HP aC++/HP ANSI C A.06.28 Release Notes

HP Integrity servers

HP Part Number: 769149-001
Published: March 2014
Edition: 15

© Copyright 2012, 2014 Hewlett-Packard Development Company L.P. All rights reserved
Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein. UNIX is a registered trademark of The Open Group.

Intel® and Itanium® is a trademark of Intel Corporation in the U.S. and other countries.

Contents

HP secure development lifecycle

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

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

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

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

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

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

1 HP aC++/HP ANSI C Release Notes

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

The HP ANSI C compiler supports ANSI programming language C standard ISO 9899:1999. HP aC++ compiler supports the ISO/IEC 14882 Standard for the C++ Programming Language (the international standard for C++). HP ANSI C and HP aC++ are distributed as a single software bundle.

All information in this document is applicable to both HP C and HP aC++, unless stated otherwise.

This document discusses the following topics:

2 What's new in this version?

HP aC++/HP C compiler version A.06.28 provides additional support for C++11 core language features, with complete binary compatibility with earlier releases and -AA compilation mode.

Following are the changes in HP aC++/HP C compiler version A.06.28:

Improved support for C++11 core language features

Preliminary support for the following C++11 core language features has been introduced in this release:

Uniform initialization

This release of the compiler supports 'uniform initialization' under the C++11 compilation mode. To initialize, a class, struct, or an array is given a list of arguments in braces, in the order of the members' definitions in the class.

For example:

```
struct BasicStruct {
       int x;   double y;
};
 class AltStruct {
  public:
  AltStruct(int x, double y) : x_{x}, y_{y} {}
  int x_;    double y_;
};
int main(int argc, char** argv)
{
  BasicStruct var1{5, 3.2};
  AltStruct var2{2, 4.3};     // Constructor will be called
  int arr[]  { 1,2,3,4,5 };
  return 0;
}
```

"noexcept" specifier and operator

This release of the compiler supports 'noexcept specifier and operator' under the C++11 compilation mode. The function is declared not to throw any exceptions if the value of the constant expression is true. The 'noexcept' operator without a constant expression is equivalent to noexcept(true).

For example:

```
template <class T>
   void foo() noexcept(noexcept(T())) {}

void bar() noexcept(true) {}
void baz() noexcept { throw 42; }
int main()
{
  foo<int>();  // fine
  bar();  // fine
  baz();  // compiles, but at runtime this calls std::terminate()
}
```

Range-based for loops

This release of the compiler supports 'range-based for loop' as described by N2778 under the C++11 compilation mode.

For example:

```
int main()
{
 int my_array[5] = {1, 2, 3, 4, 5};
 int sum = 0;
 for (auto &x : my_array) {
   sum += x;
 }
}
```

Opaque enumeration definitions

This release of the compiler supports the 'Opaque enumeration' as described by N2764 under the C++11 compilation mode. It allows forward declaration of enumerations by declaring an enumeration without providing its enumerators.

For example:

```
class S
{
 public:
 enum E : int;
 E e;
};
enum S::E : int
```

```
{
 RED, GREEN, BLUE
};
```

constexpr

This release of the compiler supports 'constexpr' under the C++11 compilation mode. The "constexpr" feature allows functions and constructors that return constant values for constant arguments, and variables whose initializers must be reduced to constants at the compile time. Check N2235 and the additions in N3078, N3268, and N3277. In the following example, all the constructors, conversion functions, and operator functions are folded at compile time to produce a constant value for a3.

For example:

```
struct A {
 int i;
 constexpr A(int p) : i(p) {}
 constexpr operator int() { return i; }
};
constexpr A operator+(A a1, A a2) {
   return A{(int)a1 + (int)a2};
}
int main() {
 constexpr A a1{1};
 constexpr A a2{2};
 constexpr A a3 = a1 + a2;
}
```

Nonstatic data member initializers

This release of the compiler supports 'Nonstatic data member initializers' as described by N2756 under the C++11 compilation mode. (NSDMIs) Nonstatic Data Member Initializers are initializers on nonstatic data member declarations within the class.

For example:

```
struct A {
 int i = 1;
 A() {}  // i gets initial value of 1
 A(int p) : i(p) {} // i is initialized to p; NSDMI is not used
};
```

Unrestricted unions

This release of the compiler supports 'unrestricted unions' under the C++11 compilation mode. In traditional C++, unions cannot contain fields with nontrivial construction, assignment, or destruction semantics. However, C++11 does permit such fields but if so, the implicit default corresponding constructor, destructor, or copy assignment operator of union is deleted, forcing a manual definition.

For example:

```
struct S { S(); };
union U {
 S s;  // Previously an error; now accepted in C++11 mode.
 int i;
};
U u;   // Error: The generated default constructor of U is deleted.
      // forcing user to provide constructor for union U.
```

Delegating constructors

This release of the compiler supports 'Delegate Constructors' as described by N1986 under the C++11 compilation mode. It allows one constructor to call another constructor of the same class to do the initialization.

For example:

```
struct S {
 S(int);
 S(): S(0) {}  // Default constructor for S delegates to constructor
};       // S::S(int).
```

Raw string literals

This release of the compiler supports 'Raw string literals' as described by N2442 and N3077 under the C++11 compilation mode. It allows the specification of string literals containing characters that otherwise have special meaning. These literals can span multiple lines and escape sequences, and trigraphs are not translated.

For example:

```
const char *p = R"+++(a\
??=\0xa
z)+++";  // Equivalent to "a\\\n\?\?=\\0xa\nz"
```

UTF-8 string literals

This release of the compiler supports 'UTF-8 string literals' as described by N2442 under the C++11 compilation mode. It allows strings with an encoding of UTF-8 (a popular representation for Unicode).

For example:

```
const char s[] = u8"\u03a9";  // equivalent to "\xce\xa9"
```

Ref-qualifier on this

This release of the compiler supports 'Ref-qualifier on this' as described by N2439 under the C++11 compilation mode. It allows the declaration of member functions that operates only on lvalue or rvalue objects.

For example:

```
extern "C" int printf(const char *, ...);
struct A {
 void p() & { printf("&\n"); }
 void p() && { printf("&&\n"); }
};
int main() {
 A a;
 a.p();    // &
 A().p();  // &&
}
```

Inline namespace

This release of the compiler supports 'Inline namespace' as described by N2535 under the C++11 compilation mode. This is done to specify that members of the namespace can be defined and specialized as though they actually belong to the enclosing namespace.

For example:

```
namespace Networking {
  namespace V1 {
    class TCPSocket;
  }
  inline namespace V2 {
    class TCPSocket;
  }
  class UDPSocket;
}
int main()
{
```

```
 Networking::TCPSocket *t;       // Networking::V2::TCPSocket,
                // because of the inline namespace
 Networking::V1::TCPSocket *t2;  // Networking::V1::TCPSocket
 Networking::V2::TCPSocket *t3;  // Networking::V2::TCPSocket
 return 0;
}
```

Implicit move constructors and assignment operators

This release of compiler supports 'Implicit move constructor and assignment operator' as described by N3053 under the C++11 compilation mode.

Converting lambdas to function pointers

This release of compiler supports 'Converting lambdas to function pointer' as described by N3052 under the C++11 compilation mode. A lambda expression with an empty capture set shall be convertible to pointer to function type R(P), where R is the return type and P is the parameter-type-list of the lambda expression.

New options in this release

Option to select compilation mode

A new option has been added in this release to select compilation mode:

+std=c89|c99|c++98|c++11|gcc|g++|gnu

+std=c89: This option invokes the compiler in ANSI C89 compilation mode. This option, when specified with the -ext option, it invokes a part of ANSI C99 features. This is equivalent to '-AC89' option.

+std=c99: This option invokes the compiler in ANSI C99 compilation mode with its features. This is the default C compilation mode. This is equivalent to '-AC99' option.

+std=c++98: This option invokes the compiler in ISO C++98 standard mode. This is the default C++ compilation mode. This is equivalent to '-AA' option.

+std=c++11: This option turns on support for several core language features introduced by the ISO C++11 language standard. It is available only in C++ compilation mode and is binary compatible with the '+std=c++98' ('-AA') compilation mode. This is equivalent to -Ax option.

+std=gcc: This option enables GNU C dialect compatibility. This option is equivalent to '–Agcc' option.

+std=g++: This option enables GNU C++ dialect compatibility. This option is equivalent to '–Ag++' option.

+std=gnu: This command line option is also used to enable gnu dialects, it switches between '+std=gcc' or '+std=g++' compilation, depending on whether the compilation mode is C or C++ respectively.

Option for selecting standard C++ library

A new option '+stl' has been added in this release to select the available implementation of Standard Template Library (STL):

+stl=rw|none

+stl=rw: This option is used to specify RogueWave STL 2.0 implementation. This option is equivalent to '–AA' option. It includes C++98 compliant STL. This is the default STL. This option causes standard C++ header files to be picked up from the directory '/opt/aCC/include_std' and linked with libstd_v2.so.

+stl=none: By eliminating references to the standard header files and libraries bundled with HP C++ compiler, this option allows experienced users to have full control over the header files and libraries used in compilation, and linking of their applications. This is equivalent to '+nostl' option.

Option to perform instruction scheduling for Poulson based Itanium® architecture

+DSpoulson: This option performs instruction scheduling for Poulson implementation of the Itanium® architecture. Instruction scheduling is different on different implementations of Itanium® based architectures. Performance on a particular model or processor of the HP-UX system can be improved by requesting that the compiler uses the instruction scheduling tuned to that particular model or processor. Object code with scheduling tuned for a particular model executes on other HP-UX Integrity systems, although less efficiently. The default option is +DSblended.

New Library with this release

Starting with this release of the compiler, the driver links in a new C++ runtime library 'libCsup11.so' to the application when compiling in C++11 mode. This supports the new integral types char16_t and char32_t introduced by the C++11 standard.

Applications built using the +std=c++11 compiler option (to enable C++11) must also be linked with the same option.

NOTE: The new C++ runtime library is completely binary compatible with the earlier runtime library. This library is not a part of the compiler release bundle. It is shipped separately as part of the runtime patch. To use this option, ensure that the appropriate runtime patch is installed. This new library 'libCsup11.so' needs to be shipped on the deployment systems along with 'libCsup.so'.

Deprecated and obsoleted options

The following options will be removed from the future version of the compiler:

Existing options marked for deprecation	New equivalent options
-AC89	+std=c89
-AC99	+std=c99
-AA	+std=c++98
-Ax	+std=c++11
-Agcc	+std=gcc
-Ag++	+std=g++
-Agnu	+std=gnu

The following options are obsolete:

- -AP

  To enable future runtime library versions, this option was deprecated earlier and is obsolete in this release. If there is a build using this option, migrate your source to comply with the C++ ANSI standard.

- -Aarm

  This option was deprecated earlier and is obsolete in this release.

CXX_MAP_FILE mapping environment variable for easy migration

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

Syntax:

```
export CXX_MAP_FILE=file path
```

Example:

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

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

Defining the Mapping Rules:

The following syntax is for defining the rules in the mapping file:

LHS => RHS

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

where:

- LHS is the third party compiler option
- RHS is the HP aC++ compiler option

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

For example:

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

Use the following example to disable the use of the mapping file set the above environment variable to NULL:

- `export CXX_MAP_FILE=`
- `export CXX_MAP_FILE=""`

Example Rules

The following example rules map gcc compiler options to corresponding HP aC++ compiler options. The same rules can be used for mapping options from any third party compiler.

