VMS Software

VSI C++ for OpenVMS User Guide

Document Number: DO-VIBHAA-012

Publication Date: May 2024

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI C++ Version 7.4-6 for OpenVMS I64
VSI C++ Version 7.4-8 for OpenVMS Alpha

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI C++ for OpenVMS User Guide

Legal Notice

Preface

This manual contains information about developing and debugging VSI C++ programs on OpenVMS systems, and includes information on other OpenVMS features and tools that work with the compiler.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for experienced programmers who need to develop VSI C++ programs on OpenVMS systems. Users of this manual should have a basic understanding of the C++ language and some familiarity with the Digital Command Language (DCL).

3. Document Structure

This manual is organized as follows:

- Chapter 1 shows how to create, compile, link, and run VSI C++ programs.
- Chapter 2 describes features and characteristics that are specific to the VSI C++ implementation.
- Chapter 3 describes guidelines and procedures for customizing your language environment.
- Chapter 4 describes how to make code used with other C++ implementations acceptable to the VSI C++ compiler.
- Chapter 5 describes how to use templates with VSI C++.
- Chapter 6 explains how to use C++ exception handling.
- Chapter 7 describes the VSI C++ implementation of the C++ Standard Library.
- Chapter 8 explains how to use the OpenVMS Debugger with VSI C++.
- Chapter 9 explains how to use 64-bit address space.
- Appendix A describes compiler command qualifiers.
- Appendix B provides information on using programming tools with VSI C++.
- Appendix C describes built-in functions.
- Appendix D describes Class Library restrictions.

4. Related Documents

The following documents contain information associated with topics in this manual:

- Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1997.

This text combines a user guide and language reference manual to provide an exhaustive introduction to the C++ programming language, including sophisticated language features. Where appropriate, section numbers shown in parentheses (for example, §r.2.3) refer to relevant portions of *The Annotated C++ Reference Manual*.

- *VSI C++ Class Library Reference Manual*

  This manual describes a library of VSI C++ classes.

- *VSI C++ Installation Guide for OpenVMS Alpha*

  This document supplies the information necessary to install VSI C++ on OpenVMS Alpha systems.

- *VSI C++ Installation Guide for OpenVMS I64*

  This document supplies the information necessary to install VSI C++ on OpenVMS I64 systems.

- *VSI C Run-Time Library Reference Manual for OpenVMS Systems [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]*

  This library manual provides information, useful to VSI C++ users, on the OpenVMS Run-Time Library (RTL) for C functions and macros, which include the ANSI C standard library. This manual also contains information about porting programs to and from other operating systems.

The Annotated C++ Reference Manual and the *STL Tutorial and Reference Guide* are available only in printed form. Online copies are not available.

5. Related Documents

- Carroll, Martin D. and Margaret E. Ellis. *Designing and Coding Reusable C++*. Reading, Massachusetts: Addison-Wesley, 1995.

  This text provides practical information for designing and implementing C++ programs.

- Myers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, 3rd edition*. Reading, Massachusetts: Addison-Wesley, 1997.

- Myers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, Massachusetts: Addison-Wesley, 1995.

  These texts provide practical information for designing and implementing C++ programs.

- *International Standard ISO/IEC 14882*

  Defines the C++ International Standard. The document is available for downloading at the ANSI Electronic Store (start at http://www.ansi.org).

  The printed version is also available for purchase from the same web site. Choose “Catalogs/ Standards Information”, then “ANSI-ISO-IEC Online Catalog”, then search for “14882”.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have

VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

8. Platform Labels

This guide contains information applicable to the VSI OpenVMS operating system on Alpha and Intel Itanium processors. The information in this guide applies to both of these processors, except when specifically labeled as follows:

(Alpha only)	Specific to the OpenVMS operating system running on an Alpha processor.
(I64 only)	Specific to the OpenVMS operating system running on an Intel Itanium processor. On this platform, the product name of the operating system is OpenVMS for Industry Standard 64 for Integrity servers (abbreviated in this manual as OpenVMS I64 or I64).

9. Conventions

The conventions found in the following table are used in this document.

Convention	**Meaning**

`class complex {`
`  .`
`  .`
`  .`
`};`
	A vertical ellipsis indicates that some intervening program code or output is not shown. Only the more pertinent material is shown in the example.

`, ...`	A horizontal ellipsis in a syntax description indicates that you can enter additional parameters, options, or values. A comma preceding the ellipsis indicates that successive items must be separated by commas.
`The complex class ...` `The get() function ...`	Monospaced type denotes the names of VSI C++ language elements, and also the names of classes, members, and nonmembers. Monospaced type is also used in text to reference code elements displayed in examples.
italic	Italic type denotes the names of variables that appear as parameters or in arguments to functions.
boldface	Boldface type in text indicates the first instance of terms defined in text.
UPPERCASE, lowercase	UNIX operating system differentiates between uppercase and lowercase characters. Literal strings that appear in examples, syntax descriptions, and function definitions must be typed exactly as shown.

10. New and Changed Features in C++ I64 Version 7.2

Some of the new or changed features supported by this version of the compiler are:

- 64-bit pointer support is added for C++ I64.

  This support is compatible with the 64-bit pointer support in the C++ and C compilers for OpenVMS Alpha. 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`). Please see the V7.2 release notes for more information on 64-bit pointer support for the I64 compiler.

- Variadic macros are now supported.

  This feature allows macros to take a variable number of arguments. It was added to VSI C Version 6.4 and is supported by a number of other C and C++ compilers. This feature is available only when the value of the /STANDARD qualifier is RELAXED (the default), MS, or GNU.

- Support is added 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 original name.

- Prologue and epilogue file header processing is now supported in VSI 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.

