compiler, both floating install enabled. In this case, A.05.50 is the prior version, and A.05.60 or A.06.00 is the more recent version.

To setup floating installation, complete the following steps:

1. Copy the prior version to another directory.

   ```
   cp -rp /opt/aCC /opt/aCC.05.55
   ```

2. Use `swinstall` to install the new version (A.06.00 or A.05.60 in this case).

2 Command-Line Options

You can specify command-line options to the `aCC` command. They allow you to override the default actions of the compiler. Each option begins with either a - or a + sign. Any number of options can be interspersed anywhere in the `aCC` command and they are typically separated by blanks. Unless specified otherwise, all options are supported by C and C++ compilers.

Default options can be set using option configuration files. See "Configuration Files (.conf Files)" (page 26).

This chapter discusses the following command-line options:

Options to Control Code Generation

The following options allow you to control the kind of code that the compiler generates:

- -c
- +DO*osname*
- +DD*data_model*
- +DS*model*
- -S

-c

You can use the `-c` option to compile one or more source files without entering the linking phase.

When compiled, the compiler produces an object file (a file ending with `.o`) for each source file (a file ending with `.c`, `.C`, `.s`, or `.i`). Note that you must link object files before they can be executed.

Example:

```
aCC -c sub.C prog.C
```

In this example, the compiler compiles `sub.C` and `prog.C` and puts the relocatable object code in the files, `sub.o` and `prog.o`, respectively.

+DO*osname*

The `+DO`*`osname`* option sets the target operating system for the compiler, and is intended for enabling optimizations that are not backward compatible.

For instance, the 11.23 compiler introduces new optimized math library functions that require library support not available in prior operating systems. `+DO` can be used at any level of optimization. The default value for *`osname`* is the operating system version on which the compiler is invoked.

The syntax for this option is `+DO`*`osname`*, where *`osname`* is either `11.20`, `11.22` or `11.23`.

Example:

The following example generates code for the HP-UX 11.22 (or later) operating system. Binary incompatible features introduced in later OS versions are inhibited.

```
aCC +DO11.22 +O3 app.C
```

+DD*data_model*

The `+DD`*`data_model`* option specifies the data model for the compiler.

`data_model` can be one of the following:

- 32 (This value generates ILP32 code and is the default.)
- 64 (This value generates LP64 code.)

This option specifies the data model for the compiler. Table 2 lists the differences between the two data models.

Table 2 ILP32 Data Model and LP64 Data Model

ILP32 Data Model	LP64 Data Model
The size of an `int`, `long`, or `pointer` data type is 32-bits.	The size of an int data type is 32-bits. The size of a `long` or `pointer` data type is 64-bits.
The preprocessor predefined macro `_ILP32` is defined.	The preprocessor predefined macros `__LP64__` and `_LP64` are defined.

Examples:

The following command generates code for the 64-bit data model:

```
aCC +DD64 app.C
```

The following command generates code for the 32-bit data model:

```
aCC app.C
```

+DS*model*

The `+DS`*`model`* option performs instruction scheduling for a particular implementation of the Itanium®-based architecture.

`model` can be one of the following values.

`blended`	Tune to run reasonably well on multiple implementations. As old implementation become less important and new implementations are added, the behavior with this value will change accordingly.
`itanium`	Tune for the Itanium® processor.
`itanium2`	Tune for the Itanium2® processor.
`mckinley`	See `itanium2®`.
`montecito`	Tune for the Montecito® processor.
`poulson`	Tune for the Poulson® processor.
`native`	Tune for the processor on which the compiler is running.

The default is `blended`. Object code with scheduling tuned for a particular model will execute on other HP-UX systems, although possibly less efficiently.

Using +DS to Specify Instruction Scheduling

Instruction scheduling is different on different implementations of Itanium®-based architectures. You can improve performance on a particular model or processor of the HP-UX system by requesting the compiler to use instruction scheduling tuned to that particular model or processor. Using scheduling for one model or processor does not prevent your program from executing on another model or processor.

If you plan to run your program on the same system where you are compiling, you do not need to use the `+DS` option. The compiler generates code tuned for your system.

If you plan to run your program on a particular model of the HP-UX system, and that model is different from the one where you compile your program, use `+DS`*`model`* with either the model number of the target system or the processor name of the target system.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, it is important that the default code generation and scheduling are based on the local host processor. The system model numbers of the hosts where the source or object files reside do not affect the default code generation and scheduling.

-S

The `-S` option compiles a program and logs the assembly language output in a corresponding file with a `.s` suffix. The `-S` option is only for displaying the assembler code. The generated code is not intended to be used as input to the assembler (`as`).

Example:

The following command compiles `prog.C` to assembly code rather than to object code, and puts the assembly code in the file `prog.s`.

```
aCC -S prog.C
```

Data Alignment and Storage

This section describes default data storage allocation and alignment for HP compiler data types.

Data storage refers to the size of data types, such as `bool`, `short`, `int`, `float`, and `char*`. Data alignment refers to the way the HP compiler aligns data structures in memory. Data type alignment and storage differences can cause problems when moving data between systems that have different alignment and storage schemes. These differences become apparent when a structure is exchanged between systems using files or inter-process communication. In addition, misaligned data addresses can cause bus errors when an attempt is made to dereference the address.

For information on unaligned data access, See "Handling Unaligned Data" (page 101).

Table 3 lists the size and alignment of the HP compiler data types:

Table 3 Size and Alignment of HP Compiler Data Types

Data Type	Size (in bytes)	Alignment
`bool`	1	1-byte
`char`, `unsigned char`, `signed char`	1	1
`wchar_t`	4	4
`short`, `unsigned short`, `signed short`	2	2
`int`, `unsigned int`	4	4
`long`, `unsigned long`	4*	4*
`float`	4	4
`__float80`	16	16
`__float128`	16	8**
`_Decimal32`	4	4
`_Decimal64`	8	8
`_Decimal128`	16	16
`double`	8	8
`long double`	16	8**
`long long`, `unsigned long long`	8	8
`enum`	4	4
arrays	Size of array element type	Alignment of array element type
`struct`	***	1-, 2-, 4-, 8-, or 16-byte
`union`	***	1-, 2-, 4-, 8-, or 16-byte
bit-fields	Size of declared type	Alignment of declared type
pointer	4*	4*

* In 64-bit mode, `long`, `unsigned long`, and pointer data types are 8 bytes long and 8-byte aligned.

** In 64-bit mode, `long double` is 16-byte aligned.

Table 3 Size and Alignment of HP Compiler Data Types *(continued)*

Data Type	Size (in bytes)	Alignment
*** `struct` and `union` alignment are same and follow strict alignment of any member. Padding is done to a multiple of the alignment size.		

-fshort-enums

```
cc -Agcc -Wc,--fshort-enums foo.c
```

```
aCC -Ag++ -Wc,--fshort-enums foo.c
```

The `-fshort-enums` option is used with the `-Agcc` or `-Ag++` options to cause each enum type to be represented using the smallest integer type that is capable of representing all values of the enum type. Because it changes the representation of types, the code generated is *not* binary compatible with code compiled without the option. The primary use of this option is for compatibility with `gcc`, but it can provide performance improvement to applications that can accept the binary incompatibility.

+u*num*

`+u`*num*

The `+u`*num* option allows pointers to access non-natively aligned data. This option alters the way that the compiler accesses dereferenced data. Use of this option may reduce the efficiency of generated code. Specify *num* as 1, 2, or 4, as follows:

1 - Assume single byte alignment. Dereferences are performed with a series of single-byte loads and stores.

2 - Dereferences are performed with a series of two-byte loads and stores.

4 - Dereferences are performed with a series of four-byte loads and stores.

Example:

```
aCC +u1 app.C
```

Debugging Options

Debugging options enable you to use the HP WDB debugger.

Information on HP WDB is available at this location: http://www.hp.com/go/wdb

+d

The `+d` option prevents the expansion of inline functions. It is useful when you debug code because breakpoints cannot be set at inline functions. Using the `+d` option disables all inlining. It is mapped to the `+inline_level 0` option.

+expand_types_in_diag

The `+expand_types_in_diag` option expands typedefs in diagnostics so that both the original and final types are present.

-g

The `-g` option causes the compiler to generate minimal information for the debugger. It uses an algorithm that attempts to reduce duplication of debug information.

To suppress expansion of inline functions, use the `+d` option.

-g0

The `-g0` option causes the compiler to generate full debug information for the debugger.

To suppress expansion of inline functions, use the `+d` option.

-g1

Like the `-g` option, the `-g1` option causes the compiler to generate minimal information for the debugger. It uses an algorithm that attempts to reduce duplication of debug information. To suppress expansion of inline functions, use the `+d` option.

Differences Between -g, -g0, and -g1 Options

The `-g`, `-g0`, and `-g1` options generate debug information. The difference is that the `-g0` option emits full debug information about every class referenced in a file, which can result in some redundant information.

The `-g` and `-g1` options emit a subset of this debug information, thereby decreasing the size of your object file. If you compile your entire application with `-g` or `-g1,` no debugger functionality is lost.

NOTE: If you compile part of an application with `-g` or `-g1` and part with debug off, (that is, with neither the `-g`, the `-g0`, nor the `-g1` option) the resulting executable may not contain complete debug information. You will still be able to run the executable, but in the debugger, some classes may appear to have no members.

When to use -g, -g0, and -g1

Use `-g` or `-g1` when you are compiling your entire application with debug on and your application is large, for example, greater than 1 MB.

Use `-g0` when either of the following is true:

- You are compiling only a portion of your application with debug on, for example, a subset of the files in your application.
- You are compiling your entire application with debug on and your application is not very large, for example, less than 1 MB.

-g, -g1 Algorithm

In general, the compiler looks for the first non-inline, non-pure (non-zero) virtual function in order to emit debug information for a class.If there are no virtual member functions, the compiler looks for the first non-inline member function.

If there are no non-inline member functions, debug information is always generated.

A problem occurs if all functions are inline; in this case, no debug information is generated.

+macro_debug

This option controls the emission of macro debug information into the object file.

Set `+macro_debug` to one of the following required options:

`ref`	Emits debug information only for referenced macros. This is the default for `-g, -g1,` or `-g0.`
`all`	Emits debug information for all macros. This option can cause a significant increase in object file size.
`none`	Does not emit any macro debug information.

One of the `-g` options (`-g`, `-g0`, or `-g1`) must be used to enable the `+macro_debug` option.

+[no]objdebug

The `+objdebug` option generates debug information in object files and not in the executable. The HP WDB debugger then reads the object files to construct debugging information; they must be present when debugging.

The `+noobjdebug` option generates debug information in object files which the linker places into the executable. The HP WDB debugger then reads the executable to construct debugging information.

NOTE: With `+objdebug`, the object files or archive libraries must not be removed.

`+objdebug` is the default at link time and at compile time. If `+noobjdebug` is used at link time, all debug information goes into the executable, even if some objects were compiled with `+objdebug`.

If `+objdebug` is used at compile time, extra debug information is placed into each object file to help the debugger locate the object file and to quickly find global types and constants.

Usage:

Use `+objdebug` option to enable faster links and smaller executable file sizes for large applications, rather than `+noobjdebug` where debug information is written to the executable.

Use `+noobjdebug` with the `-g`, `-g0`, or `-g1` option when using `+ild`.

+pathtrace

`+pathtrace[=`*`kind`*`]`

The `+pathtrace` option provides a mechanism to record program execution control flow into global and/or local path tables. The saved information can be used by the HP WDB debugger to assist with crash path recovery from the core file, or to assist when debugging the program by showing the executed branches.

Currently only `if`, `else`, `switch-case-default`, and `try-catch` execution paths are recorded in the path table. If there is no condition statement inside a `for`, `while`, or `do-while` loop, then no excution path is recorded.

Usage:

The defined values for *`kind`* are:

`local`	Generates a local path table and records basic block-execution information in it at runtime.
`global`	Generates a global path table and records basic block-execution information in it at runtime. The global path table is a fixed size. The default size of the table is 8K items. Each basic block that is executed is recorded as an item of path-trace information in the table. Each thread has its own table, and when the table is full, the runtime system wraps the path table back to the beginning of the table. The table size can be configured at runtime using the environment variable HP_PATHTRACE_CONFIG, which also lets you specify a file to be used for dumping full tables and the dumping format, before wrapping around or at thread/program termination. The syntax of the environment variable HP_PATHTRACE_CONFIG is:

```
HP_PATHTRACE_CONFIG=item[:item]
                item := TABLE_SIZE=nnn |
          FILE=[stdout|stderr|<filename>] |
                         FORMAT=[binary|text]
```

where:

- TABLE_SIZE specifies the size, expressed as the number of items (*nnn*), of the global path table.
- FILE specifies the dumping output *filename* to use when the global path table is full.
- FORMAT specifies the dumping format in either "binary or human-readable "text".

`global_fixed_size`	Generates a fixed-size (65536 items) global path table and records basic block-execution information in it at runtime. This form differs from`+pathtrace=global` because the size of the table cannot be configured at runtime, and the contents cannot be dumped to a file. The fixed-size global path table has better runtime performance than the configurable global path table. The performance difference varies depending on the optimization level and how the program is written.
`none`	Disables generation of both the global and local path tables.

The values can be combined by joining them with a colon. For example:

```
+pathtrace=global:local
```

The `global_fixed_size` and `global` values are mutually exclusive. If more than one of them are specified on the command line, the last one takes precedence. The same is true for the `none` value.

`+pathtrace` with no values is equivalent to `+pathtrace=global_fixed_size:local.`

The use of this option and the `-mt` option must be consistent for all compilation and link steps. That means if `-mt` is used with `+pathtrace` at compile time, it should also be used at link time; if `-mt` is not used with `+pathtrace` at compile time, it should not be used at link time. Otherwise, a link-time error can occur.

Error Handling

Use the following options to control how potential errors in your code are detected and handled. You can also use the `cadvise report` feature of the HP Code Advisor tool to help analyze compiler errors and warnings.

+p

The `+p` option disallows all anachronistic constructs.

Ordinarily, the compiler gives warnings about anachronistic constructs. Using the `+p` option, the compiler gives errors for anachronistic constructs.

Example:

The following command compiles `file.C` and gives errors for all anachronistic constructs rather than just giving warnings.

```
aCC +p file.C
```

-w

The `-w` option disables all warnings except those that are explicitly enabled with a `+Wwargs` option or a subsequent `+w`-prefix option. By default, the compiler reports all errors and warnings.

HP recommends against using the `-w` option. In addition to disabling messages currently output by the compiler, it will also disable any new messages added to the compiler in the future that could identify problem areas in user code. HP recommends using the `+Wargs` option to disable a message. Although it can often take a long list of `+Warg` options to disable all desired warnings, this list can be included in an options file and referenced using the `+opts` option to avoid listing them all directly on the command line.

Example:

The following command compiles `file.C` and reports errors but does not report any warnings.

```
aCC -w file.C
```

+w

The `+w` option warns you about all questionable constructs and gives pedantic warnings. `+w` enables all checks except for the `+wsecurity`, `+wendian`, `+wperfadvice`, and`+wlock` warnings. Those need to be enabled explicitly if needed. The default is to warn only about constructs that are almost certainly problems.

For example, this option warns you when calls to inline functions cannot be expanded inline.

The following command compiles `file.C` and warns about both questionable and problematic constructs:

```
aCC +w file.C
```

NOTE: This option is equivalent to the `+w1` option of legacy HP C.

+wn

The `+wn` option specifies the level of the warnings messages.

`+w{1|2|3}`

The value of `n` can be one of the following:

1 All warnings are issued. This includes low level warnings that may not indicate anything wrong with the program.

2 Only warnings indicating that code generation might be affected are issued. This is equivalent to the compiler default without the `-w` options.

3 No warnings are issued. This is equivalent to the `-w` option. This option is the same as `-W c` and`-wn`.

+Wargs

`+Warg1[,arg2,..argn]`

The `+Wargs` option selectively suppresses any specified warning messages.

Arguments `arg1` through `argn` are valid compiler warning message numbers.

Example:

```
aCC +W600 app.C
```

+Wcontext_limit

`+Wcontext_limit=`num

The `+Wcontext_limi` option limits the number of instantiation contexts output by the compiler for diagnostics involving template instantiations. At most `num` outermost contexts and `num` innermost contexts are shown. If there are more than `2*num` relevant contexts, the additional contexts are omitted.

Omitted contexts are replaced by a single line separating the outermost `num` contexts from the innermost `num` contexts, and indicating the number of contexts omitted. The default value for `num` is 5. A value of 0 removes the limit.

+We

`+We`

The `+We` option interprets all warning and future error messages as errors.

Alternatively you can also use `+We[arg1,...argn]` option, where `arg` is a valid compiler warning message number. Use of `arg` is optional.

+Weargs

`+Wearg1[,arg2,..,argn]`

The `+Weargs` option selectively interprets any specified warning or future error messages as errors. `arg1` through `argn` are valid compiler warning message numbers.

Example:

```
aCC +We 600,829 app.C
```

+Wv

`+Wv[d1,d2,..,dn]`

The `+Wv` option displays the description for diagnostic message numbers `d1` through `dn`.

Specifying this option causes the compiler to emit the descriptive text for the specified dianostics to `stderr`. This option must not be used with any other compiler options.

If the description for a diagnostic is not available, the compiler emits only the diagnostic with a note that the description is not available.

+Wwargs

`+Wwarg1[,arg2,..,argn]`

The `+Wwargs` option selectively treats compiler remarks or discretionary errors as warnings. `arg1` through `argn` are valid compiler message numbers. Conflicts between `+W`, `+Ww`, and `+We` are resolved based on their severity. `+We` is the highest and `+W` is the lowest.

+wlint

This option enables several warnings in the compiler that provide lint like functionality. Checks are made for memory leaks, out-of-scope memory access, null pointer dereference, and out-of-bounds access. These compile time diagnostics can be very useful in detecting potential problems in the source code. To disable a specific warning introduced by `+wlint`, a `+Wargs` option can be used after the `+wlint` option.

NOTE: The `+wlint` option is only supported on Itanium-based systems.

+Wmacro

`+Wmacro:`*`MACRONAME`*`:d1,d2,...,dn`

The `+Wmacro` option disables warning diagnostics `d1,d2,...,dn` in the expansion of macro *`MACRONAME`*. If -1 is given as the warning number, then all warnings are suppressed. This option is not applicable to warning numbers greater than 20000. +Wmacro gets higher priority than the other diagnostic-control command-line options that are applicable to the whole source. Diagnostic control pragmas take priority based on where they are placed.

+wperfadvice

`+wperfadvice[={1|2|3|4}]`

The `+wperfadvice` option enables performance advisory messages.

The optional level 1, 2, 3,or 4 controls how verbosely the performance advisory messages are emitted. The higher the level, the more messages generated. Level 1 emits only the most important messages, while level 4 emits all the messages.

If the optional level is not specified, it defaults to 2.

+wsecurity

The `+wsecurity` option enables compile-time diagnostics for potential security violations. Warnings are emitted for cases where untrusted (tainted) data may reach a critical reference point in the program. This is based on cross-module analysis performed by the compiler. Hence the `+wsecurity` option implicitly enables a limited form of cross-module analysis, even if `-ipo` or `+O4` options are not specified. This may lead to a significant increase in the compile time compared to a build without the `+wsecurity` option. Using this option may result in the compiler invoking optimizations other than those that are part of the user-specified optimization level. If `+wsecurity` is used in addition to `-ipo` or `+O4`, the generated code is not affected and the compile time does not significantly increase.

This option can optionally take an argument to control how verbosely the security messages are emitted:

```
+wsecurity[={1|2|3|4}]
```

The higher the check level, the more warnings can be generated. Note that this may also generate more false positives.

The default level is 2.

Exception Handling

By default, exception handling is enabled. To turn off exception handling, use the following option.

+noeh

```
+noeh
```

The `+noeh` option disables exception handling. With exception handling disabled, the keywords `throw` and `try` generate an error.

Mixing code compiled with and without `+noeh` may give undesired results.

Example:

```
aCC +noeh progex.C
```

This command compiles and links `progex.C`, which does not use exception handling.

See Chapter 8: “Exception Handling” (page 161) for more information.

Extensions to the Language

These options support extensions to the C++ language.

-ext

```
-ext
```

Specifying `-ext`, enables the following HP aC++ extensions to the C++ standard:

- 64-bit integer data type support for:
  - long long (signed 64-bit integer)
  - unsigned long long (unsigned 64-bit integer)
- Use this option to declare 64-bit integer literals and for input and output of 64-bit integers.
- `#assert`, `#unassert` preprocessor directives, which allow you to set a predicate name or predicate name and token to be tested with a `#if` directive.

When this option is used with `-AC89` or `-AC99`, it defines the following macros:

- `-D__STDC_EXT__`
- `-D_HPUX_SOURCE` (unless `-Aa` is used)

NOTE: When using `-ext`, specify it at both compile and link time. For example:

```
aCC -ext foo.C
```

compiles `foo.C` which contains a `long long` declaration.

```
#include <iostream.h>

int main(){
     long long ll = 1;
     cout << ll << endl;
}
```

+e

The `+e` option is equivalent to the `-ext` option.

Floating-Point Processing Options

The following command-line options are used for floating-point processing.

+O[no]cxlimitedrange

`+O[no]cxlimitedrange`

The `+O[no]cxlimitedrange` option enables [disables] the specific block of codes with the usual mathematical formulas. This option is equivalent to adding the pragma:

```
#pragma STDC CX_LIMITED_RANGE
```

The default is `+Onocxlimitedrange`.

+O[no]fenvaccess

`+O[no]fenvaccess`

The `+O[no]fenvaccess` option provides a means to inform the compiler when a program might access the floating-point environment to test flags or run under non-default modes.

Use of the `+Onofenvaccess` option allows certain optimizations that could subvert flag tests and mode changes such as global common subexpression elimination, code motion, and constant folding. This option is equivalent to adding `#pragma STDC FENV_ACCESS ON` at the beginning of each source file submitted for compilation.

The default is `+Onofenvaccess`.

-fpeval

`-fpeval=precision`

The `-fpeval` option specifies the minimum precision to use for floating-point expression evaluation. This option does not affect the precision of parameters, return types, or assignments.

The defined values for `precision` are:

`float`	Evaluates floating-point expressions and constants in their semantic type.
`double`	Evaluates `float` operations and constants using the range and precision of `double`, and evaluates all other floating-point expressions and constants in their semantic type.
`extended`	Utilizes hardware support of these floating-point registers for optimum speed in floating-point computations. Evaluates `float` and `double` constants and expressions using the range and precision of the extended type, and evaluates all other floating-point expressions in their semantic type. Though this option provides greater precision than `double`, it does not provide greater speed than `double` or `float`.

The default is `-fpeval=float`.

-fpevaldec

`-fpevaldec=precision`

The `-fpevaldec` option specifies the minimum precision to use for decimal floating-point expression evaluation. The possible values for `precision` are `_Decimal32`, `_Decimal64`, and `_Decimal128`. This option does not affect the precision of parameters, return types, or assignments.

The default is `-fpevaldec=_Decimal32`.

-[no]fpwidetypes

`-[no]fpwidetypes`

The `-[no]fpwidetypes` option enables [disables] `extended` and `quad` floating-point data types. `quad` is equivalent to `long double`. This option also enables `__float80` prototypes. The compiler defines `_FPWIDETYPES` when `-fpwidetypes` is in effect.

The default is `-nofpwidetypes`.

+decfp

The `+decfp` option enables full decimal floating-point functionality according to the ISO/IEC C draft Technical Report (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1312.pdf)

Decimal floating-point is also supported in C++ compilation mode.

For more information on using Decimal FP, see the *HP aC++/HP ANSI C Release Notes* section "Decimal floating-point arithmetic supported" under "New Features in the A.06.20 Release."

+FP

`+FP[flags]`

The `+FP` option specifies how the runtime environment for floating-point operations should be initialized at program startup and used at link time. The default is that all trapping behaviors are disabled.

The following flags are supported. Uppercase enables the flag, lowercase disables the flag.

Table 4 Options for +FP[flags]

Flag	Description
`V (v)`	Trap on invalid floating-point operations.
`Z (z)`	Trap on divide by zero.
`O (o)`	Trap on floating-point overflow.
`U (u)`	Trap on floating-point underflow.
`I (i)`	Trap on floating-point operations that produce inexact results.
`D (d)`	Enable sudden underflow (flush to zero) of denormalized values.

To dynamically change these settings at runtime, see `fesetenv(3M)`.

+FPmode

`+FPmode` specifies how the run-time environment for floating-point operations should be initialized at program start up. By default, modes are as specified in the IEEE floating-point standard: all traps disabled, gradual underflow, and rounding to nearest. See `ld(1)` for specific values of mode. To dynamically change these settings at run time, refer to `fenv(5)`, `fesettrapenable(3M)`, and `fesetround(3M)`.

+O[no]libmerrno

`+O[no]libmerrno`

Description:

The `+O[no]libmerrno` option enables [disables] support for `errno` in `libm` functions. The default is `+Onolibmerrno` for C++, c99, or –AC99.

In C-mode, the default is `+Olibmerrno` with `-Aa` option.

+Oprefetch_latency

`+Oprefetch_latency=cycles`

The `+Oprefetch_latency` option applies to loops for which the compiler generates data prefetch instructions. `cycles` represents the number of cycles for a data cache miss. For a given loop, the compiler divides `cycles` by the estimated loop length to arrive at the number of loop iterations for which to generate advanced prefetches.

`cycles` must be in the range of 0 to 10000. A value of 0 instructs the compiler to use the default value, which is 480 cycles for loops containing floating-point accesses and 150 cycles for loops that do not contain any floating-point accesses.

For tuning purposes, it is recommended that users measure their application's performance using a few different prefetch latency settings to determine the optimal value. Some floating-point codes may benefit by increasing the distance to 960. Parallel applications frequently benefit from a shorter prefetch distance of 150.

+O[no]preserved_fpregs

`+O[no]preserved_fpregs`

The `+O[no]preserved_fprefs` option specifies whether the compiler is allowed [not allowed] to make use of the preserved subset of the floating-point register file as defined by the Itanium runtime architecture.

The default is `+Opreserved_fpregs`.

+O[no]rotating_fpregs

`+O[no]rotating_fpregs`

The `+O[no]rotating_fpregs` option specifies whether the compiler is allowed [not allowed] to make use of the rotating subset of the floating-point register file.

The default is `+Orotating_fpregs`.

+O[no]sumreduction

`+O[no]sumreduction`

This option enables [disables] sum reduction optimization. It allows the compiler to compute partial sums to allow faster computations. It is not technically legal to do this in C or C++ because of floating-point accuracy issues. This option is useful if an application cannot use the `+Onofltacc` option but wants sum reduction to be performed.

When sum reduction optimization is enabled, the compiler may evaluate intermediate partial sums of float or double precision terms using (wider) extended precision, which reduces variation in the result caused by different optimization strategies and generally produces a more accurate result.

Header File Options

Following are the command-line options you can use for header files:

-H

`cc -H file`

The `-H` option causes HP aC++/HP C to print the order and hierarchy of included files. The `-H` option dumps the include heirarchy to stderr so that the preprocessed compiler output indicates the include file nesting.

+hdr_create

`aCC progname -c +hdr_create headername`

This option extracts the header from a program file and saves it as a precompiled header file.

Example:

`aCC ApplicTemplate.C -c +hdr_create` ApplicHeader

+hdr_use

`aCC progname +hdr_use headerfile -c`

This option adds a precompiled header file to a program when the program is compiled.

Example:

`aCC Applic.C +hdr_use ApplicHeader -c`

-I directory

`-I` *`directory`*

`directory` is the HP-UX directory where the compiler looks for header files.

During the compile phase, this option adds `directory` to the directories to be searched for `#include` files during preprocessing. During the link phase, this option adds `directory` to the directories to be searched for `#include` files by the link-time template processor.

For `#include` files that are enclosed in double quotes (`" "`) within a source file and do not begin with a `/`, the preprocessor searches in the following order:

1. The directory of the source file containing the `#include`.
2. The directory named in the `-I` option.
3. The standard include directories `/opt/aCC/include` and `/usr/include`.

For `#include` files that are enclosed in angle brackets (`< >`), the preprocessor searches in the following order:

1. The directory named in the `-I` option.
2. The standard include directories `/opt/aCC/include` and `/usr/include`.

NOTE: The current directory is not searched when angle brackets (`< >`) are used with `#include`.

Example:

The following example directs the compiler to search in the directory `include` for `#include` files.

```
aCC -I include file.C
```

-I-

```
[-Idirs] -I- [-Idirs]
```

`[-Idirs]` indicates an optional list of `-Idirectory` specifications in which a directory name cannot begin with a hyphen (-) character.

The `-I-` option allows you to override the default `-Idirectory` search-path. This feature is called view-pathing. Specifying `-I-` serves two purposes:

1. It changes the compiler's search-path for quote enclosed (`" "`) file names in a `#include` directive to the following order:
   a. The directory named in the `-I`option.
   b. The standard include directories `/opt/aCC/include*` and `/usr/include`.

      The preprocessor does not search the directory of the including file.
2. It separates the search-path list for quoted and angle-bracketed `include` files.

   Angle-bracket enclosed file names in a `#include` directive are searched for only in the `-I`directories specified after `-I-` on the command-line. Quoted include files are searched for in the directories that both precede and follow the `-I-` option.

The standard `aCC` include directories (`/usr/include` and `/opt/aCC/include*`) are always searched last for both types of include files.

Usage:

View-pathing can be particularly valuable for medium to large sized projects. For example, imagine that a project comprises two sets of directories. One set contains development versions of some of the headers that the programmer currently modifies. A mirror set contains the official sources.

Without view-pathing, there is no way to completely replace the default `-Idirectory` search-path with one customized specifically for project development.

With view-pathing, you can designate and separate official directories from development directories and enforce an unconventional search-path order. For quote enclosed headers, the preprocessor can include any header files located in development directories and, in the absence of these, include headers located in the official directories.

If `-I-` is not specified, view-pathing is turned off. This is the default.

Examples:

With view-pathing off, the following example obtains all the quoted include files from `dir1` only if they are not found in the directory of `a.C` and from `dir2` only if they are not found in `dir1`. Finally, if necessary, the standard include directories are searched. Angle-bracketed include files are searched for in `dir1`, then `dir2`, followed by the standard include directories.

```
aCC -Idir1 -Idir2 -c a.C
```

With view-pathing on, the following example searches for quoted include files in `dir1` first and `dir2` next, followed by the standard include directories, ignoring the directory of `a.C`. Angle-bracketed includes are searched for in `dir2` first, followed by the standard include directories.

```
aCC -Idir1 -I- -Idir2 -c a.C
```

NOTE: Some of the compiler's header files are included using double quotes. Since the `-I-` option redefines the search order of such includes, if any standard headers are used, it is your responsibility to supply the standard include directories (`/opt/aCC/include*` and `/usr/include`) in the correct order in your `-I-` command line.

For example, when using `-I-` on the aCC command line, any specified `-I directory` containing a quoted include file having the same name as an HP-UX system header file, may cause the following possible conflict.

In general, if your application includes no header having the same name as an HP-UX system header, there is no chance of a conflict.

Suppose you are compiling program `a.C` with view-pathing on. `a.C` includes the file `a.out.h` which is a system header in `/usr/include`:

```
aCC -IDevelopmentDir -I- -IOfficialDir a.C
```

If `a.C` contains:

```
// This is the file a.C
#include <a.out.h>
// ...
```

When `a.out.h` is preprocessed from the `/usr/include` directory, it includes other files that are quote included (like `#include "filehdr.h"`).

Since with view-pathing, quote enclosed headers are not searched for in the including file's directory, `filehdr.h` which is included by `a.out.h` will not be searched for in `a.out.h`'s directory (`/usr/include`).

Instead, for the above command line, the system header is first searched for in `DevelopmentDir`, then in `OfficialDir` and if it is found in neither, it is finally searched for in the standard include directories, `/opt/aCC/include*` and `/usr/include`, in the latter of which it will be found.

However, if you have a file named `filehdr.h` in `DevelopmentDir` or `OfficialDir`, that file (the wrong file) will be found.

Online Help Option

Use the `+help` option to view the HP aC++ Online Help.

+help

```
+help
```

The `+help` option invokes the initial menu window of this *HP aC++ Online Help*.

If `+help` is used on any command line, the compiler displays the *HP aC++ Online Help* with the default web browser and then processes any other arguments.

If `$DISPLAY` is set, the default web browser is used. If the display variable is not set, a message is displayed. Set your `$DISPLAY` variable as follows:

`export DISPLAY=YourDisplayAddress` (ksh/sh shell notation)

`setenv DISPLAY YourDisplayAddress` (csh shell notation)

Examples:

To use a browser other than the default, first set the `BROWSER` environment variable to the alternate browser's location:

```
export BROWSER=AlternateBrowserLocation
```

To invoke the online guide, use the command:

```
aCC +help
```

Inlining Options

These options allow you to specify the amount of source code inlining done by HP aC++.

+inline_level num

```
+inline_level num
```

The `+inline_level` option controls how C/C++ inlining hints influence aCC or cc. Such inlining happens in addition to functions that are explicitly tagged with the `inline` keyword. (For C89, use the `__inline` keyword). This option controls functions declared with the `inline` keyword or within the class declaration, and is effective at all optimization levels.

NOTE: The `+d` and `+inline_level 0` options turn off all inlining, including implicit inlining.

The format for ***num*** is ***N***[***.n***], where ***num*** is either an integral value from 0 to 9, or a value with a single decimal place from 0.0 to 9.0, as described in the following table:

num	**Description**
0	No inlining is done (same effect as the `+d` option).
1	Only functions marked with the `inline` keyword or implied by the language to be `inline` are considered for inlining. This is the default for C++ at `+O1`.
1.0 < *num* < 2.0	Increasingly makes inliner more aggressive below 2.0.
2	Provides more inlining than level 1. This is the default level at optimization levels `+O2`, `+O3`, and `+O4`.
2.0 < *num* < 9.0	Increasing levels of inliner aggressiveness.
9	Attempts to inline all functions other than recursive functions or those with a variable number of arguments.

The default level depends on `+Olevel` as shown in the following table:

level	**num**
0	1
1	1
2	2
3	2
4	2

The `+O[no]inline` option controls the high-level optimizer that recognizes other opportunities in the same source file (`+O3`) or amongst all source files (`+O4`). For example,

```
aCC +inline_level 3 app.C
```

Library Options

Library options allow you to create, use, and manipulate libraries.

-b

`-b`

The `-b` option creates a shared library rather than an executable file.

Example:

The following command links `utils.o` and creates the shared library `utils.so`.

```
aCC -b utils.o -o utils.so
```

For more information on shared libraries, see "Creating and Using Libraries" (page 175).

-dynamic

`-dynamic`

The `-dynamic` option produces dynamically bound executables. See "-minshared" (page 50) for partially statically bound executables.

The default is `-dynamic`.

-exec

`-exec`

The `-exec` option indicates that any object file created will be used to create an executable file. Constants with a protected or hidden export class are placed in the read-only data section. This option also implies `-Bprotected_def`. It makes all defined functions and data (even tentatively defined data) protected by default (unless otherwise specified by another binding option or pragma).

-lname

`-lname`

The `name` value forms part of the name of a library the linker searches for when looking for routines called by your program.

The `-lname` option causes the linker to search one of the following default libraries, if they exist, in an attempt to resolve unresolved external references:

- `/usr/lib/lib/hpux32/name.so`
- `/usr/lib/lib/hpux32/name.a`
- `/opt/langtools/lib/hpux32lib/name.so`
- `/opt/langtools/lib/hpux64lib/name.a`

Whether it searches the shared library (`.so`) or the archive library (`.a`) depends on the value of the `-a` linker option or the `-minshared` compiler option.

NOTE: Because a library is searched when its name is encountered, placement of a `-l` is significant. If a file contains an unresolved external reference, the library containing the definition must be placed after the file on the command line. For details refer to the description of `ld` in the HP-UX Reference Manual or the `ld(1)` manpage for more information.

Example:

```
aCC file.o -lnumeric
```

This command directs the linker to link `file.o` and (by default) search the library `/usr/lib/hpux32/libnumeric.so`.

-L directory

`-L directory`

The `directory` parameter is the HP-UX directory where you want the linker to search for libraries your program uses before searching the default directories.

The `-L directory` option causes the linker to search for libraries in directory in addition to using the default search path.

See `-lname` option for default search path.

The `-L` option must precede any `-lname` option entry on the command line; otherwise `-L` is ignored. This option is passed directly to the linker.

Example:

The following example compiles and links `prog.C` and directs the linker to search the directories and `/project/libs` for any libraries that `prog.C` uses (in this case, `mylib1` and `mylib2`).

```
aCC -L/project/libs prog.C -lmylib1 -lmylib2
```

-minshared

`-minshared`

The `-minshared` option indicates that the result of the current compilation is going into an executable file that will make minimal use of shared libraries. This option is equivalent to `-exec -Bprotected`.

+nostl

`+nostl`

By eliminating references to the standard header files and libraries bundled with HP aC++, the `+nostl` option allows experienced users full control over the header files and libraries used in compilation and linking of their applications, without potential complications that arise in mixing different libraries.

NOTE: Complete understanding of the linking process and the behavior of the actual (third party) libraries linked with the application is essential to avoid link or runtime failures.

For more information on shared libraries, see "Creating and Using Libraries" (page 175).

+Onolibcalls=

`+Onolibcalls=function1,function2,...`

This option allows you to turn off libcall optimizations (inlining or replacement) for calls to the listed functions. This option overrides system header files.

Linker Options

You can specify the following linker options on the compiler command line:

-e epsym

`-e epsym`

Using the `-e epsym` option sets the default entry point address for the output file to be the same as the symbol `epsym`. This option only applies to executable files. It does not work if `epsym=xec`.

-n

`-n`

The `-n` option causes the program file produced by the linker to be marked as sharable.

For details and system defaults, refer to the description of ld in the *HP-UX Reference Manual* or the `ld(1)` manpage for more information.

-N

`-N`

The `-N` option causes the program file produced by the linker to be marked as unsharable.

For details and system defaults, refer to the description of ld in the *HP-UX Reference Manual* or the `ld(1)` manpage for more information.

+O[no]dynopt

`+O[no]dynopt`

Supported only on HP-UX 11.31 systems, the `+O[no]dynopt` option enables [disables] dynamic optimization for the output file. Both forms of this option change the default setting, which allows the run-time environment to enable or disable dynamic optimization according to a system-wide default. This option applies only to executable files and shared libraries, if the run-time environment supports this feature. `chatr(1)` can be used to change this setting, including restoration of the default setting, after the output file has been created.

-q

`-q`

The `-q` option causes the output file from the linker to be marked as demand-loadable.

For details and system defaults, refer to the description of ld in the *HP-UX Reference Manual* or the `ld(1)` manpage for more information.

-Q

`-Q`

The `-Q` option causes the program file from the linker to be marked as not demand-loadable.

For details and system defaults, refer to the description of ld in the *HP-UX Reference Manual* or the `ld(1)` manpage for more information.

-r

`-r`

Use the `-r` option to retain relocation information in the output file for subsequent relinking.

-s

`-s`

Using the `-s` option causes the executable program file created by the linker to be stripped of symbol table information. Specifying this option prevents using a symbolic debugger on the resulting program. For details and system defaults, refer to the description of `ld` in the *HP-UX Reference Manual* or the `ld(1)` manpage for more information.

-usymbol

`-usymbol`

Enter `symbol` as an undefined symbol in `ld`'s symbol table. The resulting unresolved reference is useful for linking a program solely from object files in a library. More than one symbol can be specified, but each must be preceded by `-u`.

See `ld(1)` manpage for more information.

+ild

`+ild`

The `+ild` option specifies incremental linking. If the output file does not exist, or if it was created without the `+ild` option, the linker performs an initial incremental link. The output file produced is suitable for subsequent incremental links. The incremental link option is valid for both executable and shared library links. It is not valid for relocatable links, options or tools that strip the output module, and certain optimization options.

When sum reduction optimization is enabled, the compiler may evaluate intermediate partial sums of float or double precision terms using (wider) extended precision, which reduces variation in the result caused by different optimization strategies and generally produces a more accurate result.

See `ld(1)` manpage for more information.

+ildrelink

`+ildrelink`

The `+ildrelink` option performs an initial incremental link, regardless of the output load module. In certain situations during incremental linking (for example, internal padding space is exhausted), the incremental linker is forced to perform an initial incremental link. The `+ildrelink` option allows you to avoid such unexpected initial incremental links by periodically rebuilding the output file.

See `ld(1)` manpage for more information.

Options for Naming the Output File

These options allow you to name the compilation output file something other than the default name.

-o

`-o outfile`

The `outfile` parameter is the name of the file containing the output of the compilation. This option causes the output of the compilation to be placed in `outfile`.

Without this option the default name is `a.out`. When compiling a single source file with the `-c` option, you can use the `-o` option to specify the name and location of the object file.

-.suffix

`-.suffix`

The `suffix` parameter represents the character or characters to be used as the output file name suffix. `suffix` cannot be the same as the original source file name suffix. Using this option causes the compiler to direct output from the `-E` option into a file with the corresponding .suffix instead of into a corresponding `.c` file.

Example:

```
aCC -E -.i prog.C
```

This command preprocesses the code in `prog.C` and puts the resulting code in the file `prog.i`.

Native Language Support Option

The following is an option to enable native language support:

-Y

`-Y`

The `-Y` option enables Native Language Support (NLS) of 8-bit, 16-bit and 4-byte EUC characters in comments, string literals, and character constants.

The language value (refer to `environ(5)` for the `LANG` environment variable) is used to initialize the correct tables for interpreting comments, string literals, and character constants. The language value is also used to build the path name to the proper message catalog.

For more information and description of the NLS model, refer to `hpnls`, `lang`, and `environ` in *HP-UX Reference Manual*.

Handling Null Pointers Options

The following options allow dereferencing of null pointers.

-z

`-z`

The `-z` option disallows dereferencing of null pointers at run time.

Fatal errors result if null pointers are dereferenced. If you attempt to dereference a null pointer, a `SIGSEGV` error occurs at run time.

Example:

```
aCC -z file.C
```

The above command compiles `file.C` and generates code to disallow dereferencing of null pointers.

For more information, see `signal(2)` and `signal(5)` manpages.

-Z

`-Z`

The `-Z` option allows dereferencing of null pointers at run time. This is the default. The value of a dereferenced null pointer is zero.

Code Optimizing Options

Optimization options can be used to improve the execution speed of programs compiled with the HP compiler.

To use optimization, first specify the appropriate basic optimization level (`+O1`, `+O2`, `+O3`, or `+O4`) on the command line followed by one or more finer or more precise options when necessary.

For more information and examples, refer to Chapter 7: "Optimizing HP aC++ Programs" (page 156).

This section discusses the following topics:

- "Basic Optimization Level Options" (page 53)
- "Additional Optimization Options for Finer Control" (page 55)
- "Advanced +Ooptimization Options" (page 57)
- "Profile-Based Optimization Options" (page 64)
- "Displaying Optimization Information" (page 65)

Basic Optimization Level Options

The following options allow you to specify the basic level of optimization.

Compiling files at optimization level 2 ("`-O`" or "`+O2`") and above increases the amount of virtual memory needed by the compiler. In cases where very large functions or files are compiled at `+O2`, or in cases where aggressive (`+O3` and above) optimization is used, ensure that the `maxdsiz` kernel tunable is set appropriately on the machine where compilation takes place.

HP recommends a setting of 0x100000000, or 4 GB (the default for this parameter is 0x100000000, or 4 GB) for `maxdsiz_64bit` in such cases. Updating the `maxdsiz_64bit` tunable will ensure that the compiler does not run out of virtual memory when compiling large files or functions.

In addition, `maxssiz_64bit` should be set to 128 MB for very large or complex input files. (Normally a `maxssiz_64bit` setting of 64 MB will be sufficient.)

See the `kctune` man page for more information on how to change kernel tunable parameters.

-O

`-O`

The `-O` option invokes the optimizer to perform level 2 optimization. This option is equivalent to `+O2` option.

Example:

This command compiles `prog.C` and optimizes at level 2:

```
aCC -O prog.C
```

+O0

`+O0`

Use `+O0` for fastest compile time or with simple programs. No optimizations are performed.

Example:

This command compiles `prog.C` and optimizes at level 0:

```
aCC +O0 prog.C
```

+O1

`+O1`

The `+O1` option performs level 1 optimization only. This includes branch optimization, dead code elimination, faster register allocation, instruction scheduling, peephole optimization, and generation of data prefetch instructions for the benefit of direct (but not indirect) memory accesses. This is the default optimization level.

Example:

This command compiles `prog.C` and optimizes at level 1:

```
aCC +O1 prog.C
```

+O2

`+O2`

The `+O2` option performs level 2 optimization. This includes level 1 optimizations plus optimizations performed over entire functions in a single file.

NOTE: Compiling with this optimization setting may require additional memory resources. Refer to the memory resource discussion above.

Example:

This command compiles `prog.C` and optimizes at level 2:

```
aCC +O2 prog.C
```

+O3

`+O3`

The `+O3` option performs level 3 optimization. This includes level 2 optimizations plus full optimization across all subprograms within a single file.

NOTE: Compiling with this optimization setting may require additional memory resources. Refer to the memory resource discussion above.

Example:

This command compiles `prog.C` and optimizes at level 3:

```
aCC +O3 prog.C
```

+O4

```
+O4
```

The `+O4` option performs level 4 optimization. This includes level 3 optimizations plus full optimizations across the entire application program. Also, the defaults that depend on optimization will be the defaults for `+O3`.

When you link a program, the compiler brings all modules that were compiled at optimization level 4 into virtual memory at the same time. Depending on the size and number of the modules, compiling at `+O4` can consume a large amount of virtual memory. If you are linking a large program that was compiled with the `+O4` option, you may notice a system slow down. In the worst case, you may see an error indicating that you have run out of memory.

NOTE: Compiling with this optimization setting may require additional memory resources. Refer to the memory resource discussion above.

Example:

This command compiles `prog.C` and optimizes at level 4:

```
aCC +O4 prog.C
```

If you run out of memory when compiling at `+O4` optimization, there are several things you can do:

- Compile at `+O4` only those modules that need to be compiled at optimization level 4, and compile the remaining modules at a lower level.
- If you still run out of memory, increase the per-process data size limit. Run the System Administrator Manager (SAM) to increase the `maxdsiz_64bit` process parameter to more than 4GB. This procedure provides the process with additional data space.
- If increasing the per-process data size limit does not solve the problem, increase the system swap space.