The syntax for the rule becomes as follows:

- `gcc_option => hp_option`
- `-Wtraditional =>`
  Ignores (removes) -Wtraditional, a gcc option from the command line.
- `-shared => b`
  Replaces -shared with -b at the command line.
- `-rpath-link $1 =>`
  Deletes -rpath-link and the arguments from the command line.
- `--gccopt $1=$2 => -hpopt $2`
  Replaces "--gccopt option=name" at the command line with "-hpopt name".
- `-gccopt $1 => +xyz`
  Replaces "-gccopt optionarg" at the command-line with "+xyz".
- `-Bstatic => -a archive -noshared`
  Replaces "-Bstatic" with "-a archive -noshared".

3 Product changes in earlier versions

New features in version A.06.27

Preliminary support for the following C++11 core language features has been introduced in this release:

Improved support for C++11 languages

New char types char16_t and char32_t

This release of the compiler supports the char16_t and char32_t char types under the C++11 compilation mode, enabled by the compiler option -Ax. U-literals are also enabled in the C++11 compilation mode.

```
int main(int argc, char** argv)
{
   char16_t *str = u"A 16-bit character string";
     char32_t ch = U'\U00012345';  // A 32-bit character literal
```

```
        return 0;
}
```

Alias and alias template declarations

Alias declarations and alias template declarations are now supported in the C++11 compilation mode.

```
using X = int;
X x;  // equivalent to "int x"
template <typename T> using Y = T*;
Y<int> yi;  // equivalent to "int* yi"

int main(int argc, char** argv)
{
        yi = &x;
        return 0;
}
```

Variadic templates

This release of the compiler supports variadic templates as described by N2242 and extended by N2555 and N2933 in the C++11 compilation mode, enabled by the compiler option -Ax. This feature allows templates that take variable numbers of arguments. Pack expansions are supported in parameter lists, argument lists, brace-enclosed initializers, base class specifiers, mem-initializers, exception specifications, attributes, and capture lists. Also note that in instantiations of functions that have parameter packs it is now possible for multiple parameter variables to have the same name.

```
template<class ...T> void f(T ...args)
{
        int a[] = {0, args..., 5};
}

int main(int argc, char** argv)
{
        f(1, 2, 3, 4);
        return 0;
}
```

nullptr keyword

This release of the compiler adds support for the 'nullptr' keyword in the C++11 compilation mode. This is treated as a null pointer constant.

```
void f(int n){}
void f(char* p){}
int main(int argc, char** argv)
{
        f(nullptr); // invokes f(char*)
        return 0;
}
```

Explicit conversion functions and boolean-conversion

Conversion functions marked "explicit" are now implemented in the C++11 mode. Such conversion functions are used only for explicit casts, and not for implicit conversions. The related concept of "boolean-converted" is also now implemented. It allows explicit conversion functions to bool to be used to produce a bool value in contexts like the expression of an "if" statement even though such a context is not an explicit cast.

```
struct A
{
    explicit operator bool(){return true;}
};
int main(int argc, char** argv)
```

```
{
    A a;
    (void)bool(a); // Okay
    if (a) {}      // Okay
    //bool b = a;  // Would be an error
    return 0;
}
```

Trailing return types (with "auto")

In the C++11 mode the compiler now accepts "trailing return types" in function declarators following an "auto" type specifier.

```
auto f()->int {return 0;}  // Same as int f(){...}
int main(int argc, char** argv)
{
    return f();
}
```

Standard attributes

In C++11 mode, the attributes "align", "carries_dependency", "final", and "noreturn" are now supported.

```
[[noreturn]] void f() {}  // Warning is issued: calls to f() do return.
int main(int argc, char** argv)
{
    f();
    return 0;
}
```

Attributes for checking, hiding and overriding

In the C++11 mode, the compiler now supports the standard attributes "override", "hiding", and "base_check". Attribute "override" indicates that a virtual function overrides a matching function in a base class. "hiding" indicates that a derived-class member hides a base class member. "base_check" can appear on a class definition, and causes the compiler to issue errors if overriding virtual functions are not marked with [[override]] or hiding members are not marked with [[hiding]].

```
struct B { virtual void f(), f(int); };
struct[[base_check]] D: B {
    [[override]] virtual void f();      // Okay.
    [[override]] virtual void f(char);  // Error: No overridden f(char) in B.
    virtual void f(int);                // Error: Missing [[override]].
};
int main(int argc, char** argv)
{
    return 0;
}
```

Use of "this" in trailing-return-types

The compiler now allows references to "this" in decltype expressions in late-specified return types in member functions in the C++11 mode.

```
struct A {
   int i;
   auto f() const -> decltype(this->i) { return 0; }  // Okay now.
};
int main(int argc, char** argv)
{
    return 0;
}
```

Built-in macro __cpluspluss

The built-in macro __cplusplus now has a value of 201103L when the compiler is run in the C++11 mode (using the +std=c++11) option. Earlier, the value was 199711L irrespective of the mode of execution.

Improved support for GNU extensions

This release of aC++ supports a host of GNU compiler extensions. For more details about these features, see the GNU documentation available at http://gcc.gnu.org/onlinedocs/gcc/.

Binary literals

This release of the compiler supports 'Binary literals' under the GNU compilation mode. It allows integer values using binary number system.

Example:

```
int main()
{
  int i = 0b01;
  return 0;
}
```

Better support for type trait pseudo functions

Various type trait pseudo-functions are supported in this release of aC++ in the GNU compilation mode such as __has_nothrow_assign, __has_nothrow_constructor, __has_nothrow_copy, __has_trivial_assign, __has_trivial_constructor, __has_trivial_copy, __has_trivial_destructor, __is_abstract, __is_class, __is_convertible_to, __is_empty, __is_enum, __is_pod, __is_polymorphic and __is_union.

All the above type trait pseudo-functions are now supported in the GNU C++ compilation mode (-Ag++).

Better support for attributes

Various attributes are supported in this release of aC++ in the GNU compilation mode, such as vector_size, transparent_union, gnu_inline, deprecated, weakref and alloc_size.

Better support for templates

When compiling in GNU C++ mode, the following features are now accepted in this release of aC++:

Floating-point operations in template arguments

```
template <int xi> struct foo {};
foo< ( 3.1 < 2.3) > d;
```

Abstract class parameters in templates

```
struct A { virtual void p() = 0; };
struct B:A { virtual void p() {} };
template<typename T> void f(void (*)(T t)){}
int main()
{
    f<A>(0);  // Now elicits a warning instead of an error.
}
```

Pure specifier ("= 0") in templates

```
template<typename T> struct S
{
    void f() = 0;  // An error is issued if the template is instantiated.
```

```
};
```

T& in template that can be deduced to match "this"

```
template <class T> void gg(T &r) { r = 0; }
struct A
{
    void f()
    {
        gg(this);  // T deduced as "A *const", assignment in gg gets error
    }
};
```

Better support for built-ins

Various GNU builtins are now supported in the default compilation mode in this release of aC++, such as __builtin_exit, __builtin_trap, __builtin_vfprintf, __builtin_vfscanf, __builtin_offsetof and GNU macro __COUNTER__.

NOTE: More details on the supported GNU features shall be provided in the next release.

Support for default template arguments for function templates

This release of the compiler supports default template arguments for function templates, as updated by core issue 226 in the C++98 standard.

```
template<class T, class U = T>  void f(T, U = 1){}
int main()
{
    f(1);       // U uses the default of T (which is int in this case)
    f(1, 2.0);  // U deduced as double
}
```

New options in this release

Option to perform less aggressive optimizations to thread local variables

A new option has been added in this release:

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

Experimental features in this release

IELF file-splitting to improve the compilation time at +O4

This is an experimental feature which tries to improve the compilation time through uniform distribution of the intermediate IELF files passed to the optimizer.

This is applicable only for C programs and at optimization level +O4.

To enable this feature, set the environment variable HP_BE_IELFSPLIT during compilation.

Diagnostic Changes

This release of the compiler has significant improvements in the compiler diagnostics, with several new diagnostics and a few deprecated diagnostics. Note that the diagnostic numbers for existing diagnostics are unchanged from previous releases.

New features in version A.06.26

This chapter gives an overview of the product changes in this version of the HP aC++/HP C compiler.

HP aC++/HP C compiler version A.06.26 provides leading edge support for `C++0x` standard language features, with complete binary compatibility with earlier releases and `-AA` compilation mode.

Following are the changes in HP aC++/HP C compiler version A.06.26:

Improved GNU compatibility and new GNU features

Support for GNU statement expression

In this release, HP aC++ compiler provides support for GNU statement expressions in default compilation mode.

In the GNU C mode of compilation (`-Agcc`), a compound statement enclosed in parentheses may appear as an expression. This is now allowed in default compilation mode, that is, even without `-Agcc`. This feature allows you to use loops, switches, and local variables within an expression.

Example 1 Expression in default compilation mode

```
({ int y = foo (); int z; if (y > 0) z = y; else z = - y; z; })
```

This is a valid expression for the absolute value of `foo ()`. The last item in the compound statement should be an expression followed by a semicolon (;). The value of this sub-expression serves as the value of the entire construct.

Support for _Pragma ("once")

The `_Pragma ("once")` operator is equivalent to `#pragma once`. This operator ensures that the source file is included only once during compilation.

C++0x language extensions

This release provides two new features related to C++0x language extensions.

Unnamed types and static routines are given globally unique names

The unnamed classes and enums in namespaces (including the global namespace) as well as static functions are now mangled such that they do not collide with names from other translation units and can be used in the mangled names of template functions.

Example 2 Globally unique names for unnamed types and static routines

```
template void f(T) {} template void f(T, T1) {} struct {} s; enum {} e; static void f2(); void f1() { f2();
// ZN22_INTERNAL_3_x_c__Z2f1v2f2Ev f(s);
// _Z1fIN22_INTERNAL_3_x_c__Z2f1v4__C1EEvT f(e);
// IA-64: Z1fIN22_INTERNAL_3_x_c__Z2f1v4__E2EEvT f(s, s);
//_Z1fIN22_INTERNAL_3_x_c__Z2f1v4__C1ES1_EvT_T0_ f(e, e);
// Z1fIN22_INTERNAL_3_x_c__Z2f1v4__E2ES1_EvT_T0 f(s, e);
//_Z1fIN22_INTERNAL_3_x_c__Z2f1v4__C1ENS0_4__E2EEvT_T0_ }
static void f2() { struct A {} a; f(a);
//_Z1fIZN22_INTERNAL_3_x_c__Z2f1v2f2EvE1AEvT_ }
```

Rvalue references

This feature is enabled with the -Ax option, which turns on support for selected features from the upcoming C++ standard (called C++0x).

The existence of Rvalue references allows the declaration of move constructors, which can be used to efficiently copy class objects by transferring the resources from a source object, that is soon to be defunct, to a destination object.

Option to redirect make-dependency

The options `+make` and `+Make` generate the make-dependency list, and writes this out to the `stdout`. This list can be captured into `a .d` file that is based on the basename of the object file.

The new option `+Makef` writes out the make-dependency to a file that you specify (including the file name and location). This facility is on the lines of the GNU options `-M` and `-MF`.

Example 3 Redirecting make-dependency to a file

```
aCC +Makef make_dependency_output_file source_file
aCC +Makef hello.make-dep hello.c
```

Support for initialization of Flexible Array Member

The compiler now supports the initialization of Flexible Array Member. Following are the characteristics of this implementation:

- Flexible array members are written as contents[] without the 0.
- Flexible array members have incomplete type, and so the `sizeof` operator may not be applied. As a side effect of the original implementation of zero-length arrays, `sizeof` evaluates to zero for flexible array members.
- Flexible array members must appear as the last member of a struct that is otherwise non-empty.
- A structure containing a flexible array member, or a union containing such a structure (possibly recursively), must not be a member of a structure or an element of an array.

