HP aC++ Online Programmer's Guide

Send mail to report errors or comment on the documentation.

Welcome to the **HP aC++ Online Programmer's Guide**. The guide is divided into the following files. String search is available within each file. (Choose Edit, then Find.) Note that each file may contain links to related topics in other files.

Rogue Wave Library Reference Manuals

NOTE: If you are accessing this guide from the World Wide Web URL, http://docs.hp.com, rather than from a system on which HP aC++ is installed, Rogue Wave documentation is not available. The above two links and any other such links within this guide will not succeed.

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.

For a complete list of options, select Alphabetical List of Command Line Options. Otherwise, select a category:

- Code Generation
- Data Alignment and Storage
- Debugging
- Error Handling
- Exception Handling
- Extensions to the Language
- Header Files
- Help Online
- Inlining
- Libraries
- Linker
- Naming the Output File
- Native Language Support
- Null Pointer Handling
- Optimizing Your Code
- Precompiled Header Files
- Preprocessor
- Profiling Your Code
- Standards
- Subprocesses of the Compiler
- Templates
- Verbose Compile and Link Information

See Also:

- Concatenating Options
- CXXOPTS Environment Variable

Alphabetical List of Command Line Options

Select the option for which you want more information:

- -.*suffix*
- -Aa
- +A
- -b
- -c
- -D*name*
- +d
- +DA*architecture*
- +df*name*
- +dryrun
- +DS*model*
- -ext
- -E
- +ESfic
- +ESsfc
- -g
- -g0
- -g1
- -G
- +hdr_cache
- +hdr_create
- +hdr_dir
- +hdr_info
- +hdr_use
- +hdr_v
- +help
- +inline_level*num*
- +inst_all
- +inst_auto
- +inst_close
- +inst_directed
- +inst_implicit_include
- +inst_include_suffixes
- +inst_none
- +inst_used
- +inst_v
- -I*directory*
- -I-
- +I
- +k
- -l*name*
- -L*directory*
- +m[d]
- +M[d]
- -n
- -N
- +noeh
- +nostl
- -o *outfile*
- -O
- +O1
- +O2
- +O3
- +O4
- +O[no]all
- +O[no]aggressive
- +O[no]conservative
- +O[no]info
- +O[no]limit
- +O[no]size
- +O*optimization*
- +[no]objdebug
- +p
- -P
- +P
- +pa
- +pal
- +pgm*name*
- -q
- -Q
- -s
- -S
- +time
- -t*x*,*name*
- +tm*target*
- -U*name*
- +u*num*
- -v
- -V
- -w
- +w
- -Wc,-ansi_for_scope,[on][off]
- -Wc,-koenig_lookup,[on][off]
- -W*x*,*args*
- +W *args*
- +We *args*
- -Y
- -z
- +z
- +Z

Options to Control Code Generation

These options allow you to control what kind of code HP aC++ generates.

-c
: Compile to relocatable object file without linking.

+DA*architecture*
: Generate object code for a particular version of the PA-RISC architecture. Also specifie which version of the HP-UX math library to use.
Set the target operating system for the compiler.

+DS*model*
: Perform instruction scheduling tuned for a particular implementation of the PA-RISC architecture.

+k
: Generate code for programs that use a large number of global data items in shared libraries.

-S
: Compile to assembly language without linking.

+tm*target*
: Compile code for optimization with a specific machine architecture.

+z
: Generate position-independent code (PIC) to go into a shared library.

+Z
: Generate position-independent code (PIC) to go into a shared library.

-c Command Line Option Syntax

`-c`

Description:

Compiles one or more source files but does not enter the linking phase.

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 eventually link object files before they can be executed.

Example:

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

Compiles `sub.C` and `prog.C` and puts the relocatable object code in the files `sub.o` and `prog.o`, respectively.

+DA*architecture* Command Line Option Syntax

`+DA`*`architecture`*

architecture can be one of the following:

- a model number of an HP 9000 system (such as 730, 877, F20, or I50)
- a PA-RISC architecture designation (such as `2.0` or `1.1`)
- the term portable (Use `+DAportable` to generate code compatible across 1.1 and 2.0 workstations and servers.)

NOTE: See the `/opt/langtools/lib/sched.models` file for a list of model numbers and their PA-RISC architecture designations.

Description:

Generates object code for a particular version of the PA-RISC architecture. Also specifies which version of the HP-UX math library to link in when you have specified `-lm`. (See the *HP-UX Floating-Point Guide* for more information about using math libraries.)

NOTE: Object code generated for PA-RISC 2.0 will *not* execute on PA-RISC 1.1 systems.

To generate code compatible across PA-RISC 1.1 and 2.0 workstations and servers, use the `+DAportable` option.

For best performance use `+DA` with the model number or architecture where you plan to execute the program.

If you do not specify a +DA option, the default code generation is based on that of the system on which you compile.

If you specify neither a +DA nor a +DS option, default instruction scheduling is based on that of the system on which you compile. If you do specify a +DA option and do not specify a +DS option, default instruction scheduling is based on what you specify in +DA, and not based on that of the system on which you compile.

For example, specify +DA1.1 and do not specify +DS, and instruction scheduling will be for 1.1. Specify +DAportable and do not specify +DS, and instruction scheduling will be for 1.1. (+DAportable is currently equivalent to +DA1.1.)

Examples:

The following examples generate code for various architectures, as noted:

- aCC +DA1.1 prog.C (PA-RISC 1.1)
- aCC +DA867 prog.C (PA-RISC 1.1)
- aCC +DA2.0 prog.C (PA-RISC 2.0)
- aCC +DAportable prog.C (compatible across 1.1 and 2.0 workstations and servers)

For More Information:

- Compiling for Different Versions of the PA-RISC Architecture
- See the file `/opt/langtools/lib/sched.models` for model numbers and their architectures. Use the command `uname -m` to determine the model number of your system.

Compiling for Different Versions of the PA-RISC Architecture

The instruction set on PA-RISC 2.0 is a superset of the instruction set on PA-RISC 1.1. Code generated for HP 9000 PA-RISC 1.1 systems will run on HP 9000 PA-RISC 2.0 systems, though possibly less efficiently than if it were specifically generated for PA-RISC 2.0.

Code generated for PA-RISC 2.0 *will not* run on PA-RISC 1.1 systems.

Using +DA to Generate Code for a Specific Version of PA-RISC

When you use the `+DA` option depends on your particular circumstances.

- If you plan to run your program on the same system where you are compiling, you don't need to use `+DA`.

- If you plan to run your program on one particular model of the HP 9000 and that model is different from the one where you compile your program, use `+DA`*architecture* with the model number of the target system.

  For example, if you are compiling on a 720 and your program will run on an 855, use `+DA855`.

- If you plan to run your program on PA-RISC 2.0 and 1.1 models of the HP 9000, use `+DAportable`.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, it is important to note 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.

+DS*model* Command Line Option Syntax

`+DS`*`model`*

model can be either a model number of an HP 9000 system (such as 725, 890, or G40), PA-RISC architecture designation 1.1 or 2.0, or one of the PA-RISC processor names such as PA7000, PA7100, PA7100LC, or PA8000. See the file `/opt/langtools/lib/sched.models` for model numbers and processor names.

Description:

Performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture.

Object code with scheduling tuned for a particular model *will* execute on other HP 9000 systems, although possibly less efficiently.

If you specify neither a +DA nor a +DS option, default instruction scheduling is based on that of the system on which you compile. If you do specify a +DA option and do not specify a +DS option, default instruction scheduling is based on what you specify in +DA, and not based on that of the system on which you compile.

For example, specify +DA1.1 and do not specify +DS, and instruction scheduling will be for 1.1. Specify +DAportable and do not specify +DS, and instruction scheduling will be for 1.1. (+DAportable is currently equivalent to +DA1.1.)

If you plan to run your program on both PA-RISC 1.1 and 2.0 systems, use the +DS2.0 designation.

Examples:

`+DS720`
: Performs instruction scheduling tuned for one implementation of PA-RISC 1.1.

`+DS745`
: Performs instruction scheduling for another implementation of PA-RISC 1.1.

`+DSPA8000`
: Performs instruction scheduling for systems based on the PA-RISC 8000 processor.

For More Information:

- Using +DS to Specify Instruction Scheduling
- See the file `/opt/langtools/lib/sched.models` for model numbers and their processor names. Use

the command `uname -m` to determine the model number of your system.

Using +DS to Specify Instruction Scheduling

Instruction scheduling is different on different implementations of PA-RISC architectures. You can improve performance on a particular model or processor of the HP 9000 by requesting that the compiler 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.

By default, the compiler performs scheduling tuned for the system on which you are compiling, or, if specified, tuned for the setting of the +DA option. Use the `+DS` option to change this default behavior and to specify instruction scheduling tuned to a particular implementation of PA-RISC. For example, to specify instruction scheduling for the model 867, use `+DS867`. To specify instruction scheduling for the PA-RISC 8000 processor, use `+DSPA8000`. See the file `/opt/langtools/lib/sched.models` for model numbers and processor names.

When you use the `+DS` option depends on your particular circumstances.

- If you plan to run your program on the same system where you are compiling, you don't need to use the `+DS` option. The compiler generates code tuned for your system.
- If you plan to run your program on one particular model of the HP 9000 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.

  For example, if you are compiling on a system with a PA7100 processor and your program will run on a system with a PA7100LC processor, you can use `+DSPA7100LC`. This will give you the best performance on the PA7100LC system.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, it is important to note 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.

+k Command Line Option Syntax

`+k`

Description:

By default, the HP aC++ compiler generates short-displacement code sequences for programs that reference global data in shared libraries. For nearly all programs this is sufficient.

If your program references a large amount of global data in shared libraries, the default code generation for referencing that global data may not be sufficient. If this is the case, when you link your program the linker gives an error message indicating that you need to recompile with the `+k` option. The `+k` option generates long-displacement code sequences so a program can reference large amounts of global data in shared libraries. Use `+k` only when the linker generates a message indicating you need to do so.

Example:

```
aCC +k prog.C mylib.sl
```

Compiles `prog.C`, generates code for accessing a large number of global data items in the shared library `mylib.sl`, and links with `mylib.sl`.

+tm*target* Command Line Option Syntax

`+tm`*`target`*

Description:

+tm *target* specifies the target machine architecture for which compilation is to be performed. Using this option causes the compiler to perform architecture-specific optimizations. *target* takes one of the following values:

- K7200 to specify K-Class servers using PA-7200 processors
- K8000 to specify K-Class servers using PA-8000 processors
- V2200 to specify V2200 servers

Using the `+tm` *target* option implies +DA*architecture* and +DS*model* settings as described in the following table.

specified *target* value	+DA*architecture* implied	+DS*model* implied
K7200	`1.1`	`1.1`
K8000	2.0	2.0
V2200	2.0	2.0

NOTE: If you specify `+DA` or `+DS` on the aCC command line, your setting takes precedence over the setting implied by `+tm` *target* .

Usage:

Use `+tm` *target* at optimization levels 0, 1, 2, 3, and 4. The default *target* value corresponds to the machine on which you invoke the compiler.

-S Command Line Option Syntax

`-S`

Description:

Compiles the named HP aC++ program and leaves the assembly language output in a corresponding file with a `.s` suffix.

CAUTION: The `-S` option is informational only. Generated output is not meant to be used as input to the assembler (as).

Example:

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

Compiles `prog.C` to assembly code rather than to object code, and puts the assembly code in the file `prog.s`.

Data Alignment and Storage

+u
: Allows pointers to access non-natively aligned data.

For More Information:

- Pragma pack

+u Command Line Option Syntax

`+u`*num*

Description

The +u 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. *num* can be specified as:

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
```

Default Data Storage and Alignment

This section describes default data storage allocation and alignment for HP aC++ 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 aC++ 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.

The following lists the sizes and alignments of HP aC++ data types:

```
Data Type          Size (bytes)      Alignment

bool                    1             1-byte

char, unsigned char,    1             1-byte
signed char

wchar_t                 2             2-byte

short, unsigned short,  2             2-byte
signed short
```

```
int, unsigned int      4             4-byte

long, unsigned long    4             4-byte

float                  4             4-byte

double                 8             8-byte

long double           16             8-byte

long long,            64             8-byte
unsigned long long

enum                   4             4-byte

arrays             Size and alignment of array element type.

struct                (*)          1-, 2-, 4- or 8-byte (*)

union                 (*)          1-, 2-, 4- or 8-byte (*)

bit-fields         Size and alignment of declared type.

pointer                4             4-byte
```

(*) Alignment is the same as the strictest alignment of any member. Padding is done to a multiple of the alignment size.

Debugging Options

Debugging options enable you to use the HP WDB Debugger or the HP/DDE Debugger.

+d
: Disable all inlining of functions.

-g
: Generate minimal information for debugging.

-g0
: Generate full information for debugging.

-g1
: Generate minimal information for debugging.

+[no]objdebug
: Generate debug information in object files and not in the executable.

+d Command Line Option Syntax

```
+d
```

Description:

Prevents the expansion of inline functions.

This option is useful when you are debugging your code because you cannot set breakpoints at inline functions. This option defeats inlining thereby allowing you to set breakpoints at functions specified as inline.

See Also:

- +O[no]inline Command Line Option Syntax

-g Command Line Option Syntax

`-g`

Description:

Like the -g1 option, `-g` 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.

For More Information:

- Difference between -g, -g0, and -g1
- When to Use -g, -g0, or -g1
- HP WDB Debugger Documentation
- HP/DDE Debugger Documentation

-g0 Command Line Option Syntax

`-g0`

Description:

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

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

For More Information:

- Difference between -g, -g0 and -g1
- When to Use -g, -g0, or -g1
- HP WDB Debugger Documentation
- HP/DDE Debugger Documentation

-g1 Command Line Option Syntax

`-g1`

Description:

Like the -g option, `-g1` 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.

For More Information:

- Difference between -g, -g0, and -g1

- When to Use -g, -g0, or -g1
- HP WDB Debugger Documentation
- HP/DDE Debugger Documentation

Difference between -g, -g0, and -g1

The -g, -g0, and -g1 options all 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, on the other hand, 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, or -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 megabyte.

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 megabyte.

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.

For More Information:

- Difference between -g, -g0, and -g1

-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.

+[no]objdebug Command Line Option Syntax

```
+[no]objdebug
```

Description:

Generate [do not generate] debug information in object files and not in the executable. The HP WDB debugger then reads the object files to construct debugging information. +objdebug is not compatible with the HP DDE debugger.

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

+noobjdebug is the default at compile time and is the same as versions of the compiler prior to A.01.15. +objdebug is the default at link time.

If +noobjdebug is used at link time (not the default), all debug information goes into the executable, even if some objects were compiled with +objdebug. This can be used to enforce the debugging paradigm prior to HP aC++ version A.01.15.

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

Use +[no]objdebug with the -g, -g0, or -g1 option.

For More Information:

- See documentation for the Linker and for the HP WDB Debugger for more details.

Error Handling Options

Use these options to control how potential errors in your C++ code are detected and handled.

+p
: Disallows all anachronistic constructs.

-w
: Suppresses all compiler warning messages.

+w
: Warns about all questionable constructs.

+W *args*
: Selectively suppress compiler warnings.

+We *args*
: Selectively interpret warnings or future errors as errors.

+p Command Line Option Syntax

`+p`

Description:

Disallows all anachronistic constructs.

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

Example:

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

Compiles `file.C` and gives errors for all anachronistic constructs rather than just giving warnings.

-w Command Line Option Syntax

```
-w
```

Description:

Suppresses all warning messages.

By default, HP aC++ reports all errors and warnings.

Example:

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

Compiles `file.C` and reports errors but does not report any warnings.

+w Command Line Option Syntax

```
+w
```

Description:

Warns about all questionable constructs, as well as constructs that are almost certainly problems.

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.

Example:

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

Compiles `file.C` and warns about both questionable constructs and constructs almost certainly problematic.

+W *args* Command Line Option Syntax

```
+W arg1 [,arg2,..argn ]
```

Description

Selectively suppresses any specified warning messages, where *arg1* through *argn* are valid compiler warning message numbers.

Example:

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

+We *args* Command Line Option Syntax

`+We arg1 [,arg2,..argn ]`

Description

Selectively interpret 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
```

Exception Handling Option

By default, exception handling is in effect. To turn off exception handling, you *must* use the following option.

+noeh Command Line Option Syntax

`+noeh`

Description:

Disables exception handling.

By default, exception handling is on. To turn off exception handling, you must use this option. With exception handling disabled, the keywords throw and try elicit an error.

Note that if your application throws no exceptions, code compiled with and without +noeh can be mixed freely. However, the mixing of code compiled with and without +noeh in an application which throws exceptions is unsupported.

Example:

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

Compiles and links `progex.C`, which does *not* use exception handling.

For More Information:

- Exception Handling Overview

Extensions to the Language

This option supports extensions to the C++ language.

-ext Command Line Option Syntax

`-ext`

Description:

This option enables 64-bit integer data type support, allowing you to declare `long long` and `unsigned long long` data types.

Use this option for 64-bit integer literals and for input and output of 64-bit integers.

If you use the `-ext` option, use it at both compile and link time.

Example:

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

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

```
#include <iostream.h>

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

Header File Options

-I*directory*
: Add *directory* to the directories to be searched for `#include` files.

-I-
: Override the default -I*directory* search-path.

+m[d]
: Output quote enclosed (" ") make(1) dependency files to stdout or to a .d file.

+M[d]
: Output both quote enclosed and angle bracket enclosed (< >) make(1) dependency files to stdout or to a .d file.

For More Information:

- Using Standard HP-UX Libraries and Header Files
- Source File Inclusion (#include)

-I*directory* Command Line Option Syntax

```
-I directory
```

directory is the HP-UX directory where HP aC++ looks for header files.

Description:

During the compile phase, adds *directory* to the directories to be searched for `#include` files during preprocessing. During the link phase, 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.

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

Example:

```
aCC -I /opt/aCC/include/SC file.C
```

This example directs HP aC++ to search in the directory `/opt/aCC/include/SC` for `#include` files.

-I- Command Line Option Syntax

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

[-I*dirs*] indicates an optional list of -I*directory* specifications in which a directory name cannot begin with a hyphen (-) character.

Description:

The -I- option allows you to override the default -I*directory* search-path. This feature is called view-pathing.

Specifying -I- serves a dual purpose, as follows:

- It changes the compiler's search-path for quote enclosed `(" ")` file names in a #include directive to the following order:

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

  The preprocessor does not search the directory of the including file.

- 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 includes are searched for in the directories that both precede and succeed 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 -I*directory* 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 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 includes 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
```

CAUTION: 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, it 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 Syntax

+help Command Line Option Syntax

`+help`

Description:

Invokes the initial menu window of this *HP aC++ Online Programmer's Guide* .

If +help is used on any command line, the compiler displays the online programmer's guide with the default web browser and then processes any other arguments.

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

```
export DISPLAY=YourDisplayAddress       (ksh 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:

```
aCC +help
```

Inlining Options

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

+d
: Disables all inlining of functions.

+inline_level*num*
: Controls how C++ inlining hints influence HP aC++.

See Also:

- Options for Optimizing Your Code for information about optimizer options that affect inlining.

+inline_level*num* Command Line Option Syntax

`+inline_level`*num*

Description

This option controls how C++ inlining hints influence HP aC++. Specify *num* as 0, 1, 2, or 3.

num	Meaning
0	No inlining is done (same effect as the +d option).
1	Only small functions are inlined.
2	Only large functions are not inlined.
3	Inlining hints are respected in all cases, except when the called function is recursive or when it has a variable number of arguments.

The default level depends on +O*opt* as shown in the following table:

opt	*num*
0	1
1	1
2	2
3	2
4	2

NOTE: This option controls functions declared with the inline keyword or within the class declaration and is effective at all optimization levels.

The options +O[no]inline and +Oinlinebudget control the high level optimizer that recognizes other opportunities in the same source file (+O3) or amongst all source files (+O4).

Example:

```
aCC +inline_level3 app.C
```

See Also:

- Options for Optimizing Your Code for information about optimization options that affect inlining.

Library Options

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

+A
: Link with archive libraries.

-b
: Create a shared library.

+k
: Generate code for programs that use a large number of global data items in shared libraries.

-l*name*
: Specify a library for the linker to search.

-L*directory*

Specify a directory for the linker to search for libraries.

+nostl

Indicate header files and libraries (other than those provided with HP aC++) for compilation and linking.

-W*x*,*args*

One use of -W is to specify linking of shared or archive libraries.

+z

Generate position-independent code (PIC) to go into a shared library.

+Z

Generate position-independent code (PIC) to go into a shared library.

See Also:

- Creating and Using Libraries

+A Command Line Option Syntax

`+A`

Description:

Causes the linker to link with archive libraries rather than shared libraries and creates a completely archived executable.

The `-a,archive` linker option also links archive libraries but it links the shared library `/usr/lib/libdld.sl`. `+A` links in `/opt/aCC/lib/cxxshl.o` instead of `/usr/lib/libdld.sl`.

Example:

```
aCC +A file.o -lm
```

Links `file.o` and links in the archived version of the math library, `/lib/libm.a`, rather than the shared version, `/lib/libm.sl`, and *does not* link in `/usr/lib/libdld.sl`.

See Also:

- Linking archive or shared libraries with the `-a` linker option
- *HP-UX Linker and Libraries Online User Guide*

-b Command Line Option Syntax

`-b`

Description:

Creates a shared library rather than an executable file.

The object files must have been created with the +z or +Z option to generate position-independent code (PIC).

Example:

```
aCC -b utils.o -o utils.sl
```

Links `utils.o` (which must have been created using the `+z` option) and creates the shared library `utils.sl`.

For More Information:

For more information on shared libraries, see Creating and Using Shared Libraries, and the *HP-UX Linker and Libraries Online User Guide* .

-l*name* Command Line Option Syntax

`-l`*`name`*

The *name* parameter forms part of the name of the library the linker searchs when looking for routines called by your program.

Description:

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`*`name`*`.sl`
- `/usr/lib/lib`*`name`*`.a`
- `/opt/langtools/lib/lib`*`name`*`.sl`
- `/opt/langtools/lib/lib`*`name`*`.a`

Whether it searches the shared library (`.sl`) or the archive library (`.a`) depends on the value of the `-a` linker option or the `+A` 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) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Example:

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

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

See Also:

- Use the -L*directory* option to specify additional directories for the linker to search for libraries.
- Use of the +DA*architecture* or the -G option affects the set of default libraries.

-L*directory* Command Line Option Syntax

`-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.

Description:

Causes the linker to search for libraries in *directory* in addition to using the default search path.

The default search path is the directory `/opt/aCC/lib`.

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:

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

Compiles and links `prog.C` and directs the linker to search the directories `/opt/aCC/lib` and `/project/libs` for any libraries that `prog.C` uses (in this case, mylib1 and mylib2).

See Also:

- The *-lname* compile line option.
- The CCLIBDIR environment variable.

+z Command Line Option Syntax

```
+z
```

Description:

Causes the compiler to generate position-independent code (PIC), necessary for building a shared library.

Use -b to create a shared library.

`+z` is similar to the `+Z` option. Use `+z` unless the linker generates an error message indicating that you should use +Z.

The -G option is ignored if either `+z` or +Z is used.

Example:

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

Compiles `utils.C` and generates position-independent code in `utils.o`. `utils.o` can be placed into a shared library with the `-b` option.

For More Information:

For more information on shared libraries, see the tutorial Creating and Using Shared Libraries and the *HP-UX Linker and Libraries Online User Guide* .

+nostl Command Line Option Syntax

```
+nostl
```

Description:

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

+nostl suppresses linking of all default -I*directory* and -L*directory* paths and some of the -l*name* libraries (-lstd and -lstream). Use the -v option to see the effect of +nostl.

CAUTION: Complete understanding of the linking process and the behavior of the actual (third party) libraries linked with the application is **essential** for avoiding link or run-time failures.

For More Information:

For more information on shared libraries, see the tutorial Creating and Using Shared Libraries and the *HP-UX Linker and Libraries Online User Guide* .

+Z Command Line Option Syntax

`+Z`

Description:

Causes the compiler to generate position-independent code (PIC), necessary for building shared libraries.

Use -b to create a shared library.

`+Z` is the same as the +z option except that it allows for more imported symbols than does `+z`. Use the `+Z` option only if errors are generated when you use `+z`.

The -G option is ignored if either `+z` or `+Z` is used.

Linker Options

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

-n
: The linker marks the output as sharable.

-N
: The linker marks the output as unsharable.

-q
: The linker marks the output as demand-loadable.

-Q
: The linker marks the output as not demand-loadable.

-s
: The linker strips the symbol table from the executable file it produces.

In addition, you can use the -W*l,args* compiler option to specify *any* linker option on the compiler command line. For more information on linker options, see the ld(1) man page or the *HP-UX Reference Manual* .

CAUTION: 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.

-n Command Line Option Syntax

`-n`

Description:

Causes the program file produced by the linker to be marked as sharable.

For More Information:

For details and system defaults, refer to the description of `ld` in the *HP-UX Reference Manual* or the ld(1) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.) See also the -N option.

-N Command Line Option Syntax

`-N`

Description:

Causes the program file produced by the linker to be marked as unsharable.

Unsharable executable files generated with the `-N` option cannot be executed with `exec`.

For More Information:

For details and system defaults, refer to the `ld` description in the *HP-UX Reference Manual* or the ld(1) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

See Also:

- The -n option

-q Command Line Option Syntax

`-q`

Description:

Causes the output file from the linker to be marked as demand-loadable.

For More Information:

For details and system defaults, see the description of `ld` in the *HP-UX Reference Manual* or the ld(1) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

-Q Command Line Option Syntax

`-Q`

Description:

Causes the program file from the linker to be marked as not demand-loadable.

For More Information:

For details and system defaults, see the description of ld in the *HP-UX Reference Manual* or the ld(1) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

-s Command Line Option Syntax

```
-s
```

Description:

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 More Information:

For more details, refer to the description of `ld` in the *HP-UX Reference Manual* or the ld(1) man page if it is installed on your system. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Options for Naming the Output File

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

-o *outfile*
: Specifies the name of the output file from the compilation.

-.*suffix*
: Specifies a file name suffix to be used for the output file from the compilation.

-o Command Line Option Syntax

```
-o outfile
```

The *outfile* parameter is the name of the file containing the output of the compilation.

Description:

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* Command Line Option Syntax

```
-.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.

Description:

Causes HP aC++ to direct output from the -E option into a file with the corresponding *.suffix* instead of into a corresponding `.c` file.

Example:

```
aCC -E -.Pfile prog.C
```

Preprocesses the C++ code in `prog.C` and puts the resulting code in the file `prog.Pfile`.

Option to Enable Native Language Support

-Y Command Line Option Syntax

`-Y`

Description:

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:

Refer to `hpnls`, `lang`, and `environ` in the *HP-UX Reference Manual* for a description of the NLS model.

