HP C++ V7.4-005 Release Notes for OpenVMS Integrity servers

November 6, 2012

This document contains information about new and changed features in HP C++ V7.4-005 for OpenVMS Integrity servers Version 8.3 or higher.

Revision/Update Information:	This is an updated manual.
Software Version:	HP C++ V7.4-005 for OpenVMS Integrity servers Version 8.3 or higher.

Hewlett-Packard Company
Palo Alto, California

© Copyright 2012 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group.

Portions of the ANSI C++ Standard Library have been implemented using source licensed from and copyrighted by Rogue Wave Software, Inc. All rights reserved.

Information pertaining to the C++ Standard Library has been edited and reprinted with permission of Rogue Wave Software, Inc. All rights reserved.

Portions copyright 1994-2012 Rogue Wave Software, Inc.

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

Contents

1 Introduction 1
2 Enhancements, Changes, and Problems Corrected in the V7.4-005 Compiler 1
3 Enhancements, Changes, and Problems Corrected in the V7.4-004 Compiler 2
4 Release Notes for the V7.3 C++ Compiler 3
5 Known Problems in V7.3-009 13
6 Enhancements, Changes, and Problems Corrected in the V7.3-009 C++ Standard Library 13
7 Release Notes for the V7.2 C++ Compiler 14
7.1 New Name Mangling/Prefixing Requires Recompile from Source 14
7.2 64-bit Runtime Libraries 16
7.3 64-bit Pointer Support 16
7.3.1 Pointer_Size Control Differences 18
7.3.2 Mixed Pointer-Size Allocators 22
7.4 Other Enhancements, Changes, and Problems Corrected 24
7.5 Known Problems and Restrictions 26
8 Release Notes for the V7.2 C++ Standard Library 27
9 Release Notes for the V7.1 C++ Compiler 29
9.1 Problems Fixed in V7.1 29
9.2 New Features in V7.1 30
9.2.1 cname Header Support 30
9.2.2 __HIDE_FORBIDDEN_NAMES Predefined in Strict ANSI Mode 30
9.2.3 /[NO]FIRST_INCLUDE Qualifier Added 30
9.2.4 #pragma include_directory Added 31
9.2.5 Messages 31
9.2.6 New Front End 32
9.3 Integrity servers Differences 32
9.3.1 Quotas 32
9.3.2 Dialect Changes 32
9.3.3 ABI/Object Model changes 32
9.3.4 Command-Line Qualifiers 33
9.3.5 Floating Point 35
9.3.6 Intrinsics and Builtins 38
9.3.7 memcpy C Run-Time Library Function 41
9.3.8 ELF 41
9.3.9 Templates 42
9.3.10 Exceptions and Condition Handlers 43
9.3.11 Overriding new and delete 44
9.4 Integrity servers Known Issues 44
10 Release Notes for the V7.1 C++ Libraries 48

10.1 Library Reorganization 48
10.1.1 Standard Library and Language Run-Time Support Library 48
10.1.2 Class Library 49
10.2 Language Run-Time Support Library 49
10.3 Class Library 49
10.4 Standard Library 49
10.4.1 Changes 49
10.4.2 Library Headers 49
10.4.3 Internal Library Headers and Macros 50
10.4.4 Known Issues 50
10.4.5 Differences Between Alpha and Integrity servers system 50
10.4.6 Restrictions in Version 7.1 55
10.4.6.1 Using the C++ Standard Library in Microsoft Standard Mode 56
11 CXXLINK Changes 56
12 Installation 57
12.1 Multiple Version Support 58
13 Reporting Problems 59

1 Introduction

This document contains the release notes for HP C++ V7.4-005 for OpenVMS Integrity servers. The HP C++ product requires OpenVMS Integrity servers Version 8.3 or higher.

The release notes for previous HP C++ versions are also included:

- See Section 3 for the HP C++ V7.4-004 compiler release notes.
- See Section 4 for the HP C++ V7.3 compiler release notes.
- See Sections 7 and 8 for the HP C++ V7.2 compiler and library release notes, respectively.
- See Section 9 for the HP C++ V7.1 compiler release notes, which describe the new features, differences, and restrictions of the C++ V7.1 compiler for Integrity server systems over the C++ V6.5 compiler for Alpha systems.
- See Section 10 for the HP C++ V7.1 release notes for the standard library, language run-time support library, and class library.

2 Enhancements, Changes, and Problems Corrected in the V7.4-005 Compiler

Following are the enhancements, changes, and problems corrected in the C++ Version 7.4-005 compiler for Integrity server systems:

- Performance degradation was observed while using `getenv` calls for getting the environment variables and logical name translations.

  The C++ compiler is modified so that the `getenv` calls are executed once per environment variables and the results are saved. The changes are made to the environment variables BUJ_DEBUG, SWAP, LDP_DUMP_GRAPH, and NOSPEEP, and for logical translation of DUMP_EXTNAM. With this change, the performance of the `getenv` call has significantly improved.

- When a program is compiled with /WARN=(WARN=ALL,DISABLE=<VALUE>) option, wherein the value might be messages such as basclsnondto, the C++ compiler does not honor the DISABLE=message. This issue is fixed.

- When UNIX-style pathnames are passed to `#include` directive, the compiler discards the path and considers only the file name for file search in the standard or user defined text libraries.

  To enable the compiler to allow the UNIX-style module names to be passed to text library for file search operations, the following logical must be set to 1:

  ```
  $ DEFINE UNIX_STYLE_MODULENAME 1
  ```

- A Multithreaded program terminates with `SYSTEM-F-OPCCUS` error when using `__LOCK_LONG_RETRY` function. This issue is fixed.

3 Enhancements, Changes, and Problems Corrected in the V7.4-004 Compiler

This section describes enhancements, changes, and problems corrected since C++ Version 7.4-004 compiler for Integrity server systems.

- When there are several compilations happening simultaneously with /EXPORT_SYMBOLS qualifier specified in a multicore machine, longer compilation time was observed. This has been corrected.
- Under certain circumstances involving code with throw statement, the compiler failed to emit CODEUNREACHABLE information message for the unreachable statements. This has been fixed.
- In certain cases when /EXPORT_SYMBOLS qualifier fails because of the process exhausting its subprocess quota, the compiler would exit without any mesage. This has been corrected so as to make the compiler emit a fatal message.
- Accessing certain class arrays using a negative constant index value could cause incorrect code to be generated. This has been fixed.
- In certain cases the /EXPORT_SYMBOLS qualifier would not emit symbols for virtual thunks to the options file if they created using a template marked with the __declspec(dllexport) attribute. This has been corrected.
- In certain cases the /EXPORT_SYMBOLS qualifier would not emit symbols for subobject constructors and destructors to the options file if they created using a template marked with the __declspec(dllexport) attribute. This has been corrected.
- The __PAL_REMQ did not work properly. This has been corrected. NOTE: This compiler fix also requires a new C++ RTL update. The RTL update should be available in October of 2008.
- Type information placed in the repository was wrong when the type was a qualified typedef type and the typedef was a pointer type. This has been corrected.
- In certain cases the compiler could generate bad code when a function's eighth parameter slot was a floating point number. This has been corrected.
- The options file created when /EXPORT_SYMBOLS=(NOTEMPLATES) was specified would incorrectly output SYMBOL_VECTOR commands for some constructor vtable entries. This has been corrected.
- The options file created when /EXPORT_SYMBOLS was specified could create an options file that contained lines too long for the linker. This has been corrected.
- In some cases compiling with /DEBUG=ALL could cause the compiler to crash. This has been corrected.
- In some cases compiling with /DEBUG/OPT could cause the compiler to crash. This has been corrected.
- The compiler could generate bad code when a file-scope static variable and an function with external linkage had names that differ only by case. For example:

```
  static const int SERVICE_ELMT_ATTR        = 39; // pr_alg_stat.h line 74
and
  int service_elmt_attr(long   serv_elmt_id)      // pr_alg_service_element.h
```

This has been corrected.

- In some cases, particularly when a block with a local declaration of an object with a non-trivial destructor could be determined at compile-time never to execute, and there was control-flow from that block to a block that throws an exception, then the destructor for the local object would sometimes be invoked, even though the object was never constructed. The exact circumstances under which the destructor would be invoked erroneously were not consistent, and were sensitive to the layout of the stack frame.
- When compiling /OPT=LEVEL=2 (or higher) overly large line numbers preceeded by a "#" could be seen in the line number column of a VMS traceback or in VMS debug. This problem has been corrected. But please note that optimized debugging is not supported in this release.
- The builtin functions __MEMSET, __MEMCPY, __MEMMOVE, and __STRCPY are sometimes compiled into calls to ordinary CRTL library functions (e.g. when compiling /NOOPT). The CRTL contains separate implementations of these routines for 32-bit versus 64-bit compilations, but the compiler was always generating calls to the 32-bit versions, even under /POINTER_SIZE=64. This problem has been corrected.

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

This section describes enhancements, changes, and problems corrected in the C++ Version 7.3 compiler for Integrity server systems.

HP C++ V7.3-009 is largely a bug-fix release of the compiler, although it does contain some significant new features including multiple version support, new exception processing mode `pure_unix`, new command line qualifier /EXPORT_SYMBOLS, and an unsupported/experimental mechanism for generating a customized machine_code listing. The following describes these features along with problems fixed and restrictions in this version.

- Multiple Version Support

  Version 7.3 adds optional support for having multiple versions of the HP C++ compiler on your system. It works by appending an ident name to a previously installed compiler and saving it alongside the new compiler from this kit. Users on your system can then execute the sys$system:cxx$set_version.com and sys$system:cxx$show_versions.com command procedures to select the desired compiler for a given process and to view the list of available compiler versions.

  To set this up, have your system administrator run the installation procedure, answering NO to the question about default options:

  ```
  Do you want the defaults for all options? [YES] NO <RET>
  ```

  Then answer YES to the question about making alternate compilers available:

  ```
  Would you like to set up your system for running alternate versions of C? [NO] YES <RET>
  ```

  Users can then execute the cxx$set_version.com command procedure with an argument:

  ```
  $ @sys$system:cxx$set_version  V7.2-018
  ```

Or without an argument:

```
$  @sys$system:cxx$set_version.com

The following  HP C++ compiler(s) are available in SYS$SYSTEM

Filename                      Version     Defaults
-----------------------------------------------------
CXX$COMPILER.EXE              T7.3-018    System Default
CXX$COMPILER_T07_03-018.EXE   T7.3-018
CXX$COMPILER_V07_01-011.EXE   V7.1-011
CXX$COMPILER_V07_02-018.EXE   V7.2-018

Enter Version number or SYSTEM: V7.1-011
```

Notice that when cxx$show_versions.com is executed without an argument, it displays a list of possible compilers and prompts you for a version number. Also notice that you can revert to the installed compiler by selecting SYSTEM as the version number.

The cxx$set_version.com command procedure sets up the logicals CXX$COMPILER and CXX$COMPILER_MSG to point to the location of the target compiler and its message file. In addition, it issues a SET command to select the appropriate CDL file to select the correct set of qualifiers for the specified compiler version. Please remember that SET commands are not inherited by subprocesses. Make sure that all subprocesses reissue the necessary cxx$set_version.com command procedure.

For a sample installation with multi-version support, please see the installation section.

- A startup procedure, CXX$STARTUP.COM has been added to the PCSI product install kit. It contains commands that can be executed after the product install procedure has been run and at startup to allow for the best compilation performance. You may want to invoke this command file from your system's site-specific startup file. This command file does not have to be invoked for correct operation of HP C++.

- A new process-wide exception processing mode, `pure_unix`, has been introduced. In this mode, non-C++ exceptions, also known as OpenVMS conditions, cannot be caught in a C++ catch-all handler. This mode can be requested by calling cxxl$set_condition(condition_behavior) with a `pure_unix` argument:

  ```
  cxxl$set_condition(pure_unix);
  ```

  The condition_behavior `enum` declared in the `<cxx_exception.h>` header has been extended to include the `pure_unix` member.

  To demonstrate how `pure_unix` mode works, consider the following program. As written, it crashes with an ACCVIO. If the call to cxxl$set_condition( ) is commented out, the program outputs "caught" and exits.

  ```
  #include <stdio.h>
  #include <cxx_exception.h>

  void generateACCVIO() { *((int*)0) = 0; }
  ```

```
int main() {
   cxxl$set_condition(pure_unix);
  try {
     generateACCVIO();
  }
   catch(...) {
     puts("caught");
  }
}
```

- The alignment option of the `#pragma extern_model` directive will now be correctly processed. Earlier releases of the compiler would silently accept and ignore any alignment option.
- Optimizing the code generated for the __ATOMIC_INCREMENT_LONG builtin function sometimes produced incorrect code. E.g. the register holding the address of the location to increment would not be set up correctly, or register r0 would be used, resulting in an access violation at run-time. This problem, and its equivalent in the _QUAD version, and both the LONG and QUAD versions of __ATOMIC_DECREMENT_* have been fixed.
- In some cases, when a for loop used an unsigned int variable as the index, and compared that index in an ordered comparison against a constant of type unsigned int for the termination condition, the loop would never terminate.
- In some cases, functions containing unreachable code could cause the compiler to crash when optimized.
- In some cases, functions containing switch statements within a try block could be incorrectly optimized. This problem has been fixed.
- A workaround was added for debugging of symbols in top level unnamed namespaces. In the debugger they will appear to be global symbols so that they can be examined.
- Creating OpenVMS shareable images that contain C++ code has long been a problem for users. When building a shareable image, you must specify a list of exported global symbols. For C++ code, determining this list is very difficult for the following reasons:
  - Required C++ name mangling makes it difficult to know the name of the external symbol created for a C++ name.
  - OpenVMS CRC encoding (to 31 characters) further complicates mapping source names to object names.
  - Certain C++ constructs require compiler-generated names to be created and exported.

  To help solve the problem, this release of the compiler provides a new compiler qualifier /EXPORT_SYMBOLS and new declaration modifier `__declspec(dllexport)`. The format for /EXPORT_SYMBOLS:

  ```
  /EXPORT_SYMBOLS=(OPTIONS_FILE=<name>
                  [,EXCLUDE=<list of images>]
                  [,export_option]
                  [,NOTEMPLATES])
  ```

  The default file extension for <name> is .OPT

  If the file exists, the compiler appends to it. If the file does not exist, the compiler creates it.

The output for the compilation is:

```
!
! Entries added for <module>
!
<symbol vector>
<symbol vector>
.
.
.
```

The output file is suitable input to a linker options file that can be used to build a shareable image containing the compiled object.

The format of each <symbol vector> is:

```
SYMBOL_VECTOR=(<global name>={DATA | PROCEDURE}) ! <comment field>
```

The <comment field> format is:

```
<unmangled name> [<promoted static flag>] [<class information>]
```

The <promoted static flag> is one of the following:

PSDM - for promoted static data members
PTSDM - for promoted template static data members

The <promoted static flag> is output whenever the symbol is a promoted local static or a promoted template static data member. This is important because these variables, while declared static, actually become global symbols when compiled.

The <class information> field is present if the symbol is a member of a class. It contains the name of the class.

Notes

- When /EXPORT_SYMBOLS is specified, an object file must also be generated. So /EXPORT_SYMBOLS cannot be used with /NOOBJ, /PREPROCESS_ONLY, or any other qualifier that prevents the creation of an object file.
- When the options file already exists, the compiler reads all the symbols that are listed there. If the current compilation also defines one of those symbols, that symbol will not be added to the options file. This is necessary to prevent SYMVALRDEF warnings from the linker.
- When the compiler reads the existing file, it treats SYMBOL_ VECTOR directives that are in comments (of the form !SYMBOL_ VECTOR...) as if they were not commented. In this way, if a user does not want to export a symbol, placing it in comments will prevent the compiler from emitting a directive for that symbol when it compiles other sources that might also define the symbol.
- The symbols placed in the options file are a subset of the symbols defined in the output object file. The export_option value controls exactly which symbols are placed there. There are three choices:
  - ALL - Place all symbols suitable for placement in a sharable image into the options file. The compiler knows that certain symbols are not suited for placement in a shareable image

and excludes them from the options file. Some examples are certain compiler-generated constructor/destructor jackets and symbols in the unnamed namespace.