NOTE: These uses are permitted by GCC as extensions.

Example 4 Initialization of Flexible Array Member

```
struct test_vla_init_nums_st { int numCC; int nums[]; };
static struct test_vla_init_nums_st test_vla_init_nums ={3, {1,2,3}
```

Improved C++ Demangler

The C++ demangler `c++filt` has been improved to use better algorithms, which are more robust and standard compliant. There are no changes to the external interface.

NOTE: `__cxa_demangle` has not changed in this release.

Deprecated and obsoleted options

The following options are deprecated:

- -AP

  In order to enable future runtime library versions, the `-AP` option is being deprecated and will be removed in a future version of aC++.

  The default on Integrity servers is `-AA`, so the sources should be ported to `-AA` mode.

  For more information, see http://www.hp.com/go/aCC and click **C++ runtime environments (-AA and -AP) on HP-UX** under **HP aC++ Resources**. Also see Frequently Asked Questions under C++ runtime environments (`-AA` and `-AP`) on HP-UX.

- -Aarm

  This enables some cfront semantics. But cfront was obsoleted in 1997. Code should long have been ported to Standard C++.

- +O[no]signedpointers

  This was only useful for rare cases on PA-RISC. The C++ Standard requires pointers to be treated as unsigned when comparing them.

The following options are deleted:

- +O[no]recovery

  HP does not recommend this option as it is not appropriate for certain types of applications.

- +O[no]libcalls

  This option was deprecated earlier. Note that `+Onolibcalls=func,...` is not being deprecated or deleted.

- +O[no]whole_program_mode

  This option is useful only on PA-RISC systems.

- +O[no]moveflops

  This option was deprecated earlier. It is approximately equivalent to `+Ofltacc=strict +Ofenvaccess`.

- +Ointeger_overflow=aggressive

  HP does not recommend this option as it is not always safe. Note that `+Ointeger_overflow=conservative` and moderate flavors remain.

- +O[no]ptrs_ansi

  This option is replaced by `+Otype_safety=ansi`.

- +O[no]ptrs_strongly_typed

  This option is replaced by `+Otype_safety=strong`.

- +O[no]loop_unroll_jam

  This option is replaced by `+O[no]loop_transform`.

- +O[no]loop_block

  This option is replaced by `+O[no]loop_transform`.

- +Oinline_budget=n

  This option was deprecated earlier. It is replaced by `+inline_level=n`.

- +O[no]report

  This option is useful only on PA-RISC systems.

- +df

  This is an option used on PA-RISC systems, and is replaced by `+Oprofile=use[:filename]`.
- +ES[no]lit

  This is an option used on PA-RISC systems, and is replaced by `+Olit=all` / `+Olit=none`.
- +M[d] / +m[d]

  These are options used in PA-RISC aC++, and are replaced by the longer `+Make[d]` and `+make[d]` forms to prevent confusion with the unrelated previous `+m` in HP C.
- +O[no]all

  This option was deprecated earlier.
- +O[no]conservative

  This option was deprecated earlier.
- +O[no]extern

  This option was deprecated earlier, and is replaced by the consistent `-B` binding family: `-Bextern[=list]`
- +O[no]volatile

  This option was deprecated earlier. The volatile keyword must be used as appropriate in the source.
- +A

  This is an option used on PA-RISC systems, and is an error on Integrity machines.

New features in version A.06.25

Version A.06.25 of the HP aC++ compiler supports the following new features:

- `C99` default C compilation mode (Changed)
- Full `-AA` default C++ compilation mode (Changed)
- `-Ax` option enables support for several `C++0x` extensions (New)
  - Scoped enumeration types
  - `static_assert`
  - Extended friend types
  - Rvalue references
  - Objectless references to non-static data members
  - Defaulted and deleted functions
  - In `C++0x` mode, `auto` is always a type specifier, not a storage-class specifier
  - Lambdas
- C99 features added to `C++0x` (New)
  - Mixed string literal concatenations
  - Rule changes related to determining the type of large literal constants

    - Variadic macros
    - `_Pragma` operator
- extern template
- Decimal Floating Point supported in C++ mode (New)
- `#pragma STDC FLOAT_CONST_DECIMAL64` (New)
- `#pragma omp task` (New)
- `#pragma omp taskwait` (New)
- Performance enhancements for `+O1` (Changed)
- Non-template static data members initialized outside the class no longer treated as constants in strict mode (Changed)
- Enhancements to allow code to run well on current platforms and future multi-core processors (New)
- New diagnostic messages (New)
- Improved diagnostic messages (Changed)
- Enhanced `+wendian` warnings (New/Changed)
- New run-time abort messages (New)

C99 default C compilation mode (Changed)

In this version of the HP C/aC++ compiler, the default C compilation mode has changed from `C89` to `C99`. So by default `C99` features are enabled, and the following commands are now equivalent:

```
cc
cc -Ae
cc -AC99
aCC -Ae
aCC -AC99
```

To retain the previous behavior, use the `-AC89` command-line option.

Errors that can happen with `C89` conforming code when using the `-AC99` command-line option include:

- enum out of range:

  ```
  error #2066: enumeration value is out of "int" range
  error #4041: enumeration value is out of "char" range
  ```

- Restriction on constant expressions:

  ```
  error #2057: this operator is not allowed in a constant expression
  error #2028: expression must have a constant value
  (Instead of warning #4045-D: non-constant initialization performed at runtime.)
  ```

- Non-static inline can't reference static:

  ```
  error #3031-D: an entity with internal linkage cannot be referenced
  within an inline function with external linkage
  ```

- Use of `inline` or `restrict` as variables/functions/types:

  ```
  error #2040: expected an identifier
  ```

Full -AA default C++ compilation mode (Changed)

In this version of the HP aC++ compiler, the default C++ compilation mode has changed to full `-AA`, which now additionally enables the option `-Wc,-ansi_for_scope,on`. This is being done to reduce porting efforts by adhering to the C++ Standard.

To retain the previous behavior, use the `-Wc,-ansi_for_scope,off` command-line option.

Errors and warnings (with unintended runtime results) that can happen with `-Wc,-ansi_for_scope,on` include:

- Loop index is no longer in scope after the for loop body:

```
error #2020: identifier "i" is undefined
```

- Reference to outer scope variable instead of loop index:

```
warning #2780-D: reference is to variable "i" (declared at line X)
  [Under old for-init scoping rules it would have been
    variable "i" (declared at line Y)]

With +wlint:
warning #3348-D: declaration hides variable "i" (declared at line X)
```

To catch issues resulting from this change in the C++ default, compile with the `+We2780` option to convert the warning `2780` to an error.

If you are already using the -Aa or -AA command-line option explicitly, then there would be no change in behavior.

-Ax option enables support for several C++0x extensions (New)

The new `-Ax` option turns on support for several extensions introduced by the working paper for the next C++ standard (called `C++0x`). The `-Ax` option is available only in C++ compilation mode and is binary compatible with the `-AA` compilation mode.

The following `C++0x` features are enabled by the `-Ax` option, and in additional compiler modes where indicated:

Scoped enumeration types

The compiler now supports scoped enumeration types (defined with the keyword sequence "`enum class`") and explicit underlying integer types for enumeration types.

Example:

```
enum class Primary { red, green, blue };

enum class Danger { green, yellow, red };
// No conflict on "red".

enum Code: unsigned char { yes, no, maybe };
// sizeof(Code) == 1

Primary p = Primary::red;
// Enum-qualifier is required to access
// scoped enumerator constants.

Code c = Code::maybe;
// Enum qualifier is allowed (but not required)
// for ordinary enumeration types.
```

static_assert

`static_assert(<integral-constant>, <string-literal>);`

Support is now included for `static_assert`, which generates a compile-time error if the integral constant is zero/false.

Example:

```
template<class T>
struct S {
static_assert(sizeof(T) > 4, "Type too small");
};
```

```
S<char> s1;  // Will trigger an error when the static_assert
             // declaration is instantiated.
```

Extended friend types

Support is now included for extended friend declaration syntax which allows for non-class names and non-elaborated class names to be declared as friend.

Example:

```
typedef struct S ST;
typedef int const IC;
class C {
friend S;           // Okay in C++0x mode
friend ST;          // Okay in C++0x mode
friend int;         // Okay in C++0x mode (but no effect)
friend IC;          // Okay in C++0x mode (but no effect)
friend int const;   // Error: cv-qualifiers not (directly) allowed
};
```

Rvalue references

Rvalue references are supported. This feature is enabled implicitly in `C++0x` mode. The existence of rvalue references allows the declaration of *move constructors*, which can be used to efficiently copy class objects by transferring the resources from a source object that is soon to be defunct to a destination object.

The syntax for declaring an rvalue-reference is: `typename&& rRef` (as opposed to the lvalue syntax: `typename& lRef`). The move constructor simply takes an rvalue or lvaue reference and returns an rvalue reference. This prevents invocation of the copy constructor; it therefore 'moves' around references without the need for copying the objects.

Note that new mangling forms have been added for rvalue references; this matches the Itanium ABI specification. The demangler `c++filt` has been updated to handle these new encodings.

Example:

```
int& ref = 10;            //Error; references must bind to lvalues.
int&& const_ref = 20;     //C++0x rvalue references bind to rvalues.
```

Objectless references to non-static data members

In the 2003 C++ Standard, the name of a non-static class data member could appear only in a member function of that class or one derived from it, in a member access expression (`x.m` or `p->m`), or to form a pointer to member (`&X::m`).

`C++0x` relaxes that restriction to allow such names as an unevaluated operand, such as `sizeof(X::m)`, `typeid(X::m)`, and `decltype(X::m)`. With this release, aC++ now accepts this usage in `C++0x` mode and non-strict C++98/03 mode; such references are transformed into a compiler-generated member-access expression using 0 cast to the appropriate type as the object pointer.

As an extension, these references are also accepted as subexpressions of unevaluated operands, such as `sizeof(X::arr[0])`, even in strict `C++0x` mode.

Defaulted and deleted functions

Deleted functions (`= delete;`) and defaulted special member functions (`= default;`) are now supported.

A deleted function is a function declaration that cannot be referenced.

For example:

```
int f(int) = delete;
short f(short);
int x = f(3);          // Error: selected function is deleted.
int y = f((short)3);   // Okay.
```

Special member functions that can be implicitly defined can instead be explicitly declared, but with a *default definition*. As the following example shows, the default definition can be specified inside or outside the enclosing class definition, but only if it appears inside the class can that class be a POD type (if all other constraints for POD types are fulfilled):

```
struct S { S(S const&) = default; };
struct T { T(T const&); };
T::T(T const&) = default;
```

In C++0x mode, auto is always a type specifier, not a storage-class specifier

The traditional meaning of `auto` as a storage class specifier is no longer enabled. `auto` is therefore always a type specifier in `C++0x` mode. It is used for automatic type deduction; when used in a declaration, it makes the type of the thing declared the same as the type of whatever initialized it. For example:

```
typedef int T;
int x;
void f() {
   auto T(x);  // This is now the declaration of a variable T
               // initialized with x (and hence of the same type as x).
               // Previously, it was the declaration of a variable x of type T.
   auto int y; // Now an error: Multiple type specifiers.}
```

Lambdas

Lambda functions are now supported.