Option for Handling Null Pointers

-z Command Line Option Syntax

`-z`

Description:

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
```

Compiles `file.C` and generates code to disallow dereferencing of null pointers.

For More Information:

See signal(2) and signal(5) for more information. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Options for Optimizing Your Code

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

To use optimization, first specify the appropriate basic optimization level (`+O1, +O2, +O3, or +O4`) on the aCC command line followed by one or more finer or more precise options when necessary. For an introduction with examples, refer to Optimizing HP aC++ Programs.

Categories of options are listed below.

- Basic Optimization Levels
- Additional Options for Finer Control
- Advanced +O*optimization* Options
- Profile-Based Optimization Options
- Other Options that Affect Optimization
- Displaying Optimization Information

For More Information:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL
- Introduction to Optimizing HP aC++ Programs

Basic Optimization Level Options

These options allow you specify the basic level of optimization.

-O
: Specify level 2 optimization.

+O1
: Specify level 1 optimization.

+O2
: Specify level 2 optimization.

+O3
: Specify level 3 optimization.

+O4
: Specify level 4 optimization.

-O Command Line Option Syntax

`-O`

Description:

Invokes the optimizer to perform level 2 optimization.

Example:

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

Compiles `prog.C` and optimizes at level 2.

For More Information:

You can set other optimization levels by using the following options:

- +O1
- +O2
- +O3
- +O4

See Also:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL

+O1 Command Line Option Syntax

```
+O1
```

Description:

Performs level 1 optimization only. This includes branch optimization, dead code elimination, faster register allocation, instruction scheduling, and peephole optimization.

Example:

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

Compiles `prog.C` and optimizes at level 1.

For More Information:

You can set other optimization levels by using the following options:

- +O2
- +O3
- +O4

See Also:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL

+O2 Command Line Option Syntax

```
+O2
```

Description:

Performs level 2 optimization. This includes level 1 optimizations plus optimizations performed over entire functions in a single file.

Example:

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

Compiles `prog.C` and optimizes at level 2.

For More Information:

You can set other optimization levels by using the following options:

- +O1
- +O3
- +O4

See Also:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL

+O3 Command Line Option Syntax

```
+O3
```

Description:

Performs level 3 optimization. This includes level 2 optimizations plus full optimization across all subprograms within a single file.

Example:

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

Compiles `prog.C` and optimizes at level 3.

For More Information:

You can set other optimization levels by using the following options:

- +O1
- +O2
- +O4

See Also:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL

+O4 Command Line Option Syntax

```
+O4
```

Description:

Performs level 4 optimization. This includes level 3 optimizations plus full optimizations across the entire application program.

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.

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

1. Compile at `+O4` only those modules that need to be compiled at optimization level 4, and compile the remaining modules at a lower level.

2. If you still run out of memory, increase the per-process data size limit. Run the System Administrator Manager (SAM) to increase the `maxdsiz` process parameter from 64 MB to 128 MB. This procedure provides the process with additional data space.

   Refer to the *System Administration Tasks* manual, Chapter 11, "Reconfiguring the HP-UX Kernel." See Appendix A for full descriptions of the different process parameters, including `maxdsiz`.

3. 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, Chapter 6, "Managing Swap Space." Pay particular attention to the section "Adding File System Swap", because adding file system swap is easier than increasing the amount of device swap, which requires re-configuring your disk. However, if you find that you are consistently compiling beyond the available amount of device swap, you may not have a choice.

For a complete discussion of swap space, refer to *How HP-UX Works: Concepts for the System Administrator* .

Example:

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

Compiles `prog.C` and optimizes at level 4.

For More Information:

You can set other optimization levels by using the following options:

- +O1
- +O2
- +O3

See Also:

- Pragma OPTIMIZE
- Pragma OPT_LEVEL

Additional Optimizations for Finer Control

+ESfic
: Replace millicode calls with inline fast indirect calls.

+ESsfc
: Replace millicode calls with inline code when performing simple function pointer comparisons.

+O[no]all
: Perform maximum optimization.

+O[no]aggressive
: Optimizations that may change the behavior of code.

+O[no]conservative
: Optimize with the minimum risk of side effects.

+O[no]limit
: Optimize using [un]restricted compile time.

+O[no]size
: Enable [disable] code expanding optimizations.

+ESfic Command Line Option Syntax

Syntax

```
+ESfic
```

Description:

Replaces millicode calls with inline fast indirect calls. The `+ESfic` compiler option affects how function pointers are dereferenced in generated code. The default is to generate low-level millicode calls for function pointer calls.

The `+ESfic` option generates code that calls function pointers directly, by branching through them.

NOTE: The `+ESfic` option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries only. Using this option can improve run-time performance.

+ESsfc Command Line Option Syntax

Syntax

```
+ESsfc
```

Description:

Replaces millicode calls with inline code when performing simple function pointer comparisons. The `+ESsfc` compiler option affects how function pointers are compared in generated code. The default is to generate low-level millicode calls for function pointer comparisons.

The `+ESsfc` option generates code that compares function pointers directly, as if they were simple integers.

NOTE: The `+ESsfc` option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries only. Using this option can improve run-time performance.

Example:

Following is an example of a code fragment that performs function pointer comparisons:

```
int (*g)( );
int (*f)( );
```

```
int foo ( );
{
    . . .
}
    . . .
  if (f == g)
    . . .
  if (f == foo)
    . . .
  if (f == SIG_ERR) /* SIG_ERR is defined in signal.h */
    . . .
```

+O[no]all Command Line Option Syntax

`+O[no]all`

Description:

Use `+Oall` to obtain the best possible performance.

This option should be used with stable, well-structured code. These optimizations give you the fastest code, but are riskier than the default optimizations.

You can use +Oall at optimization levels 2, 3, and 4. The default is `+Onoall`.

Examples:

`aCC +Oall prog.C`

Compiles `prog.C` and optimizes for best performance.

`aCC -O +Oall prog.C`

Compiles `prog.C` and optimizes at level 2 with aggressive optimizations and unrestricted compile time.

For More Information:

The `+Oall` option without +O2, +O3, or +O4 combines the following options:

- +O4
- +Oaggressive
- +Onolimit

+O[no]aggressive Command Line Option Syntax

`+O[no]aggressive`

Description:

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

Aggressive optimizations can result in significant performance improvement, but can change program behavior. They can:

- convert certain library calls to **millicode** and inline instructions
- alter error handling and asynchronous interrupt handling as a result of instruction scheduling optimization
- cause less precise floating-point results
- cause programs that perform comparisons between pointers to shared memory and pointers to private memory to run incorrectly

Use `+Oaggressive` with optimization levels 2, 3, or 4. By default, aggressive optimizations are turned off.

Example:

To enable aggressive optimizations at the second, third, or fourth optimization levels, type:

```
aCC +O2 +Oaggressive sourcefile.C
```

or:

```
aCC +O3 +Oaggressive sourcefile.C
```

or:

```
aCC +O4 +Oaggressive sourcefile.C
```

For More Information:

The `+Oaggressive` option invokes the following advanced optimization options:

- +Oentrysched
- +Onofltacc
- +Onoinitcheck
- +Olibcalls
- +Oregionsched
- +Osignedpointers

+O[no]conservative Command Line Option Syntax

```
+O[no]conservative
```

Description:

The `+Oconservative` option causes the optimizer to make conservative assumptions about application code and enable *only* conservative optimizations, a subset of basic optimizations.

Use `+Oconservative` at optimization levels 2, 3, or 4 when your source code is unstructured or you are unfamiliar with the source code being optimized. The default is `+Onoconservative`.

Example:

To enable conservative optimizations at the second, third, or fourth optimization levels, use the `+Oconservative` option as follows:

```
aCC +O2 +Oconservative sourcefile.C
```

or:

```
aCC +O3 +Oconservative sourcefile.C
```

or:

```
aCC +O4 +Oconservative sourcefile.C
```

+O[no]limit Command Line Option Syntax

```
+O[no]limit
```

Description:

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

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

Use `+Onolimit` with optimization levels 2, 3, or 4. The default is `+Olimit`.

Example:

To remove optimization time restrictions at the second, third, or fourth optimization levels, use `+Onolimit` as follows:

```
aCC +O2 +Onolimit sourcefile.C
```

or:

```
aCC +O3 +Onolimit sourcefile.C
```

or:

```
aCC +O4 +Onolimit sourcefile.C
```

+O[no]size Command Line Option Syntax

```
+O[no]size
```

Description:

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 optimization levels 2, 3, or 4. The default is `+Onosize`.

Example:

To disable code size expanding optimizations at the second, third, and fourth optimization levels, use `+Osize` as follows:

```
aCC +O2 +Osize sourcefile.C
```

or:

```
aCC +O3 +Osize sourcefile.C
```

or:

```
aCC +O4 +Osize sourcefile.C
```

Advanced +O*optimization* Options

Advanced optimization options provide additional control for special situations.

+O[no]dataprefetch
: Enable [disable] optimizations to generate data prefetch instructions for data structures referenced within innermost loops.

+O[no]entrysched
: Perform [do not perform] instruction scheduling on a subprogram's entry and exit sequences.

+O[no]failsafe
: Enable [disable] fail-safe optimization.

+O[no]fastaccess
: Enable [disable] fast access to global data items.

+O[no]fltacc
: Disable [enable] all optimizations that cause imprecise floating-point results.

+O[no]initcheck
: Enable [disable] initialization of uninitialized scalar variables to null values.

+O[no]inline
: Inline [do not inline] procedure calls.

+Oinlinebudget
: Inline aggressively.

+O*level* =*name1* [,*name2* ,...*nameN*]
: Lower the optimization level for one or more named functions.

+O[no]libcalls
: Use [do not use] millicode routines instead of certain math library calls.

+O[no]looptransform
: Transform [do not transform] eligible loops for improved cache performance.

+O[no]loopunroll[=unroll factor]
: Enable [disable] loop unrolling.

+O[no]moveflops
: Move [do not move] conditional floating-point instructions out of loops.

+O[no]parmsoverlap
: Assume [do not assume] that arguments of function calls overlap in memory.

+O[no]pipeline
: Enable [disable] software pipelining.

+O[no]procelim
: Enable [disable] elimination of unreferenced procedures.

+O[no]promote_indirect_calls
: Enable [disable] the promotion of indirect calls to direct calls.

+O[no]regionsched
: Move [do not move] instructions across branches.

+O[no]regreassoc
: Enable [disable] register reassociation.

+Oreusedir=*DirectoryPath*
: Specify a directory in which to save intermediate object code files for reuse.

+O[no]signedpointers
: Optimize treating pointers as signed [unsigned] quantities.

+O[no]volatile
: Assume all global variables are [not] volatile.

+O[no]dataprefetch Command Line Option Syntax

```
+O[no]dataprefetch
```

Description:

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). It is only available for PA-RISC 2.0 targets.

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

You can use `+Odataprefetch` at optimization levels 2, 3, and 4. The default is `+Onodataprefetch`.

+O[no]entrysched Command Line Option Syntax

```
+O[no]entrysched
```

Description:

The `+Oentrysched` option optimizes instruction scheduling on a procedure's entry and exit sequences. Enabling this option can speed up an application. The option has undefined behavior for applications which handle asynchronous interrupts. The option affects unwinding in the entry and exit regions.

At optimization level +O2 and higher (using dataflow information), save and restore operations become more efficient.

This option can change the behavior of programs that perform error handling or that handle asynchronous interrupts. The behavior of `setjmp()` and `longjmp()` is not affected.

Use `+Oentrysched` at optimization levels 2, 3, or 4. The default is `+Onoentrysched`.

+O[no]failsafe Command Line Option Syntax

```
+O[no]failsafe
```

Description:

The `+Ofailsafe` option allows compilations with internal optimization errors to continue by issuing a warning message and restarting the compilation at `+O0`.

You can use `+Onofailsafe` at optimization levels 1, 2, 3, or 4 when you want the internal optimization errors to abort your build.

The default is `+Ofailsafe` at levels 1, 2, 3, 4.

+O[no]fastaccess Command Line Option Syntax

```
+O[no]fastaccess
```

Description:

The `+Ofastaccess` option optimizes for fast access to global data items.

Use `+Ofastaccess` to improve execution speed at the expense of longer compile times.

Use `+Ofastaccess` at optimization levels 0, 1, 2, 3, or 4. The default is `+Onofastaccess` at optimization levels 0, 1, 2, and 3, and `+Ofastaccess` at optimization level 4.

+O[no]fltacc Command Line Option Syntax

```
+O[no]fltacc
```

Description:

The `+Ofltacc` option allows the compiler to make transformations which are algebraically correct, but which may affect the result of computations slightly due to the inherent imperfection of computer floating-point arithmetic. These transformations typically exceed those allowed by relevant language standards.

For many programs, the results obtained under `+Onofltacc` are as valid as those obtained without the optimization. They may be slightly different, but not obviously better or worse. For applications in which roundoff error has been carefully studied, and the order of computation carefully crafted to control error, `+Onofltacc` may be unsatisfactory.

Use `+Onofltacc` at optimization levels 2, 3, or 4. The default is `+Ofltacc`.

CAUTION: The sophistication of this optimization is likely to change over time. You should review the performance of your code with subsequent releases of the HP aC++ compiler.

Example:

`+Onofltacc` allows the compiler to substitute 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.

+O[no]initcheck Command Line Option Syntax

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

Description:

The initialization checking feature of the optimizer has three possible states: on, off, or unspecified.

- When on (`+Oinitcheck`), the optimizer initializes to zero any local, scalar, non-static variables that are uninitialized with respect to at least one path leading to a use of the variable.
- When off (`+Onoinitcheck`), the optimizer does not initialize uninitialized variables, but issues warning messages when it discovers them.
- When unspecified, the optimizer initializes to zero any local, scalar, non-static variables that are definitely uninitialized with respect to all paths leading to a use of the variable.

Use `+Oinitcheck` to look for program variables that may not be initialized.

Use `+Oinitcheck` at any optimization level and `+Onoinitcheck` at optimization levels 2, 3, or 4.

+O[no]inline Command Line Option Syntax

`+O[no]inline`

Description:

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 `+Onoinline` at optimization levels 3 or 4. The default is `+Onoinline` at optimization levels 1 and 2 and `+Oinline` at levels 3 and 4.

See Also:

- +d Command Line Option Syntax

+Oinlinebudget Command Line Option Syntax

`+Oinlinebudget=`*n*

Description:

The `+Oinlinebudget` option controls the aggressiveness of inlining according to the value you specify for *n* where *n* is an integer in the range 1 - 1000000 that specifies the level of aggressiveness, as follows:

= 100
: Default level of inlining.

> 100
: More aggressive inlining. The optimizer is less restricted by compilation time and code size when searching for eligible routines to inline.

2 - 99
: Less aggressive inlining. The optimizer gives more weight to compilation time and code size when determining whether to inline.

= 1
: Only inline if it reduces code size.

The `+Onolimit` and `+Osize` options also affect inlining. Specifying the `+Onolimit` option has the same effect as specifying `+Oinlinebudget=200`. The `+Osize` option has the same effect as `+Oinlinebudget=1`.

Note, however, that the `+Oinlinebudget` option takes precedence over both of these options. This means that you can override the effect of `+Onolimit` or `+Osize` option on inlining by specifying the `+Oinlinebudget` option on the same compile line.

Use this option at optimization level 3 or higher. The default is `+Oinlinebudget=100`.

For More Information:

See also the `+O[no]inline` option.

+O*level* =*name1* [,*name2* ,...,*nameN*] Command Line Option Syntax

`+O`*`level`* `=`*`name1`* `[,`*`name2`* `,...,`*`nameN`* `]`

Description:

This 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 +O*level* or -O option.

This option works as does 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 +O*level* 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 +O*level* 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]libcalls Command Line Option Syntax

`+O[no]libcalls`

Description:

A number of math library functions are implemented in a special **millicode library** as well as in the standard math library. The millicode versions are up to 25 percent faster than the standard versions. However, they do not set `errno` and do not give an error message in the event of an exception.

The `+Olibcalls` option provides access to millicode routines for the following math library calls:

```
sin   cos   tan   atan2   pow   log10   asin   acos   atan   exp   log
```

Use `+Olibcalls` to improve the performance of these library routines only when you do not want standard error checking. For example, you might use +Olibcalls with code that has already been debugged and runs without error.

The default is `+Onolibcalls`.

Use `+O[no]libcalls` at any optimization level.

+O[no]looptransform Command Line Option Syntax

```
+O[no]looptransform
```

Description:

The `+O[no]looptransform` option enables [disables] transformation of eligible loops for improved cache performance. The most important transformation is the reordering of nested loops to make the inner loop unit stride, resulting in fewer cache misses.

The default is `+Olooptransform` at optimization levels 3 and 4. You cannot use the option at levels 0-2.

+O[no]loopunroll Command Line Option Syntax

```
+O[no]loopunroll[=unroll factor]
```

Description:

The `+Oloopunroll` option turns on loop unrolling. When you use `+Oloopunroll`, you can also use the unroll factor to control the code expansion. The default unroll factor is `4`, that is, four copies of the loop body. By experimenting with different factors, you may improve the performance of your program.

You can use `+Oloopunroll` at optimization levels 2, 3, and 4. The default is `+Oloopunroll`.

+O[no]moveflops Command Line Option Syntax

```
+O[no]moveflops
```

Description:

The `+Omoveflops` option allows moving conditional floating point instructions out of loops. The behavior of floating-point error handling may be altered by this option.

This option also allows you to enable or disable replacing integer divide by a floating point multiply. This may cause SIGFPE if IEEE inexact is enabled (+FPI).

Usage:

Use `+Onomoveflops` if floating-point traps are enabled and you do not want the behavior of floating-point errors to be altered by the relocation of floating-point instructions.

Use `+Onomoveflops` at optimization levels 2, 3, and 4. The default is `+Omoveflops`.

+O[no]parmsoverlap Command Line Option Syntax

`+O[no]parmsoverlap`

Description:

The `+Oparmsoverlap` option optimizes with the assumption that the actual arguments of function calls overlap in memory.

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

Use `+Onoparmsoverlap` at optimization levels 2, 3, and 4. The default is `+Oparmsoverlap`.

+O[no]pipeline Command Line Option Syntax

`+O[no]pipeline`

Description:

The `+Opipeline` option enables software pipelining.

Use `+Onopipeline` to conserve code space.

Use `+Onopipeline` at optimization levels 2, 3, and 4. The default is `+Onopipeline` at optimization level 1 and `+Opipeline` at levels 2, 3, and 4.

+O[no]procelim Command Line Option Syntax

`+O[no]procelim`

Description:

Enable [or disable] the elimination of dead procedure code.

Used when linking an executable file, `+Oprocelim` removes functions not referenced by the application. Used when building a shared library, `+Oprocelim` removes functions not exported and not referenced from within the shared library. This may be especially useful when functions have been inlined.

Note that any function having symbolic debug information associated with it is not removed.

The default is `+Onoprocelim` at optimization levels 1 through 3 and `+Oprocelim` at level 4.

Use `+O[no]procelim` at any optimization level. +O[no]promote_indirect_calls

+O[no]promote_indirect_calls Command Line Option Syntax

`+O[no]promote_indirect_calls`

Description:

This 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 +Onopromote_indirect_calls.

+O[no]regionsched Command Line Option Syntax

`+O[no]regionsched`

Description:

The `+Oregionsched` option applies aggressive scheduling techniques to move instructions across branches.

NOTE: This option is incompatible with the -z option. Using this option with `-z` may cause a `SIGSEGV` error at run-time. See signal(2) and signal(5) for more information. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Use `+Oregionsched` to improve application run-time speed.

Use `+Oregionsched` at optimization levels 2, 3, and 4. The default is `+Onoregionsched`.

+O[no]regreassoc Command Line Option Syntax

`+O[no]regreassoc`

Description:

The `+Onoregreassoc` option turns off register reassociation.

Use `+Onoregreassoc` to disable register reassociation if this optimization hinders application performance.

Use `+Onoregreassoc` at optimization levels 2, 3, and 4. The default is `+Oregreassoc`.

+Oreusedir=*DirectoryPath* Command Line Option Syntax

`+Oreusedir=`*`DirectoryPath`*

Description:

This option specifies a directory for the linker to save object files created from intermediate object files when using +O4 or profile-based optimization. It reduces link time by not recompiling intermediate object files when they don't need to be recompiled.

When you compile with +I, +P, or +O4, the compiler generates intermediate code in the object file. Otherwise, the compiler generates regular object code in the object file. When you link, the linker first compiles the intermediate object code to regular object code, then links the object code. With this option you can reduce link time on subsequent links by not recompiling intermediate object files that have already been compiled to regular object code and have not changed.

NOTE: When you do change a source file or command line options and recompile, a new intermediate object file will be created and compiled to regular object code in the specified directory. The previous object file in the directory will not be removed. You should periodically remove this directory or the old object files since the old object files cannot be reused and will not be automatically removed.

Use `+Oreusedir=DirectoryPath` at optimization level 4 or with profile-based optimization. The default is to use TMPDIR and remove the temporary objects after each link.

+O[no]signedpointers Command Line Option Syntax

```
+O[no]signedpointers
```

Description:

The `+Osignedpointers` option performs optimizations related to treating 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.

Use`+Osignedpointers` to improve application run-time speed.

Use `+Osignedpointers` at optimization levels 2, 3, and 4. The default is `+Onosignedpointers`.

+O[no]volatile Command Line Option Syntax

```
+O[no]volatile
```

Description:

The `+Ovolatile` option implies that memory references to global variables are volatile and cannot be removed during optimization. The `+Onovolatile` option implies that *all globals are not* of `volatile` class. This means that references to global variables can be removed during optimization.

Use this option to control the "volatile" semantics for all global variables.

Use `+Ovolatile` at optimization levels 1, 2, 3, or 4. The default is `+Onovolatile`.

Profile-Based Optimization Options

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

+df*name*
: Specifies the profile database to use with profile-based optimization.

+I
: Prepares the object code for profile-based optimization data collection.

+P
: Performs profile-based optimization.

+pgm*name*
: Specifies the execution profile data to use with profile-based optimization.

For More Information:

For more information on performing profile-based optimization, see:

- Tutorial on Profile-Based Optimization
- *HP-UX Linker and Libraries Online User Guide*

+df*name* Command Line Option Syntax

`+df`*`name`*

Description:

Specifies the path name of the profile database to use with profile-based optimization. This option can be used with the +P command line option.

The profile database by default is named `flow.data`. This file stores profile information for one or more executables. Use `+df` when the `flow.data` file has been renamed or is in a different directory than where you are linking.

You can also use the `FLOW_DATA` environment variable to specify a different path and file name for the profile database file. The `+df`*name* command line option takes precedence over the `FLOW_DATA` environment variable.

NOTE: The `+df`*name* option cannot be used to redirect the instrumentation output (with the `+I` option). It is only compatible with the `+P` option.

Example:

```
aCC +P +dfpbo.data prog.o -o myapp
```

Relinks the object file `prog.o`, optimizes using the run-time profile data in the file `pbo.data`, and puts the executable code in the file `myapp`.

name Parameter

This parameter specifies the path name of the profile database to use with profile-based optimization.

+I Command Line Option Syntax

`+I`

Description:

Instructs the compiler to instrument the object code for collecting run-time profile data. The profiling

information can then be used by the linker to perform profile-based optimization. Code generation and optimization phases are delayed until link time by this option.

After compiling and linking with `+I`, run the resultant program using representative input data to collect execution profile data. Finally, relink with the `+P` option to perform profile-based optimization.

Profile data is stored in `flow.data` by default. See the `+df`*name* option for information on controlling the name and location of this data file.

The +I option is incompatible with exception handling. To turn off exception handling, use the `+noeh` option.

The +I option is incompatible with the `-g0`, `-g1`, `-G`, `+P`, and `-S` options.

Example:

```
aCC +I -O -c prog.C
aCC +I -O -o prog.pbo prog.o
```

Compiles `prog.C` with optimization, prepares the object code for data collection, and creates the executable file `prog.pbo`. Running `prog.pbo` collects run-time information in the file `flow.data` in preparation for optimization with `+P`.

+P Command Line Option Syntax

`+P`

Description:

Directs the compiler to use profile information to guide code generation and profile-based optimization. The compiler generates intermediate compiler code instead of compiled object code. Code generation is done at link time.

The `+P` option does not affect the default optimization level, or the optimization level specified by the `+O1`, `+O2`, `+O3`, or `+O4` options.

NOTE: Source files that are compiled with the `+I` option do not need to be recompiled with `+P` in order to use profile-based optimization. You only need to relink the object files with the `+P` option after running the instrumented version of the program.

The +P option is incompatible with exception handling. To turn off exception handling, use the `+noeh` option.

The +P option is incompatible with the `+noeh`, `-g0`, `-g1`, `+I`, and `-S` options.

Example:

```
aCC +P -o myapp prog.o
```

Relinks the object file `prog.o` and optimizes using the run-time profile data.

+pgm*name* Command Line Option Syntax

`+pgm`*name*

Description:

Specifies a program name to look up in the `flow.data` file to use with profile-based optimization and the +P option.

The `+pgm`*name* option should be used when the name of the profiled executable differs from the name of the current executable specified by the -o option.

Example:

In the following example, the instrumented program file name is `sample.inst`. The optimized program file name is `sample.opt`. The `+pgm`*name* option is used to pass the instrumented program name, `sample.inst`, to the optimizer:

```
aCC -c +I sample.C
aCC -o sample.inst +I -O sample.o
sample.inst < input.file1
aCC -o sample.opt +P +pgm sample.inst sample.o
```

`+pgm` *name* Parameter

The *name* parameter is the instrumented executable program name that is used when performing profile-based optimization.

Other Options that Affect Optimization

+DA
: Generate object code for a particular version of the PA-RISC architecture. Also specifies which version of the HP-UX math library to use.

+DS
: Perform instruction scheduling tuned for a particular implementation of the PA-RISC architecture.

+O[no]info Command Line Option Syntax

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

Description:

`+Oinfo` 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.

Precompiled Header Files

You can reduce compilation time by precompiling common include (header) files. HP aC++ provides two mechanisms, header caching and manual precompiled headers.

Note that the mechanisms cannot be mixed.

Header Caching Options

+hdr_cache
: Request header caching.

+hdr_dir *DirectoryPath*
: Specify a location and name for the header caching directory.

+hdr_info
: Request information about header cache file creation or use.

Manual Precompiled Header Options

+hdr_create
: Create a manual precompiled header file.

+hdr_use
: Compile using a manual precompiled header file.

+hdr_v
: Lists verbose information when manually precompiling a header or when compiling a manual precompiled header file.

See Also:

- Creating and Using Precompiled Header Files

+hdr_cache Option Syntax

```
+hdr_cache
```

Description:

Turns header caching on.

An aCC_cache subdirectory is created to contain precompiled header files. By default, it is in the source file directory. To specify a different aCC_cache location and/or name, use the +hdr_dir option.

Usage

When used together, the manual precompiled header options (+hdr_create and +hdr_use) override the header caching option (+hdr_cache).

+hdr_cache can only be used when actually compiling a source file. If used with -P or -E, it is turned off.

Example:

```
aCC -c +hdr_cache header.C main.C
                                  // Precompile and cache header.C and main.C
                                  // (initial compile).

aCC -c +hdr_cache header.C main.C
                                  // header.C and main.C are not precompiled
                                  // (if no changes to compilation environment).
```

Note that the -c option is needed to suppress the link step.

See Also:

- Creating and Using Precompiled Header Files
- #pragma hdr_stop
- +hdr_info

+hdr_dir *DirectoryPath* Option Syntax

```
+hdr_dir DirectoryPath
```

Description:

When you use the +hdr_cache option, an aCC_cache subdirectory is automatically created to contain precompiled files. By default, it is in the source file directory. The +hdr_dir option allows you to specify a different directory path and directory name in place of the aCC_cache default.

NOTE: To maximize the efficiency of the cache mechanism, it is recommended that you specify a directory in the compilation directory or in a subdirectory of the compilation directory.

Usage

You might use +hdr_dir to specify different aCC_cache locations for debug builds versus release builds, for instance.

Example:

```
aCC -c +hdr_cache header.C    // Precompile and cache header.C
                                   // in the aCC_cache directory,
                                   // a subdirectory of that
                                   // in which header.C is located.

aCC -c -Ddebugflag +hdr_cache +hdr_dir ./debug/aCC_debug_cache header.C
                                   // Precompile and cache header.C
                                   // in the ./debug/aCC_debug_cache directory,
                                   // a subdirectory of that
                                   // in which header.C is located.

aCC -c +hdr_cache +hdr_dir ./release/aCC_release_cache header.C
                                   // Precompile and cache header.C
                                   // in the ./release/aCC_release_cache
                                   // directory, a subdirectory of that
                                   // in which header.C is located.
```

See Also:

- Creating and Using Precompiled Header Files
- #pragma hdr_stop

+hdr_info Option Syntax

```
+hdr_info
```

Description:

Generates a message stating whether a header is being re-used or precompiled. The default is off.

+hdr_create Command Line Option Syntax

```
+hdr_create
```

Description:

Creates a manual precompiled header file for subsequent use when compiling an application or a library with the +hdr_use option.

You can reduce compilation time by precompiling common include (header) files into a precompiled header file.

Example:

```
aCC headers.C -c +hdr_create precomp
```

From headers.C, creates a precompiled header file named precomp.

Note that the -c option is needed to suppress the link step.

See Also:

- Creating and Using Precompiled Header Files

+hdr_use Command Line Option Syntax

```
+hdr_use
```

Description:

Compiles a manual precompiled header file and its corresponding object (.o) file. These files must have been created by using the +hdr_create option.

This is known as a load compile.

Example:

```
aCC main.C +hdr_use precomp
```

Compiles main.C, including a precompiled header file named precomp.

See Also:

- Creating and Using Precompiled Header Files

+hdr_v Command Line Option Syntax

```
+hdr_v
```

Description:

Provides verbose information when precompiling a header or when compiling a precompiled header file.

Examples:

```
aCC headers.C -c +hdr_create precomp +hdr_v
```

Creates a precompiled header file named precomp and displays what is going into the precompiled header file.