11. New and Changed Features in C++ Version 7.1

C++ Version 7.1 runs on OpenVMS Alpha and OpenVMS Integrity servers. The compiler behaves much the same on both systems, with some differences, primarily in the support for built-in functions, default floating-point representation, and predefined macros. These differences are noted in the relevant sections of this manual.

Some of the new or changed features supported by this version of the compiler on both Alpha and I64 systems are:

- `cname` header support is added (Section 3.1).

  The C++ compiler implements section 17.4.1.2 - Headers [lib.headers] "C++ Headers for C Library Facilities" of the *C++ Standard*.

  The implementation consists of 18 `<cname>` headers defined in the Standard (*Chapter 3*). As required by the C++ standard, the `<cname>` headers define C names in the `std` namespace.

The /[NO]PURE_CNAME qualifier is added to control insertion of the names by `<cname>` headers into the `std` namespace only (/PURE_CNAME), or into the `std` namespace and the global namespace (/NOPURE_CNAME).

- The /[NO]FIRST_INCLUDE=(file[,…]) qualifier is added (see Appendix A for the detailed description).

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

- The `#pragma include_directory` preprocessor directive is added (Section 2.1.1.7).

  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.

- Changes are made to the /WARNING qualifier and compiler messages (Section 2.5).

  Changes to the /WARNINGS qualifier 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.

  Also, some compiler diagnostics might be different on Alpha and I64 systems, and some conditions detected on one platform might not be detected on the other.

- A new C++ front end is added to provide improved conformance to the C++ International Standard.
- Support for /STANDARD=CFRONT is retired.

Chapter 1. Building and Running C++ Programs

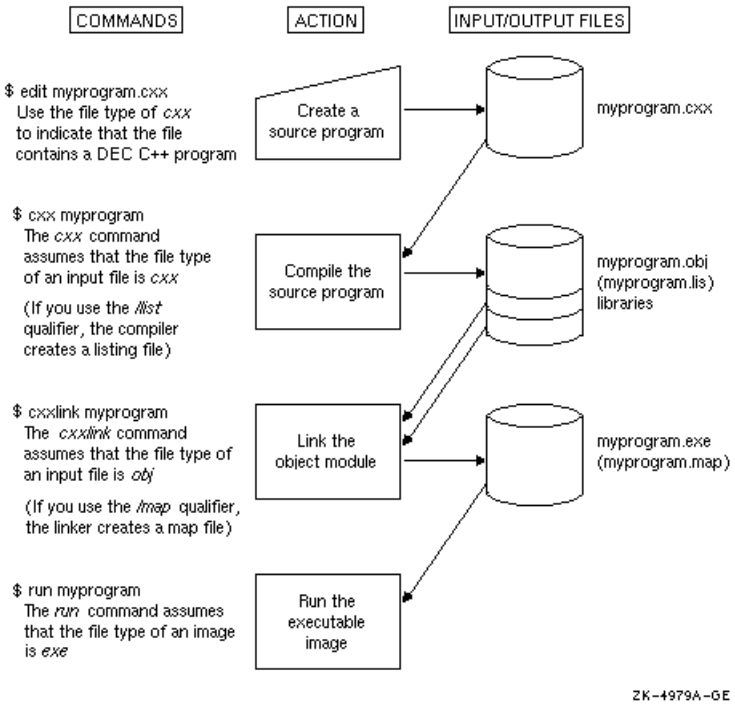
C++ is an evolving language in which new features have recently replaced outmoded constructs. The C++ Standard Library provided with this release defines a complete specification of the C++ International Standard, with some differences, as described in the in the online release notes in:

```
SYS$HELP:CXX_RELEASE_NOTES
```

When switching from a Version 5.n compiler, you might need to modify your source files, especially if you use the default language mode. In addition, language changes can affect the run-time behavior of your programs. If you want to compile existing source code with minimal source changes, compile using the /STANDARD=ARM qualifier. See Chapter 2.

This chapter provides information about basic steps in developing a C++ program on an OpenVMS system. These steps are shown in Figure 1.1.

Figure 1.1. Steps in Developing a C++ Program



To create and modify a C++ program, you must invoke a text editor. The OpenVMS system provides you with at least two text editors: VAX EDT (EDT) and the VSI Text Processing Utility (DECTPU). Another editor that you can use is the VSI Language-Sensitive Editor (LSE), which is sold separately (see Appendix B for more information on LSE). Use *.cxx* as the file type to signify that you are creating a C++ source program.

When you compile your program with the `CXX` command, you do not have to specify the file type; by default, C++ first looks for files with the *.cxx* type.

If the compilation succeeds, the compiler creates an object file with the type *.obj*. If the compiler detects errors, the system displays each error detected. You then reinvoke a text editor to make corrections.

When your program compiles successfully, you use the CXXLINK facility to create an executable image. Compiler and linker commands both take qualifiers, as described in Sections 1.2 and 1.3.

When you have an executable image file, use the `run` command, or define a foreign command, to run your program. See Section 1.5 for more information on running image files.

1.1. Using the DECTPU Text Editor

With DECTPU, you have a choice of two editing interfaces, the Extensible Versatile Editor (EVE) or the DECTPU EDT Keypad Emulator. You can also create your own interfaces with DECTPU. At any time during your editing session you have access to online help.

When you invoke DECTPU to create a file, the editor automatically creates a journal file, which you can use to recover your keyboard entries if the system fails during an editing session. To initiate recovery, use the following command format:

```
edit/tpu/recover file-spec
```

The interactive editor interface, EVE, responds to all the common editing functions, invoked using the editing keypad, and supports more advanced functions that you type as commands on the EVE command line. For more information on using EVE, see the Guide to VMS Text Processing.