A lambda expression is an anonymous function object containing a code snippet (for example, predicate logic) that can be specified very succinctly by the programmer without having to declare a function-local class; it is also referred to as *closure* because a lambda function typically encloses the runtime state or environment of the declaring function, being therefore able to access the local variables of the current function.

The way these variables are accessed can be controlled using either the '`&`' in a lambda capture list (`[&]`), in which case the variables are passed by reference; by using a '`=`' (`[=]`), whereby the variables are simply copied (analogous to call-by-value); or a combination of the two.

Example:

```
int main() {
    auto lambda = [] { printf("Lambda at work!"); };
    lambda();
}
```

C99 features added to C++0x (New)

The following `C99` features have been added to the `C++0x` standard and are available when compiling in `C++0x` mode:

Mixed string literal concatenations

In `C++0x` mode and in default C++ mode, aC++ now accepts string literal concatenations involving an ordinary `char` string literal and a wide string literal.

For example:

```
wchar_t *str1 = L"a" "b"; // Okay, same as L"ab".
wchar_t *str2 = "a" L"b"; // Okay, same as L"ab"
```

Such constructs were already accepted in `C99` and GNU modes.

Rule changes related to determining the type of large literal constants

As in `C99`, unsuffixed integer literals that do not fit in type `long`, but can fit in type `unsigned long`, are given type `long long` instead. This might cause certain expressions to differ in values in `C++0x` and default C++ compilation modes.

Example for +DD32:

```
bool b = 4000000000 > -1;  // b = true in C++0x mode.
// b = false in default C++ mode with long long enabled.
// (In that case, a warning is issued on the conversion of -1
// to an unsigned type.)
```

For +DD64, both modes are the same for that literal value.

Variadic macros

With this release, aC++ now accepts C99-style variadic macros in C++0x mode.

Like C99, the special identifier __VA_ARGS__ represents the set of all the arguments passed to the macro just as they are passed in the variable arguments mechanism for functions.

Example:

```
# define PRINTF(...) printf (__VA_ARGS__)
int main() {
char* from = "aCC";
PRINTF("Hello, World" " from %s!\n", from);
return 0;
}
// This will print:
// Hello, World from aCC!
```

_Pragma operator

The _Pragma operator (originally from C99 and already supported in GNU mode), is now supported in non-strict C++98/C++03 mode and in all C++0x modes.

The operator has the effect of expanding the pragma specified in the string (in double-quotes) in just the way a #pragma would.

Example:

```
_Pragma ("pack 1");
struct Packed {
char c;
int i;
};
int main () {
int iPackedSize = sizeof(Packed);
}
```

extern template

In C++0x mode (that is, with -Ax specified) and in non-strict C++98 mode, extern template can now be used to suppress the implicit instantiation of an entity.

Traditionally the C++ compiler instantiates a template in each translation unit where it finds a specification for the same. To suppress this (and thereby reduce the compilation times), C++0x now provides extern templates. This directs the compiler not to instantiate the template for this translation unit.

Example:

```
extern template class std::vector<SomeType>;
```

Decimal Floating Point supported in C++ mode (New)

With this release of the compiler on HP-UX 11.31 systems only, Decimal Floating Point is supported in C++ mode, along with the +decfp option.

To use Decimal Floating Point with C or C++ include files you must define __STDC_WANT_DEC_FP__. For C++ you should define __STDC_WANT_DEC_FP__ before including any headers, because some C++ headers might include <cmath.h> or <math.h> indirectly.

C++ support currently has a limitation of not being able to use the three new Decimal FP types in a throw expression or a `typeid` operator.

For more information on using Decimal FP, see the release notes section "Decimal floating-point arithmetic supported" under "New Features in the A.06.20 Release."

#pragma STDC FLOAT_CONST_DECIMAL64 (New)

```
#pragma STDC FLOAT_CONST_DECIMAL64 [ON | OFF | DEFAULT]
```

With this pragma set to OFF, unsuffixed floating-point constants are treated as having type `double`.

With this pragma set to ON, unsuffixed floating-point constants are treated as having type `_Decimal64`.

The pragma can occur in either of these two contexts:

- Outside external declarations

  In this case, the pragma takes effect from its occurrence until another `FLOAT_CONST_DECIMAL64` pragma is encountered, or until the end of the translation unit.

- Preceding all explicit declarations and statements inside a compound statement.

  In this case, the pragma takes effect from its occurrence until another `FLOAT_CONST_DECIMAL64` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement, the state for the pragma is restored to its condition just before the compound statement.

If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is OFF.

Note: In order to use this pragma safely with macros that are defined in `<math.h>` and `<float.h>`, the following Math Library patch must be installed: **PHSS_40540**.

For more information on using Decimal FP, see the release notes section “Decimal floating-point arithmetic supported” under "New Features in the A.06.20 Release."

#pragma omp task (New)

```
#pragma omp task [clause1, clause2, ...] new-line structured block
```

The OpenMP 3.0 directive `#pragma omp task` defines an explicit task.

[*clause1*, *clause2*, ...] indicates that the clauses are optional. There can be zero or more clauses.

clause may be one of the following:

if (scalar-expression)
untied
default (shared | none)
private (list)
firstprivate (list)
shared (list)

#pragma omp taskwait (New)

```
#pragma omp taskwait new-line
```

The OpenMP 3.0 directive `#pragma omp taskwait` specifies a wait on the completion of child tasks generated since the beginning of the current task.

Because the `taskwait` construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The `taskwait` directive may be placed only at a point where a base language statement is allowed. The `taskwait` directive may not be used in place of the statement following an `if`, `while`, `do`, `switch`, or `label`.

Performance enhancements for +O1 (Changed)

Various performance improvements have been made for `+O1`.

One specific improvement is that `+Odataprefetch=direct` is now supported at `+O1`, and is the default at `+O1`. This may help improve performance for code using loops.

Non-template static data members initialized outside the class no longer treated as constants in strict mode (Changed)

A static data member with a const integral type that is initialized inside the class can be used later in the program as a constant. However, if the data member is defined and initialized outside the class, the standard says it should not be considered a constant. Previous versions of aC++ failed to diagnose this issue, and treated them as constants.

Starting with this release, such non-template static data members initialized outside the class will no longer be treated as constants in strict mode.

In non-strict mode, the behavior remains unchanged for compatibility reasons.

Template static data members initialized outside the class are now always considered non-constant.

Enhancements to allow code to run well on current platforms and future multi-core processors (New)

Enhancements have been made to this release of the compiler to allow code to run well on current platforms and future multi-core processors.

New diagnostic messages (New)

The following new diagnostic messages are added to the compiler:

- Warning #3750

  Warning #3750 is now emitted for lines that end with a backslash followed by a space. For example:

```
int main(){
int something; \
int anotherthing;
}
line 2: warning #3750-D: "\" followed by white space is not a line splice
```

- Warning #2767

  In 64-bit C compilations, casting a pointer to an integral type of lower size now generates warning #2767. Previously, this was only a remark.

- Remark #3719

  In cases where the constructor initialization list is invoked in an order different from what the programmer apparently expects, a new remark #3719 is now emitted. For example:

```
class A{
public:
A(int c);
int i, j;
};

A::A(int c): i(++c), j(++c) { }
//remark #3719 - Init order is different than
//what is apparent from the code above
```

Improved diagnostic messages (Changed)

Several improvements have been made in the compiler diagnostic messages in terms of wording, better positioning of error constructs, and removal of redundant diagnostics. Some examples:

Warning #2069 is replaced with warning #2767 when appropriate.

Diagnostic #2069 is now suppressed in cases where it appeared in addition to diagnostic #2513.

Diagnostic #3056 is now suppressed in cases where it appeared in addition to diagnostic #2120.

Diagnostic #2028 is now suppressed in cases where it appeared in addition to diagnostic #2060.

Enhanced +wendian warnings (New/Changed)

When using the same code on platforms with different endian behavior, the results of many operations will vary between the two. Using the `+wendian` option with aCC will produce warnings for such statements that give differing results in different endian systems.

- Existing warnings 4291 and 4292 have been improved:

```
4291: endian porting: the read/write of the buffer may be endian dependent

4292: endian porting: the dereference of cast pointer may be endian dependent
```

- A new warning, 4364, is added to capture cases where a cast when dereferenced later can cause endian issues:

```
4364: endian porting: type cast is endian dependent
```

New runtime abort messages (New)

The following enhancements will be available in aC++ Runtime patches PHSS_40543 (11.23) and PHSS_40544 (11.31):

- The aC++ runtime library has been enhanced to provide the size of the first `std::bad_alloc` request. The string returned by `std::exception::what()` will now contain the following:

```
bad allocation exception thrown (0xxxxxxxxx bytes)
```

- The aC++ runtime library has been enhanced to provide the following information on the four existing Exception Handling abort messages. This occurs if the type being thrown is derived from `std::exception`.

  A message like the following `(with "what():")` is added:

```
aCC runtime: Uncaught exception of type "std::out_of_range".
aCC runtime: what(): /opt/aCC/include_std/string:1116:
    basic_string<>::at (size_type): argument value 10 out of range [0, 3)
```

New features in version A.06.20

Version A.06.20 of the HP aC++ compiler supports the following new features:

- Decimal floating-point arithmetic (HP-UX 11.31 only) (New)
- `+annotate=structs` (New)
- `+check=lock` (New)
- `+check=thread` (New)
- `+O[no]autopar` option now supported in C++ mode and implies `-mt` (New)
- `+O[no]dynopt` (New)
- `+inline_level num` (Enhanced)
- `-dumpversion` (New)
- `#include_next` (New)
- `#pragma diag_push` (New)
- `#pragma diag_pop` (New)
- `+Oinlinebudget` is deprecated

- In *next release*, default C compilation mode will change from `C89` to `C99`
- In *next release*, default C++ compilation mode changes to full `-AA`

Decimal floating-point arithmetic (HP-UX 11.31 only) (New)

On HP-UX 11.31 systems, support is now included for decimal floating-point arithmetic for C. This support follows the current draft revision of the IEEE 754 floating-point standard and ISO/IEC Technical Report 24732, *Extensions for the programming language C to support decimal floating-point arithmetic*. With decimal FP (unlike the usual binary FP), typical numerical strings can be represented exactly in the types, avoiding subtle input errors and confusion from inexact output. Therefore, decimal FP is WYSIWYG.

Decimal FP is designed particularly for financial applications, including banking, billing, tax calculation, currency exchange, and accounting.

A decimal FP representation is best thought of as a triple (*s*, *c*, *q*) composed of a sign (1 or –1), an integral coefficient, and a quantum exponent, representing $s * c * 10^q$. Therefore, 123. = (1, 123, 0) and 123.00 = (1, 12300, –2) are different representations, although they have the same numerical value and compare equal. Arithmetic operations are defined to preserve the position of the decimal point, much as hand-computation would. For example, 123.00 + 45.6 = 168.60 and 123.00 * 0.01 = 1.2300. These special quantum semantics facilitate exact fix-point calculation. For typical floating-point calculations, the quantum semantics can be ignored.