```
aCC main.C +hdr_use precomp +hdr_v
```

Compiles main.C and displays what is being used from the precompiled header file.

See Also:

- Creating and Using Precompiled Header Files

Preprocessor Options

The following options are accepted by the preprocessor:

-D*name*
: Defines *name* to the preprocessor.

-E
: Runs only the preprocessor and sends output to `stdout`.

+m[d]
: Output quote enclosed (" ") make(1) dependency files to stdout or to a .d file.

+M[d]
: Output both quote enclosed and angle bracket enclosed (< >) make(1) dependency files to stdout or to a .d file.

-P
: Runs only the preprocessor and sends output to a corresponding `.i` file.

-.*suffix*
: Sends preprocessed output to the corresponding output file ending with `.suffix`.

-U*name*
: Undefines *name* in the preprocessor.

See Also:

- Preprocessing in HP aC++

-D*name* Command Line Option Syntax

```
-Dname [=def ]
```

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

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

Description:

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".

Example:

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

Defines the preprocessor symbol `DEBUGFLAG` and gives it the value `1`. Following is a program that uses this symbol.

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

For More Information:

- Select Preprocessing in HP aC++

-E Command Line Option Syntax

```
-E
```

Description:

Runs only the preprocessor on the named C++ files and sends the result to standard output (`stdout`).

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.

+m[d] Command Line Option Syntax

```
+m[d]
```

Description:

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(1) command.

If +md 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:

CAUTION: Use +md when you also specify the -E or -P option, otherwise the two outputs will be intermixed.

Examples:

command line specified	.d file name	.d file location
aCC -c +m a.C	none	output to stdout
aCC -c -E -.i +md a.C	a.d	current directory
aCC -c -P +md a.C -o b.o	b.d	current directory
aCC -c -P +md a.C -o /tmp/c	c.d	/tmp directory

+M[d] Command Line Option Syntax

```
+M[d]
```

Description:

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(1) command.

If +Md 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:

CAUTION: Use +Md when you also specify the -E or -P option, otherwise the two outputs will be intermixed.

Examples:

command line specified	.d file name	.d file location
aCC -c +M a.C	none	output to stdout
aCC -c -E -.i +Md a.C	a.d	current directory
aCC -c -P +Md a.C -o b.o	b.d	current directory
aCC -c -P +Md a.C -o /tmp/c	c.d	/tmp directory

-P Command Line Option Syntax

```
-P
```

Description:

Only preprocesses the files named on the command line without invoking further phases. Leaves the result in corresponding files with the suffix `.i`.

Example:

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

Preprocesses the file `prog.C` leaving the output in the file `prog.i`. Does *not* compile the program.

For More Information:

- Select Preprocessing in HP aC++

-U*name* Command Line Option Syntax

`-Uname`

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

Description:

Undefines any *name* that has initially been defined by the preprocessing stage of compilation.

A *name* can be a definition set by HP aC++; these are displayed when you specify the -v option. Or a *name* can be a definition you have specified with the -D option on the aCC command line.

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

For More Information:

- Select Preprocessing in HP aC++

Profiling Options

HP aC++ provides the following options for profiling your code.

-G
: Prepare an object file for use with `gprof`.

+pa
: Request that an application be compiled for routine-level profiling with `CXperf`.

+pal
: Request that an application be compiled for routine-level and loop-level profiling with `CXperf`.

See Also:

- Profile-Based Optimization Options

-G Command Line Option Syntax

`-G`

Description:

Prepares an object file for use with `gprof` (to get an execution profile).

Example:

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

Compiles `file.C` and creates the executable file `a.out` instrumented for use with `gprof`.

For More Information:

Refer to the gprof(1) man page. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

+pa Command Line Option Syntax

`+pa`

Description:

Prepares an application for routine level profiling with CXperf.

The +pa option is invalid with the +O4 or +O[no]all optimization options. Also, +pal is incompatible with the +A, -G, and -s options.

For More Information:

Refer to the CXperf(1) man page. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

+pal Command Line Option Syntax

`+pal`

Description:

At +O2 and +O3, prepares an application for routine level and loop-level profiling with CXperf.

The +pal option is invalid with the +O4 or +O[no]all optimization options. Also, +pal is incompatible with the +A, -G, and -s options.

For More Information:

Refer to the CXperf(1) man page. (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Standards Related Options

The following options related to the ANSI/ISO C++ International Standard are accepted by the compiler.

-Aa
: Enable the use of standard options (-Wc,-koenig_lookup,on and -Wc,-ansi_for_scope,on).

-Wc,-ansi_for_scope,[on][off]
: Enable or disable the scoping rules for init-declarations in `for` statements.

-Wc,-koenig_lookup,[on][off]
: Enable or disable argument-dependent lookup rules (also known as Koenig lookup).

-Aa Command Line Option Syntax

```
-Aa
```

Description:

-Aa instructs the compiler to use Koenig lookup and strict ANSI `for` scope rules. The option is equivalent to specifying -Wc,-koenig_lookup,on and -Wc,-ansi_for_scope,on. The default is off.

Usage

The standard features enabled by -Aa may be source incompatible with earlier C and C++ features.

For More Information:

- Standardizing Your Code

-Wc,-ansi_for_scope,[on][off] Command Line Option Syntax

```
-Wc,-ansi_for_scope,[on]
                    [off]
```

Description:

This 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 in 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

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] Command Line Option Syntax

```
-Wc,-koenig_lookup,[on]
                   [off]
```

Description:

This 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 disabled.

Example:

In the following example, if the option is not enabled (the current default), 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);
}
```

For More Information:

- namespace and using Keywords

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.

-t*x*,*name*
: Substitutes *name* in place of subprocess *x* .

-W*x*,*args*
: Passes the option *arg* to subprocess *x* of the HP aC++ compiling system.

-t*x*,*name* Command Line Option Syntax

```
-tx ,name
```

Description:

Substitutes or inserts subprocess *x* using *name* .

This 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.

Example:

```
aCC -ta,/users/sjs/myasmb file.s
```

Invokes the assembler `/users/sjs/myasmb` instead of the default assembler `/usr/ccs/bin/as` to assemble and link `file.s`.

For More Information:

- More Examples of `-t`.
- The -W Option. (`-W` allows you to pass options to subprocesses.)
- Major Components of the HP aC++ Compiling System

More Examples of -t

Substituting for c++filt

```
aCC -tf,/new/bin/c++filt file.C
```

Compiles `file.C` and specifies that `/new/bin/c++filt` should be used rather than the default `/opt/aCC/bin/c++filt`.

Substituting for ctcom

```
aCC -tC,/users/proj/ctcom file.C
```

Compiles `file.C` and specifies that `/users/proj/ctcom` should be used instead of the default `/opt/aCC/lbin/ctcom`.

Substituting for All Subprocesses

```
aCC -tx,/new/&driver; file.C
```

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/ctcom` runs rather than the default `/opt/aCC/lbin/ctcom`.

x Parameter

x is one or more identifiers indicating the subprocess or subprocesses. The value of *x* can be one or more of the following:

`a`
: Assembler (standard suffix is `as`).

`b`
: C compiler driver (cc), used to invoke the assembler.

`C` (upper case)
: HP aC++ compiler (standard suffix is `ctcom`).

`f`
: Filter tool (`c++filt`).

`l`
: Linker (standard suffix is `ld`).

`u`
: Code generator when using +O4 or performing profile-based optimization (standard suffix is `ucomp`).

`x`
: All subprocesses.

name **Parameter**

This parameter is either the full path name of the executable file that will be run, or a prefix that will be concatenated to the default path name.

If `x` is a single identifier, `name` represents the full path name of the new subprocess. 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.

-W*x,args* Command Line Option Syntax

`-W`*`x`* `,`*`arg1[,arg2,..,argn]`*

Description:

Passes the arguments *arg1* through *argn* to the subprocess *x* of the compilation. The arguments are of the form:

```
-argoption[,argvalue]
```

Example1:

To see which include files led to an error or warning, specify the `-Wc,-diagnose_includes,on` option.

```
aCC -Wc,-diagnose_includes,on file.C
```

Specify `-Wc,-diagnose_includes,off` (the default) to turn the option off.

Example2:

```
aCC -Wc,-v file.C
```

Compiles `file.C` and passes the option `-v` to the linker.

```
aCC -Wl,-v file.C
```

Compiles `file.C` and passes the option `-v` to the linker.

For More Information:

- Passing Standards Related Options to the Compiler
- More Examples of -W
- Using -W for Linking Shared or Archive Libraries
- The +A option for linking archive libraries.
- The -t Option. (`-t` allows you to substitute subprocesses in place of the defaults.)

More Examples of -W

Passing Options to the Linker with -W

```
aCC file.o -Wl,-a,archive -lm
```

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.

Passing Multiple Options to the Linker with -W

```
aCC -Wl,-a,archive,-m,-v file.o -lm
```

Links `file.o` and passes the options `-a archive`, `-m`, and `-v` to the linker.

This case is similar to the previous example, with additional options. `-m` indicates that a load map should be produced. `-v` requests verbose messages from the linker.

argn Parameters

Each argument, *arg1* , *arg2* , through *argn* to the `-W` option 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.

x Parameter

x is one or more identifiers indicating a subprocess or subprocesses. The value of *x* can be one or more of the following:

`a`
: Assembler (standard suffix is `as`).

`b`
: C compiler driver (cc), used to invoke the assembler.

`C` (either upper or lower case)
: HP aC++ compiler (standard suffix is `ctcom`).

`d`
: The driver program, aCC

`f`
: Filter tool (`c++filt`).

`l`
: Linker (standard suffix is `ld`).

Template Options

By using a template option on the aCC command line, you can:

- Invoke the automatic instantiation mechanism (assigner).
- 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.

+inst_all

Requests instantiation of all templates.

+inst_auto
: Requests automatic instantiation.

+inst_close
: Requests closure with regard to template instantiation (for libraries that contain templates).

+inst_directed
: Requests that no templates be instantiated (except explicit instantiations) and suppresses the output of assigner information in object files.

+inst_implicit_include
: Requests HP C++ style template files.

+inst_include_suffixes
: Requests HP C++ template definition file name suffixes.

+inst_none
: For automatic (assigner) instantiation, requests that no templates be instantiated (except explicit instantiations).

+inst_used
: Requests instantiation of templates that are used.

+inst_v
: Requests verbose information about template processing.

Template Usage:

- 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.
- Only the +inst_auto option invokes the automatic instantiation mechanism and thus the assigner. All other template options (except the default, +inst_compiletime) write assigner information into .o files, allowing the files to be used with automatic instantiation if desired.

See Also:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

+inst_all Command Line Option Syntax

```
+inst_all
```

Description:

Causes the compiler to instantiate all template functions and all static data members and member functions of template classes defined in a **translation unit**, regardless of whether or not they are used, and to place these instantiations in the resulting object file.

This option allows existing instantiations in a translation unit to be used by the assigner to satisfy instantiation requests in other translation units.

NOTE: Because +inst_all instantiates all templates, it is essential to have a thorough understanding of your application and its template usage in order to use +inst_all effectively. Otherwise, duplicate symbols may result.

Usage:

This option is useful when you want to insure the file location of all templates defined in a given translation unit (for example, when preparing an object code library for distribution).

Example:

The following example compiles `file1.C` and places instantiations in file1.o.

```
aCC -c +inst_all file1.C

//file1.C
<p>
// Define foo function template.
template <class T> T foo (T i) {return i; };
<p>
// Define S class template.
template <class T> class S {
public:
          int n;
          static int m;
          int f();
          static int g();
};
<p>
template <class T> int S<T>::m = 1;
template <class T> int S<T>::f() { return 1; };
template <class T> int S<T>::g() { return 1; };
<p>
// Instantiate template class S with int to define object h.
<p>
S<int> h;         // S<int>::m, S<int>::f(), and S<int>::g()
                  // are instantiated and placed in file1.o
<p>
// Instantiate template function foo with int.
<p>
int k=foo(1);     // foo<int> is instantiated and placed in
                  // file1.o
```

Migration Note: Note that the +inst_all option differs from the HP C++ -pta option which instantiates all members of used template classes and all needed template functions.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

+inst_auto Command Line Option Syntax

```
+inst_auto
```

Description:

Requests that the automatic instantiation mechanism instantiate every template used if it is listed in the corresponding .I file. All used template functions, all static data members and member functions of template classes, and all explicit instantiations are instantiated.

When you link or create a shared library, if there is more than one object file on the command line, the assigner determines which object file is to contain a given instantiation.

To use +inst_auto, you must specify it at both compile-time and when creating an executable or a shared library.

Usage:

This option is necessary to insure that an archive library prepared for distribution is compatible with such a library prepared using the prior default (assigner) instantiation mechanism. It also facilitates use of the assigner by the library user. Refer to Deciding which Mechanism to Use.

Example:

The following example compiles file1.C and file2.C. Instantiations are placed in either file1.o or file2.o, as determined by the assigner during the closure operation.

Note the use of the +inst_close option to satisfy all needed template instantiations.

```
// initial compile
aCC -c +inst_auto file1.C file2.C

// closure operation to satisfy all unsatisfied instantiations
// in file1.o and file2.o without creating duplicate instantiations
aCC -c +inst_auto +inst_close file1.o file2.o
```

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information
- Creating and Using Libraries

+inst_close Command Line Option Syntax

```
+inst_close
```

Description:

Use +inst_close along with the +inst_auto option to specify that **automatic instantiation** be used to **close** a set of **link units**. This option prevents reinstantiation of any already instantiated templates in the .o files or libraries on the command line.

Note that the -c option must be used with +inst_close, otherwise an executable file or (with -b) a shared library is created.

Usage:

+inst_close is used when **closing** a set of .o files to create a library.

If you want to create a template library that uses templates, unlike a non-template library, you must instantiate the templates before linking the library.

Archive Library Example

To close and create an archive library containing templates, and then link the library to produce an application, use the following commands:

aCC -c +inst_auto -Idir mylib*.C
Compile library source files containing templates.

aCC -c +inst_auto +inst_close mylib*.o
Close mylib*.o files to create template instantiations in the .o files. (-c prevents linking.)

ar cr mylib.a mylib*.o

Create the mylib.a archive library.

aCC -Idir myfile.C mylib.a -o application
Link mylib.a with myfile.C to create application.

NOTE: If a library is dependent on another template library, that template library must be on the command line when you close the dependent library. If you build an application with the dependent library, the dependee library should also be used in the link.

Shared Library Example

To close and create a shared library containing templates, and then link the library to produce an application, use the following commands:

aCC +z -c +inst_auto -Idir mylib*.C
Compile library source files containing templates.

aCC -c +inst_auto +inst_close mylib*.o
Close mylib*.o files to create template instantiations in the .o files. (-c prevents linking.) Refer to the following NOTE.

aCC -b +inst_none -o mylib.sl mylib*.o
Create the mylib.sl shared library.

aCC -Idir myfile.C mylib.sl -o application
Link mylib.sl with myfile.C to create application.

NOTE: If desired, you can append one or more library names to this command line, indicating that you do not want duplicate instantiations between any libraries on the command line.

For example, you may have many shared libraries attached to an a.out. And you do not want to list all of these libraries on the -b command line when you create a shared library. However, you do want to be sure there are no duplicate symbols.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information
- Creating and Using Libraries

+inst_directed Command Line Option Syntax

```
+inst_directed
```

Description:

Indicates to the compiler that no templates are to be instantiated (except explicit instantiations) and suppresses assigner output in object files.

Without the use of +inst_directed, instantiation information needed by the assigner is placed in object files even when you have not requested automatic (assigner) instantiation with +inst_none.

Usage:

If you are using only explicit instantiation and have not requested automatic (assigner) instantiation, specify

+inst_directed instead of +inst_none.

Example:

```
aCC +inst_directed prog.C
```

Compiles `file.C` with the resulting object file containing no template instantiations, except for any explicit instantiations coded in your souce file.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

+inst_implicit_include Command Line Option Syntax

```
+inst_implicit_include
```

Description:

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, 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/or you wish to follow this rule when compiling with HP aC++, you need to specify the +inst_implicit_include option.

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.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information
- +inst_include_suffixes Command Line Option Syntax

+inst_include_suffixes Command Line Option Syntax

```
+inst_include_suffixes=list "
```

list is a set of space separated file extensions or suffixes, enclosed in quotes, that template definition files can have.

Description:

Specifies which file name extensions the compiler uses to locate template definition files. This option must be used with the +inst_implicit_include option.

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/or 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.

Example:

```
+inst_include_suffixes=".c .C"
```

Specifies that template definition files can have extensions of `.c` or `.C`.

Migration:

The +inst_include_suffixes option is equivalent to the HP C++ -ptS option.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information
- +inst_implicit_include Command Line Option Syntax

+inst_none Command Line Option Syntax

```
+inst_none
```

Description:

For automatic (assigner) instantiation, indicates to the compiler that no templates are to be instantiated (except explicit instantiations).

Usage:

If you know that templates in a translation unit have been instantiated in another translation unit that will participate in the link, you might want to use +inst_none to prevent unneeded instantiation attempts.

If you use +inst_auto to create a shared library from .o files that have already been closed, you should use +inst_none.

Example:

```
aCC +inst_none file.C
```

Compiles `file.C` with the resulting object file containing no template instantiations, except for any explicit instantiations coded in your souce file.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

+inst_used Command Line Option Syntax

```
+inst_used
```

Description:

Causes the compiler to instantiate all template class members and all template functions that are `used` in a **translation unit** and to place these instantiations in the resulting object file.

Template instantiation operates on member functions of template classes and template functions. Use of a template is a call to such a function. For example:

```
template <class T>
class A {
   public:
      void boo();
   };
...
A<int> a;
...
a.boo();                    // a use
```

This option allows existing instantiations in a translation unit to be used by the assigner to satisfy instantiation requests in other translation units.

NOTE: Because +inst_used instantiates all used templates, it is essential to have a thorough understanding of your application and its template usage in order to use +inst_used effectively. Otherwise, duplicate symbols may result.

Usage:

This option may be useful when compiling a large application or library containing many templates, only some of which are used.

+inst_used is essentially equivalent to default compile-time instantiation. However, if you intend to use the instantiations in a translation unit (X.C) to satisfy instantiation requests in other translation units, using the automatic instantiation mechanism, you should specify +inst_used instead of using the default to compile X.C. For example:

```
aCC -c +inst_used X.C
aCC +inst_auto Y.C X.o
```

Example:

```
aCC -c +inst_used file.C
```

Compiles `file.C` and places instantiations for all used members of template classes and all used template functions in `file.o`.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

+inst_v Command Line Option Syntax

`+inst_v`

Description:

Enables verbose mode, sending a step-by-step description of template processing to stderr. +inst_v works with all template processing options except the default compile-time instantiation mechanism (+inst_compiletime). Messages are produced when:

- template entities are instantiated (generated by the compiler)
- template entities are assigned (generated by the assigner)
- template entities are unassigned (generated by the assigner)
- the assigner is run (generated by the driver)
- a translation unit is re-compiled (generated by the driver)

In addition, when the assigner cannot satisfy an instantiation request, a message stating the reason is generated.

Usage:

You can use +inst_v to help determine the locations of errors in instantiation. Since verbose output tells you where instantiations have been made, you might also use it to determine the layout of explicit instantiation in applications that have many modules produced by a number of different developers.

Example:

```
aCC +inst_auto +inst_v file.C
```

Compiles `file.C` and provides details of template processing.

Migration:

The +inst_v option is similar to the HP C++ -ptv option.

For More Information:

- Using HP aC++ Templates for an overview
- Using Templates in HP aC++ for more detailed information

Requesting Verbose Compile and Link Information

Use the following options to obtain additional information about:

- what HP aC++ is doing while compiling or linking your program
- which subprocesses would execute for a given command line, without running the compiler
- the current compiler and linker version numbers
- execution time

+dryrun
: Requests compiler subprocess information without running the subprocesses.

+inst_v
: Requests verbose information about template processing.

+Oinfo
: Requests optimization information.

+time
: Requests execution times.

-v
: Requests verbose information of the compilation process.

-V
: Requests the current compiler and linker version numbers.

-Wl,-v
: Requests verbose messages from the linker.

+dryrun Command Line Option Syntax

```
+dryrun
```

Description:

Causes aCC (the driver) to generate subprocess information for a given command line without running the subprocesses.

Usage:

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:

aCC +dryrun app.C

The above command line gives the same kind of information as the -v option but without running the subprocesses.

+time Command Line Option Syntax

```
+time
```

Description:

This 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.

Usage:

Useful in the development process, for example, when tuning an application's compile-time performance.

Examples:

The following command line:

```
aCC +time app.C
```

generates information like this:

```
process: compiler       0.94/u   0.65/s   4.35/r
process: ld             0.37/u   0.76/s   3.02/r
```

The following command line:

```
aCC -v +time app.C
```

generates information like this:

```
/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/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/libcl.a -lc /usr/lib/libdld.sl >/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 Command Line Option Syntax

```
-v
```

Description:

Enables verbose mode, sending a step-by-step description of the compilation process to `stderr`.

Usage:

This is especially useful for debugging or for learning the appropriate commands for processing a C++ file.

Example:

The following compiles `file.C` and gives extra information about the process of compiling.

```
aCC -v file.C

/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/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/libcl.a -lc /usr/lib/libdld.sl >/usr/tmp/AAAa28149 2>&1
```

```
removing /usr/tmp/AAAa28149
```

-V Command Line Option Syntax

`-V`

Description:

Displays the version numbers of the current compiler and linker (if the linker is executed).

Usage:

Use this option whenever you need to know the current compiler and linker version numbers.

Example:

```
aCC -V app.C
```

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 -g. Below are equivalent command lines you could use:

```
aCC my_file.C -v -g1
aCC my_file.C -vg1
aCC my_file.C -vg1
aCC -vg1 my_file.C
```

Compiler Command Syntax and Environment Variables

Compiler Command Syntax:

`aCC [options ] [files ]`

Description:

The `aCC` command (the driver) invokes the HP aC++ compiling system.

CAUTION: 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.

Example:

```
aCC prog.C
```

Compiles the source file `prog.C` and puts the executable code in the file `a.out`.

For More Information:

- More Examples of the aCC Command
- Using Environment Variables with the aCC Command
- The Manual Page for the aCC(1) Command
- Setting Up Floating Installation (how to have more than one version of HP aC++ on the same system)

---

files on the aCC Command Line

files represents a list of one or more files containing source or object code to be compiled or linked.

Each file can be:

- A C++ source file (.C file)
- A preprocessed source file (.i file)
- An assembly language source file (.s file)
- An object file (.o file)
- A library file (.sl or .a file)

All other files are passed directly to the linker by the compiler. C++ source files can also reference C++ header files (.H files) using the `#include` preprocessor directive.

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 came from other directories.

C++ Source Files

HP aC++ **source files** must be named with extensions beginning with either `.c` or `.C`, possibly followed by additional characters.

If you compile only, 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.

CAUTION: It is recommended that your source files have extensions of `.c` or `.C` only, without additional characters. While file extensions other than `.c` or `.C` are permitted for portability from other systems, other endings may not be supported by HP tools and environments.

C++ Header Files

Typically, header files are referenced in C++ source files using the `#include` preprocessor directive.

Preprocessed C++ Source Files

Files with names ending in `.i` are assumed to be preprocessor output files.

Files ending in `.i` are processed the same 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

Files with names ending in `.s` are assumed to be assembly source files.

The compiler invokes the assembler through cc to produce `.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

Files with `.o` extensions are assumed to be relocatable object files that are to be included in the linking.

The compiler invokes the linker to link the object files and produce an executable file.

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

Library Files

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

More Examples of the aCC Command

Compiling and Renaming the Output File

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

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
```

Compiles `prog.C` and includes information allowing you to debug the program with the HP/DDE Debugger, `dde`.

Compiling Without Linking

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

Compiles `prog.C` and puts the object code in the file `prog.o`. Does not link the object file and does not create an executable file.

Linking Several Object Files

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

Links the listed object files and puts the executable code in the file `a.out`.

CAUTION: 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
```

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.

Environment Variables

You can use the following environment variables with HP aC++:

`CXXOPTS`
: Specify command line options automatically.

`CCLIBDIR`
: Specify additional directories for the linker to search for libraries.

`CCROOTDIR`
: Use this when compiler subprocesses are in alternate directories.

`TMPDIR`
: Change the location of temporary files that the compiler creates.

The CXXOPTS Environment Variable

Syntax:

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

Description:

Provides a convenient way to include frequently used command line options automatically. Options before the vertical bar (|) are placed before any command line options to aCC. Options after the vertical bar are placed after any command line options. 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.

Just set the environment variable and the options you want are automatically included each time you execute the aCC command.

Usage:

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

Example:

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

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
```

The CCLIBDIR Environment Variable

Syntax:

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

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

Description:

Causes the aCC command to search for libraries in an alternate directory before searching in the default directory, /opt/aCC/lib.

Example:

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

Specifies that HP aC++ search the directory /mnt/proj/lib for libraries, then search the directory /opt/aCC/lib.

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

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

See Also:

Use the -L*directory* option to specify additional directories for the linker to search for libraries.

The CCROOTDIR Environment Variable

Syntax:

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

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

Description:

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.

Example:

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

Specifies that HP aC++ search the directories under `/mnt/CXX2.1` (/mnt/CXX2.1/bin and /mnt/CXX2.1/lbin) for subprocesses rather than their respective default directories.

The TMPDIR Environment Variable

Syntax:

```
export TMPDIR=directory    ksh 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.

Description:

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

Example:

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

Specifies that HP aC++ should put all temporary files in `/mnt/temp`.

Floating Installation

As of HP aC++ A.01.12 (for HP-UX 10.x) and HP aC++ A.03.10 (for HP-UX 11.x) and subsequent versions, 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.

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

aC++ on the same system at the same time.

Note that only the files in /opt/aCC are affected by floating installation. No matter which HP aC++ driver you are using, the compiler still uses the libraries, linker, and other files located in /usr/lib and /usr/ccs.

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

CAUTION: Floating installation is not intended for use 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, it is easy to 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.03.10 is the prior version, and A.03.13 is the more recent version.

1. Copy the prior version to another directory.

   ```
   cp -rp /opt/aCC /opt/aCC.03.10
   ```

2. Use swinstall to install the new (in this case, A.03.13) version.

3. To invoke the A.03.10 compiler with its absolute path:

   ```
   /opt/aCC.03.10/bin/aCC app .C
   ```

   Alternatively, you could change your PATH environment variable or set up an alias for the absolute path.

4. To invoke the A.03.13 compiler:

   ```
   aCC app .C
   ```

The HP aC++ driver accesses subprocesses for the version you invoke.

Migrating from HP C++ (cfront) to HP aC++

If you are migrating code from HP C++ (cfront) to HP aC++, the following list presents differences in syntax and functionality that you may need to consider. Click here for General Migration Guidelines and Tips.

NOTE: 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++.

- Command-Line
- Compiler Coexistence
- Debugging
- Error and Warning Messages
- Exception Handling
- Libraries
- Performance and File Size
- Preprocessing
- **Standardizing Your Code (Syntax and Semantics)**
- Templates
- Translator Mode not Supported

For More Information

- Additional Sources of Migration Information

General Migration Guidelines and Tips

As you begin conversion of your HP C++ (cfront) code to HP aC++, you may want to read about the following topics:

- Getting Started
- Writing Code for Both Compilers using __cplusplus Macro
- Explicit Loading and Unloading of Shared Libraries
- Memory Allocation

Getting Started with Migration

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.
   - Remember 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. You must modify your cfront code to handle memory allocation failures to avoid having these failures cause 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.
   - You may want to use the +inst_none 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++, as in the following 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++, you will need to 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>

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
- Obsolete Command-Line Options
- Changed Command-Line Options

New Command-Line Options

New options for HP aC++ are described below. These options are not available for HP C++ (cfront), however, if a related option exists, it is noted here.

`-g0`

Replaces the -g debugger option. It generates full debug information for the debugger.

For more information, refer to HP aC++ Debugging Options.

`+help`

Invokes this *HP aC++ Online Programmer's Guide* .

`+noeh`

Turns exception handling off. In HP aC++ exception handling is on by default.

Note that in HP C++ (cfront), exception handling is off by default. To turn it on, you must use the `+eh` option which is obsolete in HP aC++.

Precompiled Header File Options

Reduce 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, refer to:
  - Using Templates.
  - Migration Considerations when Using Templates.

Obsolete Command-Line Options

HP C++ (cfront) options that are not available in HP aC++ are listed in the following sections. If a related option exists for HP aC++, it is noted.

Select from an Alphabetical List of Obsolete Options or a category of options:

- Debugging Option
- Exception Handling Option
- Library Option
- Null Pointer Option
- Preprocessor Options
- Template Options
- Translator Mode Options
- Virtual Table Options

Alphabetical List of Obsolete Command-Line Options

Select the option for which you want more information:

- +a0 +a1 -Aa -Ac -C -depth +e0 +e1 +eh -F
- -Fc +i +m -pta -ptb -pth -ptH"*list* " -ptn -ptr"*path* "
- *-pts* -ptS"*list* " *-ptv* +R*number* -tx*name* +T -Wx,*args*
- +x*file* -y -Z

Obsolete Debugging Option

The following HP C++ (cfront) option is not supported in HP aC++.

`-y`
: For 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 for HP aC++.

Obsolete Exception Handling Option

The following HP C++ (cfront) option is not supported in HP aC++.

`+eh`
: To use exception handling with HP C++, you must use this option when compiling all files in your application.

  In HP aC++, exception handling is in effect by default. To turn exception handling off, you must compile with the +noeh option.

Obsolete Library Option

The following HP C++ (cfront) library option is not supported in HP aC++.

`-depth`
: For HP C++ (cfront), this option is used to instruct the run-time 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.

For HP aC++, -depth funtionality is the default. The option is therefore unnecessary.

Obsolete Null Pointer Option

The following HP C++ (cfront) library option is not supported in HP aC++.

`-Z`
: Allow dereferencing of null pointers at run time. The value returned from a dereferenced null pointer is zero. This is the default behavior for HP C++ (cfront) and for HP aC++.

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.

`-Aa`
: Requests the ANSI mode HP C++ preprocessor, cpp.ansi. This is the HP C++ (cfront) default.

`-Ac`
: Requests the compatibility mode HP C++ preprocessor, cpp, not available in HP aC++.

For this release of HP aC++, the following options are not supported.

`-C`
: Prevents the preprocessor from stripping comments from your source file; 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, as in the following example:

  ```
  CC prog.C -I /opt/aCC/include -I /opt/aCC/include/iostream -I /usr -I /usr/include
  ```

For More Information

- Migration Considerations Related to Preprocessing
- Preprocessing in HP aC++

Obsolete Template Options

In HP aC++, templates are processed differently than in HP C++ (cfront). New template options provide additional functionality. Refer to Using HP aC++ Templates and Migration Considerations when Using Templates for information.

The following HP C++ (cfront) template options are not supported in HP aC++.

`-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 compile time.

`-ptr`*`path`*
: Specifies an alternate location for the template repository.

`-pts`
: Splits template instantiations into separate object files.

`-ptS"`*`list`*`"`
: Specifies file name extensions for template definition files.

  For HP aC++, use the +inst_include_suffixes option. Note this option must be used with the +inst_implicit_include option.

`-ptv`
: Gives verbose information about template processing.

  For HP aC++, use the +inst v option.

`-t`*`x,name`*
: The following values for *x* are related to template subprocesses and are not supported in HP aC++.
  - `i` -- Link-time template processor, c++ptlink
  - `r` -- Compile-time template processor, c++ptcomp

`-W`*`x,args`*
: The following values for *x* are related to template subprocesses and are not supported in HP aC++.
  - `i` -- Link-time template processor, c++ptlink
  - `r` -- Compile-time template processor, c++ptcomp

Obsolete Translator Mode Options

HP aC++ is ANSI C conformant and does not support a C++ to C translator mode. The following HP C++ (cfront) translator mode options are not valid in HP aC++.

`+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, sending 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 having the file name suffix ..c in the current directory.

`+m`
: Provides maximum compatibility with the USL C++ implementation.

`+R`*`number`*
: Allows only the first *number* register variables to actually be promoted to the register class.

`+T`
: Requests translator mode.

`-t`*`x,name`*
: The following values for *x* are related to translator mode and are not supported in HP aC++.
  - `0 (zero)` -- C compiler
  - `c` -- C compiler
  - `m` -- merge tool, c++merge
  - `p` -- preprocessor
  - `P` -- patch tool, c++patch

`-W`*`x,args`*
: The following values for *x* are related to translator mode and are not supported in HP aC++.
  - `0 (zero)` -- C compiler
  - `c` -- C compiler

- `m` -- merge tool, c++merge
- `p` -- preprocessor
- `P` -- patch tool, c++patch

`+x`*`file`*

Reads a file of data types, sizes, and alignments which the compiler uses when generating code.

Obsolete Virtual Table Options

The following HP C++ (cfront) virtual table options do not apply in HP aC++.

`+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.

Obsolete Template and Translator Mode Arguments for -t*x*,*name*

For the `-tx,name` option, the following arguments for *x* , related to translator mode and template subprocesses, 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

Obsolete Template and Translator Mode Arguments for -W*x*,*args*

For the `-Wx,args` option, the following arguments for *x* , related to translator mode and template subprocesses, 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

Changed Command-Line Options

Functionality for the following options is different for HP C++ (cfront) than it is for HP aC++.

-E

Functionality of the -E option has changed. It now runs the preprocessor only on named C++ files, not on assembly files, and sends the result to standard output (`stdout`).

-g

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 -g and generates full debug information for the debugger.

For more information, refer to HP aC++ Debugging Options.

`-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
- `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

Compiler Coexistence

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
```

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 -g and generates full debug information for the debugger.