1.2. Using the Compiler

The compiler detects source program errors and shows each error either in a screen display or in the batch log file, depending on whether you run the compiler interactively or in batch mode. If the compilation succeeds, the compiler generates machine-language instructions from the source statements, and groups these instructions into an object module for the linker.

The compiler command `cxx` has the following format:

```
cxx[/qualifier...][ file-spec [/qualifier...]],...
```

You use qualifiers to instruct the compiler to perform some action. A qualifier placed immediately after the CXX command affects all the files listed. A qualifier placed immediately after a file specification affects only the preceding file, unless you concatenate your files. A qualifier placed on an individual file specification overrides a qualifier placed immediately after the CXX command.

If you include more than one file specification on the same line, use commas (,) or plus signs (+) as separators. For example:

```
$ cxx/list prog_1, prog_2, prog_3
```

A comma instructs the compiler to keep source files separate and to create an object file and a listing file for each source file. A plus sign instructs the compiler to concatenate each of the specified source files, and to create one object file and one listing file. Any qualifier specified for one file within a list of concatenated files affects all these files.

Note

Comma lists are not supported on I64 systems. Their use causes a fatal error.

1.2.1. Compiler Command Qualifiers

For a complete description of command line qualifiers, refer to Appendix A or to the online HELP.

1.2.2. Compiler Error Messages

If the compiler detects errors in your source code, the compiler signals these errors by displaying diagnostic messages in the following format:

```
%CXX-s-ident, message-text
                  at line number n in file name
```

Where:

s
Is the error severity, represented as follows:

F	Fatal error. The compiler stops immediately without producing an object file. You cannot link the program until you correct this error.
E	Error. The compiler proceeds, and possibly generates other messages, but does not produce an object file. You cannot link the program until you correct this error.
W	Warning. The compiler takes some corrective action and produces an object file. However, to avoid unexpected results you must verify that the compiler's action is what you wanted.
I	Information. The compiler informs you of specific actions taken. You need not take any action yourself regarding this message.
S	Success.

ident
Is a mnemonic (abbreviation) of the message text.

message-text
Is the full text of a compiler diagnostic message explaining what happened.

n
Is an integer that gives you the number of the line where the error occurs. The number is relative to the beginning of the file or module in which the error occurs. The number appears on your terminal but not in listing files.

name
Is the name of the file or module in which the error occurs. The name appears on your terminal but not in listing files.

To be sure your program runs successfully, examine the diagnostic messages, evaluate error severity, and make any necessary corrections.

You can suppress certain information and warning diagnostic messages using the `#pragma message` preprocessor directive. For information about this directive, see Section 2.1.1.11.

1.3. Linking a Program *(Alpha only)*

This section describes how to link a C++ program on OpenVMS Alpha systems.

After your program or module successfully compiles, you **must** use the CXXLINK facility to combine your object modules into one executable image.

The CXXLINK facility is layered on the OpenVMS Linker utility and provides the ability to link your C++ application. Besides linking your C++ application, the CXXLINK facility completes the automatic

template instantiation process; see Chapter 5 for details. CXXLINK also ensures that the Standard Template Library run-time support and the exception handling run-time support are linked into your application as needed.

CXXLINK uses the same command line format that you would use to invoke the OpenVMS Linker utility; thus, you can simply replace the LINK verb with CXXLINK in your command procedures. The CXXLINK command has the following format:

```
CXXLINK[/command-qualifier]... {file-spec[/file-qualifier...]},...
```

If you include more than one input file specification, use commas or plus signs as separators. By default, the linker creates an output file with the same name as the first input file and the file type *.exe*. If you want the output file to take the name of your main program, be sure to specify your main program file first. You can also use the /EXECUTABLE=*name*`.exe` qualifier on the CXXLINK command line to specify a name for the executable image.

Do not use the linker `cluster=` option to reference OpenVMS object modules that define global static objects. Using this option prevents the constructors and destructors for global static objects from being run during image activation and exit.

Caution

The OpenVMS Linker expects a function whose identifier is main. If a C++ program lacking a definition of main is inadvertently linked, then program execution begins at the first function seen by the linker.

1.3.1. CXXLINK Interactions with OpenVMS Linker Qualifiers

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

1.3.1.1. Command Parameters and Qualifier

In addition to the following qualifiers, the CXXLINK command accepts the same parameters and qualifiers as the OpenVMS Linker utility (see Section 1.3.5 for some of the more useful OpenVMS Linker qualifiers). CXXLINK-specific qualifiers are stripped off prior to calling the OpenVMS Linker utility and therefore have no effect on default device or directory specifications applied by the OpenVMS Linker facility.

Command Qualifiers	**Defaults**
/[NO]LOG_FILE[=filename]	/NOLOG_FILE
/PREINST	/PREINST
/PRELINK=(option[,option2])	See HELP.
/REPOSITORY=(*writeable-repository*[,*readonly-repository*,...])	See HELP.
/USE_LINK_INPUT[=filename]	/NOUSE_LINK_INPUT
/VERSION	None.

For more information about CXXLINK qualifiers and parameters, type HELP CXXLINK.

1.3.2. Migrating from LINK to CXXLINK

Because a single CXXLINK command can invoke the OpenVMS Linker utility multiple times, you must not specify user mode (DEFINE/USER_MODE) logical names. If CXXLINK executes a second LINK command, the original DEFINE/USER_MODE logical name is not retained for that second command. Incorrect results can occur.

