HP C++ Version 7.3

Release Notes for OpenVMS Alpha

June 14, 2007

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

Hewlett-Packard Company Palo Alto, California

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

Contents

1 Introduction

This document contains the release notes for HP C++ Version 7.3 for OpenVMS Alpha. The HP C++ product requires OpenVMS Alpha Version 7.3-2 or higher.

The HP C++ media contains the following:

- The Version 7.3 kit, which includes the HP C++ compiler and associated files, such as the Class Library and Standard Library header files.

HTML files are provided for the release notes and some of the product manuals for use with a web browser. To view this documentation, point your browser to

```
file:/sys$common/syshlp/cxx$help/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 Standard Library versions
- Differences between compiler versions
- Differences between HP C++ and the C++ International Standard
- Retirement of `cfront` language dialect

The next sections discuss these differences.

2.1 Standard Library Differences

Any code that references the C++ Standard Library (any of the STL containers or algorithms, standard iostreams or locales) that was compiled using version V6.2 or earlier of the HP C++ compiler must be recompiled and relinked in order to be used with the code compiled with HP C++ Version 6.3 or later.

For example, if you have an object library, an object module, or a shareable image compiled with HP C++ Version 6.2, you must to recompile it (and relink in the case of a shareable image) before an application compiled with Version 6.3 or later can use it.

As stated earlier, this restriction applies only to code that references the C++ Standard Library; it 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.

Beginning with HP C++ Version 6.3, the Standard Library is guaranteed to be link-compatible in all subsequent releases.

For applications that will relink on the end-user's system, see *Deploying Your Application* in *HP C++ User's Guide for OpenVMS Systems* for information about redistributing C++ Run-Time Library components.

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 *Appendix E Compiler Compatibility* in *HP C++ User's Guide for OpenVMS Systems*.

- Language differences

  The compiler implements (with some differences, as described in Section 2.3), the C++ International Standard, which differs significantly from the language specified in the ARM (*The Annotated C++ Reference Manual*, 1991, by Ellis and Stroustrup) and implemented by the Version 5.*n* compilers. 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 existing source code with minimal source changes, compile using the `/standard=arm` option. See *Appendix E Compiler Compatibility* in *HP C++ User's Guide for OpenVMS Systems*.

  Note

  The installation procedure checks whether a Version 5.3 to 5.6 compiler is installed on your system. If so, it asks whether you want to save it, and if so, where. The default save area for a Version 5.6 compiler is SYS$COMMON:[CXX056].

  If you find that Version 6.*n* or higher requires excessive changes to your applications even when you use the `/standard=arm` option, or if you encounter problems using the Version 6.*n* or higher compiler, you can return to your previous C++ environment by invoking the command procedure

  ```
  SYS$COMMON:[CXX0nn]COMPILER_SETUP.COM
  ```

  where *nn* specifies the version of your previous compiler.

- Diagnostic differences

The Version 6.*n* or higher compiler does more error checking than Version 5.6 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 5.6, compile with the `/quiet` option. See *Message Control and Information Options* in *HP C++ User's Guide for OpenVMS Systems*.

- 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 V5.*n* instantiation process with a V5.*n* compiler before linking to code compiled with Version 6.*n* or higher. See the *Using Templates* section in the Compiler Compatibility Appendix of the *HP C++ User's Guide for OpenVMS Systems*.

2.3 Differences Between HP C++ and the C++ International Standard

The following items, specified in the C++ International Standard, are not supported in Version 7.3 but may be supported in a future version. Details about each item are provided in the indicated sections of the *C++ International Standard* document and *The C++ Programming Language, 3rd Edition* by Bjarne Stroustrup.

- The `export` keyword for templates (*Standard* §14, paragraph 6; Stroustrup §9.2.3)

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 is 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 D* in in *HP C++ User's Guide for OpenVMS Systems*.

Thread Safety

The Standard Library provided with this release is thread safe but not thread reentrant. Thread safe means that all library internal and global data is protected from simultaneous access by multiple threads. In this way, internal buffers as well as global data like `cin` and `cout` are protected during each individual library operation. Users, however, are responsible for protecting their own objects.

According to the C++ standard, results of recursive initialization are undefined. To guarantee thread safety, the compiler inserts code to implement a spinlock if another thread is initializing local static data. If recursive initialization occurs, the code deadlocks even if threads are not used.

4 Release Notes for the V7.3 C++ Compiler

The following are enhancements, changes, and restrictions for the C++ compiler environment in this version:

- A problem has fixed which could sometimes cause the compiler to crash when using the _MB builtin functions with /OPT=LEVEL=2 and higher.
- The compiler previously asserted when processing a function that belonged in a overload set, but which had not yet resolved to the specific function selected by the overload, and its parameters were not pointers or references. This has been fixed.
- A recent fix exposed another bug where the compiler failed to appropriately mark a pointer-to-member function call which returns a class with a non-trivial copy constructor by value, so the 'this' pointer and the pointer to the temporary created for the class object, were passed in an inverted order. This too has been fixed.
- The compiler bug where pointers of the correct size were not created in 64-bit modes, has been fixed. This bug typically manifested in the ?: conditional operator expressions, when either the second or the third operand of the expression dereferenced a pointer expression.

- When creating a pointer-to-member call, and the pointed member function was a function that returned a value via a copy constructor, the synthesized call would generate an ACCVIO when run, because the compiler did not correctly handle the transformed argument (for the value returned by the copy constructor). This has been fixed.

- Two other problems where the compiler was not generating pointers of the correct size in 64-bit modes have been fixed.

- Known Problem: In some situations, the compiler's loop unrolling optimization might generate memory accesses in the code stream that never actually execute at run-time, but which would violate the ANSI Aliasing rules if they were to execute. In such a situation, the compiler might emit an erroneous BADANSIALIASn message, where n is a number or is omitted.

  If the violations take place only in machine instructions that will not execute at run-time, these messages can be safely ignored.

  To determine whether or not particular instances of a BADANSIALIASn message are erroneous, recompile the module with the /OPT=UNROLL=1 qualifier. Any BADANSIALIASn messages that disappear under that qualifier can be safely ignored, so you may want to add appropriate "#pragma message" directives to the source, localized to the specific source lines known to be safe. This is preferable to disabling the message for the whole compilation, since in all other cases the message indicates a real potential for code generation that will not work as intended. And this is generally preferable to disabling the ANSI_ALIAS or loop unrolling optimizations, since that would likely degrade performance, although the amount of degradation is not predictable, and in unusual cases it might even improve performance. As always, when making changes to performance-critical code, it's best to measure the impact.

- In previous versions of the compiler, specifying /STAN=STRICT on the command-line disabled the /NOWARNINGS qualifier. This is no longer done.

- The compiler now emits a warning when encountering __declspec specifiers, which were never supported, but were silently ignored.

- The compiler incorrectly failed to compile code constructs with __builtin_va_start when in the /STANDARD=GNU mode. This has been fixed.

- In 64-bit compilations only, the compiler would produce incorrect code if a pointer to a const object was deferenced, its constness cast away with a const_cast to a non-const reference of the same object type, and an attempt was made to bind a value to this reference object. This has been fixed.

- The compiler now diagnoses some cases where long pointers were converted to short pointers, where previously these were not diagnosed.

- The compiler no longer emits incorrect MAYLOSEDATA diagnostics for some simple expressions, such as assigning the address of a local variable to a short pointer.

- Previously, the compiler incorrectly created short pointers when arrays were decayed in certain expressions. This has been fixed.

- The compiler now implements the partial ordering specification resolved by Core Issue 214 of the C++ Standard.

  This is demonstrated by the following program:

```
extern "C" int printf(const char *, ...);

typedef unsigned int size_t;
template <typename T>
void func1(const T&) { printf("T\n"); }
template <typename charT, size_t N>
void func1(charT(&)[N]) { printf("charT[N]\n"); }

template <typename X, typename T>
void func3(const T&) { printf("T\n"); }

int main() {
  func1("hello");
  func3<int>("hello");  // error before
}
```

- The compiler previously generated incorrect source location information, which made the debugger behave as if the user said "step/into" when using the command "step". This made stepping over inlined constructors and thunks particularly difficult. This has been fixed.

- When using the /MODEL=ANSI object model, the compiler sometimes passed to member functions the "this" pointer and the "pointer to struct" returned by value, in an incorrect order. This has been fixed.

- The compiler incorrectly produced an OPNDNOTCLS diagnostic for certain code constructs when compiling in the /STANDARD=GNU dialect. This has been fixed.