For more information, refer to HP aC++ Debugging Options.

HP aC++ Messages

As you migrate your code from HP C++ to HP aC++, you are likely to see many error and warning messages related to standards based syntax.

It may be helpful to compile using HP C++ (cfront) with the +p option. Then when the compiler encounters a standards based reserved word that is used as an identifier, it generates a warning message indicating that this

syntax will cause an error in HP aC++.

Interpreting HP aC++ Messages

The aC++ compiler can issue a large variety of diagnostics in response to unexpected situations or suspicious constructs. These diagnostics can be classified as follows:

Command Errors
: These are issued when the command line is not correctly formed and the compiler cannot proceed with compilation.

Command Warnings
: These sometimes occur when an option is not recognized, but compilation proceeds without that option.

Fatal Errors
: These are issued for ill formed programs for which the compiler cannot recover reliably. Syntax errors usually fall into this category. No object file will be generated if such an error is encountered.

Future Errors
: These are actually serious warnings indicating that a language rule was violated, but the compiler will continue generating code. These warnings can be turned into hard errors by using the `+p` option (pedantic mode).

Anachronisms
: These warnings also diagnose ISO/ANSI C++ language violations. Code that triggers this sort of diagnostic was considered legal in the past.

Warning
: These help in identifying possible sources of bugs, often because the code triggers behavior that is not precisely defined by the C++ standard.

Suggestion/Informational
: These diagnostics are not emitted unless the -w option is provided. In that case, the compiler attempts to identify more suspicious constructs.

Tool Errors
: Very rarely, aC++ may fail in a component that is not specific to the C++ language (e.g., the PA-RISC optimizer); in that case, a `Tool Error` is emitted. The compilation process cannot recover from these, and they are often a sign of a defect in the compiler.

Frequently Encountered Messages

Frequently encountered diagnostic message numbers are listed and described in the *HP aC++ Transition Guide* at the following URL:

`http://www.hp.com/esy/lang/cpp/tguide/`

Migration Considerations when Using Exception Handling

When migrating exception handling code, be aware of the following characteristics of HP aC++ which differ from those of HP C++ (cfront):

- Exception Handling is the Default
- Memory Allocation Failure and operator `new`
- Possible Differences when Exiting a Signal Handler
- `setjmp/longjmp` Behavior
- Calling `unexpected`
- Unreachable `catch` Clauses
- Throwing an Object having an Ambiguous Base Class

See Also

- Standard Exception Classes
- Standard Exceptions

Exception Handling is the Default

In HP aC++ exception handling is on by default. Use the `+noeh` option if you need to turn exception handling off. Note that 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 which is obsolete in HP aC++.

CAUTION: 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 (note that HP aC++ run-time 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
- run-time 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 avoid a program abort, do one of the following:

- Write try/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.

  For example:

```
operator new (size_t size, const nothrow_t &) throw();
operator new [] (size_t size, const nothrow_t &) throw();
    .
    .
    .
#include
#include

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
```

```
}

void 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 via a throw (which, according to the ANSI/ISO C++ International Standard remains "undefined"), 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 which considers that prior to the first call, a legal exception cannot be thrown which would consider the current try block's handlers as candidates to catch the exception.

In HP C++ the try keyword defines the beginning of a try block.

Thus, if a signal were taken while executing between the try keyword and the first call's return point, a throw from the signal handler would not find the associated handlers to be candidates for catching the exception.

Differences in setjmp/longjmp Behavior

Interoperability with setjmp/longjmp (undefined by the ANSI/ISO C++ International Standard) is unimplemented.

The standard has the following verbiage, suggesting that an implementation is not obligated to 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. If any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to longjmp(jbuf, val) at the throw point that transfers control to the same (destination) point has undefined behavior.

Calling `unexpected`

Unlike HP C++, in HP aC++ an unexpected-handler cannot throw anything it pleases. If it wants to exit via a throw, it must throw an exception that is legal according to the exception specification that caused unexpected() to be called, unless that exception specification includes the predefined type `bad_exception`. If it does include 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       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 another example, illegal because `my_unexpected_handler` is rethrowing an int. A possible conversion is to throw &x instead, since this is a pointer to int and therefore legal with respect to the original throw specification. Alternatively, you could 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 error for the throw of an object having 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->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

Choose from the following sections for information about library migration from HP C++ (cfront) to HP aC++. Note that the Complex Library and the Task Library are no longer supported. For information about availability of the HP Codelibs Library, contact your HP support representative.

Standards Based Libraries

HP aC++ provides the following libraries that are not part of the HP C++ (cfront) compiler: It is highly recommended that you use these standards based libraries whenever possible, instead of the cfront compatibility libraries.

- Standard C++ Library
- Tools.h++ Library
- HP aC++ Run-time Support Library

HP C++ (cfront) Compatibility Libraries

HP aC++ provides the following libraries whose functionality is part of the HP C++ (cfront) compiler. These libraries **are not standards based**.

- IOStream Library
- Standard Components Library

HP aC++ A.01.21 Run-time Support Library

The following HP aC++ libraries replace the cfront based HP C++ libraries:

- /opt/aCC/lib/libCsup.sl -- shared version, the default
- /opt/aCC/lib/libCsup.a -- archive version

Since the K & R float to double promotion rule is not supported, no equivalents to libC and libC.ansi are available.

For More Information

- What the library supports.

For your reference, the files containing the HP C++ (cfront) run-time libraries are listed below. Different libraries are used depending on whether or not you use exception handling.

HP C++ 3.x Stream Library File Locations (default version, without exception handling)

- `/opt/aCC/lib/libC.a`
- `/opt/aCC/lib/libC.sl`
- `/opt/aCC/lib/libC.ansi.a`
- `/opt/aCC/lib/libC.ansi.sl`

HP C++ 10.x Stream Library File Locations (default version, without exception handling)

- `/usr/lib/libC.a`
- `/usr/lib/libC.sl`
- `/usr/lib/libC.ansi.a`
- `/usr/lib/libC.ansi.sl`

HP C++ (cfront) Stream Library File Locations (exception handling version)

- `/opt/aCC/lib/aCC/eh/libC.a`
- `/opt/aCC/lib/aCC/eh/libC.ansi.a`

IOStream Library

The shared version of this library is located at /usr/lib/libstream.sl. The archive version is at /usr/lib/libstream.a.

Manual Pages

- `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

Header Files

The following header files are for use with the IOStream library. To direct the compiler to search these header files, use `-I/opt/aCC/include/iostream`.

- 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++ version 1.2

Standard Components Library

The Standard Components Library is provided for compatibility with the cfront based HP C++ compiler. In place of the Standard Components Library, it is highly recommended that you use the similar features of the Standard C++ Library.

When using the Standard Components Library, note the following:

- There are command-line differences between HP C++ (cfront) and HP aC++:
  - with cfront use -I /opt/CC/include/SC
  - with HP aC++ use -I /opt/aCC/include/SC

  Note that -l++ is still needed on the link line.

- The following program development tools that are part of the HP C++ (cfront) compiler are not part of the HP aC++ compiler:
  - fscpp (symbolic freestore manager)
  - g2++comp
  - dem (demangler)
  - publik
  - incl
  - hier

- compl is a member funtion in the Bit class, which is defined in Bit.h. If you use the Bit class, you should change your compl calls to compl_.

- In the HP C++ (cfront) compiler, two sets of Standard Components library files are provided, one for code that uses exception handling and one for code that does not. To see their locations choose from the following:
  - HP C++ 3.x Standard Components Library File Locations
  - HP C++ 10.x Standard Components Library File Locations

  HP aC++ has just one set of Standard Components library files located at:

  - `/opt/aCC/lib/lib++.a`
  - `/opt/aCC/lib/libGA.a`
  - `/opt/aCC/lib/libGraph.a`

Manual Pages

Manual pages are located at /opt/aCC/share/man/man3.Z. To invoke a man page from the command line, enter 3s after the man command and before the man page name. For example, to invoke the man page for Args:

```
man 3s Args
```

- `SC_intro(3C++)` -- introduction to Standard Components

- `Args(3C++)` -- command-line arguments
- `Array_alg(3C++)` -- operations on arrays
- `Bits(3C++)` -- variable-length bit strings
- `Block(3C++)` -- parameterized variable-size arrays
- `Fsm(3C++)` -- simple deterministic finite state machines
- `Graph(3C++)` -- entities and relationships
- `Graph_alg(3C++)` -- operations on graphs

- `List(3C++)` -- parameterized variable-length sequences
- `Map(3C++)` -- parameterized variable-size associative arrays
- `Objection(3C++)` -- rudimentary error-handling
- `Path(3C++)` -- path names and search paths
- `Pool(3C++)` -- special-purpose memory allocators

- `Regex(3C++)` -- regular expressions
- `Set(3C++)` -- parameterized unordered collections
- `Stopwatch(3C++)` -- program execution time measurement

- `String(3C++)` -- variable-length character strings
- `Strstream(3C++)` -- `iostream` specialized for `String(3C++)`
- `Symbol(3C++)` -- unique identifiers based on character strings
- `Time_intro(3C++)` and `Time(3C++)` -- absolute time, time zone and duration

Header Files

Standard Components header files and template source files are listed below. To direct the compiler to search these header files, specify `-I/opt/aCC/include/SC`.

- Args.h
- Array_alg.c
- Array_alg.h
- Bits.h
- Block.c
- Block.h
- Blockio.c
- Blockio.h
- Fsm.h
- Graph.h
- Graph_alg.h
- List.c
- List.h
- List_old.c
- List_old.h
- List_oldio.c
- List_oldio.h
- Listio.c
- Listio.h
- Map.c
- Map.h
- Mapio.c
- Mapio.h
- Objection.h
- Path.h
- Pool.h
- Regex.h
- Search_path.h
- Set.h
- Stopwatch.h
- String.h
- Strstream.h
- Symbol.h
- Ticket.h
- Time.h
- Tmppath.h
- bag.c
- bag.h
- bagio.c
- bagio.h
- ipcmonitor.h
- ipcstream.h
- ksh_test.h
- set.c
- set.h
- set_of_p.c
- set_of_p.h
- set_of_pio.c
- set_of_pio.h
- setio.c
- setio.h

Migration Considerations

- +inst_implicit_include option for cfront style template definition files

HP C++ 3.x Standard Components Library File Locations

Default version:

- `/usr/lib/lib++.a`
- `/usr/lib/libfs.a`
- `/usr/lib/libGA.a`
- `/usr/lib/libGraph.a`
- `/usr/lib/libg2++.a`
- `/usr/lib/incl2`
- `/usr/lib/hier2`
- `/usr/lib/publik2`

Exception handling version:

- `/usr/lib/aCC/eh/lib++.a`
- `/usr/lib/aCC/eh/libfs.a`
- `/usr/lib/aCC/eh/libGA.a`
- `/usr/lib/aCC/eh/libGraph.a`
- `/usr/lib/aCC/eh/libg2++.a`

HP C++ 10.x Standard Components Library File Locations

Default version:

- `/opt/CC/lib/lib++.a`
- `/opt/CC/lib/libfs.a`
- `/opt/CC/lib/libGA.a`
- `/opt/CC/lib/libGraph.a`
- `/opt/CC/lib/libg2++.a`
- `/opt/CC/lib/incl2`
- `/opt/CC/lib/hier2`
- `/opt/CC/lib/publik2`

Exception handling version:

- `/opt/CC/lib/eh/lib++.a`
- `/opt/CC/lib/eh/libfs.a`
- `/opt/CC/lib/eh/libGA.a`
- `/opt/CC/lib/eh/libGraph.a`
- `/opt/CC/lib/eh/libg2++.a`

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++. In place of the Complex library, it is recommended that you use the similar features of the Standard C++ Library.

To begin your migration:

- Replace #include with <complex>.
- Remove -lcomplex from the command-line.

Manual Pages Not Available

The following manual pages describing the complex library are not part of the HP aC++ product:

- `CPLX.INTRO(3C++)` -- introduction to the C++ complex mathematics library
- `cartpol(3C++)` -- cartesian 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 Not Available

The Complex library uses the complex.h header file.

HP C++ (cfront) Task Library Not Supported

The task library which is part of the HP C++ compiler product 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`.

Manual Pages Not Available

The following manual pages describing task library features are not part of the HP aC++ product:

- `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 Performance and File Size

For information about HP aC++ performance and file size considerations, refer to the following:

- Creating and Using Precompiled Header Files
- Performance Considerations when using Exception Handling
- HP aC++ Release Notes

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.

For More Information

- Obsolete Preprocessor Options
- Preprocessing in HP aC++
- Concatenating Tokens with the ## Operator

Migration Considerations Related to Standardization

The ANSI/ISO C++ International Standard redefines the C++ language in terms of rules, syntax, and features.

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.

See Also:

- Standardizing Your Code

Migration - C++ Semantics

When you migrate from HP C++ to HP aC++, your code will behave differently in the following areas, even

though it may compile without errors or warnings.

- Implicit Typing of Character String Literals
- Overload Resolution Ambiguity of Subscripting Operator
- When operator new fails, it throws an execption.
- Static constructors in shared libraries may be executed in a different order.
- Inline code is inlined more often.

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. For 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 like the following from breaking, the ANSI/ISO C++ International Standard does make a provision to allow the assignment of a string literal to a non-const pointer.

```
char *p = "B_STRING";
```

NOTE: The ANSI/ISO C++ International Standard defines the above as a deprecated feature, that is, not guaranteed to be part of the Standard in future revisions.

Also, in a conditional expression like the following, the conversion of const char * to char * is not provided for in this context.

```
char *p = f() ? "A" : "B";
```

The code could be changed in several ways, for 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 migrating 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 above with HP aC++ produces an error like the following:

```
   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 is issued (correctly) 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 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. The example above should therefore be replaced by:

```
#include

struct String {
   char& operator[](ptrdiff_t);
   operator char*();
   // ...
};

void f(String &s) {
   s[0] = '0';
}
```

Note that "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.

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 the greatest 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 .sl files together, as in the following example:

```
aCC file1.o file2.o lib1.sl lib2.sl lib3.sl
```

Given the above link-line, cfront would initialize file2.o and than file1.o, while HP aC++ initializes file1.o and than file2.o. You should 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 actually inline some functions even though you have requested inlining. 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 function you wanted inlined.

HP aC++ almost always inlines functions for which you have specified the `inline` keyword.

Migration - 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
- `for` Statement - New Scoping Rules
- `struct` as Template Type Parameter is Permitted
- Base Template Class Reference Syntax Change
- `typename` in Template Declarations
- Tokens after `#endif`
- `overload` not a Keyword
- Dangling Comma in `enum`
- Static Member Definition Required
- Declaring `friend` Classes
- Incorrect Syntax for Calls to operator new
- Using :: in Class Definitions
- Duplicate Formal Argument Names
- Ambiguous Function/Object Declaration
- Overloaded Operations `++` and `--`
- Reference Initialization
- Using `operator new` to Allocate Arrays
- Parentheses in Static Member Initialization List
- &qualified-id Required in Static Member Initialization List
- Non-constant Reference Initialization
- Digraph White Space Separators

Migration - Explicit `int` Declaration

HP C++

You do not need to explicitly specify int types.

HP aC++

You must explicitly declare int types. This change reduces opportunities for ambiguity between expressions involving function-like casts and declarations.

Change Needed

Explicitly declare int types.

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()
```

Migration - `for` Statement, New Scoping Rules

HP C++

Variables declared in the initializer list are allowed after the `for` statement.

HP aC++

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 option -WC,-ansi_for_scope,off) by default the new rules do not take effect.

Change Needed

At this time, 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. Therefore, it is recommended that you correct your code, by moving the declaration of the `for` loop variable to its enclosing block.

Example

The following code currently compiles error free 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) {
```

```
        }
}
```

Corrected code complies with ANSI/ISO C++ International Standard syntax and compiles with either compiler.

Migration - `struct` as Template Type Parameter is Permitted

HP C++

An error is generated when a `struct` is used as a template type parameter.

HP aC++

When a `struct` is used as a template type parameter, it is correctly compiled, in accordance with draft standard syntax.

Change Needed

This is a new feature.

Example

```
template  class A {
public:
struct T a;
};
struct B {};
A b;

Compiling the above code with HP C++ causes an error like the following:

CC: "DDB4325.C", line 3: error:  T of type any  redeclared as struct (1479)

The code compiles without error with HP aC++
```

Migration - Base Template Class Reference Syntax Change

HP C++

```
You can reference a member of a base template class without qualifying the member.
```

HP aC++

```
When referencing a member of a base template class, you should qualify the member by
adding a this->.
```

Change Needed

```
Adding this-> forces name resolution to be deferred until instantiation which allows the
compiler to find members in template base classes. This rule prevents the compiler from
finding names declared in enclosing scopes when that is unintended.
```

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.
```

Migration - Tokens after #endif

HP C++ Behavior

Any characters following the #endif preprocessor statement cause a warning and are ignored.

HP aC++ Behavior

Any characters following the #endif preprocessor statement cause an error and the program does not compile.

Change Needed

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++, you can change the code to the following:

```
int main(){
#ifdef FLAG
int i;
i=1;
#endif //FLAG
}
```

Migration - overload not a Keyword

HP C++

Using the overload keyword to specify that a function is an overloaded function causes an **anachronistic warning** and is ignored.

HP aC++

Using the overload keyword causes an error and the program does not compile.

Change Needed

Remove all uses 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;
}
```

Migration - Dangling Comma in enum

HP C++

A comma following the last element in an enum list is ignored.

HP aC++

A comma following the last element in an enum list generates an error.

Change Needed

Remove the comma after the last element.

Example

HP C++ accepts the following code. HP aC++ generates an error stating that the , is unexpected.

```
enum Colors { red,
              orange,
              yellow,
              green,
              blue,
              indigo,
              violet,     // This comma is illegal.
};
```

Migration - Static Member Definition Required

HP C++

Declaring a static member but not defining it is allowed.

HP aC++

Declaring a static member but not defining it is not allowed.

Change Needed

Add a definition of the 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 no warning nor error. Linking the resulting object file generates a linker (ld) error stating 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;
}
```

Migration - Declaring `friend` Classes

HP C++

Declaring friend classes without the class keyword is allowed.

HP aC++

Declaring friend classes without the class keyword generates an error.

Change Needed

Add the class keyword to all friend class declarations.

Example

Compiling the following code on HP C++ gives no warning nor 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;
}
```

Migration - Incorrect Syntax for Calls to operator new

HP C++

Incorrect syntax for the use of a value to a call to operator new is allowed.

HP aC++

An error is generated when this incorrect syntax for opertor new is used.

Change Needed

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++ gives no warning nor 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 would fix the problem.
int S:: f( ) { return 1;}
main() {
return 1; }
```

Migration - Using :: in Class Definitions

HP C++

Members of classes are incorrectly allowed to be declared inside the class using the syntax:

```
class_name::member_name
```

HP aC++

This incorrect syntax is considered an error.

Change Needed

Remove the class_name:: specification from the member definition.

Example

Compiling the following code on HP C++ gives no warning nor 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 would fix the problem and
                // run successfully on both compilers.
>
int main(){
}
```

Migration - Duplicate Formal Argument Names

HP C++

Duplicate formal argument names are allowed.

HP aC++

Duplicate formal argument names generate an error.

Change Needed

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);
```

Migration - Ambiguous Function/Object Declaration

HP C++

An ambiguous function/object declaration compiles without warning, assuming an object declaration.

HP aC++

An ambiguous function/object declaration generates an error.

Change Needed

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 above code is whether b is being 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++ causes an error like the following:

```
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 for successful compilation 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
}
```

Migration - Overloaded Operations ++ and - -

Overloaded operations ++ and -- must be correctly used. These operations require a member function with one argument. If the function has no argument:

HP C++

A warning is issued and a postfix operation is assumed.

HP aC++

The inconsistency between the overloaded function usage and definition is considered an error.

Change Needed

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++ causes a warning like the following:

```
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 either 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
};
```

Migration - Reference Initialization

Illegal reference initialization is no longer allowed.

HP C++

A warning is generated stating that the initializer for a non-constant reference is not an lvalue (anachronism).

HP aC++

An illegal initialization of a reference type causes an error and the program does not compile.

Change Needed

Use a constant reference.

Example

```
void f() {
   char c = 1;
   int & r = c;
}
```

Compiling the above code with HP C++ causes a warning like the following:

```
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 &'.
```

For successful compilation with both compilers, change the code as shown below:

```
void f() {
    char c = 1;
    const int & r = c;
}
```

Migration - Using `operator new` to Allocate Arrays

HP C++

operator new is called to allocate memory for an array.

HP aC++

operator new [] is called to allocate memory for an array.

Change Needed

Change operator new to operator new [ ].

Example

```
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];
}
```

The above code compiles without error on HP C++. On HP aC++, an error like the following is generated:

```
Error: File "DDB4269.C", Line 10
Expected 1 argument(s) for void *operator new [ ](unsigned int); had 3 instead.
```

For More Information

- Overloading new[ ] and delete[ ] for Arrays

Migration - Parentheses in Static Member Initialization List

HP C++

Redundant parentheses are allowed in a static member initialization list.

HP aC++

Redundant parentheses in a static member initialization list cause an error and the program does not compile.

Change Needed

Remove the redundant parentheses. The resulting code compiles correctly with either HP C++ or HP aC++.

Example

```
class A {
public:
   int i;
   static int (A::*p);
};

int (A::*(A::p)) = &(A::i);
```

Compiling the above code with HP aC++ causes an error like the following:

```
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 form the last line, as in the following example:

```
class A {
public:
   int i;
   static int (A::*p);
};
```

```
int (A::*(A::p)) = &A::i;
```

Migration - &qualified-id Required in Static Member Initialization List

HP C++

An unqualified function name in a static member initialization list is allowed.

HP aC++

An unqualified function name in a static member initialization list causes an error and the program does not compile.

Change Needed

Use the unary operator & followed by a qualified-id in the member initialization list. The resulting code compiles correctly with either HP C++ or HP aC++.

Example

```
class A {
public:
   int i;
   int j();
static int (A::*p)();
};
int (A::*(A::p))() = j;
```

Compiling the above code with HP aC++ causes the following error:

```
Error: File "DDB4270A.C", Line 7
Cannot initialize 'int (A::*)()' with 'int (*)()'.
```

To successfully compile with either 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;
```

Migration - Non-constant Reference Initialization

HP C++

If you do not initialize a non-constant reference with an lvalue, an anachronistic warning is issued and compilation continues.

HP aC++

An error is issued if you do not use an lvalue for a non-constant reference initialization.

Change Needed

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 the above code with HP C++ generates a warning like the following:

```
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 an error like the following:

```
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;
}

void f(int &);
int i;
int main () {
   i=3;
   f(i);              // Use an lvalue for reference initialization.
   return 0;
}
```

Migration - Digraph White Space Separators

HP C++

Alternative tokens (digraphs) are not supported.

HP aC++

Digraphs are supported and legal C++ syntax can be considered an error because of digraph substitution.

Change Needed

Insert a blank between the 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 :, as follows:

```
C< ::A> a;
```

Migration Considerations when Using Templates

In HP aC++, templates are processed differently than in HP C++ (cfront). There is no repository; instead, instantiations are placed in an object (.o) file (with additional information in a .I file if you specify the +inst_auto command-line option). You cannot modify these files as was possible with the files in a repository. Template command-line options are completely different. For information about HP aC++ templates, refer to Using Templates.

To begin migrating code containing templates to HP aC++, try to compile and link using compile-time instantiation (the default). If this fails with compilation errors, you can compile using one of the following:

- the +inst_all option to see all compile-time errors, including template instantiation errors. Note, this may generate errors that won't occur in your program, because the draft standard allows template parameters that can't instantiate all members. +inst_all forces instantiation of such members.
- the +inst_none option to mask compile-time template instantiation errors.

To reset after all translation units compile successfully:

- Remove any .o and .I files. Using a clobber makefile target to remove .I files is similar to removing the ptrepository directory in cfront.
- Recompile and link using compile-time instantiation.

Verbose Template Processing Information

Use the +inst_v option to replace the cfront -ptv option for verbose template processing information.

For More Information

- The cfront Implicit Include Convention
- Converting Directed Mode to Explicit Instantiation
- Obsolete Template Options

Common Template Migration Syntax Changes

Template code in HP aC++ needs to use the new keyword typname to distinguish types. Also, data members may need to be referenced using the this-> notation.

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.

With the HP C++ (cfront) compiler, for any .h file containing template declarations,

there is a .c file containing 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 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 are using 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, as in the following example.

Note that explicit instantiation can be used to instantiate a template class and all of its member functions, an individual template function, or a template class's member function.

```
#!/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*.$$
```

Migration - Translator Mode is not Supported

HP aC++ does not support a C++ to C translator mode. For a list of cfront translator

mode options that are obsolete, refer to Obsolete Translator Mode Options.

Distributing your C++ Products

If you write code in HP aC++ and distribute any of the following C++ files to your customers, read all of the following sections for recommendations and legal requirements.

- shared libraries containing C++ code
- 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++.

Strong Recommendations

We strongly recommend that you 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.

Be sure your customer has read this distribution information.

NOTE: If you choose to distribute archive libraries or object files, your customer must have purchased HP aC++.

Applications that use HP aC++ Shared Libraries

This section explains what you need to do to ensure that your customers can use your code correctly.

If your application uses any of the shared libraries that come with HP aC++ your customer must have those libraries installed on their system to run the application. If your customer already has the necessary HP aC++ shared libraries installed, the application will work.

The following HP aC++ run-time libraries are provided as a patch. Note, these libraries are not part of the HP-UX 10.x core system. If you search for the patch on the patch machine, look for the patch name "HP aC++ runtime libraries." For more information about the patch, refer to the *HP aC++ Release Notes* .