  Refer to the *System Administration Tasks* manual for more information.

Object Files Generated at Optimization Level 4

Object files generated by the compiler with `+O4` or `-ipo`, called intermediate object files, are intended to be temporary files. These object files contain an intermediate representation of the user code in a format that is designed for advanced optimizations. The size of these intermediate object files can typically be 3 to 10 times as large as normal object files. Hewlett-Packard reserves the right to change the format of these files without prior notice. There is no guarantee that intermediate object files will be compatible from one revision of the compiler to the next. Use of intermediate files must be limited to the compiler that created them. For the same reason, intermediate object files should not be included into archived libraries that might be used by different versions of the compiler. The compiler will issue an error message and terminate when an incompatible intermediate file is generated.

Additional Optimization Options for Finer Control

Following are the additional optimizations options for finer control:

-ipo

The `-ipo` option enables interprocedural optimizations across files. The object file produced using this option contains intermediate code (IELF file). At link time,`ld` automatically invokes the interprocedural optimizer (`u2comp`), if any of the input object files is an IELF file.

For optimization levels `+O0` and `+O1`, this option is silently ignored.

The`-ipo` option will get implicitly invoked with the `+O4` and `+Ofaster` options to match current behavior (`+O4 ==> +O3 -ipo`).

For `-ipo` compilations, the back end is parallelized, and the level of parallelism can be controlled with the environment variable PARALLEL, since the standard HP-UX `make` utility is used for the parallelization.

Object Files Generated with -ipo

Object files generated by the compiler with `+O4` or `-ipo`, called intermediate object files, are intended to be temporary files. These object files contain an intermediate representation of the user code in a format that is designed for advanced optimizations. The size of these intermediate object files can typically be 3 to 10 times as large as normal object files. Hewlett-Packard reserves the right to change the format of these files without prior notice. There is no guarantee that intermediate object files will be compatible from one revision of the compiler to the next. Use of intermediate files must be limited to the compiler that created them. For the same reason, intermediate object files should not be included into archived libraries that might be used by different versions of the compiler. The compiler will issue an error message and terminate when an incompatible intermediate file is generated.

+[no]nrv

`+[no]nrv`

`-nrv_optimization,[off|on]`

The `+[no]nrv` option enables [disables] the named return value (NRV) optimization. By default it is disabled.

The NRV optimization eliminates a copy-constructor call by allocating a local object of a function directly in the caller’s context if that object is always returned by the function.

Example:

```
 struct A{
         A(A const&); //copy-constructor
         };

         A f(A constA x) {
            A a(x);
            return a; // Will not call the copy constructor if the
         }            // optimization is enabled.
```

This optimization will not be performed if the copy-constructor was not declared by the programmer. Note that although this optimization is allowed by the ISO/ANSI C++ standard, it may have noticeable side effects.

Example:

```
aCC -Wc,-nrv_optimization,on app.C
```

+O[no]failsafe

`+O[no]failsafe`

The `+O[no]failsafe` option enables [disables] failsafe optimization. When a compilation fails at the current optimization level `+Ofailsafe` will automatically restart the compilation at `+O2` (for specific high level optimizer errors `+O3/+O4`), `O1`, or `+O0`.

The default is `+Ofailsafe`.

+O[no]aggressive

`+O[no]aggressive`

The `+Oaggressive` option enables aggressive optimizations. The `+Onoaggressive` option disables aggressive optimizations.

By default, aggressive optimizations are turned off. The `+Oaggressive` option is approximately equivalent to `+Osignedpointers +Onoinitcheck +Ofltacc=relaxed`.

NOTE: This option is deprecated and may not be supported in future releases. Instead you can use `+Ofast`option.

+O[no]limit

`+O[no]limit`

The `+Olimit` option enables optimizations that significantly increase compile time or that consume a lot of memory.

The `+Onolimit` option suppresses optimizations regardless of their effect on compile time or memory consumption.

Use `+Onolimit` at all optimization levels.

Usage:

`+O[no]limit=level`

The defined values of `level` are:

`default`	Based on tuning heuristics, the optimizer will spend a reasonable amount of time processing large procedures. This is the default option.
`min`	For large procedures, the optimizer will avoid non-linear time optimizations. This option is a synonym for `+Olimit`.
`none`	The optimizer will fully optimize large procedures, possibly resulting in significantly increased compile time. This option is a synonym for `+Onolimit`.

Example:

To remove optimization time restrictions at `O2`, `O3`, or `O4` optimization levels, use `+Onolimit` as follows:

```
aCC <opt level> +Onolimit sourcefile.C
```

+O[no]ptrs_to_globals[=list]

`+O[no]ptrs_to_globals[=list]`

The `+O[no]ptrs_to_globals` option tells the optimizer whether global variables are accessed [are not accessed] through pointers. If +Onoptrs_to_globals is specified, it is assumed that statically-allocated data (including file-scoped globals, file-scoped statics, and function-scoped statics) will not be read or written through pointers. The default is `+Onoptrs_to_globals`.

+O[no]size

`+O[no]size`

While most optimizations reduce code size, the `+Osize` option suppresses those few optimizations that significantly increase code size. The `+Onosize` option enables code-expanding optimizations.

Use `+Osize` at all optimization levels. The default is `+Onosize`.

Advanced +Ooptimization Options

Advanced optimization options provide additional control for special situations.

+O[no]cross_region_addressing

`+O[no]cross_region_addressing`

The `+O[no]cross_region_addressing` option enables [disables] the use of cross-region addressing. Cross-region addressing is required if a pointer, such as an array base, points to a different region than the data being addressed due to an offset that results in a cross-over into another region. Standard conforming applications do not require the use of cross-region addressing.

The default is `+Onocross_region_addressing`.

NOTE: Using this option may result in reduced runtime performance.

+O[no]datalayout

`+O[no]datalayout`

The `+O[no]datalayout` option enables [disables] profile-driven layout of global and static data items to improve cache memory utilization. This option is currently enabled if `+Oprofile=use` (dynamic profile feedback) is specified.

The default, in the absence of `+Oprofile=use`, is `+Onodatalayout`.

+O[no]dataprefetch

`+O[no]dataprefetch`

When `+Odataprefetch` is enabled, the optimizer inserts instructions within innermost loops to explicitly prefetch data from memory into the data cache. Data prefetch instructions are inserted only for data structures referenced within innermost loops using simple loop varying addresses (that is, in a simple arithmetic progression).

Use this option for applications that have high data cache miss overhead.

`+Odataprefetch` is equivalent to `+Odataprefetch=indirect`. `+Onodataprefetch` is equivalent to `+Odataprefetch=none`.

Usage:

`+Odataprefetch=kind`

The defined values for `kind` are:

`direct`	Enable generation of data prefetch instructions for the benefit of direct memory accesses, but not indirect memory accesses. This is the default at optimization level `+O1`.
`indirect`	Enables the generation of data prefetch instructions for the benefit of both direct and indirect memory accesses. This is the default at optimization levels `+O2` and above. It is treated the same as `direct` at optimization level `+O1`.
`none`	Disables the generation of data prefetch instructions. This is the default at optimization level `+O0`.

+O[no]fltacc

`+O[no]fltacc=level`

The `+O[no]fltacc` option disables [enables] floating-point optimizations that can result in numerical differences. Any option other than `+Ofltacc=strict` also generates Fused Multiply-Add (FMA) instructions. FMA instructions can improve performance of floating-point applications.

If you specify neither `+Ofltacc` nor `+Onofltacc`, less optimization is performed than for `+Onofltacc`. If you specify neither option, the optimizer generates FMA instructions but does not perform any expression-reordering optimizations.

Specifying `+Ofltacc` insures the same result as in unoptimized code (`+O0`).

Usage:

```
+Ofltacc=level
```

The defined values for `level` are:

`default`	Allows contractions, such as fused multiply- add (FMA), but disallows any other floating-point optimization that can result in numerical differences.
`limited`	Like default, but also allows floating-point optimizations which may affect the generation and propagation of infinities, NaNs, and the sign of zero.
`relaxed`	In addition to the optimizations allowed by limited, permits optimizations, such as reordering of expressions, even if parenthesized, that may affect rounding error. This is the same as `+Onofltacc`.
`strict`	Disallows any floating-point optimization that can result in numerical differences. This is the same as `+Ofltacc`.

All options except `+Ofltacc=strict` option allow the compiler to make transformations which are algebraically correct, but which may slightly affect the result of computations due to the inherent imperfection of computer floating-point arithmetic. For many programs, the results obtained with these options are adequately similar to those obtained without the optimization.

For applications in which round-off error has been carefully studied, and the order of computation carefully crafted to control error, these options may be unsatisfactory. To insure the same result as in unoptimized code, use `+Ofltacc`.

Example:

All the options, except `+Ofltacc=strict`, allow the compiler to replace a division by a multiplication using the reciprocal. For example, the following code:

```
for (int j=1;j<5;j++)
   a[j] = b[j] / x;
```

is transformed as follows (note that x is invariant in the loop):

```
x_inv = 1.0/x;
for (int j=1;j<5;j++)
   a[j] = b[j] * x_inv;
```

Since multiplication is considerably faster than division, the optimized program runs faster.

+Ofrequently_called

```
+Ofrequently_called=function1[,function2...]
```

The named functions are assumed to be frequently called. This option overrides any information in a profile database.

```
+Ofrequently_called:filename
```

The file indicated by `filename` contains a list of functions, separated by spaces or newlines. These functions are assumed to be frequently called. This option overrides any information in a profile database.

+O[no]initcheck

```
+O[no]initcheck
```

The initialization checking feature of the optimizer can be `on` or `off`:

When `on` (`+Oinitcheck`), the optimizer issues warning messages when it discovers uninitialized variables.

When `off` (`+Onoinitcheck`), the optimizer does not issue warning messages.

Use `+Oinitcheck` at optimization level 2 or above. If this option is used together with `+check=uninit`, uninitialized variables will remain uninitialized so that an error will be reported at runtime and trigger a program abort if the variables are accessed.

+O[no]inline

`+O[no]inline`

The `+Oinline` option indicates that any function can be inlined by the optimizer. `+Onoinline` disables inlining of functions by the optimizer. This option does not affect functions inlined at the source code level.

Use `+O[no]inline` at optimization levels 2, 3 and 4.

The default is `+Oinline` at optimization levels 3 and 4.

Usage:

`+O[no]inline=function1{,function2...]`

Enables [disables] optimizer inlining for the named functions.

`+O[no]inline:filename`

The file indicated by `filename` should contain a list of function names, separated by commas or newlines. Optimization is enabled [disabled] for the named functions.

+Olit

`+Olit=kind`

The `+Olit` option places the data items that do not require load-time or runtime initialization in a read-only data section. `+Olit=all` is the default for both HP aC++ and HP C. This represents a change from earlier versions of the HP C compiler, which defaulted to `+Olit=const`. Note that if you attempt to modify the constant or literal, a runtime signal 11 will be generated.

The defined values for `kind` are:

`all`	All string literals and all const-qualified variables that do not require load-time or runtime initialization will be placed in a read-only data section. `+Olit=all` replaces the deprecated `+ESlit` option.
`const`	All string literals appearing in a context where `const char *` is legal, and all const-qualified variables that do not require load-time or runtime initialization will be placed in a read-only data section. `+Olit=const` is mapped to `+Olit=all` with a warning, except in C mode. `+Olit=const` replaces the deprecated `+ESconstlit` option in C.
none	No constants are placed in a read-only data section. `+Olit=none` replaces the deprecated `+ESnolit` option.

+Ointeger_overflow

`+Ointeger_overflow=kind`

To provide the best runtime performance, the compiler makes assumptions that runtime integer arithmetic expressions that arise in certain contexts do not overflow (produce values that are too high or too low to represent) both expressions that are present in user code and expressions that the compiler constructs itself. Note that if an integer arithmetic overflow assumption is violated, runtime behavior is undefined.

`+Ointeger_overflow=moderate` is the default for all optimization levels. This was changed to enable a wider class of applications to be compiled with optimization and run correctly.

The defined values of `kind` are:

`conservative`	Directs the compiler to make fewer assumptions that integer arithmetic expressions do not overflow.
`moderate`	Allows the compiler to make a broad set of assumptions so that the integer arithmetic expressions do not overflow, except that linear function test replacement (LFTR) optimization is not performed.

+Olevel

`+Olevel=name1[,name2,...,nameN]`

The `+Olevel` option lowers optimization to the specified level for one or more named functions.

`level` can be 0, 1, 2, 3, or 4.

The `name` parameters are names of functions in the module being compiled. Use this option when one or more functions do not optimize well or properly. This option must be used with a basic `+Olevel` or `-O` option. Note that currently only the C++ mangled name of the function is allowed for name.

This option works like the `OPT_LEVEL` pragma. The option overrides the pragma for the specified functions. As with the pragma, you can only lower the level of optimization; you cannot raise it above the level specified by a basic `+Olevel` or `-O` option. To avoid confusion, it is best to use either this option or the `OPT_LEVEL` pragma rather than both.

You can use this option at optimization levels 1, 2, 3, and 4. The default is to optimize all functions at the level specified by the basic `+Olevel` or `-O` option.

Examples:

- The following command optimizes all functions at level 3, except for the functions `myfunc1` and `myfunc2`, which it optimizes at level 1.

  ```
  aCC +O3 +O1=myfunc1,myfunc2 funcs.c main.c
  ```

- The following command optimizes all functions at level 2, except for the functions `myfunc1` and `myfunc2`, which it optimizes at level 0.

  ```
  aCC -O +O0=myfunc1,myfunc2 funcs.c main.c
  ```

+O[no]loop_transform

`+O[no]loop_transform`

This option transforms [does not transform] eligible loops for improved cache and other performance. This option can be used at optimization levels 2, 3 and 4.

The default is `+Oloop_transform`.

+O[no]loop_unroll

`+O[no]loop_unroll [=unroll_factor]`

The `+O[no]loop_unroll` option enables [disables] loop unrolling. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Oloop_unroll`. The default is 4, that is, four copies of the loop body. The `unroll_factor` controls code expansion. Note that `+Onoloop_unroll` has no effect on loop unroll-and-jam.

+O[no]openmp

`+O[no]openmp`

The `+Oopenmp` option causes the OpenMP directives to be honored. This option is effective at any optimization level. Non OpenMP parallelization directives are ignored with warnings. `+Onoopenmp` requests that OpenMP directives be silently ignored. If neither `+Oopenmp` nor `+Onoopenmp` is specified, OpenMP directives will be ignored with warnings.

The OpenMP specification is available at `http://www.openmp.org/specs`. OpenMP programs require the `libomp` and `libcps` runtime support libraries to be present on both the compilation and runtime systems. The compiler driver automatically includes them when linking.

If you use `+Oopenmp` in an application, you must use `-mt` with any files that are not compiled with `+Oopenmp`. For additional information and restrictions, See “-mt” (page 66).

It is recommended that you use the `-N` option when linking OpenMP programs to avoid exhausting memory when running with large numbers of threads.

NOTE: HP aC++ version A.06.00 does not support C++ constructs in OpenMP. Use the `+legacy_v5` option to use this option.

+opts

`+opts filename`

The file indicated by `filename` contains a list of options that are processed as if they had been specified on the command line at the point of the `+opts` option. The options must be delimited by a blank character. You can add comments to the option file by using a "#" character in the first column of a line. The "#" causes the entire line to be ignored by the compiler.

Example:

```
$ aCC +opts GNUOptions foo.c
```

Where `GNUOptions` contains:

```
#This file contains the set of options for programs needing GNU support -Ag++ -Wc,--fshort-enums
```

+O[no]parminit

`+O[no]parminit`

The `+O[no]parminit` option enables [disables] automatic initialization to non-NaT of unspecified function parameters at call sites. This is useful in preventing NaT values in parameter registers. The default is `+Onoparminit`.

+O[no]parmsoverlap

`+O[no]parmsoverlap`

The `+Onoparmsoverlap` option optimizes with the assumption that on entry to a function each of that function’s pointer-typed formals points to memory that is accessed only through that formal or through copies of that formal made within the function. For example, that memory must not be accessed through a different formal, and that formal must not point to a global that is accessed by name within the function or any of its calls.

Use `+Onoparmsoverlap` if C/C++ programs have been literally translated from FORTRAN programs.

The default is `+Oparmsoverlap`.

+O[no]procelim

`+O[no]procelim`

The `+O[no]procelim` option enables [disables] the elimination of dead procedure code and sometimes the unreferenced data.

Use this option when linking an executable file, to remove functions not referenced by the application. You can also use this option when building a shared library to remove functions not exported and not referenced from within the shared library. This may be especially useful when functions have been inlined.

NOTE: Any function having symbolic debug information associated with it is not removed.

The default is `+Onoprocelim` at optimization levels 0 and 1; at levels 2, 3 and 4, the default is `+Oprocelim`.

+O[no]promote_indirect_calls

`+O[no]promote_indirect_calls`

The `+O[no]promote_indirect_calls` option uses profile data from profile-based optimization and other information to determine the most likely target of indirect calls and promotes them to direct calls. Indirect calls occur with pointers to functions and virtual calls.

In all cases the optimized code tests to make sure the direct call is being taken and if not, executes the indirect call. If `+Oinline` is in effect, the optimizer may also inline the promoted calls.

`+Opromote_indirect_calls` is only effective with profile-based optimization.

NOTE: The optimizer tries to determine the most likely target of indirect calls. If the profile data is incomplete or ambiguous, the optimizer may not select the best target. If this happens, your code's performance may decrease.

This option can be used at optimization levels 3 and 4. At `+O3`, it is only effective if indirect calls from functions within a file are mostly to target functions within the same file. This is because `+O3` optimizes only within a file, whereas `+O4` optimizes across files.

The default is `+Opromote_indirect_calls` at optimization level 3 and above.

`+Onopromote_indirect_calls` will be the default at optimization level 2 and below.

+Orarely_called

+Orarely_called=function1[,function2...]

The `+Orarely_called` option overrides any information in a profile database.

The named functions are assumed to be rarely called

`+Orarely_called:filename`

The file indicated by filename contains a list of functions, separated by spaces or newlines. These functions are assumed to be rarely called. This option overrides any information in a profile database.

+O[no]signedpointers

`+O[no]signedpointers`

NOTE: This option is deprecated and may not be supported in future releases.

The `+Osignedpointers` option treats pointers in Boolean comparisons (for example, <, <=, >, >=) as signed quantities. Applications that allocate shared memory and that compare a pointer to shared memory with a pointer to private memory may run incorrectly if this optimization is enabled.

The default is `+Onosignedpointers`.

NOTE: This option is supported in C-mode only. A warning is displayed in C++ when this option is used.

+Oshortdata

`+Oshortdata[=size]`

All objects of `[size]` bytes or smaller are placed in the short data area, and references to such data assume it resides in the short data area. Valid values of size are a decimal number between 8 and 4,194,304 (4MB).

If no size is specified, all data is placed in the short data area. The default is `+Oshortdata=8`.

NOTE: Using a value that is too big or without the optional size, possibly through `+Ofast`, may give various linker fix up errors, if there is more than 4Mb of short data.

+O[no]store_ordering

`+O[no]store_ordering`

The `+O[no]store_ordering` option preserves [does not preserve] the original program order for stores to memory that is visible to multiple threads. This does not imply strong ordering. The default is `+Onostore_ordering`.

+Otype_safety

`+Otype_safety=kind`

The `+Otype_safety` option controls type-based aliasing assumptions.

The defined values for kind are:

`off`	The default. Specifies that aliasing can occur freely across types.
`limited`	Code follows ANSI aliasing rules. Unnamed objects should be treated as if they had an unknown type.
`ansi`	Code follows ANSI aliasing rules. Unnamed objects should be treated the same as named objects.
`strong`	Code follows ANSI aliasing rules, except that accesses through `lvalues` of a character type are not permitted to touch objects of other types and field addresses are not to be taken.

The default is `+Otype_safety=off`.

+Ounroll_factor

`+Ounroll_factor=n`

The `+Ounroll_factor` option applies the unroll factor to all loops in the current translation unit. You can apply an unroll factor which you think is best for the given loop or apply no unrolling factor to the loop. If this option is not specified, the compiler uses its own heuristics to determine the best unroll factor for the inner loop.

A user specified unroll factor will override the default unroll factor applied by the compiler.

Specifying `n=1` will prevent the compiler from unrolling the loop.

Specifying `n=0` allows the compiler to use its own heuristics to apply the unroll factor.

NOTE: This option will be ignored if it is placed in a loop other than the innermost loop.

Profile-Based Optimization Options

Profile-based optimization is a set of performance-improving code transformations based on the runtime characteristics of your application.

+Oprofile

`+Oprofile=[use|collect]`

The `+Oprofile` option instructs the compiler to instrument the object code for collecting runtime profile data. The profiling information can then be used by the linker to perform profile-based optimization. When an application finishes execution, it will write profile data to the file`flow.data` or to the file/path in the environment variable `FLOW_DATA (if set).`

`+Oprofile=use[:filename]` causes the compiler to look for a profile database file. If a filename is not specified, the compiler will look for a file named "flow.data" or the file/path specified in the `FLOW_DATA` environment variable. If a filename is specified, it overrides the `FLOW_DATA` environment variable.

After compiling and linking with `+Oprofile=collect`, run the resultant program using representative input data to collect execution profile data. Profile data is stored in `flow.data` by default. The name is generated as `flow.<suffix>` if there is already a `flow.data`file present in the current directory. Finally, recompile with the `+Oprofile=use` option (passing it the appropriate filename if necessary) to perform profile-based optimization.

Example:

```
aCC +Oprofile=collect -O -o prog.pbo prog.C
```

The above command compiles `prog.C` with optimization, prepares the object code for data collection, and creates the executable file `prog.pbo`. Running `prog.pbo` collects runtime information in the file `flow.data` in preparation for optimization with `+Oprofile=use`.

```
+Oprofile=collect [:<qualifiers>]
```

`<qualifiers>` are a comma-separated list of profile collection qualifiers.

Supported profile collection qualifiers:

`arc`	Enables collection of `arc` counts.
`dcache`	Enables collection of data cache misses.
`stride`	Enables collection of `stride` data.
`loopiter`	Enables collection of loop iteration counts..
`all`	Enables collection of all types of profile data. This is equivalent to `+Oprofile=collect:arc,dcache,stride,loopiter`. This is the default.

This option merely enables the application for collection of the various forms of profiling data.

The environment variable `PBO_DATA_TYPE` controls the type of data collected at runtime. It may be set to one of the following values, which must be consistent with the `+Oprofile=collect` qualifiers used to create the application:

`arc-stride`	Collects `stride` and/or `arc` counts. This is the default if `PBO_DATA_TYPE` is not set.
`dcache`	Collects data cache miss metrics.

NOTE: Data cache miss metrics cannot be collected during the same run of an application as `stride` and/or `arc` data.

Information Embedding Options

The `+annotate` option annotates the compiled binary with extra information.

-annotate=structs

The`+annotate` option annotates the compiled binary with accesses to C/C++ `struct` fields for use by other external tools such as Caliper. By default, no annotations are added.

Displaying Optimization Information

The `+O[no]info` option displays informational messages about the optimization process.

+O[no]info

```
+O[no]info
```

The `+O[no]info` option displays messages about the optimization process. This option may be helpful in understanding what optimizations are occurring. You can use the option at levels 0-4.

The default is `+Onoinfo` at levels 0-4.

Parallel Processing Options

HP aC++ provides the following optimization options for parallel code.

-mt

The -mt option enables multi-threading capability without the need to set any other flags, such as -l and -D. HP aC++ examines your environment and automatically selects and sets the appropriate flags. "Performance Options" (page 167).

There are three possible sets of flags depending on your operating system and the libstd you use. Table 5 lists the option matrix for -mt.

Table 5 Option Matrix for -mt

Libraries	Flags
old-lib libstd 1.2.1 (-AP)& librwtool 7.0.x	-D_REENTRANT -DRW_MULTI_THREAD -DRWSTD_MULTI_THREAD -D_THREAD_SAFE -D_POSIX_C_SOURCE=199506L -D_HPUX_SOURCE * -lpthread
new-lib (-AA) libstd 2.2.1	-D_REENTRANT -D_RW_MULTI_THREAD -D_RWSTD_MULTI_THREAD -D_POSIX_C_SOURCE=199506L -D_HPUX_SOURCE * -lpthread
C mode -Ae/-Aa	-D_REENTRANT -D_POSIX_C_SOURCE=199506L -lpthread

* required if -D_POSIX_C_SOURCE is used.

NOTE: For C++ and C -Ae -D_HPUX_SOURCE is set to be compatible with the default when -mt is not used. For C mode options -AC89, -AC99, and -Aa, -D_HPUX_SOURCE is also set. If you do not want to use-D_HPUX_SOURCE, you can undefine it by using -U. Example: -U_HPUX_SOURCE

The following macros are used to compile multi-thread source code:

- _REENTRANT

  Required by system header files that provide reentrant functions (suffixed by _r).

- RW_MULTI_THREAD / _RW_MULTI_THREAD

  Required by Rogue Wave toolsh++ header files and libraries. RW_MULTI_THREAD is used by toolsh++ 7.0.x. _RW_MULTI_THREAD is used by toolsh++ 8.x (not available yet).

- RWSTD_MULTI_THREAD / _RWSTD_MULTI_THREAD

  Required by Rogue Wave standard library header files and libraries. RWSTD_MULTI_THREAD is used by libstd 1.2.1. _RWSTD_MULTI_THREAD is used by libstd 2.2.1 when compiling with -AA.

- `_POSIX_C_SOURCE=199506L`

  Required by `pthread`.
- `libpthread.*`

  Kernel thread library used on 11.x systems

See "Using Threads" (page 163) for more information.

NOTE: Make sure that `-mt` is used consistently at compile and link times. When you link with `-mt`, everything must be compiled with `-mt`, even if you do not think your file will be used in a threaded application. When you incorrectly mix and match with `-mt`, you get a runtime abort with the following message:

```
aCC runtime: Use of "-mt" must be consistent during both compilation and linking.
```

To find the library or object that is missing `-mt`, use `/usr/ccs/bin/footprints` and look for the following:

```
-mt [(off) 1] -mt [on 1] (Or not present)
```

The number 1 above is the count of objects with that `-mt` setting. Not present implies the source was not compiled with a recent compiler that contains this information.

+O[no]autopar

`+O[no]autopar`

The `+Oautopar` option enables automatic parallelization of loops that are deemed safe and profitable by the loop transformer.

Usage:

This optimization allows applications to exploit otherwise idle resources on multicore or multiprocessor systems by automatically transforming serial loops into multithreaded parallel code. When the `+Oautopar` option is used at optimization levels `+O3` and above, the compiler automatically parallelizes those loops that are deemed safe and profitable by the loop transformer.

Automatic parallelization can be combined with manual parallelization through the use of OpenMP directives and the `+Oopenmp` option. When both `+Oopenmp` and `+Oautopar` options are specified, then any existing OpenMP directives take precedence, and the compiler will only consider auto-parallelizing other loops that are not controlled by those directives.

Programs compiled with the `+Oautopar` option require the libcps, libomp, and libpthreads runtime support libraries to be present at both compilation and runtime. When linking with the HP-UX B.11.61 linker, compiling with the `+Oautopar`option causes them to be automatically included. Older linkers require those libraries to be specified explicitly or by compiling with `+Oopenmp`.

The `+Oautopar` option is supported when compiling C, C++, or Fortran files. Specifying `+Oautopar` implies the `-mt` option.

The default is `+Onoautopar`, which disables automatic parallelization of loops.

+tls=[static|dynamic]

`+tls=[static|dynamic]`

The `+tls` option specifies whether references to thread local data items are to be performed according to the mode.

Usage:

`+tls=mode`

The defined values of `mode` are:

`static`	This is a more efficient mode in which only thread local data in the program startup set can be accessed.

`dynamic`	This is a less efficient mode in which thread local data outside the program startup set can be accessed as well. This is the default.

Translation units compiled with different settings of this option may be freely mixed, even within the same load module.

+wlock

`+wlock`

The `+wlock` option enables compile-time diagnostic messages for potential errors in using lock/unlock calls in programs that use pthread-library-based lock/unlock functions. Warnings are emitted for acquiring an already acquired lock, releasing an already released lock, and unconditionally releasing a lock that has been conditionally acquired.

This diagnostic checking is based on cross-module analysis performed by the compiler. Therefore, the `+wlock`option implicitly enables a limited form of cross-module analysis, even if `-ipo` or`+O4` options are not specified. This can lead to a significant increase in the compile time compared to a build without the `+wlock` option. Using this option could result in the compiler invoking optimizations other than those that are part of the user-specified optimization level. If `+wlock` is used in addition to `-ipo` or `+O4`, the generated code is not affected, and the compile time does not increase much.

Performance Options

The HP compiler provides a variety of options to help improve build and runtime performance. These options are:

-fast

`-fast`

The `-fast` option selects a combination of optimization options for optimum execution speed and reasonable build times. This option is equivalent to `+Ofast`. Currently chosen options are:

- `+O2`
- `+Ofltacc=relaxed`
- `+Onolimit`
- `+DSnative`
- `+FPD`

You can override any of the options in `-fast` by specifying a subsequent option after it.

Use this option when porting C++ and C applications compiled on other UNIX operating systems to HP-UX.

NOTE: Do not use this option for programs that depend on IEEE standard floating-point denormalized numbers. Otherwise, different numerical results may occur.

+Ofast

`+Ofast`

The `+Ofast` option selects a combination of optimization options for optimum execution speed for reasonable build times. Currently chosen options are:

- `+O2`
- `+Ofltacc=relaxed`
- `+Onolimit`
- `+DSnative`

- `+FPD`
- `-Wl,+pi,1M`
- `-Wl,+pd,1M`
- `-Wl,+mergeseg`

This option is a synonym for `-fast`.

NOTE: Do not use this option for programs that depend on IEEE standard floating point denormalized numbers. Otherwise, different numerical results may occur. See `+Ofltacc=relaxed`.

+Ofaster

`+Ofaster`

The `+Ofaster` option is equivalent to `+Ofast` with an increased optimization level. The definition of `+Ofaster` may change, or the option may be deprecated in future releases.

+O[no]tls_calls_change_tp

`+O[no]tls_calls_change_tp`

The +O[no]tls_calls_change_tp option specifies whether or not function calls can change the value of the thread pointer(tp), resulting in less aggressive optimizations to TLS variables which are accessed by name.

+[no]srcpos

`+[no]srcpos`

The `+[no]srcpos` option controls the generation of source position information for HP Caliper. The default is `+srcpos`.

When `+srcpos`, is in effect, the compiler generates source position information. When `+nosrcpos` is in effect, the compiler does not generate this information and the compiler instructs the linker to discard any of this information found in the object files.

+DSmodel

`+DSmodel`

The `+DSmodel` option performs instruction scheduling for a particular implementation of the Itanium®-based architecture. The default is `blended`.

`model` any of the values below.

`blended`	Tune to run reasonably well on multiple implementations. As old implementation become less important and new implementations are added, the behavior with this value will change accordingly.
`itanium`	Tune for the Itanium® processor.
`itanium2`	Tune for the Itanium2® processor.
`mckinley`	See `itanium2®`.
`montecito`	Tune for the Montecito® processor.
`poulson`	Tune for the Poulson® processor.
`native`	Tune for the processor on which the compiler is running.

Porting Options

Use the following options as necessary when porting your code from other operating environments to HP-UX.

-fast

`-fast`

The `-fast` option selects a combination of optimization options for optimum execution speed and reasonable build times. Currently chosen options are:

- `+O2`
- `+Ofltacc=relaxed`
- `+Onolimit`
- `+DSnative`
- `+FPD`

You can override any of the options in `-fast` by specifying a subsequent option after it. This option is equivalent to `+Ofast`.

Use this option when porting C++ and C applications compiled on other UNIX operating systems to HP-UX.

NOTE: Do not use this option for programs that depend on IEEE standard floating-point denormalized numbers. Otherwise, different numerical results may occur.

+sb

`+sb`

The `+sb` option specifies unqualified `char`, `short`, `int`,`long`, and `long long` bitfields as signed. The default is `+sb`.

NOTE: When both `+sb` and `+uc` are in effect, `+uc` will override this for `char` bit fields.

+ub

`+ub`

The `+ub` option specifies unqualified `char`, `short`, `int`, `long`, and `long long` bitfields as unsigned. This option has no effect on signedness of `enum` bitfields or on signedness of non-bitfield `char`. The default is `+sb`.

+uc

`+uc`

By default, all unqualified char data types are treated as signed `char`. Specifying `+uc` causes an unqualified (plain) `char` data type to be treated as unsigned `char`. (Overloading and mangling are unchanged.)

Use this option to help in porting applications from environments where an unqualified (plain) `char` type is treated as unsigned `char`.

NOTE: Since all unqualified `char` types in the compilation unit will be affected by this option (including those headers that define external and system interfaces), it is necessary to compile the interfaces used in a single program uniformly.

+w64bit

The `+w64bit` option enables warnings that help detection of potential problems in converting 32-bit applications to 64-bit. The option is equivalent to the `+M2` option.

+wdriver

The `+wdriver` option enables warnings for PA-RISC options that would otherwise be ignored silently on Integrity servers. With the addition of this option in version A.06.05, a number of warnings for PA options that had been reported by previous compiler versions were changed to be silently ignored by default. The intent is to give good PA-RISC to Integrity makefile compatibility by default, but provide this option to help users clean up unnecessary or misleading use of legacy options when desired.

+wendian

This option allows the user to identify areas in their source code that might have porting issues when going between `little-endian` and `big-endian`.

+wendian will warn of a de-reference that could cause endian-dependent behavior:

```
char charVal = *(char *) int_ptr;
short shortVal = ((short *) long_ptr)[0];
```

This warning can be suppressed by adding an extra cast:

```
char charVal = *(char *) (void *)int_ptr; // OK
```

+wendian warns that the initialization which may be endian-dependent, such as using hex constants to init byte arrays:

```
char a[4] = { 0x11, 0x22, 0x33, 0x44 };
char b[4] = { 'a', 'b', 'c', 'd'}; // OK
```

This warning can be suppressed by not using a hex/octal constant:

```
char a[4] = { 17, 0x22, 0x33, 0x44 }; // OK
```

+wendian also warns of unions that make assumptions about data layout:

```
union u1 {
char c[4];
int v; };
```

```
union u2 {
long long ll;
short s[4];
char c[8]; };
```

This warning can be suppressed by adding a dummy member:

```
union u1 { // OK
char c[4];
int v;
char dummy; };
```

Another type of warning is on the use of IO functions that read/write persistent data from files that may be endian-dependent:

```
read(0, &i, sizeof(i));
fread(&ai[0], sizeof(int), elems_of(ai, int), stdin);

write(1, &i, sizeof(i));
fwrite(&ai[0], sizeof(int), elems_of(ai, int), stdout);
```

This warning can be suppressed by adding an extra cast:

```
fread((char*)(void*)ai, sizeof(char), 1, stdin); // OK
```

Another `+wendian` warning captures cases where a cast when later dereferenced can cause endian issues.

Preprocessor Options

The following options are accepted by the preprocessor:

-C

`-C`

Using the `-C` option prevents the preprocessor from stripping comments. See the description of `cpp` in the `cpp(1)` manpage for details.

-dM

`-dM`

When `-dM` is present, instead of normal preprocessor output the compiler lists the `#define` directives it encounters as it preprocesses the file, thus providing a list of all macros that are in effect at the start of the compilation. The `-dM` option requires that `-P` or `-E` also be specified.

A common use of this option is to determine the compiler's predefined macros. For example:

```
touch foo.c ; cc -E -dM foo.c
```

-Dname

`-Dname[=def]`

`name` is the symbol name that is defined for the preprocessor.

`def` is the definition of the symbol name (`name`).

The `-Dname` option defines a symbol name (`name`) to the preprocessor, as if defined by the preprocessing directive`#define`.

If no definition (`def`) is given, the name is defined as `1`.

NOTE: `__ia64` and `__HP_aCC` are defined automatically.

Example:

The following example defines the preprocessor symbol `DEBUGFLAG` and gives it the value `1`.

```
aCC -DDEBUGFLAG file.C
```

The following program uses this symbol:

```
#include <iostream.h>
int main(){
   int i, j;
   #ifdef DEBUGFLAG
   int call_count=0;
   #endif
   /* ... */
}
```

-E

`-E`

Using the `-E` option runs only the preprocessor on the named C++ files and sends the result to standard output (`stdout`).

An exception to this rule is when`-E` is used with `+Make[d]` option, the only output is the `make` dependency information. Unlike the `-P` option, the output of `-E` contains `#line` entries indicating the original file and line numbers.

Redirecting Output From This Option

Use the `-.suffix` option to redirect the output of this option.

make[d]

`+make[d]`

The `+make[d]` option directs a list of the quote enclosed (" ") header files upon which your source code depends to `stdout`. The list is in a format accepted by the `make` command.

If `+maked` is specified, the list is directed to a `.d` file. The `.d` file name prefix is the same as that of the object file. The `.d` file is created in the same directory as the object file.

Usage:

Use `+maked` when you also specify the `-E` or the `-P` option. When used with the `-E` option, only dependency information is generated.

Table 6 lists examples of the `+make[d]` option.

Table 6 Examples

Command line Specified	.d file name	.d file location	Preprocessing output
`aCC -c +make a.C`	none	`stdout`	none
`aCC -c -E -.i +maked a.C`	`a.d`	current directory	none
`aCC -c -P +maked a.C -o b.o`	`b.d`	current directory	`a.i`
`aCC -c -P +maked a.C -o /tmp/c`	`c.d`	`/tmp` directory	`a.i`

+Make[d]

`+Make[d]`

The `+Make[d]` option directs a list of both the quote enclosed (" ") and angle bracket enclosed (< >) header files upon which your source code depends to `stdout`. The list is in a format accepted by the `make` command.

If `+Maked` is specified, the list is directed to a `.d` file. The `.d` file name prefix is the same as that of the object file. The `.d` file is created in the same directory as the object file.

Usage:

Use `+Maked` when you also specify the `-E` or the `-P` option. When used with the `-E` option, only dependency information is generated.

Table 7 Examples

Command line specified	.d file name	.d file location	Preprocessing output
`aCC -c +Make a.C`	none	`stdout`	none
`aCC -c -E -.i +Maked a.C`	`a.d`	current directory	none
`aCC -c -P +Maked a.C -o b.o`	`b.d`	current directory	`a.i`
`aCC -c -P +Maked a.C -o /tmp/c`	`c.d`	`/tmp` directory	`a.i`

-P

`-P`

Using the `-P` option only preprocesses the files named on the command line without invoking further phases. It leaves the result in corresponding files with the suffix `.i`.

For example, the following command preprocesses the file `prog.C` leaving the output in the file `prog.i`. It does not compile the program.

```
aCC -P prog.C
```

-Uname

`-Uname`

name is the symbol name whose definition is removed from the preprocessor.

This option undefines any name that has initially been defined by the preprocessing stage of compilation.

A *name* can be a definition set by the compiler. This is displayed when you specify the `-v` option. A *name* can also be a definition that you have specified with the `-D` option on the command line.

The `-D` option has lower precedence than the `-U` option. If the same `name` is used in both, the `-U` option and the `-D` option, the `name` is undefined regardless of the order of the options on the command line.

Profiling Code Options

HP compilers provides the following options for profiling your code.

-G

`-G`

At compile time, the `-G` option produces code that counts the number of times each arc in the call graph is traversed. At link-time, when you are building an executable (but not a shared library) `-G` picks up profiled versions of certain system libraries and picks up the `gprof` support library.

Example:

```
aCC -G file.C
```

The above example compiles `file.C` and creates the executable file `a.out` instrumented for use with `gprof`.

See `gprof(1)` manpage for more information.

-p

`-p`

At compile time, the `-p` option produces code that counts the number of times each routine is called. At link-time, when you are building an executable (but not a shared library) `-p` picks up profiled versions of certain system libraries and picks up the prof support library.

Example:

The following example compiles `file.C` and creates the executable file `a.out` instrumented for use with `prof`.

```
aCC -p file.C
```

See the `prof(1)` manpage for more information.

+profilebucketsize

`+profilebucketsize=[16|32]`

This is a link-time option to support `prof` and `gprof` when building an executable, but not a shared library. When `prof` or `gprof` startup code invokes `sprofil`, this option specifies the size in bits of the counters used to record sampled values of the program counter.

The effect of this option can be overridden by setting the environment variable `LD_PROFILEBUCKET_SIZE` when running the instrumented program. This environment variable has no effect when building the instrumented program. Legal values are 16 (the default), and 32.

See `gprof(1)` and `ld(1)` manpages for more details.

Runtime Checking Options

The +check options allow you to check your application for errors at runtime.

+check

```
+check=all|none|bounds|globals|lock|malloc|stack|thread|truncate|uninit
```

The +check=xxx options provide runtime checks to detect many common coding errors in the user program. These options introduce additional instructions for runtime checks that can significantly slow down the user program. By default, a failed check will result in the program aborting at the end of execution at runtime. In most cases, an error message and a stack trace will be emitted to stderr before program termination. The environment variable RTC_NO_ABORT can be set to 0, 1, or 2 to change the behavior of failed runtime checks:

- 0 — A failed runtime check will abort the program immediately after the error message is emitted.
- 1 — The default setting, which will abort the program at the end of execution upon failure.
- 2 — A failed runtime check will not enable the end of execution abort.

The +check options need to be specified at both compile time and link time, since they may require additional libraries to be linked into the user program. If different +check options are specified while compiling different source files, all the specified +check options are needed at link time.

Multiple +check options are interpreted left to right. In case of conflicting options, the one on the right will override an earlier +check option.

NOTE: The +check option is only supported on Integrity servers.

+check=all

The +check=all option enables all runtime checks provided by the compiler, except for +check=truncate, +check=lock, and +check=thread, which must be explicitly specified to enable them. It overrides any +check=xxx options that appear earlier on the command line. The +check=all option is currently equivalent to the following options:

```
+check=bounds:array +check=globals +check=malloc
+check=stack:variables +check=uninit -z
```

The -z option, which is part of +check=all, can be overridden by an explicit -Z option.

+check=none

The +check=none option turns off all runtime checking options. It disables any +check=xxxoptions that appear earlier on the command line.

+check=bounds

The +check=bounds option enables checks for out-of-bounds references to array variables or to buffers through pointer access. The check is performed for each reference to an array element or pointer access. If a check fails, an error message is emitted and the program is aborted.

The +check=bounds option applies only to local and global array variables. It also applies to references to array fields of structs. It does not apply to arrays allocated dynamically using malloc or alloca.

You can specify one of the following +check=bounds suboptions:

- array - Enables check for out-of-bounds references to array variables.
- pointer - Enables check for out-of-bounds references to buffers through pointer access. The buffer could be a heap object, global variable, or local variable. This suboption also checks out-of-bounds access through common libc function calls such as strcpy, strcat, memset,

and so on. The check can create significant run-time performance overhead. See `+check=uninit` and `+check=malloc` for their interaction with this option.

- all - Enables out-of-bounds checks for both arrays and pointers. This is equal to `+check=bounds:array +check=bounds:pointer`.
- none - Disables out-of-bounds checks.

`+check=bounds` (with no suboption) is equal to `+check=bounds:array`. This may change in the future to also include `+check=bounds:pointer`.

When `+check=all` is specified, it enables `+check=bounds:array` only. To enable the pointer out-of-bounds check, you must explicitly specify `+check=bounds:pointer`.

You can combine `+check=bounds:[pointer | all]` with all other `+check` options, except for `+check=globals` (which would be ignored in this case).

Also see the `+check=malloc` and the `+check=stack` options for related runtime checks for heap and stack objects.

Example:

This example uses `+check=bounds:pointer` to find a program bug:

```
 $ cat rttest3.c     1  #include <stdio.h>
      2  #include <memory.h>
      3  #include <stdlib.h>
      4     5  int a[10];
      6  char b[10];
      7  int *ip = &a[0];   // points to global array
      8
      9  int i;
     10
     11  void *foo(int n)
     12  {
     13     return malloc(n * sizeof(int));
     14  }
     15
     16  int main(int argc, char **argv)
     17  {
     18     int j;     // uninitialized variable
     19
     20     int *lp = (int*)foo(10);    // points to heap object
     21
     22     // out of bound if "a.out 10"
     23     if (argc > 1) {
     24        i = atoi(argv[1]);
     25     }
     26
     27     memset(b, 'a', i);
     28
     29     lp[i] = i;
     30
     31     ip[i+1] = i+1;
     32
     33     printf("lp[%d]=%d, ip[%d]=%d, ip[j=%d]=%d\n",
     34            i, lp[i], i+1, ip[i+1], j, ip[j]);
     35
     36     return 0;
     37  }
```

Compiling with `+check=bounds:pointer`:

```
$ cc +check=bounds:pointer rttest3.c

"rttest3.c", line 34: warning #2549-D: variable "j" is used before its
     value is set
```

```
          i, lp[i], i+1, ip[i+1], j, ip[j]);
                                  ^
```

Catch out-of-bounds pointer access through an uninitialized variable (the uninitialized variable can be checked by `+check=uninit`):

```
$ RTC_NO_ABORT=1 a.out  2
Runtime Error: out of bounds buffer pointed by 0x40010320 has 40 bytes
(variable "a"), reading at 0x40010320-19824, 4 bytes ("rttest3.c", line 33)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out](1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x0000000004003920  main + 0x330 at rttest3.c:33 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]
Memory fault(coredump)
```

Check off by one out-of-bounds access:

```
 $ RTC_NO_ABORT=1 a.out  10
Runtime Error: out of bounds buffer pointed by 0x400a1890 has 40 bytes
(allocation stack trace: 0x040035c2, 0x04003612, 0xc0049c42),  writing
at 0x400a1890+40, 4 bytes ("rttest3.c", line 29)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x00000000040037b0  main + 0x1c0 at rttest3.c:29 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]
Runtime Error: out of bounds buffer pointed by 0x40010320 has 40 bytes
(variable "a"), writing at 0x40010320+44, 4 bytes ("rttest3.c", line 31)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x0000000004003810  main + 0x220 at rttest3.c:31 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]

Runtime Error: out of bounds buffer pointed by 0x400a1890 has 40 bytes
(allocation stack trace: 0x040035c2, 0x04003612, 0xc0049c42),  reading
at 0x400a1890+40, 4 bytes ("rttest3.c", line 33)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x00000000040038a0  main + 0x2b0 at rttest3.c:33 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]

Runtime Error: out of bounds buffer pointed by 0x40010320 has 40 bytes
(variable "a"), reading at 0x40010320+44, 4 bytes ("rttest3.c", line 33)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x00000000040038f0  main + 0x300 at rttest3.c:33 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]

Runtime Error: out of bounds buffer pointed by 0x40010320 has 40 bytes
(variable "a"), reading at 0x40010320-19824, 4 bytes ("rttest3.c", line 33)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x0000000004003920  main + 0x330 at rttest3.c:33 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]
Memory fault(coredump)
```

Check off by any number out-of-bounds access:

```
RTC_NO_ABORT=1 a.out  20
$Runtime Error: out of bounds buffer pointed by 0x40010350 has 10 bytes
(variable "b"), writing at 0x40010350+0, 20 bytes ("memset", line 0)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x00000000040089d0  _rtc_oob_check_unknown_bounds + 0x430 at
rtc_bounds.c:480 [./a.out]
(2)  0x60000000c5f52440  libc_mem_common + 0x280 at infrtc.c:3286
[lib/hpux32/librtc.so]
(3)  0x60000000c5f53650  _memset + 0x80 at infrtc.c:3521
[lib/hpux32/librtc.so]
(4)  0x0000000004003760  main + 0x170 at rttest3.c:27 [./a.out]
(5)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]

Runtime Error: out of bounds buffer pointed by 0x400a1890 has 40 bytes
(allocation stack trace: 0x040035c2, 0x04003612, 0xc0049c42),  writing
at 0x400a1890-2054847100, 4 bytes ("rttest3.c", line 29)
(0)  0x0000000004004770  _rtc_raise_fault + 0x560 at rtc_utils.c:164
[./a.out]
(1)  0x0000000004008790  _rtc_oob_check_unknown_bounds + 0x1f0 at
rtc_bounds.c:465 [./a.out]
(2)  0x00000000040037b0  main + 0x1c0 at rttest3.c:29 [./a.out]
(3)  0x60000000c0049c50  main_opd_entry + 0x50 [/usr/lib/hpux32/dld.so]
Memory fault(coredump)
```

+check=globals

The `+check=globals` option enables runtime checks to detect corruption of global variables by introducing and checking "guards" between them, at the time of program exit. Setting environment variable `RTC_ROUTINE_LEVEL_CHECK`will also enable the check whenever a function compiled with this option returns.