- o EXPLICIT - Place only those symbols marked with the `__declspec(dllexport)` declaration modifier into the options file.
- o AUTOMATIC (D) - If the compiler processes a `__declspec(dllexport)`, then act as if EXPLICIT was specified. If the compiler does not process a `__declspec(dllexport)`, then act as if ALL was specified.

- The EXCLUDE option of the /EXPORT_SYMBOLS qualifier can be used to specify a list of shareable images. The compiler searches these images for any symbols that it might want to place in the output options file. If it finds the symbol in the image, then that symbol will not be put into the options file.
- The NOTEMPLATES option of the /EXPORT_SYMBOLS qualifier can be used to control the emission of symbols associated with template instantiations. Specifying this option causes the compiler to suppress symbols created by template instantiation. This includes instantiations of class templates, its data members and member functions, and instantiations of function templates. This option could be used to avoid multiple definition diagnostics from the linker if multiple sharable images might be instantiating (and exporting) the same template symbols. Symbols marked with `__declspec(dllexport)` still get exported. This option has no effect on symbols from template specializations. Note that while this option might make the sharable images smaller by not exporting the template symbols, the executable image that links with these sharable images might be larger because it will contain the instantiated template symbols.

Expected Usage:

Because shareable images almost always contain a number of objects, the commands for creating the options file the first time might be:

```
$ DELETE options_file.OPT;*
$ CXX SOURCE1/EXPORT_SYMBOLS=OPTIONS_FILE=options_file
$ CXX SOURCE2/EXPORT_SYMBOLS=OPTIONS_FILE=options_file
$ CXX SOURCE3/EXPORT_SYMBOLS=OPTIONS_FILE=options_file
.
.
.
$ CXX SOURCEn/EXPORT_SYMBOLS=OPTIONS_FILE=options_file
```

Where SOURCE1 - SOURCE*n* are the sources for the shareable. After the compilations, the options_file.OPT will contain correct symbol vector information for the shareable.

The first time this options file is created, it can be considered a candidate options file. It contains all the symbol vector entries for all the C++ globals that make sense to export from the C++ language point of view. A user can then edit this file to exclude (by commenting out) entries that should not be exported, based on the design of the library.

Once an options file is created, it should be maintained for input to subsequent compilations. In this way, any new symbols caused by a change in the source will be added to the end of the compilation. Any existing symbols will not be added, as described in the NOTES section above. This technique ensures that the order of symbols remains unchanged, and that future shared libraries are compatible with existing ones.

- A new option to the /POINTER_SIZE=LONG qualifier is available. When /POINTER_SIZE=LONG=ARGV is specified, the argv argument to main will be comprised of long pointers instead of the short pointers. This can make using long pointers easier as the pointer size of argv will match the default pointer size for the compilation.
- Calls to the CRTL function `tempnam()` were recognized as intrinsic, and could be optimized not to set up the argument information register (R25). This could cause the function to behave erratically at run-time, because the CRTL implementation of `tempnam()` supports a non-standard optional 3rd parameter to control the style of the filename it produces, and thus it uses the value in R25 to determine how it should behave. The compiler no longer performs this erroneous optimization.
- Calls to the CRTL function `time()` were recognized as intrinsic, and could be optimized not to set up the argument information register (R25). This could cause the function to behave erratically at run-time, because of an undocumented CRTL feature that allows a call to the `time()` function without an argument to be treated the same as a call with a NULL pointer argument. So if the R25 register happened to contain a zero at the point of call, the function would only return the time value, and fail to store it into the memory pointed-to by the argument to the call. The compiler no longer performs this erroneous optimization.
- In some unusual situations where an object was constructed within an inner block that "could" be exited via an exception (but wasn't), the destructor for the object could be invoked a second time, after the destructor invoked at the end of the object's scope. This problem has been fixed.
- Several problems involving bad sign extensions when using /POINTER=64 have been corrected.
- A bug causing the compiler to crash when performing pointer arithmetic on function pointers has been fixed.
- The compiler no longer emits incorrect MAYLOSEDATA diagnostics for some simple expressions such as when the address of a local variable is assigned to a short pointer.
- When compiling with the /STANDARD=ARM qualifier, user-defined conversion functions are not called when casting to a reference. This matches the behavior of the Alpha compiler in the /STANDARD=ARM mode.
- The diagnostic UNINIT issued when more than one member of a union is initialized, now has a severity of ERROR. Previously, it had a severity of WARNING.
- The diagnostic VIRSTAT issued when a static member function is declared with the virtual keyword, now has a severity of ERROR. Previously, it had a severity of WARNING.

- The text for the diagnostic BADINITYP issued when a pointer to a bound function is used in expressions other than to call it, has been modified to make the diagnostic clearer.
- A bug in the compiler which caused some sign extensions to be missing when right-shifting signed values, has been fixed.
- A problem was fixed with numbering of group sections when more than 65K sections were needed.
- Beginning with this release the compiler will generate the source correlation records used by debug and the traceback facility to map a source file and line number onto the familiar listing line numbers used by VMS. This resolves various problems with source correlation when using templates and when a include files are included more than once in the same compilation unit. When using C++ you will now see listing line numbers from traceback and from the debugger. You will no longer see unix-like source file line numbers.
- A bug making it impossible for the program to catch C++ exceptions after a `kill()` signal was received and caught by the signal handler has been fixed but requires the runtime components kit VMS83I_ICXXL-V0100, VMS821I_ICXXL-V0200 or higher.
- Improvements have been made in the demangling information for thunks. Thunks are now clearly marked as such in the repository and in the object for the debugger. Also, a bug was fixed where the unmangled name for a thunk that was not defined in the current module was incorrect.

  Previous versions of the compiler generated an incorrect demangled name for the thunk CXX$ZTHN4N6PARENT5PRINTV019MIAE when compiling the test case below. The unmangled name in the cxx_reprository now appears as: "non-virtual THUNK for unsigned int parent::print()".

```
// classes.h
//
struct base
{
    virtual char * name()
    {
 return "Unknown";
    }
};

struct base2
{
    virtual unsigned int print() = 0;
};

// switching order of base and base2
//  makes thunk error go away
struct parent : base, base2
{
    unsigned int print();
};

struct child : parent
{
  child() {}
};
```

```
// thunk.cxx
#include "classes.h"
int main()
{
    child c;
    return 0;
}
```

Note: A thunk is a segment of code associated with a target function, that is called instead of the target function for the purpose of modifying parameters (for example, the this pointer) or other parts of the environment before transferring control to the target function, and possibly making further modifications after its return.

- The IA64 ABI requires that "wrapper" routines be provided around constructors and destructors. The compiler now generates an unmangled name prefix of "$complete$" for the C1/D1 wrappers, "$subbject$" for C2/D2 wrappers, and "$deleting$" for D0 wrappers. Initial versions of the V7.2 compiler placed the prefix in a different part of the unmangled name when generating repository names than when generating debugger unmangled names. The compiler is now more consistent.
- Certain cases using `#pragma message(<string literal>)` could cause the compiler to crash. This problem has been fixed.
- Certain code constructs would cause an assertion in the compiler in the routine do_all_namespace_member_promotion. This problem has been fixed.
- The HP C++ V7.2 release changed the severity of certain NEVERDEF diagnostics from -W- to -E-. This update kit changes the severity back to -W-.
- In cases where the second operand of the conditional (?:) operator was a string literal, the compiler could generate bad code if the /POINTER_SIZE=LONG qualifier was specified. This has been corrected.
- The compiler no longer accepts the `__inline` and `__inline__` language extensions when /STANDARD=STRICT or LATEST is specified.
- When /STAND=GNU is specifed, the HP C++ V7.2 compiler would sometimes emit an incorrect OPNDNOTCLS diagnostic. This has been corrected.
- The V7.2-018 compiler would crash if static data members were declared in the `common_block extern_model`. As some C++ header file contain such declarations, including those headers in a non-default `extern_model` could cause a compiler crash. For example, the following would crash the compiler:

  ```
  #pragma extern_model common_block
  #include <string>
  ```

  This update corrects the compiler crash.

  Take great care when using the non-default `extern_model`. The main purpose of `extern_model` is to allow C++ to share global data with code written in other languages. Declarations that cause data to be allocated according to the C++ object model, that is, declarations for other than POD (Plain Old Data) objects, cannot generally be shared reliably with other languages, and should only appear in regions of source that are subject to the default `extern_model` of `relaxed_refdef`.

Within regions of source subject to an `extern_model` other than `relaxed_refdef`, declarations that allocate data with names visible to the linker should be limited exclusively to POD types. In particular, declaring a C++ class containing a static data member within such a region might produce unintended behavior.

- The use of the += operator where the left operand was a 64-bit pointer could produce incorrect results. This has been corrected.
- Certain parameter information placed in the demangler database would sometimes contain too many levels of indirection. This has been corrected.
- Certain programs that were compiled /DEBUG/POINTER_SIZE=32 and also contained #pragma pointer_size 64 directives could crash the compiler with an access violation in TAG_Emit_Subprogram. This problem has been corrected.
- Using the operator new in a mixed pointer-size compilation could sometimes cause the compiler to crash with with an assert in CAREA:[SRC.IPF.EDGCPFE]EXPR.C;1. This has been corrected.
- If a user program tried to define an overloaded function called mktemp, the compiler would not create the mangled names correctly. This could lead to only one function being created. This problem has been corrected.
- While creating the unmangled routine name information for the demangler database, the compiler would access an internal representation of the parameter types that sometimes could be NULL. Instead of then accessing another internal representation which provides the same information, the compiler was incorrectly raising an assertion about the NULL value. This has been fixed.
- In the C++ language run-time support library, the implementation was incorrectly assuming that the elements in an internal linked list would get reused, and were not being deallocated. These elements were not being reused and therefore resulted in a memory leak. The library now deallocates the elements in the linked list as soon as they are not needed.
- The C++ language run-time support library, was not initially coded with the ability to be loaded into the 64-bit address space. As a result, when applications were linked with /SEGMENT_ATTRIBUTE=CODE=P2, or when the library was installed resident, the library would sometimes produce an accvio. This has been fixed.
- Restriction: The link command qualifier /SEGMENT_ ATTRIBUTE=CODE=P2 causes the executable code for an image to be loaded into P2 space when the image is activated. The code generated by the C++ compiler for 32-bit pointer applications (that is, compilations that do not specify the /POINTER_SIZE qualifier, or that specify /POINTER_ SIZE=32), is not generally compatible with this link qualifier. While some 32-bit C++ compilations may run correctly when linked this way, the code is likely to encounter an access violation at run-time; and 32-bit code compiled with optimization disabled is more likely to fail than code compiled with optimization enabled.

  The link command qualifier /SEGMENT_ATTRIBUTE=CODE=P2 should only be used when all C++ compilations in the program are compiled with /POINTER_SIZE=64, and when the C++ libraries supplied with this kit (or newer) are used. The previous libraries had a problem that could cause a run-time accvio in 64-bit C++ code that used exceptions and was linked with /SEGMENT_ATTRIBUTE=CODE=P2.

- Unsupported machine_code listing mechanism.

  An experimental/unsupported feature has been added to the compiler which causes it to invoke a user-controllable sub-process to produce the disassembly style of machine code listing (i.e. the listing that is produced under /LIST/MACHINE_CODE/OBJECT). This experimental behavior is triggered and controlled by logical names at compile time.

  Of particular note is that if the "gawk" stream editing program is available on the system, then if logical name DECCXX$GAWK_EXE is defined to point to the executable image for gawk, and logical name DECCXX$MACH_LIST_SCRIPT is defined as NL: then the compiler will attempt to generate and invoke a DCL script that runs both ANALYZE/OBJECT/DISASSEMBLE and ANALYZE/OBJECT/SECTION=DEBUG=LINE on the object module produced by the compiler. The generated DCL script then runs a gawk script that produces the machine code listing by editing the ANALYZE/OBJECT/DISASSEMBLE output to:

  - output a table of source files read by the compiler, with each file numbered for reference;
  - append a //-style comment with the listing line number, source file number, and line number within source file, to each machine code instruction for which the source information differs from the information for the preceding instruction;
  - append a //-style comment with the demangled name (from the demangler database in the repository) to each line that contains the label symbol that begins a function definition. This behavior is suppressed if a logical name or DCL symbol definition for DECCXX$MACH_LIST_NODEMANGLE exists, the purpose being to prevent the gawk script from reading the entire demangler database file into memory.

  Note that gawk can be obtained from the OpenVMS FreeWare CD, e.g. by downloading it from
  http://openvms.compaq.com/freeware/freeware80/.

  Instead of defining logical DECCXX$MACH_LIST_SCRIPT to NL:, the user may also define it to point to an existing DCL script. If the name resolves to an existing readable file with non-zero length, the compiler will attempt to invoke it as a DCL script, passing it two arguments: the filespec for the object module it generated, and the filespec for the repository (appending "CXX$DEMANGLER_DB." to the second argument produces the name of the demangler database). That script might or might not use DECCXX$GAWK_EXE - the compiler itself does not do anything with that logical name, the use of it is only within the DCL script that the compiler generates if DECCXX$MACH_LIST_SCRIPT is defined but does not resolve to a readable file with non-zero length.

  In the case when the compiler generates the DCL script that it invokes, the script is created with the name "SYS$SCRATCH:CXXLIS_''F$UNIQUE()'.COM", and the compiler deletes the script after it has been invoked. But if DCL symbol MACH_LIST_SCRIPT$DEBUG is defined with a value of 1, then the compiler does not delete the script it generated. Additionally, the generated script itself tests if DCL symbol MACH_LIST_SCRIPT$DEBUG is defined with a non-zero value, and if so it:

  - turns on DCL verification;
  - enables debugging output in the gawk script it generates and runs;

- does not delete the three temporary files it creates, in files "SYS$SCRATCH:ZZCODE-''F$UNIQUE()'.ANL-DIS", "SYS$SCRATCH:ZZCODE-''F$UNIQUE()'.ANL-LINES", and "SYS$SCRATCH:ZZCODE-''F$UNIQUE()'.GAWK".

It cannot be overemphasized that this is an experimental feature. Problems encountered by the scripts may not be handled gracefully, and may well exhaust resources. Improperly-defined logical names will typically cause the listing to revert to its normal form as if the logicals were not defined, although certainly other less desirable behaviors could occur. The feature has been only minimally tested on relatively small compilations, but in those cases the output produced was accurate and very useful, and that is why it is being made available for experimental use. There is no assurance that this mechanism, or the form of output it produces, will be continued in future releases of the compiler.

5 Known Problems in V7.3-009

The following are known problems in this release of the compiler:

- The `#pragma extern_model` directive does not support the alignment options `PAGE` and `16`.
- The compiler might emit an erroneous BADANSIALIASn message.

  In some situations the compiler's loop unrolling optimization can generate memory accesses in the code stream that never actually execute at run-time, but that would violate the ANSI Aliasing rules if they did occur. 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 disapper 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 is best to measure the impact.

6 Enhancements, Changes, and Problems Corrected in the V7.3-009 C++ Standard Library

The following problems are fixed in this version of the C++ Library:

- As described in code_example(<http://issues.apache.org/jira/browse/STDCXX-397>), __introsort_loop( ) function in code_example(<algorithm.cc>) header has a bug which, for some input sequences, can adversely affect performance of std::sort. See the Apache tracker for the issue STDCXX-397 at URL above for more details.

