HP C++ Version 7.1

Release Notes for HP Tru64 UNIX

September 8, 2005

This document contains information about new and changed features in this version of HP C++ for Tru64 UNIX.

HP Computer Corporation
Houston, Texas

This document was prepared using DECdocument, Version 3.3-1n.

Contents

1 Introduction

This document contains the release notes for HP C++ Version 7.1 for HP Tru64 UNIX.

This kit installs two compilers:

- The `cxx` command invokes the Version 7.1 compiler.
- The `cxx -oldcxx` command invokes the 5.7 compiler. The version is V5.7-002. This compiler will not be included in future kits.

Note

The Version 5.7 compiler's lack of support for 128-bit long doubles can cause crashes on HP Tru64 UNIX Version 5.0 or later.

HTML files are provided for the release notes and some of the product manuals for use with a web browser. You can install these files by selecting the subset `HP C++ HTML documentation`.

To view this documentation, point your browser to

```
/usr/share/doclib/cplusplus/index.htm.
```

2 Important Compatibility Information

HP strives to maintain a high degree of compatibility between successive versions of the compiler and its run-time environment. Because, however, each new version includes enhancements and changes, you should be aware of the following whenever you upgrade:

- Differences between Run-Time and Standard Library versions
- Differences between compiler versions
- Difference between HP C++ and the C++ International Standard
- Retirement of the `cfront` language dialect

The next sections discuss these differences.

2.1 Run-Time and Standard Library Differences

Applications must use a version of the C++ Run-Time library (`libcxx`) that provides all the functions they require. If an application is linked shared, and the correct library version is not installed, "undefined symbol" error messages appear at run time. Changes in the Run-Time Library occurred in Versions 6.0, 6.2, and 6.3.

For information about redistributing the C++ Run-Time Library, see *Deploying Your Application* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

Starting with Version 6.3 of the C++ compiler, the C++ Standard Library is guaranteed to be link-compatible in all subsequent releases of the compiler. However, there is no guarantee, that the C++ Standard Library shipped with Version 6.2 or earlier releases of the compiler is compatible with the C++ Standard Library shipped with Version 6.3 of the compiler. Due to this incompatibility issue, code that references the C++ Standard Library (any of the STL containers or algorithms, standard iostreams or locales) that was compiled with a pre-V6.3 version of the compiler may need to be recompiled and relinked in order to be used with the code compiled with HP C++ V6.3 or later.

This restriction does not apply to code that references the pre-standard class library, because the stability of that library's interface guarantees link compatibility in future releases.

2.2 Compiler Differences

Starting with Version 6.0, the HP C++ compiler differs significantly from previous versions. There are several major differences that you should be aware of before using a Version 6.*n* or higher compiler for the first time. These differences are summarized here. For more detailed information, see *Porting to HP C++* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- Language differences

  The compiler implements most of the C++ International Standard, which differs significantly from the language specified in the ARM (*The Annotated C++ Reference Manual*, 1991, by Ellis and Stroustrup) with some ANSI C++ extensions. When switching from a Version 5.*n* compiler, you might need to modify your source files, especially if you use the default language mode. In addition, language changes can affect the run-time behavior of your programs. If you want to compile Version 5.*n* source code with minimal source changes, specify the `-std arm` option. See *Porting to HP C++* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

If Version 6.*n* or higher requires excessive changes to your applications even when you use the `-std arm` option, or if you encounter problems using the Version 6.*n* or higher compiler, you can compile using the `cxx -oldcxx` command. If you discover a compatiblity problem that is not documented, please file a PTR or SPR.

- Diagnostic differences

  The Version 6.*n* or higher compiler does more error checking than Version 5.7 and generates more diagnostics. If you want the number of diagnostics issued by the Version 6.*n* or higher compiler to be similar to Version 5.7, compile with the `-msg_quiet` option. For details, see *Message Control Options* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- Implementation differences

  The automatic template instantiation model has been redesigned for the current version. Although code compiled with a Version 5.*n* compiler and a Version 6.*n* or higher compiler can be combined, you must complete the Version 5.*n* instantiation process with a Version 5.*n* (or specify the `-oldcxx` option) before linking with code compiled with Version 6.*n* or higher. See *Using Templates* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

2.3 Difference between HP C++ and the C++ International Standard

The `export` keyword for templates (*Standard* §14, paragraph 6; Stroustrup §9.2.3) is not supported.

2.4 Retirement of CFRONT language dialect

The CFRONT dialect was provided for migrating code from the CFRONT compilers to the HP C++ compiler. Because it has been over five years since the last CFRONT compiler was released, we are retiring this dialect. It has been removed from Version 7.1 of the compiler.

3 C++ Standard Library

This Standard Library string class, known as the **String Library**, is not the same as the *String Package*, which is part of the Class Library implemented in earlier versions of HP C++.

For information about the HP C++ Class Library, see *Appendix A* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

4 Release Notes for the X7.1 C++ Compiler

The following sections describe enhancements, changes, problems corrected, and restrictions for the C++ compiler.

4.1 Enhancements, Changes, and Problems Corrected in Version 7.1

- To facilitate linking against shared C++ Standard Library, a new command option `-use_shared_libcxxstd` has been added.

  This option instructs the driver to link with the shared C++ Standard Library `libcxxstd.so` instead of its archived counterpart `libcxxstd.a`. Linking with `libcxxstd.a` is the default. The `-use_shared_libcxxstd` option is ignored if any of the following options is specified: `-nolibcxx`, `-use_system_libcxx`, `-non_shared` or `-use_non_shared_libcxx`.

  As is the case for the archived C++ Standard library, the C++ Standard library specified by the C++ driver on the link command depends on the `-model` and `-nopreinst` options and can be any of the following: `libcxxstd.so`, `libcxxstd_noinst.so`, `libcxxstd_noinstma.so` or `libcxxstdma.so`.

- The compiler was incorrectly transforming loops at `-O4` when an unsigned type is used as the loop iterator and is decremented across each iteration of the loop. (11120)

- The compiler was not correctly handling certain unusual loops within a switch statement. (11108)

- The compiler was not correctly handling break statements out of loops which follow an identifier label within switch statements. This is now fixed. (11183)

- Previously, the compiler was incorrectly deducing the template argument type as a const-qualifed (or volatile-qualified) type, instead of as an unqualified type, when deducing from a const- (or volatile-)qualified array type.

  ```
  template <class T>
  void foo(const T &value) { }
  void f(void) {
    const int i[3] = { 1, 2, 3 };
    foo(i);
  }
  ```

  The compiler previously deduced T to be of type "const int[3]" while it now deduces it to be of type "int [3]". This affects `-model ansi` compilations only, since `-model arm` compilations do not mangle in the template argument type.

- For a template specialization, the compiler mangles names differently depending on whether specialization is used. For example, if a function having parameter of vector<bool> type is the only entity in a compilation unit referencing vector<bool>, the function name will be mangled differently depending on whether the function actually uses its vector<bool> argument in the function body.

  In order to eliminate this dependency, a dummy function was added to the C++ standard library header <vector>, as follows:

```
#ifndef _RWSTD_NO_CLASS_PARTIAL_SPEC
#if defined(__DECCXX) && !defined(__DECFIXCXXL1760)
inline void __function_to_use_vector_bool()
{
    vector<bool> x;
}
#endif
#endif
```

  This function ensures, that vector<bool> specialization of vector<> is always treated as "used" in any program that #includes the <vector> header. (L1760)

- When compiled with full IEEE support (-ieee option on Tru64 UNIX and /FLOAT=IEEE/IEEE=DENORM qualifiers on OpenVMS Alpha), the denorm_min( ) function of the specialization of class template std::numeric_limits for long double would return garbage. This has been corrected. For example, the program below now returns the correct value: 6.47518e-4966. (L1880)

- The compiler was incorrectly computing the element size as zero when an array was a typedef'd type. This caused problems when, for example, whole array destruction was involved. (10865)

- The cxx driver now passes -preempt_symbol to the compiler when -tweak is specified on the command-line, so multiple copies of static data members don't have the problem of not sharing the same state. This behavior may be overridden by specifying -preempt_module on the command-line to get the old behavior. (11139)

```
// begin main.cxx
extern "C" int printf(const char *, ...);

template <class T>
struct foo {
  static int bar;
```

```
  void func() {
    printf("%d\n", ++bar);
  }
};

template <class T> int foo<T>::bar = 3;
void extfunc();

int main() {
  foo<int>().func();
  extfunc();
  return 0;
}
// end main.cxx

// begin rout.cxx
extern "C" int printf(const char *, ...);

template <class T>
struct foo {
  static int bar;

  void func() {
    printf("%d\n", ++bar);
  }
};

template <class T> int foo<T>::bar = 3;

void extfunc() {
  foo<int>().func();
}
// end rout.cxx

 cxx main.cxx rout.cxx -tweak -Wl,-S; ./a.out
 main.cxx:
 rout.cxx:
 4
 5
```

- A compiler assertion when initializing a function scope static variable using a temporary, also of static duration, has been fixed. (11188)
- The text of several variants of unreachable diagnostics has been modified to indicate that the compiler is not always correct when these diagnostics are issued. The severity of these diagnostics have been lowered to that of an informational. (11189)
- A compiler crash, when processing a list of overloaded functions of a class template, when the list also contained a non-real base class member, has been fixed. (11203)

5 Release Notes for the X7.1 C++ Standard Library

The following sections describe enhancements, changes, problems corrected, and restrictions for the C++ Standard Library.

5.1 Enhancements, Changes, and Problems Corrected in Version 7.1

- Shared C++ Standard Library

  Starting with Version 7.1 of the compiler, the C++ Standard Library is delivered as both an archived and shared library. The compiler installation procedure creates the following shared libraries in the compiler version specific directory /usr/lib/cmplrs/cxx/Vn.m-xxx/:

```
. libcxxstd.so            model arm preinstantiation library
. libcxxstd_noinst.so     model arm noinstantiation library
. libcxxstdma.so          model ansi preinstantiation library
. libcxxstd_noinstma.so   model ansi noinstantiation library
```

  As is the case for the shared class library libcxx[ma].so, the compiler installation procedure creates symbolic links to the C++ Standard shared libraries in the /usr/lib/cmplrs/cxx/ directory.

  Shared C++ Standard Libraries are neither on the compiler kit nor on the C++ Library Redistribution kit. Vendors must be aware of the fact that applications linked against the shared C++ Standard Library can be executed only on a system that has this library.

  The compiler installation procedure generates shared C++ Standard Libraries from their respective archived counterpart using a link command similar to the following:

```
ld -shared -o $COMPVER/libcxxstd.so -rpath $COMPVER -L$COMPVER \
       -no_so -all -lcxxstd -none -no_archive -lcxx -lc
```

  where the COMPVER environment variable points to the compiler-version-specific location of the C++ libraries on the system.

  Linking against archived C++ Standard library remains the default. In order to link against shared C++ Standard Library, specify the -use_shared_libcxxstd option on the cxx command. See Section 4.1 for a description of this command-line option.

  Linking against the shared C++ Standard Library is especially recommended when an application dynamically loads libraries written in C++ using the C++ Standard Library. In this case, the best way to ensure that the C++ Standard library is initialized only once during the application run is to link both the main executable and dynamically loaded libraries against the shared C++ Standard Library.

Mixing dynamically loaded libraries linked against the archived and shared C++ Standard Library in the same process is not supported. Also, a main executable must be linked against the same flavor - either .a or .so - of the C++ Standard Library as the libraries it dynamically loads are linked against. Violation of this restriction can result in unpredictable behavior.