For this purpose, the definition of global is extended to be all variables that have static storage duration, including file or namespace scope variables, function scope static variables, and class (or template class) static data members.

+check=lock

The `+check=lock` option enables checking of C and C++ user applications that use pthreads. The option reports violations of locking discipline when appropriate locks are not held while accessing shared data by different threads. The check is based on the lockset method for detecting locking discipline violations, as described in the *Eraser* tool article at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.3256&rep=rep1&type=pdf`.

Note that `+check=all` does not enable `+check=lock`. Also note that since `+check=lock` requires instrumenting each memory access, it can result in a considerable slowdown of the application at runtime. `+check=lock` also increases the memory consumption of the instrumented application.

The check is performed on each memory access. It detects violations in locking discipline for mutual exclusion locks (mutexes) for applications using posix threads. When the locking discipline is violated, it is reported along with the line number and the address for which the violation occurs. Consider the following code example:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
unsigned long things_done=0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
void *thread1(void *arg) {
    pthread_mutex_lock(&mutex1);
    things_done++;
    pthread_mutex_unlock(&mutex1);
    return 0;
}
```

```
void *thread2(void *arg) {
    pthread_mutex_lock(&mutex2);
    things_done++;
    pthread_mutex_unlock(&mutex2);
    return 0;
}

int main(int argc, char *argv[])
{
  pthread_t th1, th2;
  pthread_create(&th1, NULL, thread1, (void*)NULL );
  pthread_mutex_lock(&mutex1);
  things_done++;
  pthread_mutex_unlock(&mutex1);
  pthread_create(&th2, NULL, thread2, (void*)NULL );
  sleep(10);
  return 0;
}

cc  +check=lock simple_race.c -lpthread

./a.out

Runtime Error: locking discipline violation: in file simple_race.c line 16  address 40010658

(0)  0x0000000004072ca0  _rtc_raise_fault + 0x2c0 at rtc_utils.c:382 [./a.out]
(1)  0x0000000004028650  _ZN11DRD_Runtime15HandleMemAccessEybPcjS0_ + 0x590 at lock_check.C:438
[./a.out]
(2)  0x0000000004029840  _ZN11DRD_Runtime17HandleStoreAccessEyPcjS0_ + 0x60 at lock_check.C:145
[./a.out]
(3)  0x000000000401bfa0  __DRD_RegisterStoreAccess__ + 0x160 at lock_check.H:126 [./a.out]
(4)  0x0000000004018780  thread2 + 0xd0 at simple_race.c:16 [./a.out]
(5)  0x60000000c57c3c60  __pthread_bound_body + 0x170
at /ux/core/libs/threadslibs/src/common/pthreads/pthread.c:4512
[/proj/thlo/Compilers/rt/usr/lib/hpux32/libpthread.so.1]

candidate lockset is as follows:
 lock1.c line number:23
incoming lockset is as follows:
 lock1.c line number:13
```

In the above message, the candidate lockset refers to the set of locks that are implied to be associated with the symbol acesss in its previous accesses so far. The incoming lockset refers to the set of locks that are held at the current access of the symbol. When the intersection between the candidate lockset and incoming lockset is empty, the checker reports the locking discipline violation. The candidate lockset and incoming lockset members are specified in terms of the source file and line number pointing to the `pthread_mutex_lock` call associated with that lock. For further details on detecting lock discipline violations, refer to the above-referenced *Eraser* article.

False positives are possible in certain cases, as mentioned in the *Eraser* article. Multiple locks can be used to protect the same shared variable. For example, a linked list can be protected by an overall lock and an individual entry lock. This can result in the tool reporting a false positive. False positives might also be reported as a result of memory getting recycled in certain cases because of deallocations (which the lock checker is not able to detect).

+check=malloc

The `+check=malloc` option enables memory leak and heap corruption checks at runtime. It will cause the user program to abort for writes beyond boundaries of heap objects, `free` or `realloc` calls for a pointer that is not a valid heap object, and out-of-memory conditions. Memory leak information is captured and written out to a log file when the program exits. The name of the logfile is printed out before program termination.

The `+check=malloc` option works by intercepting all heap allocation and deallocation calls. This is done by the use of a debug `malloc` library, `librtc.so`. The option works for programs that use the system `malloc` or for user provided `malloc` routines in a shared library. The `librtc.so` library is also used by the HP WDB debugger to provide heap memory checking features in the debugger. Please refer to the HP WDB debugger documentation for more information about heap memory checking. The `librtc.so` library is shipped as part of the wdb product.

Please install the HP WDB bundled with the compiler or a more recent version of wdb to get full functionality.

The default behavior of the `+check=malloc` option can be changed by users providing their own `rtcconfig` file. The user specified `rtcconfig` file can be in the current directory or in a directory specified by the `GDBRTC_CONFIG` environment variable. The default configuration used by the `+check=malloc` option is:

```
check_bounds=1;check_free=1;scramble_block=1;
abort_on_bounds=1;abort_on_bad_free=1;abort_on_nomem=1;
check_leaks=1;min_leak_size=0;check_heap=0;
frame_count=4;output_dir=.;
```

When `+check=bounds:pointer` is also turned on, it can check freed memory read/write. But the check needs to retain freed memory which is not turned on by default. To turn on the feature, set the following environment variable at runtime:

```
RTC_MALLOC_CONFIG="retain_freed_blocks=1"
```

Or add "retain_freed_blocks=1" to the `rtcconfig` file. When `malloc` failes to allocate specified memory, the runtime system will free the retained freed memory and try to allocate memory.

For a description for the above configuration parameters and the full list of other parameters, please refer to the HP WDB debugger documentation.

+check=stack[:frame|:variables|:none]

The `+check=stack[:frame|:variables|:none]` option enables runtime checks to detect writes outside stack boundaries. Markers are placed before and after the whole stack frame and around some stack variables. On procedure exit, a check is done to see if any marker has been overwritten. If any stack check fails, an error message and stack trace is written to `stderr` and the program is aborted. The stack checks are not performed for an abnormal exit from the procedure (for example, using `longjmp`, `exit`, `abort`, or exception handling).

`+check=stack:frame`

This option enables runtime checks for illegal writes from the current stack frame that overflow into the previous stack frame.

`+check=stack:variables`

This option enables runtime checks for illegal writes to the stack just before or after some variables on the stack. This includes array, `struct/class/union`, and variables whose address is taken. It also includes the overflow check for the stack frame (`+check=stack:frame`). In addition to the above checks, this option causes the whole stack to be initialized to a "poison" value, which can help detect the use of uninitialized variables on the stack.

`+check=stack:none`

This option disables all runtime checks for the stack.

`+check=stack`

The `+check=stack`option without any qualifiers is equivalent to `+check=stack:variables` at optimization levels 0 and 1. It is equivalent to `+check=stack:frame` for optimization level 2 and above.

+check=thread

The `+check=thread` option enables the batch-mode thread-debugging features of HP WDB 5.9 or later, and can detect the following thread-related conditions:

- The thread attempts to acquire a nonrecursive mutex that it currently holds.
- The thread attempts to unlock a mutex or a read-write lock that it has not acquired.
- The thread waits (blocked) on a mutex or read-write lock that is held by a thread with a different scheduling policy.

- Different threads non-concurrently wait on the same condition variable, but with different associated mutexes.
- The threads terminate execution without unlocking the associated mutexes or read-write locks.
- The thread waits on a condition variable for which the associated mutex is not locked.
- The thread terminates execution, and the resources associated with the terminated thread continue to exist in the application because the thread has not been joined or detached.
- The thread uses more than the specified percentage of the stack allocated to the thread.

The `+check=thread` option should only be used with multithreaded programs. It is not enabled by `+check=all`.

Users can change the behavior of the `+check=thread` option by providing their own `rtcconfig` file. The user specified `rtcconfig` file can be in the current directory or in a directory specified by the `GDBRTC_CONFIG` environment variable. The default configuration used by the `+check=thread` option is:

```
thread-check=1;recursive-relock=1;unlock-not-own=1;
mix-sched-policy=1;cv-multiple-mxs=1;cv-wait-no-mx=1;
thread-exit-own-mutex=1;thread-exit-no-join-detach=1;stack-util=80;
num-waiters=0;frame_count=4;output_dir=.;
```

If any thread error condition is detected during the application run, the error log is output to a file in the current working directory. The output file will have the following naming convention:

`<`*`executable_name`*`>.<`*`pid`*`>.threads`

where <pid> is the process id.

+check=truncate[:explicit|:implicit]

The `+check=truncate[:explicit|:implicit]` option enables runtime checks to detect data loss in assignment when integral values are truncated. Data loss occurs if the truncated bits are not all the same as the left most non-truncated bit for signed type, or not all zero for unsigned type.

Programs might contain intentional truncation at runtime, such as when obtaining a hash value from a pointer or integer. To avoid runtime failures on these truncations, you can explicitly mask off the value:

```
ch = (int_val & 0xff);
```

Note that the `+check=all` option does not imply `+check=truncate`. To enable `+check=truncate`, you must explicitly specify it.

`+check=truncate:explicit`

This option turns on runtime checks for truncation on explicit user casts of integral values, such as `(char)int_val`.

`+check=truncate:implicit`

This option turns on runtime checks for truncation on compiler-generated implicit type conversions, such as `ch = int_val;`.

`+check=truncate`

This option turns on runtime checks for both explicit cast and implicit conversion truncation.

+check=uninit

The `+check=uninit` option checks for a use of a stack variable before it is defined. If such a use is detected, an error message is emitted and the program is aborted. The check is done by adding an internal flag variable to track the definition and use of user variables.

If the `+check=bounds:pointer` is on, `+check=uninit` will check pointer access for uninitialized memory read (UMR). To enable checking, the runtime system will initialize the heap objects and stack variables with a special pattern. If the pointer accesses an area containing the specified pattern for the specified length, then it assumes the read is UMR. To minimize UMR false positive, the user may change the special pattern and number of bytes to check by using RTC_UMR environment variable:

```
RTC_UMR=[INIT=0xnn][:CHECK_SIZE=sz]
```

where:

- INIT specifies the char type value used to initialize heap/local objects. The default pattern is 0xDE.
- CHECK_SIZE specifies the minimum number of bytes used to check for UMR. The default number is 2.

Also see the `+Oinitcheck` option to enable compile-time warnings for variables that may be used before they are set.

Standards Related Options

The compiler accepts the following options related to the *ANSI/ISO 9899-1990 Standard* for the *C Programming Language*, the *ANSI/ISO International Standard,* and *ISO/IEC 14882 Standard for the C++ Programming Language*:

-Aa

```
-Aa
```

The `-Aa` option instructs the compiler to use Koenig lookup and strict ANSI for scope rules. This option is equivalent to specifying `-Wc,-koenig_lookup,on` and `-Wc,-ansi_for_scope,on`.

The default is `on` for C++, but off for C. Refer to the `-Ae` option for C-mode description. The standard features enabled by `-Aa` are incompatible with earlier C and C++ features.

-AA

```
-AA
```

The `-AA` option enables the use of the new 2.0 Standard C++ Library, which includes the new standard conforming (templatized) iostream library. It conforms to the ISO C++ standard.

The `–AA` option sets `-Wc,-koenig_lookup,on` and `-Wc,-ansi_for_scope,on`, and is the default C++ compilation mode.

Usage:

The standard features enabled by `-AA` are incompatible with the older Rogue Wave Standard C++ Library 1.2.1 and Tools.h++ 7.0.6. All modules must be consistent in using `-AA`. Mixing modules compiled with `-AA` with ones that are not is not supported.

NOTE:

- This option is not supported in legacy HP C. This option is ignored with warnings in C-mode.
- This option will be removed in a future version of the compiler. Use the equivalent option `+std=c++98` to ensure that your builds do not break in future.

-Aarm

```
-Aarm
```

NOTE: This option was deprecated earlier and is obsolete in this release.

This option enables the Tru64 UNIX C++ Annotated Reference Manual (ARM) dialect. This dialect was the default for Tru64 UNIX C++ compilers through compiler version 5.x. Support of this dialect is intended only to ease porting of existing Tru64 UNIX applications to HP-UX, and not for development of new programs.

For more information on the ARM dialect, refer to the *The Annotated C++ Reference Manual*, (Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-51459-1, 1990).

-AC89

`-AC89`

The `-AC89` option invokes the compiler in ANSI C89 compilation mode. This option, when specified with the `-ext` option, invokes a part of ANSI C99 features.

NOTE: This option will be removed in a future version of the compiler. Use the equivalent option `+std=c89` to ensure that your builds do not break in future.

-AC99

`-AC99`

The `-AC99` option invokes the compiler in ANSI C99 compilation mode with its features. This is the default C compilation mode, and the following commands are equivalent:

```
cc
cc -Ae
cc -AC99
aCC -Ae
aCC -AC99
```

NOTE: This option will be removed in a future version of the compiler. Use the equivalent option `+std=c99` to ensure that your builds do not break in future.

-Ae

`-Ae`

Setting the `-Ae` option invokes aC++ as an ANSI C compiler, with additional support for HP C language extensions.

This option is a synonym for the `-AC99` option.

For C++, if `-Ae` is anywhere on the command line, C-mode will be in effect. The options, `-AA` and `-AP`, are ignored with warnings. If both `-Ae` and `-Aa` are present, C-mode will be in effect and the right most option determines whether extended ANSI (`-Ae`) or strict ANSI (`-Aa`) is in effect. To get strict ANSI, both `-Ae` and `-Aa` option are required.

NOTE: Some code that is a warning in C may be a fatal error in HP aC++.

-Ag++

This option enables GNU C++ dialect compatibility. Not all GNU features are available, for instance, the `asm` mechanism. See also “-fshort-enums ” (page 35).

NOTE: This option will be removed in a future version of the compiler. Use the equivalent option `+std=g++` to ensure that your builds do not break in future.

-Agcc

This option enables GNU C dialect compatibility. Not all GNU features are available, for instance, the `asm` mechanism. See also “-fshort-enums ” (page 35).

NOTE:

- For HP aC++, the `-Ae` option must also be used.
- This option will be removed in a future version of the compiler. Use the equivalent option `+std=gcc` to ensure that your builds do not break in future.

-AOa and -AOe

`-AOa`

`-AOe`

See the following C mode options:

- "-Aa" (page 82)
- "-Ae" (page 83)

In addition to specifying the ANSI C language dialect, allows the optimizer to aggressively exploit the assumption that the source code conforms to the *ANSI programming language C standard ISO 9899:1990*.

At present, the effect is to make `+Otype_safety=ansi` the default. As new independently-controllable optimizations are developed that depend on the ANSI C standard, the flags that enable those optimizations may also become the default under `-AOa` or `-AOe`.

-AP

`-AP`

NOTE: To enable future runtime library versions, this option was deprecated earlier and is obsolete in this release. If there is a build using this option, migrate your source to comply with the C++ ANSI standard.

The `-AP` option turns off `-AA` mode and uses the older C++ runtime libraries.

NOTE: This option is not supported in legacy HP C. This option is ignored with warnings in C-mode.

-Ax

The `-Ax` option turns on support for several core language features introduced by the recently published C++11 standard. The `-Ax` option is available only in C++ compilation mode and is binary compatible with the `-AA` compilation mode. See the *HP aC++/HP ANSI C Release Notes* for a description of extensions supported.

NOTE: This option will be removed in a future version of the compiler. Use the equivalent option `+std=c++11` to ensure that your build does not break in future.

+legacy_cpp

`+legacy_cpp`

The `+legacy_cpp` option enables the use of `cpp.ansi` ANSI C preprocessor. This option is available in C-mode only.

NOTE: This option is not normally needed and may be deprecated in future.

+legacy_v5

`+legacy_v5`

This option enables the use of the A.05.60 compiler. The default compiler is the A.06.00 compiler.

+std=c89|c99|c++98|c++11|gcc|g++|gnu

+std=c89: This option invokes the compiler in ANSI C89 compilation mode. This option when specified with the -ext option, it invokes a part of ANSI C99 feature. This is equivalent to the '–AC89' option.

+std=c99: This option invokes the compiler in ANSI C99 compilation mode with its features. This is the default C compilation mode. This is equivalent to the '–AC99' option.

+std=c++98: This option invokes the compiler in ISO C++98 standard mode. This is the default C++ compilation mode and this is equivalent to the '–AA' option.

+std=c++11: This option turns on support for several core language features introduced by the ISO C++11 language standard. It is available only in C++ compilation mode and is binary compatible with the '+std=c++98' ('–AA') compilation mode.

+std=gcc: This option enables GNU C dialect compatibility. This option is equivalent to '–Agcc' option.

+std=g++: This option enables GNU C++ dialect compatibility. This option is equivalent to '–Ag++' option.

+std=gnu: This command line option is also used to enable gnu dialects. It switches between '+std=gcc ' or '+std=g++' compilation, depending on whether the compilation mode is C or C++ respectively.

+stl=rw|none

+stl=rw: This option is used to specify RogueWave STL 2.0 implementation. This option is equivalent to '–AA' option. It includes C++98 compliant standard template library. This is the default STL. This option causes standard C++ header files to be picked up from the directory '/opt/aCC/include_std' and is linked with libstd_v2.so.

+stl=none: By eliminating references to the standard header files and libraries bundled with HP C++ compiler, this option allows experienced users to have full control over the header files and libraries used in compilation and linking of their applications. This is equivalent to '+nostl' option.

+tru64

`+tru64`

This option causes return types of unprototyped functions to be treated as `long`, instead of `int`, matching Tru64 C behavior. This can prevent segfaults in `+DD64` mode, resulting from pointer truncation, for instance:

```
long *a;
long sub() {
   a = malloc(sizeof(long));    /* no prototype! */
   *a = 1234;                   /* segfault if +DD64 and no +tru64 */
   return *a;
}
```

A preferable solution is to provide the appropriate function prototypes.

NOTE: This option is applicable to C language only.

-Wc,-ansi_for_scope,[on|off]

`-Wc,-ansi_for_scope,[on|off]`

The `-Wc,-ansi_for_scope` is option enables or disables the standard scoping rules for `init` declarations in for statements; the scope of the declaration then ends with the scope of the loop body. By default, the option is disabled.

Examples:

In the following example, if the option is not enabled (the current default), the scope of `k` extends to the end of the body of `main` and statement (1) is valid (and will return zero). With the option enabled, `k` is no longer in scope and (1) is an error.

```
#include <stdio.h>

int main() {
   for (int k = 0; k!=100; ++k) {
      printf("%d\n", k);
   }
   return 100-k; // (1)
}
```

In the next example, with the option disabled, the code is illegal, because it redefines `k` in (2) when a previous definition (1) is considered to have occurred in the same scope.

With the option enabled (`-Wc,-ansi_for_scope,on`), the definition in (1) is no longer in scope at (2) and thus the definition in (2) is legal.

```
int main() {
   int sum = 0;
   for (int k = 0; k!=100; ++k)   // (1)
      sum += k;
   for (int k = 100; k!= 0; ++k)  // (2)
      sum += k;
}
```

-Wc,-koenig_lookup,[on|off]

`-Wc,-koenig_lookup,[on|off]`

The `-WC,-koenig_lookup` option enables or disables standard argument-dependent lookup rules (also known as Koenig lookup). It causes functions to be looked up in the namespaces and classes associated with the types of the function-call argument. By default, the option is enabled.

Example:

In the following example, if the option is not enabled, the call in `main` does not consider declaration (1) and selects (2). With the option enabled, both declarations are seen, and in this case overload resolution will select (1).

```
namespace N {
   struct S {};
   void f(S const&, int);  // (1)
}

void f(N::S const&, long); // (2)

int main() {
   N::S x;
   f(x, 1);
}
```

Subprocesses of the Compiler

These options allow you to substitute your own processes in place of the default HP aC++ subprocesses, or pass options to HP aC++ subprocesses.

-tx,name

`-tx,name`

The `-tx,name` option substitutes or inserts subprocess `x`, using `name`.

The parameter, *x*, is one or more identifiers indicating the subprocess or subprocesses. The value of *x* can one or more of the following:

Table 8 Identifiers

x	Description
a	Assembler (standard suffix is `as`)
c	Compiler (standard suffix is `ctcom/ecom`)
C	Same as `c`
f	Filter tool (standard suffix is `c++filt`)
l	Linker (standard suffix is `ld`)
p	Preprocessor (standard suffix is `cpp.ansi`). `-tp` must be used before any `-Wp` options can be passed to `cpp.ansi`. To enable the external preprocessor, use: `-tp,/opt/langtools/lbin/cpp.ansi`.
u	Stand-alone code generator (standard suffix is `u2comp`)
x	All subprocesses

The `-tx,name` option works in two modes:

1. If `x` is a single identifier, `name` represents the full path name of the new subprocess.
2. If `x` is a set of identifiers, `name` represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses.

For example, the following command invokes the assembler `/users/sjs/myasmb` instead of the default assembler `/usr/ccs/bin/as` to assemble and link `file.s`.

```
aCC -ta,/users/sjs/myasmb file.s
```

More Examples of -t

Following are some examples of `-t` option:

- **Substituting for C++ file:**

  The following example compiles `file.C` and specifies that `/new/bin/c++filt` should be used instead of the default `/opt/aCC/bin/c++filt`.

```
aCC -tf,/new/bin/c++filt file.C
```

- **Substituting for ecom:**

  The following example compiles `file.C` and specifies that `/users/proj/ecom` should be used instead of the default `/opt/aCC/lbin/ecom`.

  ```
  aCC -tC,/users/proj/ecom file.C
  ```

- **Substituting for all Subprocesses:**

  The following example compiles `file.C` and specifies that the characters `/new/aCC` should be used as a prefix to all the subprocesses of HP aC++. For example, `/new/aCC/ecom` runs instead of the default `/opt/aCC/lbin/ecom`.

  ```
  aCC -tx,/new/aCC file.C
  ```

-Wx,args

`-Wx,arg1[,arg2,..,argn]`

The `-Wx,args` option passes the arguments `arg1` through `argn` to the subprocess `x` of the compilation.

Each argument, `arg1`, `arg2`, through `argn` takes the form:

`-argoption[,argvalue]`

where:

- `argoption` is the name of an option recognized by the subprocess.
- `argvalue` is a separate argument to `argoption`, where necessary.

The parameter, `x`, is one or more identifiers indicating a subprocess or subprocesses. The value of `x` can be one or more of the following:

Table 9 Identifiers

x	Description
a	Assembler (standard suffix is `as`)
c	Compiler (standard suffix is `ecom`)
C	Same as `c`
f	Filter tool (standard suffix is `c++filt`)
l	Linker (standard suffix is `ld`)
p	Preprocessor (standard suffix is `cpp.ansi`). `-tp` must be used before any `-Wp` options can be passed to `cpp.ansi`. To enable the external preprocessor, use: `-tp,/opt/langtools/lbin/cpp.ansi`.
u	Stand-alone code generator (standard suffix is `u2comp`)
x	All subprocesses

Example:

The following example compiles `file.C` and passes the option `-v` to the linker.

```
aCC -Wl,-v file.C
```

Passing Options to the Linker with -W

The following example links `file.o` and passes the option `-a archive` to the linker, indicating that the archive version of the math library (indicated by `-lm`) and all other driver-supplied libraries should be used rather than the default shared library:

```
aCC file.o -Wl,-a,archive -lm
```

Passing Multiple Options to the Linker with -W

The following example links `file.o` and passes the options `-a` , `archive`, `-m`, and `-v` to the linker:

```
aCC -Wl,-a,archive,-m,-v file.o -lm
```

This case is similar to the previous example, with additional options. `-m` indicates that a load map should be produced. The `-v` option requests verbose messages from the linker.

Symbol Binding Options

The following `-B` options are recognized by the compiler to specify whether references to global symbols may be resolved to symbols defined in the current load module, or whether they must be assumed to be potentially resolved to symbols defined in another load module.

A global symbol is one that is visible by name across translation unit boundaries. A static symbol is one that is visible by name only within a single translation unit but is not associated with a particular procedure activation. A locally defined symbol is a global or static symbol with a definition in the translation unit from which it is being referenced.

-Bdefault

```
-Bdefault
```

Global symbols are assigned the default export class. These symbols may be imported or exported outside of the current load module. The compiler will access tentative and undefined symbols through the linkage table. Any symbol that is not assigned to another export class through use of another `-B` option will have the default export class. The `-Bdefault` option may also be used on a per symbol basis to specify exceptions to global `-Bprotected`, `-Bhidden`, and `-Bextern` options.

Usage:

```
-Bdefault=symbol[,symbol...]
```

The named symbols are assigned the default export class.

```
-Bdefault:filename
```

The file indicated by `filename` contains a list of symbols, separated by spaces or newlines. These symbols are assigned the default export class.

-Bextern

```
-Bextern
```

The specified symbols, or all undefined symbols if no list is provided, are assigned to default export class. Additionally, the compiler will inline the import stub for calls to these symbols. No compile time binding of these symbols will be done. All references to these symbols will be through the linkage table, so an unnecessary performance penalty will occur if `-Bextern` is applied to a listed symbol that is resolved in the same load module.

Usage:

```
-Bextern=symbol[,symbol...]
```

The named symbols, or all symbols if no list is provided, are assigned the default export class. Use of list form overrides the default binding of locally defined symbols.

`-Bextern:filename`

The file indicated by `filename` is expected to contain a list of symbols, separated by spaces or newlines. These symbols are assigned the default export class.

-Bhidden

`-Bhidden`

The specified symbols, or all symbols if no symbols are specified, are assigned the hidden export class. The hidden export class is similar to the protected export class. In addition, hidden symbols will not be exported outside the current load module. The linker may eliminate them from a shared library, but in an executable, they remain accessible to the debugger unless `+Oprocelim` is also specified.

When used with no symbol list, `-Bhidden` implies `-Wl,-aarchive_shared`, causing the linker to prefer an archive library over a shared library if one is available. This can be overridden by following the `-Bhidden` option with a subsequent `-Wl,-a` option.

Usage:

`-Bhidden=symbol[,symbol...]`

The named symbols, or all symbols if no symbols are specified, are assigned the hidden export class.

`-Bhidden:filename`

The file indicated by `filename` is expected to contain a list of symbols separated by spaces or newlines. These symbols are assigned the hidden export class.

-Bhidden_def

`-Bhidden_def`

This option is the same as `-Bhidden`, but only locally defined (non-tentative) symbols, without `__declspec(dllexport)`, are assigned the hidden export class.

-Bprotected

`-Bprotected[=symbol[,symbol...]]`

The specified symbols, or all symbols if no symbols are specified, are assigned the protected export class. That means these symbols will not be preempted by symbols from other load modules, so the compiler may bypass the linkage table for both code and data references and bind them to locally defined code and data symbols.

When used with no symbol list, `-Bprotected` implies `-Wl,-aarchive_shared`, causing the linker to prefer an archive library over a shared library, if one is available. This can be overridden by following the `-Bprotected` option with a subsequent `-Wl,-a` option.

Usage:

`-Bprotected:filename`

The file indicated by `filename` contains a list of symbols, separated by spaces or newlines. These symbols are assigned the protected export class.

-Bprotected_data

`-Bprotected_data`

The `-Bprotected_data` option marks only data symbols as having the protected export class.

-Bprotected_def

`-Bprotected_def`

The `-Bprotected_def` option is the same as `-Bprotected` but only locally defined (non-tentative) symbols are assigned the protected export class.

-Bsymbolic

`-Bsymbolic`

The `-Bsymbolic` option assigns protected export class to all symbols. This is equivalent to `-Bprotected` with no symbol list.

NOTE: This option is deprecated as of version A.06.05 and if used, it issues a warning that `-Bprotected_def`is almost always what should be used in its place.

Template Options

By using a template option on the aCC command line, you can:

- Close a library or set of link units, to satisfy all unsatisfied instantiations without creating duplicate instantiations.
- Specify what templates to instantiate for a given translation unit.
- Name and use template files in the same way as for the cfront based HP C++ compiler.
- Request verbose information about template processing.

NOTE: All template options on an `aCC` command line apply to every file on the command line. If you specify more than one incompatible option on a command line, only the last option takes effect.

+[no]dep_name

The `+[no]dep_name` option enforces strict dependent-name lookup rules in templates. The default is `+dep_name`.

+inst_compiletime

`+inst_compiletime`

The `+inst_compiletime` option causes the compiler to use the compile time (CTTI) instantiation mechanism to instantiate templates. This occurs for every template used or explicitly instantiated in this translation unit and for which a definition exists in the translation unit. This is the default.

NOTE: This option is supported in C++ only and ignored in C-mode.

+inst_directed

+inst_directed

The `+inst_directed` option indicates to the compiler that no templates are to be instantiated (except explicit instantiations). If you are using only explicit instantiation, specify `+inst_directed`. The following example compiles `file.C` with the resulting object file containing no template instantiations, except for any explicit instantiations coded in your source file.

```
aCC +inst_directed prog.C
```

See Chapter 5: “Using HP aC++ Templates” (page 132) for more information.

NOTE: This option is supported in C++ only and ignored in C-mode.

+inst_implicit_include

`+inst_implicit_include`

The `+inst_implicit_include` option specifies that the compiler use a process similar to that of the `cfront` source rule for locating template definition files. For the `cfront` based HP C++ compiler, if you are using default instantiation (that is, if you are not using a map file), you must have a template definition file for each template declaration file, and these must have the same file name prefix.

This restriction does not apply in HP aC++. Therefore, if your code was written for HP C++ and you wish to follow this rule when compiling with HP aC++, you need to specify the `+inst_implicit_include` option.

This option is strongly discouraged and the sources should be modified to conform to the standard.

Example:

```
aCC +inst_implicit_include prog.C
```

If `prog.C` includes a template declaration file named `template.h`, the compiler assumes a template definition file name determined by the `+inst_include_suffixes` option.

See Chapter 5: "Using HP aC++ Templates" (page 132) for more information.

NOTE: This option is supported in C++ only and ignored in C-mode.

+inst_include_suffixes

```
+inst_include_suffixes "list"
```

The `+inst_include_suffixes` option specifies the file name extensions that the compiler uses to locate template definition files. This option must be used with the `+inst_implicit_include` option.

`list` is a set of space separated file extensions or suffixes, enclosed in quotes, that template definition files can have.

The default extensions in order of precedence are:

- `.c`
- `.C`
- `.cxx`
- `.CXX`
- `.cc`
- `.CC`
- `.cpp`

User-specified extensions must begin with a dot and must not exceed four characters in total. Any extension that does not follow these rules causes a warning and is ignored.

These restrictions do not apply in HP aC++. Therefore, if your code was written for HP C++ and you wish to follow the `cfront`-based HP C++ template definition file naming conventions when compiling with HP aC++, you need to specify the `+inst_include_suffixes` option.

The following example specifies that template definition files can have extensions of `.c` or `.C`:

```
+inst_include_suffixes ".c .C"
```

The `+inst_include_suffixes` option is equivalent to the HP C++ `-ptS` option.

See Chapter 5: "Using HP aC++ Templates" (page 132) for more information.

NOTE: This option is supported in C++ only and ignored in C-mode.

Trigraph Processing Suppression Option

The `-notrigraph`option suppresses trigraph processing.

-notrigraph

The`-notrigraph` option inhibits the processing of trigraphs. In previous versions, [LINEBREAK]-notrigraph caused the legacy preprocessor to be invoked. Though this ignored trigraphs, trigraphs were still interpreted by the compiler in the preprocessed source. The -notrigraph option no longer invokes the legacy preprocessor, and also suppresses trigraphs from being interpreted.

This option is not recommended. The proper portable solution is to quote the `"?"` as `"\?"`.

Verbose Compile and Link Information

Use the following options to obtain additional information about:

- The HP compiler actions while compiling or linking your program.
- The subprocesses executed for a given command line, without running the compiler.
- The current compiler and linker version numbers.
- The Execution time.

-dumpversion

```
-dumpversion
```

The `+dumpversion` option displays the simple version number of the compiler, such as A.06.25.

Compare with the `-V` option, which displays more verbose product version information.

+dryrun

```
+dryrun
```

The `+dryrun` option generates subprocess information for a given command line without running the subprocesses. It is useful in the development process to obtain command lines of compiler subprocesses in order to run the commands manually or to use them with other tools.

Example:

The following command line gives the same kind of information as the `-v` option, but without running the subprocesses.

```
aCC +dryrun app.C
```

+O[no]info

```
+O[no]info
```

The `+Oinfo` option displays informational messages about the optimization process. This option may be helpful in understanding what optimizations are occurring. You can use the option at levels 0-4. The default is `+Onoinfo` at levels 0-4.

+wsecurity

```
+wsecurity[={1|2|3|4}]
```

The `+wsecurity` option can take an argument to control how verbosely the security messages are emitted. The default level is 2.

+time

```
+time
```

The `+time` option generates timing information for compiler subprocesses. For each subprocess, estimated time is generated in seconds for user processes, system calls, and total processing time. This option is useful in the development process, for example, when tuning an application’s compile-time performance.

Examples:

- The `aCC +time app.C` command generates the following information:

  ```
  process: compiler         0.94/u     0.65/s      4.35/r
  process: ld             0.37/u     0.76/s     3.02/r
  ```

- The `aCC -v +time app.C` command generates the following information:

  ```
  /opt/aCC/lbin/ctcom -inst compiletime -diags 523 -D __hppa -D __hpux
     -D __unix -D __hp9000s800 -D __STDCPP__ -D __hp9000s700 -D _PA_RISC1_1
     -I /opt/aCC/include -I /opt/aCC/include/iostream -I /usr -I
     /usr/include -I /usr/include -inline_power 0 app.C

  file name: app.C
  file size: app.o 444 + 16 + 1 = 461
  process                  user    sys   real
  ------------------------------------------
  process: compiler        0.93   0.13   1.17
  ------------------------------------------
  line numbers: app.C 7
  lines/minute: app.C 396

  LPATH=/usr/lib:/usr/lib/hpux32/pa1.1 :/usr/lib:/opt/langtools/lib:/usr/lib
  /opt/aCC/lbin/ld -o a.out /opt/aCC/lib/crt0.o -u ___exit -u main
    -L /opt/aCC/lib /opt/aCC/lib/cpprt0.o app.o -lstd -lstream -lCsup -lm
    /usr/lib/hpux32/libcl.a -lc /usr/lib/hpux32/libdld.so >/usr/tmp/AAAa28149 2>&1

  file size: a.out 42475 + 1676 + 152 = 44303
  process                  user    sys   real
  ------------------------------------------
  process: ld              0.35   0.24   0.82
  ------------------------------------------
  total link time(user+sys):   0.59
   removing /usr/tmp/AAAa28149
   removing app.o
  ```

-v

`-v`

The `-v` option enables verbose mode, sending a step-by-step description of the compilation process to `stderr`. This option is especially useful for debugging or for learning the appropriate commands for processing a C++ file.

Example:

The `aCC -v file.C` command compiles `file.C` and gives information about the process of compiling.

```
/opt/aCC/lbin/ctcom -inst compiletime -diags 523 -D __hppa -D __hpux
-D __unix -D __hp9000s800 -D __STDCPP__ -D __hp9000s700 -D _PA_RISC1_1
-I /opt/aCC/include -I /opt/aCC/include/iostream -I /usr -I
/usr/ include
   -I /usr/include -inline_power 0 app.C
LPATH=/usr/lib:/usr/lib/hpux32/pa1.1
      :/usr/lib:/opt/langtools/lib:/usr/lib
/opt/aCC/lbin/ld -o a.out /opt/aCC/lib/crt0.o -u ___exit -u main
  -L /opt/aCC/lib /opt/aCC/lib/cpprt0.o app.o -lstd -lstream -lCsup
  -lm /usr/lib/hpux32/libcl.a -lc /usr/lib/hpux32/libdld.so >/usr/tmp/AAAa28149 2>&1
  removing /usr/tmp/AAAa28149
```

-V

`-V`

The `-V` option displays the version numbers of the current compiler and linker (if the linker is executed). Use this option whenever you need to know the current compiler and linker version numbers.

Example:

```
aCC -V app.C
```

```
aCC: HP aC++/ANSI C B3910B A.06.00 [Aug 25 2004]
ld: 92453-07 linker ld HP Itanium(R) B.12.24 PBO 040820 (IPF/IPF)
```

Concatenating Options

You can concatenate some options to the `aCC` command under a single prefix. The longest substring that matches an option is used. Only the last option can take an argument. You can concatenate option arguments with their options if the resulting string does not match a longer option.

Examples:

Suppose you want to compile `my_file.C` using the options `-v` and `-g1`. Below are equivalent command lines you can use:

- `aCC my_file.C -v -g1`
- `aCC my_file.C -vg1`
- `aCC my_file.C -vg1`
- `aCC -vg1 my_file.C`

3 Pragma Directives and Attributes

A pragma directive is an instruction to the compiler. You use a `#pragma` directive to control the actions of the compiler in a particular portion of a translation unit without affecting the translation unit as a whole.

Put pragmas in your C++ source code where you want them to take effect. Unless otherwise specified, a pragma is in effect from the point where it is included until the end of the translation unit or until another pragma changes its status.

This chapter discusses the following pragma directives:

Initialization and Termination Pragmas

This section describes the `INIT` and `FINI` pragmas. These pragmas allow the user to set up functions which are called when a load module (a shared library or executable) is loaded (initializer) or unloaded (terminator).

For example, when a program begins execution, its initializers get called before any other user code gets called. This allows some set up work to take place. In addition, when the user's program ends, the terminators can do some clean up. When a shared library is loaded or unloaded, its initializers and terminators are also executed at the appropriate time.

INIT

`#pragma INIT "`*`string`*`"`

Use `#pragma INIT` to specify an initialization function. The function takes no arguments and returns nothing. The function specified by the `INIT` pragma is called before the program starts or when a shared library is loaded. For example,

```
#pragma INIT "my_init"
void my_init() { ... do some initializations ... }
```

FINI

`#pragma FINI "`*`string`*`"`

Use `#pragma FINI` to specify a termination function. The function specified by the `FINI` pragma is called after the program terminates by either calling the `libc exit` function, returning from the `main` or `_start` functions, or when the shared library, which contains the `FINI` is unloaded from memory. Like the function called by the `INIT` pragma, the termination function takes no arguments and returns nothing. For example,

```
#pragma FINI "my_fini"
void my_fini() { ... do some clean up ... }
```

Copyright Notice and Identification Pragmas

The following pragmas can be used to insert strings in code.

COPYRIGHT

```
#pragma COPYRIGHT "string"
```

`string` is the set of characters included in the copyright message in the object file.

The `COPYRIGHT` pragma specifies a string to include in the copyright message and puts the copyright message into the object file.

If no date is specified (using pragma `COPYRIGHT_DATE`), the current year is used in the copyright message. For example, assuming the year is 1999, the directive `#pragma COPYRIGHT "Acme Software"` places the following string in the object code:

```
(C) Copyright Acme Software, 1999. All rights reserved. No part of this
program may be photocopied, reproduced, or transmitted without prior
written consent of Acme Software.
```

The following pragmas

```
#pragma COPYRIGHT_DATE "1990-1999"
```

```
#pragma COPYRIGHT "Brand X Software"
```

place the following string in the object code:

```
(C) Copyright Brand X Software, 1990-1999. All rights reserved. No part
of this program may be photocopied, reproduced, or transmitted without
prior written consent of Brand X Software.
```

COPYRIGHT_DATE

```
#pragma COPYRIGHT_DATE "string"
```

`string` is a date string used by the `COPYRIGHT` pragma.

This pragma specifies a date string to be included in the copyright message.

Use the `COPYRIGHT` pragma to put the copyright message into the object file.

For example, `#pragma COPYRIGHT_DATE "1988-1992"` places the string `"1988-1992"` in the copyright message.

LOCALITY

```
#pragma LOCALITY "string"
```

`string` specifies a name to be used for a code section.

The `LOCALITY` pragma specifies a name to be associated with the code written to a relocatable object module. The string is forced to be uppercase in C.

All code following the `LOCALITY` pragma is associated with the name specified in string. Code that is not headed by a `LOCALITY` pragma is associated with the name `.text`.

The smallest scope of a unique `LOCALITY` pragma is a function.

For example, the directive,

```
#pragma LOCALITY "MINE"
```

builds the name `.text.MINE` and associates all code following this pragma with this name, unless another `LOCALITY` pragma is encountered.

LOCALITY_ALL

```
#pragma LOCALITY_ALL string
```

The `LOCALITY_ALL` pragma specifies a name to be associated with the linker procedures and global variables that should be grouped together at program binding or load time.

These are written to a relocatable object module. All procedures and global variables following the `LOCALITY_ALL` pragma are associated with the name specified in the string.

VERSIONID

```
#pragma VERSIONID "string"
```

`string` is a string of characters that HP aC++ places in the object file.

The `VERSIONID` pragma specifies a version string to be associated with a particular piece of code. The string is placed into the object file produced when the code is compiled.

For example, the directive

```
#pragma VERSIONID "Software Product, Version 12345.A.01.05"
```

places the characters `Software Product, Version 12345.A.01.05` into the object file.

Data Alignment Pragmas

This section discusses the data alignment pragmas and their various arguments available on HP-UX systems to control alignment across platforms.

ALIGN

```
#pragma align N
```

`N` is a number raised to the power of 2.

HP aC++ supports user-specified alignment for global data. The pragma takes effect on next declaration. If the align pragma declaration is not in the global scope or if it is not a data declaration, the compiler displays a warning message. If the specified alignment is less than the original alignment of data, a warning message is displayed, and the pragma is ignored. Note that for C code you must initialize the variables, otherwise the compiler will generate a warning.

```
#pragma align 2
char c;            // "c" is at least aligned on 2 byte boundary.