The bug has been fixed. However, for some input sequences, the fix can change the behaviour of std::sort with regard to the relative order in which elements that have equivalent ordering are placed into the sorted sequence. While this change in behaviour is permissible because, unlike std::stable_sort, std::sort does not guarantee any particular relative order of elements having equivalent ordering, to avoid breaking applications that rely on existing behaviour of std::sort, the fix is conditionalized with __RW_FIX_APACHE_STDCXX_397 macro and is in effect only when the program is compiled with this macro defined. [L2028]

- 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);
  }
  ```

- According to section 27.6.1.3 [lib.istream.unformatted] of the C++ Standard, the following get member functions of the std::basic_istream class should call setstate(failbit) if no characters have been stored, as is the case for an empty line. While on Integrity server systems the functions set failbit, on Alpha systems they do not:

  ```
  istream_type& get(char_type *s, streamsize n, char_type delim);
  istream_type& get(char_type *s, streamsize n);
  ```

  See Section 10.4.5 for more information.

7 Release Notes for the V7.2 C++ Compiler

This section describes enhancements, changes, and problems corrected in the C++ Version 7.2 compiler for Integrity server systems.

7.1 New Name Mangling/Prefixing Requires Recompile from Source

Note

The V7.2 compiler generates different mangled names from V7.1 for user code. For 32-bit (default) compilations, V7.2 prefixes mangled names with "CX3$", where V7.1 used "CXX$". C++ library names remain unchanged, using the "CXXL$" prefix for 32-bit code. Applications built with the V7.1 compiler must be fully-recompiled from source when moving to V7.2. An application containing both "CXX$" and "CX3$" prefixed names will not work correctly.

The introduction of 64-bit pointer support, described later, uncovered some errors in the names generated by the V7.1 compiler that could introduce incorrect run-time behaviors in standard-conforming programs without any diagnostic. These behaviors could range anywhere from harmless, to subtle, to access violations - and they are very difficult to diagnose. Basically, the names generated for certain globals such as initialization guard variables, vtables, and RTTI information used to identify exceptions embed the names of types. Type names must always be treated as case-sensitive in either C or C++. The V7.1 compiler erroneously treated these names, if they happened to be less than 32-characters long, as being subject to the `/NAMES=` command-line qualifier, which by default uppercases them. In addition, some of these names were not being given the facility prefix of "CXX$", even though they were compiler-generated and not explicitly present in the source code.

Because the Integrity server implementation uses object module "group sections" (sometimes called comdats) to enforce the C++ "one definition" rule, a mismatch in generated names usually results in more than one definition for the same source entity without any diagnostic; whereas on OpenVMS Alpha or in languages other than C++, a mismatch usually results in a link-time diagnostic for an unresolved reference.

V7.1 was the initial release of the Integrity servers compiler, and the effects of mismatches caused by the V7.1 naming bugs can be very subtle and difficult to diagnose. And it was important to make the mangled names produced by V7.2 for 64-bit compilations not only correct but identical to the names it produces for 32-bit compilations (except for the prefix that distinguishes the pointer-size model). Therefore it was decided to change the default prefix for 32-bit compilations in order to distinguish object code that could contain the naming bugs (prefixed by "CXX$") from object code that does not contain the naming bugs (prefixed by "CX3$").

Applications that are linked from object modules against the C++ library shareables should not be affected when fully recompiled from source using the new compiler and relinked.

Shareable images built from 32-bit C++ object modules would not generally have universal symbols prefixed by the compiler's default prefix, but rather they would normally use `#pragma extern_prefix` to give their universals their own namespace (or export only C-linkage names, which are unchanged in V7.2). Users of such libraries would be unaffected unless one or more universals they use actually were affected by a naming bug. From experience with the C++ libraries this is thought to be relatively uncommon. And in that case, the build of the shareable image would fail when first built from object modules compiled by the new compiler, because correcting the naming bug would change the name of such a symbol. The library provider would then need to change the symbol vector to provide the new name as a universal, and alias the old name to it, which would again leave users of the library unaffected regardless of whether they compiled with the old or new compiler.

For shareable images built from object modules compiled by V7.1 that did not use `#pragma extern_prefix`, but instead directly exported symbols prefixed by "CXX$" (or exported erroneously unprefixed mangled names, which can be recognized as those beginning with "_Z"), the link would also fail when built from objects produced by V7.2. But in this case all of the symbol vector entries would fail because the prefixes would be different. A solution would be a global edit to the options file to change all of the "CXX$" prefixes to "CX3$", and prepend "CX3$" to all symbols beginning with "_Z", and relink. Failures in the relink

would identify symbols that were affected by the naming bugs, and those would need to be corrected as in the preceding paragraph. Finally, aliases would need to be added from the original names to the new names to make the shareable usable to code produced by either V7.1 or V7.2 compilers.

For code that must link against object modules or shareable images that cannot be recompiled from source, and which contain names affected by the naming bugs (note this does *not* include the C++ libraries), the simplest soulution is to use the V7.1 compiler if any such code needs to be recompiled. If that is not feasible, unsupported switches may be available from your support contact to ease this situation.

The need to avoid mixing V7.1 32-bit object modules with V7.2 32-bit object modules cannot be over-emphasized - compiling one module with V7.2 requires full recompilation from source of all object modules. The primary purpose of changing the default prefix is to make mixing easy to detect by examining the link map: if the map contains both "CXX$" names and "CX3$" names, the modules containing "CXX$" names need to be recompiled by the new compiler.

7.2 64-bit Runtime Libraries

The runtime libraries for HP C++ ship with the OpenVMS operating system. This compiler kit adds support for 64-bit pointers, which requires the 64-bit runtime libraries be available. Those new libraries will ship with a future release of the OpenVMS operating system. A patch kit for the ICXXL component of the operating system is available which provides the new libraries for older versions of the operating system.

7.3 64-bit Pointer Support

This version of the compiler adds support for 64-bit pointers. This support is compatible with the 64-bit pointer support in the OpenVMS Alpha C++ and C compilers. It supports the same /POINTER_SIZE command-line qualifier, the `__INITIAL_POINTER_SIZE` predefined macro, and the same pragmas (`#pragma pointer_size` and `#pragma required_pointer_size`). However, the basic model for how and where pointers with a size different from the default size (the size specified by the command-line qualifier) can be declared and used in the language is considerably more limited than it is in the other compilers.

Note

Limitations on the use of non-default-sized pointers are not generally diagnosed or enforced by the compiler. Programs that do not follow the model of mixed-size pointer usage outlined below are likely to fail at run-time without any compile-time diagnostic.

The best-supported model of 64-bit pointer usage is when the command line uses the /POINTER_SIZE=64 qualifier, called a 64-bit compilation. In that case the entire C++ program is considered to use 64-bit addressing, with a few exceptions made to permit the use of data structures containing 32-bit pointers that are needed to communicate with OpenVMS services and other non-C++ libraries. Such data structures naturally contain only pointers to POD types, and the functions that operate on those types naturally have extern "C" linkage. The declarations of those data structures and functions reside in header files, and those header files are coded to use the `__INITIAL_POINTER_SIZE` macro and the pointer_size pragmas to ensure that they use appropriately-sized pointers

regardless of the compilation mode. Except for those declarations, all addresses, pointers, and references in a C++ program compiled with /POINTER_SIZE=64, are considered to be 64-bit types, and all C++ "new" operators allocate data from a 64-bit heap.

If the command line specifies the /POINTER_SIZE=32 qualifier, then it is a 32-bit compilation, and the only use of 64-bit pointers can be pointers to POD types provided by calls to _malloc64( ), or obtained from other non-C++ code.

While it is possible to use `#pragma pointer_size 32` to declare 32-bit pointers explicitly within a 64-bit compilation, the region of source code covered by such pragmas should be made as small as possible, and preferably confined just to typedefs for pointer types that must be 32-bit pointers. In general, the source region should not contain class definitions for non-POD types, template declarations, declarations of functions that have C++ linkage, or executable code. This differs significantly from C++ for OpenVMS Alpha, which permits C++ classes to be defined with 32-bit pointers in a 64-bit compilation.

Another significant difference is that the Alpha compiler attempts to determine the "best" pointer size to use when determining the type of an "address-of" expression. It uses the fact that on OpenVMS, C++ declared objects (either stack-based or static-extent) always have addresses that fit in 32-bits; and for expressions that involve pointer-dereferencing it uses the width of the pointer that was dereferenced as the width of the pointer type given to the expression. This usually allows address-of expressions to be assigned to pointers without casting. In 64-bit mode, the Integrity servers compiler assumes that an address-of expression will yield a 64-bit pointer unless its operand is a dereference of a 32-bit pointer, and so it may issue spurious `NARROWPTR` warnings for assignments of address-of expressions to 32-bit pointers; an explicit cast to the correct 32-bit pointer type is needed to silence the warning. As a special case, 64-bit pointer-to-function values may be assigned to 32-bit pointer-to-function objects without complaint (the value of a function pointer always fits in 32-bits on OpenVMS).

Neither Alpha nor Integrity server compilers include the pointer size when forming mangled names, so naturally it is not possible to overload functions based only on differences in pointer size: the Alpha compiler reports this explicitly at compile time when possible, but on Integrity servers it will just produce conflicting multiple definitions. In general, if a 32-bit pointer type needs to appear in a function prototype in a 64-bit compilation, it is a good practice to define a `struct` type just to hold the pointer, and pass the `struct` instead of the pointer type.

Except for names declared with extern "C" linkage, the Integrity servers compiler produces completely disjoint external symbols in the object modules for 64-bit compilations and 32-bit compilations. For 64-bit compilations, user-declared names are prefixed by "CX6$" (instead of "CX3$"), and library names are prefixed by "CX6L$" (instead of "CXXL$"). Following the prefix, names are mangled identically in the two modes, so a given source declaration will produce the same name in the object module differing only by the prefix if compiled in different modes. So while it is possible to include both 32-bit and 64-bit compilations in the same program, they will not interact with each other except through extern "C" linkage names.

7.3.1 Pointer_Size Control Differences

Although the syntax and use of pointer_size controls are the same for Alpha and Integrity servers, and most programs that work correctly on Alpha should also work on Integrity servers without change, the differences in the compiler's model of how the pointer_size controls work on the two platforms can affect behavior, particularly in programs that create pointers to C++ objects (non-POD types) having a size that differs from the setting of the /POINTER_SIZE command-line qualifier, or that apply the `sizeof` operator to expressions that have a pointer type:

- On Alpha systems, the compiler uses an object model that generally supports the declaration and use of both 64-bit and 32-bit pointers, and the `#pragma pointer_size` directives can be placed at most any point in the source code. The pointer_size in effect at any given point in the source generally influences both the size of pointers created for explicit pointer declarations, and also the size of pointers generated by the compiler to implement various language constructs. There are some restrictions such as no overloading of functions based on pointer size, but generally speaking the size of individual pointer types and the current setting of the pointer_size from the #pragmas is taken into account at fairly low levels, and pointers to C++ objects (non-POD types) can be of either size. Modules compiled with one setting of /POINTER_SIZE can in some cases interoperate with code compiled with a different setting, although this is not a good practice. And in general, the result of the address-of operator (&) has a pointer type whose size reflects the current pointer size.

- On Integrity server systems, the /POINTER_SIZE command-line qualifier plays a much larger role than on Alpha systems. This qualifier chooses between two different, incompatible, binary models for C++ code generation. Functions with C++ linkage produced under /POINTER_SIZE=64 cannot be called from functions with C++ linkage produced under /POINTER_SIZE=32 (the default with no /POINTER_SIZE qualifier), and vice-versa.

  And 64-bit compilations use a completely separate implementation of the C++ libraries from 32-bit compilations.

  Furthermore, the /POINTER_SIZE qualifier on Integrity server controls much more completely the compiler's determination of pointer size throughout the compilation. Basically, the only pointer types that are given a size different from the size specified on the command line are pointer types explicitly declared with the "*" declarator syntax in the context of a `#pragma pointer_size`. And the only pointer types that should be given a pointer_size different from that specified by the command line are pointers to POD types. Pointers to non-POD types, as well as all implicitly-generated pointer type are assumed to have the size specified by the command line. Of special note, the address-of operator (&) generally produces a pointer whose size is based on the size specified by the command line. This is a difference from Alpha. The one exception is when the operator is applied to a pointer dereference, in which case the result is the size of the dereferenced pointer. This exception matches the Alpha behavior.

  The compiler does not diagnose the use of pointers having a size different from the command-line size that point to non-POD types; in some situations such pointers may produce intended results, but in general their use may cause unexpected behaviors or access violations at run-time.

The following example program illustrates some of these differences:

```
#include <stdio.h>
#if __INITIAL_POINTER_SIZE == 64
#define CMD "/POINT=64"
#else
#define CMD "/POINT=32"
#endif
#if __ALPHA
#define MACH "Alpha"
#else
#define MACH "I64  "
#endif

void main(void) {
    printf(MACH CMD ":\n");
    {
    #pragma __required_pointer_size 64
    #define SZ "  #pragma 64: "
    int i;
    printf(SZ "sizeof(&i) = %d.\n", sizeof(&i));
    printf(SZ "value of &i = 0x%016llx.\n", (long long)&i);
    printf(SZ "sizeof(&\"str\"[1]) = %d.\n", sizeof(&"str"[1]));
    printf(SZ "value of &\"str\"[1] = 0x%016llx.\n", (long long)&"str"[1]);

    const char array[] = "str";
    printf(SZ "sizeof(&array[1]) = %d.\n", sizeof(&array[1]));
    printf(SZ "value of &array[1] = 0x%016llx.\n", (long long)&array[1]);

    printf(SZ "sizeof(new int) = %d.\n", sizeof(new int));
    printf(SZ "value of (new int) = 0x%016llx.\n", (long long)new int);
    char *newcp = new char[2];
    printf(SZ "sizeof(newcp) = %d.\n", sizeof(newcp));
    printf(SZ "value of newcp = 0x%016llx.\n", (long long)newcp);
    printf(SZ "sizeof(&newcp[1]) = %d.\n", sizeof(&newcp[1]));
    printf(SZ "value of &newcp[1] = 0x%016llx.\n", (long long)&newcp[1]);
    }
    printf("\n" MACH CMD ":\n");
    {
    #pragma __required_pointer_size 32
    #undef SZ
    #define SZ "  #pragma 32: "
    int i;
    printf(SZ "sizeof(&i) = %d.\n", sizeof(&i));
    printf(SZ "value of &i = 0x%016llx.\n", (long long)&i);
    printf(SZ "sizeof(&\"str\"[1]) = %d.\n", sizeof(&"str"[1]));
    printf(SZ "value of &\"str\"[1] = 0x%016llx.\n", (long long)&"str"[1]);

    const char array[] = "str";
    printf(SZ "sizeof(&array[1]) = %d.\n", sizeof(&array[1]));
    printf(SZ "value of &array[1] = 0x%016llx.\n", (long long)&array[1]);

    printf(SZ "sizeof(new int) = %d.\n", sizeof(new int));
    printf(SZ "value of (new int) = 0x%016llx.\n", (long long)new int);
    char *newcp = new char[2];
    printf(SZ "sizeof(newcp) = %d.\n", sizeof(newcp));
    printf(SZ "value of newcp = 0x%016llx.\n", (long long)newcp);
    printf(SZ "sizeof(&newcp[1]) = %d.\n", sizeof(&newcp[1]));
    printf(SZ "value of &newcp[1] = 0x%016llx.\n", (long long)&newcp[1]);
    }
}
```

Output on Integrity servers with /POINTER=32

Note that all of the pointer sizes, except for the explicitly declared 64-bit pointer variable, and the address of an array element access made through that pointer variable, reflect the command-line setting:

```
$ pipe cxx/point=32 pointers ; cxxlink pointers ; run pointers
I64  /POINT=32:
  #pragma 64: sizeof(&i) = 4.
  #pragma 64: value of &i = 0x000000007acffb28.
  #pragma 64: sizeof(&"str"[1]) = 4.
  #pragma 64: value of &"str"[1] = 0x0000000000040001.
  #pragma 64: sizeof(&array[1]) = 4.
  #pragma 64: value of &array[1] = 0x000000007acffb11.
  #pragma 64: sizeof(new int) = 4.
  #pragma 64: value of (new int) = 0x00000000001e0b70.
  #pragma 64: sizeof(newcp) = 8.
  #pragma 64: value of newcp = 0x00000000001e0b50.
  #pragma 64: sizeof(&newcp[1]) = 8.
  #pragma 64: value of &newcp[1] = 0x00000000001e0b51.