- A problem with vector container created by a constructor accepting two input iterators has been corrected. After the fix, the constructor populates the container with all the contents of the stream associated with the iterator, as it should. Before the fix, the constructor would put only the first stream record into the container. A program like the program example in Section 3.8.3 "Iterators and I/O" in Stroustrup's *C++ Programming Language*, 3rd edition, now generates the correct result.
- A bug in codecvt<wchar_t,char,mbstate_t>::encoding( ) specialization of the `codecvt::encoding()` member function has been fixed. The function used to return 0 regardless of the encoding established by the facet while, according to section 22.2.1.5.2, p7 of the C++ standard, it should return -1 if the encoding is state-dependent, a constant number of characters needed to produce a wide character (as for a single-byte character set), and 0 otherwise (as for a multibyte character set).

  For example, after the fix, the program below generates the following output:

```
1
1
0
```

  Before the fix, it would generate the following output:

```
x.cxx
-----
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <locale>
#include <iostream>

using namespace std;

int main()
{
  codecvt_byname<wchar_t,char,mbstate_t>* p;

  // C locale
  p = new codecvt_byname<wchar_t,char,mbstate_t>("");
  cout << p->encoding() << endl;
```

```
  // single-byte locale
  p = new codecvt_byname<wchar_t,char,mbstate_t>("en_US.ISO8859-1");
  cout << p->encoding() << endl;

  // multibyte locale
  p = new codecvt_byname<wchar_t,char,mbstate_t>("ja_JP.SJIS");
  cout << p->encoding() << endl;
}
```

- codecvt<wchar_t,char,mbstate_t>::max_length() specialization of the `codecvt::max_length()` member function has been modified to return MB_CUR_MAX for the encoding established by the facet instead of MB_LEN_MAX. While not a bug, MB_CUR_MAX is a more accurate return value for the `max_length()` function, and this is what the function returns in other implementations of the C++ Standard Library, including recent versions of Rogue Wave library.
- To comply with 21.2 - String classes [lib.string.classes] - in the C++ standard, declarations of the `std::getline()` function operating on basic_istream have been moved from <istream> to <string>. Accordingly, the definition of the `std::getline()` function operating on basic_istream and accepting the delim parameter has been moved from <istream.cc> to <string.cc>. This change is visible only when using the standard iostreams.
- The HP C++ library defines std::ostream_iterator as the following:

  ```
  template <class T, class charT = char, class traits = char_traits<charT> >
  class _RWSTDExportTemplate ostream_iterator :
    public iterator<output_iterator_tag,T,_TYPENAME traits::off_type,T*,T&>
  ```

  However, section 24.5.2 - Template class ostream_iterator [lib.ostream.iterator] of the C++ standard defines std::ostream_iterator as the following:

  ```
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator:
    public iterator<output_iterator_tag, void, void, void, void>
  ```

  HP C++ Version 7.1 introduces the macro __COMPLY_WITH_24_5_2. When compiled with this macro defined, the library provides the standard-compliant definition of std::ostream_iterator.

  Note that defining the __COMPLY_WITH_24_5_2 macro changes the types defined by std::ostream_iterator, namely: value_type, difference_type, pointer, and reference. Because of changing types, it is a good idea to make sure that if the macro is defined, it is defined consistently in your application and in the libraries the application is using.

For example, consider the following program:

```
x.cxx
-----
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iterator>
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
  cout << typeid(ostream_iterator<char>::value_type).name() << endl;
  cout << typeid(ostream_iterator<char>::difference_type).name() << endl;
  cout << typeid(ostream_iterator<char>::pointer).name() << endl;
  cout << typeid(ostream_iterator<char>::reference).name() << endl;
}
```

When compiled without the __COMPLY_WITH_24_5_2 macro defined, x.cxx gives:

```
char
long
char *
char
```

When compiled with the __COMPLY_WITH_24_5_2 macro defined, it gives:

```
void
void
void
void
```

- To synchronize access to the reference count in the reference counting implementation of the std::string class, the C++ Standard Library uses atomic instructions. Version 7.1 of the compiler introduces an alternate synchronization mechanism based on the pthread mutex embedded in _RWstring_ref_rep class and using the TIS interface.

  A mutex-based synchronization can provide better performance in configurations with slow memory access, especially, when an application is not threaded and TIS mutex blocking becomes a no-op. Also, a mutex-based synchronization is more robust in some situations. For example, when an application fails to provide proper high-level synchronization when operating on objects of std::string class in different threads, a mutex-based synchronization might allow the application to "survive". However, HP does not recommend relying on this feature.

To enable mutex-based synchronization in std::string class, a program should be compiled with the __USE_EMBEDDED_PTHREAD_MUTEX macro defined. Additionally, a program should be using a nopreinstantiation version of the C++ Standard Library, which assumes compiling with the __FORCE_INSTANTIATIONS macro defined and linking `-nopreinst`.

Objects generated by compiling with the __USE_EMBEDDED_PTHREAD_MUTEX macro defined are not binary-compatible with objects generated by compiling without the macro defined, and should not be linked in the same application. This is also true for dynamically loaded C++ libraries. That is, with respect to the __USE_EMBEDDED_PTHREAD_MUTEX macro, all dynamically loaded libraries and the main executable should be compiled the same way. A vendor whose library is built with the macro defined should notify their users to also compile with the macro defined (and use nopreinstantiation C++ Standard Library).

- A bug in codecvt<char,char,mbstate_t>::encoding( ) specialization of the codecvt::encoding( ) member function has been fixed. The function used to return -1 while, according to section 22.2.1.5.2, p7 of the C++ standard, it should return 1. The fix makes sure that the specialized function returns 1.
- A bug in codecvt<char,char,mbstate_t>::in( ), out( ), and unshift( ) specialization of the codecvt member functions has been fixed. These functions used to return codecvt_base::error while, according to section 22.2.1.5.2 of the C++ standard, they should return codecvt_base::noconv. The fix makes sure that the specialized functions return codecvt_base::noconv.
- The codecvt<char,char,mbstate_t>::always_noconv( ) function has been modified to return `true` to comply with section 22.2.1.5.2, p8 of the C++ standard. In previous complier releases, this function used to return `false`.
- To be consistent with Library Issue 103, the reverse_iterator typedef in set and multiset has been changed to _RWrep_type::const_reverse_iterator.
- Because of a bug in the C++ standard library, it was impossible to define and use a user-defined facet. For example, the following program would not compile. This has been fixed.

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <locale>
```

```
struct foo : std::locale::facet
{
  static std::locale::id id;
};

std::locale::id foo::id;

int main()
{
  std::use_facet<foo>(std::locale());
  return 0;
}
```

- A problem with formatting a hexadecimal number using the ios_base::internal adjustfield manipulator has been corrected. For example, after the fix, the program below outputs the correct string: "0x0021". Before the fix, it would output: "000x21".

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <iostream>
#include <iomanip>

int main()
{
std::cout << std::hex << std::showbase << std::setfill('0')
          << std::setw(6) << std::internal << 33 << '\n';
}
```

- The library previously used the same storage for `iarray` (an array of long integers) and `parray` (an array of pointers to void), manipulated by the `iword()` and `pword()` member functions, respectively, of class std::ios_base. This has been corrected so that `iarray` and `parray` now use separate storage. For example, after the fix, the following program outputs the correct result:

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <iostream>

int main()
{
  int const index = std::ios_base::xalloc();
  std::cout.iword(index) = 42L;
  std::cout << "iword=" << std::cout.iword(index) << std::endl;
  std::cout.pword(index) = 0;
  std::cout << "iword=" << std::cout.iword(index) << std::endl;
}
```

Correct result:

```
iword=42
iword=42
```

Before the fix, it would output:

```
iword=42
iword=0
```

- The C++ headers <cassert> and <assert.h> in /usr/include/cxx_cname/ have been modified to work around the bug in Tru64 system header /usr/include/assert.h which casts the argument to macro assert() to an int. The headers in /usr/include/cxx_cname/ redefine macro assert() without the "int" cast.

  For example, after the fix, the following program compiles cleanly.

```
#include <cassert>

class foo
{
  int * ptr;
public:
  explicit foo(int * p = 0): ptr(p) {}
  typedef int * foo::*unspecified_bool_type;
  operator unspecified_bool_type() const
{
    return ptr == 0? 0: &foo::ptr;
  }
};

int main()
{
  void *p = 0;
  assert(p); // used to give warning
  foo u;
  assert(u); // used to give error
  return 0;
}
```

- A problem has been corrected with the assignment operator of the tree container not storing the comparison object of the container being copied into the target container.

  The tree container is the underlying container for the map and set STL containers. Because of this problem, after assigning one STL container object to another, the target container would continue to use the comparison object it was using before the assignment. It violates section 23.1.2 - Associative containers [lib.associative.reqmts] of the C++ standard which states:

When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, either through a copy constructor or an assignment operator, the target container shall then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

6 Release Notes for the V6.5 C++ Compiler

The following sections describe enhancements, changes, problems corrected, and restrictions for the C++ compiler.

6.1 Enhancements, Changes, and Problems Corrected in 6.5-042

- Compiler assertion compiling very large programs with debug has been eliminated. (10828)
- Compiler crash in pipeline optimization has been eliminated. (10216)

6.2 Enhancements, Changes, and Problems Corrected in 6.5-041

- A program using standard iostreams for non-standard __int64 and long long datatypes could not be compiled in strict ansi mode. This restriction has been eliminated. (L1823)
- Eliminate buffer overrun in CXXLINK. (10633)
- Eliminate incorrect diagnostic on using declaration of template base class. (10513)
- Eliminate compiler assertion when using a complex expression in the condition of a for loop. (10678)

6.3 Enhancements, Changes, and Problems Corrected in 6.5-040

- Add srand to namespace STD. (9551)
- Problem looking up non-dependent names that are static has been corrected. (10458)
- Compiler crash when compiling in strict ANSI mode has been eliminate. (10460)

6.4 Enhancements, Changes, and Problems Corrected in 6.5-039

- Eliminated incorrect unreachable diagnostic introduced in the V6.5-034 compiler. (10307)
- Corrected template argument deduction bug. (10352)

6.5 Enhancements, Changes, and Problems Corrected in 6.5-038

- Improve compilation speed by eliminating the opening and closing of header files that have include once preprocessor guards. (8786)

6.6 Enhancements, Changes, and Problems Corrected in 6.5-037

- A problem with the class library fstream::tellp( ) function returning incorrect result when called after the following sequence of the calls — reading partial file data followed by positioning put pointer to zero offset followed by writing to the file—has been fixed. [1742L]

6.7 Enhancements, Changes, and Problems Corrected in V6.5-036

- Inappropriate inaccessible member diagnostic has been eliminated. (10215)
- Inappropriate diagnostic when initializing data members of a class template has been eliminated. (10238)
- Compiler assertion when processing memcpy has been eliminated. (10137)
- Bad runtime typeinfo (RTTI) has been corrected. (10156)

6.8 Enhancements, Changes, and Problems Corrected in V6.5-035

- Implemented "-FI" which includes the specified file before processing sources. (5434,10064)
- Eliminate compiler assertion for catch blocks with control flow that will not reach the end of the block. (9946,10046)
- Do not issue a warning if a template parameter is not used in signature of template if it is possible to use the template. (9909)

6.9 Enhancements, Changes, and Problems Corrected in V6.5-034

- Eliminate compiler assertion for catch blocks with control flow that will not reach the end of the block. (9946)

6.10 Enhancements, Changes, and Problems Corrected in V6.5-033

- Compiler crash when compiling member function "-g" has been eliminated. (9769)
- Incorrect handling of memmove of overlapping strings with "-arch ev56" or later has been corrected. (9791)

6.11 Enhancements, Changes, and Problems Corrected in V6.5-032

- Compiler crash in exception support has been eliminated. (9743)
- Incorrect this pointer when calling a virtual function from a constructor has been corrected. (9751)
- Incorrect type definition, resulting in incompatible intrinsic definitions and functions not being made instrinsic in Microsoft mode, has been corrected. (9755)
- Compiler crash in exception support has been eliminated. (9756)

6.12 Enhancements, Changes, and Problems Corrected in V6.5-031

- Compiler crash when using `goto` statements has been eliminated. (9666)
- Compiler crash when using multi-dimensional arrays has been eliminated. (9675)
- Undefined symbols when using RTTI information for long long types in model ANSI has been corrected. (9687)
- Incorrect error about using declaration has been eliminated. (9693)
- Runtime crash when using exceptions and optimization has been eliminated. (9697)
- Specifying the **-alternative_tokens** option now defines the macro __ALTERNATIVE_TOKENS.

6.13 Enhancements, Changes, and Problems Corrected in V6.5-030

- Incorrect informational about partially overridden virtual function has been eliminated. (9343)
- Compiler hang when generating `XREF` cross reference information has been eliminated. (9561)

6.14 Enhancements, Changes, and Problems Corrected in V6.5-029

- std::deque<>::erase(iterator position) function has been corrected to return end( ) if after erasing the element the container is empty. (L1686)
- The binaries shipped in the compiler kit were not stripped. This resulted in the kit being bigger than it needed to be. This has been corrected. (9526)
- Compiler crash when using `-omp` and inline functions has been eliminated. (9580)

6.15 Enhancements, Changes, and Problems Corrected in V6.5-028

- Code generation problem resulting in failure of virtual function override when using -nortti has been corrected. (9575)
- Fix buffer overrun in ostrstream and strstream. (L1684)
- Incorrect "extern inline function was referenced but not defined" diagnostic has been eliminated. (8862)
- Compiler crash when using `-omp` and preincrement or predecrement has been eliminated. (9550)
- Using spike with `-s` no longer results in an error finding strip. (9486)

6.16 Enhancements, Changes, and Problems Corrected in V6.5-026

- Code generation problem resulting in virtual function call to member function from class of same name, but in a different namespace, has been corrected. (9566)

6.17 Enhancements, Changes, and Problems Corrected in V6.5-024

- Release notes updated to clarify version compatibility.

6.18 Enhancements, Changes, and Problems Corrected in V6.5-021

- Compiler crash in optimizer has been eliminated. (9532)
- Compiler crash when using listing files with macro expansion and very long lines has been eliminated. (9518)
- Optimization problem with memcpy intrinsic has been corrected. (9512)
- Compiler assertion when using `-omp_eh_full` has been corrected. (9393)
- Compiler assertion when using OpenMP and data member in private list has been corrected. (9281)
- Exception handler now preallocates some memory so exceptions can be thrown even if no memory is available. (7881)

6.19 Enhancements, Changes, and Problems Corrected in V6.5-020

- Improved compiler optimization resulted in reduced abstraction penalty in Stepanov benchmark.
- When called with a null string, the basic_stringbuf::str(const string_type& str) method was not setting the underlying character buffer to zero length. It has been fixed. [10.1674]
- `<new>` and `<new.hxx>` library headers have been modified to suppress "support for placement delete is disabled" informational message at the point of declaration of nothrow version of operator delete and nothrow version of array delete. [10.1675]

6.20 Enhancements, Changes, and Problems Corrected in Version 6.5

- Improved conformance to the C++ International Standard
- Improved code optimization
- Retirement of CFRONT dialect announced
- Retirement of `-oldcxx` compiler announced

- The compiler supports C/C++ OpenMP Version 1.0. By default, the compiler ignores all C++ OpenMP directives unless you specify the `-omp` option. For example:

  ```
  cxx -omp omp_program.cxx
  ```

- The library header and template definition files have been modified to compile in STRICT_ANSI mode with the Version 6.5 compiler. Modification was necessary because Version 6.5 enforces more stringent language rules in some cases than previous versions. [10.1649]

- The `deallocate()` member function of the allocator class in the `<memory>` header has been modified so that `operator delete` is not called with the null pointer. Although calling `delete` with the null pointer is legal in HP C++, the Third Degree tool on Tru64 UNIX issues a warning when such a call is made. The modification was made to avoid the warning. [10.1657]

6.21 Restrictions in Version 6.5

The following restriction applies for the Version 6.5 release:

- When the `mbrtowc()` function on RedHat V7.0 is passed a null character, it fails to write zero (L'0') into the output wide character. This behavior can cause the `codecvt_byname<wchar_t,char,mbstate_t>::in()` function to generate an incorrect sequence of wide characters if an input sequence of (multibyte) characters contains a null character.

- Comparing an object of the Standard Library fstream class or an object of the Class Library fstream, ifstream or ofstream class with the null pointer causes an ambiguity compilation error.

  For example, the following program produces the following compilation errors:

  ```
  #include <fstream.h>

  int foo()
  {
    ifstream ifs;
    ofstream ofs;
    fstream  fs;

    if (ifs == NULL ) return 0;
    if (ofs == NULL ) return 0;
    if (fs == NULL ) return 0;

    return 0;
  }
  ```

```
$ cxx -c x.cxx
cxx: Error: x.cxx, line 9: more than one operator "==" matches these operands:
            built-in operator "pointer == pointer"
            built-in operator "pointer == pointer"
            operand types are: ifstream == long
  if (ifs == NULL ) return 0;
----------^
cxx: Error: x.cxx, line 10: more than one operator "==" matches these
          operands:
            built-in operator "pointer == pointer"
            built-in operator "pointer == pointer"
            operand types are: ofstream == long
  if (ofs == NULL ) return 0;
----------^
cxx: Error: x.cxx, line 11: more than one operator "==" matches these
          operands:
            built-in operator "pointer == pointer"
            built-in operator "pointer == pointer"
            built-in operator "pointer == pointer"
            operand types are: fstream == long
  if (fs == NULL ) return 0;
---------^
cxx: Info: 3 errors detected in the compilation of "x.cxx".
$
```

The workaround is instead of comparison with the null pointer which is based on `operator void *()` use `operator!()`. For example, if the program is modified as follows:

```
  if ( !ifs ) return 0;
  if ( !ofs ) return 0;
  if ( !fs ) return 0;
```

it compiles cleanly. [L1710]

- If executable is to be linked against several shared images using the Standard Library, these shared images must be created without linking in the `libcxxstd` library and with the `expect_unresolved ld` flag. And then, when linking the executable, if it is not written in C++, the C++ Class and the Standard library must be explicitly specified. For example:

  ```
  cxx -shared -nocxxstd -expect_unresolved "*std*" -o libfoo.so foo.cxx
  cxx -shared -nocxxstd -expect_unresolved "*std*" -o libbar.so bar.cxx
  cc main.c ... -lfoo -lbar -L<location of C++ libraries> -lcxxstd -lcxx
  ```

  This is similar to the technique of eliminating linker's multiply defined symbols described in section `Restrictions in Version 6.2`. A failure to follow build procedure described here may result in application crash during initialization of one of the shared images. [L1704]

- Consider a scenario when a main executable creates a global object of a class whose constructor dynamically loads a library and whose destructor unloads it. If the dynamically loaded library happens to be written in C++ and linked using the cxx command, the program can crash on the exit from main().

  The workaround is to link dynamically loaded library with the ld command without specifying _main.o object which is automatically added by the C++ driver to the link command.

  Here is an example:

```
$ cxx main.cxx -o main
$ cxx -c -o foo.o foo.cxx
$ cxx -shared -o libfoo.so foo.o
$ main
returning from main()
Closing
Done Closing
Segmentation fault (core dumped)
$ ld -shared -o libfoo.so -L/usr/lib/cmplrs/cxx/ foo.o -lcxxstd -lcxx -lc
$ main
returning from main()
Closing
Done Closing
$
```

```
main.cxx
--------
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iostream>
#include <dlfcn.h>

using namespace std;
```

```
struct Loader {
  void* handle;
  Loader() : handle(NULL) {}
  ~Loader() {
     if (handle) {
       cout << "Closing" << endl;
       dlclose(handle);
       cout << "Done Closing" << endl;
     }
  }
  void Load(const char* file) {
    handle = dlopen (file, RTLD_LAZY);
    if (handle == NULL)
      cout << "unable to load file " << file << dlerror() << endl;
  }
};

Loader loader;

int main()
{
  loader.Load("./libfoo.so");
  cout << "returning from main()" << endl;
  return 0;
}

foo.cxx
-------
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iostream>

void foo(void) { std::cout << "Hello, world" << std::endl; }
```

- Tight coupling of any of std::cout, std::cerr and std::clog by making them use the same strstreambuf object may cause a deadlock in a multithreaded application. For example, the following program may deadlock:

```
  ofstream log("test.log");
  cout.rdbuf(log.rdbuf());
  cerr.rdbuf(log.rdbuf());
  ...
  writing to cerr from different threads
  ...
```

The workaround is to untie the streams before tight coupling them, as the following:

```
ofstream log("test.log");
cerr.tie(0);
cout.rdbuf(log.rdbuf());
cerr.rdbuf(log.rdbuf());
```

Note, that the current implementation of the C++ Standard Library ties all four standard streams (the C++ standard only requires, that cin is tied to cout). [L1707]

7 Release Notes for the V6.3 C++ Compiler

7.1 Enhancements, Changes, and Problems Corrected in V6.3-018

- Compiler and libraries are built with a new C compiler to improve performance.
- The iversion string for libcxx.so has been updated to work around a bug in the loader in Tru64 V5.1 that prevented the library from being loaded.(8638) This is the same fix made in -011, the fix was missing from -012, -013, and -014.

7.2 Enhancements, Changes, and Problems Corrected in V6.3-014

- Compiler crash processing va_list eliminated. (8803)
- Problem with dynamic cast of standard library types corrected. (8828)

7.3 Enhancements, Changes, and Problems Corrected in V6.3-013

- Runtime crash in exit when program is compiled "-g" and atexit is used has been eliminated. (8800)
- Runtime hang when two tied streams share the same streambuf object has been eliminated. (cxxl1641)
- Runtime problem with memcpy family of intrinsics has been corrected. (8824)

7.4 Enhancements, Changes, and Problems Corrected in V6.3-012

- In previous releases, the compiler treated string literals as pointers to const char (`const char *`) in `-std strict_ansi` and `-std strict_ansi_errors` modes, but as pointers to `char` (`char *`) in all other -std modes. This could lead to incompatibilities between the object generated under the former `-std` modes and the latter ones. The `-std` option no longer controls how string literals are treated. Instead, the `-model` option controls this behavior. In the arm object model (`-model arm`, the default on Tru64 Unix), string literals are treated as `char *`. In the ansi object model (`-model ansi`,

the default on Alpha Linux), string literals are treated as `const char *`. (Objects generated using the `-model arm` option are incompatible with the objects generated using the `-model ansi` option. The `-model arm` option is unavailable on Alpha Linux.) (8659)

- The compiler once again correctly recognizes intrinsics with variable number of arguments. (4850)
- Runtime crash when using iostream with long double on V5.0 Tru64 has been eliminated. (cxxl1623)
- Destructor is no long envoked for uncreated object in "?" expression. (8736)
- Destructor is no long skipped on exception for object with initializer. (8772)
- Code optimization generating incorrect code has been correct. (8762)

7.5 Enhancements, Changes, and Problems Corrected in V6.3-011

- A multithread safety problem which can lead to corruption or deadlocking when using the C++ standard library iostreams and locales has been fixed.(L1625)
- The iversion string for libcxx.so has been updated to work around a bug in the loader in Tru64 V5.1 that prevented the library from being loaded.(8638)
- The compiler no longer aborts when using pointers to members.(8632)

7.6 Enhancements, Changes, and Problems Corrected in V6.3-010

- The compiler no long aborts when processing a class that imports a set of overloaded delete operators from a base class with a using-declaration or an access declaration. (8611)

7.7 Enhancements, Changes, and Problems Corrected in V6.3-009

- The compiler no longer generates __init_sti names incorrectly which was resulting in duplicate symbols. (8568)
- The demangler no long crashes on names beginning with __INTER. (8553)
- The archive version of libdemangle has been added to the kit. (8553)
- The compiler no longer crashes when `-xref` is used and duplicate templates are encountered. (8501)

- The compiler no longer encounters a catastrophic error on a conditional operator. (8481)
- The compiler no longer encounters a catastrophic error on a virtual base class. (8477)
- A multithread safety problem which can lead to corruption or deadlocking when using the C++ standard library iostreams and locales has been fixed.(L1561)

7.8 Enhancements, Changes, and Problems Corrected in V6.3-008

- The compiler no long generates identifier undefined warnings in <math.h> when compiling -fast with Unix standard macros.

7.9 Enhancements, Changes, and Problems Corrected in V6.3-007

- The GEM backend was upgraded.
- Documentation updated to reflect 6.3A release.

7.10 Enhancements, Changes, and Problems Corrected in V6.3-006

- Components of the "-oldcxx" compiler are now in the appropriate subset.
- Fixed arch EV68 support.

7.11 Enhancements, Changes, and Problems Corrected in V6.3-005

- The compiler no longer generates an error when compiling pthreads.h with "-std strict_ansi."
- The GEM backend was upgraded.

7.12 Enhancements, Changes, and Problems Corrected in V6.3-003

- The compiler no longer generates a compiler memory access violation for a local function declaration inside a member function with the same name as a virtual and a non-virtual function in the function's base class.

- The compiler no longer generates a memory access violation when it sees a reference to a <cname> structure in a function prototype before the <cname> structure is declared.

7.13 Enhancements, Changes, and Problems Corrected in Version 6.3

Enhancements, changes, and problems corrected are as follows:

- This version of the C++ compiler implements C++ headers for C Library Facilities. The <cname> headers avoid pollution of the global namespace by defining all C names only in namespace std. (See Stroustrup, §9.2.2 and §16.1.2.)

  The <cname> headers are located in the directory /usr/include/cxx_cname.

  If you include a <cname> header and use the -pure_cname option, all C functions and types found in that header file are declared only in namespace std, as specifed by the C++ International Standard.

  Specifying the -nopure_cname option causes <cname> headers to be handled as if the corresponding <name.h> version had been included. That is, names are available both in namespace std and global scope.

  The default is -pure_cname when compiling with -std strict_ansi and -std strict_ansi_errors. The default is -nopure_cname when compiling with -std ansi, -std arm, -std gnu, -std ms, and -std cfront.

  The compiler search order for include files has been changed from