#pragma align 64
int i, a[10];     // "i" and array "a" are at least aligned 64 byte boundary.
                  // the size of "a" is still 10*sizeof(int)
```

PACK

```
#pragma PACK [n]|[push|pop]|[,<name>][,n]|show]
```

`n` can be `1`, `2`, `4`, `8`, or `16` bytes. If *`n`* is not specified, maximum alignment is set to the default value.

This file-scoped pragma allows you to specify the maximum alignment of class fields. The alignment of the whole class is then computed as usual, to the alignment of the most aligned field in the class.

NOTE: The result of applying `#pragma pack n` to constructs other than class definitions (including struct definitions) is undefined and not supported. For example:

```
#pragma pack 1
int global_var; // Undefined behavior: not a class definition
void foo() {    // Also undefined
}
```

Example:

```
struct S1 {
   char c1;     // Offset 0, 3 bytes padding
   int i;       // Offset 4, no padding
   char c2;     // Offset 8, 3 bytes padding
};              // sizeof(S1)==12, alignment 4

#pragma pack 1

struct S2 {
   char c1;     // Offset 0, no padding
   int i;       // Offset 1, no padding
   char c2;     // Offset 5, no padding
};              // sizeof(S2)==6, alignment 1

// S3 and S4 show that the pragma does not affect class fields
// unless the class itself was defined under the pragma.
struct S3 {
   char c1;     // Offset 0, no padding
   S1 s;        // Offset 1, no padding
   char c2;     // Offset 13, nopadding
};              // sizeof(S3)==14, alignment 1

struct S4 {
   char c1;     // Offset 0, no padding
   S2 s;        // Offset 1, no padding
   char c2;     // Offset 7, no padding
};              // sizeof(S4)==8, alignment 1

#pragma pack

struct S5 {     // Same as S1
   char c1;     // Offset 0, 3 bytes padding
   int i;       // Offset 4, no padding
   char c2;     // Offset 8, 3 bytes padding
};              // sizeof(S5)==12, alignment 4

#pragma pack (push, my_new_align, 1)

struct S6 {  // Same as S2
   char c1;  // Offset 0, no padding
   int i;    // Offset 1, no padding
   char c2;  // Offset 5, no padding
};           // sizeof(S6)==6, alignment 1

#pragma pack 2
#pragma pack show  // compiler diagnostic
                   // that shows current
                   // pragma pack setting

struct S7 {
   char c1; // Offset 0, 1 byte padding
   int i;   // Offset 2, no padding
   char c2; // Offset 6, 1 byte padding
};           // sizeof(S7)==8, alignment 2

#pragma pack (pop, my_new_align)
   struct S8 {  // Same as S1
   char c1;  // Offset 0, 3 bytes padding
   int i;    // Offset 4, no padding
   char c2;  // Offset 8, 3 bytes padding
};           // sizeof(S8)==12, alignment 4
```

The pack pragma may be useful when porting code between different architectures where data type alignment and storage differences are of concern. Refer to the following examples:

Basic Example

The following example illustrates the pack pragma and shows that it has no effect on class fields unless the class itself was defined under the pragma:

```
struct S1 {
   char c1;      // Offset 0, 3 bytes padding
   int i;        // Offset 4, no padding
   char c2;      // Offset 8, 3 bytes padding
};               // sizeof(S1)==12, alignment 4

#pragma pack 1

struct S2 {
   char c1;      // Offset 0, no padding
   int i;        // Offset 1, no padding
   char c2;      // Offset 5, no padding
};               // sizeof(S2)==6, alignment 1

// S3 and S4 show that the pragma does not affect class fields
// unless the class itself was defined under the pragma.
struct S3 {
   char c1;      // Offset 0, 3 bytes padding
   S1 s;         // Offset 4, no padding
   char c2;      // Offset 16, 3 bytes padding
};               // sizeof(S3)==20, alignment 4

struct S4 {
   char c1;      // Offset 0, no padding
   S2 s;         // Offset 1, no padding
   char c2;      // Offset 7, no padding
};               // sizeof(S4)==8, alignment 1

#pragma pack

struct S5 {      // Same as S1
   char c1;      // Offset 0, 3 bytes padding
   int i;        // Offset 4, no padding
   char c2;      // Offset 8, 3 bytes padding
};               // sizeof(S5)==12, alignment 4
```

Template Example

If the pragma is applied to a class template, every instantiation of that class is influenced by the pragma value in effect when the template was defined. For example:

```
#pragma pack 1

template<class T>
struct ST1 {
   char c1;
   T x;
   char c2;
};

#pragma pack

ST1<int> obj;            // Same layout as S2 in the prior example

template <>              // Explicit specialization
struct ST1<void> {
   char c1;
   char c2;
};                       // Undefined (unsupported) behavior
                         // ST1 was defined under a #pragma pack 1 directive
```

NOTE: The alignment of specializations and partial specializations of templates is undefined and unsupported if either the primary template or the specialization is under the influence of a `#pragma pack` directive.

Handling Unaligned Data

Direct access to unaligned class fields is handled automatically by HP aC++. However, this results in slower access times than for aligned data. Indirect access (through pointers and references) to unaligned class fields is also handled automatically.

If you take the address of a data field and assign it to a pointer, it is not handled automatically and is likely to result in premature termination of the program if not handled appropriately.

Example:

```
#include <stdio.h>
#pragma pack 1
struct S1 {
   char c1;
   int i;
   char c2;
};
#pragma pack
int main() {
 S1 s;
 S1 *p = &s;
 printf("%d\n", s.i);      // OK
 printf("%d\n", p->i);     // OK
 int *ip = &p->i;          // Undefined behavior
                           // Likely Abort unless compiled with +u1
                           // The address of a reference (*ip) is
                           // assigned to an int pointer.
 printf("%d\n", *ip);
}
```

To enable indirect access to unaligned data that has been assigned to another type, use the link in the library, `-lunalign` and arm the appropriate signal handler with a call to allow_`unaligned_data_access`. This causes every signal generated due to unaligned access to be intercepted and handled as expected. It also creates significant run-time overhead for every access to unaligned data, but does not impact access to aligned data.

Implicit Access to Unaligned Data

Calls to non-static member functions require that an implicit `this` pointer be passed to these functions, which can then indirectly access data through this implicit parameter. If such an access is to unaligned data, the situation in the prior example occurs.

Furthermore, virtual function calls often require indirect access to a hidden field of a class that could be unaligned under the influence of the `#pragma pack` directive.

If you are passing the address of a field to other code, consider the following example. Unless compiled with `-DRECOVER` on the command line and linked with `-lunalign`, the following example is likely to prematurely terminate with a bus error:

```
#include <stdio.h>
#ifdef RECOVER
extern "C" void allow_unaligned_data_access();
#endif

#pragma pack 1

struct PS1 {
   PS1();
   ~PS1();
private:
   char c;
   int a;
};

#pragma pack

PS1::PS1(): a(1) {    // There appears to be no pointer, but there
                      // is an unaligned access, possibly through "this."
   printf("In constructor.\n");
}

PS1::~PS1() {
   a = 0;            // Misaligned access, possibly though "this"
   printf("In destructor.\n");
}

int main() {
#if defined(RECOVER)
   allow_unaligned_data_access();
#endif
   PS1 s;
}
```

UNALIGN

`#pragma unalign [1|2|4|8|16]`

`typedef T1 T2;`

`T1` and `T2` have the same size and layout, but with specified alignment requirements.

HP aCC supports misaligned data access using the `unalign` pragma. The `unalign` pragma can be applied on `typedef` to define a type with special alignment. The `unalign` pragma takes effect only on next declaration.

If the `unalign` pragma declaration is not in the global scope or if it is not a `typedef`, compiler displays a warning message. If the specified alignment is greater than the original alignment of the declaration, then an error message is displayed, and the pragma is ignored.

Example:

```
#pragma unalign 1
typedef int ua_int;            // ua_int is of int type with 1 byte alignment
typedef ua_int *ua_intPtr;     // this typedef is not affected by the above
                               // unalign pragma. It defines a pointer type
                               // which points to 1 byte aligned data
```

The interaction between `pack` and `unalign` pragmas is as follows:

```
#pragma pack 1
struct S {
   char c;
   int i;
};
#pragma pack 0

S s;
ua_int *ua_ip = &s.i;           // ua_ip points to 1 byte
                               // aligned int

*ua_ip = 2;                     // mis-aligned access to
                               // 1 byte aligned int
```

NOTE: The `HP_ALIGN` pragma, which is supported by the HP ANSI C compiler, is not supported by aCC. The `pack` and `unalign` pragmas can replace most of the `HP_ALIGN` functionality.

Optimization Pragmas

Following are the optimization pragmas supported by the HP aC++ compiler:

OPT_LEVEL Pragma

```
#pragma OPT_LEVEL 0
#pragma OPT_LEVEL 1
#pragma OPT_LEVEL 2
#pragma OPT_LEVEL 3
#pragma OPT_LEVEL 4
#pragma OPT_LEVEL INITIAL
```

When used with a numeric argument, the `OPT_LEVEL` pragma sets the optimization level to 0, 1, 2, 3, or 4.

The `INITIAL` argument causes the optimization level in effect at the start of the compilation, whether by default or specified on the command line, to be restored.

Example:

```
aCC -O prog.C

#pragma OPT_LEVEL 1
void A(){      // Optimize this function at level 1.
   ...
}
#pragma OPT_LEVEL 2
void B(){      // Restore optimization to level 2.
   ...
}
```

NOTE: This pragma cannot raise the optimization level above the level specified in the command line.

This pragma cannot be used within a function.

OPTIMIZE Pragma

```
#pragma OPTIMIZE ON|OFF
```

NOTE: This pragma is deprecated as of HP C/C++ A.06.15. You should use: `#pragma OPT_LEVEL` instead.

This pragma is used to toggle between optimization `on` and optimization `off` for different sections of source code whenever they are encountered in a top-to-bottom read of a source file.

Example:

The following example toggles between optimization `on` and optimization `off`:

```
aCC +O2 prog.C
#pragma OPTIMIZE OFF
void A(){     // Turn off optimization
    ...       // for this function
}

#pragma OPTIMIZE ON
void B(){     // Restore optimization
    ...       // to level 2.
}
```

You must specify one of the optimization level options on the aCC command, otherwise this pragma is ignored. This pragma cannot be used within a function.

FLOAT_TRAPS_ON Pragma

#pragma FLOAT_TRAPS_ON [*function*{,*function*}]

This pragma informs the compiler that the specified functions may enable floating-point trap handling. When the compiler is so informed, it will not perform loop invariant code motion (LICM) on floating-point operations in the functions named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled.

NOTE: This pragma is not supported in C++. It is deprecated for HP C and C++ C-mode. You should use`#pragma STDC FENV_ACCESS ON` instead.

For example,

```
#pragma FLOAT_TRAPS_ON xyz,abc
```

informs the compiler and optimizer that `xyz` and `abc` have floating-point traps turned on and therefore LICM optimization should not be performed. A dummy name `_ALL` represents all functions.

[NO]INLINE Pragma

`#pragma [NO]INLINE` *`sym`*`[,`*`sym`*`]`

The `[NO]INLINE pragma` enables[disables] inlining for all functions or specified function names.

For example, to specify inlining of the two subprograms `checkstat` and `getinput`, use:

```
#pragma INLINE checkstat, getinput
```

To specify that an infrequently called routine (`opendb`, for example) should not be inlined when compiling at optimization level 3 or 4, use:

```
#pragma NOINLINE opendb
```

Usage Notes for C++:

- Use the unmangled name of the function.
- All overloaded versions of the function will be affected.
- The pragma can affect "inline functions"--class member functions defined in the class definition; these have the same treatment as functions declared with the `inline` keyword.

Usage Notes for both C and C++:

- The pragma can be used without a function name, in which case it affects all functions until the next instance of the pragma or the end of the module. Consider the following:

  ```
  #pragma NOINLINE foo
  #pragma INLINE
  ```

  [`foo` will be inlined here]
- Inline functions (those declared with the `inline` keyword or as described above for C++), can be affected by `#pragma NOLINLINE`, which overrides the keyword.
- The `inline` keyword indicates a recommendation to the compiler that the function be inlined; the compiler can then make a profitability decision whether or not to perform the inlining.
- The `[NO]INLINE` pragma and `+O[no]inline`option are treated as directives to the compiler; the compiler obeys the pragma or option without performing profitability analysis.
- There are some cases that are not valid to inline. In these cases, the pragma or option is silently ignored.

NO_INLINE Pragma

```
#pragma NO_INLINE
```

This is equivalent to `#pragma NOINLINE`. The `NO_INLINE pragma` disables inlining for all functions or specified function names.

IVDEP Pragma

```
#pragma IVDEP
```

For the associated loop, this pragma directs the compiler to ignore any apparent loop dependencies involving references to array-typed entities.

NODEPCHK Pragma

```
#pragma NODEPCHK
```

For the associated loop, this pragma directs the compiler to ignore all loop dependencies (regardless of type) except for induction variables and some other scalar loop dependencies as determined by the compiler implementation.

NO_RETURN Pragma

`#pragma NO_RETURN` *`function1, [function2, . . .]`*

The `NO_RETURN` pragma is anassertion to the optimizer that the named functions never return to the call site.This allows the optimizer to delete any code after the function. A C++ function marked with the `NO_RETURN` pragma may still throw an exception unless it has an emty throw list.

Diagnostic Pragmas

The following are the diagnostic pragmas supported by the HP aC++ compiler.

diag_xxx Pragmas

```
#pragma diag_suppress message1 message2 ...
#pragma diag_warning message1 message2 ...
#pragma diag_error message1 message2 ...
#pragma diag_default message1 message2 ...
```

Command-line options help you generate diagnostic messages for the entire build or for a specific source file. The `diag` pragmas let you manage warnings for a specific region within a source file. The use of `#pragma diag_suppress` within the source code disables generation of the specified warning messages after the pragma in the source file. The pragma `diag_default`restores the default handling for the specified diagnostic messages. Similarly, `diag_warning` enables generation of the specified diagnostic messages, and `diag_error` converts a warning to an error.

Only diagnostics above 2000 are affected. This pragma will not affect the lower numbered diagnostics issued by the compiler's driver program.

Refer to the HP Code Advisor documentation for additional information.

The following example disables warning `#2549-D` locally:

```
int i;
#pragma diag_suppress 2549
printf ("i = %d\n", i);
#pragma diag_default 2549
```

Other Pragmas

The following are additional pragmas supported on the HP aC++ compiler:

assert Pragma

`#pragma assert non-zero(`*`constant-expression )"string"`*

When the compiler encounters this directive, it evaluates the expression. If the expression is zero, the compiler generates a message that contains the specified string and the compile-time constant expression.

For example:

```
#pragma assert non_zero(sizeof(a) == 12) "a is the wrong size"
```

In this example, if the compiler determines that `sizeof(a)` is not 12, the following diagnostic message is output:

```
"foo.c", line 10: error #2020: The assertion "(sizeof(a) == 12)" was not true.
a is the wrong size
```

Consider the following example that verifies both the size of a `struct` and the offset of one of its elements:

```
#include <stddef.h>
typedef struct {
  int a;
  int b;
} s;
#pragma assert non_zero(sizeof(s) == 8) "sizeof assert failed"
#pragma assert non_zero(offsetof(s,b) == 4) "offsetof assert failed"
```

This pragma provides a similar behavior to that of the Tru64 C compiler.

BINDING Pragma

`#pragma BINDING {hidden|protected|extern|default}`

Global symbols that follow this pragma will be given a specific binding. Command-line options and binding pragmas referring to specific symbols (for example, `pragma hidden` *symbol*) override this pragma. This pragma will override command-line bindings that do not refer to a specific symbol (for example, `-Bprotected`).

DEFAULT_BINDING Pragma

`#pragma DEFAULT_BINDING [`*`symbol`*`{,`*`symbol`*`}]`

Global symbols are assigned the default export class. These symbols may be imported or exported outside of the current load module. The compiler will access tentative and undefined symbols through the linkage table. Any symbol that is not assigned to another export class through use of another `-B` option will have the default export class.

ESTIMATED_FREQUENCY Pragma

`#pragma ESTIMATED_FREQUENCY f`

This block-scoped pragma allows you to tell the compiler your estimate of how frequently the current block is executed as compared to the immediately surrounding block. You can indicate the average trip count in the body of a `for` loop or the fraction of time a `then` clause is executed. Frequency, `f` can be expressed as a floating point or integer constant. The compiler accepts preprocessor expressions that evaluate to a compile time constant.

EXTERN Pragma

`#pragma EXTERN [`*`symbol`*`{,`*`symbol`*`}]`

The specified symbols, or all undefined symbols if no list is provided, are assigned to the default export class. Additionally, the compiler will inline the import stub for calls to these symbols. No compile time binding of these symbols will be done. All references to these symbols will be through the linkage table, so an unnecessary performance penalty will occur if `extern` is applied to a listed symbol that is resolved in the same load module. This is the pragma equivalent of `-Bextern` and is global in scope.

FREQUENTLY_CALLED Pragma

`#pragma FREQUENTLY_CALLED [`*`symbol`*`{,`*`symbol`*`}]`

This file-scoped pragma identifies functions that are frequently called within the application. The pragma must be placed prior to any definition of or reference to the named function. If not, the behavior is undefined. `FREQUENTLY_CALLED pragma` is independent of `+Oprofile=use` in that it overrides any dynamically obtained profile information.

HDR_STOP Pragma

```
#pragma HDR_STOP
```

This pragma instructs the compiler to stop precompiling headers.

HIDDEN Pragma

```
#pragma hidden [symbol{,symbol}]
```

The specified symbols, or all symbols (if no symbols are specified), are assigned the hidden export class. The hidden export class is similar to the protected export class. These symbols will not be preempted by symbols from other load modules, so the compiler may bypass the linkage table for both code and data references and bind them to locally defined code and data symbols.

In addition, hidden symbols will not be exported outside the current load module. The linker may eliminate them from a shared library, but in an executable, they remain accessible to the debugger unless `-Oprocelim` is also specified. This is the pragma equivalent of `-Bhidden` and is global in scope.

HP_DEFINED_EXTERNAL Pragma

```
#pragma HP_DEFINED_EXTERNAL name1[,name2,...nameN]
```

The specified symbols, or all undefined symbols (if no list is provided), are assigned to the default export class. Additionally, the compiler will inline the import stub for calls to these symbols. No compile time binding of these symbols will be done. All references to these symbols will be through the linkage table, so an unnecessary performance penalty will occur if extern is applied to a listed symbol that is resolved in the same load module. This is the pragma equivalent of `-Bextern` and is global in scope.

This pragma is equivalent to `#pragma EXTERN`.

HP_DEFINED_INTERNAL Pragma

```
#pragma HP_DEFINED_INTERNAL name1[,name2,...nameN]
```

The specified symbols, or all symbols (if no symbols are specified), are assigned the protected export class. That means these symbols will not be preempted by symbols from other load modules, so the compiler may bypass the linkage table for both code and data references and bind them to locally defined code and data symbols. This pragma is equivalent to `-Bprotected` and is global in scope.

This pragma is equivalent to `#pragma PROTECTED`.

IF_CONVERT Pragma

```
#pragma IF_CONVERT
```

This block-scoped pragma instructs the compiler to If-Convert the current scope. There is no command-line option equivalent.

If-Conversion is a compiler process that eliminates conditional branches by the use of predicates. The compiler is instructed to If-Convert all non-loop control flow nested within the current block.

Without this pragma, the compiler would employ its own heuristics to determine whether to perform If-Conversion. With this pragma, If-Conversion is always performed.

If-Convert can be specified in a loop containing conditional branches other than the loop-back branch. This makes it more likely the compiler will modulo schedule the loop, as loops containing conditional branches cannot be modulo scheduled. The pragma can also be used for non-looping constructs.

POP Pragma

`#pragma POP`

The last pushed pragma is removed from the pragma stack and state is restored. The binding state reverts to the binding state prior to the last push. Note that this pragma can only be used with the blanket binding pragmas.

Pragma (once)

`_Pragma ("once")`

The `_Pragma ("once")` operator is equivalent to `#pragma once`. This operator ensures that the source file is included only once during compilation.

PROTECTED Pragma

`#pragma PROTECTED [`*`symbol`*` {,`*`symbol`*`}]`

The specified symbols, or all symbols (if no symbols are specified), are assigned the `PROTECTED` export class. That means these symbols will not be preempted by symbols from other load modules, so the compiler may bypass the linkage table for both code and data references and bind them to locally defined code and data symbols. This pragma is equivalent to `-Bprotected` and is global in scope.

PTRS_STRONGLY_TYPED Pragma

`#pragma [NO]PTRS_STRONGLY_TYPED {BEGIN | END}`

This pragma turns strong pointer type testing on and off When turned on (BEGIN) if a pointer typing error is detected, it will generate a warning if the typing error can be safely ignored. If the typing error cannot be safely ignored, it will generate a warning and flag the compilation appropriately, or if this is not possible, it will generate an error. This feature is disabled using the END attribute..

PTRS_TO_GLOBALS Pragma

`#pragma [NO]PTRS_TO_GLOBALS `*`name`*

This pragma aids alias analysis. It must be specified at global scope and immediately precede the declaration of the variable or entry named. The pragma tells the optimizer whether the global variable or entry *`name`* is accessed [is not accessed] through pointers. If `NOPTRS_TO_GOBALS` is specified, it is assumed that statically-allocated data (including file-scoped globals, file scoped statics, and function-scoped static variables) will not be read or written through pointers. The default is `PTRS_TO_GLOBALS`.

PUSH Pragma

`#pragma PUSH `*`pragma_name`*

This pragma will save the current state on the pragma stack for the named pragma. All subsequent uses of the named binding pragma will be reverted when the “POP” is encountered. Note that this pragma can only be used with the blanket binding pragmas.

RARELY_CALLED Pragma

`#pragma RARELY_CALLED [`*`symbol`*`{,`*`symbol`*`}]`

This file-scoped pragma identifies functions that are rarely called within the application. The pragma must be placed prior to any definition of or reference to the named function. If not, the behavior is undefined. `RARELY_CALLED` is independent of `+Oprofile=use` option. It overrides any dynamically obtained profile information.

STDC CX_LIMITED_RANGE Pragma

```
#pragma STDC CX_LIMITED_RANGE ON
```

```
#pragma STDC CX_LIMITED_RANGE OFF
```

This pragma enables limited range mathematical behavior for specific blocks of code. Note that, this pragma applies to complex arithmetic only. Also see the *ISO/IEC 9899 Standard*.

This pragma can occur outside an external declaration or within a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `STDC CX_LIMITED_RANGE` pragma is encountered or until the end of the translation unit. When within a compound statement, the pragma takes effect from its occurrence until another `STDC CX_LIMITED_RANGE` pragma is encountered within a nested compound statement, or until the end of the compound statement.

If this pragma is used in any other context, the behavior is undefined. The default state is off.

STDC FLOAT_CONST_DECIMAL64 Pragma

```
#pragma STDC FLOAT_CONST_DECIMAL64 [ON | OFF | DEFAULT]
```

With this pragma set to OFF, unsuffixed floating-point constants are treated as having type `double`.

With this pragma set to ON, unsuffixed floating-point constants are treated as having type `_Decimal64`.

The pragma can occur in either of these two contexts:

- Outside external declarations

  In this case, the pragma takes effect from its occurrence until another `FLOAT_CONST_DECIMAL64` pragma is encountered, or until the end of the translation unit.

- Preceding all explicit declarations and statements inside a compound statement.

  In this case, the pragma takes effect from its occurrence until another `FLOAT_CONST_DECIMAL64` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement, the state for the pragma is restored to its condition just before the compound statement.

If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is OFF.

For more information on using Decimal FP, see the *HP aC++/HP ANSI C Release Notes* section "Decimal floating-point arithmetic supported" under "New Features in the A.06.20 Release."

STDC FP_CONTRACT Pragma

```
#pragma STDC FP_CONTRACT ON
```

```
#pragma STDC FP_CONTRACT OFF
```

This pragma tells the compiler whether or not it is permitted to contract expressions. Also see ISO/IEC 9899 Standard.

Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered within a nested compound statement, or until the end of the compound statement. At the end of a compound statement, the state for the pragma is restored to its condition before the compound statement.

If this pragma is used in any other context, the behavior is undefined. The default state is `ON`.

STDC FENV_ACCESS Pragma

```
#pragma STDC FENV_ACCESS ON
```

```
#pragma STDC FENV_ACCESS OFF
```

This pragma provides a means to inform the compiler when a program might access the floating-point environment to test flags or run under non-default modes. Use of the pragma allows certain optimizations that could subvert flag tests and mode changes such as global common sub expression elimination, code motion, and constant folding.

The pragma can be placed either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered or until the end of the translation unit. When inside a compound statement, the pragma is in effect from its occurrence until another `FENV_ACCESS` pragma is encountered within the nested compound statement or until the end of the compound statement. At the end of a compound statement, the state for the pragma is restored to its condition just before the compound statement.

If the pragma is used in any other context, the behavior is undefined. If part of a program tests flags or runs under non-default mode settings but was translated with the state for the `FENV_ACCESS` pragma off, then the behavior of the program is undefined.

Also see the *ISO/IEC 9899 Standard*.

UNROLL_FACTOR Pragma

```
#pragma UNROLL_FACTORn
```

```
#pragma UNROLLn
```

```
#pragma UNROLL (n)
```

This block-scoped pragma applies the unroll factor for a loop containing the current block. You can apply an unroll factor that you think is best for the given loop or apply no unroll factor to the loop. If this pragma is not specified, the compiler uses its own heuristics to determine the best unroll factor for the inner loop.

A user specified unroll factor will override the default unroll factor applied by the compiler.

Specifying *n*=1 will prevent the compiler from unrolling the loop.

Specifying *n*=0 allows the compiler to use its own heuristics to apply the unroll factor.

Note that this option has no effect on loop unroll-and-jam.

NOTE: `UNROLL_FACTOR` pragma will be ignored if it is placed in a loop other than the innermost loop. The UNROLL pragma must be immediately followed with a loop statement and will be ignored if it is not an innermost loop.

OMP ATOMIC Pragma

```
#pragma omp atomic
        expression-stmt
```

where *`expression-stmt`* must have one of the following forms:

- `x binop = expr`
- `x++`
- `++x`
- `x--`
- `--x`

Here, `x` is an `lvalue` expression with scalar type and `expr` is an expression with scalar type that does not reference the object designated by `x`.

The `atomic` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

OMP BARRIER Pragma

```
#pragma omp barrier
```

The `barrier` pragma synchronizes all the threads in a team. When encountered, each thread waits until all the threads in the team have reached that point.

The smallest statement to contain a `barrier` must be a block or a compound statement. `barrier` is valid only inside a parallel region and outside the scope of `for`, `section`, `sections`, `critical`, `ordered`, and `master`.

OMP CRITICAL Pragma

```
#pragma omp critical [(name)]
               structured-block
```

The `critical` pragma identifies a construct that restricts the execution of the associated structured block to one thread at a time.

The *`name`* parameter is optional. All unnamed critical sections map to the same name.

OMP FOR Pragma

```
#pragma omp for [clause1,clause2, ...]

                  for-loop
```

where [*`clause1, clause2`*, ...] indicates that the clauses are optional. There can be zero or more clauses.

`clause` may be one of the following:

- *`private(list)`*
- *`firstprivate(list)`*
- *`lastprivate(list)`*
- *`ordered`*
- *`schedule(kind[,chunksize])`*
- *`nowait`*

See *"OpenMP Clauses" (page 114)* for more information.

OMP FLUSH Pragma

```
#pragma omp flush [(list)]
```

where `(`*`list`*`)` names the variables that will be synchronized.

The `flush` pragma, whether explicit or implied, specifies a cross-thread sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in the memory. A `flush` directive without a list is implied for the following directives:

- `barrier`
- an entry to and exit from `critical`
- at entry to and exit from `ordered`
- at entry to and exit from `parallel`
- at entry to and exit from `parallel for`
- at entry to and exit from `parallel sections`

- at exit from `single`
- at exit from `for`
- at exit from `sections`

NOTE: The directive is not implied if a `nowait` clause is present.

OMP MASTER Pragma

```
#pragma omp master
           structured-block
```

The `master` pragma directs that the `structured-block` following it should be executed by the master thread (*`thread 0`*) of the team. Other threads in the team do not execute the associated block.

OMP ORDERED Pragma

```
#pragma omp ordered
            structured-block
```

The `ordered` pragma indicates that the following structured block should be executed in the same order in which iterations will be executed in a sequential loop.

An `ordered` directive must be within the dynamic extent of a `for` or a `parallel for` construct that has an `ordered` clause. When the `ordered` clause is used with schedule which has a `chunksize`, then the `chunksize` is ignored by the compiler.

OMP PARALLEL Pragma

```
#pragma omp parallel [clause1, clause2,...]
             structured-block
```

where `[clause1, clause2, ...]` indicates that the clauses are optional. There can be zero or more clauses.

`clause` can be one or more of the following:

- *`private(list)`*
- *`firstprivate(list)`*
- *`default(shared|none)`*
- *`shared(list)`*
- *`reduction(op:list)`*
- *`if (scalar-expression)`*
- *`copyin (list)`*
- *`num_threads`*

The `parallel` pragma defines a parallel region, which is a region of the program that is executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

OMP PARALLEL FOR Pragma

```
#pragma omp parallel for [clause1, clause2, ... ]
                for-loop
```

where [*`clause1`*, *`clause2`*, ...] indicates that the clauses are optional. There can be zero or more clauses.

The `parallel for` pragma is a shortcut for a parallel region that contains a single `for` pragma. `parallel for` admits all the allowable clauses of the `parallel` pragma and the `for` pragma except for the `nowait` caluse.

OMP PARALLEL SECTIONS Pragma

```
#pragma omp parallel sections [clause1, clause2, ...]
{
   [#pragma omp section  ]
           structured-block
   [#pragma omp section
           structured-block ]
. . .}
```

where [*clause1*, *clause2*, ...] indicates that the clauses are optional. There can be zero or more clauses.

The `parallel sections` pragma is a shortcut for specifying a parallel clause containing a single `sections` pragma. `parallel sections` admits all the allowable clauses of the `parallel` pragma and the `sections` pragma except for the `nowait` clause.

OMP SECTIONS Pragma

```
#pragma omp sections [clause1, clause2, ...]
{
#pragma omp section
          [ structured-block ]
[#pragma omp section
           structured-block ]
. . .
}
```

where `[`*`clause1, clause2,`*` ...]` indicates that the clauses are optional. There can be zero or more clauses. *`clause`* may be one of the following:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(op:list)`
- `nowait`

The `section` or `sections` pragmas identify a construct that specifies a set of constructs to be divided among threads in a team. Each section is executed by one of the threads in the team.

OMP SINGLE Pragma

```
#pragma omp single [clause1, clause2, . . .]
           [ structured-block ]
```

where `[`*`clause1, clause2,`*` ...]` indicates that the clauses are optional. There can be zero or more clauses.

`clause` may be one of the following:

- *`private(list)`*
- *`firstprivate(list)`*
- *`copyprivate(list)`*
- *`nowait`*

The `single` directive identifies a construct that specifies the associated structured block that is executed by only one thread in the team (not necessarily the master thread).

OMP TASK Pragma

```
#pragma omp task [clause1, clause2, . . .] new-line
           [ structured-block ]
```

where `[clause1, clause2, ...]` indicates that the clauses are optional. There can be zero or more clauses.

`clause` may be one of the following:

- *`if (scalar-expression))`*
- *`untied`*
- *`default (shared | none)`*
- *`private (list)`*
- *`firstprivate (list)`*
- *`shared (list)`*

The `TASK` directive defines an explicit task.

OMP TASKWAIT Pragma

```
#pragma omp taskwait new-line
```

The `TASKWAIT` directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

Because the `TASKWAIT` construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The `TASKWAIT` directive may be placed only at a point where a base language statement is allowed. The `TASKWAIT` directive may not be used in place of the statement following an `if`, `while`, `do`, `switch`, or `label`.

OMP THREADPRIVATE Pragma

```
#pragma omp threadprivate (list)
```

where `(list)` is a comma-separated list of variables that do not have an incomplete type.

The `threadprivate` directive makes the named *`file-scope`*, *`namescope-scope`*, or *`static block-scope`* variables private to a thread.

OpenMP Clauses

Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. The order in which clauses appear in directives is not significant. If variable-list appears in a clause, it must specify only variables. The following is the list of clauses in OpenMP directives:

private

```
private(list)
```

The `private` clause declares the variables in the list to be private to each thread in a team. A new object with automatic storage duration is allocated within the associated `structured block` for each thread in the stream.

firstprivate

```
firstprivate(list)
```

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause. Variables specified in the list have `private` clause semantics described earlier. The new private object is initialized, as if there is an implied declaration inside the structured block and the initializer is the value of the original object.

lastprivate

```
lastprivate(list)
```

When `lastprivate` clause is specified in a loop or section, the value of the lastprivate variable from either the sequentially last iteration of the associated loop, or the lexically last section directive is assigned to the variable's original object. The `lastprivate` clause provides a superset of the functionality provided by the `private` clause. Variables specified in the list have `private` clause semantics described earlier.

copyprivate

`copyprivate(`*`list`*`)`

The `copyprivate` clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

NOTE: The `copyprivate` clause can only appear on the `single` directive.

if

`if(`*`scalar-expression`*`)`

The associated block of code will be executed in parallel if the *`scalar-expression`* evaluates to a non-zero value. Otherwise no parallelization happens and it is executed sequentially.

Example:

```
#pragma omp parallel private(x) if (a>b) reduction(+:p)
{
   // code to be parallelized only when a is greater than b
}
```

default

`default(shared|none)`

Specifying `default(shared)` clause is equivalent to explicitly listing each currently visible variable in a `shared` clause unless it is `threadprivate` or `const-qualified`. A variable referenced in the scope of `default(none)` should be explicitly qualified by a `private` or `shared` clause.

shared

`shared(`*`list`*`)`

The `shared` clause causes the variables that appear in the *`list`* to be shared among all threads in a team. All threads within a team access the same storage area for the shared variables.

copyin

`copyin(`*`list`*`)`

The `copyin` clause copies the value of master thread's copy of a threadprivate variable to all other threads at the beginning of the parallel region. This clause can only be used with the `parallel` directive.

reduction

`reduction(`*`op:list`*`)`

The `reduction` clause performs a reduction on the scalar variables that appear in the *`list`*, with the operator `op`.

nowait

`nowait`

The `nowait` clause removes the implicit barrier synchronization at the end of a `for` or `sections` construct.

ordered

`ordered`

The `ordered` clause must be present when `ordered` directives bind to the `for` construct.

schedule

`schedule(`*`kind`*`[,`*`chunksize`*`])`

The `schedule` clause specifies how iterations of the for loop are divided among threads of the team. The *`kind`* of schedule can be: `static`, `dynamic`, `guided`, or `runtime`. *`chunksize`* should be a loop invariant integer expression.

num_threads

`num_threads(`*`interger-expression`*`)`

The `num_threads` clause allows a user to request a specific number of threads for a parallel construct. If the `num_threads` clause is present, then the value of the integer expression is the number of threads requested.

Attributes

`__attribute__` is a language feature that allows you to add attributes to functions (or with the `aligned` attribute, to variables or structure fields). The capabilities are similar to those of `#pragma`. It is more integrated into the language syntax than pragmas and its placement in the source code depends on the construct to which the attribute is being applied.

The attributes supported are:

- aligned
- malloc
- non_exposing
- noreturn
- format
- visibility
- warn_unused_result

attribute aligned

`__attribute__ (aligned (`*`alignment`*`))`

The `aligned` attribute specifies the minimum alignment for a variable or structure field, measured in bytes. For example, the following declaration causes the compiler to allocate the global variable `x` on a 16-byte boundary:

```
int x __attribute__ ((aligned (16))) = 0;
```

You can also specify the alignment of structure fields. For example, the following causes the compiler to allocate the field member "`x`" to be aligned on a 128-byte boundary:

```
struct foo { int x[2] __attribute__ ((aligned (128)));}
```

The maximum alignment that can be specified is 128. This feature is for compatibility with `gcc`.

attribute malloc

`__attribute__ ((malloc))`

The `malloc` attribute is used to improve optimization by telling the compiler that:

1. The return value of a call to such a function points to a memory location or can be a null pointer.
2. On return of such a call (before the return value is assigned to another variable in the caller), the memory location mentioned in 1. can be referenced only through the function return value; e.g., if the pointer value is saved into another global variable in the call, the function is not qualified for the malloc attribute.
3. The lifetime of the memory location returned by such a function is defined as the period of program execution between a) the point at which the call returns and b) the point at which the memory pointer is passed to the corresponding deallocation function. Within the lifetime of the memory object, no other calls to malloc routines should return the address of the same object or any address pointing into that object.

Many wrappers around `malloc()` obey these rules. (The compiler already knows that `malloc()` itself obeys these rules.)

Example:

```
void *foo(int i) __attribute__ ((malloc));
```

attribute non_exposing

```
__attribute__ ((non_exposing))
```

The `non_exposing` attribute is used to improve optimization by telling the compiler that this function does not cause any address it can _derive_ from any of its formal parameters to become visible after a call to the function returns.

An address becomes visible if the function returns a value from which it can be `_directly derived_` or if the function stores it in a memory location that is visible (can be referenced directly or indirectly) after the call to the function returns.(Note that there is no such thing as a formal parameter of array type. A formal parameter declared to be of type "array of T" is treated as being of type "pointer to T"; and when an actual argument is of type "array of T", a pointer to the first element of that array is passed.)Many wrappers around free() obey these rules. (The compiler already knows that `free()` itself obeys these rules.) Many functions that have nothing to do with memory allocation also obey these rules.For the purposes of the specification above, the definitions of the terms `_directly derived_` and `_derived_` are as follows:The addresses that can be `_directly derived_` from some value `V` are the following:

* If `V` is or can be converted to a pointer value (except by a C++ user-defined conversion), then consider `P` to be a pointer object containing that value. The value of any expression `_based on_` `P` (as defined in C99) can be `_directly derived_` from `V`. For example, if `P` is a pointer object containing the value `V`, then "`P`", "`&P->f`", "`&P[i]`", and "`P+j`" are expressions based on `P`, and thus their values are `_directly derived_` from `V`.* If `V` is an array, then any addresses that can be `_directly derived_` from `V`'s elements can be `_directly derived_` from `V`.* If `V` is a class, struct, or union, then any addresses that can be `_directly derived_` from `V`'s nonstatic data members can be `_directly derived_` from `V`.* If `V` is a reference, then `&V` can be `_directly derived_` from `V`.The addresses that can be `_derived_` from some value `V` are the addresses that can be `_directly derived_` from `V` and the addresses that can be `_derived_` from the result of dereferencing those addresses. The function does not store addresses passed to it as arguments to any memory location that is visible (can be referenced directly or indirectly) after the call to this function returns.

Example:

```
void foo(int *pi) __attribute__ ((non_exposing));
```

attribute noreturn

```
__attribute__ ((noreturn))
```

Similar to the `NO_RETURN` pragma, this attribute asserts to the optimizer that this function never returns to the call site. This allows the optimizer to delete any code after the function call. A C++ function marked with this attribute may still throw an exception unless it has an empty throw list.

Example:

```
void foo(int i) __attribute__ ((noreturn));
```

attribute format

```
__attribute__ ((format(type, arg_format_string, arg_check_start)))
```

The `format` attribute specifies that a function takes `printf`, `scanf`, `strftime` or `strfmon` style arguments which should be type-checked against a format string. In the example above, the format string is the second argument of the function foo and the arguments to check start with the third argument.

Example:

```
int foo(int i, const char *my_format, ...) __attribute__((format(printf, 2, 3)));
```

attribute visibility

```
__attribute__ ((visibility("default"|"protected"|"hidden")))
```

The `visibility` attributes `"default"`, `"protected"`, and `"hidden"`, are equivalent to the options `-Bdefault`, `-Bprotected`, and `-Bhidden`, and the pragmas `DEFAULT_BINDING`, `EXTERN`, and `HIDDEN`, respectively.

Example:

```
void foo(int i) __attribute__ ((visibility("hidden"));
```

attribute warn_unused_result

```
__attribute__ ((warn_unused_result))
```

If a caller of a function with this attribute does not use its return value, the compiler emits a warning. This is useful for functions where not checking the result can be a security problem or always a program bug, as with `realloc`. The following example results in a warning on line 5:

```
int fn () __attribute__ ((warn_unused_result));
int test()
{
 if (fn () < 0) return -1;
 fn ();
 return 0;
}
```

4 Preprocessing Directives

HP aC++ has its own, internal, preprocessor which is similar to the HP C preprocessor described in the *HP C/HP-UX Reference Manual*. When you issue the aCC command, your source files are automatically preprocessed.

This Chapter discusses the following topics:

Overview of the Preprocessor

A preprocessor is a text-processing program that manipulates the text within your source file. You enter preprocessing directives into your source file to direct the preprocessor to perform certain actions on the source file. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress the compilation of part of the file by conditionally removing sections of the text. It also expands preprocessor macros and conditionally strips out comments.

Syntax

The general syntax for a preprocessor directive is:

```
preprocessor-directive ::=
     include-directive newline
     macro-directive newline
     conditional-directive newline
     line-directive newline
     pragma-directive newline
     error-directive newline
     trigraph-directive newline
     warning-directive newline
```

Usage Guidelines

Following are rules and guidelines for using preprocessor directives:

- A preprocessor directive must be preceeded by a pound sign (#). White-space characters may precede the # character.
- The # character is followed by any number of spaces and horizontal tab characters and a preprocessor directive.

- A preprocessor directive is terminated by a newline character.
- Preprocessor directives, as well as normal source lines, can be continued over several lines. End the lines that are to be continued with a backslash (\).
- Some directives can take actual arguments or values.
- Comments in the source file that are not passed through the preprocessor are replaced with a single white space character (ASCII character number decimal 32).

Preprocessor directives provide the following functionality:

- Source File Inclusion (`#include`, `#include_next`)
- Macro Replacement (`#define`, `#undef`)
- Assertions (`#assert`, `#unassert`)
- Conditional Compilation (`#if`, `#ifdef`, `..` *`#endif`*)
- Line Control (`#line`)
- Pragma Directive (`#pragma`, `_Pragma` operator)
- Error Directive (`#error`)
- Trigraph Sequences
- Warning Directive

Source File Inclusion (#include, #include_next)

You can include the contents of other files within the source file using the `#include` or `#include_next` directives.

Syntax

```
include-directive ::=
     #include <filename>
     #include "filename"
     #include identifier

include_next-directive ::=
     #include_next <filename>
     #include_next "filename"
     #include_next identifier
```

Description

The `#include` preprocessing directive causes HP aC++ to read source input from the file named in the directive. Usually, include files are named `filename.h`.

If the file name is enclosed in angle brackets (< >), the default system directories are searched to find the named file. If the file name is enclosed in double quotation marks (“ “), by default, the directory of the file containing the `#include` line is searched first, then directories named in `-I` options in left-to-right order, and last directories on a standard list.

The arguments to the `#include` directive are subject to macro replacement before being processed. Thus, if you use a `#include` directive of the form `#include` *`identifier`*, *`identifier`* must be a previously defined macro that when expanded produces one of the above defined forms of the `#include` directive. Refer to Macro Replacement (`#define`, `#undef`) for more information on macros.

Error messages produced by HP aC++ indicate the name of the `#include` file where the error occurred, as well as the line number within the file.

Examples

```
#include <iostream.h>
#include "myheader.h"
#ifdef   MINE
#   define  filename  "file1.h"
#else
#   define  filename  "file2.h"
#endif
#include filename
```

The `#include_next` preprocessor directive is similar to the `#include` directive, but tells the preprocessor to continue the include-file search beyond the current directory, and include the subsequent instance found in the file-search path.

Macro Replacement (#define, #undef)

You can define C++ macros to substitute text in your source file.

Syntax

```
macro-directive ::=
#define identifier [replacement-list]
#define identifier( [identifier-list] )  [replacement-list]
#undef identifier

replacement-list ::=
     token
     replacement-list token
```

Description

A `#define` preprocessing directive defines the identifier as a macro name that represents the replacement-list. This is of the form:

`#define` *`identifier`* [*`replacement-list`*]

The macro name is then replaced by the list of tokens wherever it appears in the source file (except inside of a string, character constant, or comment). A macro definition remains in force until it is undefined through the use of the `#undef` directive or until the end of the compilation unit.

The *`replacement-list`* must fit on one line. If the line becomes too long, it can be broken up into several lines provided that all lines but the last are terminated by a backslash (\) character. The following is an example:

```
#define mac very very long\

replacement string
```

The \ must be the last character on the line. You cannot add any spaces or comments after it.

Macros can be redefined without an intervening `#undef` directive. Any parameter used must agree in number and spelling with the original definition, and the replacement lists must be identical. All white space within the replacement-list is treated as a single blank space regardless of the number of white-space characters you use. For example, the following `#define` directives are equivalent:

```
#define foo x  +   y

#define foo x + y
```

The *`replacement-list`* may be empty. If the token list is not provided, the macro name is replaced with no characters.

Macros with Parameters

You can create macros that have parameters. The syntax of the #define directive that includes formal parameters is as follows:

```
#define identifier( [identifier-list] ) [replacement-list]
```

The macro name is `identifier`. The formal parameters are provided by the `identifier-list` enclosed in parentheses. The open parenthesis ('(') must immediately follow the identifier with no intervening white space. If there is a space between the identifier and the parenthesis, the macro is defined as if it were the first form and the replacement-list begins with the `(` character.

The formal parameters to the macro are separated with commas. They may or may not appear in the `replacement-list`. When the macro is invoked, the actual arguments are placed in a parenthesized list following the macro name. Commas enclosed in additional matching pairs of parentheses do not separate arguments but are themselves components of arguments.

The actual arguments replace the formal parameters in the token string when the macro is invoked.

Specifying String Literals with the # Operator

If a formal parameter in the macro definition directive's replacement string is preceded by a # operator, it is replaced by the corresponding argument from the macro invocation, preceded and followed by a double-quote character (") to create a string literal. This feature, available only with the ANSI C preprocessor, may be used to turn macro arguments into strings. This feature is often used with the fact that HP aC++ concatenates adjacent strings.

Example:

```
#include <iostream.h>
#define display(arg) cout << #arg << "\n"  //define the macro
int main()
{
     display(any string you want to use);    //use the macro
}
```

After HP aC++ expands the macro definition in the preceding program, the following code results:

```
 ...
main ()
{
     cout << "any string you want to use" << "\n";
}
```

Concatenating Tokens with the ## Operator

Use the ## operator within macros to create a single token out of two other tokens. Usually, one of these two tokens is the actual argument for a macro-parameter. Upon expansion of the macro, each instance of the ## operator is deleted and the tokens preceding and following the ## are concatenated into a single token.

Example 1

The following illustrates the ## operator:

```
      // define the macro; the ## operator
     // concatenates arg1 with arg2
#define concat(arg1,arg2) arg1 ## arg2
int main()
{
     int concat(fire,fly);
     concat(fire,fly) = 1;
     printf("%d \n",concat(fire,fly));
}
```

Preprocessing this program yields the following:

```
int main()
{
     int firefly;
     firefly = 1;
     printf("%d \n",firefly );
}
```

Example 2

You can use the # and ## operators together:

```
#include <iostream.h>
#define show_me(arg) int var##arg=arg;\
    cout << "var" #arg " is " << var##arg << "\n";
int main()
{
    show_me(1);
}
```

Preprocessing this example yields the following code for the `main` procedure:

```
int main()
{
    int var1=1; cout << "var" "1" " is " << var1 << "\n";
}
```

After compiling the code with aCC and running the resulting executable file, you get the following results:

```
var1 is 1
```

Spaces around the # and ## are optional.

In both the # and ## operations, the arguments are substituted as is, without any intermediate expansion. After these operations are completed, the entire replacement text is rescanned for further macro expansions.

NOTE: The result of the preprocessor concatenation operator ## must be a `_single_` token. In particular, the use of ## to concatenate strings is redundant and not legal C or C++. For example:

```
#include
#define concat_token(a, b) a##b
#define concat_string(a, b) a b
int main() {
    // Wrong:
    printf("%s\n", concat_token("Hello,", " World!"));
    // Correct:
    printf("%s\n", concat_string("Hello,", " World!"));
    // Best: (macro not needed at all!):
    printf("%s\n", "Hello," " World!");
}
```

Using Macros to Define Constants

The most common use of the macro replacement is in defining a constant. In C++ you can also declare constants using the keyword const. Rather than explicitly putting constant values in a program, you can name the constants using macros, then use the names in place of the constants. By changing the definition of the macro, you can more easily change the program:

```
#define ARRAY_SIZE 1000float x[ARRAY_SIZE];
```

In this example, the array `x` is dimensioned using the macro `ARRAY_SIZE` rather than the constant 1000. Note that expressions that may use the array can also use the macro instead of the actual constant:

```
for (i=0; i<<ARRAY_SIZE; ++i) f+=x[i];
```

Changing the dimension of `x` means only changing the macro for `ARRAY_SIZE`. The dimension changes and so do all of the expressions that make use of the dimension.

Other Macros

Some other common macros used by C programmers include:

```
#define FALSE 0
```

```
#define TRUE 1
```

The following macro is more complex. It has two parameters and produces an inline expression which is equal to the maximum of its two parameters:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Parentheses surrounding each argument and the resulting expression ensure that the precedences of the arguments and the result interact properly with any other operators that might be used with the `MAX` macro.

Using a macro definition for `MAX` has some advantages over a function definition. First, it executes faster because the macro generates in-line code, avoiding the overhead of a function call. Second, the `MAX` macro accepts any argument types. A functional implementation of `MAX` would be restricted to the types defined for the function.

Note that because each argument each argument to the `MAX` macro appears in the token string more than once, the actual arguments to the `MAX` macro may have undesirable side effects.

The following example may not work as expected because the argument `a` is incremented two times when `a` is the maximum:

```
i = MAX(a++, b);
```

This expression is expanded to:

```
i = ((a) > (b) ? (a) : (b))
```

Given this macro definition, the statement

```
i = MAX(a, b+2);
```

is expanded to:

```
i = ((a) > (b+2) ? (a) : (b+2));
```

Example 1

```
#define isodd(n)  ( ((n % 2) == 1) ? (TRUE) : (FALSE))
```

This macro tests a number and returns `TRUE` if the number is odd. It returns `FALSE` otherwise.

Example 2

```
#define eatspace()while((c=getc(input))==c==’\n’c\ = ‘t’ )
```

This macro skips white spaces.

Using Constants and Inline Functions Instead of Macros

In C++ you can use named constants and inline functions to achieve results similar to using macros. You can use `const` variables in place of macros. You can also use inline functions in many C++ programs where you would have used a function-like macro in a C program. Using inline functions reduces the likelihood of unintended side effects, since they have return types and generate their own temporary variables where necessary.

Example

The following program illustrates the replacement of a macro with an inline function:

```
#include <stream.h>
#define distance1(rate,time) (rate * time)
// replaced by :
inline int distance2 ( int rate, int time )
{
     return ( rate * time );
}
int main()
{
     int i1 = 3, i2 = 3;

     printf("Distance from macro : %d\n",
             distance1(i1,i2) )
;     printf("Distance from inline function : %d\n",
             distance2(i1,i2) );
}
```

Predefined Macros

In addition to `__LINE__` and `__FILE__`, HP aC++ provides the following predefined macros. The list describes the complete set of predefined macros that produce special information. They cannot be undefined nor changed.

- `__cplusplus` produces the decimal constant 199707L, indicating that the implementation supports ANSI/ISO C++ International Standard features. For example,

  ```
  #if (__cplusplus >= 199711L)
  #include
  #else
  #include
  ```

- `__DATE__` produces the date of compilation in the form `Mmm dd yyyy`.
- `__FILE__` produces the name of the file being compiled.
- `__HP_aCC` identifies the HP aC++ compiler driver version. It is represented as a 6-digit number in the format `mmnnxx`, where `mm` is the major version number, `nn` is the minor version number, and `xx` is any extension. For example, for version A.01.21, `__HP_aCC=012100`.
- `__hpux` is defined.
- `__ia64` is defined.
- `__LINE__` produces the current source line number.
- `__LP64__` is defined for `+DD64`.
- `_ILP32` is defined for `+DD32`.
- `_LP64` is defined for `+DD64`.
- `__STDCPP__` produces the decimal constant 1, indicating that the preprocessor is in ANSI C/C++ mode.
- `__TIME__` produces the time of compilation in the form `hh:mm:ss`.
- `__unix` is defined.

To use some HP-UX system functions you may need to define the symbol `__HPUX_SOURCE`.

See *stdsyms(5)* manpage or the *HP-UX Reference Manual* for more information.

Assertions (#assert, #unassert)

Use `#assert` and `#unassert` to set a predicate name or predicate name and token to be tested with a `#if` directive. Note that you must also specify the `-ext` option at compile and link time.

Syntax

`#assert`*`predicate-name`*[*`token-name`*]

`#unassert`*`predicate-name`*[*`token-name`*]

Description

`#assert` sets the *`predicate-name`* [*`token-name`*] to true. `#unassert` sets the *`predicate-name`* [*`token-name`*] to false.

Note that when testing a predicate, it must be preceded by the # character.

HP aC++ predefines the following predicates:

- `#assert system(unix)`
- `#assert model(lp64) // when +DA2.0W is used`
- `#assert model(ilp32) // default`
- `#assert endian(big)`

Example:

```
int main()
{
#assert dimensions(three)  // Set predicate and token to true.
#if #dimensions(two)
#error "May not compile in 2 dimensions"
#endif

#if #dimensions(three)
int x, y, z;
#endif

#unassert dimensions       // Set predicate and all tokens to false.
}
```

Conditional Compilation (#if, #ifdef, .. #endif)

Conditional compilation directives allow you to delimit portions of code that are compiled only if a condition is true.

Syntax

```
conditional-directive ::=
#if            constant-expression newline
#ifdef         identifier newline [group]
#ifndef        identifier newline [group]
#else          newline [group]
#elif          constant-expression newline [group]
#endif
```

Here, `constant-expression` may also contain the defined operator:

`defined` *`identifier`*

`defined` *`(identifier)`*

The `constant-expression` is like other C++ integral constant expressions except that all arithmetic is carried out in `long int` precision. Also, the expressions cannot use the `sizeof` operator, a cast, an enumeration constant, or a `const` object.

Description

You can use `#if`, `#ifdef`, or `#ifndef` to mark the beginning of the block of code that will only be compiled conditionally. An `#else` directive optionally sets aside an alternative group of statements. You mark the end of the block using an `#endif` directive.

The following `#if` directive illustrates the structure of conditional compilation:

```
#if constant-expression
     ...
```

(Code that compiles if the expression evaluates to a nonzero value.)