I64  /POINT=32:
  #pragma 32: sizeof(&i) = 4.
  #pragma 32: value of &i = 0x000000007acffb30.
  #pragma 32: sizeof(&"str"[1]) = 4.
  #pragma 32: value of &"str"[1] = 0x0000000000040001.
  #pragma 32: sizeof(&array[1]) = 4.
  #pragma 32: value of &array[1] = 0x000000007acffb21.
  #pragma 32: sizeof(new int) = 4.
  #pragma 32: value of (new int) = 0x00000000001e3690.
  #pragma 32: sizeof(newcp) = 4.
  #pragma 32: value of newcp = 0x00000000001e3670.
  #pragma 32: sizeof(&newcp[1]) = 4.
  #pragma 32: value of &newcp[1] = 0x00000000001e3671.
```

Output on Alpha with /POINTER=32

Note that all of the pointer sizes, except for the result type of the "new" operator, reflect the pragma setting:

```
$ pipe cxx/point=32 pointers ; cxxlink pointers ; run pointers
Alpha/POINT=32:
  #pragma 64: sizeof(&i) = 8.
  #pragma 64: value of &i = 0x000000007ad8f9e0.
  #pragma 64: sizeof(&"str"[1]) = 8.
  #pragma 64: value of &"str"[1] = 0x00000000000145e1.
  #pragma 64: sizeof(&array[1]) = 8.
  #pragma 64: value of &array[1] = 0x000000007ad8f9d9.
  #pragma 64: sizeof(new int) = 4.
  #pragma 64: value of (new int) = 0x000000000007f310.
  #pragma 64: sizeof(newcp) = 8.
  #pragma 64: value of newcp = 0x0000000000083350.
  #pragma 64: sizeof(&newcp[1]) = 8.
  #pragma 64: value of &newcp[1] = 0x0000000000083351.
```

```
Alpha/POINT=32:
  #pragma 32: sizeof(&i) = 4.
  #pragma 32: value of &i = 0x000000007ad8f9d0.
  #pragma 32: sizeof(&"str"[1]) = 4.
  #pragma 32: value of &"str"[1] = 0x00000000000145e1.
  #pragma 32: sizeof(&array[1]) = 4.
  #pragma 32: value of &array[1] = 0x000000007ad8f9c9.
  #pragma 32: sizeof(new int) = 4.
  #pragma 32: value of (new int) = 0x0000000000083cb0.
  #pragma 32: sizeof(newcp) = 4.
  #pragma 32: value of newcp = 0x0000000000083cc0.
  #pragma 32: sizeof(&newcp[1]) = 4.
  #pragma 32: value of &newcp[1] = 0x0000000000083cc1.
```

Output on Integrity servers with /POINTER=64

Note that all of the pointer sizes, except the explicitly declared 32-bit pointer variable, and the address of an array element access made through that pointer variable, reflect the command-line setting. The warning message identifies a real problem where the 64-bit pointer produced by "new" does not fit into the 32-bit pointer variable *newcp*, and the value of *newcp* reflects this by being sign-extended:

```
$ pipe cxx/point=64 pointers ; cxxlink pointers ; run pointers

    char *newcp = new char[2];
.................^
%CXX-W-MAYLOSEDATA, cast from long pointer to short pointer will lose data.
at line number 54 in file DISK$:[DIR]POINTERS.CXX;1
%ILINK-W-COMPWARN, compilation warnings
        module: POINTERS
        file: DISK$:[DIR]POINTERS.OBJ;1
I64  /POINT=64:
  #pragma 64: sizeof(&i) = 8.
  #pragma 64: value of &i = 0x000000007acffb28.
  #pragma 64: sizeof(&"str"[1]) = 8.
  #pragma 64: value of &"str"[1] = 0x0000000000040001.
  #pragma 64: sizeof(&array[1]) = 8.
  #pragma 64: value of &array[1] = 0x000000007acffb11.
  #pragma 64: sizeof(new int) = 8.
  #pragma 64: value of (new int) = 0x000000008009c010.
  #pragma 64: sizeof(newcp) = 8.
  #pragma 64: value of newcp = 0x000000008009c030.
  #pragma 64: sizeof(&newcp[1]) = 8.
  #pragma 64: value of &newcp[1] = 0x000000008009c031.

I64  /POINT=64:
  #pragma 32: sizeof(&i) = 8.
  #pragma 32: value of &i = 0x000000007acffb30.
  #pragma 32: sizeof(&"str"[1]) = 8.
  #pragma 32: value of &"str"[1] = 0x0000000000040001.
  #pragma 32: sizeof(&array[1]) = 8.
  #pragma 32: value of &array[1] = 0x000000007acffb21.
  #pragma 32: sizeof(new int) = 8.
  #pragma 32: value of (new int) = 0x000000008009c050.
  #pragma 32: sizeof(newcp) = 4.
  #pragma 32: value of newcp = 0xffffffff8009c070.
  #pragma 32: sizeof(&newcp[1]) = 4.
  #pragma 32: value of &newcp[1] = 0xffffffff8009c071.
```

Output on Alpha with /POINTER=64

Note that all of the pointer sizes reflect the pragma setting, except for the result type of the "new" operator. The warning message identifies the same problem identified on Integrity servers, and the value of *newcp* similarly reflects this by being sign-extended:

```
$ pipe cxx/point=64 pointers ; cxxlink pointers ; run pointers

    char *newcp = new char[2];
.................^
%CXX-W-MAYLOSEDATA, cast from long pointer to short pointer will lose data.
at line number 54 in file DISK$:[DIR]POINTERS.CXX;1
%LINK-W-WRNERS, compilation warnings
        in module POINTERS file DISK$:[DIR]POINTERS.OBJ;1
Alpha/POINT=64:
  #pragma 64: sizeof(&i) = 8.
  #pragma 64: value of &i = 0x000000007ad8f9e0.
  #pragma 64: sizeof(&"str"[1]) = 8.
  #pragma 64: value of &"str"[1] = 0x00000000000145e1.
  #pragma 64: sizeof(&array[1]) = 8.
  #pragma 64: value of &array[1] = 0x000000007ad8f9d9.
  #pragma 64: sizeof(new int) = 8.
  #pragma 64: value of (new int) = 0x0000000080000010.
  #pragma 64: sizeof(newcp) = 8.
  #pragma 64: value of newcp = 0x0000000080000030.
  #pragma 64: sizeof(&newcp[1]) = 8.
  #pragma 64: value of &newcp[1] = 0x0000000080000031.

Alpha/POINT=64:
  #pragma 32: sizeof(&i) = 4.
  #pragma 32: value of &i = 0x000000007ad8f9d0.
  #pragma 32: sizeof(&"str"[1]) = 4.
  #pragma 32: value of &"str"[1] = 0x00000000000145e1.
  #pragma 32: sizeof(&array[1]) = 4.
  #pragma 32: value of &array[1] = 0x000000007ad8f9c9.
  #pragma 32: sizeof(new int) = 8.
  #pragma 32: value of (new int) = 0x0000000080000050.
  #pragma 32: sizeof(newcp) = 4.
  #pragma 32: value of newcp = 0xffffffff80000070.
  #pragma 32: sizeof(&newcp[1]) = 4.
  #pragma 32: value of &newcp[1] = 0xffffffff80000071.
```

7.3.2 Mixed Pointer-Size Allocators

Mixed pointer-size allocators are placement-new allocators accepting addr_32 and addr_64 parameters. They are documented in Chapter 9.1.2 Memory Allocators in the *HP C++ User's Guide for OpenVMS Systems*. On both OpenVMS Alpha and Integrity server systems, these allocators are implemented in the `<newext.hxx>` header file.

Note the following differences between mixed pointer-size allocators on OpenVMS Alpha and Integrity server systems:

- On Alpha systems, addr_32_space and addr_64_space enums are in the global namespace.

  On Integrity server systems, they are in the namespace __deccxx. The `<newext.hxx>` header has the 'using namespace __deccxx;' directive; so, in general, there is no need to specify a fully qualified name, and the code from an Alpha system can be compiled on an Integrity server system without any changes.

  However, if an ambiguity arises, a fully qualified name can be specified on an Interity servers system: new(__deccxx::addr_32_space) or new(__deccxx::addr_64_space).

- On Integrity server systems, the new(addr_64_space) allocator can be used only in compilations with /POINTER=LONG, where it returns a long pointer to memory allocated in 64-bit space.

  On Integrity server systems in compilations with /POINTER=SHORT, this allocator returns a NULL pointer, and the compiler issues the ADDR64NOT diagnostics. See the example below. Also, on Integrity server systems in compilations with /POINTER=SHORT, the size of the pointer (having a zero value) returned by this allocator is 4.

  On Alpha systems, the new(addr_64_space) allocator returns a long pointer to memory allocated in 64-bit space in compilations with both /POINTER=SHORT and /POINTER=LONG.

  On Integrity server systems, the new(addr_64_space) allocator is retained only to allow code compiled on Alpha systems with /POINTER=LONG to be compiled on Integrity servers with /POINTER=LONG without any changes.

`x.cxx` below demonstrates the difference in behavior of the new(addr_64_space) allocator on Integrity server and Alpha systems. Note that the behavior of the new(addr_32_space) allocator is the same on both platforms.

```
x.cxx
-----
#include <newext.hxx>
#include <stdio.h>

main() {
  __char_ptr32 x;
  __char_ptr64 y;
  x = new (addr_32) char;
  y = new (addr_64) char;
  printf("x = %llx,  y = %llx\n", x, y);
  printf("sizeof(new(addr_32)) = %d, sizeof(new(addr_64)) = %d\n",
          sizeof(new(addr_32) char), sizeof(new(addr_64) char));
}

The output on Alpha system:
---------------------------
$ pipe cxx/pointer=short x.cxx ; cxxlink x.obj ; run x.exe
x = 78690 y = 80000010
sizeof(new(addr_32)) = 4 sizeof(new(addr_64)) = 8
$ pipe cxx/pointer=long x.cxx ; cxxlink x.obj ; run x.exe
x = 78690 y = 80000010
sizeof(new(addr_32)) = 4 sizeof(new(addr_64)) = 8
$

The output on Integrity servers system:
---------------------------
$ pipe cxx/pointer=long x.cxx ; cxxlink x.obj ; run x.exe
x = 1f48e0 y = 8009c010
sizeof(new(addr_32)) = 4 sizeof(new(addr_64)) = 8
$ pipe cxx/pointer=short x.cxx ; cxxlink x.obj ; run x.exe

  y = new (addr_64) char;
...........^
%CXX-W-ADDR64NOT, Use of std::addr_64 in placement new requires
          /POINTER_SIZE=LONG on this platform.

         sizeof(new (addr_32) char), sizeof(new (addr_64) char));
.............................................^
%CXX-W-ADDR64NOT, Use of std::addr_64 in placement new requires
          /POINTER_SIZE=LONG on this platform.
x = 1f88d0 y = 0
sizeof(new(addr_32)) = 4 sizeof(new(addr_64)) = 4
$
```

7.4 Other Enhancements, Changes, and Problems Corrected

- Variadic macros are now supported. This feature allows macros to take a variable number of arguments. It was added to Version 6.4 of the HP C Compiler and is supported by a number other C and C++ compilers. This feature is available only when the value of the /STANDARD qualifier is RELAXED (the default), MS, or GNU.
- This version of the C++ compiler contains support for generation of a new section type in the object file that maps mangled names to their original unmangled form. Future versions of the linker will take advantage of this feature by using the demangled spelling of an identifier name for its error messages. In addition, the linker will be able to generate a new section in the linker map that shows mangled names and their corresponding unmangled orginal name.
- Prologue and epilogue file header processing is now supported in HP C++.
- The `__FUNCTION__` identifier is added. `__FUNCTION__` is a predefined pointer to `char` defined by the compiler, which points to the name of the function as it appears in the source program. `__FUNCTION__` is same as `__func__` of C99.
- Previously, the propagation of a C++ exception out of a thread's start routine did not result in cxxl$terminate( ) being called. A solution for the problem is available on OpenVMS Integrity servers Version V8.2-1 and higher. For V8.2-1, it requires pthreads library patch VMS821I_PTHREAD-V0300. For V8.3, it requires pthreads library patch VMS83I_PTHREAD-V0100.
- 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).
- 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.
- The /TEMPLATE_DEFINE qualifier now requires an option.
- `#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 to 31 characters, or less if the 31st character is within a Universal Character Name.
  - and /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 upper case 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.

- To use the LIB$INITIALIZE feature explicitly in either C or C++, a compilation should contain a reference to a parameterless void function named LIB$INITIALIZE, and provide a statically-initialized list of 32-bit pointers to the functions to be called in a psect named LIB$INITIALIZE with appropriate attributes. The following sample source code shows how this can be done. For simplicity in using both languages, this example gives the initialization functions extern "C" linkage.

```
/* Example to set up LIB$INITIALIZE usage by creating a reference
** to the LIB$INITIALIZE function, and an initialized list of
** functions to be called in the LIB$INITIALIZE psect.
*/

#ifdef __cplusplus
extern "C" {
#endif

/* Declarations for initialization functions. */
extern void some_init_function(void);
extern void some_other_init_function(void);
/* etc, e.g. other declarations might come from header files */

/* Use 32-bit pointers */
#if __INITIAL_POINTER_SIZE
#pragma pointer_size save
#pragma pointer_size 32
#endif

/* Create a reference to the LIB$INITIALIZE function. */
extern void LIB$INITIALIZE(void);
extern void (*unused_global_variable_1)(void) = LIB$INITIALIZE;

/* Create an array of pointers to the init functions in the special
** LIB$INITIALIZE section.
*/
#pragma extern_model save
#pragma extern_model strict_refdef "LIB$INITIALIZE" gbl,noexe,nowrt,noshr,long
extern void (* const unused_global_variable_2[])() =
{
   some_init_function
  , some_other_init_function
  /* etc, other functions to be called by LIB$INITIALIZE() */
};
#pragma extern_model restore

#if __INITIAL_POINTER_SIZE
#pragma pointer_size restore
#endif

#ifdef __cplusplus
}
#endif
/* End of example to set up LIB$INITIALIZE */
```

```
/* Begin executable test of LIB$INITIALIZE setup. */
#ifdef __cplusplus
extern "C" {
#endif
extern int printf(const char *, ...);
extern void some_init_function(void) {
  printf("In some_init_function.\n");
}
extern void some_other_init_function(void) {
  printf("In some_other_init_function.\n");
}
#ifdef __cplusplus
}
#endif

void main(void) {
    printf("In main.\n");
}

/* Compile with either C or C++ on Alpha or I64, link and run.
** The output is:
**    In some_init_function.
**    In some_other_init_function.
**    In main.
*/
```

7.5 Known Problems and Restrictions

- On Integrity server systems, the `#pragma inline` or `#pragma noinline` directives are not supported.
- On Integrity server systems, the `#pragma function` and `#pragma intrinsic` directives are ignored.
- On Integrity server systems, the intrinsic bit-counting functions `_leadz()`, `_trailz()`, `_popcnt()`, and `_poppar()` declared at the end of `<builtins.h>` are treated as intrinsic whether or not the header is included. Since these function names begin with an underscore followed by a lowercase letter, under the language standards they are reserved to the implementation for use as identifiers with external linkage. So programs that declare their own functions with any of these names are not standard-conforming. However, the Alpha C++ compiler and the C compiler for both Alpha and Integrity servers support user-written functions with these names as long as `<builtins.h>` is not included; or if it is included and followed by `#pragma function` directives for these names. The Integrity servers C++ compiler will accept user-defined functions with these names, but calls to them will generally be treated as having the intrinsic behavior, which may produce unpredictable results if the user declaration does not match the intrinsic declaration.
- ANAL/OBJ on OpenVMS 8.2-1 and earlier will issue an error message about an unknown section type that is now generated by this version of the compiler:

  ```
  %ANALYZE-E-ELF_UNKNWNSEC, Unrecognized Elf Section Type 60000007
  ```

  Please ignore this message. This section is ignored by current versions of the linker and ANAL/IMAGE and causes no harm.
- Debugging with optimized code is not supported.
- 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.

8 Release Notes for the V7.2 C++ Standard Library

This section describes enhancements, changes, restrictions and problems corrected for the V7.2 C++ Standard Library.

- 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.
- The C++ Standard Library headers have been modified to allow include-once compiler optimization. This reduces compilation time and the size of the listing file.
- The std::numeric_limits.round_error( ) function has been corrected to return a value corresponding to the dynamic rounding mode in effect for the program. In particular, to determine the current dynamic rounding mode, the std::numeric_limits.round_error( ) function now calls C Run-Time Library function read_rnd( ).

- 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 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;
}
```

- 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.