```
enum ia64_instruction_constants {
        instruction0_shift = 5
};

class uint128_t {
    unsigned lo;
public:
    uint128_t(unsigned l) : lo(l) { }
    unsigned low() { return lo; }
    operator unsigned() const { return lo; }
    uint128_t operator <<(int sa);
};

void insert_break() {
    uint128_t break_raw(167);
    (break_raw <<
    instruction0_shift).low();
}
```

- The compiler in the /MODEL=ANSI object model, sometimes created the "this" pointer of the incorrect size for base classes that are templatized. This has been fixed.

- When a predefined macro is undefined using an #undef directive, the compiler now issues the NOUNDEFPREMAC diagnostic with Warning severity in all modes except /STANDARD=STRICT. Previously, this diagnostic had Error severity.

- The compiler now issues a NEVERDEF diagnostic with the severity of a discretionary error when compiling with /STANDARD=STRICT, and with Warning severity in all other /STANDARD dialects:

  ```
  struct S { inline void foo(); void bar(); }
  void S::bar() { foo(); }  // bar is not called, but foo is "used"
  ```

- The compiler now issues the BLTINCLNK diagnostic only if the routine would have been treated as an intrinsic function had it had "C" linkage. Previously, the diagnostic was emitted for a broader set of routines.

- The compiler now has a fix for a buffer overflow problem that previously was exposed when using the /XREF qualifier.

- The compiler no longer allows specifying the /PREPROCESS_ONLY and the /OBJECT qualifiers together on the command line. Previously, the compiler accepted both qualifiers but did not produce an object file.

- The compiler would previously ignore the specification of the /ASSUME=NOHEADER_TYPE command-line qualifier when using the CXX$USER_INCLUDE logical. This has been fixed.

- The compiler would sometimes produce an assertion if union objects were passed by reference. This has been fixed.

- When compiling /OPT/DEBUG, the compiler could accvio when processing some compiler-generated code for interludes. This has been fixed.

- When compiling and linking /DEBUG, sometimes a static data member was not instantiated, resulting in a undefined message from the linker. This has been fixed.

- In rare cases, a compilation that used a comma list to specify more than one source file could cause a compiler crash. This has been fixed.

- When /STANDARD=STRICT (or LATEST) was specified, if a type was a dependent type, it would not accept more than one string literal initializer in an aggregate initializer context. This has been corrected.

```
   template <typename T> void test() {
const T    *s[] = { "foo", "bar" }; // previously an error
const char *t[] = { "foo", "bar" }; }

   int main() {
const char *s[] = { "foo", "bar" };
test<char>();
   }
```

- It is now possible to `#undef` a predfined macro (such as `__LINE__`) in all modes except /STANDARD=STRICT or LATEST. A warning diagnostic is issued, but the macro will be undefined. This matches the behavior of C++ on OpenVMS Integrity.

- The V7.1 compiler would emit a BLTINCLNK diagnostic if a C++ linkage function declared a name that would normally be an intrinsic function, even if the paramters and/or return type did not match those expected by the intrinsic. With this update kit, the diagnostic will only be output if the parameter and return type matches the intrinsic.

- The V7.1 compiler would allow a command line to specify both /PREPROCESS_ONLY and /OBJECT. In this case, the compiler would silently ignore the /OBJECT qualifier, and no .OBJ file would be produced. With this release, a %DCL-W-CONFLICT diagnostic is output for this case.

- With this update kit, the compiler now supports variadic macros. This feature allows macros to take a variable number of arguments. This feature was added to the HP C Compiler in the V6.4 release and is supported by a number other C and C++ compilers. The feature is available only when the value of the /STANDARD qualifier is RELAXED (the default), MS, or GNU.

- Certain bad uses of the `va_start` macro could crash the compiler. This has been corrected.

- The PRAGIGNORE diagnostic is no longer output for the `#pragma define_template` directive when the /TEMPLATE=NOPRAGMA qualifier is specified on the command line.

- Use of certain expressions that used the array index operator on a string constant such as `(long)&"str"[1]` could crash the compiler. This has been corrected.

- The compiler now accepts the `__inline` language extension keyword in all language modes except /STANDARD=STRICT (or LATEST). The `__inline__` keyword is also accepted, except in the strict or MS modes.

- There were two very minor flaws in the 64-bit pointer support. Both involved creating 64-bit pointers using constant values. An example is:

  ```
  void *n3 = (void *) 0x80000010ULL;
  ```

  In these cases, the V7.1 compiler could sometimes generate incorrect code when the pointers were used. This has been corrected.

- In certain cases, the compiler would emit a MAYLOSEDATA diagnostic when it should not have. This could only happen when mixed pointer-size support was enabled. The compiler would sometimes create a 64-bit pointer when it should have created a 32-bit pointer. This problem has been corrected.

- The DEFAULTTMPLARG is now a warning in all modes but the strict modes. This matches the behavior of C++ on OpenVMS Integrity.

5 Release Notes for the V7.3 C++ Standard Library

The following are enhancements, changes, and restrictions for the C++ Standard Library in this version:

- When compiled in standard GNU mode, the library now defines the _RWSTD_NO_IMPLICIT_INCLUSION macro which causes library headers to `#include` their respective template definition files. This is necessary because in standard GNU mode, implicit inclusion is disabled.

Before this change, the program below would link with undefined symbol when compiled in standard GNU mode:

```
#include <vector>

int main() {
  std::vector<int> v;
  v.push_back(0);
}
```

6 Release Notes for the Version 7.1 C++ Compiler

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

6.1 Enhancements, Changes, and Problems Corrected in Version 7.1

- The /TEMPLATE_DEFINE qualifier now requires an option.
- The compiler issues a CC-W-NOTINCRTL message when it prefixes a name not in the current C RTL.

  /PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES prefixes all functions defined by the C99 standard including those that may not be supported in the current run-time library. So calling functions introduced in C99 that are not yet implemented in the OpenVMS C RTL will produce unresolved references to symbols prefixed by DECC$ when the program is linked. The compiler now issues a CC-W-NOTINCRTL message when it prefixes a name that is not in the current C RTL.
- #pragma module module-name [module-ident | "module-ident"]

  If the module-name is too long:

  - A warning is generated if /NAMES=TRUNCATED is specified.
  - There is no warning if /NAMES=SHORTEN is specified.

  A shortened external name incorporates all the characters in the original name. If two external names differ by as little as one character, their shortened external names will be different.

  If the optional module-ident or "module-ident" is too long, a warning is generated.

  The default module-name is the filename of the first source file. The default module-ident is "V1.0" They are treated as if they were specified by a #pragma module directive.

  If the module-name is longer than 31 characters and:

- /NAMES=TRUNCATE is specified, truncate the module-name to 31 characters, or less if the 31st character is within a Universal Character Name.
- /NAMES=SHORTENED is specified, shorten the module-name to 31 characters using the same special encoding as other external names.

Lowercase characters in the module-name are converted to uppercase only if /NAMES=UPPERCASE is specified.

A module-ident that is longer than 31 characters is treated as if /NAMES=(TRUNCATED,AS_IS) were applied, truncating it to 31 characters, or less if the 31st character is within a Universal Character Name.

The default module-name comes from the source file name which always appears in the listing header. The module-name (and ident) appear in the listing header only if they come from a #pragma module directive or differ from the default.

- Certain OpenVMS conditions normally result in the delivery of signals that can be processed using a C-style signal-handler mechanism. The C++ condition-handling facility has been changed to invoke this C-style signal-handling mechanism only if a signal handler has been established. Before this change, unhandled signals would result in program termination. In addition, the following C signals are now recognized: SIGCHLD, SIGPIPE, SIGWINCH, and SIGSEGV. This now makes C++ for OpenVMS Alpha functionally equivalent to C++ for I64 systems.
- Specifying a C++ headers library and a C headers library using "+" and the /LIB qualifier on the `cxx` command line, as in the following example, can cause the compiler to fetch a C header file from the C headers library instead of a template-definition file from the C++ headers library:

```
cxx x.cxx+SYS$LIBRARY:CXXL$ANSI_DEF.TLB/LIB+SYS$LIBRARY:DECC$RTLDEF.TLB/LIB
```

This can happen if a C header file has the same filename as the C++ template-definition file: for example, the `string.h` header file in the C headers library and `string.cc` template-definition file in the C++ headers library.

- The compiler was not correctly handling break statements out of loops which follow an identifier label within switch statements. This is now fixed.
- A bug in the C++ compiler in the /STANDARD=STRICT mode of compilation, used to produce an error diagnostic when compiling code that used `setjmp` or `c$$establish`. This has been fixed.

- In the /STANDARD=STRICT mode of compilation, the compiler used to issue a diagnostic with the severity of Error, for NULL reference expression within a `sizeof` expression. The severity of the diagnostic is now an informational.
- A command procedure CXX$PRODUCT_REMOVE has been added to the HP C++ V7.1 kit for OpenVMS Alpha. This procedure allows you to remove the HP C++ compiler product. It performs the equivalent of a PCSI PRODUCT REMOVE command.

  You are required to disable the product license before issuing the command procedure to prevent a compilation from interfering with the delete process.

  If the compiler has been installed as a shared image, the command procedure will uninstall the image.

  The command procedure takes no parameters and can be run as follows:

  ```
  $ @SYS$SYSTEM:CXX$PRODUCT_REMOVE
  ```

  ```
  Do you wish to proceed with removing HP C++ <No>? Yes [Ret]
  ```

  Enter "Yes" to remove the compiler from your system.
- Previously, the propagation of a C++ exception out of a thread's start routine did not result in cxxl$terminate( ) being called. A partial solution for the problem (a back-port of the solution on OpenVMS I64) is available on OpenVMS Alpha Version V8.3 and higher. It requires pthreads library patch VMS83A_PTHREAD-V0100.

  For the new behavior, you must link against the shareable image of the C++ Standard Library, available with Version 7.1 and higher of the C++ Compiler for OpenVMS Alpha (see Section 7.2), and must define the logical CXXL$LANGRTL to point to the appropriate shareable version of the C++ Standard Library. For example:

  ```
  $ define cxxl$langrtl libcxxstd
  $ cxx test
  $ cxxl test
  $ run test
  ```

  ```
  Work thread starting.
  Custom terminate function called.