You should check command procedures that perform link operations of code generated by the C++ compiler for any /USER_MODE logical names that are intended to be in effect during a LINK operation. If you find any, you can modify the procedures CXXLINK in one of the following ways:

- Define the logical name without /USER_MODE. This means that the logical name should be deassigned, or its previous value reassigned, after the CXXLINK operation is completed to ensure that prior state is restored. Any ON ERROR cases that may be jumped to if the CXXLINK fails should check for and deassign or reassign the logical name if needed.

- Move the definition(s) into a separate command procedure. CXXLINK checks the logical name CXX$LINK_INIT, and if it is defined, executes the command procedure in its subprocess prior to executing any LINK command.

  Consider the following procedure:

  ```
  $ DEFINE/USER MYLIB MYAREA:MYLIB.OLB
  $ LINK A,B,SYS$INPUT:/OPT
  MYLIB/LIB
  $
  ```

  To have the procedure work with CXXLINK, modify it as follows:

  ```
  $ CREATE CXX$LINK_INIT.COM
  $ DEFINE MYLIB MYAREA:MYLIB.OLB
  $EOD
  $ DEFINE/USER CXX$LINK_INIT SYS$DISK:[]CXX$LINK_INIT.COM
  $ LINK A,B,SYS$INPUT:/OPT
  MYLIB/LIB
  $!
  $ DELETE CXX$LINK_INIT.COM;
  ```

Note that the CXX$LINK_INIT command procedure defines MYLIB without the /USER_MODE qualifier. This is because the command procedure is executed only once in the spawned process.

1.3.3. Linking to the C++ Standard Library

When you compile and link programs that use the C++ Standard Library, no special qualifiers are required. The C++ driver automatically includes the Standard Library run-time support on the link command, and automatic template instantiation is the default mode.

For example, to build a program called `prog.cxx` that uses the Standard Library, you enter the following command:

```
$ CXX prog.cxx
```

For detailed information about the Standard Library, refer to Chapter 7.

1.3.4. Linking to the C++ Class Library

Reusing code is a cornerstone of object-oriented programming. To minimize the time it takes to develop new applications, a set of reusable classes is an essential part of the VSI C++ compiler environment. Class libraries offer a variety of predefined classes that enable you to work more efficiently.

For a detailed explanation of the class library packages supplied with the compiler, see the *VSI C++ Class Library Reference Manual*, CXX_CLASSLIB.PS, in the SYS$COMMON: [SYSHLP.CXX$HELP] directory.

The Class Library has always been provided in shareable image format. Starting with OpenVMS Version 6.2, the Class Library is also provided in object library format.

Using the Class Library as an object library provides a functional advantage over using the shareable image. When your program redefines the global `new` and `delete` operators and uses the Class Library object library, the `new` and `delete` calls within the Class Library are directed to the `new` and `delete` operators defined by your program. On the other hand, when your program uses the Class Library shareable image, the `new` and `delete` calls within the Class Library always call the standard `new` and `delete` operators. Linking with the shareable image is the default method.

When you use the Class Library as a shareable image, the Class Library code resides in an image file in SYS$SHARE and is shared by all C++ programs. This process has the advantages of: reducing the size of a program's executable image, decreasing the amount of disk space taken up by the program's image, and letting your program swap in and out of memory faster because of decreased size.

1.3.4.1. Linking Against the Class Library Object Library

To link against the Class Library object library on OpenVMS Version 6.2 or higher systems, you need to specify the /NOSYSSHR qualifier on your CXXLINK command. For example:

```
$ CXXLINK/NOSYSSHR my_program.obj
```

If your program defines nonlocal static objects whose constructors or destructors use any part of the Class Library, you need to ensure that the Class Library is initialized before your objects' constructors are invoked. (Note that this is not an issue when you link against the Class Library shareable image.) To guarantee this order of initialization, specify the Class Library initialization object module CXXL_INIT as the first module in your CXXLINK command. To do this, use a CXXLINK command similar to the following:

```
$ CXXLINK/NOSYSSHR/EXE=my_program SYS$SHARE:STARLET.OLB -
_$ /INCLUDE=CXXL_INIT, my_program.obj
```

If your program uses the task package, you must explicitly include the CMA shared library when you link /NOSYSSHR, as in the following example:

```
$ CXXLINK/NOSYSSHR my_program.obj,SYS$INPUT:/OPT -
_$ SYS$SHARE:CMA$LIB_SHR/SHARE
^Z
```

1.3.4.2. Linking Against the Class Library Shareable Image

No special action is needed to link with the Class Library; simply specify the object modules and object libraries that you want to link. The linker automatically finds and resolves any references to objects in the Class Library when it searches the system-supplied shareable image library, SYS$LIBRARY:IMAGELIB.OLB.

1.3.5. Linker Command Qualifiers

You can use qualifiers to modify OpenVMS Linker output and to invoke debugging and traceback facilities. The following list shows some of the most useful LINK command qualifiers that you can specify on your CXXLINK command. For a full discussion of the OpenVMS Linker, see the *VSI OpenVMS Linker Utility* manual.

Command Qualifiers	**Defaults**
/BRIEF	None.
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE=first-object-file-name.exe
/FULL	None.
/[NO]MAP	/NOMAP (interactive) /MAP (batch)
/[NO]SHAREABLE	/NOSHAREABLE
/[no]TRACEBACK	/TRACEBACK

1.3.6. Linker Error Messages

If the OpenVMS Linker detects errors while linking object modules, the linker displays messages indicating the cause and severity of error. Because CXXLINK uses the OpenVMS Linker to link your C++ program, CXXLINK displays these linker messages. The linker does not produce an image file if errors or fatal errors occur (conditions with severities of E or F).

Some problems that commonly occur during linking are as follows:

- You try to link a program without defining every function that the program calls.