9 Release Notes for the V7.1 C++ Compiler

This section describes the new features, differences, and restrictions of the C++ V7.1 compiler for Integrity servers system over the C++ V6.5 compiler for Alpha systems. See Section 10 for the release notes for the standard library, language run-time support library, and class library.

This release of the compiler uses a new technology base that differs substantially from both HP C++ for OpenVMS Alpha and HP C for OpenVMS Integrity servers . Although a great deal of work has been done to make it highly compatible with HP C++ for OpenVMS Alpha, there are a number of differences that you will likely notice. Some of these differences are temporary, some are changes that will be reflected in the next version of the compiler for Alpha systems, and some are permanent. Among the permanent differences are:

- Resource requirements

  Programs will usually use more memory both at compile time and at run time. See Section 9.3.1.

- Floating-point behaviors

  The default is /FLOAT=IEEE/IEEE_MODE=DENORM_RESULTS. Consistent use of qualifiers across compilations is required. See Section 9.3.5.

- Simplified instantiation without repository. See Section 9.3.9.

- No inline assembly language. See Section 9.3.6.

- Removal of the CFRONT dialect (which will also be removed in the next release of the C++ Alpha compiler).

- String literal type change.

  For standards-compliance and link compatiblity between compiler dialects, ordinary string literals now have the type "array of const char" in all compiler dialects on Integrity servers system and on Alpha systems when compiling in /MODEL=ANSI mode.

  When compiling in /MODEL=ARM mode on Alpha systems, string literals are of type "array of char" in all compiler dialects.

9.1 Problems Fixed in V7.1

A problem has been corrected when using the common_block extern_model. A temporary global symbol is no longer emitted when generating the data.

9.2 New Features in V7.1

The following new features and changes have been made since C++ Version 6.5 for OpenVMS Alpha systems.

9.2.1 cname Header Support

The C++ compiler implements section 17.4.1.2 - Headers [lib.headers] "C++ Headers for C Library Facilities" of the C++ Standard. See also Stroustrup's *The C++ Programming Language, 3rd Edition* sections 9.2.2 and 16.1.2.

The implementation consists of 18 <cname> headers defined in the C++ Standard:

```
<cassert> <cctype>  <cerrno>  <cfloat>
<ciso646> <climits> <clocale> <cmath>
<csetjmp> <csignal> <cstdarg> <cstddef>
<cstdio>  <cstdlib> <cstring> <ctime>
<cwchar>  <cwctype>
```

As required by the C++ standard, the <cname> headers define C names in the std namespace. In /NOPURE_CNAME mode, the names are also inserted into the global namespace. See the description of the /[NO]PURE_CNAME compiler qualifier.

The <cname> headers are located in the same TLB library that contains the C++ standard and class library headers: SYS$SHARE:CXXL$ANSI_DEF.TLB.

9.2.2 __HIDE_FORBIDDEN_NAMES Predefined in Strict ANSI Mode

When compiling in /STANDARD=STRICT_ANSI mode, the compiler predefines the __HIDE_FORBIDDEN_NAMES macro, causing the C headers to expose only those symbols that are defined by the ANSI C Standard 89. While this is a change in behavior between C++ V6.5 for OpenVMS Alpha systems and C++ for Integrity servers system, the new behavior is consistent with the behavior of the C compiler in /STANDARD=ANSI89 mode.

As a result of this change, the following program would not compile on an Integrity servers system in /STANDARD=STRICT_ANSI mode (note that fdopen is not part of the ANSI C Standard 89).

```
#include <stdio.h>
void foo() {
  fdopen(0,0);
}
```

9.2.3 /[NO]FIRST_INCLUDE Qualifier Added

The /[NO]FIRST_INCLUDE qualifier is added. It has the following format:

/[NO]FIRST_INCLUDE=(file[, . . .])

This qualifier includes the specified files before any source files. It corresponds to the *Tru64 UNIX* -FI switch.

When /FIRST_INCLUDE=*file* is specified, *file* is included in the source as if the line before the first line of the source was:

```
#include "file"
```

If more than one file is specified, the files are included in their order of appearance on the command line.

This qualifier is useful if you have command lines to pass to the C compiler that are exceeding the DCL command-line length limit. Using the /FIRST_INCLUDE qualifier can help solve this problem by replacing lengthy /DEFINE and /WARNINGS qualifiers with `#define` and `#pragma message` preprocessor directives placed in a /FIRST_INCLUDE file.

The default is /NOFIRST_INCLUDE.

9.2.4 #pragma include_directory Added

The effect of each `#pragma include_directory` is as if its string argument (including the quotes) were appended to the list of places to search that is given its initial value by the /INCLUDE_DIRECTORY qualifier, except that an empty string is not permitted in the pragma form.

The `#pragma include_directory` directive has the following format:

#pragma include_directory <*string-literal*>

This pragma is intended to ease DCL command-line length limitations when porting applications from POSIX-like environments built with makefiles containing long lists of -I options that specify directories to search for headers. Just as long lists of macro definitions specified by the /DEFINE qualifier can be converted to `#define` directives in a source file, long lists of places to search specified by the /INCLUDE_DIRECTORY qualifier can be converted to `#pragma include_directory` directives in a source file.

Note that the places to search, as described in the help text for the /INCLUDE_DIRECTORY qualifier, include the use of POSIX-style pathnames, for example `"/usr/base"`. This form can be very useful when compiling code that contains POSIX-style relative pathnames in `#include` directives. For example, `#include <subdir/foo.h>` can be combined with a place to search such as `"/usr/base"` to form `"/usr/base/subdir/foo.h"`, which will be translated to the filespec "USR:[BASE.SUBDIR]FOO.H"

This pragma can appear only in the main source file or in the first file specified on the /FIRST_INCLUDE qualifier. Also, it must appear before any `#include` directives.

9.2.5 Messages

There have been some changes in the /WARNINGS qualifier. These include bug fixes and improved compatibility with the C compiler. Some changes that might affect user compilations are:

- The /WARNINGS=ENABLE=ALL qualifier now enables all compiler messages including informational-level messages.
- The /WARNINGS=INFORMATIONALS qualifier continues to enable most informationals, but we recommend that /WARNINGS=ENABLE=ALL be used instead
- Using /WARNINGS=INFORMATIONALS=<tag> no longer enables all other informational messages.

The move from Alpha systems to Integrity servers system will cause some minor differences in certain compiler diagnostics that are signaled from the code generator. As a result, diagnostics for unreachable code and fetches of uninitialized variables might be different on the two platforms. In addition to a change in message text, some conditions detected on one platform might not be detected on the other.

9.2.6 New Front End

A new C++ front end provides improved conformance to the C++ International Standard.

9.3 Integrity servers Differences

This section describes differences between the C++ compiler on Integrity servers system and Alpha systems.

9.3.1 Quotas

The C++ compiler for Integrity servers system is built from a different code base than the C++ compiler for Alpha systems, and that code base is larger than the code base for Alpha. Also, Integrity servers images tend to be somewhat larger than Alpha images in general. Image size mostly affects working-set size and the amount of pagefile quota needed to execute an image without exhausting virtual memory. If you find that programs that compile and run successfully on Alpha run out of memory on Integrity servers system (either during compilation or when run), you probably need to increase your pagefile quota. There are no specific guidelines at this time. You might start by doubling the quota that was sufficient on Alpha, and then use a "binary-search" approach to arrive at a better quota value for Integrity servers system (doubling again, or halving the increment, until your biggest programs and compilations have just enough memory, and then adding an appropriate safety margin).

9.3.2 Dialect Changes

The following dialect changes have been made:

- Support for /STANDARD=CFRONT has been retired.
- Some of the compiler dialects (options to the /STANDARD qualifier) have been updated to reflect the most recent behaviors of the compilers that the dialect is attempting to match. Other changes involve the removal of less significant or undesirable compatibility features. If a dialect has changed in a way that impacts you significantly, report it as described in Section 13.

9.3.3 ABI/Object Model changes

The object model and the name mangling scheme used by the C++ compiler on Integrity servers system are different from those used on Alpha systems (different from both /MODEL=ARM and /MODEL=ANSI). The Integrity servers compiler uses the interface described by the Integrity servers Application Binary Interface (ABI).

See http://www.codesourcery.com/cxx-abi/abi.html for a draft description of the ABI specification.

The compiler has some additional encoding rules that are applied to symbol names after the ABI name mangling is determined. All symbols with C++ linkage have CRC encodings added to the name, are uppercased, and shorten to 31 characters if necessary. Since the CRC is computed before the name is uppercased, the symbol name is case sensitive even though the final name is uppercase. /names=as_is and /names=upper are not applicable to these symbols.

All symbols without C++ linkage will have CRC encodings added if they are longer then 31 characters and /names=shorten is specified. Global variables with C++ linkage are treated as if they have non-C++ linkage for compatibility with C and older compilers.

9.3.4 Command-Line Qualifiers

This section describes qualifier differences for HP C++ on Integrity servers systems.

Qualifiers/Features Not Supported on Integrity servers system

The following command-line qualifiers and features are not supported on C++ for Integrity servers system, and are diagnosed by default because ignoring them is likely to alter program behavior:

- Comma lists are not supported. Their use provokes a fatal error.
- The /INSTRUCTION_SET=[NO]FLOATING_POINT qualifier is not available on Integrity servers system.
- /L_DOUBLE_SIZE=64 is not available on Integrity servers system. If it is specified, a warning message is issued, and /L_DOUBLE_SIZE=128 is used.
- /POINTER_SIZE=(LONG,64) is ignored.

Changed/Ignored Qualifiers

A number of other qualifiers not supported on Integrity servers system are, by default, silently ignored by the compiler. These qualifiers fall into two groups:

- Qualifiers that should not alter the behavior of a correct program so, if ignored, should have no visible effect. Qualifiers that enable optimizations typically have this characteristic.
- Qualifiers that might affect program behavior but, if ignored, produce no significant change in the vast majority of programs. Examples of qualifiers in this category are /NORTTI (the run-time information is always generated) and /MODEL=ARM (the ANSI model is functionally superior, and binary compatibility with existing object code is not an issue for the OpenVMS Integrity servers platform).

Two optional compiler messages can be enabled to diagnose most of these cases:

- The QUALNA message diagnoses uses of the first group.
- The QUALCHANGE message diagnoses uses of the second group.

If you encounter porting problems, compile /WARN=ENABLE=QUALCHANGE to determine if a qualifier change might be affecting your application.

If you wish to clean up your build scripts to remove extraneous qualifiers that are not meaningful on Integrity servers system, you can enable the QUALNA message.

A list of these qualifiers follows:

- /ARCHITECTURE=*option*

  An additional keyword has been added: ITANIUM2.

  If an Alpha keyword (EV4, EV5, EV56, PCA56, EV6, EV68, EV7) is specified for *option*, it is ignored.

- /ASSUME

  The following /ASSUME options are ignored on Integrity servers system and should not cause any behavior changes:

  NORTTI_CTORVTBLS
  NOPOINTERS_TO_GLOBALS
  TRUSTED_SHORT_ALIGNMENT

WHOLE_PROGRAM

- /CHECK=UNINITIALIZED_VARIABLES

  This qualifier has no effect in this version of the compiler.

- /DISTINGUISH_NESTED_ENUMS

  This qualifier only modified the behavior of programs compiled with /MODEL=ARM. Since that model is not supported on the Integrity servers platform, this qualifier is meaningless.

- /EXCEPTIONS=NOCLEANUP

  The NOCLEANUP keyword for the /EXCEPTIONS qualifier is ignored.

- /EXCEPTIONS=IMPLICIT

  The IMPLICIT keyword for the /EXCEPTIONS qualifier is ignored.

- /FLOAT

  The default for /FLOAT on OpenVMS Integrity servers system is IEEE_FLOAT.

  See Section 9.3.5 for more information about floating-point behavior on Integrity servers system.

- /IEEE_MODE

  The default for /IEEE_MODE on Integrity servers system is DENORM_RESULTS, which generates infinities, denorms, and NaNs without exceptions.

  On OpenVMS Alpha systems, the default for /IEEE_MODE when using /FLOAT=IEEE_FLOAT is FAST, which causes a FATAL error for exceptional conditions such as divide-by-zero and overflow.

  See Section 9.3.5 for more information.

- /MACHINE_CODE

  The /MACHINE_CODE qualifier output will appear in an .S file in the same directory as your listing file.

- The /MODEL=ARM qualifier is treated the same as the default /MODEL=ANSI (except for the optional QUALCHANGE diagnostic).

- /OPTIMIZE

  There are several changes to the /OPTIMIZE qualifier:

  - On Integrity servers system, for /OPTIMIZE=INLINE, the keywords AUTOMATIC and SPEED do the same thing.

    Also, the ALL keyword does not necessarily result in every possible call being inlined, as it does on Alpha systems.

  - The /OPTIMIZE=TUNE qualifier takes a new keyword: ITANIUM2, which is the default at this time. If you specify an Alpha keyword, it is ignored.

  - The /OPTIMIZE=UNROLL=*n* qualifier on Integrity servers system does not have the ability to control the specific number of times a loop is unrolled. The only accepted values are /OPTIMIZE=UNROLL=1 which disables loop unrolling, and /OPTIMIZE=UNROLL=0 which allows the compiler's optimizer to decide how the loop should be unrolled. The default is /OPTIMIZE=UNROLL=0.

  - /OPTIMIZE=LIMIT_INLINE is ignored.

- /PREFIX_LIBRARY_ENTRIES

  Note that /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.

- /TEMPLATE

  See Section 9.3.9 for information on template instantiation.

- /POINTER_SIZE=(SHORT,32) is ignored. Mixed pointer types are not supported at this time.

- /SHOW=STATISTICS

  The /SHOW=STATISTICS qualifier is ignored at this time.

- /STANDARD=CFRONT

  The /STANDARD=CFRONT qualifier is no longer available. If it is specified, the compiler issues a warning message and uses the default dialect, /STANDARD=ANSI.

New Qualifiers

The following command-line qualifier is new for C++ V7.1 for Integrity servers system:

- /[NO]PURE_CNAME

  Affects insertion of the names into the global namespace by `<cname>` headers.

  In /PURE_CNAME mode, the `<cname>` headers insert the names into the `std` namespace only, as defined by the C++ Standard, and the __PURE_CNAME macro is predefined by the compiler.

  In /NOPURE_CNAME mode, the `<cname>` headers insert the name into the `std` namespace and also into the global namespace.

  The default depends on the standard mode:

  - In /STANDARD=STRICT_ANSI mode, the default is /PURE_CNAME.
  - In all other standard modes, the default is /NOPURE_CNAME.

  Inclusion of a `<name.h>` header instead of its `<cname.h>` counterpart (for example, `<stdio.h>` instead of `<cstdio>`) results in inserting names defined in the header into both the `std` namespace and the global namespace. Effectively, this is the same as the inclusion of a `<cname>` header in the /NOPURE_CNAME mode.

  See Section 9.2.1 for more information.

9.3.5 Floating Point

This section describes floating-point behavior on Integrity servers system.

IEEE Now the Default

On OpenVMS Integrity servers system, /FLOAT=IEEE_FLOAT is the default floating-point representation. IEEE format data is assumed and IEEE floating-point instructions are used. There is no hardware support for floating-point representations other than IEEE, although you can specify the /FLOAT=D_FLOAT or /FLOAT=G_FLOAT compiler option.

These VAX floating-point formats are supported in the Integrity servers compiler by generating run-time code that converts VAX floating-point formats to IEEE format to perform arithmetic operations, and then converts the IEEE result back to the appropriate VAX floating-point format. This imposes additional run-time overhead and some loss of accuracy compared to performing the operations in hardware on Alpha and VAX systems. The software support for the VAX formats is provided to meet an important functional compatibility requirement for certain applications that need to deal with on-disk binary floating-point data.

On Integrity servers system, the default for /IEEE_MODE is DENORM_ RESULTS, which is a change from the default of /IEEE_MODE=FAST on Alpha systems. This means that by default, floating-point operations may silently generate values that print as Infinity or Nan (the industry-standard behavior), instead of issuing a fatal run-time error as they would when using VAX floating-point format or /IEEE_MODE=FAST. Also, the smallest-magnitude nonzero value in this mode is much smaller because results are allowed to enter the denormal range instead of being flushed to zero as soon as the value is too small to represent with normalization.