  Work thread starting.
  Custom terminate function called.
  %TRACE-F-TRACEBACK, symbolic stack dump follows
    image    module    routine             line      rel PC           abs PC
  ...
  ```

```
$ type test.cxx
    // Begin test.cxx
    #include <stdlib.h>  // EXIT_SUCCESS
    #include <pthread.h>
    #include <iostream>  // cout
    #include <exception> // set_terminate
    #include <new>       // std::bad_alloc

    void terminator() {
      cout << "Custom terminate function called." << endl;
      abort();
    }

    void* work(void* param) {
      cout << "Work thread starting." << endl;
      set_terminate(terminator);
      throw std::bad_alloc();
      return 0;
    }

    int main (int argc, const char ** argv) {
      set_terminate(terminator);
      pthread_t t;
      pthread_create(&t, NULL, &work, NULL);
      pthread_detach(t);

      sleep(2);
      throw std::bad_alloc();
      return EXIT_SUCCESS;
    }
    // End test.cxx
```

- __MEMxxx builtin functions treated length as signed.

  The builtin functions __MEMMOVE, __MEMCPY, and __MEMSET were erroneously generating code to sign-extend the length parameter before passing it to an OTS$ routine that interprets the length as a signed 64-bit value. The length parameter for the standard C library routines is of type size_t, which is unsigned int on OpenVMS. But the sign extension done by these builtins made a length greater than or equal to 2147483648 be seen as negative by the underlying OTS$ routines, and those routines treat a negative length as a no-op. Although lengths so large are unusual, they are possible and need to be supported.

  It is possible that some source code exists that actually computes negative length values, and relies on these values being treated as no-ops. That behavior is not supported by the standard, although it is a convention used in similar VMS library routines. With this bug fixed, such code will most likely ACCVIO at run-time. Such code needs to be changed to test the length value explicitly to determine if it is in a range that should be ignored, and either bypass the call or use a length of zero - that is the only way to assure correct operation.

Note that the `<string.h>` header may (under __NONAMESPACE_STD) define macros to replace invocations of the standard C functions memmove, memcpy, and memset by invocations of these builtins. And these functions are also recognized as intrinsics, so even without the macros the compiler will treat them as builtin functions. Therefore, just recompiling a module will introduce the effect of this fix, even if linking against the same version of the CRTL. The only reliable way to ensure that a negative length value will be treated as a no-op instead of a large positive value when using these standard C functions is to modify the source to test the value explicitly - that's what the standard-conforming behavior requires.

Finally, note that this bug is not present in either the builtin or CRTL implementions of these functions on OpenVMS I64, so this fix makes the Alpha behavior match the I64 behavior.

- 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.

- A problem has been corrected in the implicit include processing. The implicit inclusion will no longer select files such as "`.C`" or "`.CXX`" (where these files have no file name portion).

6.2 Restrictions in Version 7.1

- Debug is not able to break on a non-local constructor in a shareable image because of the order in which images start up on OpenVMS. Specifically, the debugger is loaded and begins to run only after the initialization of all the shareable images but before the initialization of the main image. For example, in the following program where the constructor `foo::foo()` has been built into a shareable image, the constructor runs before the debugger has been activated. As a result, there is no way to break on this constructor, even if you attempt to signal the debugger.

```
$ cxx/debug/noopt static_ctor_main.cxx
$ cxx/debug/noopt static_ctor.cxx
$ cxx_link/share/debug static_ctor/opt
$ cxx_link/debug static_ctor_main/opt
$ define STATIC_CTOR DISK$:[DIR]STATIC_CTOR.exe
$ run  static_ctor_main
foo::foo() <----- ctor has already been executed

         OpenVMS Debug Version VX.X-xxx
%DEBUG-I-INITIAL, Language: C++, Module: STATIC_CTOR_MAIN
%DEBUG-I-NOTATMAIN, Type GO to reach MAIN program

--------- static_ctor_main.cxx -----------
main()
{
}
---------- static_ctor.cxx -------------
extern "C" int printf(const char *,...);

#define SS$_DEBUG 1132
extern "C" int lib$signal(unsigned int cond);

class foo {
public:
  foo() {
    printf("foo::foo()\n");
#if SIG
    lib$signal(SS$_DEBUG);
#endif
  }
  ~foo() { }
};
foo x;
```

```
-------- static_ctor_main.opt----------
NAME = main
IDENTIFICATION = "V1.001"
GSMATCH = LEQUAL, 1, 001
static_ctor_main.obj
static_ctor/share
-------- static_ctor.opt ------------
NAME = static_ctor
IDENTIFICATION = "V1.001"
GSMATCH = LEQUAL, 1, 001
static_ctor.obj
```

- The C++ compiler incorrectly mangles top-level cv-qualifiers into function signatures in the default object model (/MODEL=ARM on OpenVMS, `-model arm` on Tru64 UNIX). For example:

```
file.cxx
-----
void f(const int);
void f(int) {}

> cxx/noobj file.cxx
void f(int) {}
.....^
%CXX-W-NOTQUACOMPREDEC, declaration is not qualifier compatible with
          "void f(const int)" (declared at line 1)
at line number 2 in file DEVICE$:[DISK]FILE.CXX;15

> cxx/noobj/model=ansi file.cxx
>
```

- The function _RWdestroy(ForwardIterator first, ForwardIterator last) has been modified to destroy the objects pointed to by iterators in the range [first, last]. Before the fix, the function was just looping through iterators without destroying the objects.

  The functions std::uninitialized_copy( ), std::uninitialized_fill( ) and std::uninitialized_fill_n( ) have been modified to increment iterator(s) after the call to the _RWconstruct( ) function. This is to make sure that if an exception is thrown during constructing the object, only objects that were fully constructed by the std::uninitialized_*( ) function are destroyed.

7 Release Notes for the Version 7.1 C++ Standard Library

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

7.1 Enhancements, Changes, and Problems Corrected in Version 7.1

- While applications using the C++ library iostreams can be compiled with the _LARGEFILE macro defined, the C++ library iostreams do not support seeking to 64-bit file offsets. For more information on _LARGEFILE macro see the HP C Run-Time Library Reference Manual for OpenVMS Systems.
- 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 same text library as the C++ Standard Library headers and template definition files: SYS$LIBRARY:CXXL$ANSI_DEF.TLB.

  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. Otherwise, the default is /PURE_CNAME when compiling with /STANDARD=STRICT_ANSI and /NOPURE_CNAME.

  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 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. Defining the `__CNAME_OVERLOADS` macro in NOPURE_CNAME mode in combination with other macros and options (for example, /STANDARD=MS, /DEFINE=_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 behavior in existing code. Consider these examples:

- o 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-E-AMBOVLFUN, more than one instance of overloaded function "std::sin"
matches the argument list:
    function "std::sin(double) C"
    function "std::sin(float)"
    function "std::sin(long double)"
    argument types are: (int)
```

- o 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 the 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;
}
```

- 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 synchronize access to the reference count in the reference counting implementation of 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
```

- 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.

- 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.

- 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
```

- The default constructor of class template std::istream_iterator<class T, class charT, class traits, class Distance> used to initialize stream data members with the address of std::cin regardless of the charT template argument. As a result, this class template could not be instantiated on charT template arguments other than 'char'. For example, compilation of the following program would result in a BADINITTYP error. This has been corrected.

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <iterator>
#include <istream>

std::istream_iterator<int, wchar_t> x;
```

- The `min()` and `max()` functions in the <long long> specialization of the std::numeric_limits class template used to return the wrong values. Specifically, they returned the same (32-bit) values that the `min()` and `max()` functions in the <long> specialization are returning. This has been corrected.

  Consider the following sample program `x.cxx`:

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

#include <limits>
#include <iostream>

using namespace std;

main() {
  cout << "max() = " << numeric_limits<long long>::max() << endl;
  cout << "min() = " << numeric_limits<long long>::min() << endl;
  cout << "max() = " << numeric_limits<unsigned long long>::max() << endl;
}
```

  Without the fix, `x.cxx` gives:

```
max() = 2147483647
min() = -2147483648
max() = 4294967295
```

  After the fix, it gives:

```
max() = 9223372036854775807
min() = -9223372036854775808
max() = 18446744073709551615
```

- The C++ Standard Library header <vector> was modified to expose std::vector<bool> overloads of relational operators only when compiling with the __DECFIXCXXL1941 macro defined. These overloads make it impossible to use relational operators on vector<bool>::iterator types; see the code example below. That the current C++ standard lists these overloads (section 23.2.5 - Class vector<bool> [lib.vector.bool]) is considered to be a defect in the standard. Some other implementations of STL do not provide these overloads.

  With std::vector<bool> overloads of all the relational operators removed, the following program compiles. Before the change, it would not compile.