  The linker responds by issuing warnings. For example:

  `%LINK-W-USEUNDEF` *symbol-name*

  A symbol name that you do not recognize could be a **mangled** name. Name mangling is the mechanism that the compiler uses to encode exceptionally long identifiers, including C++ function names. By default, CXXLINK displays such symbols in their demangled form. To see a symbol in its mangled form, use the /PRELINK=NODEMANGLE qualifier on your CXXLINK command. (See Section 1.6 for more information about name demangling.)

- You try to link a module that had warning or error messages during compilation.

  To avoid unexpected results, verify that the linker's action is acceptable.

- You try to link a nonexistent module.

  Check to see if the module exists (in the directory or library you expect it to be in) and is spelled correctly.

- You redefine a C RTL function, or override the global operators `new` or `delete`. For more information, see the /[NO]PREFIX_LIBRARY_ENTRIES Qualifier in Section 1.2.1.

For an explanation of linker messages, invoke the HELP/MESSAGE utility.

1.4. Linking a Program *(I64 only)*

This section describes how to link a C++ program on OpenVMS I64 systems.

After your program or module successfully compiles, you must use either the CXXLINK facility or OpenVMS Linker to combine your object modules into one executable image.

The CXXLINK facility is layered on the OpenVMS Linker utility and provides the ability to link your C++ application. On I64 systems, the CXXLINK facility accepts the same command qualifiers as CXXLINK on Alpha systems, including the full range of the Linker's command qualifiers that the CXXLINK facility passes to the Linker. For a description of Linker commands, see the *VSI OpenVMS Linker Utility* manual.

On I64 systems, the only benefit of using CXXLINK instead of the Linker is that CXXLINK reports non-mangled names of undefined and multiply-defined symbols. It does this by intercepting Linker diagnostics and converting mangled names reported by the Linker to their original names, using the information in the demangler database.

The demangler database is a file created by the compiler. By default, it is created in a [.CXX_REPOSITORY] subdirectory of the current directory. For both the C++ compiler and CXXLINK, the location of the repository is controlled by the /REPOSITORY qualifier. For CXXLINK to correctly translate mangled names to their original, non-mangled counterparts, it is important to use the same repository for both compiling and linking.

Do not use the Linker CLUSTER= option to reference OpenVMS object modules that define global static objects. Using this option prevents the constructors and destructors for global static objects from being run during image activation and exit.

Caution

The OpenVMS Linker expects a function whose identifier is `main`. If a C++ program lacking a definition of `main` is inadvertently linked, then program execution begins at the first function seen by the linker.

1.4.1. Linking Against C++ Class and Standard Library Shareable Images

On I64 systems, the C++ Class and Standard Library, as well as the language run-time support library, are delivered as system shareable images in SYS$LIBRARY:

CXXL$011_SHR.EXE - class library image
CXXL$RWRTL.EXE - standard library image
CXXL$LANGRTL.EXE - language run-time support image

As system shareable images, these CXXL$ images are part of the system library of shareable images, IMAGELIB.OLB, which is automatically searched by the Linker. Consequently, no special actions are required to link C++ applications against the class or standard library shareable image.

For example, if PROG.CXX uses a class from the C++ class or standard library, the following sequence of commands will compile, link, and run the program:

```
$ CXX PROG.CXX
$ CXXL PROG.OBJ  ! (or LINK PROG.OBJ)
$ RUN PROG.EXE
```

1.4.2. Linking Against the Object Library (Linking /NOSYSSHARE)

In addition to being delivered as system shareable images, the C++ class, standard, and language run-time support libraries are also delivered in object form in the system object library STARLET.OLB, thus making it possible to link C++ applications /NOSYSSHARE.

The C++ libraries themselves do not impose any restrictions on linking /NOSYSSHARE. However, because they are layered on top of the C Run-Time Library, the rules for linking an application that references the C Run-Time Library /NOSYSSHARE do apply.

For example, when linking /NOSYSSHARE, you must explicitly include CMA$TIS routines in the link by either linking against the CMA$TIS_SHR.EXE shareable image or forcing the CMA$TIS module from STARLET.OLB in the link. See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] for more details.

Here are two examples of linking a C++ program /NOSYSSHARE:

```
$ CXXL/NOSYSSHARE PROG.OBJ, SYS$INPUT:/OPT
SYS$SHARE:CMA$TIS_SHR/SHARE
^Z

$ CXXL/NOSYSSHARE prog.obj, -
_$ SYS$SHARE:STARLET.OLB/INCLUDE=CMA$TIS
```

1.5. Running a C++ Program

When your program successfully links, use the DCL RUN command to execute the image file. The RUN command has the following format:

```
RUN [/[NO]DEBUG] file-spec
```

/DEBUG
/NODEBUG

> Determines whether you invoke the OpenVMS Debugger during run time. Use the /DEBUG qualifier to invoke the debugger if your image was not linked with the debugger. However, do not use the /DEBUG qualifier on images linked with the /NOTRACEBACK qualifier. Use the /NODEBUG qualifier if you linked your image with the /DEBUG qualifier and you do not want the debugger to prompt you. The default is RUN/DEBUG if you linked your image with the /DEBUG qualifier; otherwise, the default is RUN/NODEBUG.

For more information on debugging C++ programs, see Chapter 8.

1.5.1. Run-Time Errors

When an error occurs during program execution, the OpenVMS condition handler terminates execution and displays messages and traceback information on the currently defined sys$error device. In the

symbolic stack dump traceback message, the condition handler lists the modules that were active when the error occurred, indicating the sequence in which the modules were called.

Traceback information is available at run time only for modules compiled with /DEBUG=TRACEBACK and linked with the /TRACEBACK qualifier in effect (the default for both compiler and linker commands). You should exclude traceback information only from thoroughly debugged program modules.