The conversion between VAX floating-point formats and IEEE formats on the Intel Itanium architecture is a transparent process that will not impact most applications. All you need to do is recompile your application. Because IEEE floating-point format is the default, unless your build explicitly specifies VAX floating-point format options, a simple rebuild for Integrity servers system will use the native IEEE formats directly. For the large class of programs that do not directly depend on the VAX formats for correct operation, this is the most desirable way to build for Integrity servers system.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on an Integrity servers system, the compiler automatically generates code for converting floating-point formats. Whenever the application performs a sequence of arithmetic operations, this code does the following:

1. Converts VAX floating-point formats to either IEEE single or IEEE double floating-point formats.
2. Performs arithmetic operations in IEEE floating-point arithmetic.
3. Converts the resulting data from IEEE formats back to VAX formats.

Where no arithmetic operations are performed (VAX float fetches followed by stores), no conversion will occur. The code handles such situations as moves.

VAX floating-point formats have the same number of bits and precision as their equivalent IEEE floating-point formats. For most applications, the conversion process will be transparent and, therefore, a non-issue.

In a few cases, arithmetic calculations might have different results because of the following differences between VAX and IEEE formats:

- Values of numbers represented
- Rounding rules
- Exception behavior

These differences might cause problems for applications that do any of the following:

- Depend on exception behavior
- Measure the limits of floating-point behaviors

- Implement algorithms at maximal processor-specific accuracy
- Perform low-level emulations of other floating-point processors
- Use direct equality comparisons between floating-point values, instead of appropriately ranged comparisons (a practice that is extremely vulnerable to changes in compiler version or compiler options, as well as architecture)

You can test an application's behavior with IEEE floating-point values by first compiling it on an OpenVMS Alpha system using /FLOAT=IEEE_FLOAT/IEEE_MODE=DENORM.

If that produces acceptable results, then simply build the application on the OpenVMS Integrity servers system using the same qualifier.

If you determine that simply recompiling with an /IEEE_MODE qualifier is not sufficient because your application depends on the binary representation of floating-point values, then first try building for your Integrity servers system by specifying the VAX floating-point option that was in effect for your VAX or Alpha build. This causes the representation seen by your code and on disk to remain unchanged, with some additional runtime cost for the conversions generated by the compiler. If this is not an efficient approach for your application, you can convert VAX floating-point binary data in disk files to IEEE floating-point formats before moving the application to an Integrity servers system.

/IEEE_MODE Notes

On Alpha systems, the /IEEE_MODE qualifier generally has its greatest effect on the generated code of a compilation. When calls are made between functions compiled with different /IEEE_MODE qualifiers, each function produces the /IEEE_MODE behavior with which it was compiled.

On Integrity servers system, the /IEEE_MODE qualifier primarily affects only the setting of a hardware register at program startup. In general, the /IEEE_MODE behavior for a given function is controlled by the /IEEE_MODE option specified on the compilation that produced the main program: the startup code for the main program sets the hardware register according the command-line qualifiers used to compile the main program.

When applied to a compilation that does not contain a main program, the /IEEE_MODE qualifier does have some effect: it might affect the evaluation of floating-point constant expressions, and it is used to set the EXCEPTION_MODE used by the math library for calls from that compilation. But the qualifier has no effect on the exceptional behavior of floating-point calculations generated as inline code for that compilation. Therefore, if floating-point exceptional behavior is important to an application, all of its compilations, including the one containing the main program, should be compiled with the same /IEEE_MODE setting.

Even on Alpha systems, the particular setting of /IEEE_MODE=UNDERFLOW_TO_ZERO has the following characteristic: its primary effect requires the setting of a runtime status register, and so it needs to be specified on the compilation containing the main program in order to be effective in other compilations.

Differences in Undefined Behavior

Programs containing undefined floating-point behavior, such as assigning a negative floating-point number to an unsigned integer variable, could generate different results when compiled and run on Integrity servers system. The compiler diagnostics and runtime results can both differ. One runtime difference is the case where one system generates an exception, while the other silently produces a result.

More Information

For more information on Integrity servers floating-point behavior, see the white paper *OpenVMS floating-point arithmetic on the Intel Itanium architecture* at http://www.hp.com/products1/evolution/alpha_retaintrust/download/i64-floating-pt-wp.pdf .

9.3.6 Intrinsics and Builtins

The C++ built-in functions available on OpenVMS Alpha systems are also available on Integrity servers system, with some differences, as described in this section. This section also describes built-in functions that are specific to Integrity servers system.

Builtin Differences on Integrity servers system

The `<builtins.h>` header file contains comments noting which built-in functions are not available or are not the preferred form for Integrity servers system. The compiler issues diagnostics where using a different built-in function for Integrity servers system would be preferable.

Note

The comments in `<builtins.h>` reflect only what is explicitly present in that header file itself, and in the compiler implementation. You should also consult the content and comments in `<pal_builtins.h>` to determine more accurately what functionality is effectively provided by including `<builtins.h>`. For example, if a program explicitly declares one of the Alpha built-in functions and invokes it without having included `<builtins.h>`, the compiler might issue the BIFNOTAVAIL error message, regardless of whether or not the function is available through a system service. If the compilation does include `<builtins.h>`, and BIFNOTAVAIL is issued, then either there is no support at all for the built-in function or a new version of `<pal_builtins.h>` is needed.

Here is a summary of these differences on Integrity servers system:

- There is no support for the `asm`, `fasm`, and `dasm` intrinsics (declared in the `<c_asm.h>` header file).
- The functionality provided by the special-case treatment of R26 in an Alpha system `asm`, as in `asm("MOV R26,R0")`, is provided by a new built-in function for Integrity servers system:

  ```
  __int64 __RETURN_ADDRESS(void);
  ```

  This built-in function produces the address to which the function containing the built-in call will return (the value of R26 on entry to the function on Alpha systems; the value of B0 on entry to the function on Integrity servers system). This built-in function cannot be used within a function specified to use nonstandard linkage.
- The only PAL function calls implemented as built-in functions within the compiler are the 24 queue-manipulation builtins. The queue manipulation builtins generate calls to new OpenVMS system services SYS$<name>, where <name> is the name of the builtin with the leading underscores removed.

Any other OpenVMS PAL calls are supported through macros defined in the <pal_builtins.h> header file included in the <builtins.h> header file. Typically, the macros in <pal_builtins.h> transform an invocation of an Alpha system builtin into a call to a system service that performs the equivalent function on an Integrity servers system. Two notable exceptions are __PAL_GENTRAP and __PAL_BUGCHK, which instead invoke the Integrity servers specific compiler builtin __break2.

- There is no support for the various floating-point built-in functions used by the OpenVMS math library (for example, operations with chopped rounding and conversions).
- Most built-in functions that take a retry count are not supported. This is necessary because the retry behavior allowed by Alpha load-locked/store-conditional sequences does not exist on Integrity servers system. There are two exceptions to this: __LOCK_LONG_RETRY and __ACQUIRE_SEM_LONG_RETRY; in these cases, the retry behavior involves comparisons of data values, not just load-locked/store-conditional.
- The __CMP_STORE_LONG and __CMP_STORE_QUAD built-in functions are not supported. Use the new __CMP_SWAP form instead.

Built-in Functions Specific to Integrity servers system

The <builtins.h> header file contains a section at the top conditionalized to just __ia64 with the support for built-in functions specific to Integrity servers system. This includes macro definitions for all of the registers that can be specified to the __getReg, __setReg, __getIndReg, and __setIndReg built-in functions. Parameters that are const-qualified require an argument that is a compile-time constant.

The following lists the C++ built-in functions available on OpenVMS Integrity servers systems.

```
/* Intel compatible */

unsigned __int64 __getReg(const int __whichReg);

void __setReg(const int __whichReg, unsigned __int64 __value);

unsigned __int64 __getIndReg(const int __whichIndReg,
                             __int64 __index);

void __setIndReg(const int __whichIndReg, __int64 __index,
                 unsigned __int64 __value);
```

```
void __break(const int __break_arg); /* Native I64 arg */
void __dsrlz(void);
void __fc(__int64 __address);
void __fwb(void);
void __invalat(void);
void __invala(void); /* alternate spelling of __invalat */
void __isrlz(void);
void __itcd(__int64 __address);
void __itci(__int64 __address);
void __itrd(__int64 __whichTransReg, __int64 __address);
void __itri(__int64 __whichTransReg, __int64 __address);
void __ptce(__int64 __address);
void __ptcl(__int64 __address, __int64 __pagesz);
void __ptcg(__int64 __address, __int64 __pagesz);
void __ptcga(__int64 __address, __int64 __pagesz);
void __ptri(__int64 __address, __int64 __pagesz);
void __ptrd(__int64 __address, __int64 __pagesz);
void __rsm(const int __mask);
void __rum(const int __mask);
void __ssm(const int __mask);
void __sum(const int __mask);
void __synci(void);
__int64 /*address*/ __thash(__int64 __address);
__int64 /*address*/ __ttag(__int64 __address);

/* Intel _Interlocked intrinsics */

unsigned __int64 _InterlockedCompareExchange_acq(
                        unsigned int *__Destination,
                        unsigned __int64 __Newval,
                        unsigned __int64 __Comparand);

unsigned __int64 _InterlockedCompareExchange64_acq(
                        unsigned __int64 *__Destination,
                        unsigned __int64 __Newval,
                        unsigned __int64 __Comparand);

unsigned __int64 _InterlockedCompareExchange_rel(
                        unsigned int *__Destination,
                        unsigned __int64 __Newval,
                        unsigned __int64 __Comparand);

unsigned __int64 _InterlockedCompareExchange64_rel(
                        unsigned __int64 *__Destination,
                        unsigned __int64 __Newval,
                        unsigned __int64 __Comparand);

/* GEM-added builtins */

void __break2(__Integer_Constant __break_code,
                        unsigned __int64 __r17_value);

void __flushrs(void);
void __loadrs(void);
int __prober(__int64 __address, unsigned int __mode);
int __probew(__int64 __address, unsigned int __mode);
unsigned int __tak(__int64 __address);
__int64 __tpa(__int64 __address);

/* _Interlocked* builtins return the old value and have the
** newval and comparand arguments in a different order than
** __CMP_SWAP* builtins that return the status (1 or 0).
** Forms without trailing _ACQ or _REL are equivalent to
** the _ACQ form. On Alpha, _ACQ generates MB after the swap,
** _REL generates MB before the swap.
*/
```

```
int __CMP_SWAP_LONG(volatile void *__addr,
                    int __comparand,
                    int __newval);

int __CMP_SWAP_QUAD(volatile void *__addr,
                    __int64 __comparand,
                    __int64 __newval);

int __CMP_SWAP_LONG_ACQ(volatile void *__addr,
                    int __comparand,
                    int __newval);

int __CMP_SWAP_QUAD_ACQ(volatile void *__addr,
                    __int64 __comparand,
                    __int64 __newval);

int __CMP_SWAP_LONG_REL(volatile void *__addr,
                    int __comparand,
                    int __newval);

int __CMP_SWAP_QUAD_REL(volatile void *__addr,
                    __int64 __comparand,
                    __int64 __newval);

/*
** Produce the value of R26 (Alpha) or B0 (I64) on entry to the
** function containing a call to this builtin. Cannot be invoked
** from a function with nonstandard linkage.
*/
__int64 __RETURN_ADDRESS(void);
```

9.3.7 memcpy C Run-Time Library Function

Programs must not use `memcpy` with overlapping arguments.

HP C++ for OpenVMS Integrity servers optimizes the use of standard C library functions `memcpy` and `memmove` somewhat differently than HP C for OpenVMS Integrity servers and the HP C and C++ compilers for OpenVMS Alpha. Because of this, the use of `memcpy` with arguments that overlap in memory can produce results that differ from the other compilers.

Note that the behavior of `memcpy` is formally undefined if its arguments overlap; only `memmove` should be used in this case. However, the other compilers generally treat `memcpy` in a way that handles overlapping arguments in the same way as `memmove`, while HP C++ for OpenVMS Integrity servers does not. So when using `memcpy`, take care that the arguments do not refer to memory that might overlap; if overlap is a possibility, you must use `memmove` instead of `memcpy`.

9.3.8 ELF

ELF Used on Integrity servers system.

On OpenVMS Alpha systems, the C++ compiler uses a proprietary object format specific to OpenVMS.

On OpenVMS Integrity servers system, the compiler generates ELF objects. ELF is an industry standard object format used on many UNIX platforms, including Linux. This change should be transparent to most users; it is primarily of interest to compiler and tools developers. The greatest benefit of this change is that it should make it easier to create development tools that work on OpenVMS and other platforms.

Extensions to ELF have been used as needed to provide functionality unique to OpenVMS. See the *Porting Applications from HP OpenVMS Alpha to HP OpenVMS Industry Standard 64 for Integrity Servers* for more information on ELF.

COMDATS/Group Sections

One feature that ELF provides that is new to OpenVMS is the COMDAT section group—a group of sections in an object file that can be duplicated in one or more other object files. The linker is expected to keep one group and ignore all others. The benefit of this feature is that it permits compilers to generate definitions for symbols for things used in multiple objects without having to worry about creating a single definition in one place. The most notable uses for this feature are templates and inline functions.

New ELF Type for Weak Symbols

A new Executable and Linkable Format (ELF) type was generated to distinguish between the two types of weak symbol definitions.

For modules with ABI versions equal to 2 (the most common version used by compilers):

- Type STB_WEAK represents the UNIX-style weak symbol (formerly, the OpenVMS-style weak symbol definition for ABI Version 1 ELF format).
- Type STB_VMS_WEAK represents the OpenVMS-style of weak symbol definition.

The Librarian supports both the ELF ABI versions 1 and 2 of the object and image file formats within the same library.

9.3.9 Templates

This section describes template instantiation for Integrity servers system.

Implemented using ELF COMDATS/Groups Sections

The Alpha C++ compiler had numerous models for instantiating templates. Each attempted to solve the issue of how to generate one and only one copy of each template. The use of ELF on OpenVMS Integrity servers system provided the compiler with the additional option of using COMDAT section groups. Since this technique is superior to all the models supported on Alpha, this is the only model supported on Integrity servers system.

In this model, templates are instantiated in a COMDAT section group inside every object module that uses them. This is very similar to the /TEMPLATE=LOCAL on Alpha systems, except that when the objects are linked together, the linker removes the duplicate copies. The primary advantage of this technique over /TEMPLATE=LOCAL and /TEMPLATE=IMPLICIT_LOCAL is the reduction in image size.

A secondary advantage is the elimination of distinct data for each template. For example, if a template maintained a list of elements it created, each module would have a separate copy of the list. This behavior does not conform to the standard. If you are currently using /TEMPLATE=LOCAL or /TEMPLATE=IMPLICIT_LOCAL, you will likely experience no difficulty from this change.

Not in Repository

The most visible difference that results from this new instantiation model occurs in models that instantiate templates into the repository (/TEMPLATE=AUTOMATIC|ALL_REPOSITORY|USED_REPOSITORY).

With the new model, no repository is needed. Build procedures that use CXXLINK will work transparently. Builds that attempt to manipulate objects in the repository will fail and will need to be changed. In most cases, the reason for manipulating the repository directly has been eliminated with the new template instantiation model.

Restriction

For OpenVMS Integrity servers and Alpha systems, the /TEMPLATE_DEFINE=ALL qualifier is not guaranteed to work with the Standard Library. Use automatic instantiation or specify /TEMPLATE_DEFINE=USED instead.

9.3.10 Exceptions and Condition Handlers

The command-line option /EXCEPTIONS=NOCLEANUP is not implemented. As a result, you might see destructors being called during cleanup in code previously compiled with this option.

Exception specifications are not implemented. Exception specifications on routine declarations and definitions are accepted syntactically, but their run-time behavior has not yet been implemented.

Stack unwinding

According to the C++ Standard, an implementation may or may not unwind the stack before calling terminate when no matching handler is found for a thrown exception. On Integrity servers system, the implementation unwinds the stack. On Alpha systems, it does not.