```
#include <iterator>
#include <vector>

class D : public std::reverse_iterator<std::vector<bool>::iterator> {
};

int main(void)
{
  D x, y;
  if ( std::operator== <std::vector<bool>::iterator>(x,y) )
    return 0;
  if ( std::operator!= <std::vector<bool>::iterator>(x,y) )
    return 0;
  if ( std::operator< <std::vector<bool>::iterator>(x,y) )
    return 0;
  if ( std::operator<= <std::vector<bool>::iterator>(x,y) )
    return 0;
  if ( std::operator> <std::vector<bool>::iterator>(x,y) )
    return 0;
  if ( std::operator>= <std::vector<bool>::iterator>(x,y) )
    return 0;
  return 1;
}
```

7.2 The C++ Standard Library in the Form of a Shareable Image

Starting with C++ Version 7.1, the compiler kit provides linker options files and a CXXL$BUILD_SHARED_LIBCXXSTD_IMAGES.COM DCL procedure for building the following shareable images:

- LIBCXXSTD.EXE - model ARM preinstantiation library
- LIBCXXSTD_NOINST.EXE - model ARM noinstantiation library
- LIBCXXSTD_MA.EXE - model ANSI preinstantiation library
- LIBCXXSTD_MA_NOINST.EXE - model ANSI noinstantiation library

Notice that the filenames of the shareable images are the same as the filenames of their OLB counterparts in SYS$LIBRARY.

7.2.1 How To Create Shareable Images

The compiler installation procedure places CXXL$BUILD_SHARED_LIBCXXSTD_IMAGES.COM into the SYS$SYSTEM directory and places the linker options files into SYS$LIBRARY. The filenames of the options files are the same as the filenames of the images they create, but with a CXXL$ prefix. For example, the options file for the model ARM preinstantiation library image LIBCXXSTD.EXE is CXXL$LIBCXXSTD.OPT. The compiler installation procedure does not invoke CXXL$BUILD_SHARED_LIBCXXSTD_IMAGES.COM.

To build shareable images, invoke CXXL$BUILD_SHARED_LIBCXXSTD_IMAGES.COM as follows:

@SYS$SYSTEM:CXXL$BUILD_SHARED_LIBCXXSTD_IMAGES.COM

The procedure creates four shareable images from their respective OLB counterparts. It accesses the OLB libraries using the SYS$LIBRARY logical name and places the images into the same directory on the SYS$LIBRARY search list where the OLB libraries are located. The procedure does not insert images into IMAGELIB.OLB.

7.2.2 Linking Against the C++ Standard Library Shareable Image

There are two ways of using the C++ Standard Library in the form of a shareable image: with and without inserting the library image into SYS$LIBRARY:IMAGELIB.OLB.

Note that the following procedures are the same for object models ANSI and ARM, and for linking against the preinstantiation and noinstantiation libraries. Only name of the C++ Standard Library image is different.

When linking against the OLB library, the CXXLINK utility automatically chooses the library based on the /MODEL and /NOPREINST qualifiers, so that the selection of OLB library is transparent. When linking against the shared C++ Standard Library, you must specify the correct Standard Library image yourself.

The following examples use the model ARM preinstantiation library LIBCXXSTD.EXE. This is the default library; it is used when a program is compiled without any /MODEL qualifier and without the __FORCE_INSTANTATIONS macro defined.

7.2.2.1 Linking Without the C++ Standard Library Image in IMAGELIB.OLB

The following example shows how to link without the C++ Standard Library image in IMAGELIB.OLB:

```
cxx foo.cxx
cxxl foo.obj, sys$input:/opt
LIBCXXSTD.EXE/share
^Z
define LIBCXXSTD disk:[directory]LIBCXXSTD.EXE;
run foo
```

7.2.2.2 Linking With the C++ Standard Library Image in IMAGELIB.OLB

This is probably the most convenient way of linking against shared C++ Standard Library. However, there are several restrictions:

- Model ARM and model ANSI images cannot both be inserted into IMAGELIB.OLB because there are some symbols with the same names in the two libraries.
- Preinstantiation and noinstantiation images cannot both be inserted into IMAGELIB.OLB. This is because the noinstantiation library is a subset of the preinstantiation library. Therefore, inserting a preinstantiation library image into IMAGELIB.OLB will break linking /NOPREINST because library instantiation symbols will be resolved from IMAGELIB.OLB instead of from the user instantiations in the repository.

So inserting a C++ Standard Library image into IMAGELIB.OLB is suitable for a site where only a single object model is used and where linking is consistently done against either preinstantiation or noinstantiation library.

In order to link against the C++ library shareable image in IMAGELIB.OLB, create an empty LIBCXXSTD*.OLB library and make CXXLINK use it instead of the OLB library in SYS$LIBRARY. You can do this by defining a logical name CXX$LINK_LIBCXXSTD_DIR pointing to the location where the empty OLB library resides. For example, for the model ARM preinstantiation library:

```
libr/create disk:[directory]LIBCXXSTD.OLB
define CXX$LINK_LIBCXXSTD_DIR disk:[directory]
```

Note that while the name of the logical remains the same, the name of the OLB library it points to depends on the object model and the preinstantiation/noinstantiation library. For example, for model ANSI it would be LIBCXXSTD_MA.OLB; for the model ARM noinstantiation library, it would be LIBCXXSTD_NOINST.OLB, and so on.

With the C++ library shareable image in IMAGELIB.OLB, you can link against it without having to explicitly specify it on the CXXLINK command. For example:

```
cxx foo.cxx
cxxl foo.obj
run foo
```

7.2.3 Restrictions

7.2.3.1 Overriding Global Operators new and delete As the C++ User's Guide indicates, programs overriding global new and delete must be linked /NOSYSSHARE. Such programs will have to continue linking /NOSYSSHARE against the OLB libraries.

7.2.3.2 Mixing OLB and Shared C++ Library in the Same Process Mixing dynamically loaded libraries linked against OLB and the shared C++ Standard Library in the same process is not supported. Also, a main executable must be linked against the same flavor (either .OLB or .EXE) of the C++ Standard Library against which the libraries it dynamically loads are linked. Violating this restriction can result in unpredictable behavior.

7.3 Restrictions in Version 7.1

This section describes problems you might encounter when using the current release of the C++ Standard Library with the HP C++ compiler. Where appropriate, workarounds are suggested.

7.3.1 Using the C++ Libraries in Microsoft Standard Mode

When compiled /STANDARD=MS, the following restrictions apply:

- `typeid(__int64).name()` returns `__int64` instead of `long long`.
- Header `<new>` does not declare operator `delete[](void*, void*)`.
- Header `<new>` does not declare type `new_handler`.
- `mem_fun` function objects from `<functional>` cannot be used with a void first template argument. For example, the following program does not compile in Microsoft mode:

```
template <class S, class T>
struct mem_fun_t
{
  S (T::*pmf)();

S operator()(T* p) { return (p->*pmf)(); }
};

struct X {};
void (mem_fun_t<void, X>::*pf) (X*) = &(mem_fun_t<void, X>::operator());
```

- It is impossible to create a list container of elements of type `size_t` using the list(size_type) constructor. For example, the following program does not compile in Microsoft mode:

```
#include <stddef.h>
#include <list>
std::list<size_t> x(1);
```

- Using `ostream<<` and `istream>>` operators with variables of type `[unsigned] __int64` results in undefined linker symbols.
- Using `vector<bool>` iterators results in compilation errors. For example, the following program does not compile in Microsoft mode:

```
#include <vector>

void f()
{
  std::vector<bool> x;
  x.insert(x.begin(), true);
}
```

7.3.2 Other Restrictions

Other Standard Library restrictions in this release are:

- The std::use_facet<FACET> function template can be instantiated only on facets having a default constructor.

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

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

8.1 Enhancements, Changes, and Problems Corrected in 6.6-???

- The compiler was not correctly handling certain unusual loops within a switch statement. (11108)
- 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 compilation unit referencing `vector<bool>`, 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 the following:

```
#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 `#include`s the `<vector>` header. (L1760)

- Changes to `<time.h>` and `<time.cc>`

  This kit supplies updated versions of DECC$RTLDEF.TLB (C library files) and CXXL$ANSI_DEF.TLB (C++ standard library header files) that address various issues in an upward-compatible fashion. However, there are changes to the conditionalization of the layout and member names for the type "struct tm" (from `<time.h>`), as well as conditionalization of the implementation of the gmtime( ) and localtime( ) functions [and their reentrant variants gemtime_r( ) and localtime_r( )] to be used, that might affect some programs. As a general precaution to minimize possible problems, it is recommended that if any source module that #includes either `<time.h>` or `<time>` is recompiled, all modules in the application that `#include` either of those headers should also be recompiled. Note that the C++ header `<time>` #includes `<time.cc>` (when /NOIMPLICIT_INCLUDE is in effect). The C++ header `<time>` in itself is not affected.

  The purpose of the changes is to address an interrelated set of functionality, user-namespace, and binary compatibility issues in a way that will be consistent between C and C++ going forward with new versions of the compilers and run-time libraries.

  - Functionality:

    The C++ standard library's "time_put" time formatting facet supports timezone formatting from a value of type "struct tm". Therefore, C++ compilations need to ensure that the implementations of localtime( ) and gmtime( ) selected [and their reentrant _r variants] are the ones that support timezone information. One specific functionality bug addressed by the changes is that some versions of the header provided two additional members for timezone information in the declaration of struct tm for use by C++, but did not cause the timezone-aware implementations of the time functions to be selected under those same conditions. This could lead to use of uninitialized data, or an access violation when the user code then tried to access the additional members of struct tm.