The traceback information makes reference to numbered lines that are listing lines in your program. If you include header files in the source file using the `#include` directive, the line numbers do not correspond to the source-file lines. To see the numbers that correspond to those referenced in the traceback information, generate a listing file using the /LIST qualifier to the compiler command.

1.5.2. Passing Arguments to the main Function

The `main` function in a C++ program can accept arguments from the command line from which it was invoked. The syntax for a `main` function is as follows:

```
int main(int argc,
char *argv[ ],
char *envp[ ])
{
. . .
return status;
}
```

In this syntax, parameter *argc* is the count of arguments present in the command line that invoked the program, and parameter *argv* is a character-string array of the arguments. Parameter *envp* is the environment array, which contains process information such as the user name and controlling terminal. Parameter *envp* has no bearing on passing command-line arguments; its primary use in C++ programs is during `exec` and `getenv` function calls. For more information, see the *VSI C Run-Time Library Reference Manual for OpenVMS Systems [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]*.

In the `main` function definition, the parameters are optional. However, you can access only the parameters that you define. You can define the `main` function in any of the following ways:

```
int main()
int main(int argc)
int main(int argc, char *argv[])
int main(int argc, char *argv[], char *envp[])
```

To pass arguments to the `main` function, you can use a DCL foreign command to point to the program, or you can define the logical name DCL$PATH to point to an area containing the program.

To make use of DCL$PATH in the previous example, the resulting program executable would have to be named "echo.exe".

You can then place echo.exe into a specific directory and point the logical name DCL$PATH to it.

For example:

```
$ RENAME commarg.exe echo.exe
$ COPY echo.exe sys$login:
$ DEFINE DCL$PATH SYS$LOGIN:
```

The output would be identical to that shown in the previous example when a foreign command was used. To pass arguments to the `main` function, you must define the program as a DCL foreign command. When a program is defined and run as a foreign command, the parameter `argc` is always greater than or equal to 1, and `argv[0]` always contains the name of the image file.

The procedure for defining a foreign command involves using a DCL assignment statement to assign the name of the image file to a symbol that is later used to invoke the image. For example:

```
$ echo == "$ dsk$:commarg.exe" Return
```

The symbol `echo` is defined as a foreign command that invokes the image in `commarg.exe`. The definition of `echo` must begin with a dollar sign ($) and include a device name.

For more information about the procedure for defining a foreign command, see the *VSI OpenVMS DCL Dictionary*.

The following example shows a C++ program called `commarg.cxx`, which displays the command-line arguments that were used to invoke it:

```
// This program echoes the command-line arguments.

#include <iostream.h>

main(int argc, char *argv[])
{
        int i;
        for (i = 0; i < argc; ++i)
                cout << i << " := >" << argv[i] << "<\n";
        return 0;
}
```

A sample output for this example is as follows:

```
$ echo  Long  "Day's"  "Journey into Night" Return
0 := >db7:commarg.exe;1<
1 := >long<
2 := >Day's<
3 := >Journey into Night<
```

DCL converts unquoted arguments on the command line to uppercase letters. However, the C RTL internally parses the altered command line and puts all unquoted arguments back in lowercase. This makes access to arguments in VSI C++ programs compatible with C++ programs developed on other systems.

All arguments in the command line are delimited by spaces or tabs. Arguments with embedded spaces or tabs must be enclosed in quotation marks (" ").

1.6. Name Demangling

Because of the need to provide type-safe linking, VSI C++ encodes type information in external function names. This encoding is called **name mangling**.

Mangled names can appear in diagnostic messages from commands such as CXXLINK/NOEXPAND or from the OpenVMS Linker utility. To enable users to decode (or demangle) these names, the compiler provides the CXXDEMANGLE facility. The CXXDEMANGLE facility translates mangled names into the names as they originally appeared in C++ source code.

To do the translation, CXXDEMANGLE uses a data file written by the compiler during compilation. The data file contains a mapping of mangled names to their demangled forms.

1.6.1. Creating the Data File

Each time you compile a program, the compiler stores, in a data file, all the program's external symbols in their mangled and demangled forms. If the data file does not exist, the compiler creates the data file. Otherwise, the compiler appends information to the existing data file.

You can specify the name and location of the data file using the logical name CXX$DEMANGLER_DB. For example, if you want your data file to be named MYCXXDB.DAT in the DISK1:[MYDIR] directory, define the CXX$DEMANGLER_DB logical name as follows:

```
$ DEFINE CXX$DEMANGLER_DB DISK1:[MYDIR]MYCXXDB.DAT
```

If the CXX$DEMANGLER_DB logical name is not defined, the compiler uses the default file name CXX$DEMANGLER_DB in the writeable repository. Refer to Chapter 5 for details on how to specify the writeable repository.

1.6.2. Using the CXXDEMANGLE Facility

To demangle a symbol name, CXXDEMANGLE must use the same data file as the compiler used when it compiled the program containing the symbol.

Hence, if you defined the CXX$DEMANGLER_DB logical name when you compiled the program, you should also define the logical name when you use the CXXDEMANGLE facility.

Similarly, if you did not define the CXX$DEMANGLER_DB logical name but specified the /REPOSITORY qualifier during compilation, specify the same /REPOSITORY qualifier on your CXXDEMANGLE command.

If you did not specify the /REPOSITORY qualifier on your compile command, the compiler uses the data file in the default writeable repository. To use the CXXDEMANGLE facility in this case, either issue the CXXDEMANGLE command from the same directory where the compile command was issued, or specify the appropriate /REPOSITORY qualifier on your CXXDEMANGLE command.