- /usr/lib/libCsup.sl
- /usr/lib/libstd.sl
- /usr/lib/librwtool.sl
- /usr/lib/libstream.sl

CAUTION: If you distribute either executable files or shared libraries as part of your product, you should not ship the above HP aC++ run-time libraries with your product in such a way that it results in overwriting a newer library version with an older, incompatible version. If you ship any HP aC++ run-time library, then it is your responsibility to ensure that an old library version is not installed over a new one.

Refer also to the **CAUTION** in the section Installing your Application.

Linking Your HP aC++ Libraries with Other Languages

This section discusses what you and your customers need to do if your product is an HP aC++ library to be called with another language.

The C++ language requires that nonlocal static objects be initialized before any function or object is used. HP aC++ initializes nonlocal static objects in all object files, including shared libraries, before the first statement in main() executes. If you distribute HP aC++ libraries that your customers will use, they must do the following to ensure that nonlocal static objects are correctly initialized and destroyed:

- Your customer must have purchased HP aC++ and must link their code with the aCC command.
- If your customer's main program is written in a language other than C++, your customer's main program must first call _main() before doing anything else. _main() calls all nonlocal static constructors.
- HP aC++ runtime libraries and /usr/lib/dld.sl patches must be installed by your customer.

If your libraries are C++ shared libraries, the above restrictions can be relaxed as follows:

- At least the A.01.07 HP aC++ runtime patch is needed along the with either the /usr/lib/aCC/dld.sl patch or the /usr/lib/dld.sl patch.
- _main() must still be called, before any use of aC++ code. It could be placed in the C++ library itself.
- The a.out must be linked with:

  /usr/lib/libcl.sl (or on the link line -lcl)

- Except when dynamically loading the C++ shared library, the a.out must be linked with the HP aC++ runtime libraries in the following order:

  -lstd -lstream -lCsup -lm -lcl -ldld

- If the library does not use libstd (STL) or libstream (iostreams), then they can be eliminated. If tools.h++ is used, then add -lrwtool to the left.
- The following stub file needs to be linked into the shared library or every a.out. Assemble as follows:

  as cpprt0_stub.s

  or

  cc -c cpprt0_stub.s

  A copy of the stub file cpprt0_stub.s can be shipped.

```
================ cpprt0_stub.s ===================
        .code
; stubs for static constructors in a.out
        .export __StaticCtorTable_Start,data
        .export __StaticCtorTable_End,data
__StaticCtorTable_Start
__StaticCtorTable_End

        .data
; stubs for static constructors in a.out, compiled with +z/+Z
        .export __ZStaticCtorTable_Start,data
        .export __ZStaticCtorTable_End,data
__ZStaticCtorTable_Start
__ZStaticCtorTable_End
```

In addition, your customers should review Mixing C++ with other Languages for information on linking HP aC++ modules with HP C, HP Pascal, and HP FORTRAN 77.

NOTE: HP aC++ code cannot be mixed with HP C++ code.

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 patch.

Your application's installation procedure should install the appropriate HP aC++ components in the standard places on your customer's systems. This will ensure that the aCC command can find them.

CAUTION: If your customer already has HP aC++ installed and their version is newer than yours, you should never overwrite any of the existing HP aC++ components. In addition, you should not install your product on a system that has a newer version of HP aC++ if that newer version is incompatible with your version.

You should also warn your customers not to install a version of HP aC++ after installing your product if their version of HP aC++ is incompatible with your version.

HP aC++ Files You May Distribute

For this release , Hewlett-Packard grants you permission to package and redistribute the following subset of HP aC++ components to your customers. The following HP aC++ run-time libraries are provided as a patch. Note, these libraries are not part of the HP-UX 10.x core system. If you search for the patch on the patch machine, look for the patch name "HP aC++ runtime libraries." For more information about the patch, refer to the *HP aC++ Release Notes* .

- /usr/lib/libCsup.sl
- /usr/lib/libstd.sl
- /usr/lib/librwtool.sl
- /usr/lib/libstream.sl

Refer to the **CAUTION** in the prior section Applications that use HP aC++ Shared

Libraries.

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:

1. These HP aC++ components cannot be redistributed as part of a C++ compiler, linker, or interpreter product.
2. All copyright notices in the code must be retained.
3. The HP aC++ executable components can only be redistributed by HP aC++ customers.

- Return to Distributing your C++ Products

Exception Handling

Exception handling provides a standard mechanism for coding responses to run-time errors or **exceptions**.

Exception Handling is the Default

Exception handling is on by default. To turn it off, you *must* use the +noeh option.

CAUTION: 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 (note that HP aC++ run-time 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
- run-time 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 error 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 runtime_error class defines errors due to events beyond the scope of the program.
- The logic_error class defines errors in the internal logic of the program.

For More Information

- Standard Exception Classes
- Standard Exceptions

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.
- 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"));
}
void main ( )
{
    try {
        fx ();
    }
    catch (runtime_error& r) {
        // handle any kind of error, including range_error
        cout <<r.what() << '\n';
    }
}
```

Debugging Exception Handling

The HP WDB Debugger and the HP/DDE Debugger support C++ exception handling. For more information see the following:

- HP WDB Debugger Documentation
- HP/DDE Debugger Documentation

Performance Considerations when using Exception Handling

HP aC++ exception handling has no significant performance impact at either compile-time nor run-time, if you are not using optimization. If you are using optimization, be aware of the following run-time performance implications:

- Optimization of an automatic object with a destructor is inhibited, since no part of the object can be in a register when a function which may throw an exception is called. This is because run-time exception handling uses an object's assigned address if it decides to "clean it up" (run its destructor).
- Optimization in try blocks is inhibited because there are implicit branches from each call in a try block to each catch clause.

Getting Started with HP aC++

Choose from the following for introductory information:

- Major Components of the Compiling System
- Using the aCC Command
- Compiling and Executing a Simple Program
- Debugging Programs
- Accessing the Online Example Source Files

Major Components of the Compiling System

HP aC++ includes the following pieces:

- aCC -- the driver
- ctcom -- compiles C++ source statements
- assigner -- uses an automatic instantiation algorithm for template processing (The assigner is not used by the default template instantiation mechanism. It is invoked only when you specify +inst_auto or +inst_close on the command-line.).

For More Information

- HP aC++ Executables
- Run-time Libraries and Header Files

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.

For More Information:

- For more details about the aCC command, see Compiler Command Syntax.

Compiling a Simple Program

The best way to get started with HP aC++ is to write, compile, and execute a simple program, like the following one:

```
#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:

```
$ 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 named getting_started.C:

```
$ a.out
Enter an integer: 7

14 is twice 7.
```

Debugging Programs

You can use the HP WDB Debugger or the HP/DDE Debugger to debug your C++ programs.

To do so, first compile your program with either the -g, the -g0, or the -g1 option.

Example

The -g0 option to aCC enables generation of debug information:

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

The gdb command runs the HP WDB Debugger in terminal interface mode:

```
gdb -tui -xdb a.out

The dde command runs the HP/DDE Debugger:

dde a.out
```

HP Specific Features of `lex` and `yacc`

Following is a list of HP specific features of **`lex`** and **`yacc`**. For more information on these tools, see the lex and yacc man pages or the *HP-UX Reference* . Another general source of information is *lex and yacc* by John R. Levine, Tony Mason, and Doug Brown.

- 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.

Notes on Using `lex` and `yacc`

When using lex and yacc, please note the following:

- 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>
  ```

Creating and Using Libraries

Choose from the following topics for information about the libraries provided with HP aC++ as well as how you can create and use your own libraries.

For additional background, refer to the *HP-UX Linker and Libraries Online User Guide* which is frequently referenced in these sections.

- HP aC++ Libraries
- Creating and Using Shared Libraries
- Advanced Shared Library Features

- Using Standard HP-UX Libraries and Header Files

Migration

- Migration Considerations when Using Libraries

See Also

- Closing a Library with the +inst_close Option (for libraries that contain templates)
- Mixing C++ with Other Languages

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++ Run-time Support Library

HP C++ (cfront) Compatibility Libraries

NOTE: HP aC++ provides the following libraries whose functionality is also provided with HP C++ (cfront). These libraries are not standards based.

- IOStream Library
- Standard Components Library

See Also:

- HP aC++ File Locations
- Linking to C++ Libraries
- Migration Considerations when Using Libraries

Standard C++ Library

The International Standards Organization (ISO) and the American National Standards Institute (ANSI) have completed the process of standardizing the C++ programming language. A major result of this standardization process is the Standard C++ Library, a large and comprehensive collection of classes and functions. Choose from the following for more details.

- Introductory Concepts
- Details
- Standard C++ Library Documentation and Example Code
- Incompatibilities Between the Library and the Standard
- Run-time Libraries and Header Files

Introductory Concepts

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.

  For an introduction, see the following topics:

  - How the Standard C++ Library Differs from Other Libraries
  - The Non-Object-Oriented Design of the Standard C++ Library

- A templatized string class
- A templatized class for representing complex numbers
- A uniform framework for describing the execution environment, through the use of a template class named numeric_limits and specializations for each fundamental data type
- Memory management features
- Language support features
- Exception handling features

For More Information

- Introduction to Using the Standard C++ Library

How the Standard C++ Library Differs from Other Libraries

A major portion of the Standard C++ Library is comprised of 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 was derived from the Standard Template Library or 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. Through the proper use of encapsulation, even exceedingly complex software systems can be divided into manageable units and assigned to various members of a team of programmers for development.

Inheritance is a powerful technique for permitting code sharing and software reuse, but it is most 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 will 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 was designed as a collection of algorithms and, separate from these, a collection of data structures that could be 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 fifty different algorithms and about a dozen major data structures. This separation has the effect of reducing the size of source code, and decreasing some of the risk that similar activities will have dissimilar interfaces. Were it not for this separation, for example, each of the algorithms would have to be re-implemented in each of the different data structures, requiring several hundred more member functions than are found in the present scheme.

Flexibility

One advantage of the separation of algorithms from data structures is that such algorithms can be used with conventional C++ pointers and arrays. Because C++ arrays are not objects, algorithms encapsulated within a class hierarchy seldom have this ability.

Efficiency

The STL in particular, and the Standard C++ Library in general, provide a low-level approach to developing C++ applications. This low-level approach can be useful when specific programs require an emphasis on efficient coding and speed of execution.

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 library's architecture, it is not possible to verify that these iterator elements are matched, that is, that they are derived from the same container. Using (either intentionally or by accident) a beginning iterator from one container with an ending iterator from another is a recipe for certain disaster. It is very important to know that iterators can become invalidated as a result of a subsequent insertion or deletion from the underlying container class. This invalidation is not checked, and use of an invalid iterator can produce unexpected results. Familiarity with the Standard C++ Library will help reduce the number of errors related to iterators.

Templates: Errors and "Code Bloat"

The flexibility and power of templatized algorithms is, with most compilers, purchased at a loss of precision in diagnostics. Errors in the parameter lists to generic algorithms will sometimes be manifest only as obscure compiler errors for internal functions that are defined many levels deep in template expansions. Again, familiarity with the algorithms and their requirements is a key to successful use of the standard library. Heavy reliance on templates can cause programs to grow larger than expected. You can minimize this problem by learning to recognize the cost of instantiating a particular template class, and by making appropriate design decisions. Be aware that as compilers become more and more fluent in templates, this will become less of a problem.

Multithreading Problems

The Standard C++ Library must be used carefully in a multithreaded environment. Iterators, because they exist independently of the containers they operate on, cannot be safely passed between threads. Since iterators can be used to modify a non const container, there is no way to protect such a container if it spawns iterators in multiple threads. Use thread-safe wrappers, such as those provided by Tools.h++ Library, if you need to access a container from multiple threads.

Introduction to Using the Standard C++ Library

Within a few years the Standard C++ Library will be the standard set of classes and libraries delivered with all ANSI-conforming C++ compilers. Although the design of a

large portion of the Standard C++ Library is in many ways not object-oriented, C++ itself excels as a language for manipulating objects. How do you integrate the library's non-object-oriented architecture with the language's strengths for manipulating objects?

The key is to use the right tool for each task. Object-oriented design methods and programming techniques are almost without peer as guideposts in the development of large complex software. For the large majority of programming tasks, object-oriented techniques will remain the preferred approach. Products such as Rogue Wave's Tools.h++ Library which encapsulates the Standard C++ Library with a familiar object-oriented interface, can provide you with the power and the advantages of object-orientation.

Use Standard C++ Library components directly when you need flexibility or highly efficient code. Use the more traditional approaches to object-oriented design, such as encapsulation and inheritance, when you need to model larger problem domains and knit all the pieces into a full solution. When you need to devise an architecture for your application, always consider the use of encapsulation and inheritance to compartmentalize the problem. But if you discover that you need an efficient data structure or algorithm for a compact problem (the kind of problem that often resolves to a single class), look to the Standard C++ Library. The library excels in the creation of reusable classes, where low-level constructs are needed, while traditional OOP techniques really shine when those classes are combined to solve a larger problem.

In the future, most libraries will use the Standard C++ Library as their foundation. By using the Standard C++ Library, either directly or through an encapsulation such as Tools.h++ Library, you help insure interoperability. This is especially important in large projects that may rely on communication between several libraries. A good rule of thumb is to use the highest encapsulation level available to you, but make sure that the Standard C++ Library is available as the base for interlibrary communication and operation.

The C++ language supports a wide range of programming approaches because the problems we need to solve require that range. The language, and now 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 more traditional OOP techniques, puts a very flexible tool into the hands of anyone building a collection of C++ classes, whether those classes are intended to stand alone as a library or are tailored to a specific task.

Standard C++ Library Reference

Select from the following categories to see further explanation from the *Rogue Wave Software Standard C++ Library Class Reference* . Corresonding header file(s) are noted at the beginning of each category. Note that system man pages are also available.

NOTE: If you are accessing this guide from the World Wide Web URL, http://docs.hp.com, rather than from a system on which HP aC++ is installed, Rogue Wave documentation is not available. The following links will not succeed.

- Algorithms
- Allocators
- Complex Number Library
- Containers
- Exception Class
- Function Adaptors
- Function Objects
- Generalized Numeric Operations
- Insert Iterators
- Iterator Operations
- Iterators
- Memory Handling Primitives
- Memory Management
- Numeric Limits Library

- Stream Iterators
- String Library
- Utility Classes
- Utility Operators

Algorithms

Generic Algorithms-- for performing various operations on containers and sequences:

```
#include <algorithm>
```

- adjacent_find-- find the first adjacent pair of elements in a sequence that are equivalent
- binary_search-- algorithm to perform a binary search on ordered containers
- copy-- algorithm to copy values from one specified range to another; use to copy values from one container to another, or to copy values from one location in a container to another location in the same container
- copy_backward-- algorithm to copy elements in one specified range to another, starting from the end of the sequence and progressing to the front
- count-- count the number of elements in a container that satisfy a given value
- count_if-- count the number of elements in a container that satisfy a given predicate
- equal-- compares two ranges for equality
- equal_range-- find the largest subrange in a collection into which a given value can be inserted without violating the ordering of the collection
- fill-- initialize a range with a given value
- fill_n-- assign a value to the elements in a sequence
- find-- find an occurence of value in a sequence
- find_end-- find a subsequence of equal values in a sequence
- find_first_of-- find the first occurrence of any value from one sequence in another sequence
- find_if-- in a sequence, find an occurrence of a value that satisfies a specifed predicate
- for_each-- apply a function to each element in a range
- generate-- initialize a container with values produced by a value-generator class
- generate_n-- initialize a container with values produced by a value-generator class
- includes-- compare two sorted sequences and returns true if every element in one range is contained in the other
- inplace_merge -- merge two sorted sequences into one
- iter_swap-- exchange values pointed at in two locations
- lexicographical_compare-- compares two ranges lexicographically
- lower_bound-- determine the first valid position for an element in a sorted container
- make_heap-- creates a heap
- max-- find and return the maximum of a pair of values
- max_element-- find the maximum value in a range
- merge-- merge two sorted sequences into a third sequence
- min-- find and return the minimum of a pair of values
- min_element-- find the minimum value in a range
- mismatch-- compare elements from two sequences and return the first two elements that don't match
- next_permutation-- generate successive permutations of a sequence based on an ordering function
- nth_element-- rearrange a collection so that all elements lower in sorted order than the *n* th element come before it and all elements higher in sorter order than the *n* th element come after it
- partial_sort-- templated algorithm for sorting collections of entities
- partial_sort_copy-- templated algorithm for sorting collections of entities
- partition-- place all of the entities that satisfy the given predicate before all of the entities that do not
- pop_heap-- move the largest element off the heap

- prev_permutation-- generate successive permutations of a sequence based on an ordering function
- push_heap-- place a new element into the heap
- random_shuffle -- randomly shuffle elements of a collection
- remove-- move desired elements to the front of a container, and return an iterator that describes where the sequence of desired elements ends
- remove_copy-- similar to remove
- remove_copy_if-- similar to remove
- remove_if-- similar to remove
- replace-- substitute elements stored in a collection with new values
- replace_copy-- similar to replace
- replace_copy_if-- similar to replace
- replace_if-- similar to replace
- reverse-- reverse the order of elements in a collection
- reverse_copy-- reverse the order of elements in a collection while copying them to a new collecton
- rotate-- left rotates the order of items in a collection, placing the first item at the end, second item first, etc., until the item pointed to by a specified iterator is the first item in the collection
- rotate_copy-- rotate elements as in rotate, but instead of swapping elements within the same sequence, copies the result of the rotation to a container specified by result
- search-- find a subsequence within a sequence of values that is element-wise equal to the values in an indicated range
- search_n-- similar to search but searches for a given number of occurrences
- set_difference-- construct a sorted difference that includes copies of the elements present in onw range but not in another, return the end of the constructed range
- set_intersection-- construct a sorted intersection of elements from two ranges, return the end of the constructed range
- set_symmetric_difference -- construct a sorted symmetric difference of the elements from two ranges, include copies of the elements present in only one of the ranges
- set_union-- construct a sorted union of the elements from two ranges, return the end of the constructed range
- sort-- templated algorithm for sorting collections of entities
- sort_heap-- convert a heap into a sorted collection
- stable_partition-- place all entities that satisfy the given predicate before all entities that do not, while maintaining the relative order of elements in each group
- stable_sort-- templated algorithm for sorting collections of entities
- swap-- exchange values
- swap_ranges-- exchange a range of values in one location with those in another
- transform-- apply an operation to a range of values in a collection and stores the result
- unique-- remove consecutive duplicates from a range of values and place the resulting unique values into the result, overwriting the existin elements
- unique_copy-- remove consecutive duplicates from a range of values and place the resulting unique values into the resulting OutputIterator
- upper_bound-- determines the last valid position for a value in a sorted container

Allocators

The Standard C++ Library allocator interface encapsulates the types and functions needed to manage the storage of data in a generic way. The interface wraps the mechanism for managing data storage and separates this mechanism from the classes and functions used to maintian associations between data elements. This eliminates the need to rewrite containers and algorithms to suit different storage mechanisms. You can encapsulate all the storage mechanism details in an allocator, then provide that allocator to an existing container when appropriate.

The Standard C++ Library provides a default allocator class that implements this interface using the standard new and delete operators for all storage management.

Complex Number Library

Operations used to create and manipulate complex numbers.

```
#include <complex>
```

- complex-- a template class used to create objects for representing and manipulating complex numbers

Containers

Collection classes are often described as Containers. A container stores a collection of other objects and provides certain basic functionality that supports the use of generic algorithms. Containers come in two basic flavors: sequences and associative_containers. They are further distinguished by the type of iterator they support.

```
#include <bitset>
#include <deque>
#include <ul>
#include <map> for map and multimap
#include <queue> for queue and priority_queue
#include <set> for set and multiset
#include <stack>
#include <vector>
```

- bitset-- random access to a set of binary values; operations can be performed using logical bit-wise operators, no iterators for accessing elements
- deque -- random access, insertion at front or back
- list-- insertion and removal throughout
- map -- access to values via keys, insertion and removal
- multimap-- map permitting duplicate keys
- multiset-- set with repeated copies
- priority_queue-- access and removal of largest value
- queue-- insertion at back, removal from front
- set-- elements maintained in order, test for inclusion, insertion, and removal
- stack-- insertion and removal only from top
- vector-- random access to elements, insertions at end

Exception Class

```
#include <exception>
```

- exception-- base class to support logic and runtime errors

Function Adaptors

Used to build new function objects out of existing function objects.

```
#include <functional>
```

- not1-- use to reverse the sense of a unary predicate function object
- not2-- use to reverse the sense of a binary predicate function object
- ptr_fun-- adapt a pointer to a function to work where a function is called for

Function Objects

Objects with an operator() defined. Function objects are used in place of pointers to functions as arguments to templated algorithms.

#include <functional>

- binary_function-- base class for creating binary function objects
- binary_negate-- returns the complement of the result of its binary predicate
- bind1st and binder1st-- templatized utilities to bind a value to the first argument of a binary function object
- bind2nd and binder2nd-- templatized utilities to bind a value to the second argument of a binary function object
- divides-- returns the result of dividing its first argument by its second
- equal_to-- returns true if its first argument equals its second
- greater-- returns true if its first argument is greater than its second
- greater_equal-- returns true if its first argument is greater than or equal to its second
- less-- returns true if its first argument is less than its second
- less_equal-- returns true if its first argument is less than or equal to its second
- logical_and-- returns true if both of its arguments are true
- logical_not-- returns true if its argument is false
- logical_or-- returns true if either of its arguments is true
- minus-- returns the result of subtracting its second argument from its first
- modulus-- returns the remainder obtained by dividing the first argument by the second argument
- negate-- returns the negation of its argument
- not_equal_to-- returns true if its first argument is not equal to its second
- plus-- returns the result of adding its first and second arguments
- pointer_to_binary_function-- adapts a pointer to a binary function to work where a binary_function function object is called for
- pointer_to_unary_function-- adapts a pointer to a function to work where a unary_function function object is called for
- times-- returns the result of multiplying its first and second arguments
- unary_function-- base class for creating unary function objects
- unary_negate-- function object class that returns the complement of the result of its unary predicate

Generalized Numeric Operations

#include <numeric>

- accumulate-- accumulates all elements within a range into a single value
- adjacent_difference-- outputs a sequence of the differences between each adjacent pair of elements in a range
- inner_product-- computes the inner product A X B of two ranges A and B
- partial_sum-- calculates successive partial sums of a range of values

Insert Iterators

Insert Iterators are adaptors that allow an iterator to insert into a container rather than overwrite elements in the container.

#include <iterator>

- back_insert_iterator-- class to insert items at the end of a collection
- back_inserter-- function to create an instance of a back_insert_iterator for a particular collection type
- front_insert_iterator-- class to insert items at the beginning of a collection
- front_inserter-- function to create an instance of a front_insert_iterator for a particular collection type
- insert_iterator-- class to insert items into a specified location of a collection
- inserter-- function to create an instance of an insert_iterator given a particular collection type and iterator

Iterator Operations

#include <iterator>

- advance-- move an iterator forward or backward (if available) by a specified distance
- distance-- compute the distance between two iterators
- distance_type-- determine the type of distance used by an iterator
- iterator_category-- determine the category to which an iterator belongs
- value_type-- determine the type of value to which an iterator points

Iterators

Iterators are pointer generalizations for traversal and modification of collections.

#include <iterator>

- reverse_bidirectional_iterator-- read and write, forward and backward moving
- reverse_iterator-- read and write, random access

Memory Handling Primitives

#include <memory>

- get_temporary_buffer-- pointer based primitive to reserve the largest possible buffer that is less than or equal to the size requested
- return_temporary_buffer-- pointer based primitive to return to available mamory a buffer previously allocated via get_temporary_buffer

Memory Management

#include <memory>

- auto_ptr -- a simple, smart pointer class
- raw_storage_iterator-- enable iterator-based algorithms to store results into uninitialized memory
- uninitialized_copy-- algorithm that uses the construct primitive to copy values from one range to another location
- uninitialized_fill-- algorithm that uses the construct primitive to set values in a collection
- uninitialized_fill_n-- algorithm that uses the construct primitive to set values in a collection

Numeric Limits Library

#include <limits>

- numeric_limits-- a class for representing information about scalar types

Stream Iterators

#include <iterator>

Stream iterators allow generic algorithms to be used directly on streams.

- istream_iterator-- stream iterator provides iterator capabilities for istreams
- ostream_iterator-- stream iterator to provide iterator capabilities for ostreams and istreams

String Library

#include <string>

- basic_string-- templated class for handling sequences of character-like entities
- string-- a specialization of the basic_string class
- string_char_traits -- a traits class providing types and operations to the basic_string container
- wstring-- a specializatio of the basic_string class

Utility Classes

#include <utility>

- pair-- class that provides a template for encapsulating pairs of values that may be of different types

Utility Operators

#include <utility>

- operator!=
- operator>
- operator<=
- operator>=

Incompatibilities Between the Library and the Standard

As the ANSI/ISO C++ International Standard has evolved over time, the Standard C++ Library has not always kept up. Such is the case for the "times" function object in the functional header file. In the standard, "times" has been renamed to "multiplies."

If you want to use "multiplies" in your code, to be compatible with the ANSI/ISO C++ International Standard, use a conditional compilation flag on the aCC command line.

For example, for the following program, compile with the command line:

```
aCC -D__HPACC_USING_MULTIPLIES_IN_FUNCTIONAL test.c

// test.c
int times;                    //user defined variable
#include <functional>
// multiplies can be used in

int main() {}
// end of test.c
```

Depending on the existence of the conditional compilation flag, functional defines either "times", or "multiplies", not both.

So, if you have old source that uses "times" in header functional and also new source that uses "multiplies", the sources cannot be mixed. Mixing the two sources would constitute 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 now non-standard "times" function object, you do not need to do anything to compile the old source.

Tools.h++ Library

The Tools.h++ Library is a fondation class library built on the Standard C++ Library. Use its object oriented capabilities to simplify coding and facilitate code reuseablility. Choose from the following for more information:

- Introduction to Using the Standard C++ Library
- Tools.h++ Library Documentation and Example Code
- Run-time Libraries and Header Files

HP aC++ Run-time Support Library

The HP aC++ run-time support library is provided as a shared library, /usr/lib/libCsup.sl and as an archive library, /usr/lib/libCsup.a.

The library supports the following functionality:

- Exception Handling
- Memory Management (operators new and delete)
- Start and termination of a C++ program
- Run-time type identification (type_info)
- static object constructors and desctructors

For More Information

- Run-time Libraries and Header Files

IOStream Library

NOTE: At 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.

For More Information

- Migration Considerations when Using Libraries

Standard Components Library

NOTE: The Standard Components library is provided for compatibility with HP C++ (cfront).

The library is not standards based. It is strongly recommended that you use the capabilities of the Standard C++ Library.

For More Information

- Migration Considerations when Using Libraries.

Linking to C++ Libraries

You can compile and link any C++ modules to one or more libraries. HP aC++ automatically

links the following with a C++ executable. Note that when you specify the -b option to create a shared library, these defaults do not apply.

- /usr/lib/libCsup.sl (the HP aC++ run-time support library)
- /usr/lib/libstd.sl (standard C++ library)
- /usr/lib/libstream.sl(iostream library)
- /opt/aCC/lib/cpprt0.o (for an executable)
- /opt/aCC/lib/shlrt0.o (for a shared library)
- /opt/aCC/lib/cxxshl.o (routines used when creating archived executables with the +A option; used instead of libdld.sl)
- /opt/langtools/lib/crt0.o (start-up routines)
- /usr/lib/libc.sl (the HP-UX system library)
- /usr/lib/libdld.sl (routines for managing shared libraries)
- /usr/lib/libcl.sl (routines for exception handling)
- /usr/lib/libm.sl (math library)

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.

NOTE: To maintain compatibility on future releases, archive and shared libraries should not be mixed. Refer to the "Mixing Shared and Archive Libraries" section in the *HP-UX Linker and Libraries Online User Guide* .

Specifying Other Libraries

You can specify other libraries using the -l option. For example, in order to use the Tools.h++ library, specify -lrwtool:

```
aCC myapp.C -lrwtool
```

Creating and Using Shared Libraries

This section provides information about shared libraries that is specific to HP aC++.

Select one of the following topics:

- Compiling for Shared Libraries
- Creating a Shared Library
- Using a Shared Library
- Example of Creating and Using a Shared Library
- Linking Archive or Shared Libraries
- Updating a Shared Library

For More Information

- Refer to Advanced Shared Library Features.
- For additional information about creating and using shared libraries, refer to the *HP-UX Linker and Libraries Online User Guide*
- For information on using the options to the aCC command, select command-line options.

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

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

This example compiles util.C, generates position-independent code, and puts the code into the object file util.o. util.o can later be put into a shared library.

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.

CAUTION: You must use the aCC command to create a C++ shared library. This is because the aCC command ensures that any static constructors and destructors in the shared library are executed at the appropriate times.

Example

```
aCC -b -o util.sl util.o
```

This example links util.o and creates the shared library util.sl.

Using a Shared Library

To use a shared library, you simply include the name of the library on the aCC command line as you would with an archive library, or use the -l option, as with other libraries.