- User-namespace:

  The members of struct tm that support timezone information are not part of the C standard. They are actually a common extension from BSD Unix, with conventional names of tm_gmtoff and tm_zone. Although the C standard allows an implementation to define additional members in struct tm, it does not reserve the prefix "tm_" for that purpose. So a strictly-conforming C program could #define tm_zone as a macro expanding into something that would cause a syntax error when encountering tm_zone as the name of a member in a struct type. Or conversely, a program compiled with the strict C namespace in effect could refer to the members by their BSD names, without getting a diagnostic, and infer that the names were defined under the C standard.

  Previous versions of `<time.h>` handled this issue by suppressing the declaration of these two members entirely under the strict C namespace (_ANSI_C_SOURCE). This new version (when used with C++ versions *newer* than V6.5-042) defines struct tm such that the timezone members have names in the namespace reserved to the implementation when the strict C namespace is in effect: instead of tm_gmtoff and tm_zone it uses __tm_gmtoff and __tm_zone (which is the convention on Tru64 and Linux systems). Because the `<time.cc>` header contains source code that refers to the "tm_zone" member of struct tm, that code had to be conditionalized to use "__tm_zone" when the strict C namespace is in effect (and continue to use the BSD names when it is not in effect).

- Binary compatibility:

  Previous versions of `<time.h>` that suppressed the two non-standard members in struct tm under the strict C namespace resulted in potential binary incompatibility between modules compiled with/without the strict C namespace. The practice in the new `<time.h>` of just changing the names of the members to meet the requirements of the strict namespace makes modules compiled with either setting binary compatible, but it introduces potential for binary incompatibility with pre-existing object modules that cannot be recompiled from source.

Note that modules compiled with different sizes for struct tm only have potential for encountering a problem; in typical usage no problem will occur because struct tm is not often embedded within a struct or used as an element of an array, with that struct or array then shared beteen compilation units. Typical code using struct tm only accesses it locally through the pointer returned by the library functions, or contructs a local instance that is then passed by address to library functions. Those kinds of uses are not affected by a size difference that occurs between different compilations.

But to help deal with possible situations where a real binary compatibility problem is encountered, and it is not feasible to recompile all of the modules involved, the struct declaration in `<time.h>` can be forced to the short version (without the timezone members) by defining the macro "_TM_SHORT" before the header is included. Similarly, the declaration can be forced to the long version by defining "_TM_LONG". These macros will override any other conditionalization based on the version of the CRTL and the compiler, and produce a struct tm declaration that is of the specified size.

As described above, the use of implementation-reserved identifiers to make the size of struct tm uniform regardless of whether or not the strict C namespace is in effect is conditionalized to the new C++ compiler. These headers are shared between C and C++ compilers, and that change will not affect existing versions of the C compiler. Unlike C++, the C compiler enables the strict C namespace under its /stand=ansi89 or /stand=c99 qualifiers as well as when any of the _XOPEN, _POSIX, or _ANSI_C standard-conformance macros are specified (see section 1.5 Feature-Test Macros for Header-File Control in the C Run-Time Library Reference Manual). It was felt that the risk of binary compatibility problems would be slightly greater for C than for C++ because these options are more commonly used with C compilations than they are with C++. However, the _TM_LONG (and _TM_SHORT) macros will affect both compilers. If your application contains C modules that #include `<time.h>` (as well as C++ modules that do), it is advisable to recompile those C modules with /define=_TM_LONG to avoid possible binary incompatibility between the C and C++ modules.

- A problem with catching a VMS exception when compiled /names=as_is and/or with the __NEW_STARLET macro defined, has been fixed. In this compiler release, however, the fix is available for object model ARM only. The fix for object model ANSI is planned to be released in the next major compiler release for OpenVMS Alpha.

For example, the following program now outputs "caught signal". Before the fix, if compiled /names=as_is or with the __NEW_STARLET macro defined, it would output "missed VMS signal". (10655)

```
#include <chfdef.h>
#include <lib$routines.h>
#if __DECCXX_VER <= 60590042
#define cxxl$set_condition CXXL$SET_CONDITION
#endif
#include <cxx_exception.h>
#include <stdio.h>

main() {
  try {
      cxxl$set_condition(cxx_exception);
      lib$signal(99628);
  }
  catch (chf$signal_array&) {
      puts("caught signal");
  }
  catch (...) {
      puts("missed VMS signal");
  }
}
```

- Exception handler now preallocates some memory so exceptions can be thrown even if no memory is available. (7881)
- The compiler was incorrectly computing the element size as zero when the array was a typedef-ed type. This caused problems, for instance, when whole array destruction was invloved. (10865)
- When compiled with full IEEE support (-ieee option on Tru64 and /FLOAT=IEEE/IEEE=DENORM qualifiers on AlphaVMS), denorm_min( ) function of the specialization of class template std::numeric_limits for long double would return garbage. It has been corrected. For example, the program below now returns the correct value: 6.47518e-4966. (L1880)

8.2 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)

8.3 Enhancements, Changes, and Problems Corrected in V6.5-041

- 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)

8.4 Enhancements, Changes, and Problems Corrected in V6.5-040

- 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)

8.5 Enhancements, Changes, and Problems Corrected in V6.5-039

- Eliminated incorrect unreachable diagnostic introduced in the V6.5-034 compiler. (10307)
- Corrected template argument deduction bug. (10352)
- Corrected bad debug information generated for importing typedef into namespace. (10280)

8.6 Enhancements, Changes, and Problems Corrected in V6.5-038

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

8.7 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)

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

- Implemented "/first_include" 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)

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

- Implemented "/main=posix_exit" which causes main to call __posix_exit instead of exit when returning. (9724)
- Eliminate compiler assertion for catch blocks with control flow that will not reach the end of the block. (9946)
- Eliminate compiler assertion for code with embedded newlines. (9962)

8.10 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)

8.11 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)
- Compiler crash in exception support has been eliminated. (9756)

8.12 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)
- Implemented prototype for `pragma include_directory`. (9677)
- 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)

8.13 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)

8.14 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)
- Compilation errors when compiling `builtins.h` in `ms` mode have been eliminated. (9588)
- Problem with `dynamic_cast` using class defined in shared images has been corrected. (9603)
- Error message for `utimes` function prototype when including `time.h` has been eliminated. (9609)

8.15 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)

8.16 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)

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

- Release notes updated to clarify version compatibility.

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

- Compiler crash in optimizer has been eliminated. (9532)
- Missing initialization of static variables corrected. (9530)
- 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)
- 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]

8.19 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]

8.20 Enhancements, Changes, and Problems Corrected in Version 6.5

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

8.21 Restrictions and Known Problems in Version 6.5

- If you compile and link with /MODEL=ANSI and then try to use the VMS debugger to access members of a virtual base class, you might see this error:

  ```
  %DEBUG-E-INTERR, debugger error in DBG$GET_BASE_CLASS_OFFSET: can't find __bptr
  or session corruption
  ```

  The error is displayed because the debugger has not yet been updated to understand the new internals of the ANSI object model.

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

  For example, the following program produces compilation error:

  ```
  #include <fstream.h>

  int foo()
  {
    fstream  fs;
    if (fs == NULL ) return 0;
    return 1;
  }

  $ cxx/noobj x.cxx

    if (fs == NULL ) return 0;
  .........^
  %CXX-E-AMBIGUOUSOPRFUN, more than one operator "==" matches these operands:
              built-in operator "pointer == pointer"
              built-in operator "pointer == pointer"
              operand types are: fstream == int
  ```

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 following:

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

it compiles cleanly. [L1710]

- 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]

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

9.1 Enhancements, Changes, and Problems Corrected in Version 6.3

- 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 undocumented operator char*(), which converts a String to a pointer to char in the non-standard String class has been conditionalized with the macro __DEC_STRING_COMPATIBILITY so that it is available only when the program is compiled with the macro defined.

  This operator has been conditionalized beccause it allows passing a String object to functions like istream::get() or istream::getline() that expect a pointer to char.

The `String::operator char*()` returns a pointer to the “data component” of the `String`, which is a private data member inside the `String` class. When a `String` object is passed to a function expecting a pointer to `char`, the function cannot determine how much memory has been allocated by the `String`, and the `String` cannot determine the status of the exposed pointer and the memory to which it points.

This design flaw in the non-standard `String` class has been fixed in a way that leaves the legacy functionality available as an option. The standard ANSI C++ `String` class, for example, lets users retrieve only a `const char*`, not a `char*`, to the pointer owned by the `String`. [10.1563]

- In response to user requests, CXXLINK by default now deletes the option file SYS$DISK[]CXX_REPOSITORY.OPT that previously remained in the user's current directory on completion of CXXLINK.