CXXDEMANGLE provides both a command-line interface and an interactive interface, as follows:

- To use the command-line interface, enter the CXXDEMANGLE command followed by a comma-separated list of mangled symbol names. CXXDEMANGLE then displays the demangled form of each symbol and exits. The command-line interface has the following syntax:

  ```
  CXXDEMANGLE mangled-symbol-name [,...]
  ```

  The following example shows appropriate use of this syntax:

  ```
  $ CXXDEMANGLE COPY_ _XPIPIPI
  int * copy(int *, int *, int *)
  $ CXXDEMANGLE COPY_ _XPPCPPCPPC, CXX$ADJCNTDFFRNCXPPP9MNS0IUE0NU
  char ** copy(char **, char **, char **)
  int * adjacent_difference(int *, int *, int *, minus<int >)
  $
  ```

  If you specify a mangled symbol name using the command-line interface and the symbol contains lowercase letters, you must place the symbol within quotes. For example:

```
$ CXXDEMANGLE "MyFunction_ _xic"
```

- To use the interactive interface, enter the CXXDEMANGLE command without specifying a symbol name. CXXDEMANGLE then waits for you to enter a symbol name in its mangled form. When you enter a symbol, CXXDEMANGLE displays the demangled form of the symbol and waits for you to enter another symbol, and so forth. To exit the interactive interface, enter Ctrl/Z. The syntax for the interactive interface is as following:

  ```
  CXXDEMANGLE
  mangled-symbol-name
  [...]
  Ctrl/Z
  ```

  The following example shows appropriate use of this syntax:

  ```
  $ CXXDEMANGLE
  COPY_ _XPIPIPI
  int * copy(int *, int *, int *)
  COPY_ _XPPCPPCPPC
  char ** copy(char **, char **, char **)
  CXX$ADJCNTDFFRNCXPPP9MNS0IUE0NU
  int * adjacent_difference(int *, int *, int *, minus<int >)
  Ctrl/Z
  $
  ```

  When you use the interactive interface, quotes are not necessary when entering mangled symbol names that contain lowercase letters.

If CXXDEMANGLE is unable to translate a mangled symbol name, it echoes the mangled symbol name.

1.6.2.1. Command Qualifier

The CXXDEMANGLE command accepts a single qualifier, /REPOSITORY.

`/REPOSITORY=(repository[,...])`

Names the repository directories that contain the data files used by CXXDEMANGLE.
The /REPOSITORY qualifier is ignored if you define the CXX$DEMANGLER_DB logical name. See the preceding text for details.

1.7. Performance Optimization Qualifiers

The following compiler qualifiers can be used to improve performance. However, they can also change behavior for nonstandard-compliant programs:

- /[NO]ANSI_ALIAS – Specifies whether the compiler assumes the ANSI C aliasing rules to generate better optimized code. The default is /ANSI_ALIAS.
- /ASSUME=[NO]POINTERS_TO_GLOBALS – Controls whether the compiler can safely assume that global variables have not had their addresses taken in code that is not visible to the current compilation. The default is /ASSUME=POINTERS_TO_GLOBALS.
- /ASSUME=[NO]TRUSTED_SHORT_ALIGNMENT – Allows the compiler additional assumptions about the alignment of `short` types that, although naturally aligned, may cross a quadword boundary. The default is /ASSUME=NOTRUSTED_SHORT_ALIGNMENT.

- /ASSUME=[NO]WHOLE_PROGRAM – Tells the compiler that except for “well-behaved library routines”, the whole program consists only of the single object module being produced by this compilation. The optimizations enabled by /ASSUME=WHOLE_PROGRAM include all those enabled by /ASSUME=NOPOINTER_TO_GLOBALS and possibly other optimizations. The default is /ASSUME=NOWHOLE_PROGRAM.

You can use the /OPTIMIZE qualifier to improve performance. This qualifier will not change application behavior.

On I64 systems, the floating-point formats D_FLOAT, G_FLOAT, and F_FLOAT are emulated using IEEE_FLOAT. Because this can hinder performance, using the /FLOAT=IEEE_FLOAT default is recommended.

See Appendix A for detailed descriptions of these qualifiers.

1.8. Improving Build Performance

Partitioning a large application into several shared libraries, which are then linked into an executable, is a useful technique for reducing link times during development. See Section 3.5 for more information.

1.9. Deploying Your Application

The VSI C++ kit contains two Run-Time Library components that you might need to redistribute with your applications:

- C++ Standard Library Object Library (LIBCXXSTD)
- C Run-Time Object Library (DECC$CRTL.OLB)

The next sections describe the method that developers must use to redistribute Run-Time Library components to user systems. Redistribution of other components on the VSI C++ kit is prohibited. The redistribution rights set forth in the Software Product Description do not apply to the DECC$CRTL.EXE or DECC$CRTL.README files which are distributed with this kit.

1.9.1. Redistribution of the DECC$CRTL.OLB Object Library

Redistribution of this library is only required by those applications which need to be linked during or after installation on an end user target system. If you link your application and ship either a shareable or executable image to your customers, then redistribution of the object library is not necessary. The linking process of your application causes those library modules to be incorporated into your resultant image.

There are two options that you can use to redistribute the DECC$CRTL.OLB object library. The options are based on whether the library is needed after the installation is completed.

The first option is for applications which link during installation, but have no need for the object library once installation is completed. For that set of developers, we recommend placing DECC$CRTL.OLB on your kit, but to link using the copy in VMI$KWD and not issue a PROVIDE_FILE option which would move this file onto the system. In other words, the object library resides only on your kit, is used during installation to link your application, but is not placed onto the end user system.