  ```
  /usr/include/cxx
  /usr/include
  ```

  to

  ```
  /usr/include/cxx
  /usr/include/cxx_cname
  /usr/include
  ```

  Including <name.h> after including the corresponding <cname> header brings all names declared in that <cname> header into global namespace with "using std::name" declarations.

  New Header Files and protect_headers_setup

  The addition of <cname> headers adds several new files and the new subdirectory /usr/include/cxx_cname. If you are using protect_headers_setup on your system, you might need to rerun it after installing or upgrading to the HP C++ Version 6.3 compiler. For more information, see the protect_headers_setup(8) reference page or Protecting System Header Files in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

Backward Compatibility

In -pure_cname mode, <cwchar> does not include <cstdio>. Therefore, printf() is not declared unless <cstdio> is included.

For example, the vfwprintf function, declared in <cwchar>, requires that <cstdarg> and <cstdio> be included:

```
int vfwprintf(FILE *, const wchar_t *, va_list);
```

New overloaded function signatures have been added to several <cname> headers (see *Standard* §21.4, §25.4, and §26.5). These overloaded signatures have been made available when including the <cname> header in -pure_cname mode. If the __CNAME_OVERLOADS macro is defined, the new signatures are available in both -pure_cname and -nopure_cname modes. On some operating system platforms, defining the __CNAME_OVERLOADS macro in -nopure_cname mode in combination with other macros and options (for example, -ms, -D_XOPEN_SOURCE) can cause compile errors.

cmath (*Standard* §26.5) now provides float and long double overloaded signatures for math functions in -pure_cname mode, or in -nopure_cname mode with the __CNAME_OVERLOADS macro defined.

These added signatures could cause type ambiguity problems or different runtime behaviour in existing code. Consider theses examples:

- sin(1) is now ambiguous because overloads are provided for float, double, and long double. A user sees the following differences, because the argument of sin(1) is assumed to be of type int:

  ```
  cout << "sin(1) = " << sin(1) << endl;  // generates an error
  ```

  ```
  cxx: Error: t.cxx, line 15: more than one instance of overloaded function
           "sin" matches the argument list:
             function "std::sin(long double)"
             function "std::sin(float)"
             function "std::sin(double) C"
             argument types are: (int)
           cout << "sin(1) = " << sin(1) << endl;
           ----------------------^
  ```

- The type of the argument to an overloaded math function determines the type of its return value and associated precision. Calls to math functions using float or long double arguments may return less precise or more precise values than previously. Compare the following:

  - Previous compiler release:

    ```
    long double  ldout = sin(1.0);
    ldout = 0.84147098480789650000
    ```

- Current compiler release:

```
long double  ldout = sin(1);    // type int argument - ambiguous

long double  ldout = sin(1.0l); // type long double argument
ldout = 0.84147098480789650000

long double  ldout = sin(1.0f); // type float argument
ldout = 0.84147095680236816000
```

Signatures have been added in cstring, cwchar, and cstdlib header files for the following functions:

cstring: (*Standard* §21.4) strchr, strpbrk, strrchr, strstr, memchr
cwchar: (*Standard* §21.4) wcschr,wcspbrk, wcsrchr,wcsstr, wmemchr
cstdlib: (*Standard* §25.4) bsearch, qsort, (§26.5) abs, div

The added signatures could cause problems in existing code. For example, because char* strchr(const char*, int) now has overloads const char* strchr(const char*, int) and char* strchr(char*, int), the following code does not work:

```
#include <cstring>
void f(char*) {;}
int main() {
   f(strchr("abc",1)); // strchr returns a const char*
   return 0;
}
```

- The -t option has two new variants, -ti (include C++ header files) and -tj (include cname header files).

  For a complete description of the -t option, see the cxx(1) reference page.

- In strict_ansi mode, the name of a class is now entered as a member of itself, as required by clause 9 (para 2) of the Standard; this behavior is implemented more or less as an implicitly declared member typedef and might cause some existing programs to fail. For example:

```
namespace std {
     class iterator {};
}
```

```
struct tree
{
    struct iterator {};
    struct nested : public std::iterator
    {
      // HP C++ 6.2 and below thinks this is tree::iterator
      // HP C++ ?? in strict_ansi mode thinks this is std::iterator
      nested(const iterator&);
    };
};
```

[6613]

- The error "incompatible parameter", issued when there is a difference in sign between pointers, has been made discretionary. As a result you can now reduce/increase the severity of this message or enable/disable it using its error tag `incompatibleprm` or its error number. The same can also be done by enclosing the offending code in #pragma. For Example, the error message for the following program can now be controlled.

```
void f(unsigned int *i) {
}
void main() {
  f((int *)0x05);
}
```

  In addition, specifying `-std gnu` reduces the message severity to warning.

- The tree data structure, which sets and maps usage, has been refined to decrease the amount of space allocated for small element size containers. [10.1475]
- To ensure thread safety, the `basic_string` reference count used to be protected by a mutex, which called thread locking and unlocking routines. Performance of this class in multithreaded applications has been improved by changing the implementation to use instead atomic builtins (see *Appendix B* in *Using HP C++ for Tru64 UNIX and Linux Alpha*). [10.1138]
- The Standard Library vector class now allocates space correctly for elements greater than 1024 bytes; runtime core dumps caused by incorrect allocation in previous versions no longer occur. [10.1459]
- The `iostreams` and `locales` are now multi-thread safe. [10.1429]
- Compiling a program in `-std strict_ansi` mode using the `basic_fstream` class no longer causes run-time seg faults or core dumps. [10.1357]
- The `basic_string::find_first_not_of(charT, size_type)` function now works correctly if the string contained embedded nulls. [10.1316]

- The Version 6.2 string extraction operator no longer removes the extra space at the end of the string, as in the following example

```
ifstream inFile("input.dat"); // input.dat contains "abc de"
inFile >> word; // read "abc"
inFile.get(ch1); // ch1 should be space, was 'd'
```

  [10.1300]

- Two of the `basic_string::compare()` member functions no longer throw an exception if the length of the second string is longer than the length of the current string. They no do so only if the position the user specifies within the second string is greater than the length of the second string. [10.1298]
- The prototype for the `set_new_handler()` is now included in the `<new>` header file when you specify `-nostdnew` on the command line.
- The `basic_string::resize()` member function now works correctly. If two strings point at the same underlying `char*` and the `resize()` function is called on one of them, you can change the underlying string for one of the strings without affecting the value of the other.
- You can now call the algorithm `stable_sort()` more than once with the same container without causing a seg fault.

7.14 Restrictions in Version 6.3

The following restrictions apply for the Version 6.3 release:

- In Version 5.*n*, if you allocate an array of objects of a class type with a constructor but without a destructor using `new[n]`, you will be unable to deallocate this object using `delete[]` in a module compiled with Version 6.*n* or higher because the hidden count was not stored during the allocation. [7232]
- Because the task library is being retired, it is no longer a shared object. You must therefore specify `-threads -lcmalib` on the cxx command line in addition to `-ltask` for the link. [7976]
- A template mangling problem can occur if you use class template partial specializations with non-type template arguments in the compiler's model arm mode.

For example, the following program produces a compilation error:

```
template <int i, int j> struct C {};
template <int i, int j, class S> struct D {};

template<int Dim1, int Dim2, int Dim3>
struct D<Dim1, Dim2, C<Dim1, Dim3> >
{
  void foo();
};

template<int Dim1, int Dim2, int Dim3>
struct D<Dim2, Dim1, C<Dim1, Dim3> >
{
  void foo();
};

int main() {
    D< 1, 2, C<2, 1> > d;
    d.foo();
    D< 2, 1, C<2, 1> > d2;
    d2.foo();
    return 0;
}
```

The problem occurs because the two instantiations of `foo()` are incorrectly mangled identically. As shown in the example, this happens when the template parameter list is instantiated with the same arguments in the same order, but the template argument list is actually different.

If the calls to `D<...>::foo()` were made from two different source files, the compiler would not give an error; instead it would incorrectly call the same instantiation for both calls.

This problem cannot be corrected without breaking link compatibility with objects produced from the previous releases. The name is mangled correctly in the compiler's model ansi mode. For more details on the model ansi compiler option, see the description of the `-model [ansi | arm]` option in the cxx.1 reference page. [8012]

- A problem in `libexc` on Tru64 UNIX before Version 4.0E generates an error message when an exception is thrown from a constructor that will be called by a static initialization, as in the following example. The error message appears after the example. There is no workaround for the problem.