The linker links the shared library to the executable file it creates. Once you create an executable file that uses a shared library, you must not move the shared library or the dynamic loader (dld.sl(5)) will not be able to find it.

CAUTION: You must use the aCC command to link any program that uses a C++ shared library. This is because the aCC command ensures that any static constructors and destructors in the shared library are executed at the appropriate times.

Example

```
aCC prog.C util.sl
```

This example compiles prog.C, links it with the shared library util.sl, and creates the executable file a.out.

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 a shared library named libshape.sl from the object files Strings.o and Arrays.o:

```
aCC -b -o libshape.sl Strings.o Arrays.o
```

The following command compiles a program, draw_shapes.C, that uses the shared library, libshape.sl:

```
aCC draw_shapes.C libshape.sl
```

Linking Archive or Shared Libraries

If both 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: You can use the +A option if you are using only archive libraries to create a completely archived executable.

The -a linker option tells the linker which type of library to use. 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:

The syntax of the -a linker option when used with aCC is:

```
-Wl,-a,{archive
        shared
        default}
```

The different meanings of this option are:

```
-Wl,-a,archive
       Select archive libraries. If the archive library does not exist, the linker
       generates a warning message and does not create the output file.
-Wl,-a,shared
       Select shared libraries. If the shared library does not exist, the linker
       generates a warning message and does not create the output file.
-Wl,-a,default
       Select the shared library if it exists; otherwise, select 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.

If, for example, a module in an existing shared library requires a fix, simply recompile the fixed module with the +z or +Z option, then recreate the shared library with the -b option. Any programs that use this library will now be using the new versions of the routines. That is, you do not have to relink any programs that use this shared library because they are attached at run time.

Advanced Shared Library Features

This section explains additional things you can do with shared libraries.

Select one of the following:

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 an executable's shared libraries do *not* reference the program's main routine, the linker does *not* include the main symbol in the a.out file's export list.

Normally, this is a problem only when a program explicitly calls shared library management routines. (See Routines and Options to Manage C++ Shared Libraries.)

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 the overhead of binding 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, which 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 program's execution. Immediate binding also detects unresolved symbols at startup time, rather than during program execution. Another use of immediate binding is to provide better interactive performance, if you don't mind program startup taking longer. To force immediate binding, use the option -Wl,-B,immediate.

Example

The following example forces immediate binding:

```
aCC -Wl,-B,immediate draw_shapes.o -lshape
```

To specify default binding, use -Wl,B,deferred.

For More Information

For more information, see the *HP-UX Linker and Libraries Online User 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 (see *errno* (2)). Thus, if a program checks for these values, it must include errno.h:

```
#include <errno.h>
```

Linker Options to Manage Shared Libraries

Linker options are available for specifying 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.

For More Information

- For more information about library management routines and linker options, refer to the *HP-UX Linker and Libraries Online User Guide* .
- Man pages for *shl_load* (3X) and *shl_unload* (3X).

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 module's source file. This pragma should only be used if incompatible changes are made to a source file.

For More Information

For more information about version control in shared libraries, refer to the *HP-UX Linker and Libraries Online User Guide* .

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.

For more information refer to the *HP-UX Linker and Libraries Online User Guide* .

Using Standard HP-UX Libraries and Header Files

Several libraries providing system services are included with HP-UX. You can access HP-UX standard libraries by using **header files** that declare interfaces to those libraries. These library routines are documented in the *HP-UX Reference Manual* .

Location of Standard HP-UX Header Files

The standard HP-UX header files are located in /usr/include.

Using Header Files

To use a system library function, your HP aC++ source code must include the preprocessor directive #include. For 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; if none are found, it then searches /usr/include.

You can use header file options to modify the search path.

Example of Using a Standard Header File

If you want to use the getenv function that is in the standard system libraries (/usr/lib/libc.sl and /usr/lib/libc.a), you should specify:

```
#include <stdlib.h>
```

because the external declaration of getenv is found in the header file /usr/include/stdlib.h.

HP aC++ File Locations

HP aC++ Executable Files

- /opt/aCC/bin/aCC -- driver

  the only supported interface to HP aC++ and to the linker for HP aC++ object files

- /opt/aCC/lbin/ctcom -- compiler

  performs source compilation; preprocessing is incorporated into the compiler

- /opt/aCC/lbin/assigner -- assigner

  implements the automatic instantiation algorithm

- /opt/aCC/bin/c++filt -- name demangler

  implements the name demangling algorithm which encodes function name, class name, parameter number and name

- /usr/ccs/bin/ld -- linker

  links executables and builds shared libraries

HP aC++ Run-time Libraries and Header Files

Note that some of the following run-time libraries are provided in both shared and

archive versions.

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

- Standard C++ Library
  - /usr/lib/libstd.sl -- shared version
  - /usr/lib/libstd.a -- archive version
- HP aC++ Run-time Support Library
  - /usr/lib/libCsup.sl -- shared version
  - /usr/lib/libCsup.a -- archive version
- IOStream Library
  - /usr/lib/libstream.sl -- shared version
  - /usr/lib/libstream.a -- archive version
- /opt/aCC/lib/cpprt0.o -- non-shared library initializer
- /opt/aCC/lib/shlrt0.o -- shared library initializer
- /opt/aCC/lib/cxxshl.o -- used with the +A option
- /opt/aCC/lib/lib++.a -- used with Standard Components library
- /opt/aCC/lib/libGA.a -- used with Standard Components library
- /opt/aCC/lib/libGraph.a -- used with Standard Components library

Mixing C++ with Other Languages

This section provides guidelines for linking HP aC++ modules with modules written in HP C, HP Pascal, and HP FORTRAN 90 on HP 9000 Series 700/800 systems.

Select from the following topics:

General Information When Calling Other Languages

A module is a file containing 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 need to 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 between languages.
- You must use the extern "C" linkage specification to declare any modules that are not written in C++; this is true whether or not the module is written in C.

  NOTE: Do not use extern "C" when including standard C header files because these header files already contain extern "C" directives.
- You must use the extern "C" linkage specification to declare any modules that are written in C++ and called from other languages.

  NOTE: HP aC++ classes are not accessible to non-C++ routines.

Data Compatibility between C and C++

Since C++ is for the most part a superset of C, many of the data types are identical. Both languages support the same primitive types of char, short, int, long, float, and double. 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 (that is, having 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

Since C++ is for the most part a superset of C, calling between C and C++ is a normal operation. You should, however, be aware of the following:

- Using the extern "C" Linkage Specification -- You must use the extern "C" linkage specification to declare C functions.
- Differences in Argument Passing Conventions -- Because of function prototypes, C++ has argument-widening rules that are different from C's rules.
- The main() Function -- The overall control of the program should be written in C++.
- HP aC++ Calling HP C: An Example -- An example C++ program that calls a C program.

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 do so, 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.

If you want to call a C function from a C++ program, you must tell the compiler not to

use its usual encoding scheme when you declare the C function. In other words, you must tell the compiler not to generate a new name for the function. If you don't turn off the usual encoding scheme, the function name declared in your C++ program won't match the function name in your C module defining the function. If the names don't 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"

For instance, 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 if you wish to use C library functions in a C++ program.

```
extern "C"
{
 #include "myclibrary.h"
}
```

NOTE: Do not use extern "C" when including standard C header files because 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* .

Differences in Argument Passing Conventions

If your C++ code calls functions written in C, you should make sure that the called C functions do not use function prototypes that suppress argument widening. If they do, your C++ code will be passing "wider" arguments than your C code is expecting.

The main() Function

When mixing C++ modules with C modules, the overall control of the program must be written in C++, with two exceptions. In other words, the main() function should appear in some C++ module, rather than in a C module. The exceptions are C++ programs and

libraries, including HP-supplied libraries, without any global class objects containing constructors or destructors and C++ programs and libraries, including HP-supplied libraries, without static objects.

HP aC++ Calling HP C: An Example

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 Program

Here's 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++

When mixing C++ modules with C modules, usually the overall control of the program must be written in C++. In other words, the main() function must appear in some C++ module, rather than in a C module, and you must link using aCC. The two exceptions to this rule are C++ programs and libraries (including HP-supplied libraries) without any global class objects containing constructors or destructors and C++ programs and libraries (including HP-supplied libraries) without static objects. Since most C++ programs use the HP aC++ run-time libraries, few programs meet these restrictions. Therefore, you can call a C++ module from a C module by following the points below, as well as the points in General Information When Calling Other Languages:

- To prevent a function name from being mangled, the function definition and all declarations used by the C++ code must use extern "C".
- The C programmer must generate a call to function _main as the first executable statement in main(). Object libraries require this as _main calls the static constructors to initialize the libraries' static data items.
- Member functions of classes in C++ are not callable from C. If a member function routine is needed, a non-member function in C++ can be called from C which 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 should be done in the C++ module.

The following example programs illustrate some of the above points, as well as reference parameters in the interface routine to the constructor.

HP C Calling HP aC++: An Example

```
//************************************************
//  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 the 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.

     _main();

     /* 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 Example Programs

To compile the example, enter the following commands:

```
cc -c cfilename.c
aCC -c C++filename .C
aCC -o executable cfilename .o C++filename .o
```

CAUTION: 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 Pascal and HP FORTRAN 90 from HP aC++

This section covers the following topics:

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 mixing C++ modules with modules in HP Pascal and HP FORTRAN 90, the overall control of the program must be written in C++. In other words, the main() function must appear in some C++ module and no other outer block should be present.

If you wish to have a main() function in a module other than a C++ module, you can add a call to _main() as the first non-declarative statement in the module. However, if you use this method, your code is not portable.

Function Naming Conventions

When calling an HP Pascal or HP FORTRAN 90 function from HP aC++ you must keep in mind the differences between the way the languages handle case sensitivity. HP FORTRAN 90 and HP Pascal are not case sensitive, while HP aC++ is case sensitive. Therefore, all C++ global names accessed by FORTRAN 90 or Pascal routines must be lowercase. All FORTRAN 90 and Pascal 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 Pascal or HP FORTRAN 90 functions from HP aC++, you need to 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 Pascal or HP FORTRAN 90, you must know the calling convention for the order of arguments.

It is not recommended that you pass structures or classes to HP FORTRAN 90 or HP Pascal. For maximum compatibility and portability, only simple data types should be passed 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, &.

HP Pascal passes arguments by value, unless specified as var parameters. There are two ways to pass variables to Pascal var parameters. One way is to use the address operator, &. The other way is to declare the variable as a pointer to the appropriate type, assign the address to the pointer, and pass the pointer.

So, 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
}
```

Refer to "References" in *The C++ Programming Language, Third Edition* for details about using reference variables.

Using extern "C" Linkage

If you want to mix C++ modules with HP FORTRAN 90 or HP Pascal modules, be sure to use extern "C" linkage to declare any C++ functions that are called from a non-C++ module and to declare the FORTRAN or Pascal 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. See the *HP FORTRAN/9000 Programmer's Reference* and the *HP FORTRAN/9000 Programmer's Guide* for information about the +800 option.

HP Pascal strings and HP aC++ strings are not compatible. See your HP Pascal manual for details.

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. For HP Pascal, the lower bound may be any user-defined scalar value.

Definition of TRUE and FALSE

HP aC++ does not have a Pascal boolean type. On the HP 9000 Series 700/800, HP Pascal allocates 1 byte for boolean variables and only accesses the rightmost bit to determine its value, 1 to represent TRUE and 0 for FALSE.

On the HP 9000 Series 300/400, 2 bytes are allocated for a boolean and any nonzero value represents TRUE and 0 represents FALSE. On the HP 9000 Series 300/400, HP aC++ and HP Pascal do share a common definition of TRUE and FALSE.

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, nor can a C++ file pointer 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 opened 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 details.

Files in Pascal

A C++ file pointer cannot be passed to a Pascal routine for performing input/output. A Pascal file variable cannot be used by a C++ program. 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 opened within the latter program.

If I/O from Pascal is required, it is recommended that you use HP-UX input/output routines and intrinsics. This allows C++ and Pascal to use the same I/O mechanism.

See the HP Pascal manual for your system for more details.

Linking HP FORTRAN 90 Routines

When calling HP FORTRAN 90 routines on the HP 9000 Series 700/800, you must include the appropriate run-time libraries by adding the following argument to the aCC command when linking your program:

```
-lisamstub
```

Linking HP Pascal Routines

When calling HP Pascal routines, no additional arguments to the CC command are needed when linking your program. Simply use the aCC command and include your Pascal object files.

Optimizing HP aC++ Programs

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:

Pragmas That Control Optimization

Compiler options provide a high-level, global approach to optimization. To give you more refinement in optimization, HP aC++ provides two pragmas: OPTIMIZE and OPT_LEVEL.

These pragmas must appear outside any function and they apply for the remainder of the file or until superseded by another pragma. For these pragmas to work, the source program *must* be compiled with one of the optimization options. Otherwise the pragmas are ignored.

Pragma OPTIMIZE

The OPTIMIZE pragma turns on or off optimization. It is useful for turning off optimization in sections of a source program.

Syntax of Pragma OPTIMIZE

To turn off optimization for a particular function, put #pragma OPTIMIZE OFF immediately before the function and #pragma OPTIMIZE ON immediately after the function. Then compile the function with one of the aCC command line options that enables optimization.

Example:

```
#pragma OPTIMIZE OFF
void g()     // Turn optimization off.
{
    ...
}
#pragma OPTIMIZE ON
void f()     // Restore optimization level.
{
    ...
}
```

This example, when compiled with -O, turns off optimization for function g() and restores it to level 2 for f().

Pragma OPT_LEVEL

The OPT_LEVEL pragma directs the compiler to change the current optimization level to level 1, 2, 3, or 4. It is useful for switching from one level to another within a source program.

You cannot use this pragma to raise the optimization level beyond the original level set by the option you used on the aCC command line. The compiler issues a warning if you attempt to raise the original optimization level. OPT_LEVEL 3 and 4 are only allowed at the beginning of a file.

Syntax of Pragma OPT_LEVEL

To change optimization levels for a particular function, put #pragma OPT_LEVEL *n* immediately before the function, where *n* is the level of optimization you want for the function.

Examples:

```
#pragma OPT_LEVEL 1
void m()
{
   ...
}
#pragma OPT_LEVEL 2
void n()
{
   ...
}
```

This example, when compiled with -O, lowers the optimization level to level 1 for function m() and restores it to level 2 for n().

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 2.

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 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 optimizations, plus 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:

- 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 run-time 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. Programs compiled at level 2 can be used with the HP Distributed Debugging Environment (DDE) debugger. Use the debugger option -g0 or -g1.

Level 3 Optimization

Level 3 optimization includes level 2 optimizations, plus 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 run-time 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, plus 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 run-time 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 +Oaggressive option as follows:

```
aCC +O2 +Oaggressive sourcefile.C
```

or:

```
aCC +O3 +Oaggressive sourcefile.C

or:

aCC +O4 +Oaggressive sourcefile.C
```

This option enables *additional* optimizations at each level.

CAUTION: 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 +Oconservative option, as follows:

```
aCC +O2 +Oconservative sourcefile.C

or:

aCC +O3 +Oconservative sourcefile.C

or:

aCC +O4 +Oconservative 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.

Specifying Maximum Optimization

For maximum optimization, use the +Oall option as follows:

aCC +Oall *sourcefile.C*

This combination performs aggressive optimizations with unrestricted compile time at the highest level of optimization.

CAUTION: Use +Oall with stable, well-structured code. These types of optimizations give you the fastest code, but are *riskier* than the default optimizations.

The +Oall option combines the +O4, +Oaggressive, and +Onolimit options.

Combining Optimization Options

Optimization options that affect code size, (+Osize), compile-time (+Olimit), and the aggressiveness of the optimizations performed (+Oaggressive or +Oconservative) can be combined at any of the optimization levels 2 through 4.

You can use +Olimit or +Osize with either +Oaggressive or +Oconservative, but you cannot use +Oaggressive with +Oconservative.

Example:

For example, to specify conservative optimizations at level 2 and disable code-expanding optimizations, use:

aCC +O2 +Oconservative +Osize *sourcefile.C*

Profile-Based Optimization

Profile-based optimization (PBO) is a set of performance-improving code transformations based on the run-time characteristics of your application.

There are three steps involved in performing this optimization:

1. Instrumentation - Use +I with any level of optimization to insert data collection code into the object program:

```
aCC +I -O -c sample.C
aCC +I -O -o sample.exe sample.o
```

2. Data Collection - Run the program with representative data to collect execution profile statistics:

   ```
   sample.exe < input.file1
   ```

3. Optimization - Use +P to generate optimized code based on the profile data:

   ```
   aCC +P -o sample.exe sample.o
   ```

Compile times will be fast and link times will be slow when using PBO because code generation happens at link time.

Notes on Using Profile-Based Optimization

When using profile-based optimization, please note the following:

- Because the linker performs code generation for profile-based optimization, linking object files compiled with +I and +P takes more time than linking ordinary object files. However, compile-times will be relatively fast. This is because the compiler is only generating the intermediate code.
- You can compile and instrument in one step, but you will have to recompile again when optimizing. You must use the same options on both compiles, otherwise profile-based optimization cannot be done. For example:

  ```
  aCC +I -O sample.C -o sample.exe  // Compile to instrumented executable.
  sample.exe < input.file1              // Collect execution profile data.
  aCC +P -O sample.C -o sample.exe  // Recompile with optimization.
  ```

- Numerical applications which 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, re-profile and re-optimize your application after making source code changes.

For More Information:

For more information on profile-based optimization, you can refer to the *HP-UX Linker and Libraries Online User Guide* .

Instrumenting the Code

To instrument your program, use the +I option as follows:

```
aCC +I -O -c sample.C              Compile for instrumentation.
aCC +I -O -o sample.exe sample.o   Link to make instrumented executable.
```

The first command line uses the -O option to perform level 2 optimization and the +I option to prepare the code for instrumentation. (+I generates intermediate code.) The -c option in the first command line suppresses linking and creates an intermediate object file called sample.o. The .o file can be used later in the optimization phase, avoiding

a second compile.

The second command line uses the -o option to link sample.o into sample.exe. The +I 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.

Instrumenting Code at Level 4 Optimization

When optimizing at level 4, (where code generation is delayed until link time), use the +I option as follows:

```
aCC +I +O4 -c x.C  y.C     Create intermediate file for instrumentation.
aCC +I +O4 x.o y.o         Create optimized code with instrumentation.
```

For More Information:

- Maintaining Instrumented and Optimized Program Files

Collecting Data for Profiling

To collect execution profile statistics, run your *instrumented* program with representative data as follows:

```
sample.exe < input.file1   Collect execution profile data.
sample.exe < input.file2   Collect execution profile data.
```

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 +I or +P options. The +df command line option can be used to specify the name of the profile data file when used with the +P option.

The +df option takes precedence over the FLOW_DATA environment variable.

Examples:

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 +I +O3 sample.C
aCC -o sample.exe +I sample.o
sample.exe < input.file1
aCC -o sample.exe +P  sample.o
```

In the next example, the +df option is used to override the flow.data file name with the name /users/profiles/prog.data.

```
aCC -c +I +O3 sample.C
```

```
aCC -o sample.exe +I sample.o
sample.exe < input.file1
mv flow.data /users/profile/prog.data
aCC -o sample.exe +df /users/profiles/prog.data +P sample.o
```

Performing Profile-Based Optimization

To optimize the program based on the previously collected run-time profile statistics, relink the program as follows:

```
aCC -o sample.exe +P sample.o
```

When optimizing at level 4, (where code generation is delayed until link time), use the +P option as follows:

```
aCC +P +O4  x.o  y.o
```

When +P is used, no recompilation is necessary. The .o file saved from the instrumentation phase can be used as input.

Maintaining Instrumented and Optimized Program Files

You can maintain both instrumented and optimized versions of a program. You might keep an instrumented version of the program on hand for development use, and several optimized versions on hand for performance testing and program distribution.

Care must be taken when maintaining different versions of the executable file because the *instrumented* program file name is used as the *key identifier* when *storing* execution profile data in the data file.

The optimizer must know what this *key identifier* name is in order to find the execution profile data. By default, the *key identifier* name used to *retrieve* the profile data is the *instrumented* program file name.

When you optimize a program file and the optimized program file name is different from the instrumented program file name, you must use the +pgm option. Specify the instrumented program file name with this option. The optimizer uses this value as the *key identifier* to retrieve execution profile data.

Example:

In the following example, the instrumented program file name is sample.inst. The optimized program file name is sample.opt. The +pgm name option is used to pass the instrumented program name to the optimizer:

```
aCC -c +I +O3 sample.C
aCC -o sample.inst +I sample.o
sample.inst < input.file1
aCC -o sample.opt +P +pgm sample.inst sample.o
```

Pragma Directives

You typically use a #pragma directive to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.

Put **pragmas** in your C++ source code where you want them to take effect. Unless otherwise noted below, a pragma is in effect from the point where it is included to the end of the compilation unit or until another pragma changes its status.

A #pragma directive is an instruction to the compiler and is ignored during

preprocessing.

Syntax:

#pragma *pragma-string*

pragma-string can be one of the following instructions to the compiler with any required parameters.

- COPYRIGHT -- Specify a copyright string.
- COPYRIGHT_DATE -- Specify a copyright date for the copyright string.
- hdr_stop -- When using header caching, specify the end of the prefix header region. In a given source file, this header cannot be reset.
- HP_SHLIB_VERSION -- Create versions of a shared library routine.
- LOCALITY -- Name a code subspace.
- OPTIMIZE -- Turn optimization on or off.
- OPT_LEVEL -- Set an optimization level.
- pack -- Allows maximum alignment of class fields having non-class types.
- VERSIONID -- Specify a version string.

See Also: **Pragmas for Improving Shared Library Performance**

- HP_NO_RELOCATION -- Omit floating-point parameter relocation stubs.
- HP_LONG_RETURN -- Use a long return instruction sequence instead of an interspace branch and omit export stubs.
- HP_DEFINED_EXTERNAL -- Inline import stubs.

Pragma OPTIMIZE

Syntax:

```
#pragma OPTIMIZE ON
#pragma OPTIMIZE OFF
```

Description:

Turns optimization on or off.

Use this pragma to turn off optimization in sections of a source program.

NOTE:

- You must specify one of the optimization options on the aCC command, otherwise this pragma is ignored.
- This pragma cannot be used within a function.

Example:

```
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.
}
```

Pragma OPT_LEVEL

Syntax:

```
#pragma OPT_LEVEL 1
#pragma OPT_LEVEL 2
#pragma OPT_LEVEL 3
#pragma OPT_LEVEL 4
```

Description:

The OPT_LEVEL pragma sets the optimization level to 1, 2, 3, or 4.

NOTE:

- You must specify one of the optimization options on the aCC command, otherwise this pragma is ignored.
- This pragma cannot raise the optimization level above the level specified in the command line.
- This pragma cannot be used within a function.
- OPT_LEVEL 3 and 4 are allowed only at the beginning of a file.

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.
   ...
}
```

Pragma hdr_stop

Syntax:

```
#pragma hdr_stop
```

Description:

When you request header caching with the +hdr_cache option, this pragma specifies the end of the prefix header region.

In a given source file, this pragma cannot be reset. It only allows the prefix header region to be shortened, not expanded. Also, there is no equivalent hdr_start pragma to alter the beginning of the prefix header region.

If the #pragma hdr_stop occurs as the first line in a source file, header caching is abandoned for that source file. If it occurs as the first line in a header file, header caching is abandoned for the source file that includes the header file.

Usage

If you do not want certain headers precompiled, no matter what source file they are included in, instead of searching for all the sources that include them and turning the +hdr_cache option off for each of those sources, you can specify #pragma hdr_stop as the first line in each such header file.

Another example might be useful when only a subset of headers are undergoing code changes. You could place these headers at the end of the list of #include directives, after specifying #pragma hdr_stop. Then, if the more stable headers comprising the prefix header region and the compile environment itself do not change, no re-compile will take place.

Examples

In the following example, using the default prefix header region, changes to any header file will cause a re-compile.

```
#include <vector>     // stable headers
#include <list>
#include <SuperBase>
#include <IOBase>

#include <Magikal>    // headers you are changing
#include <Util>
                           // default END of prefix header region
void poof();
```

In the next example, by ending the prefix header region prior to the headers you are changing, a re-compile does not occur unless changes are made to a stable header file.

```
#include <vector>     // stable headers
#include <list>
#include <SuperBase>
#include <IOBase>

#pragma hdr_stop           // END the prefix header region
                           // prior to the default

#include <Magikal>    // headers you are changing
#include <Util>

void poof();
```

For More Information:

- Creating and Using Automatic Precompiled Headers
- +hdr_cache option

Pragma HP_SHLIB_VERSION

Syntax:

```
#pragma HP_SHLIB_VERSION ["]date ["]
```

The *date* argument is of the form *month/year* , optionally enclosed in quotes.

- *month* must be 1 through 12, corresponding to January through December.
- *year* can be specified as either the last two digits of the year (94 for 1994) or a full year specification (1994). Two-digit year codes from 00 through 40 represent the years 2000 through 2040, respectively.

Description:

Creates different versions of a routine in a shared library.

HP_SHLIB_VERSION assigns a version number based on *date* to a module in a shared library. The version number applies to all global symbols defined in the module's source file.

This pragma should only be used if incompatible changes are made to a source file. If a version number pragma is not present in a source file, the version number of all symbols defined in the object module defaults to 1/90.

For More Information:

- Creating and Using Shared Libraries
- Advanced Shared Library Features
- See the manual *HP-UX Linker and Libraries Online User Guide* .

Pragma COPYRIGHT

Syntax:

```
#pragma COPYRIGHT "string"
```

string is the set of characters included in the copyright message in the object file.

Description:

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.

Examples:

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.
```

NOTE: To see the COPYRIGHT string as well as any other strings in the object file, use the strings(1) command with the -a option for example:

```
strings -a ObjectFileName.o
```

Pragma COPYRIGHT_DATE

Syntax:

```
#pragma COPYRIGHT_DATE "string "
```

string is a date string used by the COPYRIGHT pragma.

Description:

Specifies a date string to be included in the copyright message.

Use the COPYRIGHT pragma to put the copyright message into the object file.

Example:

```
#pragma COPYRIGHT_DATE "1988-1992"
```

Places the string "1988-1992" in the copyright message.

NOTE: To see the COPYRIGHT_DATE string as well as any other strings in the object file, use the strings(1) command with the -a option for example:

```
strings -a ObjectFileName.o
```

Pragma LOCALITY

Syntax:

```
#pragma LOCALITY "string "
```

string specifies a name to be used for a code subspace.

Description:

Specifies a name to be associated with the code written to a relocatable object module.

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 $CODE$.

The smallest scope of a unique LOCALITY pragma is a function.

Example:

```
#pragma LOCALITY "MINE"
```

Builds the name $CODE$MINE$ and associates all code following this pragma with this name, unless another LOCALITY pragma is encountered.

Pragma pack

Syntax:

```
#pragma pack [n ]
```

Where *n* can equal 1, 2, 4, 8, or 16 and indicates, in bytes, the maximum alignment of class fields having non-class types. If *n* is not specified, maximum alignment is set to the default value.

Description:

This pragma allows you to specify the maximum alignment of class fields having non-class types. The alignment of the whole class is then computed as usual, 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
}
```

Usage

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
- Template Example
- Handling Unaligned Data
- Implicit Access to Unaligned Data

Refer also to Default Data Storage and Alignment.

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.

Example 1:

```
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.

CAUTION: 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.

Example 2:

```
#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.
```

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.

CAUTION: 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. For example:

Example 3:

```
#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, either of the following two options are available.

- Compile with the *+unum* option, to generate safe but less efficient code for every indirect access to memory.
- Link in the library libhppa.a (part of the fileset ProgSupport.PROG-AUX) and arm the appropriate signal handler with a call to allow_unaligned_data_access(). This causes every signal 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 3 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 passing the address of a field to code not compiled with the +u1 option, consider the

following example. Unless compiled with -DRECOVER on the command-line and linked with /usr/lib/libhppa.a, Example 4 is likely to prematurely terminate with a bus error.

Example 4:

```
#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;
}
```

Pragma VERSIONID

Syntax:

```
#pragma VERSIONID "string "
```

string is a string of characters that HP aC++ places in the object file.

Description:

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.

Example:

```
#pragma VERSIONID "Software Product, Version 12345.A.01.05"
```

Places the characters Software Product, Version 12345.A.01.05 into the object file.

NOTE: To see the VERSIONID string as well as any other strings in the object file, use the strings(1) command with the -a option for example:

```
strings -a ObjectFileName.o
```

Pragmas for Improving Shared Library Performance

The pragmas described here can improve the performance of shared libraries by reducing the overhead of calling shared library routines. All three pragmas should be used together, where applicable, as they depend on one another to a certain extent. You must be very careful in using them because incorrect use can result in incorrect and unpredictable behavior. See the *HP-UX Linker and Libraries Online User Guide* for more information on improving shared library performance.

Pragma HP_NO_RELOCATION

Syntax:

```
#pragma HP_NO_RELOCATION name1 [,name2 ,...nameN ]
```

name1 through *nameN* are names of functions in shared libraries. Note that C++ mangled names are not supported.