Consider the following program:

```
#include <exception>
#include <cstdio>
#include <cstdlib>

 class C {
 public:
   C() { std::printf("Created\n"); }
   ~C() { std::printf("Destroyed\n"); }

 };

 void announce1() {
   std::printf("In terminate\n");
   exit(0);
 }

 int main() {
   C c;
   std::set_terminate(announce1);
   throw 5;

   return 0;
 }
```

For the above program, the output on OpenVMS Alpha and Integrity servers system is:

```
Alpha:            I64:

Created           Created
In terminate      Destroyed
                  In terminate
```

Exceptions Not Caught

The compiler assumes that the only two ways an exception can be propagated into a function are:

- From a throw expression, or
- From a routine call that itself can throw an exception.

As a result of this assumption, some exceptions such as those thrown from a signal handler will not be caught.

9.3.11 Overriding new and delete

Full support for allowing a user to override operators `new` and `delete` is not yet completed. As a result, when you override `new` and `delete`, you might see multiply-defined symbols when linking. This will be fixed in a future release.

9.4 Integrity servers Known Issues

Version 7.1 of the C++ compiler has the following known issues:

Qualifiers

By default the compiler's optimizer will not unroll loops. If you wish to turn the optimization on, you must compile /OPTIMIZE=UNROLL=N

Destruction of Initialized Aggregate Members

If an aggregate class/struct (such as struct B in the example below) contains a member with a destructor, and an object of that class is initialized with a brace-enclosed initializer list, and initialization of some member terminates by throwing an exception (thereby interrupting the initialization of the B object), members already completely initialized at that point will not be destructed. However, if the B object is completely initialized, each of its members will be destructed correctly.

```
// Aggregate initialization, cleanup on throw
extern "C" int printf(const char *, ...);
int count;
int stopon;
struct A {
  int n;
  A(int i) : n(i) {
    if (n == stopon) {
      printf("throwing...\n");
      throw "help";
    }
    printf("A::A(%d)\n", n);
    count++;
  }
  ~A() {
    printf("A::~A(%d)\n", n);
    count--;
  }
};
struct B {
  A a;
  const A &r;
  const A &r2;
  A aa;
  ~B() { printf("B::~B()\n"); }
};
main () {
  for (stopon = 1; stopon <= 5; stopon++) {
    try {
      B b = {1, A(2), A(3), 4};
```

```
      throw 5;
    } catch (...) {
      printf("catch\n");
      printf("----------\n");
    }
  }
  return !(count == 0);
}
```

Debugger

- Optimized debugging is not supported in HP C++ V7.1 for OpenVMS Integrity servers system.
- Examining floating point values when compiling /FLOAT=(D_FLOAT,G_ FLOAT)

  On OpenVMS Integrity servers system, the default floating point type is IEEE. If you compile your program with any of these qualifiers: /G_FLOAT, /FLOAT=(D_FLOAT) or /FLOAT=(G_FLOAT) your floating point numbers will be represented internally as integers. Therefore to examine them as floating point values, you must specify the appropriate data type (EXAMINE/D_ FLOAT, EXAMINE/G_FLOAT or EXAMINE/F_FLOAT) when using the EXAMINE command in the OpenVMS Integrity servers debugger.
- Inherited Data Members

  You cannot specify multiple base names. For example, in the program below, to examine member f_member, we can say: "examine j_object.f_member" or "examine j_object.I::f_member" but we cannot say "j_object.I::G::f_member".

```
struct A { int a_member; };
struct B : A { int b_member; };

struct C { int c_member; };
struct D : B, C { int d_member; };

struct E : D { int e_member; };

struct F { int f_member; };
struct G : F { int g_member; };

struct H : E, G { int h_member; };

struct I : H { int i_member; };
struct J : I { int j_member; };

static J j_object;

main(){
    j_object.j_member = 1;
}

DBG> exam j_object.B::A::a_member
 %DEBUG-W-NOFIELD, 'B::A::a_member' is not a field in this record
DBG> deposit j_object.I::a_member = 13
```

- Breakpoints on opening and closing braces of a function

  The OpenVMS Integrity servers debugger may give unexpected results when examining values at the final brace (}) of a routine.

  On OpenVMS Integrity servers system, a workaround to set a break on the last source line of a routine. This is because destructors for the routine may run after the final source line and before the final (}).
- Overloaded Functions

If you attempt to set a break on an overloaded function without providing the argument list, debug may display the %DEBUG-E-NOSYMBOL message instead of displaying the overload list. To see the list of overloaded functions use the sho symbol command.

Currently there is a problem with setting breakpoints on an operator. By using the %name feature of debug, it should be possible to set a breakpoint on an a user defined operator, just as one can set a breakpoint on a user defined destructor. However, this feature is not working at the present time. Please workaround this problem by setting a breakpoint on the appropriate line number.

```
// using %name you can set a break on a destructor
DBG> set break stack::%name'~stack'
// however setting a break on an operator is not working yet
DBG> set break stack::%name'operator++'()
%DEBUG-I-NOTUNQOVR, symbol 'stack::operator++' is overloaded
                    overloaded name stack::operator++
     instance stack::operator++(int)
     instance stack::operator++()
%DEBUG-E-REENTER, reenter the command using a more precise pathname
```

- Pointer to member

  On OpenVMS Integrity servers system, the compiler is not yet emitting full debug information for pointers to members.

- Limited support for namespaces

  On OpenVMS Integrity servers system, the compiler is not yet emitting full debug information for namespaces.

- Unused type and unused labels

  On OpenVMS Integrity servers system, the /DEBUG qualifier means /DEBUG=(TRACEBACK,BRIEF), which causes the compiler to omit debug information for unused labels and unused types, even when /NOOPTIMIZE is specified. This feature results in much smaller object files. If unused labels and unused types are desired, please specify the SYMBOLS keyword explicitly (/DEBUG=(SYMBOLS).

- Source line numbers and listing line numbers

  on OpenVMS Alpha systems, the debugger will display listing line numbers. However, the HP C++ V7.1 compiler on OpenVMS Integrity servers system does not yet support debugging with listing line numbers. Instead you will see source line numbers when in the debugger.

  For Example: in the following program which contains two include files, on OpenVMS Alpha the debugger displays the same listing line numbers as in the listing file. However on OpenVMS Integrity servers , the generic DWARF 3 filename and line number system is used in the debugger, while a listing file will contain standard VMS style listing line numbers.

```
#include <stdio.h>

void t2();
int main() {
  /* comment lines within executable code visible*/
  printf("in main\n");
  t2();
  } /* last line in routine main */
    /* This comment will not be visible */

#include "t2.c"
```

Notice that when we are in the debugger, we see source line numbers, and the location information on a show calls displays module name, source line number.

```
DBG> go
break at routine GEM_BUGS10682\main
     4: int main() {
%DEBUG-I-DYNLNGSET, setting language C++
DBG> type 1:20
module GEM_BUGS10682
     1: #include <stdio.h>
     2:
     3: void t2();
     4: int main() {
     5:   /* comment lines within executable code visible*/
     6:   printf("in main\n");
     7:   t2();
     8:   } /* last line in routine main */
     9: extern "C" {int printf(const char *,...);}
    10: void t2() {
    11:   printf("in t2\n");
    12:   } /* last line in routine t2 */
DBG> sho calls
 module name    routine name     line           rel PC           abs PC
*GEM_BUGS10682  main                4       0000000000000000 0000000000010000
*GEM_BUGS10682  MAIN                        00000000000000F0 0000000000010170
                                            FFFFFFFF80B1CC20 FFFFFFFF80B1CC20
DBG> set break t2
DBG> go
in main
break at routine GEM_BUGS10682\t2
    10: void t2() {
DBG> sho calls
 module name    routine name     line           rel PC           abs PC
*GEM_BUGS10682  t2                 10       0000000000000000 00000000000101C0
*GEM_BUGS10682  main                7       0000000000000050 0000000000010050
*GEM_BUGS10682  MAIN                        00000000000000F0 0000000000010170
```

Below you see the listing file and all of the source lines from the main source file are shown. (The contents of the include would be visible too if /LIS/SHOW=INCLUDE were specified.)

```
     1 #include <stdio.h>
1552
1553 void t2();
1554 int main() {
1555   /* comment lines within executable code visible*/
1556   printf("in main\n");
1557   t2();
1558   } /* last line in routine main */
1559     /* This comment will not be visible */
1560
1561 #include "t2.c"
1568
1569
```

- Breakpoints on `for` loops

  On OpenVMS Alpha systems, if you set a breakpoint on a `for` loop that resides on a single source line, the debugger triggers only once; the `for` loop is considered a single statement.

  On OpenVMS Integrity servers system, the same situation causes the debugger to trigger on each iteration of the `for` loop.

Consider the following for statements:

```
for (i=0; i<10; i++) y_v10_int[i] = x_v10_int[i];

for (i=0; i<10; i++) {y_v10_int[i] = x_v10_int[i];}

for (i=0; i<10; i++) {
        y_v10_int[i]   = x_v10_int[i];
}
```

On OpenVMS Alpha systems, the debugger treats the first two example loops above as single statements, even with delimiting braces. But if the loop is spread over several source lines, as in the third example, the OpenVMS Alpha debugger treats it as a loop.

On OpenVMS Integrity servers system, all three of the above for statements are treated as loops.

10 Release Notes for the V7.1 C++ Libraries

For Integrity servers system, the C++ standard library has been upgraded and organized as a shareable image. All applicable fixes and enhancements done in the C++ standard library for Alpha systems, have been applied to the C++ standard library for Integrity servers system.

For Integrity servers system, the C++ class library is based on the same code as the C++ class library on Alpha systems. The major change in the C++ class library for Integrity servers system is the removal of the tasks and complex packages.

10.1 Library Reorganization

The standard library, language run-time support library, and class library have been reorganized for Integrity servers system.

10.1.1 Standard Library and Language Run-Time Support Library

On Alpha systems, the C++ standard library and language run-time support library is delivered in an object library, LIBCXXSTD.OLB, shipped with the compiler kit.

On Integrity servers system, the C++ standard library and language run-time support library are delivered as separate system shareable images shipped with the base operating system. The names of the images are: CXXL$RWRTL.EXE and CXXL$LANGRTL.EXE, respectively. The images reside in the SYS$LIBRARY directory and are installed at system startup. The LIBCXXSTD.OLB object library does not exist on Integrity servers system.

On Alpha systems, the default C++ standard library LIBCXXSTD[_MA].OLB is a preinstantiation library. It contains instantiations of commonly used templates on commonly uses types like std::basic_string<char>, std::basic_iostream<char>, and so on. When you compile with /NOIMPLICIT_INCLUDE, standard library symbols defined in template definition files can be resolved from the preinstantiation library.

On Integrity servers system, there is no preinstantiation library. As a result, compiling with /NOIMPLICIT_INCLUDE can result in undefined symbols for standard library symbols that could have been resolved on Alpha systems from the preinstantiation library.

10.1.2 Class Library

On Alpha systems, there are three class library shareable images: CXXL$011_SHR.EXE, CXXL$011_SHRTASK.EXE, and CXXL$011_TASK.EXE.

On Integrity servers system, the C++ class library continues to ship as a system shareable image. Because the tasks and complex packages have been removed, there is only one class library image: CXXL$011_SHR.EXE.

10.2 Language Run-Time Support Library

The following language run-time support library change has been made:

- The language run-time support library no longer validates if a negative value has been specified in a call to operator `new`. Instead, the value is treated as an unsigned value, and an attempt is made to dynamically allocate the specified memory.

10.3 Class Library

The following class library changes have been made:

- The tasks and complex packages have been removed. The recommended replacements are the pthreads routines and complex template class, respectively, from the C++ standard library.
- In the String class, the `char*()` operator, which converts String to a pointer to `char`, has been removed. The String class has a `const char*()` operator, which can be used instead of the removed one.

10.4 Standard Library

This section describes changes to the C++ standard library.

10.4.1 Changes

There are two major changes in the C++ standard library for Integrity servers system as compared with the standard library for Alpha systems:

- The C++ standard library has been upgraded from Version 2.0 of the Rogue Wave C++ Standard Library to Version 3.0.
- The C++ standard library is delivered with the operating system as the installed system shareable image SYS$SHARE:CXXL$RWRTL.EXE, and also in STARLET.OLB in the object form for linking /NOSYSSHARE. On Integrity servers system, there is no LIBCXXSTD.OLB, which is the object library where the C++ standard library for OpenVMS Alpha resides.

Additional standard library changes, known issues, and platform differences are noted in the following sections.

10.4.2 Library Headers

While from the customers' perspective the change in the library distribution model is supposed to be transparent (except that application images will be much smaller on Integrity servers system), users on Integrity servers system may find that the new C++ Standard Library is much less forgiving in terms of including all necessary library headers than the old Standard Library.

For example, the following program compiles cleanly on OpenVMS Alpha systems despite the fact that it does not include the <iostream> header necessary for the `std::cout` object:

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

using namespace std;

main() {
  cout << "hello, world";
}
```

However, on OpenVMS Integrity servers system, compilation fails with the following error:

```
%CXX-E-UNDECLARED, identifier "cout" is undefined
```

It is nearly impossible to describe all combinations of library constructs and header files that would compile cleanly on Alpha systems and yet fail to compile on Integrity servers system because a library header required by the C++ standard for a particular construct has not been included. If a program that used to compile cleanly on an Alpha system fails to compile on an Integrity servers system, it is always a good idea to check that all necessary library headers are included.

10.4.3 Internal Library Headers and Macros

A program that includes internal RW stdlib V2.0 library headers, like `<stddefs>` or `<stdcomp>`, or that uses internal library macros _RW_*, will have to be modified because the new C++ standard library does not necessarily have the same internal headers or use the same internal macros as the old one.

10.4.4 Known Issues

10.4.5 Differences Between Alpha and Integrity servers system

The following are differences between the Integrity servers and Alpha standard libraries:

- According to section 27.6.1.3 [lib.istream.unformatted] of the C++ Standard, the following get member functions of the std::basic_istream class should call setstate(failbit) if no characters have been stored, as is the case for an empty line. While on Integrity servers system the functions set failbit, on Alpha systems they do not:

  ```
  istream_type& get(char_type *s, streamsize n, char_type delim);
  istream_type& get(char_type *s, streamsize n);
  ```

  For example, on an Alpha system, the following program invoked to read its own source (like `pipe run x.exe < x.cxx`), reads the whole file and outputs: "21 lines were read".

  On an Integrity servers system, the program exits the `for` loop after encountering an empty line and outputs: "6 lines were read".