Decimal FP support includes the following:

- Three built-in decimal FP types:

  `_Decimal32` `_Decimal64` `_Decimal128`

  with 7, 16, and 34 decimal digits of precision, respectively.
- The usual built-in arithmetic operators for decimal FP operands: `+`, `–`, `*`, `/`, assignments, comparisons, and conversions with integer and binary FP types, all with correctly-rounded IEEE arithmetic. An operation may combine a decimal FP operand with an operand of a different decimal FP type or with an integer type. However, mixing operands of decimal and binary FP types is not allowed.
- 60 math functions for each decimal FP type. Function suffixes are `d32` for _Decimal32, `d64` for _Decimal64, and `d128` for _Decimal128. Included are:
  - Decimal FP versions of the `C99` math functions.
  - New functions to manage quantum exponents (for fixed-point calculation).
  - Routines to encode and decode data for either of the two standard encodings for decimal FP data. Details of the encodings are in the draft revision of IEEE 754, which refers to them as the "binary encoding", which the HP C/aC++ compiler uses, and the "decimal encoding". Both encodings provide exactly the same data, analogous to big endian and little endian encodings.
- Decimal FP I/O and string conversion. The decimal FP length modifiers for `printf()` and `scanf()` floating-point conversion specifiers (a, A, e, E, f, F, g, G) are `H` for _Decimal32, `D` for _Decimal64, and `DD` for _Decimal128. The decimal a, A specifiers for `printf()`, given no precision or sufficient precision, produce an exact quantum-preserving representation of the decimal FP value being converted.
- WDB debugger support for printing values of decimal FP types.
- Suffixes to designate decimal floating constants: `df` or `DF` for _Decimal32, `dd` or `DD` for _Decimal64, and `dl` or `DL` for _Decimal128. Note that an unsuffixed floating constant still has type double, regardless of the context. Explicit suffixing with a `d` or `D` to specify type double is also allowed.

- Decimal FP versions of macros in <float.h>, <math.h>, and <fenv.h>.
- Five rounding modes for decimal FP: to nearest with ties to even, to nearest with ties away from zero, upward, downward, and toward zero.
- Compiler option +decfp, which enables full decimal FP functionality according to the ISO/IEC C draft Technical Report: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1312.pdf
- Compiler option -fpevaldec=precision, which specifies the minimum precision to use for decimal floating-point expression evaluation. The possible values for precision are _Decimal32, _Decimal64, and _Decimal128. This option does not affect the precision of parameters, return types, or assignments. The default is -fpevaldec=_Decimal32.
- Type-generic functions in <tgmath.h> that take on the type of decimal FP (or binary FP) arguments.
- Our Decimal FP support incorporates code from the Intel Decimal Floating-Point Math Library.

To use decimal FP:

- Install the following on an HP-UX 11i V3 (11.31) Integrity system:
  - core Math patch PHSS_38137
  - math manpage patch PHCO_38388
  - libc patch PHCO_38048
  - libcEnhancement package (http://docs.hp.com/en/5992-3373/ch10s12.html)
  - libc man page patch PHCO_37128 (for printf and scanf man pages)
- To use any decimal FP functionality (even built-in operators), add the +decfp option to the compile and link lines.
- To use the decimal FP functionality in <math.h>, <float.h>, <fenv.h>, or <tgmath.h>, define __STDC_WANT_DEC_FP__ before including the header.
- To use strtod32, strtod64, or strtod128, include <strtodec.h>. (These interfaces are not declared in <stdlib.h>, as specified in the ISO/IEC C draft Technical Report.)
- Also see the "HP-UX floating-point guide for HP Integrity servers" at http://www.hp.com/go/fp.

Decimal Floating-Point Example

```
/*****************************************************************************
 *
 * This program is intended to illustrate how C decimal FP might be
 * used in a commercial billing program.
 *
 * The program reads input data from a file, interprets each datum
 * as a number of service minutes, converts to hours (rounding up to
 * the nearest tenth of an hour), and multiplies by a billing rate
 * (rounding to the nearest cent with halfway cases rounding away
 * from zero). For each input1, the program writes the billing amount
 * as an ASCII character string to a file. Along the way, it
 * accumulates  sums of service minutes, hours billed, and amounts
 *  billed, and at the end writes the total sums to stdout.
 *
 * Program constants determine the input and output filenames and
 * whether the input is ASCII, BID ("binary" encoding for decimal FP),
 * or DPD ("decimal" encoding for decimal FP). The HP-UX
 * implementation uses BID encoding for its decimal FP types.
```

```
 *
 ************************************************************************/

const char * ifilename = "biller.in";              // input file
const char * ofilename = "biller.out";           // output file
const enum { ascii, bid, dpd } encoding = ascii; // input encoding

const _Decimal64 rate = 149.95DD;           // hourly billing rate

#define __STDC_WANT_DEC_FP__

#include <stdio.h>
#include <std.lib.h>
#include <math.h>
#include <fenv.h>
#include <strtodec.h>
#include <assert.h>

#pragma STDC FENV_ACCESS ON

int main() {
    unsigned int n;                          // number of inputs
    _Decimal64 m;                          // number of minutes
    _Decimal64 h;                           // number of hours
    _Decimal64 b;                           // amount billed
    _Decimal64 sumM, sumH, sumB;    // sums
    const _Decimal64 onert = 0.0DD;     // for rounding to tenths
    const _Decimal64 twort = 0.00DD;    // for rounding to hundredths
    FILE *inp=NULL;                          // input stream
    FILE *outp=NULL;                        // output stream
    int r, s;                                // fread, scanf returns
    _Binaryencoding64 be;      // for binary-encoded (bid) input
    _Decimalencoding64 de;    // for decimal-encoded (dpd) input

    // open input file
    if (encoding == ascii)
        inp = fopen(ifilename, "r");
    else
        inp = fopen(ifilename, "rb");
    if (inp == NULL) {
        fprintf(stderr, "FAILURE: could not open %s\n", ifilename);
        exit(EXIT_FAILURE);
    }

    // open output file
    outp = fopen(ofilename, "w");
    if (outp == NULL) {
        fprintf(stderr, "FAILURE: could not open %s\n", ofilename);
        exit(EXIT_FAILURE);
    }

// initialize sums
    sumM = 0.DD;
    sumH = 0.0DD;
    sumB = 0.00DD;

    // main loop
    for (n=0; ; n++) {
        // read a number
        if (encoding == ascii) {
            s = fscanf(inp, "%De", &m);
            if (s == 0) {
                fprintf(stderr, "FAILURE: on input from %s\n", ifilename);
                fclose(inp);
                fclose(outp);
```

```
                exit(EXIT_FAILURE);
            }
            if (s == EOF) break;
        }
        else if (encoding == bid) {
               r = (int) fread(&be, sizeof(_Binaryencoding64), 1, inp);
            if (r != 1) break;
            m = _decodebinary64(be); // decode bid
        }
              else /* encoding == dpd */ {
                 r = (int) fread(&de, sizeof(_Decimalencoding64), 1, inp);
            if (r != 1) break;
            m = _decodedecimal64(de); // decode dpd
        }

// compute hours billed
        fe_dec_setround(FE_DEC_UPWARD);
        h = m / 60;
        h = quantized64(h, onert); // round to tenths

        // compute billing amount
        fe_dec_setround(FE_DEC_TONEARESTFROMZERO);
        b = rate * h;
        b = quantized64(b, twort); // round to hundredths

        // restore decimal rounding mode to default
        fe_dec_setround(FE_DEC_TONEAREST);

              // print billing amount to file
fprintf(outp, "%Da\n", b);

        // update sums
              sumM += m;
              sumH += h;
        sumB += b;
    } // main loop

    // close files
    fclose(inp);
    fclose(outp);

    // confirm expected quantum in sums
    assert(samequantumd64(sumH, onert));
    assert(samequantumd64(sumB, twort));

    // print summary
    printf("SUMMARY:\n");
    printf("  number of calls      = %d\n", n);
    printf("  total minutes        = %Da\n", sumM);
    printf("  total hours billed   = %Da\n", sumH);
    printf("  total amount billed  = %Da\n", sumB);

    return 0;
}
```

Notes:

- The quantize functions do the work of rounding to the desired number of places to the right of the decimal point. They return the value of their first argument represented with the quantum exponent of their second argument, rounding if necessary.
- Mixing decimal FP and integer operands, as in the expression *m* / 60, is allowed. However, the expression *m* / 60.0, which mixes decimal and binary FP, would cause an error.

- On the HP-UX implementation, BID data could be read directly into the decimal FP variable *m*, without the need of decoding. However, that code would not be portable to systems that use DPD encoding for their decimal FP types.
- Restoring the default rounding has no effect in this program, except perhaps if the ASCII input had more than 16 decimal digits, but doing so is a good programming practice.

+annotate=structs (New)

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

+check=lock (New)

The new runtime `+check=lock` option enables the checking of locking discipline violations; for example, whether appropriate locks are held when threads access shared data in applications using Posix threads. Refer the online programmer's guide for additional detail and code example. Note that `+check=lock` is not enabled by `+check=all`.

+check=thread (New)

The new runtime `+check=thread` option enables the batch-mode thread-debugging features of HP WDB. This feature requires HP WDB 5.9 or later. The following thread-related conditions can be detected with `+check=thread`:

- The thread attempts to acquire a nonrecursive mutex that it currently holds.
- The thread attempts to unlock a mutex or a read-write lock that it has not acquired.
- The thread waits (blocked) on a mutex or read-write lock that is held by a thread with a different scheduling policy.
- Different threads non-concurrently wait on the same condition variable, but with different associated mutexes.
- The threads terminate execution without unlocking the associated mutexes or read-write locks.
- The thread waits on a condition variable for which the associated mutex is not locked.
- The thread terminates execution, and the resources associated with the terminated thread continue to exist in the application because the thread has not been joined or detached.
- The thread uses more than the specified percentage of the stack allocated to the thread.

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

+O[no]autopar now supported in C++ Mode (New)

The `+O[no]autopar` option introduced in version A.06.15 of the compiler was supported only when compiling C or Fortran files. Version A.06.20 also supports this option when compiling C++ files. In addition, specifying `+Oautopar` now implies the `-mt` option.

+O[no]dynopt (HP-UX 11.31 only) (New)

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

+inline_level *num* (Enhanced)

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

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

-dumpversion (New)

The `-dumpversion` option displays the simple version number of the compiler, such as A.06.20. Compare with the -V option, which displays more verbose version information.

#include_next (New)

The `#include_next` preprocessor directive is similar to the `#include` directive, but tells the preprocessor to continue the include-file search beyond the current directory, and include the subsequent instance found in the file-search path.

#pragma diag_push (New)

This scoped pragma saves the current severity state of all diagnostics. Subsequent uses of pragmas that modify the severity of a given diagnostic will be in effect within the scope of the `diag_push` pragma. The effective scope ends with a corresponding `#pragma diag_pop`. You can specify a `#pragma diag_push` within the scope of another `#pragma diag_push`, which results in a new saved severity state and a new effective scope. A compilation unit should have an equal number of `#pragma diag_push` and `#pragma diag_pop` uses.

#pragma diag_pop (New)

This pragma restores the severities of all diagnostics to the state prior to the last `#pragma diag_push`. A compilation unit should have an equal number of `#pragma diag_push` and `#pragma diag_pop` uses.

+Oinlinebudget is deprecated (Change)

The `+Oinlinebudget` option is deprecated in this release and will not be supported in future releases. Use `+inline_level`.

In next release, default C compilation mode will change from C89 to C99

In the next version of the HP C/aC++ compiler, the default C compilation mode will change from `C89` to `C99`. So `cc -Ae` or `aCC -Ae` will be the same as `-AC99`, and `C99` features will be enabled.