```
     ...
#else
     ...
```

(Code that compiles if the expression evaluates to zero.)

```
     ...
#endif
```

Using the defined Operator

You can use the `defined` operator in the `#if` directive to use expressions that evaluate to 0 or 1 within a preprocessor line. This saves you from using nested preprocessing directives.

The parentheses around the identifier are optional. Below is an example:

```
#if defined (MAX) && ! defined (MIN)
     ...
```

Without using the defined operator, you would have to include the following two directives to perform the above example:

```
#ifdef max
#ifndef min
```

Using the #if Directive

The #if preprocessing directive has the form:

`#if`*`constant-expression`*

Use #if to test an expression. HP aC++ evaluates the expression in the directive. If the expression evaluates to a non-zero value (`TRUE`), the code following the directive is included. Otherwise, the expression evaluates to `FALSE` and HP aC++ ignores the code up to the next `#else`, `#endif`, or `#elif` directive.

All macro identifiers that appear in the constant-expression are replaced by their current replacement lists before the expression is evaluated. All defined expressions are replaced with either 1 or 0 depending on their operands.

The #endif Directive

Whichever directive you use to begin the condition (`#if`, `#ifdef`, or `#ifndef`), you must use `#endif` to end the if section.

Using the #ifdef and #ifndef Directives

The following preprocessing directives test for a definition:

`#ifdef identifier`

`#ifndef identifier`

These preprocessing directives behave like the `#if` directive, but `#ifdef` is considered true if the identifier was previously defined using a `#define` directive or the `-D` option. `#ifndef` is considered `TRUE` if the identifier is not defined yet.

Nesting Conditional Compilation Directives

Use the `#else` directive to specify an alternative section of code to be compiled if the `#if`, `#ifdef`, or `#ifndef` conditions fail. The code after the `#else` directive is included if the code following any of the `#if` directives is not included.

Using the #else Directive

Use the `#else` directive to specify an alternative section of code to be compiled if the `#if`, `#ifdef`, or `#ifndef` conditions fail. The code after the `#else` directive is included if the code following any of the `#if` directives is not included.

Using the #elif Directive

The `#elif` constant-expression directive tests whether a condition of the previous `#if`, `#ifdef`, or `#ifndef` was false. `#elif` has the same syntax as the `#if` directive and can be used in place of an `#else` directive to specify an alternative set of conditions.

Examples

The following examples show valid combinations of conditional compilation directives:

```
#ifdef SWITCH         // compiled if SWITCH is defined
#else                 // compiled if SWITCH is undefined
#endif                // end of if

#if defined(THING)    // compiled if THING is defined
#endif                // end of if

#if A>47              // compiled if A is greater than 47
#else
#if A < 20            // compiled if A is less than 20
#else                 // compiled if A is greater than or equal
                      // to 20 and less than or equal to 47
#endif                // end of if, A is less than 20
#endif                // end of if, A is greater than 47
```

Following are more examples showing conditional compilation directives:

```
#if (LARGE_MODEL)
#define INT_SIZE 32      // Defined to be 32 bits.
#elif defined (PC) && defined (SMALL_MODEL)
#define INT_SIZE 16      // Otherwise, if PC and SMALL_MODEL
                         // are defined, INT_SIZE is defined
                         // to be 16 bits.
#endif
#ifdef DEBUG             // If DEBUG is defined, display
cout << "table element : \n";  // the table elements.
for (i=0; i << MAX_TABLE_SIZE; ++i)
     cout << i << " " << table[i] << '\n';
#endif
```

Line Control (#line)

You can cause HP aC++ to set line numbers during compilation from a number specified in a line control directive. The resulting line numbers appear in error message references, but do not alter the line numbers of the actual source code.

Syntax

```
line-directive ::=
     #line digit-sequence [filename]
```

Description

The `#line` preprocessing directive causes HP aC++ to treat lines following it in the program as if the name of the source file were filename and the current line number were digit-sequence. This serves to control the file name and line number that are given in diagnostic messages. This feature is used primarily by preprocessor programs that generate C++ code. It enables them to force HP aC++ to produce diagnostic messages with respect to the source code that is input to the preprocessor rather than the C++ source code that is output.

HP aC++ defines two macros that you can use for error diagnostics. The first is `__LINE__`, an integer constant equal to the value of the current line number. The second is `__FILE__`, a quoted string literal equal to the name of the input source file. You can change `__FILE__` and `__LINE__` using `#include` or `#line` directives.

Example

```
#line 5 "myfile"
```

IOSTREAM Performance Improvement Pragma

The `-AA -D_HP_NONSTD_FAST_IOSTREAM` Performance Improvement macro can be used to improve the `-AA iostream` performance.

Syntax:

`#define _HP_NONSTD_FAST_IOSTREAM 1` (or)

`aCC options -D_HP_NONSTD_FAST_IOSTREAM`

This macro enables the following non-standard features:

- Sets `std::ios_base::sync_with_stdio(false)`, which disables the default synchronization with stdio
- Sets `std::cin.tie(0)`. which unties the cin from other streams.
- Replaces all occurrences of "`std::endl`" with "`\n`".

Enabling this macro can provide noticeable performance improvement if the application uses iostreams often.

NOTE: Note: Do not enable the HP_NONSTD_FAST_IOSTREAM macro in any of the following cases:.

- If the application assumes a C++ stream to be in sync with a C stream.
- If the application depends on stream flushing behavior with `endl`.
- If the user uses "`std::cout.unsetf(ios::unitbuf)`" to unit buffer the output stream.

Pragma Directive (#pragma) and _Pragma Operator

A `#pragma` directive is an instruction to the compiler. Use a pragma to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.

Syntax

`pragma-directive ::=`

`#pragma` [*`token-list`*]

Description

The `#pragma` directive is ignored by the preprocessor, and instead is passed on to the HP aC++ compiler. It provides implementation-dependent information to HP aC++. Any pragma that is not recognized by HP aC++ will generate a warning from the compiler.

The `_Pragma` operator, supported in non-strict C++98/C++03 mode and in all C++0x modes, has the effect of expanding the pragma specified in the string (in double-quotes) in just the way a `#pragma` directive would. For example:

```
_Pragma ("pack 1");
struct Packed {
char c;
int i;
};
int main () {
int iPackedSize = sizeof(Packed);
}
```

See Chapter 3 (page 96) for more information on pragmas.

Example

```
#pragma OPTIMIZE ON
```

Error Directive (#error)

The `#error` directive causes a diagnostic message, along with any included token arguments, to be produced by HP aC++.

Syntax

`error-directive ::=`

`#error [`*`preprocessor tokens`*`]`

Example

```
    // This directive will produce the diagnostic
    // message "FLAG not defined!".
#ifndef FLAG
#error "FLAG not defined!"
#endif

    // This directive will produce the diagnostic
    // message "TABLE_SIZE must be a multiple of 256!".
#if TABLE_SIZE % 256 != 0
#error "TABLE_SIZE must be a multiple of 256!"
#endif
```

Warning Directive

The `#warning` directive causes a diagnostic message, along with any included token arguments, to be produced by HP aC++.

Syntax

`warning-directive ::=`

`#warning [`*`preprocessor tokens`*`]`

Trigraph Sequences

The C++ source code character set is a superset of the ISO 646-1983 Invariant Code Set. To enable you to use only the reduced set, you can use trigraph sequences to represent those characters not in the reduced set.

A trigraph sequence is a set of three characters that is replaced by a corresponding single character. The preprocessor replaces all trigraph sequences with the corresponding character. The list below gives the complete list of trigraph sequences and their replacement characters. The following are all the trigraph sequences and their respective replacement characters:

- `??=` is replaced by #
- `??/` is replaced by \
- `??'` is replaced by ^
- `??(` is replaced by `[`
- `??)` is replaced by `]`
- `??!` is replaced by |
- ??< is replaced by {
- `??>` is replaced by }
- `??-` is replaced by ~

Examples

The line below contains the trigraph sequence ??=:

```
??=line 5 "myfile"
```

When this line is compiled it becomes:

```
#line 5 "myfile”
```

5 Using HP aC++ Templates

The following sections overview template processing and describe the instantiation coding methods available to you.

Invoking Compile-Time Instantiation

There are three methods of invoking compile-time instantiation:

- Explicit Instantiation (developer-directed)
- Command-Line Option Instantiation (developer-directed)
- Compile-Time Instantiation (default)

Scope and Precedence

Explicit instantiation provides instantiation for a particular template class or template function. While command line options and the default compile-time instantiation provide instantiation at the level of the translation unit.

If you use explicit instantiation in addition to command-line options or default instantiation, explicit instantiation takes precedence.

For example, using the `+inst_compiletime` option requests instantiation of all used template functions and all static data members and member functions of instantiated template classes within a translation unit. Using explicit instantiation requests instantiation of all members of a particular template class or a particular template function.

Template Processing

In HP aC++, compile-time instantiation is the default template instantiation mechanism. During compile-time instantiation, the compiler instantiates every template entity it sees in a translation unit provided it has the required template definition.

Following is the overview of template processing:During compile-time instantiation, the compiler instantiates every template entity it sees in a translation unit provided it has the required template definition

- The compiler places an instantiation in every `.o` file in which a template is used and its definition is known. The linker arbitrarily chooses a `.o` file to satisfy an instantiation request. Only the chosen instantiation appears in the `a.out` or `.so` file. Any redundant instantiations in other `.o` files are ignored.
- No instantiation information is placed in object (`.o`) files. The linker is responsible for ignoring duplicate instantiations.
- No `.I` files are created. All `.o` files are compiled only once.

Explicit Instantiation

You request explicit instantiation by using the explicit template instantiation syntax (as defined in the ANSI/ISO C++ International Standard) in your source file.

You can request explicit instantiation of a particular template class or a particular template function. In addition, member functions and static data members of class templates may be explicitly instantiated.

Explicit instantiation of a class instantiates all member functions and static data members of that class, regardless of whether or not they are used.

For example, following is a request to explicitly instantiate the `Table` template class with char*:

```
template class Table<char*>;
```

When you specify an explicit instantiation, you are asking the compiler to instantiate a template at the point of the explicit instantiation in the translation unit in which it occurs.

Usage

This might be useful when you are building a library for distribution and want to create a set of compiler-generated template specializations that you know will most commonly be used. Then when an application is linked with this library, any of these commonly used specializations need not be instantiated.

Another scenario might be a frequently used library that contains a repository of template specializations for your development team. Instantiating all such specializations in one, known translation unit would allow easy maintenance when changes are needed and eliminate cases of duplicate definition.

Performance

Although time is required to analyze and design code for explicit instantiation, compilation may be faster than for the equivalent implicit instantiation.

Examples

Following are the examples for explicit and implicit instantiation:

Class Template

Following are examples of explicit and implicit instantiation syntax for a class template:

```
template <class T> class Array;             // forward
                                            // declaration
                                            // for the
                                            // Array class
                                            // template


template <class T> class Array {/*...*/};   // definition
                                            // of the Array
                                            // class
                                            // template


template class Array <int>;                 // request to
                                            // explicitly
                                            // instantiate
                                            // Array<int>
                                            // template class


Array <char> tc;                            // use of
                                            // Array<char>
                                            // template
```

```
                                           // class which
                                           // results in
                                           // implicit
                                           // instantiation
```

Function Template

Following are examples of explicit and implicit instantiation syntax for a function template:

```
template <class T> void sort(Array<T> &);  // declaration
                                           // for the
                                           // sort()
                                           // function
                                           // template

template <class T> void sort(Array<T> &v) {/* ... */};
                                           // definition
                                           // of the
                                           // sort()
                                           // function
                                           // template

template void sort<char> (Array <char>&);  // request to
                                           // explicitly
                                           // instantiate
                                           // the
                                           // sort<char> ()
                                           // template
                                           // function


       //  NOTE <char> is not required if
       //  the compiler can deduce this.


void foo() {
Array <int> ai;
sort(ai);                 // use of the sort<int> ()
}                         // template function which
                          // results in implicit instantiation
```

NOTE: All template options on an aCC command-line apply to every file on the command line. If you specify more than one option on a command-line, only the last option takes effect.

For More Information, refer to the *ANSI/ISO C++ International Standard* for additional details including explicit specialization syntax.

Command-Line Option Instantiation

See "Template Options" (page 91) for more information on command-line instantiation.

Compile-Time Instantiation

By default, compile-time instantiation is in effect. Instantiation is attempted for any use of a template in the translation unit where the instantiation is used. All used template functions, all static data members and member functions of instantiated template classes, and all explicit instantiations are instantiated in the resulting object file.

If there are duplicate instantiations at link-time, the linker arbitrarily selects an instantiation for inclusion in the a.out or shared library.

The following command-lines are equivalent; each compiles `a.C` using compile-time instantiation.

```
aCC -c +inst_compiletime a.C
```

```
aCC -c a.C
```

Why Use Compile-Time Instantiation

Compile-time instantiation is the default. It is easy to use. Your code may compile faster when using compile-time instantiation.

If your development environment uses a version control system that is sensitive to file modifications, you may want to use the current default, compile-time instantiation, to avoid major code rebuilds.

NOTE: If you have used automatic instantiation with earlier versions of HP aC++ there may be some migration issues. See "Migrating from Automatic Instantiation to Compile-time Instantiation" (page 135) for more information.

Scope

If your source code contains templates and you do not specify any template command-line options nor explicit instantiations, compile-time instantiation takes place for any use of a template. If you specify a template command-line option, the option takes precedence for all translation units on the command line. Any explicit instantiation takes precedence over either a command-line option or compile-time instantiation.

Usage

Compared with developer-directed instantiation, compile-time instantiation involves less coding time for the developer. However, the design of your application may require the use of some form of directed instantiation.

Migrating from Automatic Instantiation to Compile-time Instantiation

If you have used automatic instantiation with earlier versions of HP aC++ there will be some known migration problems. The following migration problems may occur:

- Creating object files
- Creating an executable
- Closing a set of object files prior to creating a library (`.a` or `.so`)
- Creating a shared library (`.so`)

The following sections describe specific migration scenarios and illustrate possible migration problems and solutions:

Possible Duplicate Symbols in Shared Libraries

An existing compiler defect may be more apparent, if in HP aC++ A.02.00 or A.01.04 and prior versions you built a shared library using automatic instantiation (the prior default using the assigner) and now build that library using the current default (compile-time) instantiation. The defect relates to template objects with constructors or other runtime initializers that have been globally defined in more than one shared library on the link line. If such an object is defined in `n` shared libraries, it will be initialized and destructed `n` times at runtime.

When building the same application with the current default, the libraries are not closed prior to the final link, and the likelihood of a template symbol being defined in more than one shared library will increase.

Possible Duplicate Symbols in Archive Libraries

If you have built an archive library using automatic instantiation in HP aC++ A.02.00 or A.01.04 and prior versions, and you rebuild that library using the current (compile-time) instantiation, it is possible that duplicate symbol problems not apparent in the prior release will generate errors in the current release.

This is because the current default uses the linker, rather than the assigner, to determine which object file to pick to satisfy instantiation requests.

For example, when your archive library is linked with an application, library objects in the link may be different than those used when linking the library in a prior release.

Following are two examples of building an archive library; one built with `+inst_auto/+inst_close` (the prior default), and the other built with the current (compile-time) default:

Building an Archive Library with +inst_auto/+inst_close

Suppose for `lib.inst_auto.a`, the linker chooses `foo2.o` to resolve symbol `x`, and `foo3.o` to resolve symbol `stack <int>`, symbols `x`, `y`, and `stack <int>` are each resolved with no duplicates.

```
lib.inst_auto.a
-------------------------------------------------
| foo.o          | foo2.o         | foo3.o         |
|                |                |   stack<int>   |
|   x            |   x            |   y            |
|   y            |                |                |
-------------------------------------------------
```

Building an Archive Library with Compile-time Instantiation

Suppose for `lib.default.a`, the linker chooses `foo2.o` to resolve symbol `x`, and `foo.o` to resolve symbol `stack <int>`, symbols `x`, `y`, and `stack <int>` are each resolved, but now there's a duplicate definition of symbol `x`. This will cause a linker duplicate symbol error. This is really a user error, but was not visible before.

```
lib.default.a
-------------------------------------------------
| foo.o          | foo2.o         | foo3.o         |
|   stack<int>   |   stack<int>   |   stack<int>   |
|   x            |   x            |   y            |
|   y            |                |                |
-------------------------------------------------
```

NOTE: This example is not meant to account for all cases of changed behavior.

C++ Template Tutorial

You can create class templates and function templates. A template defines a group of classes or functions. A template can have one or more types as parameters. When you use a template, you provide the particular types or constant expressions as actual parameters thereby creating a particular object or function.

Class Templates

A class template defines a family of classes. To declare a class template, you use the keyword `template` followed by the template's formal parameters. Class templates can take parameters that are either types or expressions. You define a template class in terms of those parameters. For example, the following is a class template for a simple stack class. The template has two parameters, the type specifier `T` and the `int` parameter `size`. The keyword `class` in the `< >` brackets is required to declare any template type parameters. The first parameter `T` is used for the stack element type. The second parameter is used for the maximum size of the stack.

```
template<class T, int size>
class Stack
{
public:
     Stack(){top=-1;}
     void push(const T& item){thestack[++top]=item;}
     T& pop(){return thestack[top--];}
private:
     T thestack[size];
     int top;
};
```

Class template member functions and member data use the formal parameter type, `T`, and the formal parameter expression, `size`. When you declare an instance of the class `Stack`, you provide an actual type and a constant expression. The object created uses that type and value in place of `T` and `size`, respectively.

For example, the following program uses the `Stack` class template to create a stack of 20 integers by providing the type `int` and the value `20` in the object declaration.

```
int main()
{       Stack<int,20> myintstack;
        int i;

        myintstack.push(5);
        myintstack.push(56);
        myintstack.push(980);
        myintstack.push(1234);
        i = myintstack.pop();
}
```

The compiler automatically substitutes the parameters you specified, in this case `int` and `20`, in place of the template formal parameters. You can create other instances of this template using other built-in types as well as user-defined types.

Function Templates

A function template defines a family of functions. To declare a function template, use the keyword `template` to define the formal parameters, which are types, then define the function in terms of those types. For example, the following is a function template for a swap function. It simply swaps the values of its two arguments:

```
template<class T>
void swap(T& val1, T& val2)
{
        T temp=val1;
        val1=val2;
        val2=temp;
}
```

The argument types to the function template `swap` are not specified. Instead, the formal parameter, `T`, is a placeholder for the types. To use the function template to create an actual function instance (a template function), you simply call the function defined by the template and provide actual parameters. A version of the function with those parameter types is created (instantiated).

For example, the following main program calls the function `swap` twice, passing `int` parameters in the first case and `float` parameters in the second case. The compiler uses the `swap`template to automatically create two versions, or instances, of `swap`, one that takes `int` parameters and one that takes `float` parameters.

```
int main()
{       int i=2, j=9;
        swap(i,j);

        float f=2.2, g=9.9;
        swap(f,g);
}
```

Other versions of `swap` can be created with other types to exchange the values of the given type.

6 Standardizing Your Code

HP aC++ largely conforms to the ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++). This chapter discusses the following topics:

HP aC++ Keywords

HP aC++ supports the following list of keywords. Keywords cannot be abbreviated and must always be entered in lowercase letters.

Table 10 HP aC++ Keywords

• and	• friend	• static_cast
• and_eq	• inline	• template
• bitand	• mutable	• this
• bitor	• namespace	• throw
• bool	• new	• true
• catch	• not	• try
• class	• not_eq	• typeid
• compl	• operator	• typename
• const (also an ANSI C keyword)	• or	• using
• const_cast	• or_eq	• virtual
• delete	• private	• volatile (also an ANSI C keyword)
• dynamic_cast	• protected	• wchar_t
• explicit	• public	• xor
• false	• reinterpret_cast	• xor_eq

bool Keyword

The keyword `bool` represents a data type. Variables and expressions of type `bool` can have a value of either `true` or `false`. The value of `true` equals `1`. The value of `false` equals `0`.

Usage

The ANSI/ISO C++ International Standard states that values of type `bool` are either `true` or `false`. There are no signed, unsigned, short, or long bool types or values. bool values behave as integral types and participate in integral promotions. Types `bool`, `char`, `wchar_t`, and the signed and unsigned integer types are collectively called integral types. A synonym for integral type is integer type. The representations of integral types shall define values by use of a pure binary numeration system.

Example

```
int main(){
bool b=true; // Declare a variable of type bool and set it to true.
if (b)       // Test value of bool variable.
```

```
   b=false;  // Set it to false.
}
```

dynamic_cast Keyword

The keyword `dynamic_cast` is used in expressions to check the safety of a type cast at runtime. It is the simplest and most useful form of runtime type identification. You can use it to cast safely within a class hierarchy based on the runtime type of objects that are polymorphic types (classes including at least one virtual function). At runtime, the expression being cast is checked to verify that it points to an instance of the type being cast to.

Usage

A dynamic cast is most often used to cast from a base class pointer to a derived class pointer in order to invoke a function appearing only in the derived class. Virtual functions are preferred when their mechanism is sufficient. Usually a dynamic cast is necessary because the base class is being specialized, but cannot (or should not) be modified.

Example

```
class Base {
   virtual void f();       // Make Base a polymorphic type.
                           // other class details omitted
};

class Derived : public Base {
   // class details omitted
};

void Base::f()
{
   // define Base function
}

int main()
{
   Base *p;
   Derived *q;

   Base b;
   Derived d;

   p = &b;
   q = dynamic_cast<Derived *>  (p);   // Yields zero.

   p = &d;
   q = dynamic_cast<Derived *>  (p);   // Yields p treated
                                       // as a derived pointer.
}
```

Static and dynamic casts are used to move within a class hierarchy. Static casts use only static (compile-time) information to do the conversions. In the example above, if `p` is really pointing to an object of type `Derived`, either a static or dynamic cast of `p` to `q` yields the same result. This is also true if `p` were the null pointer. But, if `p` is not pointing to an object of type `Derived`, a dynamic cast returns zero, and a static cast returns a stray pointer. Dynamic casts must be done to a pointer or reference type. For example, if the cast above is written as:

```
q = dynamic_cast <Derived> (p);
```

The compile time error message is:

```
The result type of a dynamic cast must be a pointer or reference to a
complete class; the actual type was Derived.
```

If you attempt a dynamic cast from a non-polymorphic type, you will also get a compile-time error. For example:

```
class Base {
   // class details omitted
};

class Derived : public Base {
   // class details omitted
};

int main()
{
   Base *p;
   Derived *q;

   Base b;
   p = &b;
   q = dynamic_cast<Derived *> (p);
}
```

The above generates a compile-time error:

```
Dynamic down-casts and cross-casts must start from a polymorphic class
(one that contains or inherits a virtual function); but class Base is
not polymorphic.
```

The syntax of conditions allows declarations in them. For example:

```
class Base {
   virtual void f();      // Make Base a polymorphic type
   // other class details omitted
};

class Derived : public Base {
public:
   void derivedFunction();
   // other class details omitted
};

void Base::f()
{
   // Define Base function.
}

void Derived::derivedFunction()
{
}

int main()
{
   Base *p = new Derived;

   // details omitted

   if (Derived *q = dynamic_cast<Derived *>  (p))
      q->derivedFunction();          // use derived function
}
```

You can use dynamic casts with references as well. Since a reference cannot be zero, when the cast fails, it raises a `Bad_cast` exception. Before the implementation of the dynamic cast operator, you could not cast from a virtual base class to one of its derived classes because there was not enough information in the object at runtime to do this cast. Once runtime type identification was added, however, the information stored in a polymorphic virtual base class is sufficient to allow a dynamic cast from this base class to one of its derived classes. For example:

```
class Base1 {
   // Not a polymorphic type.
   // additional class details omitted
};

class Base2 {
   virtual void f();     // Make Base2 polymorphic.
                  // additional class details
                  // omitted
};

void Base2::f()
{
             // Define Base2 function.
}

class Derived : public virtual Base1, public virtual Base2 {
   // additional class details omitted
};

int main()
{
   Base1 *bp1;
   Base2 *bp2;
   Derived *dp;

   bp1 = new Derived;
   bp2 = new Derived;

   // dp = (Derived *) bp1;
                              // Problem: compile time error
                              // Can't cast from virtual base.


   // dp = (Derived *) bp2;
                             // Problem: compile time error
                             // Can't cast from virtual base.

   // dp = dynamic_cast<Derived *> bp1;

                            // Problem: compile time error
                            // Can't cast from
                            // non-polymorphic type.

   dp = dynamic_cast<Derived *> bp2;     // OK
}
```

explicit Keyword

explicit Keyword

The `explicit` keyword is used for declaring constructor functions within class declarations. When these functions are declared explicit, they cannot be used for implicit conversions.

Usage

While constructors taking one argument are often useful in the design of a class, they can allow inadvertent conversion in expressions. This can introduce subtle bugs. The `explicit` keyword allows a class designer to prohibit such implicit conversions. It is often used in the production of class libraries.

Example

```
class C {
public:
  explicit C(int);
};

C::C(int)
{
  // empty definition
}

int main()
{
  C c(5);       // Legal
  c = C(10);    // Legal
  // c = 15;    // Produces a compile time error:
                // Message: Cannot assign ‘C’ with ‘int’.
  // c + 20;    // Produces a compile time error
}
```

A classic example of this problem is an array class:

```
class Vector {
public:
  Vector(int n);           // create a vector of n items

  // other class details omitted

};

int main()
{
  Vector operator + (Vector, Vector);

  Vector v1(10), v2(10);  // create two 10 element vectors
  // details omitted
  v1 = v2 + 5;            // Legal - converts int 5 to a 5
                          // element vector and adds to v2.
                          // Not something you want to be
                          // legal
}
```

With the explicit keyword, the constructor can be made explicit and the declarations are legal, but the addition is a compilation error:

```
class Vector {
public:
  explicit Vector(int n);  // create a vector of n items

  // other class details omitted
};

int  main()
{
  Vector operator + (Vector, Vector);

  Vector v1(10), v2(10); // create two 10 element vectors

  // details omitted

  // v1 = v2 + 5;    // Not legal - generates compile-
                     // time error
                     // Message: Illegal typEs
                     // associated with operator ‘+’:
                     // ‘Vector’ and ‘int’.
}
```

mutable Keyword

The `mutable` keyword is used in declarations of class members. It allows certain members of constant objects to be modified in spite of the *`const`* of the containing object.

Usage

Often some class members are part of the implementation of the object, not part of the actual information stored by the object. Although the information in the object needs to stay unmodified in a const object, the implementation members may need to change. These are declared mutable.

An example of this is use or reference count in an object that keeps track of the number of pointers referring to it.

Example

```
class C {
public:
  C();
  int i;
  mutable int j;
};

C::C() : i(1), j(3)
{
  // Define constructor
}

int main()
{
  const C c1;
  C c2;

  // c1.i =0;      // Problem: compilation error
                   // Message: The left side of '='
                   // must be a modifiable lvalue.
  c1.j = 1;        // OK
  c2.i = 2;        // OK
  c2.j = 3;        // OK
}
```

The mutable keyword can only be used on class data members. It cannot be used for const or static data members. Notice the difference in the two pointer declarations below:

```
class C {
  C() { }                   // define constructor

  mutable const int *p;     // OK
                            // mutable pointer to int const
                            // p in constant C object
                            // can be modified

  mutable int *const q;     // Compile time error
                            // mutable const pointer to int
                            // const data member can't be
                            //  mutable
                            // Message: 'mutable' may be
                            // used only in non-static
                            // and non-constant data
                            // member declarations within
                            // class declarations
};
```

namespace and using Keywords

Namespaces were introduced into C++ primarily as a mechanism to avoid naming conflicts between various libraries. The following example illustrates how this is achieved:Every namespace introduces a new scope. By default, names inside a namespace are hidden from enclosing scopes. Selection of a particular name can be achieved using the qualified-name syntax. Namespaces can be nested very much like classes.

```
#include <stdio.h>

namespace N {
 struct Object {
  virtual char const* name() const { return “Object from N”; }
   };
}


namespace M {
 struct Object {
  virtual char const* name() const { return “Object from M”; }
   };

  namespace X {    // a nested namespace
      struct Object: M::Object {  // inherit from a class
                                  // in the outer space
       char const* name() const { return “Object from M::X”; }
      };
   }
}


int main() {
   N::Object o1;
   M::Object o2;
   M::X::Object o3;
   printf(“This object is: %s.\n”, o1.name());
   printf(“This object is: %s.\n”, o2.name());
   printf(“This object is: %s.\n”, o3.name());
   return 0;
}
```

Connections Across Translation Units

If a type, function, or object is declared inside of a namespace, then using that entity will require naming this namespace in some explicit or implicit way; even if the use happens in another translation unit (or source file).

A unique feature of namespaces is that they can be extended. The following example shows this; as well as the connections between a namespace extending across different translation units.

The example also illustrates the concept of unnamed namespaces. These namespaces can only be extended within a translation unit. Unnamed namespaces in different translation units are unrelated; hence their names effectively have internal linkage. In fact, the ANSI/ISO C++ International Standard specifies that using static to indicate internal linkage is deprecated in favor of using namespaces.

```
#include <stdio.h>

namespace N {
   char const* f() { return “f()”; }
}

namespace {      // An unnamed namespace
   char const* f(double);
} // Names in unnamed namespaces are visible in their surrounding scope.
  // They cannot be qualified since the space has no name.
```

```
namespace N {    // An extension of the first part of namespace N
   char const* f(int); // Leave the implementation to another
}                      // translation unit.

int main() {
   printf("Calling: %s.\n", N::f());  // OK, declared and defined above
   printf("Calling: %s.\n", N::f(7)); // OK, declared above (defined elsewhere)
   printf("Calling: %s.\n", f(3.0));  // OK, declared above (defined below)
   return 0;
}

namespace { // An extension of the unnamed namespace in this translation unit
   char const* f(double) { return "f(double) in main() translation unit"; }
}
```

An Auxiliary Translation Unit

Following is an auxiliary translation unit that illustrates how namespaces interact across translation units.

```
namespace { // An unnamed namespace unrelated to the
            // one in the other translation units.
 char const* f(double) { return "f(double) in auxiliary translation unit"; }
}


namespace N { // This namespace is the same as the
              // one in the main() translation unit.
              // We implement f(int) here.
   char const* f(int) { return "f(int) defined in auxiliary translation unit"; }
}
```

using- declarations and using- directives

C++ provides two alternatives to explicitly qualifying names in namespaces. These are the *using*- declaration and the *using*- directive.

using- declaration

A `using-` declaration introduces a declaration in the current scope as follows:

```
using N::x; // Where N is a namespace, x is a name in N
```

After this declaration, all uses of `x` in this scope are taken to defer to `N::x`. (The `N::` prefix is no longer required.)

If another declaration of x were introduced in the same scope, for example:

```
int x;
```

then a compiler error occurs.

using- directive

The `using-` directive directs the lookup for names not declared in current scope, for example:

```
using namespace N; // If not found, lookup names in namespace N
```

If `x` is a name in namespace `N`, but another declaration of `x` is present in the current scope, for example:

```
int x;
```

a compiler error is not necessarily emitted. Only if that name is used will an ambiguity occur.

NOTE: `Using-` directives are transitive. If you specify a `using-` directiveto one namespace which itself specifies a directive to another namespace, then names used in your scope will also be looked up in that other namespace.

Using namespace directives can be a powerful means to migrate code to libraries that use namespaces. Occasionally, however, they may silently make unwanted names visible. It is therefore often suggested not to use using-directives unless the alternatives are very inconvenient.

```
#include <stdio.h>

namespace N {
   char const* f() { return "N::f()"; }
   char const* f(double) { return "N::f(double)"; }
   char const* g() { return "N::g()"; }
}

char const* g(double) {
   using N::f;            // Declare all f's in namespace N
   return f(2.0);
}

namespace M {             // Illustrate how using-directives
   using namespace N;     // are transitive
}

int main() {
   using namespace N;
   printf("Calling: %s.\n", f());         // calls N::f()
   printf("Calling: %s.\n", g(1.0));      // calls ::g(double)
                                          // which calls
                                          // N::f(double)
   printf("Calling: %s.\n", N::g());      // calls N::g()
   printf("Calling: %s.\n", M::f());      // calls N::f()
   return 0;
}
```

typeid Keyword

The `typeid` keyword is an operator, called the type identification operator, used to access type information at runtime. The operator takes either a type name or an expression and returns a reference to an instance of `type_info`, a standard library class.

Usage

You can use runtime type identification when you need to know the exact type of an object. This might, for example, be necessary to find the name of the object class for diagnostic output. It also might be used to perform some standard service on an object such as via a database or I/O system.

typeid Example

Following is an example of the `typeid` keyword:

```
# include <iostream.h>
# include <typeinfo>

class Base {
   virtual void f();     // Must have a virtual function
                         // to be a polymorphic type.
   // additional class details omitted
};

class Derived : public Base {

   // class details omitted
};

void Base::f()
{
   // Define function from Base.
}

int main ()
{
   Base *p;

   // Code which does either
   //        p = new Base; or
   //        p = new Derived;

   // Note that this is NOT a good design for this
   // functionality Virtual functions would be better.

   if (typeid(*p) == typeid(Base))
      cout << "Base Object\n";
   else if (typeid(*p) == typeid(Derived))
      cout << "Derived Object\n";
   else
   cout << "Another Kind of Object\n";
}
```

If a `typeid` operation is performed on an expression that is not a polymorphic type (a class which declares or inherits a virtual function), the operation returns the static (compile-time) type of the expression. In the example above, if class Base did not include the virtual function `f`, `typeid(p)` would always yield the type `Base`. The style of programming used in the above example is called a typeid switch statement. It is not recommended. One alternative is to use a virtual function in a base class specialized in each of its derived classes. In some cases, this may not be possible, for example, when the base class is provided by a library for which source code is not available. In other cases it may not be desirable, for example, some base class interfaces might be too big if all derived class functionality is included.

You can rewrite the previous example, using virtual functions, as:

```
class Base {
   virtual void outputType() { cout << "Base Object\n"; }

   // additional class details omitted
};

class Derived : public Base {
   virtual void outputType() { cout << "Derived Object\n"; }
// additional class details omitted
};

int main ()
{
   Base *p;

   // code which does either
   //        p = new Base; or
   //        p = new Derived;

   p->outputType();
}
```

A second alternative is to use a dynamic cast. In many cases, this alternative is less desirable than using virtual functions, but it is better than a typeid switch statement in nearly every case. There is a subtle difference between this alternative and the typeid switch statement above. The typeid operation allows access to the exact type of an object; a dynamic cast returns a non-zero result for the target type or a type publicly derived from it.

You can rewrite the previous example as follows using dynamic casts:

```
class Base {
    virtual void f();     // Must have a virtual function to
```

```
                              // be a polymorphic type.
    // additional class details omitted
};

class Derived : public Base {

   // class details omitted
};

void Base::f()
{
    // Define function from Base.
}

int main ()
{
    Base *p;

    // code which does either
    //       p = new Base; or
    //       p = new Derived;

    if (dynamic_cast <Derived *> (p))
   cout << "Derived (or class derived from Derived) Object\n";
    else
   cout << "Base Object\n";
}
```

volatile Keyword

The `volatile` keyword is used in declarations. It tells the compiler not to do aggressive optimization because a value might be changed in ways the compiler cannot detect.

This keyword is part of the ANSI C standard with the same syntax and semantics.

Usage

Objects that are hardware addresses or those used by concurrently executing pieces of code are frequently declared volatile. Examples are an address used for the current clock time, objects used by a signal handler, or objects used for memory mapped I/O.

NOTE: You can declare an identifier to be both `const` and `volatile`. This declares a value that the program cannot change but which can be changed by some means external to the program (such as by a piece of hardware like a clock).

Example

```
class C {
public:                  // public to make example simpler
volatile int i;
// other class details omitted
};

C someData[10];

int main ()
{
   int j = someData[5].i;
   j = someData[5].i;   // Without the volatile specifier,
                        // the compiler could optimize these
                        // two statements into one. With it,
                        // it must execute both in case the
                        // i field of someData[5] has changed
                        // by some other means.
}
```

wchar_t Keyword

Wide (or multi-byte) characters can be declared with the data type `wchar_t`. It is an integral type that can represent all the codes of the largest character set among the supported locales defined in the localization library. This keyword was a typedef of the ANSI C standard.

Usage

This type was added to maintain ANSI C compatibility and to accomodate foreign (principally Oriental) character sets.

Example

In the following example, literals of type `wchar_t` consist of the character L followed by a character constant in single quotes.