```
#include <stdio.h>
#include <exception>
#include <stdlib.h>
```

```
void term_handler()
{
     printf("in terminate handler\n");
     abort();
}

class C {
  public:
     C() {
           std::set_terminate(&term_handler);
           throw 5;
     }
};

C c;

int main()  {
    return 0;
}
```

The code generates the following error message:

```
exception system: exiting dues to multiple internal errors:
        exception dispatch or unwind stuck in infinite loop
        exception dispatch or unwind stuck in infinite loop
```

[CPP 5021]

- When using the Standard Library iostreams for interactive input to `cin` from a terminal, a user may have to type more than one Ctrl D to indicate end-of-file. [10.1413]
- Currently you might encounter compilation errors if you try to use a user-defined allocator and pointer class with the STL containers. This problem will be fixed in a future release. [10.1430]

8 Release Notes for the V6.2 C++ Compiler

8.1 Enhancements, Changes, and Problems Corrected in Version 6.2-040

Enhancements, changes, and problems corrected are as follows:

- In certain cases, dynamic initialization of a static variable of aggregate type with an initializer list containing both constant and non-constant values did not occur correctly. Aggregate members initialized with constant values were initialized with 0, not with the constant values. Consider this example:

```
const float x = 3;

struct S {
    static long *_GetEntries() {
        static long _entries[] = { 2, (long)x, 8 };
        return _entries;
    }
}
```

In this example _entries[0] and _entries[2] were not initialized correctly to 2 and 8, respectively. Instead, the fields were both were incorrectly initialized to 0.

The compiler inserts code into _GetEntries to initialize _entries the first time _GetEntries is called. Because it failed to insert the initializing code correctly, the code that initialized members with contant values was not being executed.

8.2 Enhancements, Changes, and Problems Corrected in Version 6.2-037

Enhancements, changes, and problems corrected are as follows:

- In certain cases, dynamic initialization of a static variable of aggregate type with an initializer list containing both constant and non-constant values did not occur correctly. Aggregate members initialized with constant values were initialized with 0, not with the constant values. Consider this example:

```
const float x = 3;

struct S {
    static long *_GetEntries() {
        static long _entries[] = { 2, (long)x, 8 };
        return _entries;
    }
}
```

In this example _entries[0] and _entries[2] were not initialized correctly to 2 and 8, respectively. Instead, the fields were both were incorrectly initialized to 0.

The compiler inserts code into _GetEntries to initialize _entries the first time _GetEntries is called. Because it failed to insert the initializing code correctly, the code that initialized members with contant values was not being executed.

- When generating EV6 code, the peephole optimizer could display an assertion failure complaining that an operand is not fixed or not float. This has been corrected. [6787]

- When generating EV6 code, the compiler produced a code pattern (specific to conversion between integer and floating types) that could produce incorrect results. This release corrects the problem. [6816]
- If the exception handling mechanism calls the two-parameter `delete operator` to clean up an allocated object that had an exception in the constructor, the mechanism now passes the correct size to the second (size) parameter. [6823]
- When compiling with `-std arm`, C++ treated the types `const char *&`, and `char *&` as equivalent. Effects-compatible types did not recognize this behavior. A new ARM-compatible flag for `types_are_compatible` with `effects_compatible_types` is now set when in ARM mode. The new flag specifies that all type qualifiers are ignored when comparing the type compatiblity of two pointer or reference types. [6949]

  The ARM mode of the compiler allows a `const char *` reference to reference a `char *` object, as show in the following example:

```
static char * f() {
    char *value = 0;
    const char * & d1 = value;  // d1 can reference value
    d1 = "abc";                 // changes value
    return value;               // should return "abc", not 0
}
```

  In standard mode, however, this behavior is not allowed, and the compiler did not recognize that assignments to the reference change the value of the object referenced. In the example, because the compiler did not recognize that assignments to `d1` would change value, it assumed that the assignment to `d1` did not occur and that the correct return value fo the function `f()` was 0.

  This version of the compiler fixes the problem.
- A change in the debugging symbol table produced by the C++ compiler causes all namespace members to be generated in the local symbol table of the file descriptor associated with the namespace, within the scope of the namespace. This change make it easier to debug namespaces. [LDB1569]
- Fix for unexpected null cleanup block error

  When an exception was raised and caught in the catch clause at runtime, some loops with a try-catch block would generate an internal error about an unexpected null cleanup block. This error was caused by the runtime environment and has been corrected.
- Fix for `raw_storage_iterator` assignment operator

In versions 6.2 and earlier, a problem in the assignment operator for the class `raw_storage_iterator` could cause a run-time seg fault if, for example, you called the algorithm `stable_sort()` more than once with the same container. The problem has been corrected. [10.1284]

- Fix for `basic_string::compare()` member functions

  In version 6.2, two of the `basic_string::compare()` member functions were throwing an exception if the length of the second string was longer than the length of the current string. This has been fixed so that they throw an exception only if the position the user specifies within the second string is greater than the length of the second string. [10.1287]

- Fix for `basic_string::resize()`

  A problem in the `basic_string::resize()` member function in Version 6.2 has been corrected. The incorrect behavior was that if two strings pointed at the same underlying `char*`, and the `resize()` function was called on one of them, and if you then changed the underlying value for one string, the value for the other string would also be changed. [10.1287]

- Fix to string assignment operator when assigning string with embedded nulls

  A problem in the `basic_string` assignment operator prevented strings containing embedded nulls from being copied correctly. The problem has been corrected. [10.1238]

- Fix for missing `set_new_handler()` prototype when specifying `-nostdnew`

  The prototype for the `set_new_handler()` function was missing from the `<new>` header file if you specified `-nostdnew` on the command line. The problem has been corrected.

8.3 Enhancements and Changes in Version 6.2

Version 6.2 includes the following enhancements and changes:

- Some functions needed for language compatibility run-time support have been moved from `libcxxstd.a` to the shared library `libcxx.so`. For details, see *Deploying Your Application* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Many compiler and library problems have been corrected (see Section 8.4).
- Many improvements have been made to cross-reference information generated by the compiler.

- The compilation system and run-time library now call destructors that are defined in shared images which have been closed correctly using `dlclose()`, with the linker option `-depth_ring_search`. This feature requires that a new `libcxx` be installed on your system with Version 6.2. Otherwise, the compiler reports the undefined symbol `__cxx_call_static_dtors`.
- The `-gall` option now outputs debugging information about unused variables, so that you can queries the debugger about them. For more information about this option, see *Using the -gall and -gall_pattern Options* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Debugging support for constant variables is improved. Although the compiler does not generate debug information for constant externs (because you might not be able to link), it does generate debug information for all constant variables whose underlying type is `int` or `float`
- Diagnostics from the driver during the link phase are now sent to `stderr` instead of to `stdout`.
- Consumption of virtual memory is significantly reduced when computing the addresses of deeply nested virtual base classes. [6473]
- The compiler diagnoses more cases of unreachable code. If you receive warnings you believe to be inappropriate, please report them. [6534]
- Cleaner header file inclusion policy

  This new version of the Standard Library is much cleaner in its inclusion of unnecessary headers. For example, the header file `<algorithm>` no longer includes `<functional>`. `<ostream>` and `<istream>` no longer include `<locale>`. Programs that used to count on these inclusions might break. You can correct them by explicitly including any header files you use in your own sources.
- New interface to `get_temporary_buffer` and `use_facet`

  Because the Version 6.2 C++ compiler now supports explicit template function arguments, it also supports the standard interface to the `get_temporary_buffer()` function. The following example shows how code must change:

  Change this:

  ```
  get_temporary_buffer(len,(T*)0); // two arguments
  ```

  to this:

  ```
  get_temporary_buffer<T>(len); // one argument
  ```

where "T" is the value type of the container.

The standard interface to the locale class `use_facet()` function is also now supported. The following example shows how code must change:

Change this:

```
use_facet(loc,(ctype<char>*)0);
```

to this:

```
// use_facet only takes one argument
use_facet<ctype<char> >(loc);
```

- Many common `iostream` and `locale` instantiations (for example, those based on the `char` type) have been put into the Standard Library to improve compile-time performance. If you want to instantiate them yourself and use your own instantiations (perhaps to have a debugging version), follow these steps:

  1. Compile with the macro `__FORCE_INSTANTIATIONS` defined (`-D__FORCE_INSTANTIATIONS`).
  2. Link with the option `-nopreinst`. The linker then finds the instantiations in your repository before it finds those in the Standard Library.

8.4 Problems Corrected in Version 6.2

This section summarizes compiler changes and the most important problems corrected in Version 6.2.

- Specifying `-noglobal_array_new` no longer causes generation of rogue cleanup handlers. [6095]
- Enum types larger than `int` are now supported when compiling with `-std arm`. [4499]
- Specifying `-v` now displays command-line, driver-generated, and predefined macros on standard output. The listing file now contains correct list of command-line and predefined macros, taking into account macro undefines and redefines. [4723] [5609]
- Within a template member function instantiation, the compiler now correctly calls a function with short pointer in `-xtaso_short` mode. The compiler no longer displays an error for the call.
- Some compiler-generated wrapper functions with the `__STF___default_version` prefix were not resolved in template automatic instantiation mode. The problem is corrected. [6362]

- Debugging support is now provided for anonymous union variables inside namespaces. In the following example, debuggers support referencing `s.a` and `s.b`:

```
struct S {
     union {
         int a;
         int b;
     }
} s;
```

  In the following example, the compiler generates two variables, `x` and `y`, which the debuggers can examine.

```
union {
    int x;
    char y[4];
};
```

  The compiler no longer generates tag names for tagless structs and unions.

- Under the following conditions, the compiler could generate code that returned the value of `i` before the store of the new value:

  1. Take the address of a variable, using a type other than the variable's type, `(p = (char *)&i)`.
  2. Assign to the variable using a pointer addition expression, `*(p + 0) = new_value`.
  3. Fetch the variable's value directly `(return i)`.

  The problem can occur only if the store using the pointer addition expression appears within an `if` statement, as in the following code:

```
int f(int flag) {
  int len = 1;
  if (flag) {
    char *ppp = (char *)&len;
    *(ppp + 0) = 2;
  };
  return len;
}
```

  The problem has been corrected. [6421]

- Bad code is no longer generated for an array reference within a template instantiation. [6394]

- The compiler no longer generates incorrect code for offsets to external arrays. [6386]

- The routine for new[] now calls delete[] if an exception occurs during construction. [6243]
- A compiler crash caused by a label statement in a switch statement has been corrected. [6614]
- Macro expansions are now output to the listing file when both -show expansion and -source_listing are specified. [6107]
- When a local stack is used in constructors, the compiler displays the warning "Initialization of references requires temporaries of automatic storage duration". [4522]
- Null characters in comments are now ignored in -std arm mode. The Version 6.0 compiler treated null characters in comments as errors. Outside of comments, null characters are now diagnosed as warnings in -std arm mode, and as errors otherwise. [3740]
- The compiler has implemented the _poppar builtin function and added code to convert output type for _poppar, _popcnt, _leadz, and _trailz to match contents of the UNIX builtins.h file. See *Built-In Functions* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- A template class name can now be reused as a nontemplate class in a private namespace without generating an error message. [6541]
- A Version 6.0 bug caused the the compiler to emit a union layout incompatible with earlier versions if the union contained bitfield of size 8, 16, 24 . . . up to the size of the type for the bitfield. For example:

```
typedef struct size4 {
        union {
            unsigned ttn :8;
            struct {
                unsigned incr :3;
                unsigned rep :5;
            } v1;
        } u;
    } size4;
```

  The size of size4 is 4 bytes for Version 5.7, and was incorrectly set to 1 since Version 6.0. This bug is fixed in Version 6.2. If you have such a bitfield in your code, you must recompile. [6567]
- If -nocleanup is specified, the compiler does not reference the destructors or delete operators used for cleanup code that is not generated. The -noexceptions option now implies -nocleanup. [6216]
- -std arm mode, the compiler no longer generates code to the left side of the -> and . operators if the right hand operand is a static member. [4469]

- For compatibility with Version 5.6, the ec_inaccessible_base_class error on function return expressions in -std arm mode is now disabled. [3744]
- The compiler now warns about differences in meaningless qualifiers.
- The driver now terminates if the compilation results in a fatal error or ctrl-c.
- In -std arm mode, the compiler no longer checks for suitable copy constructors of classes with volatile qualifiers. [6087]
- To maintain compatibility with arm, cfront, and ms modes, null preservation code is enabled for cases where a pointer should not be null. [4695]
- The compiler now checks for comparison of two incompatible enums. The compiler reports this condition with a new informational message ec_incompatible_enum_comparison. [6172]
- money_get/money_put locale facets now conform to C++ International Standard

  The money_get and money_put locale facets have been corrected to match the standard. In the previous version, for example, money_get appeared as:

  ```
  template <class charT,
       bool Intl = false,
       class InputIterator = istreambuf_iterator<charT> >
  class money_get;
  ```

  They now correctly match the standard, where the interface appears as:

  ```
  template <class charT,
       class InputIterator = istreambuf_iterator<charT> >
  class money_get;
  ```

  Note that the second template argument "Intl" has been removed. The member functions get() and put() now accept Intl as an argument.
- ios_base::openmode flags set to conform to Standard

  The standard file stream classes have been corrected to conform to the Standard with regard to setting the ios_base::openmode flags. In the previous release, it was possible to create a file for reading and writing with this code:

```
#define __USE_STD_IOSTREAM
#include <stdlib.h>
#include <fstream>

int main() {
        fstream fs("foo.out", ios_base::in
                | ios_base::out);
        fs << "abc" << endl;
        return EXIT_SUCCESS;
}
```

In the current release, this code works only if the file already exists. If the file does not exist, you need to also specify `ios_base::trunc`; that is, you must change the first line in `main()` to:

```
fstream fs("foo.out", ios_base::in | ios_base::out
          | ios_base::trunc);
```

This conforms to table 92 in the Standard, which specifies the "C" equivalent of the File open modes.

- Correction to `list::sort(Compare)`

  A bug in the `list::sort(Compare comp)` member function is corrected. Previously, if users supplied their own `Comparison` function object for the element of the list, the compiler issued a message stating that it required an operator< defined for the element type. This no longer occurs.

- `reverse_iterator` now matches the Standard

  `reverse_iterator` has been changed to match the standard. It now takes only one template argument of type `iterator` instead of five. Users must change existing code to remove the additional unnecessary arguments.

- `bitset` constructors no longer accept a `const char`

- A bitset can no longer be constructed with a `const char*` argument. For example, the following no longer compiles:

  ```
  bitset<32> b("111111111");
  ```

  The constructor that takes a string is a templatized constructor, and thus can perform type deductions only on exact matches, not conversions (for example, `const char*` to string). To make the code in the previous example compile with the current version, the argument must be explicitly cast to a string, as follows:

  ```
  bitset<32> b( string("111111111"));
  ```

- `assign(size_t)` removed from `vector`, `deque`, `list`

Previous releases of the Standard Library contained a member function called assign() inside the vector, deque, and list classes. This function accepted only a size_t argument. This has been removed, because it is not in the Standard. You must add an extra argument indicating the value you want assigned. For example, you change calls like the following:

```
v.assign(5); // where v is a vector<int>
```

to:

```
v.assign(5, int());
```

- allocator<>::deallocate(pointer) removed

  The member function allocator<>::deallocate(pointer) has been removed. The Standard requires two arguments for this member function. The second argument should be of size_type and have the same value as the first argument passed to allocator<>::allocate().

- basic_ios now initializes skipws|dec

  To conform to the Standard, the following basic_ios constructor constructs a basic_ios object and initializes the format control bits to skipws | dec:

  ```
  explicit basic_ios(basic_streambuf<charT, traits>*sb)
  ```

  Previously ,this constructor also initialized the bit indicating that output is right justified. Because the constructor is called while constructing any of the IOStream objects cout, clog, cerr, wcout, wclog, or wcerr, the difference is apparent if you examine the format control bits set after initializing one of these objects.

  Consider the following program:

  ```
  #include <stdlib.h>
  #include <iostream>
  using namespace std;
  int main()
  {
      cout << cout.flags() << endl;
      cout << clog.flags() << endl;
      cout << cerr.flags() << endl;
      cout << wcout.flags() << endl;
      cout << wclog.flags() << endl;
      cout << wcerr.flags() << endl;
      return EXIT_SUCCESS;
  }
  ```

  The output now indicates that only the skipws and dec format control bits are initialized. Previously it would have indicated that the right bit was also set.

- Some `iterator` classes removed

  The following classes no longer exist in the Standard and have been removed from library headers.

  ```
  reverse_bidirectional_iterator
  random_access_iterator
  bidirectional_iterator
  forward_iterator
  output_iterator
  input_iterator
  ```

  Use instead the template class iterator with the template argument category to indicate which type of iterator you are constructing.

- The default allocator argument changed for `basic_string`

  The default allocator argument for the class `basic_string` has been changed from `allocator<void>` to `allocator<charT>`. Any STL container constructed with an `allocator<void>` template argument no longer compiles, because the specialization of `allocator<void>` does not contain all the necessary typedefs.

- `strstream` now deletes underlying `strstreambuf`

  A problem has been corrected in the Standard Library `strstream` classes that prevented underlying `strstreambuf` (and thus the string) from being deleted when the `strstream` object was destroyed. The standard states that they should be deleted if `strmode & allocated` is true and `strmode & frozen` is not true.

  For example:

  ```
  #define __USE_STD_IOSTREAM
  #include <strstream>

  void func()
  {
    ostrstream myostr;
    myostr << "abc";
  }
  ```

  If you called `func()` the string “abc” was never deleted when the `myostr` stream was destroyed. This problem has been corrected. Note that the Class Library `strstream` classes have always deleted the underlying string.

- `sync_with_stdio()` function is static

In previous versions, the function sync_with_stdio() was incorrectly declared as a member function of ios_base. The function is now correctly defined as a static member function; it is no longer necessary to call it with the "->" or or "." notation.

8.5 Restrictions in Version 6.2

This release is not totally compatible with previous versions; source changes might be required. The following general restrictions apply for the current release:

- The C++ International Standard permits overriding a virtual member function based only on a derived class return type. The current release does not support this capability.
- Intrinsic bcopy generates warning

  Starting with C++ Version 6.2, the bcopy function has been made intrinsic to improve performance. On HP Tru64 UNIX Version 4.*n* systems, this change can cause the compiler to issue the following warning message:

```
cxx: Warning: /usr/include/strings.h, line 83: Expected type
     "void (const void *, void *, unsigned long) C" is
     incompatible with declared type "void (const char *,
     char *, int) C", function will not be made intrinsic
extern void bcopy __((const char *, char *, int));
------------^
```

  HP Tru64 UNIX Version 4.*n* systems provide two function prototypes for bcopy, one that conforms to the standard, and another, the system default, for compatibility with previous operating system versions. The compiler issues the warning when it encounters the nonstandard version. For details, see the bcopy(1) reference page.

  To suppress the message, you can do one of the following:

  - Use an ANSI-standard copy function such as memcpy
  - Enable the standard function by defining -D_XOPEN_SOURCE_EXTENDED -D_OSF_SOURCE before including the header file
  - Compile with the -std strict_ansi option
  - Use #pragma function(bcopy)

  Because the next major operating system release implements a change in the UNIX98 standards and provides only the standard definition, the first or second method is recommended. The third and fourth methods prevent the function from being made intrinsic, resulting in degraded performance.

- Some features implemented in Version 5.7 are not supported by the current compiler. See *Porting to HP C++* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Some objects might have their exception unwinding information set to a negative index in the cleanup table, resulting in a core dump at runtime if an exception is raised. [7091]
- The UNIX utility `nm` does not always work on HP Tru64 UNIX Version 4.*n* systems with object files generated by the current compiler. The symptom is a seg fault when `nm` attempts to dump its list. An alternative to `nm` might be the `odump` utility. For example, the command `odump -tv foo.o` lists all the symbols in the archive. See the `odump` reference page for more details on the `odump` utility. Another possible workaround is to compile your object file with `-g` (debugging), then recompile it without `-g`. That procedure might avoid triggering the `nm` bug.

  If you want to obtain a version of `nm` that corrects this problem, these are the patch kit identifiers:

OPERATING SYSTEM VERSION	PATCH ID
v40	OSF400-438
v40a	OSF405-438
v40b	OSF410-438
v40c	OSF415-438
v40d	OSF420-42

- If you use a non-default pointer size or member alignment and header files are not protected, the following warning is issued:

```
< cxx: Warning: A non-default pointer size or member alignment is
specified and the system header files are not protected. This may
yield unpredictable results. The protect_headers_setup script can
help. See the protect_headers_setup(8) reference page for details.
```

  The `protect_headers_setup(8)` reference page page is planned for a future release of HP Tru64 UNIX. For now, see Protecting System Header Files in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Restricted address space size for precompiled headers in some versions of HP C++.

  Some versions of HP C++ might restrict the size of the address space available for use by precompiled headers. If the compiler terminates with a message about mapped memory, this may be the problem.

To increase the address space available for precompiled headers you need to increase the mapentries limit.

Follow these steps to change the mapentries limit:

1. Become the root user.
2. Create a new file named /tmp/xxx and insert the following lines:

```
vm:
 vm-mapentries=5000
```

3. Execute the sysconfigdb command as follows:

```
# /sbin/sysconfigdb -f /tmp/xxx -m vm
```

4. Reboot the system.

This procedure increases the mapentries limit from the default 200 to 5000. Increasing the limit by that amount has no adverse effect on the system.

- When using PCH, the line number associated with an inline function is the line at which the PCH stop is encountered.
- Certain utilities, such as GNU Make, do not recognize compressed object files, which the compiler creates by default. If you encounter problems, you can specify the -nocompress command-line option.
- Instantiating function templates with array types can result in different external name encoding than with C++ Version 5.*n*. To avoid link errors, recompile the template definition with the current version of the compiler.
- Linker -no_archive does not work with HP C++

  The switch does not work because the Standard Library, libcxxstd.a, is supplied only in archive form.

  Specifying -no_archive causes the cxx drive to issue the following messages:

```
ld (prelink):
Can't locate file for: -lcxxstd
ld:
Can't locate file for: -lcxxstd
```

- Class and Standard Libraries generate Third Degree messages. *Appendix C* in *Using HP C++ for Tru64 UNIX and Linux Alpha* describes these messages.
- Do Not Use Standard Library template Definition File Names

The Standard Library supplies the following template definition files in /usr/include/cxx:

algorithm.cc	fstream.cc	streambuf.cc	vector.cc	time.cc
bitset.cc	ios.cc	locimpl.cc	string.cc	ctype.cc
istream.cc	numbrw.cc	tree.cc	collate.cc	messages.cc
complex.cc	iterator.cc	ostream.cc	valarray.cc	money.cc
deque.cc	list.cc	sstream.cc	valimp.cc	numeral.cc
rwlocale.cc				

If you use the same prefix name for any of your local files and have the directory that contains them in your include search path before /usr/include/cxx, the automatic instantiation mechanism picks up your local copy and does not correctly find the library files. (See *Using HP C++ for Tru64 UNIX and Linux Alpha* for more information about how the prelinker finds template definition files.)

Do not use any of these names as source file names for your application.

- Redeclaration of Standard Library Functions

  Many of the prototypes in the Standard Library have been changed to conform to the C++ International standard by the addition of exception specifications. This means that if you have redeclared the declarations in your own code, you need to add the correct exception specification in order to match what's declared in the header.

  For example:

```
#include <new.h>

// override default operator new
// this gives an error
inline void* operator new(size_t s);
```

  To prevent this, you'd need to change your new() declaration to:

```
inline void* operator new(size_t s) throw(std::bad_alloc);
```

- Files/Macros for Internal Use Only

  HP C++ Version 6.1 ships the following non-Standard headers which are for internal use only. Their contents are subject to change and can not be relied upon.

<stdcomp>	<stl_macros>	<stddefs>	<compnent.hxx>
<stdmutex>	<stdexcept>	<lochelp>	<locimpl>
<locimpl.cc>	<valimp>	<valimp.cc>	<vendor>
<codecvt>	<codecvt.cc>	<collate>	<collate.cc>
<ctype>	<ctype.cc>	<locvector>	<math>
<messages>	<messages.cc>	<money>	<money.cc>
<numeral>	<numeral.cc>	<random>	<rwcats>
<rwdispatch>	<rwlocale>	<rwlocale.cc>	<rwstderr>
<rwstderr_macros>	<string_ref>	<time>	<time.cc>
<traits>	<usefacet>		

In addition both Standard and non-Standard headers make use of macros beginning with `_RW` or `__RW`. These `_RW*` and `__RW*` macros are for internal use only. They are subject to change and can not be relied on.

- The size of long double has changed in *HP Tru64 UNIX* Version 5.0. If you are using the standard iostreams or locale library for inputting or outputting long doubles, you must specify `-nopreinst` on your link line to obtain instantiations that work correctly.

  A program using the standard iostreams (not necessarily for long doubles) which dynamically loads a shareable which is using the standard iostreams for inputting or outputting long doubles, must also be linked `-nopreinst`

- Use of exception in <math.h> and Standard C++ Library

  In HP C++ Version 6.1, namespaces resolve a conflict between the name exception found in the header `<math.h>`, which represents a structure, and the name exception found in the C++ Standard Library header `<exception>`, which represents the Standard Library exception class. The structure exception from `<math.h>` is visible in the global namespace. The Standard Library exception class from `<exception>` is visible in the `std` namespace.

  Because of this behavior, if you include, directly or indirectly, both of the headers `<math.h>` and `<exception>` you must qualify actual uses of `exception` to avoid name conflicts:

```
#include <math.h>
#include <exception>
int main () {
  ::exception e;     // exception from math.h
  std::exception e1; // Standard Library exception class
  return 0;
}
```

- collate_byname<wchar_t>::do_transform()

  The function collate_byname<wchar_t>::do_transform() seg faults if you use it with any locale other than the "C" locale. This happens because of a bug with the underlying C function wcsxfrm(). No workaround currently exists for this problem.

- ctype_base::graph

  In the current implementation of the ctype_base class, a character has a graph property if and only if it also has an alpha, a digit or a punct property. Thus, mathematical and scientific symbols and dingbats from the UNICODE character set will not be classified properly.

- IOStreams cannot output IEEE NaNs/Infinities

  The Standard IOStreams Library does not support output for IEEE NaNs and Infinities.

- ios_base::out does not truncate a file to zero length

  The ios_base::openmode ios_base::out should open a file for output. This means that the file is either truncated to zero length if it exists or created for writing if it does not. Therefore ios_base::out is the same as ios_base::out | ios_base_trunc.

  With our sources, the following program behaves incorrectly.

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream fs ("t.in",ios_base::out);
    fs << "A";
    return EXIT_SUCCESS;
}
```

  Where t.in contains xyz.

  After running this program, t.in contains

```
Ayz
```

  It should contain

```
A
```

  You can work around this problem by replacing ios_base::out with ios_base::out | ios_base::trunc.

- Overriding operator new in library

You can define a global `operator new()` to displace the version used by the C++ Standard Library or C++ Class Library. For instructions, see *Overriding operator(new)* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- The `-define_templates` and `-tall` options are not guaranteed to work with the Standard Library. Use automatic instantiation or specify `-tused` instead.

  When compiled with `-define_templates` or `-tall`, the following code generates compilation errors indicating that no operator "<" (and ">" or "+" or "-") matches these operands:

  ```
  #include <map>
  map<int,int> foo;
  ```

  The instantiation options are not guaranteed to work with the Standard Library because they request the compiler to instantiate all templates, even those that are not used.

  The `-tall` option does not work because `rb_tree`, the underlying implementation of `map` and `set` supports a bidirectional iterator class. Thus, `operator+`, `operator-`, `operator<` and `operator>` are not defined in the iterator for that class.

  When you instantiate the tree with `cxx -tall` or `cxx -define_templates`, the compiler attempts to instantiate recursively everything that is `typedef`ed, even if not used. Thus, the tree contains a `typedef` for `std::reverse_iterator<iterator>`, which then instantiates the global class `reverse_iterator` with the tree iterator as the template argument `RandomAccessIterator`, a misnomer in this case.

  This behavior generates undefined symbols for these operators because they are used within the definition of the operator member functions inside `reverse_iterator`. The compiler therefore attempts to instantiate them even though they do not exist.

  Specifying `-tused` for the Standard Library directs the compiler to only instantiate those templates that are used.

- Description of the initial state of the `stringstream ctor`

  There has been some controversy in the `comp.lang.c++` reflector and within the standards committee on the semantics of the `stringstream` constructor. Consider the following example:

```
#define __USE_STD_IOSTREAM
#include <stdlib.h>
#include <sstream>

int main() {
    ostringstream ost("Hello, ");
    // ost.rdbuf()->pubseekoff(0,ios::end,ios::out);
    ost << "world!";
    cout << ost.str() << endl;
    return EXIT_SUCCESS;
}
```

Depending on the setting of the `streambuf` `put` pointer after the initial construction, the program could print either "Hello, world!" or "world!". The Rogue Wave (and HP C++) interpretation is that the `stringstream` constructor does not change the initial position of the `streambuf` pointer, so that the program prints "world!".

If you want to change the setting of the `put` pointer to match the other interpretation (that the pointer should move to the end of the initializer string), you must insert a call to `pubseekoff()`, as shown in the commented line.

If the ANSI C++ committee issues a clarification on this matter HP C++ will implement their decision.

- Linker might produce multiply defined symbols from Standard Library

  The linker might produce multiply defined symbols from the Standard Library when linking against your own shared library or libraries. If you find multiply defined symbols like the following coming from `libcxxstd.a`, you might be creating one or more shared libraries and then using those shared libraries to create an executable.

  ```
  ld:
  /usr/lib/libcxxstd.a(typeinfo.o):
       std::__vtbl_3std9type_info: multiply defined
  /usr/lib/libcxxstd.a(vec_newdel.o):
       __vec_new_eh: multiply defined
  ```

  The problem is that `libcxxstd.a` is an archive library. When you create a shared object that references this library, the symbols are pulled into that shared object. When you then use the shared library, it finds the symbol both in your shared object and in the `libcxxstd.a` library archive. To work around this problem, follow these steps:

  1. If you are creating multiple shared libraries, create them without linking in the `libcxxstd` library and with the `expect_unresolved ld` flag to prevent linker errors. For example:

```
cxx -shared -nocxxstd -expect_unresolved "*std*" -o libA.so obj1.o
cxx -shared -nocxxstd -expect_unresolved "*std*" -o libB.so obj2.o
.
.
.
```

2. Link the final executable normally, so that unresolved symbols from libcxxstd are linked in. For example:

```
cxx -L. main.cxx -lA -lB
```

A release of libcxxstd as a shared library is planned once once the library interface is stablilized.

- Compilation warnings and errors with auto_ptr

  Using auto_ptr in non strict_ansi mode generates warnings about initializing a non-const ref with an lvalue. These warnings are due to the lack of enforcement of the rule in the C++ standard that binding a reference to a non const to a class rvalue is illegal. To make auto_ptr work correctly, you must compile the module(s) that use auto_ptrs in ansi or strict_ansi language mode.

  In strict_ansi mode, you may still encounter compilation errors when converting an auto_ptr<Derived> to an auto_ptr<Base>. For example, this does not work:

```
struct Base {};
struct Derived : Base {};

auto_ptr<Derived> source2() {return auto_ptr<Derived>(new Derived);}

int main() {
  auto_ptr<Derived> d;
  auto_ptr<Base> b(d); // compiles
  auto_ptr<Base> p3(source2()); // doesn't compile
  return 0;
}
```

  This is a known deficiency of the auto_ptr class, see language issue 84 at:

```
http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/core-issues.htm
```

  for a discussion of the problem. You must either use casts or avoid temporaries under these conditions.

  Remember that you should not use auto_ptrs as elements of an STL container, because they do not meet the CopyConstructible and Assignable requirements for Standard Library container elements. Compilation warnings occur if, for example, you try to insert an auto_ptr into a container. See

```
http://www.gotw.ca/gotw/025.htm
```

and

```
http://www.awl.com/cseng/titles/0-201-63371-X/auto_ptr.html
```

for discussions of the history and current restrictions of the `auto_ptr` class.

9 Release Notes for the V6.1 C++ Compiler

9.1 Problems Corrected in Version 6.1-029

This section summarizes compiler changes, enhancements, and the most important problems corrected in Version 6.1-029.

- The optimization phase of the compiler has been changed to disable a specific optimization that caused ATOM based tools to behave unpredictably. This optimization may be restored once these tools have been enhanced to expect this optimization.
- New warning messages indicate when assigning from a larger to a smaller data type might cause a truncation. These messages facilitate porting from 32-bit to 64-bit platforms. [3971]
- A workaround has been implemented to prevent the linker from erroneously declaring too many GOTs during the prelink phase. [6027] [43-4-305]
- Templates made friend are now instantiated with external linkage when `-timplicit_local` is specified. [6056]
- Access checking for pointer to members in ARM mode has been relaxed to be more compatible with Version 5.*n*. [6088]
- Calling `operator(new)` on an array of objects allocates extra header information to describe the array so that `operator delete` can call destructors on the elements. In this release, the header has been made compatible with object code generated with version 5.*n* releases. [6151]
- During the prelink phase, the driver no longer splits linker output lines at 8000 characters. [6166]
- The driver no longer suppresses error messages during the prelink phase. As a result, messages might apprear multiple times. This new behavior helps prevent silent or confusing failures. [6168]
- A problem unwinding from multiple exceptions has been corrected. [6185]
- A compiler fatal error using volatile structures has been corrected. [6186]

- The cxx driver no longer runs post-link phases that are unnecessary to determine required templates. [6191]
- A problem with interaction between -vptr_size_short and -xtaso_short has been corrected. [6195]
- If a local static object is referenced by more than one template, the compilation no longer results in an unresolved symbol for the __fini routine. The routines to call the destructors for static objects, which are prefixed with __fini, are now allocated properly with automatic instantiation mode. [6278]
- list::sort(Compare comp) member function now correct

  Previously, if users supplied their own Comparison function object for the element of the list, the compiler still required an operator< defined for the element type. This is no longer the case.
- strstream now correctly deletes underlying strstreambuf

  The Standard Library strstream classes now delete the underlying strstreambuf (and string) when a strstream object is destroyed. The standard states that they should be deleted if strmode & allocated is true and strmode & frozen is not true.

  For example:

```
#define __USE_STD_IOSTREAM
#include <strstream>

void func()
{
  ostrstream myostr;
  myostr << "abc";
}
```

  If you called func() the string “abc” was never deleted when the myostr stream was destroyed. This problem has been corrected. Note that the Class Library strstream classes have always deleted the underlying string.
- stringstream.str() can now be accessed after seekp(0)

  The underlying string in a stringstream can now be accessed properly after a seekp(0). For example, the following program no longer causes a run-time core dump.

```
#include <iostream>
#include <sstream>

int main()
{
   std::ostringstream out;
   out << std::string("This is a test");
   out.seekp(0);
   std::cout << "out.str(): " << out.str() << std::endl;
   return EXIT_SUCCESS;
}
```

- <wchar.h> can be included before <fstream>

  Including <wchar.h> before <fstream> no longer causes compilation errors.

- std::ostrstream::freeze() no longer leaks memory

  The std::ostrstream::freeze() function no longer leaks memory when used in combination with an ostrstram object. If you declare an ostrstream object as std::ostrstream out, the memory to which output is written is dynamically allocated. Previously, calling freeze(false) incorrectly froze a stream, thereby preventing the deletion of allocated memory. This is no longer the case.