Users can prepare for this transition by adding `-AC99` to their build options and addressing any issues. Errors that can happen with `-AC99` include:

- enum out of range:

```
error #2066: enumeration value is out of "int" range
error #4041: enumeration value is out of "char" range
```

- Restriction on constant expressions:

```
error #2057: this operator is not allowed in a constant expression
error #2028: expression must have a constant value
Was this:
warning #4045-D: non-constant initialization performed at runtime
```

- Non-static inline can't reference static:

```
error #3031-D: an entity with internal linkage cannot be referenced
within an inline function with external linkage
```

- Use of `inline` or `restrict` as variables/functions/types. New diagnostics will be added to warn users about the use of `inline` and `restrict` in this release (A.06.20). For example:

```
4347 %s is a keyword in the C99 C Standard, and its usage as an
identifier will  cause an error in C99 mode
```

In next release, default C++ compilation mode will change to full -AA

In the next version of the HP C/aC++ compiler, the default C++ compilation mode will change from almost `-AA` to `-AA`. This will enable `-Wc,-ansi_for_scope,on`. This is being done to reduce porting efforts by meeting the C++ Standard.

Users can prepare for this transition by adding `-AA` to their build options and addressing any issues. Errors and warnings (with unintended runtime results) that can happen with `-Wc,-ansi_for_scope,on` include:

- Loop index is no longer in scope after the for loop body:

```
error #2020: identifier "i" is undefined
```

- Reference to outer scope variable instead of loop index:

```
  warning #2780-D: reference is to variable "i" (declared at line X) --
  under old for-init scoping rules it would have been variable "i"
  (declared at line Y)
With +wlint:
warning #3348-D: declaration hides variable "i" (declared at line X)
```

By also compiling with `+We2780`, the two errors, 2020 and 2780, should catch any issues resulting from the change in the C++ default.

If you are already using `-Aa`, this also enables the new default.

New features in version A.06.15

Version A.06.15 of the HP aC++ compiler supports the following new features:

- `printf`, `fprintf` Optimization (New)
- `+Wmacro` Option (New)
- `+Wcontext_limit` Option (New)
- `+wperfadvice` Option (New)
- `+Wv` Option (New)
- `+wlock` Option (New)
- `+O[no]autopar` Option (New)
- `+O[no]loop_block` Option (New)
- `+O[no]loop_unroll_jam` (Default Changed)
- `+Olit=all` is the new default for HP C (Change)

- `+macro_debug=` Option (New)
- `+pathtrace` Option (New)
- `+check` Suboptions (New)
- `-Bhidden_def` Option (New)
- `-dM` Option (New)
- `#pragma OPT_LEVEL INITIAL` (New)
- `#pragma OPTIMIZE` (Deprecated)
- `#pragma [NO] INLINE` (New for C++ mode)
- `_Asm_ld, _Asm_ldf, _Asm_st, _Asm_stf` intrinsics added (New)
- Debugging code compiled with opt levels above `+O1` now supported (New)
- `__attribute__ ((visibility("default"|"protected"|"hidden"))) (New)`
- `__attribute__ ((warn_unused_result)) (New)`
- Change in treatment of cv-qualified assignment operators

printf, fprintf optimization (New)

For optimization level`+O2`or above, `printf` and `fprintf` calls are now optimized into `fputc` or `fputs` calls, in trivial cases. This optimization will only occur when both the following are true

- A return value is not used for `printf` and `fprintf`. For example, use `(void)fprintf(...);`
- The functions `printf`, `fprintf`, `fputs`, and `fputc` are declared in `<stdio.h>` or `<cstdio>`.

To disable this feature for `+O2`and above, use: `+Onolibcalls="fprintf"` (or `"printf"`).

To have the compiler list the lines where the optimization was performed, use `+Oinfo`.

+Wmacro option (New)

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

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

+Wcontext_limit option (New)

`+Wcontext_limit=num`

This option limits the number of instantiation contexts output by the compiler for diagnostics involving template instantiations. At most `num` outermost contexts and `num` innermost contexts are shown. If there are more than 2 * `num` relevant contexts, the additional contexts are omitted. Omitted contexts are replaced by a single line separating the outermost `num` contexts from the innermost `num` contexts, and indicating the number of contexts omitted. The default value for `num` is 5. A value of 0 removes the limit.

+wperfadvice option (New)

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

This option enables performance advisory messages. The optional level 1, 2, 3, or 4 controls how verbosely the performance advisory messages are emitted. The higher the level, the more messages

are generated. Level 1 emits only the most important messages, while level 4 emits all the messages. If the optional level is not specified, it defaults to 2.

+Wv option (New)

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

The `new +Wv`option displays the description for diagnostic message numbers `d1,d2,...,dn`.

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

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

+wlock option (New)

`+wlock`

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

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

+O[no]autopar option (New)

`+O[no]autopar`

This release adds support on the Itanium platform for a new optimization -auto-parallelization - which is enabled by adding the `+Oautopar`option to the command-line. This optimization allows applications to exploit otherwise idle resources on multicore or multiprocessor systems by automatically transforming serial loops into multithreaded parallel code.

When the `+Oautopar`option is used at optimization levels `+O3`and above, the compiler will automatically parallelize those loops that are deemed safe and profitable by the loop transformer.

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

Automatic parallelization can be combined with manual parallelization through the use of OpenMP directives and the `+Oopenmp` option. When both `+Oopenmp` and `+Oautopar` options are specified, the compiler honors the OpenMP directives first, and then looks for loops that have not been parallelized manually with OpenMP directives. For these loops, the compiler automatically parallelizes each loop that is both safe and likely to have improved performance when executed in parallel.

Programs compiled with the `+Oautopar` option require the `libcps`, `libomp`, and `libpthreads` runtime support libraries to be present at both compilation and runtime. When linking with the HP-UX B.11.61 linker (patch PHSS_36342 or PHSS_36349), compiling with the `+Oautopar`option causes them to be automatically included. Older linkers require those libraries to be specified explicitly or by compiling with `+Oopenmp`.

At present, `+Oautopar`is only supported when compiling C or Fortran files, and not C++ files. If you use `+Oautopar` with C or Fortran code in a mixed-language application that also includes C++ files, you must use `-mt` when compiling and linking the C++ files, similar to the current requirements for `+Oopenmp`. Please refer the documentation of the aCC compiler's`-mt` option for additional information and restrictions.

+O[no]loop_block option (New)

`+O[no]loop_block`

Loop blocking is a combination of strip mining and interchange that improves data cache locality. It is provided primarily to deal with nested loops that manipulate arrays that are too large to fit into the data cache. Under certain circumstances, loop blocking allows reuse of these arrays by transforming the loops that manipulate them so that they manipulate strips of the arrays that fit into the cache.

At optimization levels 3 and 4, using `+Oloop_block` (the default) allows automatic loop blocking. Specifying `+Onoloop_block` disables loop blocking.

+O[no]loop_unroll_jam (Default Change)

At optimization levels 3 and 4, the default for this option has changed from `+Onoloop_unroll_jam` to `+Oloop_unroll_jam`, which allows automatic loop unroll-and-jam.

+Olit=all (Default change for HP C)

The default in C mode of the compiler is now `+Olit=all`, placing string constants in read-only memory by default. This change over previous versions of the compiler (which defaulted to `+Olit=const`) improves performance and is in keeping with similar changes in the industry. Note that this may cause a runtime signal 11 if an attempt is made to modify string literals.

To force the string to be read/write, you can change the source as follows:

```
static CHAR forall_00[] = "FORALL.CHARTYPE ";
static CHAR forall_01[] = "FORALL.COMPLEXTYPE ";
...
static CHAR *foralltemptypename[LASTINTRINTYPE-FIRSTINTRINTYPE+1] = {
   forall_00, forall_01, ...

Using an array will force the string to be read/write.

Other simpler cases can be handled as:
changing to array will work in most cases:
   char *temp = "abcd";
   char temp[] = "abcd";

If temp is used as a pointer:
   static char temp_arr[] = "abcd";
   char *temp = temp_arr;
```

+macro_debug= (New)

The +macro_debug option controls the emission of macro debug information into the object file:

`+macro_debug={ref|all|none}`

Set +macro_debug to one of the following required values:

`ref` Emits debug information only for referenced macros. This is the default for `-g`, `-g1`, or `-g0`.

`all` Emits debug information for all macros. This option can cause a significant increase in object file size.

`none` Does not emit any macro debug information.

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

`+nomacro_debug` suppresses emission of macro debug information into the object file.

+pathtrace (New)

`+pathtrace[=kind]`

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

Usage:

The defined values for kind are:

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

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

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

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

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

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

+check suboptions (New)

The following new suboptions have been added to the +check option:

```
+check=globals
+check=truncate[:explicit|:implicit]
```

The following new suboptions have been added to the +check=bounds option:

```
+check=bounds:array
+check=bounds:pointer
+check=bounds:all
+check=bounds:none
```

Descriptions:

- +check=globals

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

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

The `+check=globals` option is implied by `+check=all`.

- `+check=truncate[:explicit|:implicit]`

  The `+check=truncate` option enables runtime checks to detect data loss in assignment when integral values are truncated. Data loss occurs if the truncated bits are not all the same as the left most non-truncated bit for signed type, or not all zero for unsigned type. Programs may contain intentional truncation at runtime, such as when obtaining a hash value from a pointer or integer. To avoid runtime failures on these truncations, the user can explicitly mask off the value: `ch = (int_val & 0xff);`

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

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

  `+check=truncate` (with no suboptions) turns on runtime checks for both explicit cast and implicit conversion truncation

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

- `+check=bounds[:array|pointer|all|none]`

  The `+check=bounds` option has been enhanced to provide the option of checking for out-of-bound references to buffers through pointer access as well as to array variables. You can specify one of the following `+check=bounds` suboptions:

  `array` Enables check for out-of-bounds references to array variables.

  `pointer` Enables check for out-of-bounds references to buffers through pointer access. The buffer could be a heap object, global variable, or local variable. This suboption also checks out-of-bounds access through common libc function calls such as `strcpy`, `strcat`, `memset`, and so on. This check can create significant run-time performance overhead.

  `all` Enables out-of-bounds checks for both arrays and pointers. This is equal to `+check=bounds:array +check=bounds:pointer`.

  `none` Disables out-of-bounds checks..

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

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

-Bhidden_def (New)

`-Bhidden_def`

This option is the same as `-Bhidden`, but only locally defined (non-tentative) symbols, without `__declspec(dllexport)`, are assigned the hidden export class.

As with any `-B` option, `-Bhidden_def` can be overridden by subsequent `-B` options on the command line or any binding pragmas in the source.

-dM (New)

`-dM`

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

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

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

#pragma OPT_LEVEL INITIAL (New)

```
#pragma OPT_LEVEL INITIAL
```

#pragma `OPT_LEVEL` has been enhanced to accept an additional keyword, `INITIAL`.

When used with a numeric argument, the `OPT_LEVEL` pragma sets the optimization level to 0, 1, 2, 3, or 4.

The `INITIAL` argument causes the optimization level in effect at the start of the compilation, whether by default or specified on the command line, to be restored.

#pragma OPTIMIZE (Deprecated)

As of this release of the compiler, `#pragma OPTIMIZE` is deprecated. Use `#pragma OPT_LEVEL` instead.

#pragma [NO]INLINE (New for C++ Mode)

Previously, `#pragma [NO]INLINE`was supported only on HP C and aC++ C-mode. With this release, `#pragma [NO]INLINE`is now supported in C++ mode as well.

_Asm_ld, _Asm_ldf, _Asm_st, _Asm_stf Intrinsics (New)

The following new assembly intrinsics have been added to the compiler:

```
_Asm_ld _Asm_st
```

```
_Asm_ldf _Asm_stf
```

For more information, see the "Inline assembly for Itanium(R)-based HP-UX" link off http://www.hp.com/go/aCC.

Debugging Code Compiled with Opt Levels above +O1 Is Supported

Debugging code compiled with optimization levels above `+O1`is now supported, as described in section 14.22 ("Debugging optimized code") of the Debugging with GDB manual under the "Documentation" link at: `http://www.hp.com/go/wdb`.