```
int main()
{
  wchar_t ch = L'a';
}
```

`wchar_t` must be implemented the same as another integral type. In other words, it must have the same size, signedness and alignment requirements. It promotes to the smallest integral type when used in an expression and cannot have a signed or unsigned modifier.

The standard library includes a string of wide characters known as wstring. The IOStream library supports I/O of wide characters.

In ANSI C, `wchar_t` is a synonym for another type, declared using a `typedef` in a standard header file.

template Keyword

Use the `template` keyword when calling a member template to specify that a `name` is a member template.

Usage

This construct is used for function calls to indicate that the `name` is a member template.

Example

```
struct S {
   template <class T> void foo() {}
};
template <class T>
void sam(T x, S y) { y.template foo<T>(); }
```

typename Keyword

Use the `typename` keyword in template declarations to specify that a qualified name is a type, not a class member.

Usage

This construct is used to access a nested class in the template parameter class as a type in a declaration within the template.

Example

```
template<class T>
class C1 {
  // class details omitted

  // T::C2 *p;       // Problem: flagged as compile-time
                     // error. T is a type, but T::C2 is not.
```

```
                          // Message: 'C2' is used as a type, but
                          // has not been defined as a type.


typename T::C2 *p;        // Solution: the keyword typename flags
                          // the qualified name T::C2 as a type.
};

class C {
  // details omitted
  class C2 {
    //details omitted
  };
};

int main ()
{
  C1<C> c;
}
```

In a template, a name is not taken to be a type unless it is explicitly declared as one. Ways to declare a name as a type include:

- Use it as the argument to the template (T below):

  ```
  template<class T>
  class C {
    // Additional details omitted
  };
  ```

- Use it as the name of the template (C below):

  ```
  template<class T>
  class C {
    // Additional details omitted
  };
  ```

- Declare a class as a member of the class template (C2 below):

  ```
  template<class T>
  class C1 {
    class C2;
    // Additional details omitted
  };
  ```

- Declare a class in the context the template is declared within (C1 below):

  ```
  class C1;
  template<class T>
  class C2 {
    // details omitted
  };
  ```

Overloading new[] and delete[] for Arrays

HP aC++ defines `new` and `delete` operators for arrays that are different from those used for single objects. These operators, operator `new[ ] ( )` and operator `delete[ ] ( )`, can be overloaded both globally, and in a class. If you use operator `new( )` to allocate memory for a single object, you should use operator `delete( )` to deallocate this memory. If you use operator `new[ ] ( )` to allocate an array, you should use operator `delete[ ] ( )` to deallocate it.

Usually, the allocation and deallocation of operators is overloaded for a particular class, not globally. This overloading allows you to put all instances of a particular class on a class-specific heap. You can then take control of allocation either for efficiency or to accomplish other storage management functions, for example garbage collection. If allocation and deallocation of single objects is overloaded, you may or may not want to overload the operators for arrays. If the overloading was done for efficiency, it may be that for arrays the default operator is the most efficient.

Example

```
# include <iostream.h>
class C {
  public:
   void* operator new[ ] (size_t);    // new for arrays
   void operator delete[ ] (void*);   // delete for arrays

    // additional class details omitted
};

void* C::operator new[ ] (size_t  allocSize)
{
  cout << "Use operator new[ ] from class C\n";

    // here, real usage would include allocation

return ::operator new[ ] (allocSize); // global operator
}                                     // for this simple
                                      // example
void C::operator delete[ ] (void *p)
{
  cout << "Use operator delete[ ] from class C\n";

    // here, real usage would include deallocation

  ::operator delete[ ] (p);          // global operator
}                                    // for this simple
                                     // example
int main()
{
  C *p;

  p = new C[10];
  delete[ ] p;
}
```

Notice that the new operator takes a class with an array specifier as an argument. The compiler uses the class and array dimension to provide the `size_t` argument. In the example above, the argument provided is ten times the size of a class C object. Also, the operator must return a `void*` which the compiler converts to the class type. The void constructor for the class (if one exists) is invoked to initialize the elements in the array.

Multidimensional arrays can be allocated and deallocated with these operators. The operator is used with several array dimensions, and the compiler provides the `size_t` argument which is the space required for the entire array. For example:

```
// call C::operator new[ ] ( ) with
// an argument of 10 * 20 * sizeof(C)

p = new C [10] [20];
```

Additional arguments can be provided to this operator new just as for the operator for single objects. In this way, the operator can be overloaded in a class. The additional arguments can be used by the storage allocation scheme for additional storage management.

The global `new` and `delete` for both arrays and single objects are provided in the Standard C++ Library. This library also provides a version of new for arrays and single objects that takes a second `void*` argument and constructs the object at that address.

Standard Exception Classes

Classes are provided in the Standard C++ Library to report program errors. These classes are declared in the `<stdexcept>` header. All of these classes inherit from a common base class named exception. The two classes `logic_error` and `runtime_error` inherit from exception and serve as base classes for more specific errors.

These classes provide a common framework for the way errors are handled in a C++ program. System-specific error handling can be provided by creating classes that inherit from these standard exception classes.

Example

```
# include <stdexcept>
# include <iostream>
# include <string>
void f()
{
  // details omitted

  throw range_error(string("some info"));
}

int main()
{
  try {
    f();
  }
  catch (runtime_error& r) {
    // handle any kind of runtime error including range_error
    cout << r.what() << '\n';
  }
}
```

The class `logic_error` defines objects thrown as exceptions to report errors due to the internal logic of the program. The errors are presumably preventable and detectable before execution. Examples are violations of logical preconditions or class invariants. The subclasses of `logic_error` are:

- `domain_error` (the operation requested is inconsistent with the state of the object it is applied to)
- `invalid_argument`
- `length_error` (an attempt to create an object whose size equals or exceeds allowed size)
- `out_of_range` (an argument value not in the expected range)

Runtime errors are due to events out of the scope of the program. They cannot be predicted before they happen. The subclasses of `runtime_error` are:

- `range_error`
- `overflow_error` (arithmetic overflow)

The exception class includes a void constructor, a copy constructor, an assignment operator, a virtual destructor, and a function `what` that returns an implementation-defined character string. None of these functions throw any exceptions.

Each of the subclasses includes a constructor taking an instance of the Standard C++ Library string class as an argument. They initialize an instance such that the function `what`, when applied to the instance, returns a value equal to the argument to the constructor.

Exceptions Thrown by the Standard C++ Library

The following exceptions are thrown by the Standard C++ Library:

- `operator new ()` and `operator new [ ]` throw a `bad_alloc` exception when they cannot obtain a block of storage.
- A `dynamic_cast` expression throws a `bad_cast` exception when a cast to a reference type fails.
- Operator `typeid` throws a `bad_type` exception when a pointer to a `typeid` expression is zero.
- A `bad_exception` exception can be thrown when the unexpected handler function is invoked by `unexpected()`.

NOTE: If no `catch` clauses are available to catch these exceptions, the default action is program termination with a call to `abort()`. (Using the `+noeh` option does not disable the exceptions thrown by these library functions.)

type_info Class

`type_info` is a class in the standard header file `<typeinfo>`. A reference to an instance of this class is returned by the `typeid` operation.

Implementations may differ in the exact details of this class, but in all cases it is a polymorphic type (has virtual functions) that allows comparisons and a way to access the name of the type.

Usage:

This class is useful for diagnostic information and for implementing services on objects where it is necessary to know the exact type of the object.

Example:

```
# include <iostream.h>
# include <typeinfo>

class Base {
   virtual void f();           // Must have a virtual
                               // function to be a
                               // polymorphic type

   // additional class details omitted

};

class Derived : public Base {
   // class details omitted
};

void Base::f()
{
// Define function from Base.
}

int main ()
{
   Base *p;

   // code which does either
   //       p = new Base; or
   //       p = new Derived;

   if (typeid(*p) == typeid(Base))        // Standard requires
                                          // comparison as part
```

```
                                              // of this class.
        cout << "Base Object\n";

     cout << typeid(*p).name() << '\n';       // Standard requires
                                              // access to the name
                                              // of the type.
}
```

The standard requires the class `type_info` to be polymorphic. You cannot assign or copy instances of the class (the copy constructor and assignment operators are private). The interface must include:

```
int operator == (const type_info&) const
int operator !=( const type_info&) const
const char * name() const
int before (const type_info&) const
```

The operators allow comparison of object types. The `name` function allows access to the character string representing the name of the object. The `before` function allows types to be sorted. This allows them to be accessed through hash tables. The `before` function is not a lexical ordering; it might not yield the same results. The `name` function now returns the mangled name of a type as per the C++ ABI.

Unsupported Functionality

Functionality defined in the ANSI/ISO C++ International Standard and not supported in this release of HP aC++ is listed in Table 11. Library functionality is listed separately.

Table 11 Unsupported Functionality

Functionality	**Rogue Wave Standard C++ Library 2.2.1**	**Rogue Wave Standard C++ Library 1.2.1**	**libc **** **(HP-UX System Libraries)**
`<allocator>`	Yes	Provided as a class rather than a template.	Not Applicable

`<cstring>`	Yes	The following C++ overloaded functions are not provided.
		Instead, ANSI C signatures are implemented.
		`memchr`
		`strchr`
		`strpbrk`
		`strrchr`
		`strstr`
			Not Applicable

`<cwchar>`	Yes	The following C++ overloaded functions are not provided, instead, ANSI C signatures are implemented.
		`wcschr`
		`wcspbrk`
		`wcsrchr`
		Missing Functions:
		`wcsstr`
		`wmemchr`
			For missing functions, see wide character support in this table.

`<cwctype>`	Partial Support	Partial Support	See wide character support in this table.

`<functional>`	Yes	The following types are not provided:
		`mem_fun_t`
		`mem_fun1_t`
			Not Applicable

Table 11 Unsupported Functionality *(continued)*

Functionality	Rogue Wave Standard C++ Library 2.2.1	Rogue Wave Standard C++ Library 1.2.1	libc ** (HP-UX System Libraries)
		`mem_fun1_ref_t` `mem_fun_ref_t`	

`<iostream>`	Yes	Not templatized and the following headers are not provided:
		`<fstream>`
		`<iostream>`
		`<istream>`
		`<ostream>`
		`<streambuf>`
		`<sstream>`
		`<iomanip>`
		`<ios>`
		`<iosfwd>`
			Not Applicable

`<iterator>`	Yes	iterator template is not provided	Not applicable
`<locale>`	Yes	Not provided	Not applicable
`printf(3)` formats	Yes	Not provided	Not applicable
`<utility>`	Not applicable	Not applicable	`%ls` and `%lc` are not provided
`<valarray>`	Yes	Not provided	Not applicable

wide character support	Not applicable	Not applicable	The following functions are not provided:
			`btowc`
			`fwide`
			`fwprintf`
			`fwscanf`
			`mbrlen`
			`mbrtowc`
			`mbsinit`
			`mbsrtowcs`
			`swprintf`
			`swscanf`
			`towctrans`
			`vfwprintf`
			`vswprintf`
			`vwprintf`
			`wcrtomb`
			`wcsrtombs`
			`wcsstr`
			`wctob`
			`wctrans`
			`wmemchr`
			`wmemcmp`
			`wmemcpy`
			`wmemset`
			`wprintf`
			`wscanf`

** Available when compiled with`-D_XOPEN_SOURCE=500`. Also, the application must be linked with `/usr/lib/hpux##/unix98.o`, where `##` is either `32` or `64`.

7 Optimizing HP aC++ Programs

HP C/HP aC++ provides options to the `aCC` command and pragmas to control optimization. The following sections introduce the basic concepts of optimizing your HP aC++ code for improved efficiency:

Requesting Optimization

By default, the compiler performs constant folding and simple register assignment. There are several ways to increase and control the level of optimization performed on your program.

Setting Basic Optimization Levels

HP aC++ provides four basic levels of optimization, the higher the level the more optimization performed and the longer the optimization takes.

You can specify an option on the aCC command line or in the `CXXOPTS` environment variable.

Example:

```
aCC -O prog.C
```

Compiles `prog.C` and optimizes the program at the default level 1.

Level 1 Optimization

Level 1 optimization includes branch optimization, dead code elimination, faster register allocation, instruction scheduling, and peephole (statement-by-statement) optimization. Use `+O1` to get level 1 optimization. Level 1 is the default.

Level 1 optimization produces faster programs than without optimization and compiles faster than level 2 optimization. Programs compiled at level 1 can be used with the HP Distributed Debugging Environment (DDE) debugger. Use the debugger option `-g0` or `-g1`.

Level 2 Optimization

Level 2 optimization includes level 1 optimization, along with optimizations performed over entire functions in a single file. Level 2 optimizes loops in order to reduce pipeline stalls and analyzes data-flow, memory usage, loops, and expressions. Use `-O` or `+O2` to get level 2 optimization.

Specifically, level 2 provides the following:

- Coloring register allocation.
- Induction variable elimination and strength reduction.
- Local and global common subexpression elimination.
- Advanced constant folding and propagation. (Simple constant folding is done by default.)
- Loop invariant code motion.
- Store/copy optimization.
- Unused definition elimination.

- Software pipelining.
- Register reassociation.

Level 2 can produce faster runtime code than level 1 if programs use loops extensively. Loop-oriented floating-point intensive applications may see run times reduced by 50%.

Operating system and interactive applications that use the already optimized system libraries can achieve 30% to 50% additional improvement. Level 2 optimization produces faster programs than level 1 and compiles faster than level 3 optimization.

Level 3 Optimization

Level 3 optimization includes level 2 optimizations, along with full optimization across all subprograms within a single file. Level 3 also inlines certain subprograms within the input file. Use `+O3` to get level 3 optimization.

Level 3 optimization produces faster runtime code than level 2 on code that does many procedure calls to small functions. Level 3 links faster than level 4. But level 3 does not work with the debugger options `-g0` and `-g1`.

Level 4 Optimization

Level 4 optimization includes level 3 optimizations, along with full optimizations across the entire application program. Level 4 includes global and static variable optimization and inlining across the entire program. Optimizations are performed at link time rather than at compile time. Use `+O4` to get level 4 optimization.

Level 4 optimization produces faster runtime code than level 3 if programs use many global variables or if there are many opportunities for inlining procedure calls. But level 4 does not work with the debugger options `-g0` and `-g1`.

Additional Options for Finer Control

In addition to basic optimization levels, optimization options are provided should you require a more precise level of control.

Some introductory examples follow:

Enabling Aggressive Optimizations

To enable aggressive optimizations at the second, third, or fourth optimization levels, use the `+Ofast` option as follows:

```
aCC +Ofast +O2 sourcefile.C
```

or:

```
aCC +Ofast +O3 sourcefile.C
```

or:

```
aCC +Ofast +O4 sourcefile.C
```

This option enables additional optimizations at each level.

NOTE: Use aggressive optimizations with stable, well-structured code. These types of optimizations give you faster code, but may change the behavior of programs.

These optimizations may do any of the following:

- Relocate conditional floating-point instructions from within loops
- Convert certain library calls to millicode and inline instructions
- Alter error-handling requirements

Enabling Only Conservative Optimizations

You can enable only conservative optimizations at the second, third, or fourth optimization levels by using the `+Ofltacc=strict +Ofenvaccess` option, as follows:

```
aCC +O2 +Ofltacc=strict +Ofenvaccess sourcefile.C
```

or:

```
aCC +O3 +Ofltacc=strict +Ofenvaccess sourcefile.C
```

or:

```
aCC +O4 +Ofltacc=strict +Ofenvaccess sourcefile.C
```

This option disables all but the most conservative optimizations at each level. Conservative optimizations do not change the behavior of code, in most cases, even if the code does not conform to standards.

Use only conservative optimizations provided with level 2, 3, and 4 when your code is unstructured.

Removing Compilation Time Limits When Optimizing

You can remove optimization time restrictions at the second, third, or fourth optimization levels by using the `+Onolimit` option as follows:

```
aCC +O2 +Onolimit sourcefile.C
```

or:

```
aCC +O3 +Onolimit sourcefile.C
```

or:

```
aCC +O4 +Onolimit sourcefile.C
```

By default, the optimizer limits the amount of time spent optimizing large programs at levels 2, 3, and 4. Use this option if longer compile times are acceptable because you want additional optimizations to be performed.

Limiting the Size of Optimized Code

You can disable optimizations that expand code size at the second, third, and fourth optimization levels by using the `+Osize` suboption, as follows:

```
aCC +O2 +Osize sourcefile.C
```

or:

```
aCC +O3 +Osize sourcefile.C
```

or:

```
aCC +O4 +Osize sourcefile.C
```

Most optimizations improve execution speed and decrease executable code size. A few optimizations significantly increase code size to gain execution speed. The `+Osize` option disables these code-expanding optimizations.

Use this option if you have limited main memory, swap space, or disk space.

Combining Optimization Options

Optimization options that affect code size, (`+Osize`), compile-time (`+Olimit`), and the aggressiveness of the optimizations performed can be combined at any of the optimization levels 2 through 4.

Profile-Based Optimization

Profile-based optimization (PBO) is a set of performance-improving code transformations based on the runtime characteristics of your application.

When using profile-based optimization, please note the following:

- Numerical applications that perform the same calculations independent of the input data will only see a small performance boost.
- Profile-based optimization has the greatest impact on application performance when used with level 2 or greater optimizations.
- Profile-based optimization benefits most applications, especially large applications with multiple compilation units, such as compilers, editors, database managers, and user interface managers.
- Profile-based optimization should be enabled during the final stages of application development. To obtain the best performance, reprofile and reoptimize your application after making source code changes.

These steps are involved in performing profile-based optimization:

1. Instrumentation
2. Collecting Data for Profiling
3. Maintaining Profile Data Files
4. Performing Profile-Based Optimization

Instrumentation

To instrument your program, use the `+Oprofile=collect` option as follows:

```
aCC +Oprofile=collect -O -c sample.C
```

```
aCC +Oprofile=collect -O -o sample.exe sample.o
```

The first command line uses the `+Oprofile=collect` option to prepare the code for instrumentation. The `-c` option in the first command line suppresses linking and creates an object file called `sample.o`.

The second command line uses the `-o` option to link `sample.o` into `sample.exe`. The `+Oprofile=collect` option instruments `sample.exe` with data collection code.

NOTE: Instrumented programs run slower than non-instrumented programs. Only use instrumented code to collect statistics for profile-based optimization.

Collecting Data for Profiling

To collect execution profile statistics, run your instrumented program with representative data as follows:

```
sample.exe < input.file1
```

```
sample.exe < input.file2
```

This step creates and logs the profile statistics to a file, by default called `flow.data`. The data collection file is a structured file that may be used to store the statistics from multiple test runs of different programs that you may have instrumented.

Maintaining Profile Data Files

Profile-based optimization stores execution profile data in a disk file. By default, this file is called `flow.data` and is located in your current working directory.

You can override the default name of the profile data file. This is useful when working on large programs or on projects with many different program files.

The `FLOW_DATA` environment variable can be used to specify the name of the profile data file with either the `+Oprofile=collect` or `+Oprofile=use` options.

The `+Oprofile=use:`*`filename`* command line option can be used to specify the name of the profile data file when used with the `+Oprofile=use` option.

Example 1

In the following example, the `FLOW_DATA` environment variable is used to override the `flow.data` file name. The profile data is stored instead in `/users/profiles/prog.data`.

```
export FLOW_DATA=/users/profiles/prog.data
aCC -c +Oprofile=collect sample.C
aCC -o sample.exe +Oprofile=collect sample.o
sample.exe < input.file1
aCC -o sample.exe +Oprofile=use +O3 sample.C
```

Example 2

In this example, the `+Oprofile=use:filename` option is used to override the `flow.data` file name with the name `/users/profiles/prog.data`.

```
aCC -c +Oprofile=collect +O3 sample.C
aCC -o sample.exe +Oprofile=collect sample.o
sample.exe < input.file1
mv flow.data /users/profile/prog.data
aCC -o sample.exe +Oprofile=use:/users/profiles/prog.data +O3 sample.C
```

Performing Profile-Based Optimization

To optimize the program based on the previously collected runtime profile statistics, recompile the program as follows:

```
aCC -o sample.exe +Oprofile=use +O3 sample.C
```

For more information on profile-based optimization, refer to the *HP-UX Online Linker and Libraries User’s Guide*.

Pragmas That Control Optimization

Compiler options provide a high-level, global approach to optimization. To give you more refinement in optimization, HP aC++ provides pragma `OPT_LEVEL`.

See “Optimization Pragmas” (page 103) for more information.

8 Exception Handling

Exception handling provides a standard mechanism for coding responses to runtime errors or exceptions.

This chapter discusses the following topics:

Exception Handling

Exception handling provides a standard mechanism for coding responses to runtime errors or exceptions. Exception handling is on by default. To turn it off, you must use the `+noeh` option.

If your executable throws no exceptions, object files compiled with and without the +noeh option can be mixed freely. However, in an executable which throws exceptions (HP aC++ runtime libraries throw exceptions), you must be certain that no exception is thrown in your application which will unwind through a function compiled without the exception handling option turned on.

In order to prevent this, the call graph for the program must never have calls from functions compiled without exception handling to functions compiled with exception handling (either direct calls or calls made through a callback mechanism). If such calls do exist, and an exception is thrown, the unwinding can cause:

- Non-destruction of local objects (including compiler generated temporaries).
- Memory leaks when destructors are not executed.
- Runtime errors when no catch clause is found.

Exception Handling in C++

Following is an overview of the elements of C++ exception handling:

- A try block encloses (logically) code that can cause an exception that you want to catch.
- A catch clause, which immediately follows the try block, handles an exception of the type that can occur in the try block. The catch clause is the exception handler. You can have multiple catch clauses associated with a try block.
- If an error occurs, code in the try block throws an exception to an appropriate catch clause. The catch clause is ignored if an error does not occur.
- When an exception is thrown, control is transferred to the nearest handler defined to handle that type of exception. Nearest means the handler whose try block was most recently entered by the thread of control, and not yet exited.

Exception Handling as Defined by the ANSI/ISO C++ International Standard

The Standard C++ Library provides classes that C++ programs can use for reporting errors. These classes are defined in the header file `<stdexcept>` and described in the ANSI/ISO C++ International Standard.

- The class, `exception,` is the base class for object types thrown by the Standard C++ Library components and certain expressions.
- The class, `runtime_error`, defines errors due to events beyond the scope of the program.
- The class, `logic_error`, defines errors in the internal logic of the program.

Basic Exception Handling Example

The simple program shown here illustrates exception handling concepts. This program contains a try block (from which a range error is thrown) and a catch clause, which prints the operand of the throw.

This program also uses the `runtime_error` class defined in the Standard C++ Library to report a range error.

```
#include <stdexcept>
#include <iostream.h>
#include <string>
void fx ()
{
    // details omited
    throw range_error(string("some info"));

}
int main ( )
{
    try {
        fx ();
    }    catch (runtime_error& r) {
                cout <<r.what() << '\n';
    }
}
```

Function Try Block Examples

A function can catch exceptions thrown during its execution by associating catch handlers with its body, using a function try block. Note the difference between the following example and the previous exception handling example. In this case, the try keyword comes before the function body's opening brace, and the catch handler comes after the function body's closing brace.

```
#include <stdexcept>
#include <iostream.h>
#include <string>
void fx ()
{
    // .......
    throw range_error(string("some info"));
}
int main ( )
try {
        fx ();
    }
catch (runtime_error& r) {
        cout <<r.what() << '\n';     }
```

Function try blocks are sometimes necessary with class constructor destruction. A function try block is the only means of ensuring that all exceptions thrown during the construction of an object are caught within the constructor. For example,

```
A::A()
try
   : _member(fx())
{
   cout << _member << '\n';
}
catch (runtime_error& r) {
      cout <<r.what() << '\n';
}
```

Note that the function try block ensures the exception thrown from the member initializer is caught within the constructor.

Debugging Exception Handling

The HP WDB Debugger supports C++ exception handling. For more information refer to HP WDB documentation at http://www.hp.com/go/wdb.

Performance Considerations

HP aC++ exception handling has no significant performance impact at compile time or runtime.

Using Threads

The HP aC++ runtime environment supports multi-threaded applications. The following HP aC++ libraries are thread-safe with the limitations cited below:

Rogue Wave Standard C++ Library 2.2.1

For both 32-bit and 64-bit libraries:

- `libstd_v2.so` and `libstd_v2.a`
- `libCsup.so` and `libCsup.a`
- `libCsup11.so` — ISO C++11 standard compliant

Rogue Wave Standard C++ Library 1.2.1 and Tools.h++ 7.0.6

For both 32-bit and 64-bit libraries:

- `libstd.so` and `libstd.a`
- `librwtool.so` and `librwtool.a`
- `libCsup.so` and `libCsup.a`
- `libCsup11.so` — ISO C++11 standard compliant
- `libstream.so` and `libstream.a`

Using Locks

To guarantee that your I/O results from one thread are not intermingled with I/O results from other threads, you must protect your I/O statements with locks. For example:

```
// create a mutex and initialize it
pthread_mutex_t the_mutex;
#ifdef _PTHREADS_DRAFT4      // for user threads
pthread_mutex_init(&the_mutex, pthread_mutexattr_default);
#else                        // for kernel threads
pthread_mutex_init(&the_mutex, (pthread_mutexattr_t *)NULL);
#endif

pthread_mutex_lock(&the_mutex);
cout << "something" ... ;
pthread_mutex_unlock(&the_mutex);
```

Note that conditional compilation may be necessary to accommodate both the user threads and the kernel threads interfaces, as in the above example. An alternative might be to compose a buffer with an `ostrstream` and output with one write. The following example could be used with the `cfront` compatible`libstream`:

```
ostrstream ostr;
ostr << "something" /*...*/ ;
ostr << " or another" /*...*/ << endl;
cout.write(ostr.str(), ostr.pcount());
ostr.rdbuf()->freeze(0);
```

Note that the above example works with the new library, though with the deprecated `ostrstream`.

Or something similar can be done with the Rogue Wave Standard C++ Library 2.2.1 (`libstd_v2`) with standard `ostringstream`, as in the following example:

```
ostringstream ostr;
ostr << "something" /*...*/ ;
ostr << " or another" /*...*/ << endl;
cout.write(ostr.str().c_str(), ostr.str().length());
```

Note that `cout.flush` may be needed if sharing the file with `stdio`.

Required Command-line Options

To use the multi-thread safe capabilities of the Standard C++ Library, you need to specify the following options at both compile and link time. Note that the options differ depending on which set of libraries you are using.

Rogue Wave Standard C++ Library 2.2.1

For both 32-bit and 64-bit libraries:

- `-D_RWSTD_MULTI_THREAD`
- `-D_REENTRANT`
- `-lpthread` (This option applies only to kernel threads.)
- `-mt`

Rogue Wave Standard C++ Library 1.2.1 and Tools.h++ 7.0.6

For both 32-bit and 64-bit libraries:

- `-D__HPACC_THREAD_SAFE_RB_TREE` (Code compiled with this option is binary incompatible with code that is not compiled with this option. Only HP aC++ version A.01.21 and subsequent versions incorporate this option.)
- `-DRWSTD_MULTI_THREAD`
- `-DRW_MULTI_THREAD` (needed only for the Tools.h++ Library)
- `-D_REENTRANT`
- `-lcma` (This option applies only to user threads.)
- `-lpthread` (This option applies only to kernel threads.)
- `-D_THREAD_SAFE` (Unlike the other options in this table, this option is not required. You can use it with the `cout`, `cin`, `cerr`, and `clog` objects, if you are not using locks.)
- `-mt`

NOTE: If you do not specify these options as described in both cases, a runtime error will be generated or multi-thread behavior will be incorrect. If you use `+Oopenmp` in an application, you must use `-mt` on files that are not compiled with `+Oopenmp`.

Limitations

In most cases, thread safety does not imply that the same object can be shared between threads. In particular, when objects have user visible state, it would not make sense to share them between threads. Consider the following:

```
void f(ostream &out, int x, int y) {
        out << setw(3) << x << setw(10) << y;
}
```

This function would not be thread safe if called from multiple threads with the same object, since the `width` in the shared object could be changed at any time. Therefore, such objects are not protected from interactions between multiple threads, and the result of sharing such an object between threads is undefined.

If the same object is shared between threads, a runtime crash, abort, or intermingled output may occur. With the Rogue Wave Standard C++ Library 2.2.1, output may be intermingled but no aborts will occur.

Using -D_THREAD_SAFE with the cfront Compatible libstream

There is an exception to the above rule for the `cfront` compatible `libstream`. For the frequently used objects `cout`, `cin`, `cerr`, and `clog`, you can specify the `-D_THREAD_SAFE` compile time flag for any file that includes `<iostream.h>`. In this case, a new instance of the object is transparently created for each thread that uses it. All instances share the same file descriptor. The `f` function in the above example will now work, because it receives one new `out` object per thread. However, the results of two simultaneous executions of `f` will be mixed in any order in the output.

Using `-D_THREAD_SAFE` with the global scope operator is not supported for `cout`, `cin`, `cerr`, and `clog`. For example, the following code would generate an error:

```
::cout << endl;
```

NOTE: If you use locks, you need not use the `-D_THREAD_SAFE` compile time flag since you are now responsible for ensuring thread safety.

Differences between Standard iostreams and cfront Compatible libstream

The cfront compatible libstream supports locking for each insertion. Rogue Wave Standard C++ Library 1.2.1 and Tools.h++ 7.0.6 do not support locking but do provide a thread private buffer.

Visible differences would be as follows. In the case of standard iostreams, there is intermingling of each component being inserted. With cfront compatible iostreams, there is intermingling of complete buffers (depending on when `endl` or `flush` is called).

Using -D__HPACC_THREAD_SAFE_RB_TREE

The Rogue Wave Standard C++ Library 1.2.1 (`libstd`) and Tools.h++ 7.0.6 (`librwtool`) are not thread safe if the underlying implementation `rb_tree` class is involved. In other words, if the tree header file (which includes `tree.cc`) under `/opt/aCC/include/` is used, these libraries are not thread safe. Most likely, it is indirectly referenced by including the standard C++ library container class map or set headers, or by including a RogueWave tools.h++ header like `tvset.h`, `tpmset.h`, `tpmset.h`, `tvset.h`, `tvmset.h`, `tvmset.h`, `tpmap.h`, `tpmmap.h`, `tpmmap.h`, `tvmap.h`, and `tvmmap.h`. Since changing the `rb_tree` implementation to make it thread safe would break binary compatibility, the preprocessing macro, `__HPACC_THREAD_SAFE_RB_TREE`, must be defined. The macro is automatically defined in the Itanium® based environment. A new object file compiled with the macro defined should not be linked with older ones that were compiled without the macro defined. Library providers whose library is built with the macro defined may need to notify their users to also compile their source with the macro defined when the tree header is included.

Exception Handling

It is illegal to throw out of a thread.

The following example illustrates that you cannot catch an object which has been thrown in a different thread. To do so will result in a runtime abort since HP aC++ finds no available catch handler and terminate is called.

```
#include <pthread.h>
void foo() {
   int i = 10;
   throw i;
}
int main() {
   pthread_t tid;
   try {
      ret=pthread_create(&tid, 0, (void*(*)(void*))foo, 0);
   }
   catch(int n) {}
}
```

Pthreads (POSIX Threads)

Pthreads (POSIX threads) refers to the Pthreads library of thread-management routines. For information on Pthread routines see the *pthread(3t)* man page. To use the Pthread routines, your program must include the `<pthreads.h>` header file and the Pthreads library must be explicitly linked to your program.

Example:

```
aCC -mt prog.c
```

Limitations

When using STL containers as local objects, the destructor will not get called when `pthread_exit` is called, which leads to a memory leak. Do not call `pthread_exit` from C++. Instead you must throw or return back to the thread's initial function. There you can do a return instead of `pthread_exit`.

Pthread library has no knowledge of C++ stack unwinding. Calling `pthread_exit` for will terminate the thread without doing proper C++ stack unwind. That is, destructors for local objects will not be called. (This is analogous to calling `exit` for single threaded program.)

This can be fixed by calling destructors explicitly right before calling `pthread_exit`.

Example:

```
#include <pthread.h>
#include <stdlib.h>
#include <exception>

extern "C" int printf(const char*...);
struct A {
       A () { printf("ctor called\n"); }
      ~A () { printf("dtor called\n"); }
};
struct B {
       B () { printf("B ctor called\n"); }
      ~B () { printf("B dtor called\n"); }
};

__thread A* pA;  // thread specific data.

void thread_specific_destroy(A *p) {
   delete p;
}
typedef void fp(void*);
```

```
void* foo(void*) {
   pA = new A();
   B ob;
   pthread_cleanup_push(reinterpret_cast<fp*>(thread_specific_destroy),pA);
   pthread_cleanup_pop(1);
   ob.~B();  // potential problem when the thread is canceled.
   pthread_exit(0);
   return 0;
}
int main() {
   //A oa; exit(0);
   //dtor for oa won't be called if line above is uncommented.
   pthread_t thread_id;
   for (int i = 0; i < 3; i++)
   pthread_create(&thread_id, 0, &foo, 0);
   pthread_join(thread_id, 0);
}
```

NOTE: `vector::clear` does not free all of the memory. The storage is put back into a free pool for that one container.

This does not happen if a thread is canceled. In such cases, use thread specific data or thread local storage support along with `pthread_cleanup_[push|pop]` utilities.

`pthread_cancel` is not supported.

Function Scoping

The`set_terminate`, `set_unexpected`, and `set_new_handler`, functions apply to all threads in the process. For information on specific functions, refer to the appropriate library documentation.

Performance Options

You can use the`-D__HPACC_FIXED_REFCNT_MUTEX` flag to reduce the amount of space used for string mutexes and thereby increase performance when using either `-AA` or `-AP` strings. Instead of having one mutex per string, there will be a fixed array of mutexes shared among all strings. This feature requires C++ runtime version A.05.61 or newer. For additional information refer to the `-mt` option.

The number of string mutexes defaults to 64 and can be configured by:

```
export aCC_MUTEX_ARRAY_SIZE=##
```

You can mix code compiled with and without `-D__HPACC_FIXED_REFCNT_MUTEX`.

Parallel Programming Using OpenMP

OpenMP is an industry-standard parallel programming model that implements a fork-join model of parallel execution. The HP C++ OpenMP pragmas are based on the OpenMP Standard for C/C++, version 2.5.

To view the details about the standard and details about usage, syntax and values, please go to `http://www.openmp.org/drupal/node/view/8`.

OpenMP Implementation

This section summarizes some of the OpenMP directives behavior that are described as implementation-dependent in the OpenMP v2.5 API. Each behavior is cross-referenced back to its description in the OpenMP v2.5 main specification. HP, in conformance with the OpenMP v2.5 API, define and document the following behavior.

- Due to resource constraints, it is not possible for an implementation to document the maximum number of threads that can be created successfully during a program's execution. This number is dependent upon the load on the system, the amount of memory allocated by the program, and the amount of implementation dependent stack space allocated to each thread. For a 32 bit process, the stack space for each thread is allocated from the heap. The heap defaults 1 gigabyte, and the default stack size is 8 megabytes. See the linker option -N for increasing data area size to 2 gigabytes.

If the dynamic threads mechanism is disabled, requests for additional threads will result in no additional threads being created. Programs should not assume that a request will result in additional threads for all requests. If the dynamic threads mechanism is enabled, requests for more threads than an implementation can support are satisfied by creating additional pthreads which are then scheduled by the HP-UX scheduler using a smaller number of threads.

- Number of processors: The number of physical processors actually hosting the threads at any time is the lesser of the number of threads or the number of physical processors on the system.
- Creating teams of threads: The number of threads in a team that executes a nested parallel region is dependent on the number of threads created when the application is started and the number of threads already in use. The number of threads created at application startup defaults to the number of processors on the system, but may be increased or decreased using the `OMP_NUM_THREADS` environment variable. If all threads are already in use when a nested parallel region is encountered, the number of threads in the team that executes the parallel region is one. If all threads are not already in use when a nested parallel region is encountered, the number of threads in the team used to execute the parallel region will be the lesser of the number of threads requested by a `num_threads` clause or the most `omp_set_num_threads` call, if any, or the number of threads not already in use. (Section 2.3, page 10 of OpenMP C/C++ specs).
- When schedule(`runtime`) clause is specified, the decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the resulting schedule defaults to static schedule with a chunk size of 1.
- In the absence of the `schedule` clause, the default schedule is static with chunksize computed as iterations.
- An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section. HP implements the `ATOMIC` clause using a slightly more efficient form of critical section roughly 60-70% faster than critical, although there is still a runtime call.
- `omp_get_num_threads`: If the number of threads has not been explicitly set by the user, the default is the number of physical processors on the system.
- `omp_set_dynamic`: The default for dynamic thread adjustment is 0 (disabled).
- `omp_set_nested`: When nested parallelism is enabled, the number of threads used to execute nested parallel regions is determined at runtime by the underlying OpenMP parallel library.
- `OMP_SCHEDULE` environment variable: The default value for this environment variable is `STATIC`.
- `OMP_NUM_THREADS` environment variable: The default value is the number of physical processors on the system.
- `OMP_DYNAMIC` environment variable: The default value is `FALSE`.

OpenMP Header File

Every C++ program that contains OpenMP pragmas is to be compiled for the current version of HP-UX and must include the header file `<omp.h>`. If it does not, the OpenMP pragmas are ignored. The default path for `<omp.h>` is `/usr/include`.

OpenMP Library

The OpenMP APIs are defined in the library `libomp`.

+O[no]openmp Command Line Option

The `+Oopenmp` option is accepted at all optimization levels. The `+Oopenmp` option enables the recognition of OpenMP pragmas. Using the `+Onoopenmp` option will ignore all OpenMP directives silently.

See Chapter 3: "Pragma Directives and Attributes" (page 96) for more information on OpenMP pragmas.

_OPENMP Macro

The `_OPENMP` macro name is defined by OpenMP compliant implementation as the decimal constant 200203. This macro must not be the subject of `#define` or `#undef` preprocessing directive. Following is an example of conditional compilation:

```
#ifdef _OPENMP
iam = omp_get_thread_num() + index;
#endif
```

Environment Variables in OpenMP

The OpenMP environment variables recognized by HP aC++ compiler control the execution of parallel code. Note that the environment variable names are case sensitive and they must be in uppercase.

The following environment variables are available in HP aC++ compiler:

- OMP_SCHEDULE
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED

OMP_SCHEDULE

```
export OMP_SCHEDULE="kind[,chunk_size]"
```

```
setenv OMP_SCHEDULE "kind[,chunk_size]"
```

where, `kind` is either of of `static`, `dynamic`, or `guided`.

This environment variable applies to `for` and `parallel for` directives that have the schedule type as runtime. The schedule type and chunk size for all such loops can be set at runtime by setting this environment variable to any of the recognized schedule types and to an optional `chunk_size`.

The default value of the environment variable is a `static` schedule with a `chunk_size` of 1. If the optional `chunk_size` is set, the value must be positive. If `chunk_size` is not set, a value of 1 is assumed, except for `static` schedule. For a `static` schedule, the default `chunk_size` is set to the loop iteration space divided by a number of threads applied to the loop.

NOTE: `OMP_SCHEDULE` is ignored for `for` and `parallel for` directives that have a schedule type other than runtime.

OMP_NUM_THREADS

```
export OMP_NUM_THREADS=value
```

```
setenv OMP_NUM_THREADS value
```

The `OMP_NUM_THREADS` environment variable sets the default number of threads to use during execution. The value of `OMP_NUM_THREADS` must be a positive integer. Its effect depends on whether dynamic adjustment of the number of threads is enabled, and its interaction with the `omp_set_num_threads` library routine and any `num_threads` clause on a parallel directive.

The default value is the number of physical processors on the system.

OMP_DYNAMIC

```
export OMP_DYNAMIC=value
```

```
setenv OMP_DYNAMIC value
```

The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The value must be either `TRUE` or `FALSE`. The default value is `FALSE`.

If the value is set to `FALSE`, dynamic adjustment is disabled. If the value is set to `TRUE`, the number of threads that are used for executing parallel regions may be adjusted by the runtime environment to best utilize system resources.

OMP_NESTED

```
export OMP_NESTED=value
```

```
setenv OMP_NESTED value
```

The `OMP_NESTED` environment variable enables or disables nested parallelism. Its value must be `TRUE` or `FALSE`.

If the value is set to `TRUE`, nested parallelism is enabled and if the value is set to `FALSE`, the nested parallelism is disabled. The default value is `FALSE`.

Runtime Library Functions in OpenMP

The OpenMP library functions are external functions. The header `<omp.h>` declares three types of functions:

- Several functions that can be used to control and query the parallel execution environment.
- Lock functions that can be used to synchronize access to data.
- Timing functions that support a wall-clock timer.

Description of library functions are divided into the following topics:

- Execution Environment Functions
- Lock Functions
- Timing Functions

Execution Environment Functions

The execution environment functions affect and monitor threads, processors, and the parallel environment. This section discusses the following environment functions:

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num
- omp_get_num_procs
- omp_in_parallel
- omp_set_dynamic
- omp_get_dynamic
- omp_set_nested
- omp_get_nested

omp_set_num_threads

```
#include <omp.h>
```

```
void omp_set_num_threads(int num_threads);
```

The `omp_set_num_threads` function sets the number of threads to use for subsequent parallel regions. The value of the parameter `num_threads` must be positive. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value is used as the number of threads for all subsequent parallel regions prior to the next call to this function; otherwise, the value is the maximum number of threads that will be used. This function has effect only when called from serial portions of the program. If it is called from a portion of the program where the `omp_in_parallel` function returns non-zero, the behavior of this function is undefined.

For more information on this subject, see the omp_set_dynamic and `omp_get_dynamic` functions. This call has precedence over the `OMP_NUM_THREADS` environment variable.

omp_get_num_threads

```
#include <omp.h>
```

```
int omp_get_num_threads(void);
```

The `omp_get_num_threads` function returns the number of threads currently in the team executing the parallel region from which it is called. The `omp_set_num_threads` function and the `OMP_NUM_THREADS` environment variable control the number of threads in a team. If the number of threads has not been explicitly set by the user, the default is implementation dependent. This function binds to the closest enclosing parallel directive. If called from a serial portion of a program, or from a nested parallel region that is serialized, this function returns 1.

omp_get_max_threads

```
#include <omp.h>
```

```
int omp_get_max_threads(void);
```

The `omp_get_max_threads` function returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a `num_threads` clause were to be encountered at that point in the code.

omp_get_thread_num

```
#include <omp.h>
```

```
int omp_get_thread_num(void);
```

The `omp_get_thread_num` function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and `omp_get_num_threads` -1, inclusive. The master thread of the team is thread 0. If called from a serial region, `omp_get_thread_num` returns 0. If called from within a nested parallel region that is serialized, this function returns 0.

omp_get_num_procs

```
#include <omp.h>
```

```
int omp_get_num_procs(void);
```

The `omp_get_num_procs` function returns the number of processors that are available to the program at the time the function is called.

omp_in_parallel

```
#include <omp.h>
```

```
int omp_in_parallel(void);
```

The `omp_in_parallel` function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. This function returns non-zero from within a region executing in parallel, including nested regions that are serialized.

omp_set_dynamic

```
#include <omp.h>
```

```
void omp_set_dynamic(int dynamic_threads);
```

The `omp_set_dynamic` function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. This function has effect only when called from serial portions of the program. If it is called from a portion of the program where the `omp_in_parallel` function returns non-zero, the behavior of the function is undefined. If `dynamic_threads` evaluates to non-zero, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the runtime environment to best utilize system resources. As a consequence, the number of threads specified by the user is the maximum thread count. The number of threads always remains fixed over the duration of each parallel region and is reported by the `omp_get_num_threads` function. If `dynamic_threads` evaluates to 0, dynamic adjustment is disabled. A call to `omp_set_dynamic` has precedence over the `OMP_DYNAMIC` environment variable.

The default for the dynamic adjustment of threads is 0.

omp_get_dynamic

```
#include <omp.h>
```

```
int omp_get_dynamic(void);
```

The `omp_get_dynamic` function returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.

omp_set_nested

```
#include <omp.h>
```

```
void omp_set_nested(int nested);
```

The `omp_set_nested` function enables or disables nested parallelism. If nested evaluates to 0, which is the default, nested parallelism is disabled, and nested parallel regions are serialized and executed by the current thread. If nested evaluates to non-zero, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form the team. This call has precedence over the `OMP_NESTED` environment variable.

omp_get_nested

```
#include <omp.h>
```

```
int omp_get_nested(void);
```

The `omp_get_nested` function returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Lock Functions

The functions described in this section manipulate locks used for synchronization.

For the following functions, the lock variable must have type `omp_lock_t`. For nestable lock functions, the lock variable must have type `omp_nest_lock_t`.

This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to `omp_lock_t` type for lock functions and `omp_nest_lock_t` for nestable lock functions.

- omp_init_lock and *`omp_init_nest_lock`*
- omp_destroy_lock and omp_destroy_nest_lock
- omp_set_lock and omp_set_nest_lock
- omp_unset_lock and omp_unset_nest_lock
- omp_test_lock and omp_test_nest_lock

omp_init_lock and omp_init_nest_lock

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter lock for use in subsequent calls. The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero

omp_destroy_lock and omp_destroy_nest_lock

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

These functions ensure that the pointed to lock variable lock is uninitialized. The argument to these functions must point to an initialized lock variable that is locked.

omp_set_lock and omp_set_nest_lock

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. For a simple lock, the argument to the `omp_set_lock` function must point to an initialized lock variable. Ownership of the lock is granted to the thread executing the function.

For a nestable lock, the argument to the `omp_set_nest_lock` function must point to an initialized lock variable. The nesting count is incremented, and the thread is granted, or retains, ownership of the lock.

omp_unset_lock and omp_unset_nest_lock

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

These functions provide the means of releasing ownership of a lock. The argument to each of these functions must point to an initialized lock variable owned by the thread executing the function. The behavior is undefined if the thread does not own that lock.

For a simple lock, the `omp_unset_lock` function releases the thread executing the function from ownership of the lock.

For a nestable lock, the `omp_unset_nest_lock` function decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

omp_test_lock and omp_test_nest_lock Functions

```
#include <omp.h>
```

```
int omp_test_lock(omp_lock_t *lock);
```

```
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

These functions attempt to set a lock but do not block execution of the thread. The argument must point to an initialized lock variable. These functions attempt to set a lock in the same manner as `omp_set_lock` and `omp_set_nest_lock`, except that they do not block execution of the thread.For a simple lock, the `omp_test_lock` function returns non-zero if the lock is successfully set; otherwise, it returns zero.

For a nestable lock, the `omp_test_nest_lock` function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

Timing Functions

The functions described in this section support a portable wall-clock timer:

- omp_get_wtime
- omp_get_wtick

omp_get_wtime

```
#include <omp.h>
```

```
double omp_get_wtime(void);
```

The `omp_get_wtime` function returns a double-precision floating-point value equal to the elapsed wall clock time in seconds since some time in the past. The actual time in the past is arbitrary, but it is guaranteed not to change during the execution of the application program.