  The following program (which would have illustrated the memory leak when instrumented with third degree) no longer leaks memory.

```
#include <stdlib.h>
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iostream>
#include <strstream>
void TestIt()
{
   std::ostrstream out;
   out << std::string("This is a test");
   out.freeze(false);
   char *s = out.str();
   return;
}
int main()
{
   for (int i = 0; i < 100; i++)
       TestIt();
   return EXIT_SUCCESS;
}
```

- Math overloads in <valarray> OK

Including <valarray> and then calling some of the math routines (for example, abs()) with an int argument might have produced an ambiguity error. This has been corrected.

- Smaller executables for users of the basic_string library component

  The Standard Library now provides a better separation of the STL and Standard IO components. As a result, users of the basic_string component of the Standard Library should obtain smaller executables. In Version 6.1, using only a string in an application caused all iostreamand locale object files from the Standard Library to be included in the executable. These unnecessary files are no longer included.

9.2 Problems Corrected in Version 6.1

This section summarizes compiler changes, enhancements, and the most important problems corrected in Version 6.1.

- The Version 6.0 compiler incorrectly mangled the name of a class virtual function table when that class was nested within another class or namespace. If object modules compiled with Version 5.*n* and Version 6.0 were linked together, the compiler might display the following message:

  ```
  ld:
  Unresolved:
  std::__vtbl_3std9type_info
  ```

  This problem has been corrected. Any code generated with Version 6.0 that demonstrates this problem should be recompiled with the current compiler. [5144]

- The compiler used to mark all member functions of the class that had not already been specialized as needing to be instantiated when processing a #pragma define_template class <args>. But the compiler did not instantiate the functions until processing the end of the scope. If a specialization occurred after the #pragma but before the compiler tried to instantiate it, the compiler found that the member function had already been specialized and reported the following error:

  "*function*" cannot be instantiated -- it has been explicitly specialized

  This error is reduced to a warning and is issued only once per instantiation. The warning can be suppressed by using the command-line switch -msg_disable 490. See *HP C++ Implementation* in *Using HP C++ for Tru64 UNIX and Linux Alpha*. [43-4-137]

- The compiler no longer generates incorrect code when dereferencing an element of an array of pointers. [5236]

- A problem creating shareable libraries has been corrected by allocating static local `const` variables initialized with pointers to `.rdata` instead of to `.rconst`. [43-4-166]
- To facilitate setting default compiler flags, you can now create an optional configuration file named `comp.config` or an environment variable named `DEC_CXX`.
  - The `comp.config` file allows system administrators to establish a set of compilation flags that are applied to compilations on a system-wide basis. The compiler flags in comp.config must be specified on a single line, and the comp.config file should be stored in the compiler target directory, `/usr/lib/cmplrs/cxx`.
  - The `DEC_CXX` environment variable allows users to establish a set of compilation flags that are applied to subsequent compilation on a per user basis.

    The `DEC_CXX` environment variable can contain two distinct sets of compilation flags separated by a single vertical bar (|). The flags before the vertical bar are known as prologue flags and the flags after the bar are know as epilogue flags.

    The `DEC_CXX` environment variable can begin or end with a vertical bar, or have no vertical bar at all. If no vertical bar is present, the flags are treated as prologue flags by default. Any vertical bar found after the first vertical bar is treated as whitespace and a warning is issued.

  During a compilation, compiler flags are processed in the following order:

  1. `comp.config` flags
  2. `DEC_CXX` prologue flags
  3. command line flags
  4. `DEC_CXX` epilogue flags If `-v` is specified on the command line, the contents of DEC_CXX and `comp.config`, if present, are displayed. [4488]
- By default, C files are compiled with `-std`. Users can now override the default by specifying `-std0`/`-std1`. [4392]
- The compiler no longer mishandles "?" operations on a boolean type in constructs like that in the following example. [43-4-135]

```
bool f(int i)
{
   return (i<5 ? false : true );
}
```

- Parallel compiles now work correctly because the compiler no longer deletes an empty template repository. [5544]
- The current version does not support the `-show statistics` option implemented in Version 5.7.

  The following are now supported:

  `-nocpp`
  `-show expansion`
  `-xref`
  `-xref_stdout`
  listing of predefined macros
  `#pragma message`
- STL map containers are now accessible in multithreaded environments

  Erasing elements from two STL map containers in a multithreaded environment no longer causes a segmentation fault. With previous versions, the problem was caused by an incorrect locking mechanism in an implementation class used when a map is being “erased”.

  To correct the original problem reported against the STL map class, we reworked map’s entire underlying implementation class. The underlying implementation class is found in the `<tree>` header. This class has been modified so that it no longer contains static data members.
- Standard Library IOStream `cerr.flush()` no longer hangs in multithreaded environments

  Applications making use of the Standard Library IOStream `cerr.flush()` function in combination with other I/O related activities in a multithreaded environment no longer experience hangs in their executing code.
- Piping standard output or standard error to a file no longer loses output when using Standard Library IOStreams

  The symptoms of this Standard Library bug varied. One noticeable effect could be seen when executing a program using Standard Library IOStreams which inserted items to `cout` and then piped this output along with other output to a file.

For example, if a program inserted into `cout` as follows:

```
#define __USE_STD_IOSTREAM
#include <stdlib.h>
#include <iostream>
int main() {
  cout << "Hello from C++" << endl;
  return EXIT_SUCCESS
}
```

And if a script, `d.sh` executed the above program and made use of a command that output to Standard Error like this:

```
a.out
ls -X
```

And if you piped both Standard Output and Standard Error to a file like this:

```
d.sh >& testlog
```

The output to Standard Output was lost. The `testlog` would contain:

```
+ ls -X
ls: illegal option -- X
usage: ls [ -1ACFLRabcdfgilmnopqrstux ]  [files]
```

Rather than:

```
Hello from C++
+ ls -X
ls: illegal option -- X
usage: ls [ -1ACFLRabcdfgilmnopqrstux ]  [files]
```

This problem has been corrected.

- Support for C++ International Standard `iostream` and `locale`

  The Version 6.1 kit contains detailed documentation in PostScript and PDF formats:

  ```
  /usr/share/doclib/cplusplus/intzln.ps
  /usr/share/doclib/cplusplus/intzln.pdf
  /usr/share/doclib/cplusplus/locale.ps
  /usr/share/doclib/cplusplus/locale.pdf
  ```

- Reference pages for the Standard Library. You can access them by typing `man cxxlibstd_intro`.
- Several new options (see *The C++ Standard Library* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.)
- pre-ANSI/ANSI iostreams compatibility.

- Support for ANSI/pre-ANSI `operator new()`.
- Support for global `array new` and `delete`.
- Support for `long long` and unsigned `long long` types.

  The non-ANSI standard types `long long` and unsigned `long long` are now supported in the Standard Library iostreams as well as being valid types for `numeric_limits` specializations and as types for which `destroy()` specializations are provided. For example, you can now say:

  ```
  long long  l;
  cout << l << endl; // compiles without error
  ```

  Note that these types are not supported under `-std strict_ansi` or `-std strict_ansi_errors` compiler mode.

  `long long` and `unsigned long long` are also supported in the `iostream` class library in the default compiler mode.

- Common instantiation libraries no longer needed

  Because Version 6.*n* and higher uses compile time rather than link time template instantiation, there is no use for common instantiation libraries. No `build_common_instantiation_library` script is supplied. See *Using Templates* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- Interface change for STL distance() function

  Because the Version 6.1 compiler now supports partial specialization of class templates, the interface to the `distance()` algorithm has been updated to conform to the latest C++ standard. This means that previously, if you made a call to `distance`, the result was returned by using a reference argument for the third argument:

  ```
  // pre 6.1 the result was returned in d
  distance(first,last,d);
  ```

  Beginning with V6.1, the result is returned in the return type:

  ```
  d = distance(first,last); // 6.1
  ```

  You must therefore change all your calls to `distance()`.

- Specific changes to match the November C++ International Standard:
  - `iterator_traits::distance_type` now `iterator_traits::difference_type`

    The name of the typedef inside the classes `iterator_traits` and `iterator` that specifies the type of the result when two iterators are subtracted has been changed from `distance_type` to `difference_type`.

  - `typedef` name changes in iterator classes

The names of some of the typedefs in the reverse_iterator and reverse_bidirectional_iterator classes have changed as follows:

iter_type is now iterator_type
reference_type is now reference
pointer_type is now pointer
distance_type is now difference_type

- slice/gslice classes member function length() name change

  The member function length() in the slice and gslice classes has changed its name to size().

- has_denorm data member of the numeric_limits class changed The has_denorm data member of the numeric_limits class, <limits>, has been updated. has_denorm is changed from type bool to an enum type, float_denorm_style, to reflect that support for denormalized values might not be detectable at compile time. The float_denorm_style type looks like this:

```
namespace std {
    enum float_denorm_style {
        denorm_indeterminate = -1;
        denorm_absent = 0;
        denorm_present = 1;
    };
}
```

  The values representing the presence or absence of denorms are as follows:

```
denorm_indeterminate: cannot determine if type supports
                      denormalized at compile time
denorm_absent:        the type does not support denormalized values
denorm_present:       the type supports denormalized values
```

- Default allocator value for map and multimap has changed

  The default value for the template argument Allocator in map and multimap has changed from allocator<T> to allocator<pair<const Key, T> >.

- Some Standard Library ambiguities corrected

  The Standard Library vector, deque and list containers no longer generate ambiguities when using their constructors or insert member functions. However, if you declare a basic_string of int, you might encounter ambiguities in the constructors, append, assign, insert, and replace member functions. For example if you write:

```
#include <stdlib.h>
#include <string>
int main() {
    basic_string<int> si (5,0);
    return EXIT_SUCCESS;
}
```

A compilation error results from the overload resolution between integral and iterator types. The constructors involved are:

```
  // construct n elements and initialize with value
  basic_string(size_type n, charT c,
               const Allocator& a = Allocator());

  // construct a vector using iterator ranges
  template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
```

The compiler matches on the constructor `basic_string(InputIterator begin, InputIterator end, ...)` The reason is that `size_type` a `size_t` is an unsigned long on HP C++. So you have:

```
    basic_string(unsigned long, int, ...)  vs.
    basic_string(int, int,...)
```

The second constructor is the better match, but it is not what we want, because we are not constructing with iterator ranges.

The workaround is to avoid matching on iterator types by casting integral arguments. So for example, the previous program would compile correctly if the size argument were cast to a `(size_t)`:

```
#include <stdlib.h>
#include <string>
int main() {
    basic_string<int> si ((size_t)5,0);
    return EXIT_SUCCESS;
}
```

9.3 Enhancements and Changes in Version 6.0

This section briefly summarizes changes and enhancements made in Version 6.0. For information about compatibility issues that you might encounter using V6.0 if you have used Version 5.*n* in the past, refer to *Porting to HP C++* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- Support for the C++ International Standard (with some differences, as described in Section 2.3), including the C++ Standard Library. See Section 3 for information about and changes to the Standard Library.
- Language mode options

For compatibility with previous versions, the compiler supports an ARM language mode and provides both `-std ansi` and `-std arm` language mode options, as well as options to support other C++ dialects. For details, see *Porting to HP C++* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.

- Improved automatic instantiation of templates, including fewer restrictions. In particular, HP C++ no longer requires that template declarations and definitions appear in header files. For details, see *Using Templates* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Support for precompiled headers to decrease compilation times. For complete information, refer to *Precompiled Headers* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Performance optimization options

  To improve performance, the compiler provides the following options:

  - `-[no]ansi_alias`
  - `-assume [no]pointers_to_globals`
  - `-assume [no]whole_program`

  For details, see the cxx(1) reference page.
- Run-Time type identification

  The compiler emits type information for **Run-Time type identification** (RTTI) in the object module with the virtual function table, for classes that have virtual function tables. For details, see *Run-Time Type Identification* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.
- Changes to behavior of the `-gall` option. See *Using the -gall and -gall_pattern Options* in *Using HP C++ for Tru64 UNIX and Linux Alpha*.