__attribute__ ((visibility("default"|"protected"|"hidden"))) Added (New)

The visibility attributes "default", "protected", and "hidden", are equivalent to the options `-Bdefault`, `-Bprotected`, and `-Bhidden`, and the pragmas `DEFAULT_BINDING`, `EXTERN`, and `HIDDEN`, respectively.

__attribute__ ((warn_unused_result)) Added (New)

The `warn_unused_result` attribute tells the compiler to emit a warning if a caller of a function with this attribute does not use its return value. This is useful for functions where not checking the result can be a security problem or always a program bug, as with `realloc()`.

Change in treatment of cv-qualified assignment operators

A user-defined assignment operator that is a `const` or volatile member function, is not considered a copy assignment operator, and the compiler will not suppress the declaration of an implicit copy assignment operator. As a result, the implicitly generated version might be chosen during overload resolution, if it is a better match. For example:

```
struct S {
   S();
   const S& operator=(const S&) const;
};
```

```
int main() {
   S s1, s2;
   s1 = s2;         // calls implicitly generated operator
                    // previously used to call the user-defined one
   const S s3;
   s3 = s2;        // still calls the user-defined operator
   return 0;
}"
```

New features in version A.06.12

Version A.06.12 of the HP aC++ compiler supports the following new features:

- `+Ofast` (`-fast`) and `+Ofaster` Options
- Interaction between `+Oinit_check` and `+check=uninit`

+Ofast (-fast) and +Ofaster options

The `+Ofast` and `+Ofaster` options now cause the data and text page size to be 1MB instead of 4MB, for increased performance The `+Ofast` option no longer implies the unsafe optimizations `+Ointeger_overflow=aggressive` and `+Olibcalls`.

Interaction between +Oinit_check and +check=uninit

When `+Oinit_check` and `+check=uninit` are used together, warnings will still be generated at compile time for potentially uninitialized variables, but they will not be automatically initialized by the compiler, so that lack of initialization can be detected at runtime.

New features in version A.06.10

Version A.06.10 of the HP aC++ compiler provides complete source and binary compatibility (including OpenMP features) with earlier versions of the A.06.xx family.

HP aC++ compiler version A.06.10 supports the following new features:

- HP Code Advisor
- +cond_rodata Option (Obsoleted)
- +[no]dep_name Option (New)
- +expand_types_in_diag Option (New)
- +FPmode (Enhanced)
- +Ointeger_overflow Option (Default Changed)
- +Onolibcalls= Option (New)
- +wendian Option (New)
- +wlint Option (Enhanced)
- +wsecurity= Option (New)
- System-wide option configuration (New)
- [NO]PTRS_TO_GLOBALS Pragma (New)
- -AA -D_HP_NONSTD_FAST_IOSTREAM performance Macro (New)
- New function attributes
- Improved Diagnostics
- C++ Standard Library

HP Code Advisor

This release introduces a new tool "HP Code Advisor", that can be used for detecting various programmer errors in C/C++ source code. Use this tool to identify potential coding errors, porting issues and security errors.

The HP Code Advisor is being made available on both HP-UX PA and Integrity servers. It leverages the advanced analysis capabilities of the C/C++ compilers for the Integrity servers.

Use "`/opt/cadvise/bin/cadvise`" to invoke the tool. A brief description is available with the `-help` option.

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

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

+cond_rodata option (Obsoleted)

The compiler now allocates more data in read-only memory. This option is now obsolete. The compiler always behaves as if it had been specified. (Note: a previous posting of these notes contained a typographic error. +cond_rodata was mispelled.)

+[no]dep_name option (New)

+[no]dep_name enforces strict dependent name lookup rules in templates. The default is +nodep_name.

+expand_types_in_diag option (New)

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

+FPmode option (Enhanced)

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

+Ointeger_overflow (Default Changed)

The default setting for all optimization settings is now "moderate".`+Ointeger_overflow=moderate`

+Onolibcalls= option (New)

`+Onolibcalls=` allows you to turn off optimization for listed fun ctions.

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

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

+wendian option (New)

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

+wlint option (Enhanced)

New diagnostic features have been added for this option. New checks have been added for memory leaks, out-of-scope memory access, null pointer dereference, and out-of-bounds access.

+wsecurity= option (Enhanced)

The `+wsecurity` option now optionally takes a argument to control how verbosely the security messages are emitted:

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

The default level is 2.

System-wide option configuration

There is a new ability to configure compiler options on a system-wide basis. The compiler now reads the configuration file:

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

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

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

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

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

The final option ordering would be:

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

`<envvar-options-2><file-options-2>`

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

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

[NO]PTRS_TO_GLOBALS pragma

`#pragma [NO]PTRS_TO_GLOBALS <name>`

This pragma aids alias analysis. It must be specified at global scope and immediately precede the declaration of the variable or entry named. The pragma tells the optimizer whether the global variable or entry "name" is accessed [is not accessed] through pointers.

-AA -D_HP_NONSTD_FAST_IOSTREAM performance improvement macro

A new performance improvement preprocessor macro, `_HP_NONSTD_FAST_IOSTREAM` can be used to improve `-AA iostream` performance.

This macro enables the following non-standard features:

- Sets `std::ios_base::sync_with_stdio(false)`, which disables the default synchronization with `stdio`.
- Sets `std::cin.tie(0)`, which unties the `cin` from other streams.
- Replaces all occurrences of "`std::endl`" with "`\n`".

Enabling this macro might result in noticeable performance improvement if the application uses `iostreams` often.

NOTE: Do not enable the macro in any of the following cases:

- If the application assumes a C++ stream to be in sync with a C stream.
- If the application depends on stream flushing behavior with `endl`.
- If the user uses "`std::cout.unsetf(ios::unitbuf)`" to unit buffer the output stream.

New function attributes

`__attribute__` is a language feature that allows you to add attributes to functions. The capabilities are similar to those of #pragma. It is more integrated into the language syntax than pragmas and its placement in the source code depends on the construct to which the attribute is being applied.

The new function attributes in this release are "`malloc`", "`non-exposing`", "`noreturn`", and "`format`". Refer the Pragmas and Attributes online help topic for additional information.

Improved diagnostics

Diagnostic messages now include more context to aid in tracing their root causes, including an improved template instantiation traceback.

C++ Standard library change

Technical Corrigenda 1 has changed the STL function `make_pair` to take their arguments by value instead of const reference. This change brings the HP library into compliance if the enabling macro `-D__HP_TC1_MAKE_PAIR` is specified at compile time. For binary compatibility reasons, the default behavior is unchanged.

Earlier versions

Information on features introduced in versions before A.06.12 can be found in Release Notes for each compiler version located at the HP documentation website: **www.docs.hp.com**

4 Installation information

Read this entire document and any other release notes or readme files you may have before you begin an installation.

To install your software, run the SD-UX `swinstall` command. This invokes a user interface that will lead you through the installation.

For more information about installation procedures and related issues, see *Managing HP-UX Software with SD-UX* and other README, installation, and upgrade documentation provided or described in your HP-UX 11.x operating system package.

Depending on your environment, you may also need documentation for other parts of your system, such as networking, system security, and windowing.

Hardware requirements

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

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

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

HP recommends not reducing the maxfiles setting below the default value of 2048.

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

HP aC++ requires approximately 150 MB for the files in

`/opt/aCC`

The other components require approximately:

- 40 MB for WDB
- 110 MB for u2comp/be
- 95 MB for HP Caliper
- 205 MB for HP Code Advisor

For more precise sizes, use the command:

```
/usr/sbin/swlist -a size B9007AA
```

5 Compatibility information

Maintaining binary compatibility is a key release requirement for new versions of HP aC++. The compiler has maintained the same object model and calling convention and remains compatible with the HP-UX runtime in the code that it generates as well as its intrinsic runtime library (`libCsup`) across the various releases of HP aC++ and its run-time patch stream.

aC++ standard conformance and compatibility changes

The following document provides the deattails of the differences that you can experience when upgrading from HP aC++ Version 5.x to HP aC++ Version 6.0:

http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801/?ciid=2708d7c682f02110d7c682f02110275d6e10RCRD

Caliper compatibility

For binaries containing objects generated with version A.05.38 of the compiler, it is recommended that you use Caliper version 2.1 for performance measurements and PBO. You can download Caliper 2.1 from www.hp.com/go/caliper.

WDB compatibility

It is recommended that you use the latest version of the WDB debugger. You can obtain the latest information on the debugger at Developer and Solution Partner page of the HP web site: www.hp.com/go/wdb.

Difference in class size when compiling in 32-bit and 64-bit mode

The size of a class containing any virtual functions varies when compiled in 32-bit mode versus 64-bit mode. The difference in size is caused by the virtual table pointer (a pointer to an internal compiler table) in the class object. (The pointer is created for any class containing one or more virtual functions.)

When compiling the following example in 32-bit mode, the output is 8. In 64-bit mode, the output is 16.

```
extern "C" int printf(const char *,...);
   class A {
   int a;
public:
   virtual void foo();  //virtual function foo, part of class A
   };
void A::foo() {
   return;
   }
int main() {
   printf("%d\n",sizeof(A));
   }
```

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

The compiler lists Errors, Future Errors, and Warnings. Expect to see more warnings, errors and future errors reported in your code, many related to standards based syntax. For more complete information, see the *HP aC++ Transition Guide* at http://www.hp.com/go/aCC.

General programming information and support questions

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

The information about subscribing to the cxx-dev list server can be obtained from:

http://www.hp.com/go/aCC

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)

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`.

For specific support questions, contact your HP support representative.

For generic C++ questions, see documents and URLs listed in the HP aC++ Programmer's Guide, Information Map.

6 Known problems and workarounds

This section describes known problems and workarounds. Customers on support can use the product number to assist them in finding SSB and SRB reports for HP aC++ or HPC. The product number you can search for is B3910BA.

To verify the product number and version for your HP aC++ or HP C compiler, execute the following HP-UX commands:

```
what /opt/aCC/bin/aCC
what /opt/aCC/lbin/ecom
```

Obsolete LANG-STARTUP files

As of HP aC++ version A.06.15, the LANG-STARTUP fileset is obsoleted because its only two files are no longer being delivered:

```
/opt/langtools/lib/hpux32/fastmem.o
/opt/langtools/lib/hpux32/effmem.o
```

codecvt_byname facet needed for C locale conversions

You must use the `codecvt_byname` facet to use C locale conversions.

To do locale-specific conversions, the `codecvt_byname` facet must be installed in the locale:

```
locale::global(locale("ja_JP.eucJP"));
typedef std::codecvt_byname<wchar_t, char, std::mbstate_t> ucs2utf;
std::wcout.imbue(std::locale(std::locale(),new ucs2utf("ja_JP.eucJP")));
```

The call to `imbue` is required because otherwise the default behavior of `std:wcout` is not defined with respect to which codevt to use.

Using +check= options and running on test and deployment systems

Using various `+check=` options requires a fairly recent version of wdb (`+check=malloc` and `+check=bounds:pointer`) and `/opt/langtools/lib/hpux##/librtc.so.1` where the application is run. The debugger and library is automatically updated on the machine only where the compiler is installed. This may not be true where the application is executed.

Older versions of wdb may cause hangs. Older versions of`librtc.so.1`may produce runtime unsats.