  To support this change, CXXLINK creates a unique prefix for the repository .OPT filename in the following format:

  ```
  <nodename><PID>.OPT
  ```

  For example, if the node name is MYNODE and the PID is 020204AEA, the file name would be MYNODE020204AEA.OPT. This option file is created in the [.CXX_REPOSITORY] directory, if one exists. If the [.CXX_REPOSITORY] directory does not exist, the .OPT file is created in the user's current directory. CXXLINK passes this file to LINK as the second file specified on the LINK command. To preserve LINK's directory search order, the directory specification of the first file is prefixed to the name of the second file, if a second file is specified. Depending on the CXXLINK option selected, CXXLINK deletes this .OPT file by default, or renames it to CXX_REPOSITORY.OPT or to a user-specified file name. For detailed information on CXXLINK options, type HELP CXXLINK or point your browser to CXXLINK. [7135]

- In Version 6.2, signaling of VMS exceptions or a C signal might cause a program to terminate with `%CXXL-F-TERMINATE` status regardless of which VMS exception or C signal has been raised and regardless of whether a VMS exception handler or C signal handler has been established.

  For example, the following `test1.cxx` program calling `lib$stop(SS$_MCNOTVALID)` terminates with `%CXXL-F-TERMINATE`:

```
#include <lib$routines.h>
#include <ssdef.h>
main() { lib$stop(SS$_MCNOTVALID); }
```

```
$ run test1
%CXXL-F-TERMINATE, terminate() or unexpected() called
%TRACE-F-TRACEBACK, symbolic stack dump follows
  image    module    routine             line      rel PC           abs PC
...
```

Similarly, the following `test2.cxx` program, establishing a C signal handler to catch a `SIGILL` signal, also terminates with `%CXXL-F-TERMINATE` without a signal handler being invoked:

```
#include <signal.h>
#include <stdio.h>
static void signal_handler(int x)
{
  puts("signal arrived");
}
void main()
{
   signal(SIGILL, signal_handler);
   raise(SIGILL);
}
```

Version 6.3 corrects this behavior: the C++ run-time environment no longer issues a call to `terminate()` if a VMS exception (possibly, generated by a `raise()` or `gsignal()` function arrives. Instead, the exceptions is resignaled to the next exception handler, and the exception is handled properly.

For example, the `test1.cxx` program now correctly terminates with the OpenVMS condition value specified in the call to `lib$stop()` as follows:

```
$ run test1
%SYSTEM-F-MCNOTVALID, device microcode is not valid
%TRACE-F-TRACEBACK, symbolic stack dump follows
  image    module    routine             line      rel PC           abs PC
...
```

And `test2.cxx` now catches the signal and terminates correctly as follows:

```
$ run test2
signal arrived
$
```

[7550, 8234]

- The compiler can no longer generate machine code listing when requested to not produce object files. [8448]

- CXXLINK no longer incorrectly concatenates arguments passed to it if such arguments are defined as logical names. [8467]
- CXXLINK now correctly handles filenames when using extended ODS-5 file parsing. [8626]
- In the ANSI object model (/model=ansi), string literals are treated as pointers to const char (const char *). In the ARM object model (/model=arm, the default), string literals continue to be treated as pointers to char (char *). [8659]
- To generate the correct code to make calls to virtual functions that return a structure, class, or union, the compiler needs the full definition of the type returned, which is missing in the following code fragment:

```
struct Cities;

struct Storage {
       int x;
       virtual Cities visit() = 0;
       };

struct Reader : virtual Storage {
       Cities visit();
       };

       struct T2 : Reader {
       };

extern void ggg(Storage *);

       void fff(void) {
       Storage *s = new T2();
       s-x = 23;

       ggg(s);
       }
```

  Because the complete definition for the type Cities is missing, the compiler cannot correctly generate the code that allows `ggg()` to call `s-visit()` for all possible definitions of the type Cities. Previous compiler versions generated code that worked for some definitions of Cities but not for others.

  The current version of generates a diagnostic when it encounters a situation:

```
    virtual Cities visit() = 0;
...................^
%CXX-E-INCTYPPREVIRTAB, The incomplete type Cities precludes correct
          generation of the virtual table for the type Reader.
          Supply the complete type definition for Cities or use
         /MODEL=ANSI.
         at line number ... in file ...
```

If you provide the definition of the type Cities in the module that generates this diagnostic, the compiler can generate the correct code. This code generation problem has been corrected in the ANSI object model. It could not be corrected in the ARM object model because an model incompatibility would have resulted. [8668]

- In `basic_istream` and `basic_ostream`, sentry constructor has been modified to not create the guard object if the stream is `tie()`'ed to a stream using the same streambuf object. This eliminates the mutex deadlock which can occur in a multithreaded application. [10.1641]

  With this modification, the following program no longer hangs when compiled with `/define=DEADLOCK` and linked against the PTHREAD library.

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <iostream>

int main() {
  ostream x(std::cout.rdbuf());
  ostream y(std::cout.rdbuf());
#ifdef DEADLOCK
  y.tie(&x);
#endif
  y << "Hello, world" << std::endl;
  return 0;
}
```

- 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 C* in *HP C++ User's Guide for OpenVMS Systems*). [10.1138]

- Undefined symbols from the Standard Library no longer occur when using the `/names=as_is` qualifier. [10.1158]

- On OpenVMS Alpha Version 7.0 and above, the Standard Library headers now include `pthread.h` to implement multithread safety. Currently this has the side effect of polluting the global namespace with non-reserved macro and `typedef` names such as EXCEPTION, THROW, CATCH, and so on from `<pthread_exception.h>`. This problem has been acknowledged as a defect and is being investigated In the meantime, if you are using any of these names you must change them or devise a workaround.

For example, if you have:

```
#include <vector>
#define CATCH 42 // trying to define your own macro, but get redefinition error
```

you would need to make the following change:

```
#include <pthread.h>
#undef CATCH // undef the pthread macro
#include <vector>
#define CATCH 42 // define your own macro
```

[10.1205]

- The 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]

- The `basic_string::find_first_not_of(charT, size_type)` function now works correctly if the string contained embedded nulls. [10.1316]
- In Version 6.2, `basic_ostream::flush()` did not always flush the buffer if the type of the stream was an `fstream`. The buffer is now always flushed. [10.1348]
- 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]
- 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]
- The `iostreams` and `locales` are now multi-thread safe. [10.1429]
- 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]
- 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 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]

- The standard library headers no longer include pthread.h or tis.h. They can therefore be used in conjunction with the macro _PTHREAD_USE_D4 (DECthreads POSIX 1003.4a/Draft 4 (or d4) interface). [10.1540]
- The following functions have been added to basic_filebuf class:

```
bool __sync_with_device() const;
  // Returns the value of _RWsync_with_device flag.

bool __sync_with_device(bool state);
  // Sets the value of _RWsync_with_device flag and returns
  // the previous value of the flag.
```

  These functions are not described in the C++ standard, and are provided as an extension in order to control the behaviour of basic_ostream::flush() method and std::endl and std::flush manipulators. The functions are available only on OpenVMS systems.

  If _RWsync_with_device flag is true, and if the type of the stream is an fstream, the basic_ostream::flush() function transfers all buffered data to the device as if the C Run-Time Library fsync() function were called.

  If _RWsync_with_device flag is false, it is not guaranteed that basic_ostream::flush() function or std::endl or std::flush manipulators will flush the data all the way to the device.

  To maintain compatibility with the behaviour of the Standard Library shipped with the HP C++ Version 6.2 compiler, the default value of _RWsync_with_device flag is false. The following example shows how the new functions are used.

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <fstream>
#include <stat.h>

int main(void)
{
    const char * filename = "file.dat";
    std::ofstream testfile(filename);
    struct stat buf;

    // display file synchronization state

    cout << "1? file synchronization state is: " <<
            testfile.rdbuf()->__sync_with_device() << std::endl;

    // write a record to the file

    testfile << "x" << std::endl;
```

```
    // check file size. Expected zero because by default endl
    // manipulator does not force data to be transferred to
    // the device.

    stat(filename, &buf);
    cout << "file size is: " << buf.st_size << std::endl;

    // set _RWsync_with_device flag to TRUE and display
    // previous file synchronization state

    cout << "file synchronization state is: " <<
            testfile.rdbuf()->__sync_with_device(1) << std::endl;

    // write another record to the file

    testfile << "y" << std::endl;

    // check file size. Expected four because now std::endl
    // will cause data to be transferred to the device.

    stat(filename, &buf);
    cout << "file size is: " << buf.st_size << std::endl;

    return 0;
}
```

[10.1574]

9.2 Restrictions and Known Problems in Version 6.3

If you compile and link with /MODEL=ANSI and then try to use the VMS debugger to access members of a virtual base class, you might see this error:

```
%DEBUG-E-INTERR, debugger error in DBG$GET_BASE_CLASS_OFFSET: can't find __bptr
or session corruption
```

The error is displayed because the debugger has not yet been updated to understand the new internals of the ANSI object model.

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

10.1 Enhancements, Changes, and Problems Corrected in Version 6.2A

Enhancements, changes, and problems corrected are as follows:

- The compiler now generates code that reports a static data member as an undefined symbol at link time if the data member is referenced but not defined. [CXXC_BUGS 6826]
- The new cxxlink design implemented in Version 6.2 to use .olbs instead of objects is not a general solution. The default behavior of cxxlink was changed back to using objects (`/prelink=use_object_files`) and the switch `/prelink=use_olb` was added to obtain the new mechanism. The new mechanism was simplified for better performance. No attempt is made

to try alternative link methods if the link fails; the default link must be used. [6804, 6795, 6583, 5997]

- In the C++ Version 6.1 compiler, some objects might have their exception unwinding information set to a negative index in the cleanup table. This condition generates the following message:

  ```
  Unexpected dtor_block_tag in do_dtor_cleanup
  ```

  [7091]

- 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 funrtions 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. [10.1298]

- Fix for `basic_string::resize()`

  A problem in the`basic_string::resize()` member function in Version 6.2 has been corrected. The incorrect behaveior 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 you changed the underlying string for one string, the value for the other strings value 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]

10.2 Enhancements and Changes in Version 6.2

This release solves several problems in earlier versions of the compiler and includes an updated OpenVMS debugger (dated 4-Mar-1999) that solves problems with the previous debugger.