Any named function:

- **must** be the name of an extern"C" function
- **must not** have C++ linkage
- **must not** be a member function

Description:

This pragma improves performance of shared library calls by omitting floating-point parameter relocation stubs in calls to shared library functions.

Parameter relocation stubs are instructions that move (relocate) floating-point parameters and function return values between floating-point registers and general registers. They are generated for calls to routines in shared libraries. Relocation stubs are generated when passing floating-point parameters or using a floating-point function return in routines in shared libraries. The HP_NO_RELOCATION pragma prevents this unnecessary relocation.

Usage

Put the HP_NO_RELOCATION pragma in header files of functions that take floating-point parameters or return floating-point data and that will be placed in shared libraries. Putting it in the header file and ensuring all calls reference the header file is one way to ensure that it is specified at the function definition and at all calls.

This pragma is automatically supplied by HP aC++ for non-static member functions.

WARNING: The HP_NO_RELOCATION pragma must be at the function definition and at all call sites. If the pragma is omitted from the function definition or from any call, the linker will generate parameter relocation code and the application will behave

incorrectly since floating-point parameters will not be in expected registers.

CAUTION: Do not use the HP_NO_RELOCATION pragma with functions that use the stdarg macros. See the stdargs(5) man page for information on stdargs macros.

Pragma HP_LONG_RETURN

Syntax:

#pragma HP_LONG_RETURN *name1* [,*name2* ,...*nameN*]

name1 through *nameN* are names of functions in shared libraries. Note that C++ mangled names are not supported.

Any named function:

- **must** be the name of an extern"C" function
- **must not** have C++ linkage
- **must not** be a member function

Description:

This pragma improves performance of shared library calls by using a long return instruction sequence instead of an interspace branch and by omitting export stubs.

An export stub is a short code segment generated by the linker for a global definition in a shared library. External calls to shared library functions go through the export stub. An export stub is generated by default for each function in a shared library. Each call to the function goes through the export stub. The export stub serves two purposes: to relocate parameters and perform an interspace return.

The HP_LONG_RETURN pragma generates a long return sequence in the export stub instead of an interspace branch. If you also use the HP_NO_RELOCATION pragma (for functions taking floating-point parameters), all the code in the export stub is omitted, eliminating the export stub entirely. For functions taking non-floating-point parameters, the HP_LONG_RETURN pragma by itself eliminates the need for export stubs.

Usage

Put this pragma in header files of functions that will go in shared libraries. Specify it at the function definition and at all calls. For functions with floating-point parameters or returns, use the HP_NO_RELOCATION pragma along with the HP_LONG_RETURN pragma.

This pragma is automatically supplied by HP aC++ for non-static member functions.

This pragma is not required if you compile on PA-RISC 2.0 or later or with the +DA2.0 option since the effect is the default. That is, if no relocations are generated, export stubs are not generated on PA-RISC 2.0 and later, and a long return instruction sequence is generated by default, so this pragma has no effect.

WARNING: The HP_LONG_RETURN pragma must be at the function definition and at all call sites. If the pragma is omitted from the function definition or from any call, the compiler will generate incompatible return code and the application will behave incorrectly and likely abort.

CAUTION:

With floating-point parameters, using HP_LONG_RETURN without using HP_NO_RELOCATION could actually degrade performance by creating export stubs and relocation stubs.

These pragmas improve performance of calls to shared library functions from outside the shared library. Therefore do not use this pragma for hidden functions (see the -h and +e linker options) nor for functions called only from within the same shared library linked with the -B symbolic linker option, otherwise this pragma may degrade performance. (See the *HP-UX Linker and Libraries Online User Guide* for information on the above mentioned options.)

Pragma HP_DEFINED_EXTERNAL

Syntax:

#pragma HP_DEFINED_EXTERNAL *name1* [,*name2* ,...*nameN*]

name1 through *nameN* are names of functions in shared libraries. Note that C++ mangled names are not supported.

Any named function:

- **must** be the name of an extern"C" function
- **must not** have C++ linkage
- **must not** be a member function

Description:

This pragma improves performance of shared library calls by inlining import stubs.

Import stubs are code sequences generated at calls to shared library routines. The import stub queries the PLT (Procedure Linkage Table) to determine the address of the shared library function and calls it. The HP_DEFINED_EXTERNAL pragma inlines this import stub.

Usage

Place this pragma at calls to shared library routines along with the HP_NO_RELOCATION pragma (if using floating-point parameters or return values) and the HP_LONG_RETURN pragma.

Unlike HP_NO_RELOCATION and HP_LONG_RETURN, HP_DEFINED_EXTERNAL is not automatically supplied by HP aC++.

WARNING:

Do not use the HP_DEFINED_EXTERNAL pragma at function definitions, only at function calls. Specifying it at function definitions will result in incorrect behavior.

On PA-RISC 1.1, use the HP_DEFINED_EXTERNAL pragma only when calling a shared library from an executable file. Using it on calls within an executable file will cause the program to abort.

CAUTION:

If your function takes floating-point parameters, you should also use the HP_NO_RELOCATION pragma (if floating-point parameters are present).

You should also use the HP_LONG_RETURN pragma with the HP_DEFINED_EXTERNAL pragma. If you don't, the import stub may be too large to inline.

Use the HP_DEFINED_EXTERNAL pragma only on calls to functions in shared libraries. On PA-RISC 2.0, it will degrade performance of calls to any other functions.

Creating and Using Precompiled Header Files

You can reduce compilation time by precompiling common include (header) files. HP aC++ provides two mechanisms for precompiling headers. **Header caching** is easy to use and was first incorporated in version A.01.21 (for HP-UX 10.x) and HP aC++ A.03.13 (for HP-UX 11.x). The prior mechanism, **manual precompiled headers** , remains available.

NOTE: Mixing the two mechanisms for a single compilation is not supported. The +hdr_cache option is incompatibile with the +hdr_create and +hdr_use options.

Deciding which Mechanism to Use

When deciding whether to use header caching or manual precompiled headers, filespace and ease of use are of major concern.

With header caching, HP aC++ precompiles what is included in the source file (prefix header region) and is responsible for correctness. You need only specify the +hdr_cache option. However, significantly more disk space is required compared to manual precompiled headers. And an initial compile or a re-compile may take more time.

With manual precompiled headers, you control exactly what is precompiled, and when, and you are responsible for the correctness of the code. However, you can minimize disk space usage and precompilation time.

One way of deciding which mechanism to use would be to specify +hdr_cache to determine if file space is a concern. If it is, then consider the manual precompiled header mechanism.

Header Caching

Header caching enables the compiler to cache and use the result of compiling header files from one build to the next. The first build after enabling this option is likely to take somewhat longer but subsequent builds, to the extent that the contents of the cache remain valid, should be measurably faster.

To request header caching, just specify the +hdr_cache option on the aCC command line or in your Makefile or use the CXXOPTS environment variable. There are no other requirements. The header caching mechanism automatically decides if and when to create a precompiled header and if and when to reuse an existing one.

NOTE: Compilation caches for programs using templates or large headers can require substantial disk space. If you need to reclaim disk space, it is safe to remove the cache directory (aCC_cache) in its entirety (rm -rf aCC_cache), but avoid removing only portions of the directory.

Header Cache Processing

When you are compiling with +hdr_cache for the first time, or if the elements of the compilation environment have changed since the prior compilation, the compiler does the following for each source file on the command line:

- Creates an aCC_cache sub-directory in the directory where the source file resides (for an initial compile only). This directory acts as a repository for precompiled header files.

  Note, use the +hdr_dir option to specify a different location and/or name for the aCC_cache directory.

- Identifies the prefix header region in each source file, precompiles it, and stores the precompiled version of its contents in the the aCC_cache directory.

- Encapsulates the unique compilation environment of the source file and stores its content in the aCC_cache directory. This information is used in future compiles to determine whether or not a given precompiled header is valid for reuse.

  Use the +hdr_info option to see if a cache is being reused or recreated. If you find that you need more than one cache directory, use the +hdr_dir option.

By contrast, for an unchanged compilation in which the compiler has examined the compilation environment and found no significant changes, the existing precompiled header is reused and compilation continues. Typically, for such compiles, performance is significantly improved.

The Prefix Header Region

The prefix header region is the initial region of a source file that contains #include and #define directives. The end of the prefix header region (and thus the automatic precompile) is reached when the compiler encounters a code sequence other than a #include or #define directive (i.e., a declaration or operator), or when a #pragma hdr_stop is encountered.

Performance and File Space Considerations

In order to use header caching most effectively, consider the following points.

- In general, compile time will be faster for a re-compile using an existing precompiled header file, than for a compilation that does not use precompiled headers. Most notable improvement would be for code that uses many header files.

- Compile time will be slower for an initial compilation or one for which the prefix header region has changed than for a re-compile using an existing precompiled header.

- A given application using automatic precompiled header files may require more file space than would a manual precompiled header file. It is good programming practice to not include unnecessary headers.

- Try to include all necessary headers before any of the other code in a source file. This will insure that maximum precompilation occurs.

- In compiling a source-file, if you do not want to precompile all #include header files, specify the #pragma hdr_stop directive to end the prefix header region prior to those headers that are not to be precompiled.

- Remember that compilation time is generally improved for source code in which the prefix header region is not modified.

Manual Precompiled Headers

You can manually create a precompiled header file. Then when you compile your application or library, you specify the precompiled header file on the command line. Note, with this method, just one precompiled header file is allowed per compilation.

To manually create and use a precompiled header file, follow these steps:

1. First you need a source (.C) file that includes all the header files you want to precompile (precomp.C in the following example).

2. Next create a precompiled header file (in this case named precomp) from precomp.C by using the +hdr_create option.

   Use the -c option to suppress creation of the executable file.

   Each time you use the +hdr_create option to create a precompiled header file, by default, a corresponding .o file is also created. Information resulting from the compilation of declarations is put into this .o file. The .o file may contain information related to debugging, virtual function tables, and inline function bodies. This saves space and time by eliminating duplication in future +hdr_use compiles.

   ```
   aCC precomp.C -c +hdr_create precomp
   ```

3. When you want to compile precomp, use the +hdr_use option. This is known as a load compile.

   ```
   aCC main.C +hdr_use precomp
   ```

Verbose Information

Use the +hdr_v option for verbose information when precompiling a header or when compiling a precompiled header file.

To see what goes into the precompiled header file:

```
aCC precomp.C -c +hdr_create precomp +hdr_v
```

To see what is being brought into the compiler during a load compile:

```
aCC main.C +hdr_use precomp +hdr_v
```

For More Information

- Example Source Files
- Writing Headers that can be Either Compiled or Precompiled
- Safeguarding your Precompiled Header Files
- Creating a Two-tiered Headers Process

Example Source Files

The files in the example below are written so that you can compile them either with or without precompiled headers, as described in Writing Headers that can be Either Compiled or Precompiled. Thus you could issue the following command to compile the files without precompiling:

```
aCC main.C

Or to use the precompiled header created in the prior example:

aCC main.C +hdr_use precomp

In the following example, header file a.h is included in precomp.C and precomp.C is
included in main.C.

// a.h

#ifndef A_H
#define A_H
extern "C" int printf(char *, ...);
class foo {
private:
   int x;
public:
   foo() { printf("constructor for foo\n");x++; }
};
#endif // A_H

// precomp.C

#ifndef PRECOMP
#define PRECOMP
#include "a.h"
class bar : foo {
private:
   int y;
public:
   bar() { printf("constructor for bar\n");y++; }
};
#endif //  PRECOMP

// main.C

#include "precomp.C"   // Use this include statement
                       // ONLY if you want
                       // headers that can be
                       // either precompiled or compiled.
void main()
{
   bar b;
}
```

Writing Headers that can be Either Compiled or Precompiled

If you want to be able to either compile a header file directly or to precompile and than compile it (a load compile), the file must have the following characteristics.

- You must #include the precompiled header source (.C) file in your main program. Even though this is not necessary for a load compile, it is required when you compile without first precompiling.
- In order to create a precompiled header file, you must have one source (.C) file that includes all of the other header files you intend to precompile.
- Use include guards in the source file that includes the header files and in the header files themselves. Include guards are generally used to ensure that the

contents of the header files are included only once. They are required in a load compile since you #include the precompiled header source (.C) file in main.C.

The +hdr_use option has the side effect of defining a compilation flag (e.g. A_H and PRECOMP), so the conditional directives prevent redefinitions of the contents of a.h and precomp.C respectively.

For more information about preprocessor directives, refer to Preprocessing in HP aC++

Safeguarding your Precompiled Header Files

Be certain you remake a precompiled header file if anyone could have changed any of the header files it represents. Otherwise these header changes will not be part of the load compile.

It is also recommended that, from time to time, you re-compile everything (header and .C files) without the +hdr_use option. This ensures that each .C file contains exactly the correct #include directives.

For example, suppose you have source files prog1.C and prog2.C that each require a #include prog.h directive. If the #include directive is missing from prog2.C, an error message would be generated if you re-compile that file without the +hdr_use option. With the +hdr_use option, however, no error is generated since prog.h is in the precompiled header file because of the prog1.C #include directive.

Creating a Two-tiered Headers Process

When project files are under active development, it is sometimes better to divide your header files and precompiled header files into groups, depending on how often they are being modified. For example,

1. Build a file called StableHeaders.C that includes only headers that are changed infrequently.
2. Build a file called VolatileHeaders.C that includes only headers that are frequently changed. (If you use include guards, this file can include headers from the set in StableHeaders.C.)
3. Build two precompiled header files, and combine them into one using commands like the following:

   ```
   aCC StableHeaders.C -c +hdr_create Stable

   aCC VolatileHeaders.C +hdr_use Stable -c +hdr_create Total
   ```

4. Compile a program with the resulting precompiled header (in this case, Total).

   ```
   aCC Prog.C +hdr_use Total
   ```

Preprocessing in HP aC++

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**.

Directives

By coding **preprocessing directives** in a source file, you request that the file be processed in a particlar way.

Command Line Options

Migration

For More Information

- See the aCC man page.

Summary of Preprocessor Directives

Preprocessor directives provide the following functionality:

- Source File Inclusion (#include)

  Include other source files at a given point, for example, to centralize declarations or to access standard system headers such as iostream.h.
- Macro Replacement (#define, #undef)

  Replace token sequences with other token sequences. In C, this technique is frequently used to define names for constants rather than explicitly putting the constant value into the source file. In C++ you can also use the keyword const to define constants.

- Conditional Compilation (#if, #ifdef, .. #endif)

  Check values and flags to either compile or skip source code based on the outcome of a comparison. Useful, for example, when writing a single source for use with several different configurations.

- Line Control (#line)

  Set the line number and file name of the next line.

- Pragma Directive (#pragma)

  Give implementation-dependent instructions, called **pragmas**, to the compiler. Because they are system-dependent, pragmas are not portable.

- Error Directive (#error)

  Create diagnostic messages for the compiler to issue.

- Trigraph Sequences

  Use trigraph sequences if you don't have the full C++ character set.

Preprocessor Directives: Syntax, Guidelines, and Examples

Syntax

General syntax for a preprocessor directives is:

```
preprocessor-directive ::=
     include-directive newline
     macro-directive newline
     conditional-directive newline
     line-directive newline
     pragma-directive newline
     error-directive newline
     trigraph-directive newline
```

Guidelines

Following are rules and guidelines for using preprocessor directives:

- A preprocessor directive must be preceeded by a pound sign (#) as the first character on a line of your source file. (However, if you are in ANSI C mode, 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).

Examples

Following are some examples of preprocessor directives:

```
include-directive:
       #include <iostream.h>
macro-directive:
       #define MAC x+y
conditional-directive:
       #ifdef MAC

       # define x 25

       # endif
line-directive:
       #line 5 "myfile"
pragma-directive:
       #pragma OPTIMIZE ON
error-directive:
```

```
      #error "FLAG not defined!"
trigraph-directive:
      ??=line 5 "myfile"
```

Source File Inclusion (#include)

You can include the contents of other files within a given source file by using the #include directive in the source file.

Syntax:

```
include-directive ::=
     #include <filename >
     #include "filename "
     #include 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 (&dquote; &dquote;), 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.

For More Information:

- Using Standard HP-UX Libraries and Header Files
- Header File Options

Files that are included may contain #include directives themselves. HP aC++ supports a nesting level of at least 35 #include files.

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
```

Macro Replacement (#define, #undef)

You can define C++ macros to substitute text in your source file.

Macro Syntax and Description:

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 of the form:

#define *identifier [replacement-list]*

defines the *identifier* as a macro name that represents the *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.

NOTE: 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 "\" 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 the *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.

For 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 the preceding 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 1000
float 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

Two other macros 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))
```

NOTE: 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.

Because 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 might not work as expected because the argument a is incremented two times when a is the maximum:

```
i = MAX(a++, b);
```

which is expanded to

```
i = ((a) > (b) ? (a) : (b))
```

Given the above macro definition, the statement

```
i = MAX(a, b+2);
```

is expanded to:

```
i = ((a) > (b+2) ? (a) : (b+2));
```

More Examples

```
// This macro tests a number and returns TRUE if
// the number is odd.  It returns FALSE otherwise.
#define isodd(n)  ( ((n % 2) == 1) ? (TRUE) : (FALSE))

// This macro skips white spaces.
#define eatspace()while((c=getc(input))==c=='\n'c\
 = '\t' )
```

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__ (refer to Line Control (#line)), HP aC++ provides the predefined macros listed below. 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.
- `__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 six 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
- `__LINE__` produces the current source line number.
- `__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*.

For More Information

To use some HP-UX system functions you may need to define the symbol `__HPUX_SOURCE`. See the stdsyms(5) man page if it is installed on your system, or in the *HP-UX Reference Manual* . (If you see the message "Man page could not be formatted," ensure the man page is installed.)

Conditional Compilation (#if, #ifdef, .. #endif)

Conditional compilation directives allow you to delimit portions of code that are compiled only if a condition is true.

Conditional Compilation Syntax and Description

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 )
```

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
```

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.

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 nonzero 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`*

They 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 yet defined.

Nesting Conditional Compilation Directives

You can nest conditional compilation constructs. Delimit portions of the source program using conditional directives at the same level of nesting, or with a `-D` option on the command line.

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"
```

Pragma Directive (#pragma)

A `#pragma` directive is an instruction to the compiler. You typically 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.

Example:

```
#pragma OPTIMIZE ON
```

For More Information:

- Refer to Pragma Directives for descriptions of pragmas recognized by HP aC++

Error Directive (#error)

Syntax:

```
error-directive  ::=
     #error [preprocessor tokens]
```

Description:

The `#error` directive causes a diagnostic message, along with any included token arguments, to be produced by HP aC++

Examples:

```
     // 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
```

Trigraph Sequences

Description:

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"
```

Standardizing Your Code

HP aC++ A.01.21 largely conforms to the ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++). Choose from the following for more information:

- Standard Functionality
  - Explicit Template Instantiation
  - Keywords
  - Overloading `new[ ]` and `delete[ ]` for Arrays
  - Passing Standards Related Options to the Compiler
  - Standard Exception Classes
  - Standard Exceptions
  - `type_info` Class

- HP aC++ Extensions
- Unsupported Functionality

Migration

- Migration Considerations Related to Standardization

HP aC++ Keywords

- and
- and_eq
- bitand
- bitor
- bool
- catch
- class
- compl
- const (also an ANSI C keyword)
- const_cast
- delete
- dynamic_cast
- explicit
- false
- friend
- inline
- mutable
- namespace
- new
- not
- not_eq
- operator
- or
- or_eq
- private
- protected
- public
- reinterpret_cast
- static_cast
- template
- this
- throw
- true
- try
- typeid
- typename
- using
- virtual
- volatile (also an ANSI C keyword)
- wchar_t
- xor
- 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

```
void 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 can't (or shouldn't) be modified.

Example Code with Discussion

```
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
}

void 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
};

void main()
{
   Base *p;
   Derived *q;

   Base b;
   p = &b;
   q = dynamic_cast<Derived *> (p);
}
```

The above generates a compile-time error stating that:

```
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()
{
}

void 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 can't 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
};

void 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

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 Code with Discussion

```
class C {
public:
  explicit C(int);
```

```
};

C::C(int)
{
  // empty definition
}

void 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

};

void 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
};

void  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 constness 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 a use or reference count in an object that keeps track of the number of pointers referring to it.

Example Code with Discussion

```
class C {
public:
  C();
  int i;
  mutable int j;
};

C::C() : i(1), j(3)
{
  // Define constructor
}

void 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

Basic Concepts

Namespaces were introduced into C++ primarily as a mechanism to avoid naming conflicts between various libraries. The example below illustrates how this is achieved.

As can be seen, 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, object, etc. 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).

One somewhat unique feature of namespaces is that they can be extended. The example below shows this as well as the connections between a namespace extending across different translation units.

The example also illustrates the concept of so-called `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

The 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 would occur.

using-derective

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.

CAUTION: Using-directives are transitive. If you specify a using-directive to 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.

For More Information

- `typeid` Example Code with Discussion
- `type_info` Class

`typeid` Example Code with Discussion

```
# 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.
}

void 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 might be called a typeid switch statement. It is not generally a reasonable design. 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 could rewrite the above 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
};

void 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 could rewrite the above 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.
}

void 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 keyword `volatile` is used in declarations. It tells the compiler not to do aggressive optimization because a value might be changed in ways the compiler couldn't 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 Code with Discussion

```
class C {
public:                      // public to make example simpler
volatile int i;
// other class details omitted
};

C someData[10];

void 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 part of the ANSI C standard.

Usage

This type was added to maintain ANSI C compatibility and to accomodate foreign (principally Oriental) character sets.

Example Code with Discussion

In the following example, literals of type `wchar_t` consist of the character L followed by a character constant in single quotes.

```
void 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.

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 Code

```
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
  };
};

void main ()
{
  C1<C> c;
}
```

Discussion

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.

Usage

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.

For More Information

- Example Code with Discussion
- Migration -- Using `operator new` to Allocate Arrays

Example Code with Discussion

```
# 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

void 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.

Usage

These classes provide a common framework for the way errors are handled in a C++ program. Systen-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"));
}

void main()
{
  try {
    f();
  }
  catch (runtime_error& r) {
    // handle any kind of runtime error including range_error
    cout << r.what() << '\n';
  }
}
```

Discussion

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. **CAUTION:** If no catch clauses are available to catch these exceptions, the default action is program termination with a call to abort(). (Note that using the +noeh option does not disable the exceptions thrown by these library functions.)

- `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()`.

Usage

You need to write try/catch clauses to handle the standard exceptions. For an example, refer to Memory Allocation Failure and operator `new`.

---

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.
}

void 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 can't 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

---

HP aC++ Extensions

HP aC++ A.01.21 supports the following language features in addition to those defined in the ANSI/ISO C++ International Standard:

- long long -- integral type specifying a signed 64-bit integer
- unsigned long long -- integral type specifying an unsigned 64-bit integer

---

Unsupported Functionality

Functionality defined in the ANSI/ISO C++ International Standard and not supported in this release of HP aC++ is listed below.

NOTE: This is not an all inclusive list.

- Standard C++ Library components not in namespace std::
- covariant return types with multiply inheriting types
- Template Features
  - separation model for template compilation (`export` keyword)
  - template template parameters
  - omission of template parameter names
- function try blocks
- support for universal-character-sequences (\uxxxx)

Using HP aC++ Templates

The following sections overview template processing and describe the instantiation coding methods available to you. Refer to the technical document *Using Templates in HP aC++* for more detailed explanation.

Comparing Template Instantiation Mechanisms

You have the choice of two template instantiation mechanisms. The default **compile-time** mechanism instantiates every template used in a given translation unit in that translation unit. You can invoke the **automatic instantiation** mechanism by using a comand-line option. The assigner then decides in which object file an instantiation is placed.

When you link an application or library with aCC, you can combine object files that have been compiled using either mechanism.

- Deciding which Mechanism to Use
- Template Processing

Invoking Instantiation

- Explicit Instantiation (developer-directed)
- Command-Line Option Instantiation (developer-directed)
- Compile-time Instantiation, the 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_all option requests instantiation of all used template functions and all static data members and member functions of instantiated template classes within a translation unit. Whereas, using explicit instantiation requests instantiation of all members of a particular template class or a particular template function.

Migration Considerations

- Migrating from HP C++ (cfront) to HP aC++
- Migrating from the Automatic Instantiation Default to the Compile-time Instantiation Dafault

See Also:

- An introductory C++ Template Tutorial
- HP aC++ Template Documentation
- Debugging Templates

Deciding which Mechanism to Use

Why Use Automatic Instantiation

To **close** a set of link units, you must use automatic instantiation. More specifically, you may want to use automatic instantiation for the following reasons.

- If you provide archive or shared libraries for distribution, you may want to use the +inst_auto and

+inst_close options to insure consistent behavior between each distribution of your libraries.

- If you provide either archive or shared library products, and your customers need to use the prior template instantiation default in their builds, you must build your libraries by using the +inst_auto and +inst_close options.

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.

Migration Considerations

If you used automatic instantiation with HP aC++ A.02.00 or A.01.04 and prior versions and you wish to continue using it with subsequent versions of HP aC++, be aware of some possible migration problems and solutions.

Template Processing

HP aC++, provides two template instantiation mechanisms, **compile-time instantiation** (the default) and **automatic instantiation** (invoked by using a command-line option). Following are overviews of each type of template processing. For more detailed information, refer to the technical document *Using Templates in HP aC++* .

The major difference between compile-time and automatic instantiation processing is that, with compile-time instantiation, the compiler instantiates every template entity it sees in a translation unit provided it has the required template definition; with automatic instantiation, the compiler instantiates only what the assigner tells it to (except for explicit instantiations). It is the assigner's responsibility to make sure that every template entity is instantiated and that it is instantiated only once.

Compile-time Template Processing

- The assigner is not invoked. 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 (use). Only the chosen instantiation appears in the a.out or .sl 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 since the assigner is not used. All .o files are compiled only once.

Automatic Template Processing

Automatic instantiation uses the assigner, an executable file that runs at pre-link time to help perform the following tasks:

- The assigner's automatic instantiation algorithm determines in which object (.o) file an instantiation is placed.
- Instantiation information is placed in object (.o) files. This aids the assigner in selecting a unique instantiation site for a given instantiation.
- Assignment information resides in a .I file. This indicates to the compiler which instantiations are to be placed in the corresponding .o file (upon recompilation).

For More Information

- More about Automatic Instantiation Files
- Major Components of the Compiling System
- Migrating from the Automatic Instantiation Dafault to the Compile-time Instantiation Dafault

Automatic Instantiation Files

Automatic instantiation involves the creation of .o and .I instantiation files, described below. By default, these files are placed in the directory in which you are compiling.

.o Instantiation Files

A `.o` file contains information generated by the compiler to tell the assigner about templates in a given translation unit. It contains the following types of information recognized or produced by the assigner:

- template members in the translation unit (def)
- demands for instantiation as a result of explicit instantiation (dem)
- requests for instantiation as a result of template object declarations (req)
- actual instantiations resulting from an assignment request (ins)
- assignments of instantiations to this translation unit (asi)

.I Instantiation Files

The `.I` file, produced by the assigner, contains assignment of instantiations to a translation unit by the instantiation algorithm. It is an ASCII file. The command line below uses `c++filt` to view the instantiation information in file a.I.

```
/opt/aCC/bin/c++filt < a.I
```

The output shown below tells you the a.c translation unit has been assigned three instantiations.

```
asi Stack<int>::Stack()
asi Stack<int>::~Stack()
asi Stack<int>::push(int)
```

Migrating from the Automatic Instantiation Default to the Compile-time Instantiation Default

If you used automatic instantiation with HP aC++ A.02.00 or A.01.04 and prior versions and you wish to continue using it with subsequent versions of HP aC++, modify each of your existing aCC command-lines by adding the +inst_auto option. This applies to command-lines for:

- creating object files
- creating an executable
- closing a set of object files prior to creating a library (.a or .sl)
- creating a shared library (.sl) provided you do not specify +inst_none

The following sections describe specific migration scenarios and illustrate possible migration problems and solutions.

- Possible Duplicate Symbols in Shared Libraries
- Possible Duplicate Symbols in Archive Libraries
- Mixing Old .o and .a Files with New Ones

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 in HP aC++ A.02.00 or A.01.04 and prior versions you built an archive library using automatic instantiation (the prior default using the assigner) and you rebuild that library using the current default (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), 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 the Default (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.