GPREL22 relocation error

If a variable is declared as extern non-array in one module and then defined as an array in another, a linker error may occur:

```
Definition: foo.c:uint64_t variable[SIZE];
Reference: bar.c:extern uint64_t variable;

ld: The value 0xXXX does not fit when applying the relocation GPREL22 for
symbol "variable" at offset 0xYYY in section index ZZZ of file bar.o.
```

Workaround: The declaration should be changed to:

```
extern uint64_t variable[];
```

This error can also occur in assembly code if items <= 8 bytes are put into `.data/.bss` instead of `.sdata/.sbss`.

Workaround (assembler): For "small" variables defined in assembly, change the section name from `.bss` to `.sbss` or `.data` to `.sdata`:

```
.section .sdata = "asw", "progbits"
       .align 8
gggggggg::     data4   0x000003e7
```

Object files generated at +O4 or -ipo

Object files generated by the compiler at optimization level 4, called intermediate object files, are intended to be temporary files. These object files contain an intermediate representation of the user code in a format that is designed for advanced optimizations. The size of these intermediate object files may be 3 to 10 times as large as normal object files. Hewlett-Packard reserves the right to change the format of these files without notice in any compiler release or patch. Use of intermediate files must be limited to the compiler that created them. For the same reason, intermediate object files should not be included in archived libraries that might be used by different versions of the compiler. When an incompatible intermediate file is detected, the compiler issues a message and terminates.

Because we do not guarantee the iELF compatibility across major releases, we strongly recommend that customers who mix Fortran with C/C++ use the same version of the compiler when they use `-ipo`.

Refer the discussion of tunables in the Installation section for additional information.

Incompatibilities between the standard C++ library ver. 1.2.1 and the draft standard

As the ANSI C++ 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 ISO/ANSI C++ 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
```

The following flags are now automatically set with A.05.* and A.06.* compilers:

- `-D__HPACC_USING_MULTIPLIES_IN_FUNCTIONAL`
- `-D__HPACC_THREAD_SAFE_RB_TREE`
- `-D__HPACC_USING_MULTIPLIES_IN_FUNCTIONAL`
- `-D__HPACC-FIX_FUNC_ADAPTER_OPERATOR`
- `-D__HPACC_FULL_ITERATOR_REL_OPS`
- `-D__HPACC_TEMPLATE_PAIR_CTOR`
- `-D__HPACC_MEM_FUN_ADAPTOR`

Conflict between macros.h and numeric_limits class (min and max)

If your code includes `/usr/include/macros.h`, note that the `min` and `max` macros defined in `macros.h` conflict with the `min` and `max` functions defined in the `numeric_limits` class of the Standard C++ Library. The following code, for example, would generate a compiler Error 134:

```
numeric_limits<unsigned int>::max();
```

If you must use the `macros.h` header, try undefining the macros that conflict:

```
...
#include <macros.h>
#undef max
#undef min
...
```

Known limitations

The following is a list of known limitations for this release. Some of these limitations will be removed in future releases of HP aC++. Please be aware that some of these limitations are platform-specific.

- HP aC++ does not support large files (files greater than 2 GB) with `<iostream.h>` or `<iostream>`
- Symbolic debugging information is not always emitted for objects which are not directly referenced. For instance, if a pointer to an object is used but no fields are ever referenced, then HP aC++ only emits symbolic debug information for the pointer type and not for the type of object that the pointer points to. For instance, use of `Widget *` only emits debug information for the pointer type `Widget *` and not for `Widget`. If you wish such information, you can create an extra source file which defines a dummy function that has a parameter of that type (`Widget`) and link it into the executable program.
- Known limitations of exception handling features:
  - Interoperability with `setjmp/longjmp` (undefined by the ISO/ANSI C++ international standard) is unimplemented. Executing `longjmp` does not cause any destructors to be run.
  - If an unhandled exception is thrown during program initialization phase (that is, before the main program begins execution) destructors for some constructed objects may not run.
  - HP aC++ does not support the linker option `-Bsymbolic`. Use the compile time option `-Bprotected_def` if you want to throw types out of a shared library. This limitation also occurs for `dynamic_cast` and `RTTI`.
- Known limitations of signal handling features:
  - Throwing an exception from a signal handler is not supported, since a signal can occur anyplace, including optimized regions of code in which the values of destructible objects are temporarily held in registers. Exception handling depends on destructible object being up-to-date in memory, but this condition is only guaranteed at call sites.
  - Issuing a `longjmp` in a signal handler is not recommended for the same reason that throwing an exception is not supported. The signal handler interrupts processing of the code resulting in undefined data structures with unpredictable results.
- Source-level debugging of C++ shared libraries is supported. However, there are limitations related to debugging C++ shared libraries, generally associated with classes whose member functions are declared in a shared library, and that have objects declared outside the shared library where the class is defined. Refer the appropriate release notes and manuals for the operating system and debugger you are using. Refer also to the Software Status Bulletin for additional details.
- Instantiation of shared objects with virtual functions in shared memory is not supported.
- Using `shl_load(3X)` or `dlopen(3C)` with Library-Level Versioning:

  Once library-level versioning is used, calls to `shl_load()` or `dlopen()` (see `shl_load(3X)`) should specify the actual version of the library that is to be loaded.

For example, if `libA.so` is now a symbolic link to `libA.so.1`, then calls to dynamically load this library should specify the latest version available when the application is compiled, such as:

```
shl_load("libA.so.1", BIND_DEFERRED, 0);
```

This will insure that, when the application is migrated to a system that has a later version of `libA` available, the actual version desired is the one that is dynamically loaded.

- Memory Allocation Routine `alloca()`

  The compiler supports the built in function, `alloca`, defined in the /usr/include/alloca.h header file. The implementation of the `alloca`() routine is system dependent, and its use is not encouraged.

  `alloca()` is a memory allocation routine similar to `malloc()` (see `malloc(3C)`). The syntax is:

  ```
  void *alloca(size_t <size>);
  ```

  `alloca()` allocates space from the stack of the caller for a block of at least `<size>` bytes, but does not initialize the space. The space is automatically freed when the calling routine exits.

  NOTE: Memory returned by `alloca()` is not related to memory allocated by other memory allocation functions. Behavior of addresses returned by `alloca()` as parameters to other memory functions is undefined.

  To use this function, you must use the `<alloca.h>` header file.

7 Related documentation

Documentation for HP aC++ / HP C is described in the following sections.

Online documentation

The following online documentation is included with the HP aC++/HP C products:

- *HP aC++ Programmer's Guide*

  Access this guide in any of the following ways:

  - Use the `+help` command-line option: `/opt/aCC/bin/aCC +help`
  - From your web browser, enter the appropriate URL:

    `file:/opt/aCC/html/C/guide/index.htm`

    NOTE: All of the files composing the guide are installed in the `/opt/aCC/html/C/guide/` directory. If you choose to move the entire English guide to a different location without having to edit any links, you need to move all of the subdirectories in `/opt/aCC/html/C/guide/`.

  - The guide (excluding Rogue Wave documentation) is also available on the World Wide Web on the Business Support Center website at http://www.hp.com/go/hpux-C-Integrity-docs.

- *HP-UX 64-bit Porting and Transition Guide*

  This guide helps developers transition applications from an HP-UX 32-bit platform to the HP-UX 64-bit platform. It is available on the HP-UX 11.x CD-ROM and on the World Wide Web at http://www.hp.com/go/aCC. In this page, click **Documentation**, and in the HP aC++ documentation page, look for *HP-UX 64-bit porting and transition guide*.

- *HP Linker and Libraries Online User Guide*

  To access, use the command: `/usr/ccs/bin/ld +help`

- *HP Wildebeest Debugger (HP WDB)*

  HP WDB documentation is available in `/opt/langtools/wdb/doc`

  HP WDB and its related documentation are available online at http://www.hp.com/go/wdb

- *Rogue Wave Software Standard C++ Library 2.2.1 Class Reference*

  This reference contains an alphabetical listing of all of the classes, algorithms, and function objects in the updated Rogue Wave Standard C++ Library. The library includes the standard iostream library and has `namespace std` enabled.

  The reference is provided as HTML formatted files. You can view these files with an HTML browser by opening the file `/opt/aCC/html/libstd_v2/stdref/index.htm` or select the hyperlink from *HP aC++ Programmer's Guide*.

- *Rogue Wave Software Standard C++ Library 2.2.1 User's Guide*

  This guide gives information about library usage and includes an extensive discussion of locales and iostreams.

  The guide is provided as HTML formatted files. You can view these files with an HTML browser by opening the file `/opt/aCC/html/libstd_v2/stdug/index.htm` or select the hyperlink in the main online help topic for *HP aC++*.

- *Rogue Wave Software Standard C++ Library 1.2.1 Class Reference*

  This reference provides an alphabetical listing of all of the classes, algorithms, and function objects in the prior Rogue Wave implementation of the Standard C++ Library. It is provided as HTML formatted files. You can view these files with an HTML browser by opening the file `/opt/aCC/html/libstd/ref.htm`.

- *Rogue Wave Software Tools.h++ 7.0.6 Class Reference*

  This reference describes all of the classes and functions in the Tools.h++ Library. It is intended for use with Rogue Wave Standard C++ Library 1.2.1.

  The reference is provided as HTML formatted files. You can view these files with an HTML browser by opening the file `/opt/aCC/html/librwtool/ref.htm`.

  There are 8 templates documented in the main part of the manual as still supported. This is incorrect. The interfaces for the following 8 templates must be translated to the new interface with two extra template arguments:

```
RWTPtrHashDictionary          ==> RWTPtrHashMap
RWTPtrHashDictionaryIterator ==> RWTPtrHashMapIterator
RWTPtrHashTable               ==> RWTPtrHashMultiSet
RWTPtrHashTableIterator       ==> RWTPtrHashMultiSetIterator
RWTValHashDictionary          ==> RWTValHashMap
RWTValHashDictionaryIterator ==> RWTValHashMapIterator
RWTValHashTable               ==> RWTValHashMultiSet
RWTValHashTableIterator       ==> RWTValHashMultiSetIterator
```

  Refer defect CR JAGaa90638.

  NOTE: Refer the *HP aC++ Online Programmer's Guide* Information Map for how to obtain additional Rogue Wave documentation and information.

- *HP aC++ Release Notes* is this document. The online ASCII file can be found at `/opt/aCC/newconfig/RelNotes/ACXX.release.notes`.
- Online man pages for aCC and `c++filt` are at `/opt/aCC/share/man/man1.Z`.

  Man pages for the Standard C++ Library and the `cfront` compatibility libraries (IOStream and Standard Components) are provided under `/opt/aCC/share/man/man3.Z`.

Online C++ example source files

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

Other documentation

Refer the *HP aC++ Online Programmer's Guide* Information Map for documentation listings, URLs, and course information related to the C++ language.

The following documentation is available for use with HP aC++.

- *Parallel Programming Guide for HP-UX Systems* describes efficient parallel programming techniques available for the HP Fortran 90, HP C, and HP aC++ compilers on HP-UX.

  This document is available on the HP-UX 11.x CD-ROM and on the World Wide Web at the following URL:

  http://www.hp.com/go/hpux-perftools-docs.

HP aC++ world wide web homepage

Access the HP aC++ World Wide Web Homepage at the following URL:

http://www.hp.com/go/aCC.

Refer the home page for the latest information regarding:

- Frequently Asked Questions
- Release Version and Patch Table
- Purchase and Support Information
- Documentation Links
- Compatibility between Releases.

HP C world wide web homepage

Access the HP C World Wide Web Homepage at the following URLs:
http://www.hp.com/go/aCC

8 Documentation feedback

HP is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (docsfeedback@hp.com). Include the document title and part number, version number, or the URL when submitting your feedback.