Enhancements and changes are as follows:

- Support for explicit template function arguments.
- Improved EV6 support.
- CXXLINK creates an object library.

  CXXLINK now creates an object library (CXX$LINK.OLB) in the writeable repository directory. The object library is populated with all object files found in the repository.

  CXXLINK also checks any other repository directory listed on the command line (by use of `/REPOSITORY=` *`qualifier`*) for CXX$LINK.OLB object libraries.

  Each object library is then used as input to the OpenVMS Linker through LNK$LIBRARY logical names.

  In most cases, the OpenVMS Linker can resolve any missing instantions by searching the object libraries. CXXLINK can then proceed directly to the final link, without having to parse the linker output looking for unresolved symbols and their matching object files.

  Note the following restrictions:

  – CXXLINK makes use of the OpenVMS Linker Utility's LNK$LIBRARY logical names to reference specific object libraries as input to the linker. If the CXXLINK command includes any form of the `/USERLIBRARY`, an informational message appears, and CXXLINK lists any required object libraries in a linker options file.

  – CXXLINK always creates and repopulates the object library. For large repositories, this mechanism is slower than processing the linker output and creating the linker options file listing each object file.

  – CXXLINK does not create the CXX$LINK.OLB file in nonwritable repository directories. Currently, this must be done outside of CXXLINK by entering the command:

  ```
  $ LIBRARY/CREATE non-writeable-repository-dirCXX$LINK.OLB -
     non-writeable-repository-spec*.OBJ
  ```

- Programs that use `extern_model` can encounter problems, because object files from an object library do not produce the same `extern_model` code that is generated when the object file is used directly.
- Modules that use `extern_model` must be listed separately as an input to the linker.
- A compiler crash caused by a label statement in a switch statement has been corrected. [6614]
- If a user calls `VAXC$ESTABLISH` or `LIBC$ESTABLISH` in a function with a try/catch, the establish routine now works correctly. Note that C++ exception handling is disabled in these routines, as described in *C++ Exceptions, lib$establish and vaxc$establish* in *HP C++ User's Guide for OpenVMS Systems*.
- 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 new *OpenVMS* `builtins.h` file. See *Built-In Functions* in *HP C++ User's Guide for OpenVMS Systems*.
- SCA Version 4.6-4 and later can import C++ cross reference data into its analysis data file. To import the data, follow these steps:
  1. Enter the command `CXX/XREF` *`filename`*`.cxx` to have compiler emit the cross reference data file *`filename`*`.xref`.
  2. Enter the command `SCA/IMPORT` *`filename`*`.xref` to generate *`filename`*`.ana`, the analysis data file.
- Because of a bug introduced in Version 6.0, the compiler emitted a `union` layout incompatible with earilier 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 V6.2. If you have such a bitfield in your code, you must recompile. [6567]

- A problem with `/extern_model=strict_refdef` using template static data members and inline functions has been fixed. Previously, the compiler reported multiply defined external errors for symbols starting with `__SDG` and `__LSG`. When creating shared images, these symbols must be put in the symbol vector of the shared image. For more information, see *Sample Code for Creating OpenVMS Shareable Images* in *HP C++ User's Guide for OpenVMS Systems*. [6506]
- A template class name can now be reused as a non-template class in a private namespace without generating an error message. [6541]
- Mangled names beginning with `CXXL$` and `DECC$` are now included in the demangler database and can be demangled. [6458]
- More cases of unreachable code are now diagnosed. If you receive warnings you believe to be inappropriate, please report them. [6534]
- The compiler no longer can exhaust virtual memory when computing the addresses of deeply nested virtual base classes. [6473]
- `va_start` now handles `parmN` arguments of types that are promoted. [6471]
- 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 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`
- Bad code is no longer generated for an array reference within a template instantiation. [6394]
- Runtime type identification has been corrected for shared images. The RTTI calls `dynamic_cast` and `typeid` now work for objects created inside *OpenVMS* shared images. With this correction, the name of virtual function tables have been changed to `__vtbl2__<class>`, to prevent the new virtual function tables that point to RTTI information from conflicting with old tables. [6348]
- 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 could occur only if the store using the pointer addition expression appeared within an if statement, as in the following:

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

  The problem has been corrected. [6421]

- The compiler no longer generate incorrect code for offsets to external arrays. [6386]
- The routine for new[ ] now calls delete[ ] if an exception occurs during construction. [6243]
- Many improvements have been made to cross-reference information generated by the compiler and read by the SCA tool.
- The compiler now replaces embedded line feeds with blank spaces during the initial scan.

- Cleaner header file inclusion policy

  The new version of the Standard Library is much more effecient than previous versions in its header inclusion policy. 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);
  ```

- 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. A correction to the Standard Library helps programmers users of just the basic_string component to obtain smaller executables. In the previous version, using just a string in an application caused the iostream and locale object files from the Standard Library to be included in the executable. The new version corrects that problem.

- `money_get`/`money_put` `locale` facets now conform to International C++ 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 the International Standard

  The Standard Library file stream classes have been corrected to conform to the International 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 must 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.

10.3 Restrictions in Version 6.2

This section describes problems you might encounter when using the current release of the C++ Standard Library with the HP C++ compiler. Where appropriate, workarounds are suggested.

- Do not use Standard Library template definition file names.

  The C++ Standard Library supplies the following template definition files in SYS$LIBRARY:CXXL$ANSI_DEF.TLB:

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, the automatic instantiation mechanism picks up your local copy and does not find the library files. (See *Compiling Programs with Automatic Instantiation* in *HP C++ User's Guide for OpenVMS Systems*.)

It is best not to 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 is declared in the header.

  For example:

  ```
  #include <new.h>

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

  To prevent this behavior, you must change your `new()` declaration to:

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

- Files/Macros for internal use only

  HP C++ Version 6.*n* and higher ships the following non-Standard headers for HP internal use only. Their contents are subject to change and can not be relied on.

  ```
  <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>, <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 HP internal use only. They are subject to change and can not be relied on.

- Pre-ANSI iterators no longer available

  The following classes are no longer in the ANSI draft standard and should not be used:

  ```
  input_iterator
  output_iterator
  forward_iterator
  bidirectional_iterator
  random_access_iterator
  ```

The functionality of these classes is now provided by the single class iterator, which is templatized on the iterator category.

- `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`.

- Specifying a different `l_double_size` than the default size for your particular version of the operating system does not work correctly with the standard library.

- Input and output to and from long double types on when specifying /L_DOUBLE_SIZE=64 does not work if you use standard iostreams. This restriction will be lifted in a future release.
- Overriding operator new() or operator delete()

  You can define a global operator new() or operator delete() to displace the versions used by the C++ Standard Library or C++ Class Library. For instructions, see *Overriding operator(new)* in *HP C++ User's Guide for OpenVMS Systems*.
- The /template_define and /template_define=all options are not guaranteed to work with the Standard Library. Specify /template_define=used instead.

  When compiled with /template_define or /template_define=all, 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 /template_define=all 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 /template_define=all or cxx /template_define, the compiler attempts to instantiate recursively everything that is typedefed, 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 /template_define=used 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.

- basic_string ambiguities

  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;
}
```

  Compilation errors result from the overload resolution between integral and iterator types. The constructors involved are the following:

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

// construct 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, ...)` because `size_type` a `size_t` is an unsigned `int` on OpenVMS Alpha systems. So you have:

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

The second constructor is the better match, but it is undesirable because we are not constructing via 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;
}
```

- Restartable/non-restartable conversion routines in Locale classes

  In `codecvt` class from localization library, the C++ standard assumes availability of restartable conversion routines such as `mbrtowc`, `wcrtomb`, and so forth.

  These routines. introduced by ISO C Amendment 1: C Integrity, were implemented in OpenVMS Alpha Version 6.2A and were never ported back to the previous version.

  Because restartable conversion routines are unavailable, the C++ Standard Library on *OpenVMS* Version 6.2 uses their non-restartable counterparts such as `mbtowc`, `wctomb` and so forth.

  Note that restartable versions of conversion routines are functional only when used with locales that support state-dependent codeset encoding. Currently, neither the OpenVMS Alpha nor the *Tru64 UNIX* operating system provides any locales based on a state-dependent codeset.

With a codeset that is not state-dependent, both restartable and non-restartable conversion routines behave exactly the same. Therefore, using non-restartable conversion routines instead of restartable ones does not constitute any lack of functionality of C++ Standard Library on *OpenVMS* Version 6.2.

- `cxxlink` errors with heavily templatized applications

  When linking an application that uses many templates, cxxlink might report and error indicating that no I/O channels are available. For example:

  ```
  $ cxxlink test
  %LINK-F-OPENIN, error opening
  T4:[TEST.CXX_REPOSITORY]CXX$CTQ173DGTWRTRB0K
  64LIR.OBJ;1 as input
  -RMS-F-CHN, assign channel system service request failed
  -SYSTEM-F-NOIOCHAN, no I/O channel available
  ```

  This error is generated when the process of linking all of an application's template instantiations requires more open files than the underlying system can support. In C++ Version 6.*n* and higher, automatic template instantiation occurs at compile time (not link time as in previous versions. Automatic template instantiation files are therefore written into a repository as object files during compilation. Cxxlink then invokes the *OpenVMS* linker to link each instantiation file into the application.

  To work around this problem, you must allocate sufficient system resources to handle an extended number of open files. You might need to increase the SYSGEN parameter CHANNELCNT, which specifies the number of permanent I/O channels available to the system. You might also need to increase the following process quotas:

  open file quota (`fillm`)
  buffered I/O quota (`biolm`)
  page file quota (`pgflquo`)
  enqueue quota (`enqlm`)

- `cxxlink` might fail with many large template instantiations

  If you have many large template instantiations in your source file, the cxxlink command may fail with the following message:

  ```
  %LINK-F-OPENIN, error opening <some repository> as input
  -RMS-E-ACC, ACP file access failed
  -SYSTEM-F-EXBYTLM, exceeded byte count quota
  ```

To work around this limit, you must increase your Buffered I/O byte count quota, which you can see by entering the command SHOW PROCESS/QUOTA. Ask your system administrator to use the Authorize utility to increase the size. The recommended value is 199296.

- C++ Class Library dependencies on the Standard Library

  As of Version 6.0, the Class Library has dependencies on the Standard Library. The dependencies have to do with incompatible name mangling changes made between Version 5.*n* and Version 6.0 to address Version 5.*n* name mangling problems. In all cases, entry points to the Version 6.0 mangled names are available in the Version 6.0 Standard Library. This means that all Version 6.0 programs (even those using only the Class Library) should be linked with the Standard Library either with `cxxlink` or by adding `sys$library:libcxxstd.olb` to your link command. Using `cxxlink` is recommended.

  The following specific dependencies require support from the Standard Library:

  - Class library routines that use an ios::seek_dir type input parameter:

    ```
    istream::seekg()
    ostream::seekp()
    streambuf::seekoff()
    filebuf::seekoff()
    strstreambuf::seekoff()
    stdiobuf::seekoff()
    ```

  - In Version 5.*n*, the mangled name for an `enum` nested in a class did not contain the class name. This was corrected in Version 6.0. Version 6.0 provides a switch (`/distinguish_nested_enums`) that allows you to compile programs amd generate either a Version 5.*n* or Version 6.0 style mangled name.

    Because support for the routines with Version 6.0 mangling resides in the Standard Library, Version 6.0 programs making use of the above routines compiled using `/distinguish_nested_enums` must link with the Standard Library either with `cxxlink` or by inclusion of `sys$library:libcxxstd.olb` on the link line.

  - Class Library inserter/extractor routines that accept`__int64` type parameters.

In Version 5.*n*, the mangled name for a routine accepting an `__int64` parameter differed from the mangled name for a routine accepting a `long long`. This problem was corrected for Version 6.0 on *OpenVMS*. Because support for the routines with Version 6.0 mangling resides in the Standard Library, Version 6.0 programs making use of such routines in code like the following

```
ifstream instream ("t.dat");
__int64 s_value
unsigned __int64 u_value
     ...
cout << u_value;
instream >> s_value;
```

must link with the Standard Library either with `cxxlink` or by inclusion of `sys$library:libcxxstd.olb` on the link line.

- Access violation when using Standard Library version of `cout cin`, and other standard streams inside static constructors

  If you are using the standard iostreams library (if you have defined the macro `__USE_STD_IOSTREAM`) , and if you attempt to use `cout`, `cin`, `cerr`, `clog`, `wcout`, `wcin`, `wcerr`, or `wclog` in a static constructor, a core dump occurs.

  For example:

```
#include <ostream>

using namespace std;

struct C { C(); };

C::C() {
    cout << "hello world"; // cout not initialized here
}
C c1;

int main() {}
```

  To work around the problem, must modify your `cxxlink` step as follows:

  1. Create an options file, `image_name.opt`, whose content is `SYS$SHARE:LIBCXXSTD/INCLUDE=CXXL_STD_INI`
  2. Modify your cxxlink command to be:

     `$ CXXLINK/EXE=programname.exe image_name.opt/opt,`*`[rest of command]`*

- 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.

- 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.
- 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.

- Specifying a different long double size than the default size for your version of the operating system (using the `/l_double_size` qualifier) does not work correctly with the Standard Library.

11 Release Notes for the V6.0 C++ Compiler

11.1 Enhancements, Changes, and Restrictions in Version 6.0

This section briefly summarizes changes, enhancements, and restrictions made in Version 6.0, including problems fixed.

- 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 on and changes to the Standard Library.
- An updated debugger (dated 6-Jan-1999) that fixes problems with the previous version of the debugger.
- Language mode options For compatibility with previous versions, the compiler supports an ARM language mode and provides both `/standard=ansi` and `/standard=arm` language mode options, as well as options to support other C++ dialects. For details, see *Compiler Compatibility* in *HP C++ User's Guide for OpenVMS Systems*.
- Improved automatic instantiation of templates, including:
  - Fewer restrictions. In particular, HP C++ no longer requires that template declarations and definitions appear in header files.
  - Build-time performance improvements that further reduce build times for applications that use templates extensively.

  For details, see *Using Templates* in *HP C++ User's Guide for OpenVMS Systems*.
- Language feature options

  The compiler supports the following options:
  - Use of alternative tokens (see *Alternative Tokens* in *HP C++ User's Guide for OpenVMS Systems*).
  - Run-Time type identification (see *Run-Time Type Identification* in *HP C++ User's Guide for OpenVMS Systems*).
  - Restriction:

When using /EXTERN_MODEL=RELAXED_REFDEF (the default) or /EXTERN_MODEL=COMMON, the C++ compiler describes each global variable in the VMS Object language as using a unique Program Section (PSECT). The name of the program section is the same as the name of the variable, but it is subject to type encoding and shortening. However, because the compiler must use several PSECTs with fixed names to encode other aspects of the compiled program, these fixed-name PSECTs are not available to users when these EXTERN_MODEL values are in effect. All the names are prefixed and suffixed with a dollar sign ($) character, which generally identifies global names reserved to OpenVMS system usage. The following fixed name PSECTs not available as user variables with /EXTERN_MODEL=RELAXED_REFDEF or /EXTERN_MODEL=COMMON:

ABS, BSS, $CODE$, CXX$$RTTI, $DATA$, $LINK$, $LITERAL$, $READONLY$, TLS

- Performance optimization options

  Version 6.0 provides the following performance optimization options:

  - /[no]ansi_alias
  - /assume=[no]pointers_to_globals
  - /assume=[no]trusted_short_alignment
  - /assume=[no]whole_program

  For details, see *Performance Optimization Options* in *HP C++ User's Guide for OpenVMS Systems*.

The C++ Standard Library provided with this release defines a complete specification (with some differences as described in Section 2.3) of the C++ International Standard. The Standard Library in this release includes for the first time the ANSI `locale` and `iostream` libraries.

Tutorial programs illustrating functionality found in the standard C++ library including the `locale`, `iostream`, and STL classes shipped with this release can be found in:

```
SYS$SYSROOT:[SYSHLP.EXAMPLES.CXX]*.CXX
```

You can compile and run these programs and use them as models for your own coding. The expected output for each program can be found in:

```
SYS$SYSROOT:[SYSHLP.EXAMPLES.CXX]*.RES
```

Version 6.0 introduces the following major enhancements and changes. For detailed information about the HP C++ Standard Library, refer to *HP C++ User's Guide for OpenVMS Systems*.

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

  If you plan to use the standard iostream classes or the standard locale (internationalization) classes, *The C++ Standard Library* in *HP C++ User's Guide for OpenVMS Systems* provides more information than that available in *The C++ Programming Language, 3rd Edition*. The Version 6.0 kit also contains detailed documentation in PostScript and PDF formats:

  Standard Library Internationalization and Localization (PDF)
  Standard Library Internationalization and Localization (PS)
  Standard Library Stream Input/Output (PDF)
  Standard Library Stream Input/Output (PS)

- Several new options.
- pre-ANSI/ANSI iostreams compatibility.
- Support for ANSI/pre-ANSI `operator new()`.
- Support for global `array new` and `delete`.

Additional changes include the following:

- Specific changes to match the 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 was changed from `allocator<T>` to `allocator<pair<const Key, T> >`.

- Interface Change for STL `distance()` function

  Because the V6.0 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 6.0 the result was returned in d
distance(first,last,d);
```

  Beginning with V6.0, the result is returned in the return type:

```
d = distance(first,last); // 6.0
```

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

- 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 in the `/standard=strict_ansi` 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 6.0 uses compile time rather than link time template instantiation, there is no use for common instantiation libraries and we have therefore not supplied any `build_common_instantiation_library` script. See the discussion of templates in *HP C++ User's Guide for OpenVMS Systems*.