The function may be used to measure elapsed times as shown in the following example:

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf(“Work took %f sec. time.\n”, end-start);
```

The time returned is per-thread times. They are not required to be globally consistent across all the threads participating in an application.

omp_get_wtick

```
#include <omp.h>
```

```
double omp_get_wtick(void);
```

The `omp_get_wtick` function returns a double-precision floating-point value equal to the number of seconds between successive clock ticks.

9 Tools and Libraries

This chapter discusses tools and libraries bundled with HP aC++. It discusses the following topics:

HP Specific Features of lex and yacc

`lex` and `yacc` are bundled with HP aC++. The following is a list of HP specific features of `lex` and `yacc`:

- `LC_CTYPE` and `LC_MESSAGES` environment variable support in `lex`:

  Determines the size of the characters and language in which messages are displayed while you use `lex`.

- `-m` command line option for lex:

  Specifies that multibyte characters may be used anywhere single byte characters are allowed. You can intermix both 8-bit and 16-bit multibyte characters in regular expressions if you enable the `-m` command line option.

- `-w` command line option for `lex`:

  Includes all features in `-m` and returns data in the form of the `wchar_t` data type.

- `%l <locale>` directive for `lex`:

  Specifies the locale at the beginning of the definitions section. Any valid locale recognized by the `setlocale` function can be used. This directive is similar to using the `LC_CTYPE` environment variable. To receive `wchar_t` support with `%l`, use the `-w` command line option.

- `LC_CTYPE` environment variable support in `yacc`:

  Determines the native language set used by `yacc` and enables multibyte character sets. Multibyte characters can appear in token names, on terminal symbols, strings, comments, or anywhere ASCII characters can appear, except as separators or special characters.

NOTE: When using `lex` and `yacc`:

- Programs generated by `yacc` or `lex` can have many unreachable `break` statements, causing multiple aC++ warnings.
- If you want to call the `yacc` generated routines, `yyerror`, `yylex` and `yyparse`, your program must include the `yacc.h` header file:

  ```
  #include<yacc.h>
  ```

For more information on these tools, refer to the `lex` and `yacc` man pages or the *HP-UX Reference Manual*. Another general source of information is *lex and yacc* by John R. Levine, Tony Mason, and Doug Brown.

Creating and Using Libraries

This section gives an overview of libraries provided with HP aC++. It also discusses how you can create and use your own libraries.

This section discusses the following topics:

- Advanced Shared Library Features
- Standard HP-UX Libraries and Header Files
- Allocation Policies for Containers

For more information, see *HP-UX Online Linker and Libraries User's Guide*.

HP aC++ Libraries

In addition to standard HP-UX system libraries, HP aC++ provides the following C++ libraries:

- Standards Based Libraries
  - Standard C++ Library
  - Tools.h++ Library
  - HP aC++ Runtime Support Library
- HP C++ (cfront) Compatibility Library
  - IOStream Library

Standard C++ Library

The International Standards Organization (ISO) and the American National Standards Institute (ANSI) have standardized the C++ programming language. A result of this standardization process is the Standard C++ Library, a large and comprehensive collection of classes and functions.

Introduction

HP aC++ provides the Rogue Wave implementation of the ANSI/ISO Standard C++ Library. This implementation includes the following features:

- A subset of data structures and algorithms, updated from the original library developed at Hewlett-Packard by Alex Stepanov and Meng Lee and known as the C++ Standard Template Library (STL)

  NOTE: The public domain C++ Standard Template Library is not supported by this Standard C++ Library.

  Technical Corrigenda 1 has changed the STL function `make_pair` to take their arguments by value instead of const reference. This change brings the HP library into compliance if the enabling macro `-D__HP_TC1_MAKE_PAIR` is specified at compile time. For binary compatibility reasons, the default behavior is unchanged.

- A templatized `string` class
- A templatized class for representing complex numbers.
- A framework that describes the execution environment, using a template class, `numeric_limits`, and specializations for each fundamental data type
- Memory management features
- Language support features
- Exception handling features

Introduction to Using the Standard C++ Library

Although the design of a large portion of the Standard C++ Library is, in many ways, not object-oriented, C++ excels as a language for manipulating objects. How do you integrate the non-object-oriented architecture of the library with the strengths of the language for manipulating objects?

The key is to use the right tool for each task. For a majority of programming tasks, object-oriented techniques is the preferred approach. Products such as Rogue Wave's Tools.h++ Library, which encapsulate the Standard C++ Library with a familiar object-oriented interface, provide the power and advantages of object-orientation.

Use Standard C++ Library components directly when you need flexibility or highly efficient code. Use the more traditional approach to object-oriented design, such as encapsulation and inheritance, to model larger problem domains and knit all the pieces into a complete solution. To devise an architecture for your application, use encapsulation and inheritance to compartmentalize the problem. For an efficient data structure or algorithm for a compact problem that often resolves to a single class, use the Standard C++ Library. Use this library to create reusable classes when low-level constructs are needed. Use traditional OOP techniques to combine classes to solve a larger problem.

In future, most libraries will use the Standard C++ Library as their foundation. Using the Standard C++ Library, either directly or through an encapsulation such as Tools.h++ Library, ensures interoperability. This is important in large projects that may rely on communication among several libraries. A good thumb rule is to use the highest encapsulation level available to you, but make sure that the Standard C++ Library is available as the base for inter-library communication and operation.

The C++ language supports a wide range of programming approaches. The language, and the Standard C++ Library that supports it, are designed to give you the flexibility to approach each unique problem from the best possible angle.

The Standard C++ Library, when combined with traditional OOP techniques, is a flexible tool to build a collection of C++ classes. These classes can stand alone as a library or are tailored to a specific task.

Differences between Standard C++ Library and Other Libraries

A major portion of the Standard C++ Library is comprises a collection of class definitions for standard data structures, and a collection of algorithms commonly used to manipulate such structures. This part of the library is derived from the Standard Template Library (STL). The organization and design of this part of the library differs in almost all respects from the design of most other C++ class libraries, because it avoids encapsulation and uses almost no inheritance.

An emphasis on encapsulation is a key hallmark of object-oriented programming. The emphasis on combining data and functionality into an object is a powerful organization principle in software development; indeed it is the primary organizational technique.

Inheritance is a powerful technique to share code and reuse software. It is applicable when two or more classes share a common set of basic features. For example, in a graphical user interface, two types of windows may inherit from a common base window class, and the individual subclasses provide any required unique features. In another use of inheritance, object-oriented container classes may ensure common behavior and support code reuse by inheriting from a more general class, and factoring out common member functions.

The designers of the STL decided against using an entirely object-oriented approach, and separated the tasks to be performed using common data structures from the representation of the structures themselves. The STL is designed as a collection of algorithms and a separate collection of data structures that are manipulated using the algorithms.

The Non-Object-Oriented Design of the Standard C++ Library

The portion of the Standard C++ Library derived from the STL was purposely designed with an architecture that is not object-oriented. This design has both advantages and disadvantages. Some of them are mentioned below.

Smaller Source Code

There are approximately 50 different algorithms and about a dozen major data structures. This separation reduces the source code size, and decreases the risk of similar activities with dissimilar interfaces. In the absense of this separation, each algorithm must be implemented in each data structure. This requires additional member functions apart from those in the present scheme.

Flexibility

The algorithms that are separated from data structures can be used with conventional C++ pointers and arrays. Algorithms encapsulated within a class hierarchy do not have this ability because C++ arrays are not objects.

Efficiency

The STL in particular, and the Standard C++ Library in general, provide a low-level approach to develop C++ applications. This low-level approach is useful when specific programs require an emphasis on efficient coding and execution speed.

Iterators: Mismatches and Invalidations

The Standard C++ Library data structures use pointer-like objects called iterators to describe the contents of a container. Given the architecture of the library architecture, it is impossible to verify whether the iterator elements are matched or derived from the same container. Using a beginning iterator from one container with an ending iterator from another produces unexpected results. It is very important to know that iterators are invalidated as a result of a subsequent insertion or deletion from the underlying container class. The use of an invalid iterator produces unexpected results. Familiarity with the Standard C++ Library reduces the number of errors related to iterators.

Templates: Errors and Code Bloat

The flexibility and power of templatized algorithms is, with most compilers, results in a loss of precision in diagnostics. Errors in the parameter lists of generic algorithms is sometimes displayed only as obscure compiler errors for internal functions. These errors are defined many levels deep in template expansions. Familiarity with algorithms and their requirements is necessary to use the standard library. Heavy reliance on templates causes programs to grow larger than expected. You can minimize this problem by recognizing the cost of instantiating a particular template class, and by making appropriate design decisions. As compilers become more fluent in templates, the problem of size is reduced.

Multithreading Problems

When you use the Standard C++ Library in a multithreaded environment, the iterators cannot safely pass between threads. This is because iterators are independent of the containers they operate on. It is impossible to protect a container when it spawns iterators in multiple threads as iterators are used to modify a non-const container. Use thread-safe wrappers, such as those provided by Tools.h++ Library to access a container from multiple threads.

Standard C++ Library Reference

The *Standard C++ Library Reference* provides an alphabetical listing of all of the classes, algorithms, and function objects in the prior Rogue Wave implementation of the Standard C++ Library.

Incompatibilities Between the Library and the Standard

The ANSI/ISO C++ International Standard is different from Standard C++ Library. For example, the `times` function object in the functional header file. In the standard, `times` is renamed to `multiplies`.

To use `multiplies` in your code and to be compatible with the ANSI/ISO C++ International Standard, use a conditional compilation flag on the `aCC` command line.

Example:

The following program uses the `times` function object:

```
// test.c
int times;                    //user defined variable
#include <functional>
                              //multiplies can be used in

int main() {}
// end of test.c
```

Compile this program using the following command:

```
aCC -D__HPACC_USING_MULTIPLIES_IN_FUNCTIONAL test.c
```

Depending on the existence of the conditional compilation flag, `functional` defines either `times`, or `multiplies`, not both.

If you have old source that uses `times` in header `functional` and a new source that uses `multiplies`, the sources cannot be mixed. Mixing the two sources constitutes a non-conforming program, and the old and new sources may or may not link.

If your code uses the old name `times`, and you want to continue to use the present non-standard `times` function object, you need not to do anything to compile the old source.

Tools.h++ Library

The Tools.h++ Library is a foundation class library built on the Standard C++ Library. Use its object-oriented capabilities to simplify coding and facilitate code reuseablility.

The Rogue Wave Software Tools.h++ Class Reference describes all classes and functions in the Tools.h++ Library. It is intended for use with Rogue Wave Standard C++ Library. It is provided as HTML formatted files. You can view these files with an HTML browser by opening the `/opt/aCC/html/librwtool/ref.htm` file.

HP aC++ Runtime Support Library

The HP aC++ runtime support library is provided as a shared library (`/usr/lib/hpux32/libCsup.so`, `/usr/lib/hpux32/libCsup11.so`, `/usr/lib/hpux64/libCsup.so` and `/usr/lib/hpux64/libCsup11.so`).

The library supports the following functionality:

- Exception Handling
- Memory Management (operators `new` and `delete`)
- Start and termination of a C++ program
- Runtime type identification (`type_info`)
- Static object constructors and desctructors

See "HP aC++ Runtime Libraries and Header Files" (page 187) for more information.

IOStream Library

In this release of HP aC++, the standards based iostream capabilities of the Standard C++ Library are still evolving. As a result, an HP C++ (cfront) compatible IOStream library is provided.

Standard Components Library Not Provided

The Standard Components Library is not provided with the HP aC++ compiler for Integrity servers. HP recommends that you use the similar features of the Standard C++ Library in place of the Standard Components Library.

Linking to C++ Libraries

You can compile and link any C++ module to one or more libraries. HP aC++ automatically links the following libraries with a C++ executable.

- `/usr/lib/hpux##/libCsup.so` (the HP aC++ runtime support library)
- `/usr/lib/hpux##/libstd_v2.so` (standard C++ library: `-AA`)
- `/usr/lib/hpux##/libc.so` (the HP-UX system library)
- `/usr/lib/hpux##/libdl.so` (routines to manage shared libraries)
- `/usr/lib/hpux##/libunwind.so` (routines to unwind exceptions)
- `/usr/lib/hpux##/libm.so` (math library)

NOTE:

- When you specify the `-b` option to create a shared library, these defaults do not apply.
- When you specify the +std=c++11 (or the currently deprecated -Ax) option during the link time, libCsup11.so gets linked instead of libCsup.so.

Linking with Shared or Archive Libraries

If you want archive libraries instead of shared libraries, use the `-a`, archive linker option. To create a completely archived executable, use the `+A` option. To maintain compatibility with future releases, do not mix archive and shared libraries should not be mixed.

Refer to the *HP-UX Linker and Libraries User's Guide* for more information.

Specifying Other Libraries

You can specify other libraries using the `-l` option. For example, to use the Tools.h++ library (`-AA` version), specify `-lrwtool_v2`:

```
aCC myapp.C -lrwtool_v2
```

Creating and Using Shared Libraries

This section describes shared libraries that are specific to HP aC++. It discusses the following topics:

- Compiling for Shared Libraries
- Creating a Shared Library
- Using a Shared Library
- Linking Archive or Shared Libraries
- Updating a Shared Library

Compiling for Shared Libraries

To create a C++ shared library, you must first compile your C++ source with either the `+z` or `+Z` option. These options create object files containing position-independent code (PIC).

Example

The following example compiles `util.C`, generates position-independent code, and puts the code into the object file `util.o. util.`This object file can later be put into a shared library.

```
aCC -c +z util.C
```

Creating a Shared Library

To create a shared library from one or more object files, use the `-b` option at link time. (The object files must have been compiled with `+z` or `+Z`.) The `-b` option creates a shared library rather than an executable file.

NOTE: Use the `aCC` command to create a C++ shared library. This ensures that static constructors and destructors are executed at appropriate times.

Example

The following example links `util.o` and creates the shared library `util.so`.

```
aCC -b -o util.so util.o
```

Using a Shared Library

To use a shared library, include the name of the library in the `aCC` command line or use the`-l` option.

The linker links the shared library to the executable file it creates. Once you create an executable file that uses a shared library, do not move the shared library as the dynamic loader (`dld.so`) cannnot find it.

NOTE: Use the `aCC` command to link any program that uses a C++ shared library. This ensures that static constructors and destructors in the shared library are executed at appropriate times.

Example

The following example compiles `prog.C`, links it with the shared library `util.so`, and creates the executable file `a.out`.

```
aCC prog.C util.so
```

Example of Creating and Using a Shared Library

The following command compiles the two files, `Strings.C` and `Arrays.C`, and creates the two object files, `Strings.o` and `Arrays.o`. These object files contain position-independent code (PIC):

```
aCC -c +z Strings.C Arrays.C
```

The following command builds the shared library, `libshape.so`, from the object files `Strings.o`and `Arrays.o`:

```
aCC -b -o libshape.so Strings.o Arrays.o
```

The following command compiles a program, `draw_shapes.C`, that uses the shared library, `libshape.so`:

```
aCC draw_shapes.C libshape.so
```

Linking Archive or Shared Libraries

When an archive and shared version of a particular library reside in the same directory, the linker links in the shared version by default. You can override this behavior with the `-a` linker option.

NOTE: Use the `+A` option when using only archive libraries to create a completely archived executable.

The `-a` option identifies the library type for the linker. The `-a` option is positional and applies to all subsequent libraries specified with the `-l` option until the end of the command line or until the next `-a` option is encountered. Pass the `-a` option to the linker with the `-Wx,args` option.

Syntax

`-Wl,-a,{archive|shared|shared_archive|archive_shared|default}`

where,

`-Wl,-a,archive`	Selects *`archive`* libraries. If the *`archive`* library does not exist, the linker generates a warning message and does not create the output file.
`-Wl,-a,archive_shared`	If *`archive_shared`* is active, the *`archive`* form is preferred, but the *`shared`* form is allowed.
`-Wl,-a,shared`	Selects *`shared`* libraries. If *`shared`* libraries do not exist, the linker generates a warning message and does not create the output file.
`-Wl,-a,shared_archive`	If *`shared_archive`* is active, the shared form is preferred, but the archive form is allowed.
`-Wl,-a,default`	Selects the *`shared`* library. If the *`shared`* library does not exist, the linker selects the *`archive`* library.

Example

The following example directs the linker to use the archive version of the library `libshape`, followed by standard shared libraries if they exist; otherwise select archive versions.

```
aCC box.o sphere.o -Wl,-a,archive -lshape -Wl,-a,default
```

Updating a Shared Library

The `aCC` command cannot replace or delete object modules in a shared library. To update a C++ shared library, you must recreate the library with all the object files you want the library to include.

For example, when a module in an existing shared library requires a fix, recompile the fixed module with the `+z` or `+Z` option, and recreate the shared library with the `-b` option.

Programs that use this library will now use the new versions of the routines. You do not have to relink programs that use this shared library because they are attached at run time.

Advanced Shared Library Features

This section explains additional things you can perform with shared libraries. It discusses the following topics:

- Forcing the Export of Symbols in main
- Binding Times
- Side Effects of C++ Shared Libraries
- Routines and Options to Manage C++ Shared Libraries
- Version Control for Shared Libraries
- Adding New Versions to a Shared Library

Forcing the Export of Symbols in main

By default, the linker exports from a program only those symbols that were imported by a shared library. For example, if shared libraries of an executable does not reference the `main` routine of the program, the linker does not include the `main` symbol in the export list of `a.out`.

Normally, this is a problem only when a program explicitly calls shared library management routines. (See "Routines and Options to Manage C++ Shared Libraries" (page 183).)

To make the linker export all symbols from a program, use the `-Wl,-E` option, which passes the `-E` option to the linker.

Binding Times

Because shared library routines and data are not actually contained in the `a.out` file, the dynamic loader must attach the routines and data to the program at run time. To accelerate program startup time, routines in a shared library are not bound until referenced. (Data items are always bound at program startup.) This deferred binding distributes binding overhead across the total execution time of the program and is especially helpful for programs that contain many references that are not likely to be executed.

Forcing Immediate Binding

You can force immediate binding, that forces all routines and data to be bound at startup time. With immediate binding, the overhead of binding occurs only at program startup time, rather than across the execution of the program. Immediate binding also detects unresolved symbols at startup time, rather than during program execution. Immediate binding provides better interactive performance, but the program startup time is longer. To force immediate binding, use the option `-Wl,-B,immediate`.

Example:

```
aCC -Wl,-B,immediate draw_shapes.o -lshape
```

To specify default binding, use the `-Wl,B,deferred` option.

For more information, refer to the *HP-UX Online Linker and Libraries User's Guide*.

Side Effects of C++ Shared Libraries

When you use a C++ shared library, all constructors and destructors of nonlocal static objects in the library are executed. This differs from a C++ archive library where only the constructors and destructors that are actually used in the application are executed.

Routines and Options to Manage C++ Shared Libraries

You can call any of several routines to explicitly load and unload shared libraries, and to obtain information about shared libraries.

If an error occurs when calling shared library management routines, the system error variable, `errno`, is set to an appropriate error value. Constants are defined for these error values in `/usr/include/errno.h`. When a program checks for these values, it must include `errno.h`:

```
#include <errno.h>
```

Linker Options to Manage Shared Libraries

Use the linker options to specify shared library binding time, symbol export, and other shared library management features.

NOTE: You must use the `-Wx,args` compiler option to specify any linker option on the compiler command line.

Refer to *HP-UX Online Linker and Libraries User's Guide* for more information about library management routines and linker options.

Version Control for Shared Libraries

You can create different versions of a routine in a shared library with the `HP_SHLIB_VERSION` pragma. `HP_SHLIB_VERSION` assigns a version number to a module in a shared library. The version number applies to all global symbols defined in the source file. Use this pragma only if incompatible changes are made to a source file. Refer to *HP-UX Online Linker and Libraries User's Guide* for more information about version control in shared libraries.

Adding New Versions to a Shared Library

To rebuild a shared library with new versions of some of the object files, use the aCC command and the -b option with the old object files and the newly compiled object files. The new source files should use the HP_SHLIB_VERSION pragma. Refer to *HP-UX Online Linker and Libraries User's Guide* for more information.

Standard HP-UX Libraries and Header Files

HP-UX includes Several libraries that provide system services. You can access HP-UX standard libraries by using header files that declare interfaces to those libraries. Refer to the *HP-UX Reference Manual* for more information on library routines.

Location of Standard HP-UX Header Files

The standard HP-UX header files are located in /usr/include directory.

Using Header Files

To use a system library function, your HP aC++ source code must include the preprocessor directive #include.

Example:

```
#include <filename.h>
```

where filename.h is the name of the C++ header file for the library function you want to use. By enclosing filename.h in angle brackets, the HP aC++ compiler looks for that particular header file in a standard location on the system. The compiler first looks for header files in /opt/aCC/include directory. When no header files are found in this directory, it searches /usr/includeUse header file options to modify the search path..

Example

To use the getenv function that is in the standard system libraries (/usr/lib/libc.so and /usr/lib/libc.a), specify:#include <stdlib.h>

because the external declaration of getenv is found in the header file /usr/include/stdlib.h.

Allocation Policies for Containers

By default, allocating memory for STL containers is optimized for large applications. Defaults have been tuned with speed efficiency as a primary concern. Space efficiency was considered, but was secondary. Typically, therefore, memory is not allocated as required, because this method is slow and inefficient. The containers obtain a block of memory to hold many elements, and when this fills up, they get another block. The size of the block depends on the element size. As a result, containers with only a few items might end up allocating too much memory. This default behavior can be adjusted to individual application needs.

For -AP Standard Library

The inline template function__rw_allocation_size can be explicitly specialized to return the number of units for each type's use in any container:

```
template <>
inline size_t __rw-allocation_size(bar*,size_t) {
return 1;
}
```

This would initially allocate one unit when dealing with containers of type bar. Alternatively, if RWSTD_STRICT_ANSI is not defined, then container member functionallocation_size can be used to directly set buffer_size, the number of units to allocate.

For -AA Standard Library

The following 4 defines can change the container initial allocation and growth ratio:

For an arbitrary container type:

```
# define _RWSTD_MINIMUM_NEW_CAPACITY size_t(32)
# define _RWSTD_NEW_CAPACITY_RATIO float(1.618)
```

For a string type:

```
# define _RWSTD_MINIMUM_STRING_CAPACITY size_t(128)
# define _RWSTD_STRING_CAPACITY_RATIO float(1.618)
```

For more precise control of containers, the following explicit specialization can be used.

The namespace scope function`template __rw` can be explicitly specialized to return the current size in elements of any container.

```
template <>
inline size_t __rw_new_capacity (size_t __size, const _Container*)
```

The parameters passed in are the current size in elements and the container's pointer. The default behavior results in an amortized constant time algorithm that dramatically increases rapidity while retaining a regard for space efficiency.

The defaults have been tuned with speed versus space optimization of container performance with regard to allocation of memory.

The ratio parameter must be above 1 for an amortized constant time algorithm. Lowering the ratio will lower rapidity and improve space efficiency. This effect will be most noticable when working with containers of few elements (less than 32).

The following is a container allocation example for both the -AP and -AA Standard library:

```
#include
namespace std {} using namespace std;
#include
#include
#define NUM  4
int printMallocInfo(const char*);
#ifndef DEFAULT
// specialize default size
// Default buffer size for containers.
#ifdef _HP_NAMESPACE_STD
namespace __rw {
template <>
inline size_t __rw_new_capacity(size_t __size, const list*) {
   printf("......\n");
   // if small grow by 5, else by 1/8 current size
   return __size < 100 ? 5 : __size / 8;
}
}
#else // -AP
template <>
inline size_t __rw_allocation_size(int*, size_t) {
   printf("......\n");
   return sizeof(int) >= 1024 ? 1 : 5;
}
#endif // -AA
#endif // DEFAULT
int main() {
   int count = 0;
   list *tryit;   for (int i = 0;ipush_back(i);
      printMallocInfo("1st entry added");
      tryit->push_back(i + 1);
      printMallocInfo("2nd entry added");
      count++;
   }
   printMallocInfo("at end");
```

```
    printf("%d\n", count);
}int printMallocInfo(const char* title) {
   static long lastValue;
   struct mallinfo info;
   info = mallinfo();
   printf("%s\n",title);
   printf("Memory allocation info:\n");
   printf("  total space in arena            = %d\n", info.arena);
#ifdef DETAILS
   printf("  number of ordinary blocks       = %d\n", info.ordblks);
   printf("  number of small blocks          = %d\n", info.smblks);
   printf("  space in holding block headers  = %d\n", info.hblkhd);
   printf("  number of holding blocks        = %d\n", info.hblks);
   printf("  space in small blocks in use    = %d\n", info.usmblks);
   printf("  space in free small blocks      = %d\n", info.fsmblks);
   printf("  space in ordinary blocks in use = %d\n", info.uordblks);
   printf("  space in free ordinary blocks   = %d\n", info.fordblks);
   printf("  keep option space penalty       = %d\n", info.keepcost);
#else
   printf("  space in use                    = %d\n",
           info.usmblks + info.uordblks);
   printf("  space free                      = %d\n",
            info.fsmblks + info.fordblks);
#endif
//printf("\n\n size used is  %d \n",( info.arena -lastValue )/NUM);
//lastValue = info.arena;
   return 0;
}
```

HP aC++ File Locations

This section gives you information on the HP aC++ file locations. It discusses the following topics:

- HP aC++ Executable Files
- HP aC++ Runtime Libraries and Header Files

HP aC++ Executable Files

Following are the HP aC++ executable files and their locations:

- `/opt/aCC/bin/aCC` (Driver)

  This is the only supported interface to HP aC++ and to the linker for HP aC++ object files.
- `/opt/aCC/lbin/ecom` (A.06.* Compiler)

  The compiler performs source compilation; preprocessing is incorporated into it.
- `/opt/aCC/lbin/ctcom` (A.05.* Compiler)

  The compiler performs source compilation; preprocessing is incorporated into it.
- `/opt/aCC/bin/c++filt` (Name Demangler)

  The name demangler implements the name demangling algorithm, which encodes function name, class name, parameter number and name.
- `/usr/ccs/bin/ld` (Linker)

  The linker links executables and builds shared libraries.

HP aC++ Runtime Libraries and Header Files

Following lists the HP aC++ runtime libraries and locations:

- Standard C++ Library
  - `/usr/lib/hpux32/libstd.so` - 32-bit shared version
  - `/usr/lib/hpux32/libstd.a` - 32-bit archive version
  - `/usr/lib/hpux64/libstd.so` - 64-bit shared version
  - `/usr/lib/hpux64/libstd.a` - 64-bit archive version
- HP aC++ Runtime Support Library
  - `/usr/lib/hpux##/libCsup.so`
  - `/usr/lib/hpux##/libCsup11.so` — ISO C++11 standard compliant
  - `/usr/lib/hpux##/libstd.so` and `libstd_v2.so`
  - `/usr/lib/hpux##/libstd_v2.so` and `librwtool_v2.so`
  - `/usr/lib/hpux##/libstream.so`
  - Libraries in `/usr/include/hpux##` directory
- (## is 32 or 64 - provided as part of the HP-UX core system.)
- IOStream Library
  - `/usr/lib/hpux32/libstream.so` - 32-bit shared version
  - `/usr/lib/hpux32/libstream.a` - 32-bit archive version
  - `/usr/lib/hpux64/libstream.so` - 64-bit shared version
  - `/usr/lib/hpux64/libstream.a` - 64-bit archive version

10 Mixing C++ with Other Languages

This chapter provides guidelines for linking HP aC++ modules with modules written in HP C and HP FORTRAN 90 on HP 9000 Series 700/800 systems. It discusses the following topics:

Calling Other Languages

A module is a file that contains one or more variable or function declarations, one or more function definitions, or similar items logically grouped together. Mixing modules written in C++ with modules written in C is relatively straightforward since C++ is for the most part a superset of C. Mixing C++ modules with modules in languages other than C is more complicated.

When creating an executable file from a group of programs of mixed languages, one of them being C++, you must be aware of the following:

- In general, the overall control of the program must be written in C++. In other words, the `main` function should appear in a C++ module and no other outer block should be present.
- You must pay attention to case sensitivity conventions for function names in the different languages.
- You must make sure that the data types in the different languages correspond. Do not mismatch data types for parameters and return values.
- Storage layouts for aggregates differ among languages.
- You must use the `extern "C"` linkage specification to declare modules that are not written in C++; this is true whether or not the module is written in C.
- You must use the `extern "C"` linkage specification to declare modules that are written in C++ and called from other languages.
- Do not use `extern "C"` when you include standard C header files because these header files already contain `extern "C"` directives.

NOTE: HP aC++ classes are not accessible to non-C++ routines.

Data Compatibility between C and C++

Many of the data types between C and C++ are identical as C++ is, for most part, a superset of C. Both languages support `char`, `short`, `int`, `long`, `float`, and `double` data types. ANSI C and HP C++ also support a `long double` type. In addition, HP aC++ supports `bool`, `wchar_t`, `long long`, and `unsigned long long` data types.

Pointers, structs, and unions that can be declared in C are also compatible. Arrays composed of any of the above types are compatible.

C++ classes are generally incompatible with C structs. The following features of the C++ class facility may cause the compiler to generate extra code, extra fields, or data tables:

- Multiple visibility of members (including both private and public data members in a class)
- Inheritance, either single or multiple
- Virtual functions

It is the use of these features, as opposed to whether the `class` keyword is used rather than `struct`, that introduces incompatibilities with C structs.

HP aC++ Calling HP C

Calling between C and C++ is a normal operation, since C++ is for the most part a superset of C. You should, however, be aware of the following:

- Using the extern "C" Linkage Specification
- Differences in Argument Passing Conventions
- The main() Function

Using the extern "C" Linkage Specification

To handle overloaded function names the HP aC++ compiler generates new, unique names for all functions declared in a C++ program. To geneate these names, the compiler uses a function-name encoding scheme that is implementation dependent. A linkage directive tells the compiler to inhibit this default encoding of a function name for a particular function.

To call a C function from a C++ program, you must disable the usual encoding scheme when you declare the C function. When you do not disable the usual encoding scheme, the function name declared in your C++ program will not match the function name in your C module defining the function.

When the names do not match, the linker cannot resolve them. To avoid these linkage problems, use a linkage directive when you declare the C function in the C++ program.

Syntax of extern "C"

All HP aC++ linkage directives must have either of the following formats:

```
extern “C” function_declaration

extern “C”
     {
     function_declaration1
     function_declaration2
          ...
     function_declarationN
     }
```

Examples of extern "C"

The following declarations are equivalent:

```
extern “C” char* get_name(); // declare the external C module
```

and

```
extern “C”
{
 char* get_name();   // declare the external C module
}
```

You can also use a linkage directive with all the functions in a file, as shown in the following example. This is useful when you use C library functions in a C++ program.

```
extern “C”
{
 #include “myclibrary.h”
}
```

NOTE: Do not use `extern "C"` when you include standard C header files. These header files already contain `extern "C"` directives.

Although the string literal following the extern keyword in a linkage directive is implementation-dependent, all implementations must support C and C++ string literals. Refer to linkage specifications in *The C++ Programming Language, Third Edition* for more information.

Differences in Argument Passing Conventions

When your C++ code calls functions written in C, ensure that the called C functions do not use function prototypes that suppress argument widening. If they do, your C++ code passes arguments wider than the expectations of your C code.

The main() Function

When you mix C++ modules with C modules, the overall control of the program must be written in C++. The `main` function should appear in a C++ module, rather than in a C module. There are two exceptions:

1. C++ programs and libraries, including HP-supplied libraries, without any global class objects containing constructors or destructors.
2. C++ programs and libraries, including HP-supplied libraries, without static objects.

Examples: HP aC++ Calling HP C

The following examples show a C++ program, `calling_c.C` that calls a C function, `get_name`. The C++ program contains a `main` function.

```
//**************************************************************
// This is a C++ program that illustrates calling a function *
// written in C. It calls the get_name() function, which is  *
// in the "get_name.c" module. The object modules generated  *
// by compiling the "calling_c.C" module and by compiling    *
// the "get_name.c" module must be linked to create an       *
// executable file.                                          *
//**************************************************************
#include <iostream.h>
#include "string.h"
//**************************************************************
// declare the external C module
extern "C" char* get_name();
class account
{
private:
    char* name;         // owner of the account
protected:
    double balance;     // amount of money in the account
public:
    account(char* c)    // constructor
        { name = new char [strlen(c) +1];
          strcpy(name,c);
          balance = 0; }
    void display()
        { cout << name << " has a balance of "
            << balance << "\n"; }
};
int main()
{
  account* my_checking_acct = new account (get_name());
  // send a message to my_checking_account to display itself
  my_checking_acct->display();
}
```

The following example shows the module get_name.c. This function is called by the C++ program.

```
/*****************************************************/
/* This is a C function that is called by main() in */
/* a C++ module, "calling_c.C". The object          */
/* modules generated by compiling this module and   */
/* by compiling the "calling_c.C" module must be    */
/* linked to create an executable file.             */
/***************************************************/
#include <stdio.h>
#include "string.h"
char* get_name()
{
  static char name[80];
  printf("Enter the name: ");
  scanf("%s",name);
  return name;
}
/***************************************************/
```

Running the Example

Following is a sample run of the executable file that results when you link the object modules generated by compiling `calling_c.C` and `get_name.c`:

```
Enter the name:Joann
```

```
Joann has a balance of 0
```

HP C Calling HP aC++

If you mix C++ modules with C modules, refer to "Linking Your HP aC++ Libraries with Other Languages" (page 196).

Since most C++ programs use the HP aC++ run-time libraries, you can call a C++ module from a C module using the following procedure:

- To prevent a function name from being mangled, the function definition and all declarations used by the C++ code must use `extern "C"`.
- You cannot call member functions of classes in C++ from C. When a member function routine is needed, call a non-member function in C++. This in turn calls the member function.
- Since the C program cannot directly create or destroy C++ objects, it is the responsibility of the writer of the C++ class library to define interface routines that call constructors and destructors, and it is the responsibility of the C user to call these interface routines to create such objects before using them and to destroy them afterwards.
- The C user should not try to define an equivalent struct definition for the class definition in C++. The class definition may contain bookkeeping information that is not guaranteed to work on every architecture. All access to members must be done in the C++ module.

The following examples illustrate some of these points, as well as reference parameters in the interface routine to the constructor.

```
//************************************************
//  C++ module that manipulates object obj.      *
//************************************************
#include <iostream.h>

typedef class obj* obj_ptr;

extern "C" void initialize_obj (obj_ptr& p);
extern "C" void delete_obj (obj_ptr p);
extern "C" void print_obj (obj_ptr p);

struct obj {
private:
     int x;
public:
     obj() {x = 7;}
     friend void print_obj(obj_ptr p);
};
```

```
// C interface routine to initialize an
// object by calling the constructor.
void initialize_obj(obj_ptr& p) {
     p = new obj;
}

// C interface routine to destroy an
// object by calling the destructor.
void delete_obj(obj_ptr p) {
     delete p;
}

// C interface routine to display
// manipulating the object.
void print_obj(obj_ptr p) {
cout << "the value of object->x is " << p->x << "\n";
}
```

Following is a C program that calls the C++ module to manipulate an object:

```
/***************************************************/
/* C program to demonstrate an interface to the    */
/* C++ module.  Note that the application needs    */
/* to be linked with the aCC driver.               */
/***************************************************/
typedef struct obj* obj_ptr;

int main () {
     /* C++ object. Notice that none of the
        routines should try to manipulate the fields.
     */
     obj_ptr f;

/* The first executable statement needs to be a call
   to _main so that static objects will be created in
   libraries that have constructors defined.  In this
   application, the stream library contains data
   elements that match the conditions.
*/

/* NOTE: In 64-bit mode, you MUST NOT call _main. */

#if !defined(__LP64__) && !defined(__ia64)
     _main();
#endif

     /* Initialize the data object. Notice taking
        the address of f is compatible with the
        C++ reference construct.
     */
     initialize_obj(&f);

     /*  Call the routine to manipulate the fields */
     print_obj(f);

     /*  Destroy the data object */
     delete_obj(f);
}
```

Compiling and Running the Sample Programs

To compile the example, run the following commands:

`cc -c`*`cfilename.c`*

`aCC -c`*`C++filename.C`*

```
aCC -oexecutable cfilename.o C++filename.o
```

NOTE: During the linking phase, the aCC driver program performs several functions to support the C++ class mechanism. Linking programs that use classes with the C compiler driver `cc` leads to unpredictable results at run time.

Calling HP FORTRAN 90 from HP aC++

This section discusses the following topics:

- The main() Function
- Function Naming Conventions
- Using Reference Variables to Pass Arguments
- Using extern "C" Linkage
- Strings
- Arrays
- Files in FORTRAN

NOTE: As is the case with calling HP C from HP aC++, you must link your application using HP aC++.

The main() Function

In general, when you mix C++ modules with modules in HP FORTRAN 90, the overall control of the program must be written in C++. In other words, the `main` function must appear in a C++ module, and no other outer block should be present.

Function Naming Conventions

When you call an HP FORTRAN 90 function from HP aC++, keep in mind the differences in handling case sensitivity. HP FORTRAN 90 is not case sensitive, while HP aC++ is case sensitive. Therefore, all C++ global names accessed by FORTRAN 90 routines must be lowercase. All FORTRAN 90 external names are downshifted by default.

Using Reference Variables to Pass Arguments

There are two methods of passing arguments, by reference or by value. Passing by reference means that the routine passes the address of the argument rather than the value of the argument.

When calling HP FORTRAN 90 functions from HP aC++, ensure that the caller and called functions use the same method of argument passing for each individual argument. Furthermore, when calling external functions in HP FORTRAN 90, you must know the calling convention for the order of arguments.

HP does not recommend passing structures or classes to HP FORTRAN 90. For maximum compatibility and portability, pass simple data types to routines. All HP aC++ parameters are passed by value, as in HP C, except arrays and functions which are passed as pointers.

HP FORTRAN 90 passes all arguments by reference. This means that all actual parameters in an HP aC++ call to a FORTRAN routine must be pointers, or variables prefixed with the unary address operator, `&`.

The simplest way to reconcile these differences in argument-passing conventions is to use reference variables in your C++ code. Declaring a parameter as a reference variable in a prototype causes the compiler to pass the argument by reference when the function is invoked.

Example of Reference Variables as Arguments

The following example illustrates a reference variable:

```
int main( void )
{
     // declare a reference variable
     extern void pas_func( short & );
     short x;
          ...
     pas_func( x );    // pas_func should accept
          ...          // its parameters by reference
}
```

Using extern "C" Linkage

To mix C++ modules with HP FORTRAN 90 modules, you must use `extern "C"` linkage to declare any C++ functions that are called from a non-C++ module and to declare the FORTRAN routines.

Strings

HP aC++ strings are not the same as HP FORTRAN 90 strings. In FORTRAN 90, the strings are not null terminated. Also, strings are passed as string descriptors in FORTRAN 90. This means that the address of the character item is passed and a length by value follows.

NOTE: If you use the HP FORTRAN 90 `+800` option, the length follows immediately after the character pointer in the parameter list. If you do not use this option, HP FORTRAN 90 passes character lengths by value at the end of the parameter list.

Refer the *HP FORTRAN/9000 Programmer's Reference Guide* and *HP FORTRAN/9000 Programmer's Guide* for information about the `+800` option.

Arrays

HP aC++ stores arrays in row-major order, whereas HP FORTRAN 90 stores arrays in column-major order. The lower bound for HP aC++ is 0. The default lower bound for HP FORTRAN 90 is 1.

Files in FORTRAN

HP FORTRAN I/O routines require a logical unit number to access a file, whereas HP aC++ accesses files using HP-UX I/O subroutines and intrinsics and requires a stream pointer.

A FORTRAN logical unit cannot be passed to a C++ routine to perform I/O on the associated file; also a C++ file pointer cannot be used by a FORTRAN routine. However, a file created by a program written in either language can be used by a program of the other language if the file is declared open within the latter program. HP-UX I/O (stream I/O) can also be used from FORTRAN instead of FORTRAN I/O.

Refer to your system FORTRAN manual on inter-language calls for more information.

11 Distributing Your C++ Products

Distribute your products in such a way that your customer does not need to use the HP aC++ compiler or driver. That is, only distribute executables and shared libraries.

If you write code in HP aC++ and distribute any of the following C++ files to your customers, read the following sections for recommendations and legal requirements.

- Shared libraries containing C++ code with the exception of the following libraries:
  - `/usr/lib/hpux##/libCsup.so`
  - `/usr/lib/hpux##/libCsup11.so`
  - `/usr/lib/hpux##/libstd.so` (and `libstd_v2.so`)
  - `/usr/lib/hpux##/librwtool.so` (and `librwtool_v2.so`)
  - `/usr/lib/hpux##/libstream.so`

  where ## is 32 or 64, which are provided as part of the HP-UX core system.
- Executable files produced by HP aC++ and applications that use shared libraries provided with HP aC++.
- Object files produced by HP aC++.
- Archived libraries containing C++ code.
- Any combination of the above.

NOTE: If you choose to distribute archive libraries or object files, your customer must have purchased HP aC++. Be sure your customer has read this distribution information.

This chapter discusses the following topics:

- "Applications that use HP aC++ Shared Libraries" (page 195)
- "Linking Your HP aC++ Libraries with Other Languages" (page 196)
- "Installing your Application" (page 196)
- "HP aC++ Files You May Distribute" (page 196)
- "Terms for Distribution of HP aC++ Files" (page 197)

Applications that use HP aC++ Shared Libraries

Following lists the HP aC++ runtime libraries that are shipped as part of the HP-UX 11.x core system:

- `/usr/lib/hpux##/libCsup.so`
- `/usr/lib/hpux##/libCsup11.so`
- `/usr/lib/hpux##/libstd.so` and `libstd_v2.so`
- `/usr/lib/hpux##/librwtool.so` and `librwtool_v2.so`
- `/usr/lib/hpux##/libstream.so`
- Libraries in `/usr/include/hpux##`

where ## is 32 or 64, which are provided as part of the HP-UX core system.

NOTE: If you distribute either executable files or shared libraries as part of your product, do not ship these HP aC++ runtime libraries with your product in such a way that it results in overwriting a newer library version with an older, incompatible version. Ensure that an older library version is not installed over a newer one.

Linking Your HP aC++ Libraries with Other Languages

The C++ language requires that non-local static objects be initialized before any function or object is used. HP aC++ automatically initializes non-local static objects in all object files, including shared libraries, before the first statement in `main()` executes. No special files or link options are needed.

If the library is being dynamically loaded from pure C or Java as a plugin, the library should be linked with the HP aC++ runtime libraries in the following order:

```
-AA:   -lstd_v2 -lCsup -lunwind -lm
```

If `tools.h++` is used, then add `-lrwtool` (or `-lrwtool_v2`) to the left.

In addition, your customers must review Chapter 10: "Mixing C++ with Other Languages" (page 188) for information on linking HP aC++ modules with HP C, and HP FORTRAN 90.

Installing your Application

HP aC++ releases are usually forward compatible, but HP cannot guarantee that this will be true for all releases. If you have questions about the compatibility of HP aC++ releases, you should contact your HP support representative.

Normally your customer will already have the correct runtime installed. If your product requires a newer version, it is recommended that the customer install the latest runtime patch.

Your application's installation procedure should install the appropriate HP aC++ components in the standard directories on your customer's systems. This will ensure that the `aCC` command can find them.

NOTE: If your customer already has HP aC++ installed and their version is newer than yours, do not overwrite any of the existing HP aC++ components. Do not install your product on a system that has a newer version of HP aC++ if that newer version is incompatible with your version.

Ensure that your customer does not install a version of HP aC++ after installing your product; if that version of HP aC++ is incompatible with your version.

HP aC++ Files You May Distribute

You can package and redistribute the following HP aC++ components to your customers. The following HP aC++ runtime libraries are provided as a patch to the HP-UX core system:

- `/usr/lib/hpux##/libCsup.so`
- `/usr/lib/hpux##/libCsup11.so`
- `/usr/lib/hpux##/libstd.so` and `libstd_v2.so`
- `/usr/lib/hpux##/librwtool.so` and `librwtool_v2.so`
- `/usr/lib/hpux##/libstream.so`

where ## is 32 or 64, which are provided as part of the HP-UX core system.

Terms for Distribution of HP aC++ Files

Permission to distribute the above mentioned HP aC++ runtime shared libraries is based on the following terms and conditions:

- These HP aC++ components cannot be redistributed as part of a C++ compiler, linker, or interpreter product.
- All copyright notices in the code must be retained.
- The HP aC++ executable components can only be redistributed by HP aC++ customers.

12 Migrating from HP C++ (cfront) to HP aC++

This chapter discusses differences in syntax and functionality that you need to consider when migrating from HP C++ (cfront) to HP aC++.

It discusses the following topics:

NOTE: The HP C++ and HP aC++ compilers execute independently and can be installed on a single system. HP C++ is located at `/opt/CC`. HP aC++ is located at `/opt/aCC`.

General Guidelines for Migration

Because of incompatibilities in areas such as name mangling, libraries, and object layout, all of your C++ code for an application or library must be compiled and linked with either HP C++ (cfront) or with HP aC++. You cannot mix object files compiled with HP C++ (cfront) with those compiled with HP aC++.

This section discusses the general guidelines when migrating from HP C++ (cfront) to HP aC++.

Getting Started with Migration

Complete the following procedure to migrate your code from HP C++ (cfront) to HP aC++:

1. Compile your code with the HP C++ (cfront) compiler using the `+p` option. This option requests the compiler to treat anachronistic constructs as errors. Fix the anachronisms. For example:

   ```
   CC +p cfrontfile.C
   ```

2. In your Makefiles:
   - Change `CC` to `aCC`.
   - Set the path to`/opt/aCC/bin`.
   - Review command-line options and change when necessary.
3. Compile and fix syntax errors.
   - Note that cfront-generated object code and libraries are not compatible with those produced by aCC.
   - If your program uses `operator new`, allow for memory allocation exceptions that may occur. Modify your cfront code to handle memory allocation failures to avoid a program abort.
4. Make library changes. Begin migration to the Standard C++ Library and Tools.h++ Library.
5. Make template changes.
   - If a program or library uses templates, consider source code changes that may be required to direct template instantiation.
   - Use the `+inst_directed` option with the initial compilation to defer consideration of compile-time errors due to template instantiation.

Writing Code for both Compilers

Use the __cplusplus macro (defined by the draft standard) to write code that can be compiled by both HP C++ and HP aC++.

Example:

```
#if __cplusplus >= 199707L

   // HP aC++ code

#else

   // HP C++ code

#endif // __cplusplus >= 199707L
```

Explicit Loading and Unloading of Shared Libraries

HP aC++ uses system calls rather than C++ function calls to explicitly load and unload shared libraries. When migrating to HP aC++, make the following source code changes:

- Change `cxxshl_load()` to `shl_load()`.
- Change `cxxshl_unload()` to `shl_unload()`.
- Change `#include <cxxdl.h>` to `#include <dl.h>`.

Memory Allocation

See "Memory Allocation Failure and operator new" (page 203) for more information.

Command-Line Differences

In HP aC++, you invoke the compiler with the `aCC` command instead of the `CC` command used to invoke HP C++.

The following sections describe differences in command-line options:

- "New Command-Line Options" (page 199)
- "Obsolete Command-Line Options" (page 200)
- "Changed Command-Line Options" (page 201)

New Command-Line Options

Table 12 describes the new options for HP aC++. These options are not available for HP C++ (cfront). However, if a related option exists, it is noted here.

See Chapter 2: "Command-Line Options" (page 31) to for a complete list of HP aC++ command-line options.

Table 12 New Command-Line Options

Option	Description
`-g0`	Replaces the `-g` debugger option. It generates complete debug information for the debugger.
`+help`	Invokes the HP aC++ Online Programmer's Guide.
`+noeh`	Disables exception handling. In HP aC++, exception handling is enabled by default. In HP C++ (cfront), exception handling is disabled by default. To enable it, use the `+eh` option, which is obsolete in HP aC++.

Table 12 New Command-Line Options *(continued)*

Option	Description
Precompiled Header File Options	Reduces compilation time and executable file size by precompiling common `include` (header) files.
Template Options	There are new options and new functionality for template processing. For more information about HP aC++ templates, see Chapter 5: "Using HP aC++ Templates" (page 132).

Obsolete Command-Line Options

Table 13 describes obsolete command-line options for HP aC++.

Table 13 Obsolete Command-Line Options

Option	Description
Debugging Option	
`-y`	In HP C++ (cfront), the `-y` option generates a Static Analysis database if SoftBench is installed and `/opt/softbench/bin` is at the beginning of your path. The option is not required in HP aC++.
Exception Handling Option	
`+eh`	Enables exception handling in HP C++. In HP aC++, exception handling is enabled by default. To disable exception handling off, compile with the `+noeh` option.
Library Option	
`-depth`	In HP C++, this option instructs runtime system to traverse the shared library list in a *depth-first* manner when calling static constructors and when loading the libraries. The default is to traverse the shared libraries in a left-to-right order when calling static constructors. The order of execution of static constructors within each shared library is not affected by this option. In HP aC++, `-depth` functionality is the default option.
Preprocessor Options	
`-Ac`	Requests the compatibility mode HP C++ preprocessor, `cpp`. This option is not available in HP aC++.
`-C`	Prevents the preprocessor from stripping comments from your source file. In HP aC++ comments are retained.
`-Wp`	The `-W` option no longer accepts `p` as a subprocess parameter. In HP aC++, there is no separate subprocess for the preprocessor. Use the `CC` command (HP C++) as a workaround: **Example:** `CC prog.C -I /opt/aCC/include -I /opt/aCC/include/iostream -I /usr -I /usr/include` See "Migration Considerations Related to Preprocessing" (page 208) for more information.
Template Options	
`-pta`	Instantiates all members of used template classes and all needed template functions.
`-ptb`	Invokes `ld` instead of `nm` to do simulated linking.
`-pth`	Uses short file names for template instantiation files.
`-ptH"list"`	Specifies file name extensions for template declaration files (header files).
`-ptn`	Instantiates at link time rather than at compile time.
`-ptrpath`	Specifies an alternate location for the template repository.

Table 13 Obsolete Command-Line Options *(continued)*

Option	Description
`-pts`	Splits template instantiations into separate object files.
`-ptS"list"`	Specifies file name extensions for template definition files.
`-ptv`	Provides verbose information about template processing. For HP aC++, use the `+inst v` option.
Translator Mode Options	
`+a0`	Causes the translator to produce `Classic C` style declarations.
`+a1`	Causes the translator to produce ANSI C style declarations.
`-F`	Runs only the preprocessor and translator, and sends the resulting source code to standard output (`stdout`).
`-Fc`	Similar to the `-F` option, except that C source code is generated.
`+i`	Generates an intermediate C language source file that has the file name suffix `..c` in the current directory.
`+m`	Provides maximum compatibility with the USL C++ implementation.
`+Rnumber`	Promotes only first number register variables to the register class.
`+T`	Requests translator mode.
`+xfile`	Reads a file of data types, sizes, and alignments that the compiler uses when generating code.
Virtual Table Options	
`+e0`	Causes virtual tables to be external and defined elsewhere, that is, uninitialized.
`+e1`	Causes virtual tables to be declared externally and defined in a given module, that is initialized.

Changed Command-Line Options

Functionality for the following options is different for HP C++ (cfront) than it is for HP aC++. Table 14 lists and describes the obsolete command-line options for HP aC++.

Table 14 Changed Command-Line Options

Option	Description
`-E`	Runs the preprocessor only on named C++ files, not on assembly files, and sends the result to standard output (`stdout`).
`-g`	Generates minimal information for the debugger (as does the `-g1` option). This is the default option. The `-g0` option replaces `-g` and generates complete debug information for the debugger. See "Debugging Options" (page 35) for more information.
`-tx,name`	The following values for x are related to translator mode and template subprocesses and are not supported in HP aC++. • `0` (zero) - C compiler • `c` - C compiler • `i` - Link-time template processor, `c++ptlink` • `m` - merge tool, `c++merge` • `p` - preprocessor

Table 14 Changed Command-Line Options *(continued)*

Option	Description
	• `P` - patch tool, `c++patch` • `r` - Compile-time template processor, `c++ptcomp`
`-Wx,args`	The following values for x are related to translator mode and template subprocesses and are not supported in HP aC++. • `0` (zero) - C compiler • `c` - C compiler • `i` - Link-time template processor, `c++ptlink` • `m` - merge tool, `c++merge` • `p` - preprocessor • `P` - patch tool, `c++patch` • `r` - Compile-time template processor, `c++ptcomp`

Migration Considerations when Debugging

The HP/DDE Debugger supports HP aC++. The HP Symbolic Debugger, `xdb`, is not supported. Functionality of the `-g` debugger option has changed. It now generates minimal information for the debugger as does the `-g1` option. This is the default.

The `-g0` option replaces the `-g` option and generates full debug information for the debugger.

See "Debugging Options" (page 35) for complete information.

Migration Considerations when Using Exception Handling

When migrating exception handling code, the following characteristics of HP aC++ differ from those of HP C++ (cfront):

- "Exception Handling is the Default" (page 202)
- "Memory Allocation Failure and operator new" (page 203)
- "Possible Differences when Exiting a Signal Handler" (page 203)
- "Differences in setjmp/longjmp Behavior" (page 204)
- "Calling unexpected" (page 204)
- "Unreachable catch Clauses" (page 205)
- "Throwing an Object having an Ambiguous Base Class" (page 205)

Exception Handling is the Default

In HP aC++ exception handling is enabled by default. Use the `+noeh` option to disable exception handling.

NOTE: With exception handling disabled, the keywords `throw` and `try` generate a compiler error. The HP C++ (cfront) compiler, behaves differently; the default is exception handling off. To turn it on, you must use the `+eh` option.

If your executable throws no exceptions, object files compiled with and without the `+noeh` option can be mixed freely. However, in an executable that throws exceptions (HP aC++ runtime libraries throw exceptions), you must be certain that no exception is thrown in your application which will unwind through a function compiled without the exception handling option turned on.

In order to prevent this, the call graph for the program must never have calls from functions compiled without exception handling to functions compiled with exception handling (either direct calls or calls made through a callback mechanism). If such calls do exist, and an exception is thrown, the unwinding can cause:

- non-destruction of local objects (including compiler generated temporaries)
- memory leaks when destructors are not executed
- runtime errors when no catch clause is found

Memory Allocation Failure and operator new

In HP aC++ programs, when either `operator new ( )` or `operator new [ ]` cannot obtain a block of storage, a `bad_alloc` exception results. This is required by the ANSI/ISO C++ International Standard.

In HP C++, memory allocation failures return a null pointer (zero) to the caller of `operator new ()`.

To handle memory allocation failures in HP aC++ and to avoid a program abort, do one of the following:

- Write `try` or `catch` clauses to handle the `bad_alloc` exception.
- Use the `nothrow_t` parameter to specify the type when calling operator new and check for a null pointer.

Example:

```
operator new (size_t size, const nothrow_t &) throw();
operator new [] (size_t size, const nothrow_t &) throw();
    .
    .
    .
#include <new.h>
#include <stdexcept>

class X{};

void foo1() {
 X* xp1 = new(nothrow())X;    // returns 0 when creating a nothrow
                              // object, if space is not allocated
}

void foo2() {
 X* xp2 = newX:               // may throw bad_alloc
}

int main() {
  try {
    foo1();
    foo2();
  }

  catch (bad_alloc) {
    // code to handle bad_alloc
  }
  catch(...) {
    // code to handle all other exceptions
  }
}
```

Possible Differences when Exiting a Signal Handler

Behavior when exiting a signal handler through a `throw` may differ between the two compilers.

In HP aC++, a `try` block begins following the first call after the `try` keyword. This conforms to the standard that a legal exception cannot be thrown prior to the first call. The current handlers of `try` block are considered candidates to catch the exception.

In HP C++ the `try` keyword defines the beginning of a `try` block.

In this situation, when a signal is taken while executing between the `try` keyword and the return point of the first call, a throw from the signal handler does not find the associated handlers as candidates for catching the exception.

Differences in setjmp/longjmp Behavior

Interoperability with `setjmp/longjmp` is not implemented.

The standard specifies that an implementation need not clean up objects whose lifetimes are shortened by a `longjmp`:

The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. When automatic objects are destroyed by a thrown exception, transferring control to a destination point in the program, a call to `longjmp(jbuf, val)` at the throw point transfers control to the destination point results in undefined behavior.

Calling unexpected

Unlike HP C++, in HP aC++, when an unexpected handler wants to exit through a throw, it must throw an exception that is legal according to the exception specification that calls `unexpected()`, unless that exception specification includes the predefined type `bad_exception`. If it includes `bad_exception`, and the type thrown from the unexpected handler is not in the exception specification, then the thrown object is replaced by a `bad_exception` object and throw processing continues.

The following example is legal in HP C++ but not in HP aC++. You can make the example legal by including the exception header and adding `bad_exception` to `foo`'s throw specification. The `catch(...)` in `main` will then catch a `bad_exception` object. This is the only legal way an unexpected-handler can rethrow the original exception.

```
//     #include <exception>      Needed to make the example legal.

void my_unexpected_handler() { throw; }

void foo() throw() {

//     void foo() throw(bad_exception) {   To make the example legal,
//                                          replace the previous line
//                                          of code with this line.

throw 1000;
}

int main() {
set_unexpected( my_unexpected_handler );
try {
foo();
}
catch(...) {
printf("fail - not legal in aCC\n");
}
return 0;
}
```

Following is an example, illegal because `my_unexpected_handler` rethrows an `int`. A possible conversion is to throw `&x` instead, as this is a pointer to `int` and therefore legal with respect to the original throw specification. Alternatively, you can add `bad_exception` to the throw specification, as in the prior example.

```
int x = 1000;

void my_unexpected_handler() { throw; }

void foo() throw( int * ) {
throw 1000;
}

int main() {
set_unexpected( my_unexpected_handler );
try {
foo();
}
catch(...) {
printf("fail - not legal in aCC\n");
}
return 0;
}
```

Unreachable catch Clauses

Unreachable `catch` clauses are diagnosed by HP C++ but not by HP aC++. For example,

```
class C {
// ...
};

class D : public C {
// ...
};

...

catch(C) {
}
catch(D) {          // Unreachable since previous catch masks this one.
                    // Throw of D will be caught by catch for base class.
}

catch(C * ) {
}
catch(D * ) {       // Unreachable since previous catch masks this one.
                    // Throw of D * will be caught by catch for pointer
                    // to base class.)
}
```

Throwing an Object having an Ambiguous Base Class

HP C++ generates an `object throw` error that has an ambiguous base class. In HP aC++, a `throw` of an object having an ambiguous base class is not caught by a handler for that base, since that would involve a prohibited derived-to-base conversion.

In the following example, the throws are caught by the handlers for `D1` and `D1*`, respectively. The handlers for `C` are disqualified because `C` is an ambiguous base class of `E`:

```
extern "C" int printf(char*,...);

class C {
public:
C() {};
};

class D1 : public C {
public:
D1() {};
};
```

```
class D2 : public C {
public:
D2() {};
};

class E: public D1, public D2 {
public:
E() {};
};

int main() {
E e;
try {
throw e;
}
catch(C) {
printf("caught a C object\n");
}
catch(D1) {
printf("caught a D1 object\n");
}
catch(D2) {
printf("caught a D2 object\n");
}
catch(E) {
printf("caught an E object\n");
}

try {
throw & e;
}
catch(C*) {
printf("caught ptr to C object\n");
}
catch(D1*) {
printf("caught ptr to D1 object\n");
}
catch(D2*) {
printf("caught ptr to D2 object\n");
}
catch(E*) {
printf("caught ptr to E object\n");
}
return 0;
}
```

Migration Considerations when Using Libraries

The following sections contain information about library migration from HP C++ (cfront) to HP aC++.

Standards Based Libraries

HP aC++ provides the following libraries that are not part of the HP C++ (cfront) compiler:

- Standard C++ Library
- Tools.h++ Library
- HP aC++ Runtime Support Library

HP recommends that you use these standards based libraries whenever possible, instead of the cfront compatibility libraries. See Chapter 9: “Tools and Libraries” (page 175) for more information.

HP C++ (cfront) Compatibility Libraries

HP aC++ provides the following library, whose functionality is part of the HP C++ (cfront) compiler. This library is not Standards based.

- IOStream Library

IOStream Library

The shared version of this library is located at `/usr/lib/hpux##/libstream.so`. The archive version is at `/usr/lib/hpux##/libstream.a`. (## is `32` or `64` - provided as part of the HP-UX core system).

Manpages

The following manpages are located in the `/opt/aCC/share/man/man3.Z` directory:

- `IOS.INTRO(3C++)` - Introduction to the C++ stream library
- `filebuf(3C++)` - Buffer for file input and output
- `fstream(3C++)` - `iostream` and `streambuf` specialized to files
- `ios(3C++)` - input/output formatting
- `istream(3C++)` - formatted and unformatted input
- `manip(3C++)` - `iostream` manipulators
- `ostream(3C++)` - insertion (storing) into a `streambuf`
- `sbuf.prot(3C++)` - interface for derived classes
- `sbuf.pub(3C++)` - public interface of character buffering class
- `ssbuf(3C++)` - `streambuf` specialized to arrays
- `stdiobuf(3C++)` - `iostream` specialized to `stdio` file
- `strstream(3C++)` - `iostream` specialized to arrays

To invoke a manpage from the command line, enter `3s` after the `man` command and before the manpage name.

Example:

Enter the following command to invoke the manpage for `filebuf`:

```
man 3s filebuf
```

Header Files

Use the following header files with the IOStream library.

- `iostream.h` - I/O streams classes `ios`, `istream`, `ostream`, and `streambuf`
- `fstream.h`
- `strstream.h` - `streambuf` specialized to arrays
- `iomanip.h` - predefined manipulators and macros
- `stdiostream.h` - specialized streams and streambufs for interaction with `stdio`
- `stream.h` - includes `iostream.h`, `fstream.h`, `stdiostream.h` and `iomanip.h` for compatibility with AT&T USL C++ v 1.2

To direct the compiler to search these header files, enter the following command:

```
-I/opt/aCC/include/iostream.
```

Standard Components Library Not Provided

The Standard Components Library is not provided with the HP aC++ compiler for Integrity servers. HP recommends that you use the similar features of the Standard C++ Library in place of the Standard Components Library.

HP C++ (cfront) Complex Library Not Supported

The Complex library which is part of the cfront based HP C++ compiler product is not included with HP aC++. HP recommendeds that you use the similar features of the Standard C++ Library in place of the Complex library.

Complete the following procedure to begin migration:

1. Replace `#include` with `<complex>`.
2. Remove `-lcomplex` from the command-line.

Manpages

The following manpages describe the complex library are not part of the HP aC++ product. These manpages are no longer available.

- `CPLX.INTRO(3C++)` - Introduction to The C++ Complex Mathematics Library
- `cartpol(3C++)` - Partesian and Polar Functions
- `cplxexp(3C++)` - Exponential, Logarithm, Power, and Square Root Functions for Complex Numbers
- `cplxerr(3C++)` - Error Handling Function
- `cplxops(3C++)` - Complex Number Operators
- `cplxtrig(3C++)` - Trigonometric and Hyperbolic Functions for Complex Numbers

Header File

The Complex library uses the `complex.h` header file.

HP C++ (cfront) Task Library Not Supported

The task library, that is part of the HP C++, is not included with HP aC++. To develop multi-threaded applications with HP aC++, use the `pthread`programming interface routines that are available as part of HP DCE/9000.

Manpages

The following manpages describe task library features are not part of the HP aC++ product. These manpages are no longer available.

- `TASK.INTRO(3C++)` - Introduction to the C++ Task Library
- `interrupt(3C++)` - Signal Handling
- `queue(3C++)` - Queue Routines for Message Passing and Data Buffering
- `task(3C++)` - the C++ task library
- `tasksim(3C++)` - Histograms and Random Numbers for Simulations with C++ Tasks

Migration Considerations Related to Preprocessing

The HP C++ (cfront) compiler provides ANSI mode (the default) and K&R compatibility mode preprocessing. HP aC++ preprocessing complies with the ANSI/ISO C++ International Standard. Therefore, if you are migrating from cfront ANSI mode preprocessing to HP aC++, in general, no changes are required.

HP aC++ does not support K&R compatibility mode preprocessing.

Obsolete Preprocessor Options

HP aC++ provides support for ANSI/ISO C++ International Standard preprocessing. Since the standard categorizes support of pre-ISO preprocessing as an anachronism, the ANSI preprocessing options of HP C++ (cfront) are not supported. For a list of obsolete preprocessor options, see Table 13: "Obsolete Command-Line Options" (page 200).

Migration Considerations Related to Standardization

The ANSI/ISO C++ International Standard redefines the rules, syntax, and features of C++ language. If your existing code contains any of the standards based keywords as variable names, you must change the variable names when you convert your program to an HP aC++ program. In addition to keyword changes, there are changes in C++ Semantics and C++ Syntax.

Changes in C++ Semantics

Following lists the differences in code behavior when you migrate from HP C++ to HP aC++:

- Implicit Typing of Character String Literals
- Overload Resolution Ambiguity of Subscripting Operator
- Execution Order of Static Constructors in Shared Libraries
- More Frequent Inlining of Inline Code

NOTE: These differences can occur inspite of compiling your code without errors.

Implicit Typing of Character String Literals

HP C++ implicitly types character `string` literals as `char *`. HP aC++, in accordance with the ANSI/ISO C++ International Standard, types character `string` literals as `const char *`. This difference affects function overloading resolution.

Example:

In the following code, HP aC++ calls the first function `a`; cfront calls the second.

```
void a(const char *);
void a(char *);

f() {
    a("A_STRING");
    }
```

To prevent existing code from breaking, assign a string literal to a non-const pointer.

Example:

```
char *p = "B_STRING";
```

NOTE: This feature may not be a part of the Standard in future revisions.

Also, you cannot convert `const char *` to `char *`in a conditional expression in this context.

Example:

```
char *p = f() ? "A" : "B";
```

In such a scenario, you must change the code.

Example:

```
const char *p = f() ? "A" : "B";
```

or

```
char *p = const_cast(f() ? "A" : "B");
```

Overload Resolution Ambiguity of Subscripting Operator

HP C++ and HP aC++ have different overload resolution models. When you migrate to HP aC++, you may see an overload resolution ambiguity for the subscripting operator. The following code illustrates the problem:

```
struct String {
   char& operator[](unsigned);
   operator char*();
// ...
};

void f(String &s) {
   s[0] = '0';
}
```

HP C++ accepts the above code, selecting `String::operator[](unsigned)` rather than the user-defined conversion, `String::operator char*()`, followed by the `built-in operator[]`.

Compiling the code with HP aC++ produces the following error:

```
 Error 225: "c.C", line 8 # Ambiguous overloaded function call;
       more than one acceptable function found. Two such functions
       that matched were "char &String::operator [](unsigned int)"
       ["c.C", line 2] and "char &operator [](char *,int)"
       [Built-in operator].
               s[0] = '0';
```

The error message appears because the compiler cannot choose between:

1. Not converting `s`, but converting `0` from type `int` to type `unsigned int`; this implies using the user- provided subscript `operator[]`
2. Converting `s` to type `char*` (using the user-defined conversion operator), but not converting `0`; this corresponds to using the built-in subscript `operator[]`.

In order to disambiguate this situation in favor of the user-provided subscript `operator[]`, make the conversion of `0` in alternative (1.) no worse[1] than the conversion of `0` in alternative (2.).

Because the subscript type for the built-in `operator[]` is `ptrdiff_t` (as defined in `<stddef.h>`), this is also the type that should be used for user-defined subscript operators. Replace the previous example by:

```
#include <stddef.h>

struct String {
   char& operator[](ptrdiff_t);
   operator char*();
   // ...
};

void f(String &s) {
   s[0] = '0';
}
```

Execution Order of Static Constructors in Shared Libraries

In HP C++ (cfront), static constructors in shared libraries listed on the link-line are executed, by default, in left-to-right order. HP aC++ executes static constructors in depth-first order; that is, shared libraries on which other files depend are initialized first. Use the `-depth` command-line option on the `CC` command line for enhanced compatibility with HP aC++.

In addition, HP aC++ reverses the initialization order of `.o` files on the link-line. To aid in migration, you can group all `.o` files together and all `.so` files together.

1. *worse* is relative to a ranking of conversions as described in the ANSI/ISO C++ International Standard on overloading. In general, a user-defined conversion is worse than a standard conversion, which in turn is worse than no conversion at all. The complete rules are more fine grained.

Example:

```
aCC file1.o file2.o lib1.so lib2.so lib3.so
```

In this scenario, cfront would initialize `file2.o` first, and then `file1.o`, while HP aC++ initializes `file1.o` and then `file2.o`. You must take this into account in your cfront code to avoid link problems with HP aC++.

More Frequent Inlining of Inline Code

HP C++ does not inline some functions even when you request for it. This happens when the function is too complex. If you use the `+w` option, the compiler displays a message whenever it does not inline a requested function.

HP aC++ almost always inlines functions for which you have specified the `inline` keyword.

Changes in C++ Syntax

When you migrate from HP C++ to HP aC++, in addition to changes related to standards based keywords, you may need to make changes to your source code in the following areas:

- "Explicit int Declaration" (page 211)
- "The for Statement, New Scoping Rules" (page 212)
- "struct as Template Type Parameter is Permitted" (page 212)
- "Base Template Class Reference Syntax Change" (page 213)
- "Tokens after #endif" (page 213)
- "overload not a Keyword" (page 213)
- "Dangling Comma in enum" (page 214)
- "Static Member Definition Required" (page 214)
- "Declaring friend Classes" (page 214)
- "Incorrect Syntax for Calls to operator new" (page 215)
- "Using :: in Class Definitions" (page 215)
- "Duplicate Formal Argument Names" (page 215)
- "Ambiguous Function or Object Declaration" (page 215)
- "Overloaded Operations ++ and --" (page 216)
- "Reference Initialization" (page 216)
- "Using operator new to Allocate Arrays" (page 217)
- "Parentheses in Static Member Initialization List" (page 217)
- "&qualified-id Required in Static Member Initialization List" (page 218)
- "Non-constant Reference Initialization" (page 218)
- "Digraph White Space Separators" (page 219)

Explicit int Declaration

In HP C++, you do not need to explicitly specify `int` types. In HP aC++, you must explicitly declare `int` types. This change reduces ambiguity among expressions involving function-like casts and declarations.

Example:

The following code is valid in HP C++:

```
void f(const parm);
const n = 3;
main()
```

The equivalent, valid HP aC++ code follows:

```
void f(const int parm);
const int n = 3;
int main()
```

The for Statement, New Scoping Rules

In HP C++, variables declared in the initializer list are allowed after the `for` statement. In the ANSI/ISO C++ International Standard, variables declared in the initializer list are not allowed after the `for` statement. HP aC++ provides this functionality when you specify the following `aCC` command-line option:

`-WC,-ansi_for_scope,on`

If you do not specify this option, (or you specify the `-WC,-ansi_for_scope,off` option), by default, the new rules do not take effect.

In this scenario, HP aC++ provides this standard functionality as an option to ease conversion of existing code to the standard. No code change is currently required.

Future plans are to make the ANSI/ISO C++ International Standard syntax the default. HP recommends that you correct your code, by moving the declaration of the `for` loop variable to its enclosing block.

Example:

The following code currently compiles without errors with HP C++ and HP aC++. In the future, HP aC++, will generate an error.

```
int main(int argc) {
 for (int i = 1; i < argc; ++i) {
        }
        for (i = 0; i < argc; ++i) {
        }
}
```

Correct the code as follows:

```
int main(int argc) {
 int i;
        for (i = 1; i < argc; ++i) {
        }
        for (i = 0; i < argc; ++i) {
        }
}
```

This code complies with ANSI/ISO C++ International Standard syntax and compiles with both compilers.

struct as Template Type Parameter is Permitted

In HP C++, an error is generated when a `struct` is used as a template type parameter. In HP aC++, when a `struct` is used as a template type parameter, it is correctly compiled, in accordance with draft standard syntax. This is a new feature.

Example:

```
template  class A {
public:
struct T a;
};
struct B {};
A b;
```

The following error appears when you compile this code with HP C++:

```
CC: "DDB4325.C", line 3: error: T of type any redeclared as struct
(1479)
```

This code compiles without error with HP aC++.

Base Template Class Reference Syntax Change

In HP C++, you can reference a member of a base template class without qualifying the member. In HP aC++, when you reference a member of a base template class, you must qualify the member by adding `this->`.

Adding `this->` defers name resolution until instantiation. This allows the compiler to find members in template base classes. However, it prevents the compiler from finding names declared in enclosing scopes.

Example:

```
template  class BaseT {
public:
  T t;
  int i;
};
template  class DerivedT : public BaseT {
public:
  void foo1 () { t = 1; i = 1; }               //  warning 721
                                               //  t and i could be global.

  void foo2 () { this->t = 2; this->i = 2; }   //  Correct syntax, no warning.
};
DerivedT d;             // Here is the point of instantiation.
```

Tokens after #endif

In HP C++, any character that follows the `#endif` preprocessor statement causes a warning and is ignored. In HP aC++, characters following the `#endif` preprocessor statement cause an error and the program does not compile. To change this, remove all characters following all `#endif` preprocessor statements or put the token in comments.

Example:

Compiling the following code with HP C++ causes a warning. Compiling with HP aC++ generates an error.

```
int main(){
#ifdef FLAG
int i;
i=1;
#endif FLAG
}
```

To compile with HP aC++, change the code to:

```
int main(){
#ifdef FLAG
int i;
i=1;
#endif //FLAG
}
```

overload not a Keyword

In HP C++, using the `overload` keyword to specify a function as an overloaded function causes an anachronistic warning and is ignored. In HP aC++, using the `overload` keyword causes an error and the program does not compile. To change this, remove all occurrences of the `overload` keyword.

Example:

Compiling the following code with HP C++ causes a warning. Compiling with HP aC++ generates an error stating that overload is used as a type, but has not been defined as a type.

```
int f(int i);
overload int f(float f);   // Remove the word overload.
int main () {
return 1;
}
```

Dangling Comma in enum

In HP C++, a comma following the last element in an `enum` list is ignored. In HP aC++, a comma following the last element in an `enum` list generates an error. To avoid this error, remove the comma after the last element.

Example:

HP C++ accepts the following code. HP aC++ generates an error stating that the comma (,) is unexpected.

```
enum Colors { red,
              orange,
              yellow,
              green,
              blue,
              indigo,
              violet,     // This comma is illegal.
};
```

Static Member Definition Required

In HP C++, you can declare a static member and not define it. However, in HP aC++, you cannot do so. You must define the declared static data member.

Example:

Compiling and linking the following code on HP C++ gives no warning nor error. Compiling the code on HP aC++ gives neither a warning nor an error. Linking the resulting object file generates a linker (`ld`) error that states that there are unsatisfied symbols.

```
class A {
public:
   static int staticmember;
};
// int A::staticmember=0;   // This would fix the problem.
int main ()
{
   A::staticmember=1;
}
```

Declaring friend Classes

In HP C++, you can declare `friend` classes without the `class` keyword. In HP aC++, declaring `friend` classes without the `class` keyword generates an error. To change this, add the `class` keyword to all `friend` class declarations.

Example:

Compiling the following code on HP C++ does not generate a warning or an error. Compiling the code on HP aC++ generates an error stating that the `friend` declaration for `B` is not in the right form for either a function or a class.

```
class foo{

public:
   friend bar;     // Need to say:  friend class B
};
int main (){
   return 1;
}
```

Incorrect Syntax for Calls to operator new

In HP C++, you can use incorrect syntax to call `operator new`. In HP aC++, an error is generated when incorrect syntax for `operator new` is used. To change this, add parentheses around the use of `operator new`. This code compiles correctly with both HP C++ and HP aC++.

Example:

Compiling the following code on HP C++ does not generate a warning or an error. Compiling the code on HP aC++ generates errors stating `operator` expected instead of `new` and undeclared variable operator `S`.

```
struct S {int f();};
int g() { return new S->f();}
// int g() { return (new S)->f();}    // This will fix the problem.
int S:: f( ) { return 1;}
main() {
return 1; }
```

Using :: in Class Definitions

In HP C++, you can declare members of classes inside the class using the following incorrect syntax:

```
class_name::member_name
```

In HP aC++, this incorrect syntax is considered an error. You must remove the `class_name::` specification from the member definition.

Example:

Compiling the following code on HP C++ does not generate a warning or an error. Compiling the code on HP aC++ generates an error stating that you cannot qualify members of `class X` in the class definition.

```
class X{
   int X::f();
// int f();     // This will fix the problem and
                // run successfully on both compilers.
>
int main(){
}
```

Duplicate Formal Argument Names

In HP C++, duplicate formal argument names are allowed. In HP aC++, duplicate formal argument names generate an error. To avoid this, use unique formal parameter names.

Example:

The following code compiles with HP C++. With HP aC++, an error is generated stating that symbol `aParameter` has been redefined and where it was previously defined.

```
int a(int aParameter, int * aParameter);
```

Ambiguous Function or Object Declaration

In HP C++, an ambiguous function or object declaration compiles without warning, assuming an object declaration. In HP aC++, an ambiguous function or object declaration generates an error. To change this, change the code to remove the ambiguity.

Example:

```
struct A {A(int);};
struct B {B(const A &); void g();};
void f(int p) {
   B b(A(p));   // Declaration of function or object?
   b.g();        // Error?
}
```

The ambiguity in the example code is whether `b` is declared as:

- A function with one argument (named `p`) returning an object of type `B`.
- An object of type `B` initialized with a temporary object of type `A`.

HP C++ compiles this code successfully and assumes `b` is an object. Compiling the code with HP aC++ generates the following error:

```
Error: File "objDeclaration.c", Line 5
Left side of '.' requires a class object; type found was a function 'B (A)'.
     Did you try to declare an object with a nested constructor call?
     Such a declaration is interpreted as a function declaration B b(A)
     [File "objDeclaration.c, Line 4].
```

Modify the code as shown below to successfully compile it with both compilers.

```
struct A {A(int);};
struct B {B(const A &); void g();};

void f(int p) {
   B b = A(p);    // declaration of object
   b.g();         // method call
}
```

Overloaded Operations ++ and --

You must use the overloaded operations `++` and `--` correctly. These operations require a member function with one argument. If the function has no argument, a warning is issued and a `postfic` is assumed in HP C++. In HP aC++, the inconsistency between the overloaded function usage and definition is considered an error. To avoid this error, change the class definition so that each overloaded function definition has the correct number of arguments.

Example:

```
class T {
   public:
      T();
      const T& operator++ ();
};
int main () {
T t;
t++;
}
```

Compiling the above code with HP C++ generates the following warning:

```
CC: "pre.C", Line 8: warning: prefix ++/-- used as postfix (anachronism)
(935)
```

Compiling the code with HP aC++ generates an error like the following:

```
Error 184: File "pre.C", Line 8
Arithmetic or pointer type expected for operator '++'; type found was 'T'.
```

To compile the code with HP C++ or HP aC++ use the following class definition:

```
class T {
   public:
      T();
      const T& operator++ ();     // prefix    old style postfix definition
      const T& operator++ (int);  // postfix
};
```

Reference Initialization

Illegal reference initialization is no longer allowed. In HP C++, a warning is generated stating that the initializer for a non-constant reference is not an `lvalue` (anachronism). In HP aC++, an illegal initialization of a reference type generates an error and the program does not compile. To avoid this error, use a constant reference.

Example:

```
void f() {
   char c = 1;
   int & r = c;
}
```

Compiling the above code with HP C++ generates the following warning:

```
C: "nonConstRef.C", line 6: warning: initializer for non-const
reference not an lvalue (anachronism) (235)
```

Compiling the code with HP aC++ generates an error like the following:

```
Error: File "nonConstRef.C", Line 6
Type mismatch; cannot initialize a 'int &' with a 'char'.
Try changing 'int &' to 'const int &'.
```

To successfully compile with both compilers, make the following changes to the code:

```
void f() {
    char c = 1;
    const int & r = c;
}
```

Using operator new to Allocate Arrays

In HP C++, `operator new` is called to allocate memory for an array. In HP aC++, `operator new []` is called to allocate memory for an array.

Example:

The following code compiles without error on HP C++.

```
typedef char CHAR;
typedef unsigned int size_t;
typedef const CHAR *LPCSTR, *PCSTR;
typedef unsigned char BYTE;

void* operator new (size_t nSize, LPCSTR lpszFileName, int nLine);
static char THIS_FILE[] = "mw2.C";
int main() {
   BYTE *p;
   p = new(THIS_FILE, 498) BYTE[50];
}
```

On HP aC++, the following error is generated:

```
Error: File "DDB4269.C", Line 10
Expected 1 argument(s) for void *operator new [ ](unsigned int); had 3 instead.
```

Parentheses in Static Member Initialization List

In HP C++, redundant parentheses are allowed in a static member initialization list. In HP aC++, redundant parentheses in a static member initialization list generate an error and the program does not compile. You must remove the redundant parentheses to compile the program with both compilers.

Example:

```
class A {
public:
   int i;
   static int (A::*p);
};

int (A::*(A::p)) = &(A::i);
```

Compiling this code HP aC++ generates the following error:

```
Error: File “DDB4270.C”, Line 7
A pointer to member cannot be created from a parenthesized or unqualified name.
```

To successfully compile the code, remove the parentheses from the last line.

Example:

```
class A {
public:
   int i;
   static int (A::*p);
};

int (A::*(A::p)) = &A::i;
```

&qualified-id Required in Static Member Initialization List

In HP C++, you can use an unqualified function name in a static member initialization list. In HP aC++, an unqualified function name in a static member initialization list causes an error and the program does not compile. Use the unary operator `&` followed by a qualified-id in the member initialization list. The resulting code compiles correctly with HP C++ and HP aC++.

Example:

```
class A {
public:
   int i;
   int j();
static int (A::*p)();
};
int (A::*(A::p))() = j;
```

Compiling this code with HP aC++ generates the following error:

```
Error: File “DDB4270A.C”, Line 7
Cannot initialize ‘int (A::*)()’ with ‘int (*)()’.
```

To successfully compile with HP C++ and HP aC++, change the initialization list in line 7 to `&A::j;`

```
class A {
public:
   int i;
   int j();
static int (A::*p)();
};
int (A::*(A::p))() = &A::j;
```

Non-constant Reference Initialization

In HP C++, if you do not initialize a non-constant reference with an `lvalue`, an anachronistic warning is issued and compilation continues. In HP aC++, an error is issued if you do not use an `lvalue` for a non-constant reference initialization. Use an `lvalue` for the reference initialization, or define the reference as a `const` type.

Example:

```
void f(int &);
int main () {
   f(3);
   return 0;
}
```

Compiling this code with HP C++ generates the following warning:

```
CC: “DDB04313A.C”, line 4: warning: temporary used for non-const int & argument;
    no changes will be propagated to actual argument (anachronism) (283)
```

Compiling the above code with HP aC++ generates the following error:

```
Future Error: File “DDB04313A.C”, Line 4
The initializer for a non-constant reference must be an lvalue.
 Try changing ‘int &’ to ‘const int &’.
```

To successfully compile the code with either compiler, use one of the two alternatives shown below:

```
void f(const int &);  // Use a constant reference.
int main () {
   f(3);
   return 0;
}
```

Or

```
void f(int &);
int i;
int main () {
   i=3;
   f(i);              // Use an lvalue for reference initialization.
   return 0;
}
```

Digraph White Space Separators

HP C++ does not support alternative tokens (digraphs). In HP aC++, digraphs are supported and legal C++ syntax can be considered an error because of digraph substitution. Insert a blank between two characters of the digraph.

Example:

```
C<::A> a;
```

The characters `<:` are one of the alternative tokens (digraphs) for which HP aC++ performs a substitution. In this case, `<:` becomes `[`. The statement to be compiled becomes `C[:A a;`, which produces many compilation errors.

To successfully compile this program with either compiler, insert a blank between `<` and `:`.

Example:

```
C< ::A> a;
```

Migration Considerations when Using Templates

In HP aC++, templates are processed differently than in HP C++ (cfront). HP aC++ does not have a repository. All instantiations are placed in an object (`.o`) file (with additional information in a `.Ia` file if you specify the `+inst_auto` command-line option). You cannot modify these files as was possible with the files in a repository.

See Chapter 5: "Using HP aC++ Templates" (page 132) for more information.

To begin migrating code containing templates to HP aC++, try to compile and link using the default compile-time instantiation. If this fails with compilation errors, you can compile using one of the following:

- The `+inst_all` option to view all compile-time errors, including template instantiation errors. This may generate errors that will not occur in your program, because the draft standard allows template parameters that cannot instantiate all members. The `+inst_all` option forces instantiation of such members.
- The `+inst_directed` option to mask compile-time template instantiation errors.

To reset after all translation units compile successfully:

1. Remove any `.o` and `.I` files. Using a clobber makefile target to remove `.I` files is similar to removing the `ptrepository` directory in cfront.
2. Recompile and link using compile-time instantiation.

Verbose Template Processing Information

Use the `+inst v` option to replace the cfront `-ptv` option tp process verbose template information.

Common Template Migration Syntax Changes

You must use the keyword `typname` to distinguish types in template code in HP aC++. Also, use the `this->` notation to reference data members.

The cfront Implicit Include Convention

The preferred method for specifying template declarations and definitions in HP aC++ is to put declarations and definitions in the same file.

In HP C++ (cfront), for any `.h` file that contains template declarations, there is a `.c` file that contains definitions for those templates.

HP aC++ provides the following options to ease migration from HP C++ (cfront):

- `+inst implicit_include`: This option instructs the compiler to use the cfront default file, `name lookup`, for template definition files.
- `+inst include_suffixes`: Use this option to replace the cfront `-ptS"list"` option. This specifies file name extensions for template definition files.

Converting Directed Mode to Explicit Instantiation

If you use directed mode instantiation with the cfront based compiler, an `awk` script can be used to convert your file to an instantiation file that uses the explicit instantiation syntax:

Example:

```
#!/usr/bin/ksh
# For a Directed-Mode Instantiation file that is the parameter
# to the script, create a file that can be compiled with the
# aC++ compiler using the Explicit Instantiation Syntax.
# (Note that this will only work for classes.)

closure_file=$1
closure_file_base_name=${1%\.*}
eis_file=$closure_file_base_name.eis.C

print “Output file:  $eis_file”
# Get all of the include directives.
grep “#include” $closure_file > /tmp/dmi2eis1.$$

# Collect all of the Directed-Mode Instantiation directives.
grep -v “#include” $closure_file \
   | grep -e “>” -e “<“ \
   | grep -v “(“ \
   | awk ‘ {if ($1 != “//”) {print $0;} }’ >/tmp/dmi2eis2.$$

# Print the line assuming that the last element is the variable
# name followed immediately by a semi-colon.
awk ‘{ n=split($0,sp0);
   printf(“template class”);
   for (i=1; i<=(n-1); i++) {
     printf(“ %s”, sp0[i]);
   }
   printf(“;\n”);
 }’ < /tmp/dmi2eis2.$$ > /tmp/dmi2eis3.$$

    cat /tmp/dmi2eis1.$$ /tmp/dmi2eis3.$$ > $eis_file
    rm -f /tmp/dmi2eis*.$$
```

NOTE: You can use explicit instantiation to instantiate a template class and all its member functions, an individual template function, or a member function of a template class.

13 Documentation feedback

HP is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (docsfeedback@hp.com). Include the document title and part number, version number, or the URL when submitting your feedback.

A Diagnostic Messages

The aC++ compiler can issue a large variety of diagnostics in response to unexpected situations or suspicious constructs. This appendix presents a summary of these diagnostics organized into the following sections:

- "aC++ Message Catalog" (page 222)
- "Frequently Encountered Messages" (page 222)

aC++ Message Catalog

The aC++ message catalog is located in the following directory:

`/opt/aCC/lib/nls/msg/C/ecc.msgs`

Error and warning messages in the aC++ message catalog can be classified as shown below:

Command Errors

Command errors are issued when the command line is not correctly formed and the compiler cannot proceed with compilation.

Command Warnings

Command warnings occur when code is compiled with an unrecognized option.

Fatal Errors

Fatal errors are issued for ill-formed programs that the compiler cannot recover. Syntax errors usually fall into this category. Object files are not generated if such errors are encountered.

Future Errors

Future errors are serious warnings that indicate violation of a language rule. However, the compiler continues to generate code. You must resolve these errors, as they may result in other cumulative errors. Use the `+p` option (pedantic mode) to convert these warnings into hard errors.

Anachronisms

Anachronisms are warnings that diagnose ISO/ANSI C++ language violation. Code that triggers anachronisms was previously considered legal.

Warnings

Warnings identify bugs in code, often because the code triggers behavior that is not precisely defined by the C++ standard.

Suggestion/Information

You must use the `+w` option to view these diagnostics. Without the `+w` option, the compiler identifies more suspicious constructs.

Tool Errors

Sometimes, HP aC++ fails in a component that is not specific to the C++ language. In such a case, a tool error is emitted. This error indicates defect in the compiler.

Frequently Encountered Messages

For a list and description of frequently encountered diagnostic messages, use the HP Code Advisor analysis tool (cadvise) or refer to the *aC++ standard conformance and compatibility changes* document accessible from the list of "HP aC++ Resources" at the bottom of the following URL:

`http://www.hp.com/go/aCC`

Glossary

A

aggressive optimization	Optimization that changes the behavior of structured code. This is a superset of basic optimizations.
anachronistic constructs	Elements of the C++ language that are not supported in future releases.
archive library	A collection of object files grouped using the `ar` command. At link time, only object files with symbols are extracted from the library.
argument declaration file	A file containing the declaration of a class, struct, union, or enum types for templates.
automatic instantiation	An instantiation mechanism that uses an automatic instantiation algorithm to determine in which object file instantiations are placed. Instantiation is attempted for any use of a template. Use the `+inst_auto` command line option to request automatic instantiation.

B

base class	A class from which another class, the derived class, inherits public and protected members. A derived class inherits the nonprivate member data and nonprivate member functions from its base class. Sometimes also called a parent class or superclass.
basename	The part of a pathname after the last /.
basic block	A sequence of instructions with a single entry point, single exit point, and no internal branches.
basic optimizations	Any optimizations that does not generally change the behavior of structured code. Basic optimization is performed by default when you specify a level of optimization. Basic optimizations are a subset of aggressive optimizations and a superset of conservative optimizations.

C

class	A user-defined type. A class can have member data and member functions and these can be public, protected, or private members.
class template	A template that defines an unbounded set of related classes.
closing	The process of satisfying all template instantiations for a set of link units.
closing a library	Satisfying all template instantiations needed by a library when building the library, not when linking the library with an application.
compile-time instantiation	In HP aC++, this is the default instantiation mechanism. Instantiation is attempted for every template used in a translation unit in that translation unit.
conservative optimizations	Any optimization that does not change the behavior of code, in most cases, even if the code is unstructured or does not conform to standards. This is a subset of basic optimizations.
constructor	An initialization function for the objects of a class. Constructors have the same name as their class.

D

derived class	A class that inherits the public and protected member data and the public and protected member functions from its base class. It is also called a child class or subclass.
destructor	A function that cleans up or deinitializes each object of a class immediately before the object is destroyed. Destructors are executed when the program leaves the scope in which objects are defined and when any object is destroyed by `delete`. Destructors have the same name as their class, prefixed by a tilde, ~.
directed instantiation	Template instantiation that is specified by the developer through an explicit instantiation or a compiler command-line option.

E

exception	An exception is a runtime error condition. Exception handling is a C++ mechanism that allows the error detector to pass the error condition to the exception handler. An exception is raised by a `throw` statement within a `try` block and handled by a `catch` clause. The ANSI/ISO C++ International Standard defines only synchronous exceptions.
explicit instantiation	A method of instantiation that instantiates a template at the point of its use. You can code an explicit template instantiation, as defined in the *Final Draft International Standard*, in your source file.
external symbol	A name of a function or data item in an object file that you can link with other object files.

F

friend	A class or a function that has access to data of a class and member functions. Friend has access to the public, protected, and private members of a class.
function template	A template that defines an unbounded set of related functions.

H

header file	A C++ source file typically containing class or function declarations. It is referenced by other C++ source files using the `#include` preprocessor directive.
HP aC++	The latest C++ compiler from HP. It closely complies with most features of the ANSI/ISO C++ International Standard.
HP C++	An initial, pre-C++ draft proposed international standard C++ compiler from HP. It is based on the cfront compiler and provides functionality for templates and exception handling.

I

include guards	Preprocessor commands, such as, `#ifndef`, `#define`, and `#endif`, used in a header file to prevent compiling that file more than once.
inline function	A function whose code is copied in place of each function call.
instantiate	To form an instantiation by binding a template to particular argument types.
instantiated class	A class generated from a class template by instantiation.
instantiated function	A function generated from a function template by instantiation.
instantiation	A generated class or function (a definition) that is the result of binding a template to particular argument types. Also known as a generation.

L

lex	A program generator for lexical analysis of text.
link unit	A single entity submitted to the linker. A link unit can be an object file (`.o` file, the output of a translation unit), an archive library (`.a` file), or a shared library (`.so` file).
load compile	Invoking the compiler using the `+hdr_use` option, and a manual precompiled header file.

M

member data	Any data element declared to be part of a class.
member function	Any function declared to be part of a class.
multiple inheritance	The ability of a class to inherit from more than one base class. The derived class inherits all public and protected members from all of its base classes. Also see *single inheritence*.

N

name demangling	The process of changing the internal representation of identifiers back to their original C++ source names. Also see *name mangling*.

name mangling	The process of generating unambiguous internal identifiers from C++ identifiers to resolve the scope of variables, overloaded operators, and overloaded functions. Also see *name demangling*.

O

object	An instance of a class

P

parameterized type	See *template*.
position-independent code (PIC)	Object code that contains no absolute addresses. All addresses are relative to the program counter. Position-independent code is used to create shared libraries.
pragma	An instruction to the compiler to compile your program in a certain way. For example, you can use pragmas to insert copyright information into your object files, to specify a particular template instantiation, and to specify optimization levels.
precompiled header file	A `.C` file that is compiled using either the `+hdr_create` option (for subsequent use in a load compile) or the `+hdr_cache` option.
preprocessing directive	A command entered into a source file to direct the preprocessor to perform certain actions on the source file. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress the compilation of part of the file by conditionally removing sections of text. It also expands preprocessor macros and conditionally strips out comments.
preprocessor	A portion of the HP aC++ compiler that manipulates the contents of your source file according to the preprocessing directives coded in the source file.
private member	A private member of a class is a data member or member function that is only accessible from within the class defining the member and from any friends of the class defining the private member.
profile-based optimization	An optimization in which the compiler and linker work together to optimize an application based on profile data obtained from running the application on a typical input data set.
protected member	A protected member of a class is a data member or member function that is only accessible from within the class defining the member, or from any class derived from that class, or from any friends of the class defining the protected member.
public member	A public member of a class is a data member or member function that is accessible from everywhere outside the class defining the member as well as from inside the class and from any derived classes.

S

shared library	A collection of object files grouped using the `aCC` command. It comprises position-independent code. At link time, all object files are made available.
single inheritance	The ability of a derived class to inherit from its base class. Also see *multiple inheritence*.
software pipelining	A code transformation that optimizes program loops. It is useful for loops that contain arithmetic operations on floats and doubles.
source file	An HP-UX file that contains C and/or C++ program code.
specialization	An instantiation of a template class or template function that overrides the standard version.

T

template	A skeleton or description for an infinite set of classes or functions. A class template is a specification for a family or group of classes. A class template is also known as a parameterized type. A function template is a specification for a family or group of functions.
template argument	A type or constant specified to a template to distinguish a particular usage of the template.
template function	An instantiated function template.
timestamp	The date and time a file was last changed.

translation unit	The standard term for a compilation unit. It refers to a single source file submitted to the compiler along with all files included by the compilation of that single source file. A translation unit normally results in a single object file. It is also a variable name explicitly declared static has the scope of its translation unit and can be used as a name for other objects, functions, and so on in other translation units in the same application.
trigraph sequences	A set of three characters that is replaced by a corresponding single character by the preprocessor.

Y

yacc	A programming tool to describe input to a computer program.

Index