  ```
  #ifndef __USE_STD_IOSTREAM
  #define __USE_STD_IOSTREAM
  #endif
  #include <iostream>
  #include <cassert>
  using namespace std;
  ```

```
int main() {
  char buffer[1024], c;
  int lines;
  for (lines = 0; cin.good(); ++lines) {
    cin.get(buffer, sizeof(buffer));
    if ( !cin.eof() ) {
      // consume delimeter
      assert( (cin.get(c), c) == '\n' );
    }
  }
  cout << lines << " lines were read\n";
}
```

- On OpenVMS Integrity servers system, a program mixing objects of std::complex class instantiated on different integral types or on an integral type and a floating type in the same expression, needs to be compiled with the _RWSTD_NO_MEMBER_TEMPLATES macro defined. Note that according to Section 26.2 - Complex numbers [lib.complex.numbers] of the C++ standard, the template std::complex can be instantiated only on a floating type. From the standard: "The effect of instantiating the template complex for any type other than float, double or long double is unspecified".

  For example, the following program compiles on Alpha systems without any special macro defined. On Integrity servers system, it compiles only with the _RWSTD_NO_MEMBER_TEMPLATES macro defined:

  ```
  #include <complex>
  int main() {
    std::complex<int> ci(1, 2);
    std::complex<long> cl(3, 4);
    ci += cl;
  }
  ```

- On OpenVMS Alpha systems, the following constructors for the C++ standard library classes strstream and ostrstream initialize ptr[count-1] with a null byte:

  ```
  strstream(char *ptr, streamsize count,
            ios_base::openmode mode = ios_base::in | ios_base::out);

  ostrstream(char *ptr, streamsize count,
             ios_base::openmode mode = ios_base::out);
  ```

  This initialization is not required by the C++ standard, and on Integrity servers system the C++ standard library does not do it.

- On Integrity servers system, map and multimap containers require the standard-conformant form of allocator class: allocator<pair<const Key, T> >.

  For example, on Alpha systems, it is possible to declare an object of class multimap as the following, with the second template argument of allocator class omitted:

  ```
  multimap<string, int, less<string>, allocator<string> > x;
  ```

  But for Integrity servers system, this must be changed to:

  ```
  multimap<string, int, less<string>, allocator<pair<const string, int> > > x;
  ```

- On Integrity servers system, the exception.what( ) function reports the module name, and the message text might be different.

For example, an output on Alpha systems:

```
Got an exception: string index out of range in function:
basic_string:::replace(size_t,size_t,size_t,char) position: 100 is
greater than length: 0
```

An output on Integrity servers system:

```
Got an exception: CSRC:[STDIPF_INCLUDE]STRING.CC;:416:
basic_string::replace(size_type, size_type, size_type, value_type):
argument value 100 out of range [0, 0)
```

- On Integrity servers system, iostreams extraction operators truncate out-of-range integer values to the maximum possible value for a given type, and set the failbit for the stream.

  For example, consider the following program:

  ```
  #ifndef __USE_STD_IOSTREAM
  #define __USE_STD_IOSTREAM
  #endif

  #include <strstream>
  #include <iostream>

  using namespace std;

  main() {
    istrstream is("32768"); // SHRT_MAX is 32767
    short s;
    is >> s;
    cout << is.fail() << endl;
    cout << s << endl;
  }
  ```

  On Alpha systems, this program gives:

  ```
  0
  -32768
  ```

  On Integrity servers system, it gives:

  ```
  1
  32767
  ```

  Note that on Integrity servers system, the failbit for the stream is set.

  According to the C++ Standard, Section 22.2.2.1 - Template class num_get [lib.locale.num.get], an input that would have caused `scanf` to report an input failure should result in setting ios_base::failbit to err. Since on OpenVMS, `scanf` reports an input failure in this case (this is an undefined behavior from the point of view of the C standard), the behavior of the C++ standard library on Integrity servers system is standard-compliant.

- On Alpha systems, the find template function is implemented using `operator!=`. On Integrity servers system, this function is implemented using `operator==`, which according to the C++ standard is the operator the `find` function should be using.

  Consequently, if no conversion from `*InputIterator` to `T` exists, on Alpha systems the following function can be instantiated only if `operator!=(*InputIterator,T)` exists:

  ```
  find(InputIterator first, InputIterator last, const T& value)
  ```

  On Integrity servers system, however, the function can be instantiated only if `operator==(*InputIterator,T)` exists.

The following program illustrates the difference. If you comment out the line `bool operator!=(S, int);`, the program does not compile on Alpha systems. If you comment out the line `bool operator==(S, int);`, the program does not compile on Integrity servers systems. The behavior on Integrity servers system is the standard-conformant behavior.

```
include <algorithm>
#include <vector>

struct S {
  int i;
};

bool operator!=(S, int);
bool operator==(S, int);

void foo() {
  std::vector<S> v;
  std::find(v.begin(), v.end(), 0);
}
```

- On Integrity servers system, an attempt to write into a stream opened for read (ios::in), causes the stream badbit bit to be set.

  On both Alpha and Integrity servers system, nothing is written into a stream opened for read. However, on Alpha systems, the stream badbit bit is not set.

  The C++ standard does not provide explicit guidance about what to do in this case. However, the behavior on Integrity servers system is more reasonable—at least there is an indication that something was wrong.

- On Integrity servers system, reverse_iterator cannot be instantiated on vector<bool>::iterator type.

  For example, the following program, which compiles cleanly on Alpha systems, does not compile on Integrity servers system:

  ```
  #include <vector>

  typedef std::reverse_iterator<std::vector<bool>::iterator> ri;

  main()
  {
  ri::pointer (ri::*foo)() const = &ri::operator->;
  }
  ```

  A recently adopted resolution for the library issue 120 has made this construct invalid. See http://std.dkuug.dk/JTC1/SC22/WG21/docs/lwg-active.html#120 for more details.

- On Integrity servers system, for a random access iterator, operator-(const random_access_iterator&) returning `difference_type` must be `const`.

  For example, the following program compiles cleanly on Alpha systems. However, on Integrity servers system it compiles only if `// const` is uncommented.

  ```
  #include <algorithm>

  template <class T> class randomaccessiterator {

  public:

       typedef T value_type;
       typedef int difference_type;
       typedef T* pointer;
       typedef T& reference;
       typedef std::random_access_iterator_tag iterator_category;
  ```

```
    bool operator==(const randomaccessiterator&);
    bool operator!=(const randomaccessiterator&);
    T& operator*() const;
    T* operator->();

    randomaccessiterator& operator++();
    const randomaccessiterator& operator++(difference_type);
    randomaccessiterator& operator--();
    const randomaccessiterator& operator--(difference_type);
    randomaccessiterator& operator+=(difference_type);
    randomaccessiterator& operator+(difference_type);
    randomaccessiterator& operator-=(difference_type);
    randomaccessiterator& operator-(difference_type);
    difference_type operator-(const randomaccessiterator&); // const;
};

struct S {};
typedef randomaccessiterator<S> Iterator;
typedef bool (*Predicate)(Iterator::value_type);
template Iterator std::stable_partition<Iterator, Predicate>(Iterator,
Iterator, Predicate);
```

Table 76 in the C++ standard specifies the requirements for a random access iterator. It says the expression `b - a` must be valid, where `a` and `b` denote values of X, the random access iterator. It is not completely clear from the standard whether values of X also imply `const` values of X, but if the answer is yes, the behavior on Integrity servers system is correct.

- On Integrity servers system, an attempt to call the strstream.seekg(0) function for an empty stream (the one whose 'next' pointer is NULL) causes the stream failbit to be set.

  This is a standard-compliant behavior. Notice that after the failbit is set for the stream, the strstream.str( ) function returns a NULL pointer.

- On Integrity servers system, after a call to string.resize(newsize), string.capacity( ) does not necessarily returns newsize.

  While on Alpha systems the string.capacity( ) function returns newsize, this is not required by the C++ standard. A program relying on Alpha behavior should be modified to call the string.size( ) function instead.

- On Integrity servers system, there is no overload of basic_string class for type `bool`.

  Version v3.0 of the Rogue Wave C++ standard library does not have this problematic nonstandard overload. For OpenVMS Alpha, it has been recently removed from the library.

- On Integrity servers system, class std::fpos<std::mbstate_t> does not have the nonstandard member function offset( ). You can use fpos::operator streamoff( ) instead. For example:

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif

#include <sstream>

using namespace std;
```

```
void foo() {
  istringstream in("hello, world");
  streamoff offset;
  offset = in.tellg().offset();    // Alpha only
  offset = streamoff(in.tellg());  // either Alpha or Integrity servers
}
```

- On OpenVMS Alpha systems, operator< used by the non-predicate forms of the sorting algorithms like push_heap( ) may or may not be declared constant.

  On Integrity servers system, this operator must be a constant operator. For example, on OpenVMS Alpha systems, the following program would compile without 'bool operator<(const S&)' qualified as `const`. On Integrity servers system, the `const` is necessary.

```
#include <vector>
#include <algorithm>

struct S {
  bool operator<(const S&) const;
};

void foo() {
  std::vector<S> v;
  std::push_heap(v.begin(),v.end());
}
```

  The C++ standard does not specify whether or not this program compiles. Since operator< must induce a strict-weak ordering on its arguments, which prevents it from modifying their values, there should be no reason not to declare it `const`.

- On OpenVMS Alpha systems, for an unsigned type, the C++ standard library iostreams ignores std::showpos manipulator.

  On Integrity servers system, the plus sign is displayed. This behavior is standard-compliant.

  Consider the following example:

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

main() {
  std::cout << std::showpos << (unsigned)1 << std::endl;
}
```

  On OpenVMS Alpha systems, `x.cxx` outputs:

```
1
```

On Integrity servers system, `x.cxx` outputs:

```
+1
```

10.4.6 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.

10.4.6.1 Using the C++ Standard Library in Microsoft Standard Mode

Compiling /STANDARD=MS has the following restrictions:

- Header <typeinfo> does not compile.
- Header <new> does not declare operator delete[](void*, void*).
- Header <new> does not declare type new_handler.
- You cannot use std::reverse_iterator directly. For example, the following program does not compile in Microsoft mode:

```
#include <iterator>
std::reverse_iterator<int> x;
```

- You cannot use types defined in std::iterator_traits. For example, the following program does not compile in Microsoft mode:

```
#include <iterator>

template <class T>
typename std::iterator_traits<T>::value_type
foo(void);

void bar() {
  foo<int*>();
}
```

- The std::count and std::count_if algorithms are not available in Microsoft mode.
- You cannot use the std::sort and std::stable_sort algorithms on containers of pointers. For example, the following program does not compile in Microsoft mode:

```
#include <vector>
#include <algorithm>

void foo()
{
  std::vector<int*> v;
  std::stable_sort(v.begin(),v.end());
  std::sort(v.begin(),v.end());
}
```

- You cannot take the address of a bitmask member of std::ios_base class. For example, the following program does not compile in Microsoft mode:

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <ios>
const std::ios_base::fmtflags *x = &std::ios_base::boolalpha;
```

11 CXXLINK Changes

Because of changes in the architecture on Integrity servers system, CXXLINK plays a much smaller role. Its only remaining purpose is to provide human readable (demangled) names for mangled C++ names in diagnostics generated by the linker.

Specific changes are:

- There is no LIBCXXSTD.OLB

On Integrity servers system, there is no LIBCXXSTD.OLB, which is the object library where the C++ standard library for OpenVMS Alpha resides. See Section 10.4 for more information.

- The library is reorganized

  The C++ libraries have been reorganized and incorporated into the base system. CXXLINK no longer needs to specify any C++ libraries when invoking the system linker. See Section 10 for more information.

- There are no templates in a repository

  With the new template instantiation model, objects are no longer placed in a repository. Therefore, CXXLINK no longer needs to look at the repositories for templates. See Section 9.3.9 for more information.

12 Installation

To install HP C++ for OpenVMS Integrity servers system, set the default directory to a writeable directory to allow the IVP to succeed. Then run the PRODUCT INSTALL command, pointing to the kit location. For example:

```
$ SET DEFAULT SYS$MANAGER
$ PRODUCT INSTALL CXX/SOURCE=node::device:[kit_dir]
```

After installation, these C++ release notes will be available at:

SYS$HELP:CXX.RELEASE_NOTES

SYS$HELP:CXX_RELEASE_NOTES.PS

Here is a sample installation log:

```
$ PRODUCT INSTALL CXX/SOURCE=NODE1$::DEV1$:[I64_CPP_KIT]

The following product has been selected:
    HP I64VMS CXX V7.2-052                Layered Product

Do you want to continue? [YES]

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for
any products that may be installed to satisfy software dependency requirements.

HP I64VMS CXX V7.2-052: HP C++ for OpenVMS Industry Standard

    Copyright 2006 Hewlett-Packard Development Company, L.P.

    This software product is sold by Hewlett-Packard Company

    PAKs used: CXX-V

Do you want the defaults for all options? [YES]

    Copyright 2006 Hewlett-Packard Development Company, L.P.

      HP, the HP logo, Alpha and OpenVMS are trademarks of
      Hewlett-Packard Development Company, L.P. in the U.S. and/or
      other countries.

      Confidential computer software. Valid license from HP
      required for possession, use or copying.  Consistent with
      FAR 12.211 and 12.212, Commercial Computer Software, Computer
      Software Documentation, and Technical Data for Commercial
      Items are licensed to the U.S. Government under vendor's
      standard commercial license.

Do you want to review the options? [NO]

Execution phase starting ...
```

```
The following product will be installed to destination:
    HP I64VMS CXX V7.2-052                DISK$ICXXSYS:[VMS$COMMON.]
The following product will be removed from destination:
    HP I64VMS CXX V7.1-152                DISK$ICXXSYS:[VMS$COMMON.]

Portion done: 0%...60%...70%...80%...90%...100%

The following product has been installed:
    HP I64VMS CXX V7.2-052                Layered Product
The following product has been removed:
    HP I64VMS CXX V7.1-155                Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for HP I64VMS CXX V7.2-052 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully

HP I64VMS CXX V7.2-052: HP C++ for OpenVMS Industry Standard

  The release notes are located in the file SYS$HELP:CXX.RELEASE_NOTES
  for the text form and SYS$HELP:CXX_RELEASE_NOTES.PS for the postscript form.
```

12.1 Multiple Version Support

Version 7.3 adds optional support for having multiple versions of the C compiler on your system. It works by appending an ident name to a previously installed compiler and saving it alongside the new compiler from this kit. Users on your system can then execute the sys$system:cxx$set_version.com and sys$system:cxx$show_versions.com command procedures to select the desired compiler for a given process and to view the list of available compiler versions.

To set this up, have your system administrator run the installation procedure, answering NO to the question about default options:

```
Do you want the defaults for all options? [YES] NO <RET>
```

Then answer YES to the question about making alternate compilers available:

```
Would you like to set up your system for running alternate versions of C? [NO] YES <RET>
```

Users can then execute the cxx$set_version.com command procedure with an argument to set up process default logicals that point to alternate compiler versions. For more information on using cxx$set_version.com and cxx$show_version.com see section: "Enhancements, Changes, and Problems Corrected in V7.3".

Sample installation for multiple-version Support:

```
$ product install CXX /source=disk$:[dir]

The following product has been selected:
    HP I64VMS CXX T7.3-18                 Layered Product

Do you want to continue? [YES]

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for
any products that may be installed to satisfy software dependency requirements.

HP I64VMS CXX T7.3-18: HP C++ for OpenVMS Industry Standard

    Copyright 2003-2007 Hewlett-Packard Development Company, L.P.

    This software product is sold by Hewlett-Packard Company

    PAKs used: CXX-V or CXX-V-USER

Do you want the defaults for all options? [YES] no

    HP I64VMS VMS V8.3 [Installed]
```

```
    * Configuration options for this referenced product cannot
    * be changed now because the product is already installed.
    * (You can use PRODUCT RECONFIGURE later to change options.)

    Copyright 2003-2007 Hewlett-Packard Development Company, L.P.

      HP, the HP logo, Alpha and OpenVMS are trademarks of
      Hewlett-Packard Development Company, L.P. in the U.S. and/or
      other countries.

      Confidential computer software. Valid license from HP
      required for possession, use or copying.  Consistent with
      FAR 12.211 and 12.212, Commercial Computer Software, Computer
      Software Documentation, and Technical Data for Commercial
      Items are licensed to the U.S. Government under vendor's
      standard commercial license.


     Multi_Version Support:
     If you would like to set up your system to be able to run different
     versions of the compiler then answer yes. The installation procedure will
     then copy the previously installed C++ compiler and associated files along
     side the new compiler in this kit with a suffix appended to the name that
     corresponds to the version number.  Users may then execute
     CXX$SET_VERSION.COM to run an alternate version of the compiler
     and CXX$SHOW_VERSIONS.COM to show available versions at anytime
     after this installation.

    Would you like to set up your system for running alternate versions of C++? [NO] yes

Do you want to review the options? [NO]

Execution phase starting ...

The following product will be installed to destination:
    HP I64VMS CXX T7.3-18                 DISK$ICXXSYS:[VMS$COMMON.]

Portion done: 0%...60%...70%...80%...90%...100%

The following product has been installed:
    HP I64VMS CXX T7.3-18                 Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for HP I64VMS CXX T7.3-18 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully

HP I64VMS CXX T7.3-18: HP C++ for OpenVMS Industry Standard

    Text form of the release notes are located in the file SYS$HELP:CXX.RELEASE_NOTES

      SYS$HELP:CXX_RELEASE_NOTES.PS contains the postscript form.

    A startup file SYS$STARTUP:CXX$STARTUP.COM has been provided.

        It contains commands which can be executed after the product install
        procedure has been run and at startup to allow for the best compilation
        performance.  You may want to invoke this command file from your
        system's site-specific start up file.  This command file does not
        have to be invoked for correct operation of HP C++.
```

13 Reporting Problems

Please report problems or offer feedback using the mechanisms specified in the HP C++ User's Guide preface.