NOTE: Note that this example is not meant to account for all cases of changed behavior.

```
lib.default.a
-------------------------------------------------
| foo.o         | foo2.o        | foo3.o        |
|   stack<int>  |   stack<int>  |   stack<int>  |
|   x           |   x           |   y           |
|   y           |               |               |
-------------------------------------------------
```

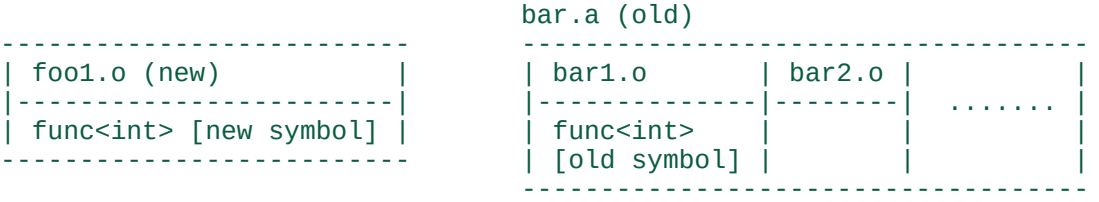
Mixing Old .o and .a Files with New Ones

What happens when you mix .o and .a files compiled with HP aC++ A.02.00 or A.01.04 and prior versions with files compiled with subsequent versions of HP aC++? The linker gives an old symbol precedence over a new symbol of the same name. For example:

```
foo1.o (old)          foo2.o (new)          foo3.o (new)
-----------           -----------           -----------
func<int>             func<int>             func<int>
[old symbol]          [new symbol]          [new symbol]
```

In this case the linker chooses the func <int> from foo1.o and ignores the other two, because the old func <int> takes precedence over any of the new ones. Note that if there were more than one old func <int>, the linker would give a duplicate symbol error.

A Special Case of Mixing Old .o and .a Files with New Ones



Since old symbols take precedence, you would expect the linker to choose func <int> from bar1.o. However, since bar1.o is part of an archive library, the linker will never even try to load it unless it's needed to resolve some other symbol in foo1.o.

So if ld loads bar1.o, then it will choose func <int> from bar1.o However, if ld does not load bar1.o, it will choose the only func <int> that's available, and that's the one in foo1.o.

And bar1.o might not be loaded, even though it was loaded under the old default. This behavior may not cause a problem, however, in some cases it may. For example, in a correctly written program in which multiple definitions of func <int> are equivalent, there is no problem. However, if multiple definitions of func <int> are not equivalent, the compiler does not detect the error.

Linker Error Checking Messages

Given all of the above, the linker provides error checking to help find out what may be happening.

- By default, ld issues generic warnings like the following when it sees a new/old pair: aCC old.o new.o

  ```
  /opt/aCC/lbin/ld:  (Warning) Linker features were used that may not
  be supported in future releases.  The +vallcompatwarnings option
  can be used to display more details, and the ld(1) man page contains
  additional information.  This warning can be suppressed with the
  +vnocompatwarnings option.
  ```

- If you want more information: aCC old.o new.o -Wl,+vallcompatwarnings

  ```
  /opt/aCC/lbin/ld: (Warning) An automatic template instantiation for
  member "func <int>()" in file t.o has been overridden by an explicit
  definition in fi le new.o. This behavior may not be supported in
  future releases.
  ```

- If you want to see duplicate symbol messages based on what would happen if compatibility with old .a and .o files were *not* supported, issue the command-line: aCC old.o new.o -Wl,+strictctti This generates a message like the following:

  ```
  /opt/aCC/lbin/ld: Duplicate symbol "func<int>()" in files old.o and
  ```

```
new.o /opt/aCC/lbin/ld: Found 1 duplicate symbol(s)
```

- To suppress all linker compatibility warnings, use the command-line: aCC old.o new.o -Wl,+vnocompatwarnings

---

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, for example, 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 maintenence 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 of 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 requird 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
```

For More Information

- Refer to the ANSI/ISO C++ International Standard for additional details including explicit specialization syntax.

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.

Compile-time Instantiation, the Default

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
```

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**.

Debugging Templates

The HP WDB Debugger and the HP/DDE Debugger support C++ templates.

For More Information

- HP WDB Debugger Documentation
- HP/DDE Debugger Documentation

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:

```
void 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.

```
void 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.

Using Threads

The HP aC++ run-time environment supports multi-threaded applications.

The following HP aC++ libraries are thread-safe:

- `libstd.sl` and `libstd.a`
- `librwtool.sl` and `librwtool.a`
- `libCsup.sl` `libCsup.a`
- `libstream.sl` `libstream.a`

In order to pick the thread safe version of I/O routines (cout, cin, cerr, and clog) when you include iostream.h in your source files, you can add the -D_THREAD_SAFE compile time flag to your compilation line.

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 statement with locks. (Note, if you use locks, you do not have to use the -D_THREAD_SAFE compile time flag.) 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 accomodate both the user threads and the kernel threads interfaces, as in the above example.

Required Command-line Options

To use the multi-thread safe capabilities of the Standard C++ Library and the Tools.h++ Library, you need to specify the following options at both compile and link time:

- -D__HPACC_THREAD_SAFE_RB_TREE (Note, code compiled with this option is binary incompatible with code that is not compiled with this option. This option is available for HP aC++ version A.01.21 and subsequent versions.)
- -DRWSTD_MULTI_THREAD
- -DRW_MULTI_THREAD (needed only for the Tools.h++ Library)
- -D_REENTRANT
- -lcma (Note, this option applies only to user threads.)
- -lpthread (Note, this option applies only to kernel threads.)

WARNING: If you do not specify these options, a run-time error will be generated or multi-thread behavior will be incorrect.

Using `-D__HPACC_THREAD_SAFE_RB_TREE`

The current standard C++ library (libstd) and RogueWave's tools.h++ library (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, tvmmap.h`.

Since changing the current `rb_tree` implementation to make it thread safe would break binary compatibility, the preprocessing macro, __HPACC_THREAD_SAFE_RB_TREE, must be defined. Whether or not this macro is defined when compiling a file that includes the `tree` header, its use must be consistent. For example, 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 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) {}
}
```

For More Information

See documentation on using threads and writing multi-threaded applications.

Introduction to HP aC++

HP aC++ is Hewlett-Packard's implementation of the ANSI/ISO C++ International Standard. The compiler largely conforms to the standard and is evolving towards full conformance. Refer to Standardizing Your Code for listings of standards based features and extensions. Some of the many supported features are listed here:

- Precompiled Header Files
- Standard C++ Library
- Tools.h++ Library
- Templates

New Features in this HP aC++ A.01.21 Release

- Header File Caching is an additional, simplified method of precompiling header files.
- A new debugging option, `+[no]objdebug`, enables faster links and smaller executable file sizes for large applications.
- Additional Options for Standardizing Your Code:
  - `-Wc,-ansi_for_scope,[on]` enables standard scoping rules for init-declarations in `for` statements.
  - `-Aa` sets all C++ standard options on (currently Koenig lookup and `for` scoping rules).
- Additional Options for Code Optimization:
  - `+O`*`level`* `=`*`name1`* `[,`*`name2`* `,...,`*`nameN`* `]`
  - `+Oreusedir=`*`DirectoryPath`*
- A new template option, `+inst_directed`, to suppress assigner output in object files. Use it instead of the `+inst_none` option with code that contains explicit instantiations only and does not require automatic (assigner) instantiation.
- The #pragma pack directive allows you to specify the maximum alignment of class fields having non-class types. This pragma may be useful when importing code from other architectures where data type alignment may be different from default PA-RISC alignment.
- Three new pragmas for improving the performance of shared libraries.

- 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 the compilation and linking of their applications.
- See additional information about HP aC++ diagnostic messages.
- To see which include files led to an error or warning, specify the `-Wc,-diagnose_includes,on` option.
- Floating installation allows more than one version of HP aC++ to be installed on one system at the same time.
- The `__HP_aCC` predefined macro now contains the HP aC++ driver version number. For example, for version A.01.21, `__HP_aCC` = 012100

  The `__HP_aCC` predefined macro was introduced in HP aC++ version A.01.15. It's value was 1 for HP aC++ A.01.15 and A.01.18.
- In prior releases, the standard C++ library (libstd) and RogueWave's tools.h++ library (librwtool) were not thread safe in all cases. The -D__HPACC_THREAD_SAFE_RB_TREE preprocessor macro insures thread safety.

Release Notes

For the latest information on new features, see the *HP aC++ Release Notes*

If you see the message "Text file data could not be formatted," ensure the release notes are installed as `/opt/aCC/newconfig/RelNotes/ACXX.release.notes`.

Migration

If you are migrating code from HP C++ (cfront) to HP aC++, click here to find out where to obtain information.

Features Introduced in Prior Releases

- New Default Template Instantiation Mechanism
- Support for Member Templates as Defined in the ANSI/ISO C++ International Standard

---

New Default Template Instatiation Mechanism

The HP aC++ default template instantiation mechanism has changed to compile-time instantiation (CTTI). For source code containing templates, the new default may result in faster compile-time processing.

The previous default behavior remains available by specifying the +inst_auto command-line option when compiling and linking. If you provide archive or shared libraries for distribution, you may want to use +inst_auto to insure consistent behavior between each distribution of your libraries.

Also, if you provide either archive or shared library products, and your customers need to use the prior template instantiation default in their builds, you must build your libraries by using the +inst_auto option.

For More Information

Refer to Using HP aC++ Templates in this online programmer's guide and to the online technical document, *Using Templates in HP aC++* for details about template instantiation and migration.

Information Map

NOTE: This information map is current as of this release of HP aC++ A.01.21.

If you are accessing the map for the first time, please read the DISCLAIMER.

Choose from the following categories for information about C++ and related topics. Most books listed here are available from technical bookstores, some of which provide online ordering on the World Wide Web.

HP aC++ A.01.21 Product Documentation

- Release Notes
- Online HTML Files
- Online Man Pages
- Example Source Files
- Linker and Libraries
- Migration from HP C++ (cfront) to HP aC++
- Standardizing Your Code
- Threads
- HP aC++ World Wide Web Homepage
- HP aC++ Libraries
- HP aC++ Linking
- HP aC++ Templates
- HP WDB Debugger
- HP SoftBench Development Environment
- HP-UX

Technical Books and Courses

- C++ Syntax and Basics
- C++ Concepts
- C++ Examples
- C++ Standards
- Object Oriented Programming

- C++ Courses

Release Notes

HP aC++ Release Notes -- For the latest information about HP aC++, release notes are provided as follows:

- an ASCII file which is part of the HP aC++ product, /opt/aCC/newconfig/RelNotes/ACXX.release.notes
- a printed copy which is part of the HP aC++ product
- on the World Wide Web at the following URL: http://docs.hp.com/hpux/development/
- on the HP-UX CD-ROM

HTML Files

- The *Rogue Wave Software Standard C++ Library Class Reference* and *Rogue Wave Software Tools.h++ 7.0 Class Reference* are provided as HTML formatted files. Refer to HP aC++ Libraries for details.

- A technical paper summarizing template features defined in the proposed C++ standard and describing template instantiation as implemented in HP aC++ is provided. Refer to HP aC++ Templates for details.

- ***HP aC++ Online Programmer's Guide*** -- The guide you are currently viewing focuses primarily on HP aC++ specific information. Refer to Technical Books and Courses for information related to the C++ language and object oriented programming.

    - **Navigating Online** -- Click with the left mouse button on any underlined word to link to additional information. To return to a prior topic, click the right mouse button and choose **back**.

    - **Search on a Keyword** -- To find a particular character string within the file you are currently viewing, use the left mouse button to select **Edit** and **Find**.

      On HP-UX, a method of searching the entire set of files comprising the guide is to use the grep command. For example, to display all lines containing the exact string, `performance` followed by a space, in all files in the named directory:

      ```
      grep 'performance ' /opt/aCC/html/C/guide
      ```

      If you want the search to be case insensitive and the results directed to a file named perf:

      ```
      grep -i 'performance ' /opt/aCC/html/C/guide > perf
      ```

    - **Printing** -- From your HTML viewer menu bar, use the left mouse button to select **File** and **Print**.

    - **Invoking this Online Guide** -- The guide is provided as HTML formatted files, viewable by means of your web browser. Invoke the guide in either of the following ways:

        - Specify the +help option on the aCC command line.

        - From your web browser, enter the appropriate URL:

          ```
          file:/opt/aCC/html/C/guide/index.htm (English)

          file:/opt/aCC/html/ja_JP.SJIS/guide/index.htm (Japanese)
          ```

          To see Japanese characters when using the Netscape browser, choose:

          ```
          1.  Options
          2.  Document Encoding
          3.  Japanese (Auto-Detect)
          ```

      NOTE: All of the files composing the English version of the guide are installed in the /opt/aCC/html/C/ directory. If you choose to move the entire English guide to a different location without having to edit any links, you will need to move all of the subdirectories in /opt/aCC/html/C/. All of the files composing the Japanese guide are installed in /opt/aCC/html/ja_JP.SJIS/.

Man Pages

The following online manual pages are provided:

- *aCC* (1) -- The online manual page for the HP aC++ compiler command, aCC is located in the directory /opt/aCC/share/man/man1.Z. (If you see the message "Man page could not be formatted," ensure the man page is installed.
- Online man pages for the Standard C++ Library are located in the directory /opt/aCC/share/man/man3.Z.

  Japanese man pages are located at:
  - /opt/aCC/share/man/ja_JP.eucJP/man1.z and /opt/aCC/share/man/ja_JP.eucJP/man3.z for the euc character set
  - /opt/aCC/share/man/ja_JP.SJIS/man1.z and /opt/aCC/share/man/ja_JP.SJIS/man3.z for the SJIS character set
- The online manual page for c++filt is at /opt/aCC/share/man/man1.Z.
- Online manual pages for the cfront compatibility libraries are at /opt/aCC/share/man/man3.Z. For listings of these man pages with brief descriptions, refer to:
  - Standard Components Library
  - IOStreams Library

Example Source Files

Online C++ 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.

Threads

- For information specific to HP aC++ see Using Threads in this programming guide.
- *Programming with Threads on HP-UX* (B2355-90060) -- To order a paper copy contact Hewlett-Packard's Support Materials Organization (SMO) at 1-800-227-8164 and provide the above part number.
- *Thread Time: The Multithreaded Programming Guide* , by Scott J. Norton and Mark D. DiPasquale (ISBN 0-13-190067-6) -- Hewlett-Packard Professional Books, published by Prentice Hall

HP aC++ World Wide Web Homepage

Refer to the Homepage for the latest information regarding:

- Frequently Asked Questions
- Release Version and Patch Table
- Purchase and Support Information
- Documentation Links

Access the HP aC++ World Wide Web Homepage at the following URL:

`http://www.hp.com/go/c++`

HP aC++ Libraries

Standard C++ Library

- *Rogue Wave Software Standard C++ Library Class Reference* -- This reference provides an alphabetical listing of all of the classes, algorithms, and function objects in the Rogue Wave implementation of the Standard C++ Library. It is provided as HTML formatted files which you can view with an HTML viewer such as Netscape. The files are located at /opt/aCC/html/libstd/ref.htm.

  Note that the *Standard C++ Library Class Reference* can be ordered in paper copy from Rogue Wave Software Inc.

- Man Pages -- Online man pages for the Standard C++ Library are located in the directory /opt/aCC/share/man/man3.Z.

  Japanese man pages are located at:
  - /opt/aCC/share/man/ja_JP.eucJP/man1.z and /opt/aCC/share/man/ja_JP.eucJP/man3.z for the euc codeset
  - /opt/aCC/share/man/ja_JP.SJIS/man1.z and /opt/aCC/share/man/ja_JP.SJIS/man3.z for the SJIS codeset

- Example Source Files -- Online example source files for the Standard C++ Library are located in the directory, /opt/aCC/contrib/Examples/RogueWave.

- The Rogue Wave Software, Inc. home page provides additional information. To view the home page on the World Wide Web, enter the following URL:

  `http://www.roguewave.com/`

Tools.h++ Library

- *Rogue Wave Software Tools.h++ 7.0 Class Reference* -- This reference describes all of the classes and functions in the Tools.h++ Library. It is provided as HTML formatted files which you can view with an HTML viewer such as Netscape. The files are located at /opt/aCC/html/librwtool/ref.htm.

  Note that the *Tools.h++ Class Reference* can be ordered in paper copy from Rogue Wave Software Inc. Also available in paper are:

    - *Tools.h++ User's Guide* -- How to write programs using Tools.h++.
    - *Tools.h++ Getting Started Guide* -- Installing and using Tools.h++.

- Example Source Files -- Online example source files for the Tools.h++ Library are located under the directory, /opt/aCC/contrib/Examples/RogueWave.

- The Rogue Wave Software, Inc. home page provides additional information. To view the home page on the World Wide Web, enter the following URL:

  `http://www.roguewave.com/`

HP C++ (cfront) Compatibility Libraries

- Standard Components and IOStream Man Pages -- HP C++ (cfront), compatible versions of the `standard compoments` class library and the `iostream` library are provided with HP aC++.

- Online man pages are located in the directory /opt/aCC/share/man/man3.Z. For listings of these man pages with brief descriptions, refer to:
    - Standard Components Library
    - IOStreams Library

To Order Books from Rogue Wave Software, Inc.

`800-487-3217 (phone)`

`541-757-6650 (FAX)`

HP aC++ Linking

HP-UX Linker and Libraries Online User Guide -- This online help guide is provided with HP aC++, C, COBOL, and Fortran compiler products. It replaces the *Programming on HP-UX* manual.

The guide describes fundamentals of software development on HP-UX, including how the basic pieces of the development environment fit together --compilers, assemblers, linker, libraries, and object files. Also covers using the ld linker to create executable programs, creating and linking archive and shared libraries, writing position-independent code (used to build shared libraries), managing shared libraries from within a program, porting applications to HP-UX, and advanced system programming techniques.

The guide is provided online in HP VUE or CDE format. To access, click the question mark icon (**?**) or enter the command **ld +help**.

For More Information

- Refer to Creating and Using Libraries in this programming guide.

Migration from HP C++ (cfront) to HP aC++

If you are migrating code from HP C++ (cfront) to HP aC++:

1. Refer to the migration section for information about differences between the two compilers.

Also refer to the *HP aC++ Transition Guide* at the following URL:

http://www.hp.com/esy/lang/cpp/tguide

2. For general background information and experience, subscribe to the cxx-dev list server (like a notes group). Send a message to majordomo@cxx.cup.hp.com with the following command in the body of the message: subscribe *list-name*

   Available list-names are as follows:

   ```
   cxx-dev            HP C++ Development Discussion List
   cxx-dev-announce   HP C++ Development Announcements
   cxx-dev-digest     HP C++ Development Discussion List Digest
   ```

   cxx-dev-announce is also broadcast to cxx-dev, so there is only a need to subscribe to one of the lists. The digest also includes both cxx-dev and cxx-dev-announce.

   For additional help or information about the list server, send a message to majordomo@cxx.cup.hp.com with the following command in the body of the message: help

3. For specific support questions, contact your HP support representative.

4. For generic C++ questions, see documents and URL's listed in this Information Map.

HP aC++ Templates

- Using HP aC++ Templates (in this programming guide) describes the instantiation coding methods and options available to you and provides an overview of HP aC++ template processing.

- *Using Templates in HP aC++* -- This technical document summarizes template features defined in the proposed C++ standard and describes template instantiation as implemented in HP aC++. It is provided with HP aC++ in the following locations and formats:

  ```
  /opt/aCC/newconfig/TechDocs/templates.ps -- postscript format
  /opt/aCC/html/C/templates/templates.htm -- HTML format
  ```

C++ Syntax and Basics

The following books are available at technical bookstores.

- *The Annotated C++ Reference Manual* , by Margaret Ellis and Bjarne Stroustrup (ISBN 0-201-51459-1) -- A complete C++ language reference manual plus annotations and commentary that describe in detail why features are defined as they are.

- *C++ Primer* , second edition, by Stanley Lippman (ISBN 0-201-54848-8) -- A complete tutorial introduction to C++, for those with little or no C or C++ experience.

  See Also:

- Standards -- For information related to the ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++).
- Standardizing Your Code -- For standards that are supported by HP aC++ and migration considerations related to standardization.

C++ Concepts

The following books are available at technical bookstores.

- *Effective C++ Plus: 50 Specific Ways to Improve Your Programs and Designs* , by Scott Meyers (ISBN 0-201-563-649) -- 50 concise rules based on what experienced C++ developers almost always do (or almost always avoid) to create efficient, portable, and maintainable software. Each rule is accompanied by examples that illustrate the rule at work.

- *More Effective C++ Plus: 35 New Ways to Improve Your Programs and Designs* , by Scott Meyers (ISBN 0-201-633-71X) -- Drawing on years of experience, Meyers explains how to write software that is more effective: more efficient, more robust, more consistent, more portable, and more reusable.

- *Advanced C++ Programming Styles and Idioms* , by James Coplien (ISBN 0-201-54855-0) -- For programmer's having knowledge of C++ basics, this book imparts information gained from a broad range of C++ programming experience.

- *The Design and Evolution of C++* , by Bjarne Stroustrup (ISBN 0-201-54330-3) -- A history of the C++ language by its creator.

C++ Examples

- *C/C++ Annotated Archives* , by Art Friedman, Lars Klander, Mark Michaelis, and Herb Schidlt (ISBN 0-07-882504-0) -- A collection of carefully documented C/C++ components and programs covering a wide range of computing applications.

C++ Standards

- The ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++) was ratified by the C++ standardization committees in July, 1998. It is available for download from the World Wide Web at the ANSI Electronic Store.

  The document is in PDF format with total size of 2794KB. Its cost is $18.00 USD payable online via credit card.

- A December, 1996, HTML version of the draft (which does **not** reflect recent changes) is publically available on the World Wide Web at the following URL:

  `http://www.cygnus.com/misc/wp`

The following books are available from technical book stores.

- *The C++ Programming Language, Third Edition* , by Bjarne Stroustrup (ISBN 0-201-88954-4) -- Based on the C++ *Final Draft International Standard* , this book is a complete rewrite of the second edition. It covers the language, its standard library, and key design techniques as an integrated whole.
- *C++ Solutions: Companion to the C++ Programming Language, Third Edition* , by David Vandevoorde (ISBN 0-201-30965-3) -- This book describes solutions to a selection of examples from Bjarne Stroustrup's book on standard C++.
- *STL Tutorial & Reference Guide: C++ Programming with the Standard Template Library* , by David R. Musser R. and Atul Saini (ISBN 0-201-633-981) -- This book introduces the STL and provides the information and techniques you need to become a proficient STL programmer. The book includes a tutorial, a thorough description of each element of the library, numerous sample applications, and a comprehensive reference.

See also:

- C++ Libraries -- For additional information about the Standard C++ Library.
- Standardizing Your Code -- For standards that are supported by HP aC++ and migration considerations related to standardization.

Object Oriented Programming

The following books are available at technical bookstores.

- *Object-Oriented Design with Applications* by Grady Booch (ISBN 0-805-35340-2) -- Object oriented analysis and design concepts and implementation, using C++ and a unified notation that incorpoartes Booch and other widely used methods.
- *Design Patterns: Elements of Reusable Object-Oriented Software* , by Erich Gamma, Richard Help, Ralph Johnson, John Vissles (ISBN 0-201-63361-2) -- A catalog of 23 design patterns to help solve commonly occurring design problems. Based on real-world examples. Each pattern describes the circumstances in which it is applicable, when it can be applied in view of other design constraints, and the consequences and trade-offs of using the pattern within a larger design.
- *Design Patterns for Object Oriented Programming* by P (ISBN 0-201-42294-8) -- Conceptual background in the development and reuse of semifinished software architectures (application frameworks) rather than single components. These frameworks embody foundational components and their behavioral interaction. A design example of a hypertext system is illustrated with ET++, the famed user-interface framework available in the public domain.

HP-UX Information

- *Assembly Language Reference* (92432-90001) -- Describes the use of the Precision Architecture RISC (PA-RISC) Assembler.
- *HP-UX Floating Point Guide* (B2355-90624) -- Describes how floating-point arithmetic is implemented on HP 9000 Series 700/800 workstations, and discusses how floating-point behavior

affects the programmer. This book provides information useful to any programmer writing or porting floating-point-intensive programs.

General HP-UX Documentation

- *The Ultimate Guide to the vi and ex Text Editor* (HP 97005-90015) -- Complete information about using the vi and ex text editors on HP-UX.

HP WDB Debugger Information

To download the HP WDB product and/or documentation, access the following World Wide Web URL:

`http://www.hp.com/go/wdb`

HP/DDE Debugger Information

- *HP/DDE Debugger User's Guide* (B3476-90011) -- Information on debugging C++ programs with the HP Distributed Debugging Environment, dde, on the HP 9000. This document can be ordered by contacting Hewlett-Packard's Support Materials Organization (SMO) at 1-800-227-8164 and providing the above part number.
- Also refer to the *dde* (1) man page. (If you see the message "Man page could not be formatted," ensure the man page is installed and your `MANPATH` variable includes `/opt/langtools/share/man`.)
- Online help for DDE is available from the DDE Menu Bar.

HP SoftBench Development Environment

The following manuals are available for SoftBench products. Copies can be ordered by contacting Hewlett-Packard's Support Materials Organization (SMO) at 1-800-227-8164.

- *Getting Started with SoftBench on HP-UX 10.x* -- contains SoftBench tutorials for C, C++, and COBOL.
- *C and C++ SoftBench User's Guide for HP-UX 10.x* -- contains information on using C and C++ SoftBench.
- *Installing and Customizing SoftBench Products* -- contains installation and customization information for SoftBench Products on HP-UX 9.x, HP-UX 10.x and Solaris.

Online Ordering

You can research and order technical books on the World Wide Web. Following is a subset of the many URL's related to technical publications.

- CompuBooks
- Computer Literacy Bookshops, Inc.
- Computer Manuals Online Bookstore (Europe)
- Ecola's Computer Publications
- Hotline (Australia)
- Softpro Books

C++ Courses

- Rogue Wave Software, Inc. provides courses on their Standard C++ Library, Tools.h++ Library, and other products. For information, access Rogue Wave on the World Wide Web:

  `http://www.roguewave.com/products/profserv/profserv.html`

Glossary

aggressive optimizations
: Any optimizations that can change the behavior of structured code. This is a superset of basic optimizations.

anachronistic constructs
: Elements of the C++ language that will be obsoleted and therefore unsupported in some future release.

archive library
: A collection of object files grouped using the ar command. At link time, only object files that have needed symbols are extracted from the library.

argument declaration file
: For templates, a file containing the declaration of a class, struct, union, or enum type.

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.

  Note that in versions A.02.00 and A.01.04 and prior versions of HP aC++, automatic instantiation was the default. The default is now compile-time instantiation.

base class
: A class from which another class, the derived class, inherits the public and protected members. That is, 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 do not generally change the behavior of structured code. This category of 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.

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 a library
: Satisfying all template instantiations needed by a library when building the library, not when linking the library with an application.

closing
: The process of satisfying all template instantiations for a set of **link units**.

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.

  Note that in versions A.02.00 and A.01.04 and prior versions of HP aC++, automatic instantiation was the default.

conservative optimizations
: Any optimizations that do 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.

derived class
: A class that inherits the public and protected member data and the public and protected **member functions** from its base class. Sometimes 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 execute 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 by means of an explicit instantiation or a compiler command-line option.

exception

An exception is a run-time error condition. Exception handling is a C++ mechanism that allows the detector of the error to pass the error condition to code (the exception handler) that is prepared to handle it. An exception is raised by a `throw` statement within a `try` block and handled by a `catch` clause.

Note, 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 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 is available to other object files to link against.

friend
: Either a class or a function that has access to all of a class's data and member functions. That is, the friend has access to the class's **public**, **protected**, and **private members**.

function template
: A template that defines an unbounded set of related functions.

HP C++
: HP's initial, pre-*C++ draft proposed international standard* C++ compiler. It is based on the cfront compiler and provides functionality for templates and exception handling.

HP aC++
: HP's most recent C++ compiler. It closely complies with most features of the ANSI/ISO C++ International Standard.

header file
: An C++ source file typically containing class or function declarations and referenced by other C++ source files using the `#include` preprocessor **directive**.

include guards
: Preprocessor commands (typically `#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 funtion generated from a funtion 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.

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 (.sl file).

load compile
: Invoking the compiler using the +hdr_use option, and a manual precompiled header file.

member data
: Any data elements declared to be part of a class.

member function
: Any function declared to be part of a class.

millicode library
: The millicode library contains special purpose routines that are tailored for performance. The routines are implemented in PA-RISC assembly code and follow a special stream-lined procedure calling convention. The millicode routines are not intended to be called directly by user programs due to the strict coding, calling, and register usage requirements. Refer to the "PA-RISC Procedure Calling Conventions Reference Manual" for details on the special millicode calling convention.

multiple inheritance
: The ability of a class to inherit from more than one base class. That is, the derived class inherits all public and protected members from all of its base classes. Compare to single inheritance.

name demangling
: The process of changing the internal representation of identifiers back to their original C++ source names. Compare to 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. Compare to name demangling.

object
: An instance of a class.

parameterized type
: See template.

position-independent code (PIC)
: Object code that contains no absolute addresses. All addresses are specified 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 has been 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
: A kind of 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,
  - from any class derived from that class, and
  - 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.

shared library
: A collection of object files grouped using the aCC command and comprised of position-independent code. At link time, all object files are made available.

single inheritance
: The ability for a class, the derived class, to inherit from exactly one class, its base class. Compare to multiple inheritance.

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 containing C++ program code.

specialization
: An instantiation of a template class or template function that overrides the standard version.

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 (technically, the output of the preprocessor). A translation unit normally results in a single object file.

Looking at it another way, 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.

yacc
: A programming tool for describing the input to a computer program.