The second option is for applications which do need the object library after installation is completed. For this class of applications, the object library should be placed in a product specific location on the

target system and not in SYS$LIBRARY. The contents of this object library must not be inserted into the SYS$LIBRARY:STARLET.OLB library.

1.9.2. Redistribution of the LIBCXXSTD.OLB Object Library

Redistribution of this library is only required by those applications which need to be linked during or after installation on an end user target system. If you link your application and ship either a shareable or executable image to your customers, then redistribution of the object library is not necessary. The linking process of your application causes those library modules to be incorporated into your resultant image.

There are two options that you can be used to redistribute the LIBCXXSTD.OLB object library. The options are based on whether the library is needed after the installation is completed.

The first option is for applications which link during installation, but have no need for the object library once installation is completed. For that set of developers, we recommend placing LIBCXXSTD.OLB on your kit, but to link using the copy in VMI$KWD and not issue a PROVIDE_FILE option which would move this file onto the system. In other words, the object library resides only on your kit, is used during installation to link your application, but is not placed onto the end user system.

The second option is for applications that do need the object library after installation is completed. For this class of applications, the object library should be placed in a product specific location on the target system and not in SYS$LIBRARY. The contents of this object library must not be inserted into the SYS$LIBRARY:STARLET.OLB library.

Chapter 2. VSI C++ Implementation

This chapter discusses the features and characteristics specific to the VSI C++ implementation, including pragmas, predefined names, numerical limits, and other implementation-dependent aspects of the language definition.

2.1. Implementation-Specific Attributes

This section describes pragmas, predefined names, and limits placed on the number of characters and arguments used in C++ programs.

2.1.1. #pragma Preprocessor Directive

The `#pragma` preprocessor directive is a standard method for implementing features that differ from one compiler to the next. This section describes pragmas specifically implemented in the C++ compiler for OpenVMS systems.

The following `#pragma` directives are subject to macro expansion. A macro reference can occur anywhere after the `pragma` keyword.

builtins	inline	linkage[1]	use_linkage[1]
dictionary	noinline	module	extern_model
member_alignment	message	define_template	extern_prefix

[1]Not supported; specific to C

This manual displays keywords used with `#pragma` in lowercase letters. However, these keywords are not case sensitive.

2.1.1.1. #pragma [no]builtins

The `#pragma builtins` directive enables the C++ built-in functions that directly access processor instructions. If the pragma does not appear in your program, the default is `#pragma nobuiltins`.

C++ supports the `#pragma builtins` preprocessor directive for compatibility with VAX C, but it is not required.

2.1.1.2. #pragma define_template Directive

The `#pragma define_template` directive instructs the compiler to instantiate a template with the arguments specified in the pragma. This pragma has the following syntax:

```
#pragma define_template identifier
```

For example, the following statement instructs the compiler to instantiate the template `mytempl` with the arguments `arg1` and `arg2`:

```
#pragma define_template mytempl<arg1, arg2>
```

For more information on how to use templates with the `#pragma define_template` directive, see Section 5.2.

2.1.1.3. #pragma environment Directive

The `#pragma environment` directive offers a way to single-handedly set, save, or restore the states of *context* pragmas. This directive protects include files from contexts set by encompassing programs and protects encompassing programs from contexts that could be set in header files that the encompassing programs include.

On OpenVMS systems, the `#pragma environment` directive affects the following pragmas:

```
#pragma member_alignment
#pragma message
#pragma extern_model
#pragma extern_prefix
```

This pragma has the following syntax:

```
#pragma environment command_line
#pragma environment header_defaults
#pragma environment restore
#pragma environment save
```

`command_line`

Sets, as specified on the command line, the states of all the context pragmas. You can use this pragma to protect header files from environment pragmas that take effect before the header file is included.

`header_defaults`

Sets the states of all the context pragmas to their default values. This is almost equivalent to the situation in which a program with no command line options and no pragmas is compiled; except that this pragma sets the pragma message state to `#pragma nostandard`, as is appropriate for header files.

`save`

Saves the current state of every pragma that has an associated context.

`restore`

Restores the current state of every pragma that has an associated context.

Without requiring further changes to the source code, you can use `#pragma environment` to protect header files from things like language extensions and enhancements that might introduce additional contexts.

A header file can selectively inherit the state of a pragma from the including file and then use additional pragmas as needed to set the compilation to non-default states. For example:

```
#ifdef __PRAGMA_ENVIRONMENT
#pragma __environment save  ❶
#pragma __environment header_defaults ❷
#pragma member_alignment restore ❸
#pragma member_alignment save ❹
#endif
.
.  /* contents of header file */
.
#ifdef __PRAGMA_ENVIRONMENT
#pragma __environment restore
#endif
```

In this example:

❶ Saves the state of all context pragmas
❷ Sets the default compilation environment
❸ Pops the member alignment context from the `#pragma member_alignment` stack that was pushed by `#pragma __environment save` (restoring the member alignment context to its preexisting state)
❹ Pushes the member alignment context back onto the stack so that the `#pragma __environment restore` can pop the entry off

Thus, the header file is protected from all pragmas, except for the member alignment context that the header file was meant to inherit.

2.1.1.4. #pragma extern_model Directive

The `#pragma extern_model` directive controls the compiler's interpretation of data objects that have external linkage. You can use this pragma to select the global symbol model to use for `extern`s. The default is the relaxed refdef model.

Note

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 may produce unintended behavior.

After you select a global symbol model with `#pragma extern_model`, the compiler treats all subsequent declarations of objects of the `extern` storage class accordingly, until it encounters another `#pragma extern_model` directive.

The global symbol models are as follows:

- Common block model

  In this model, all declarations are definitions and the linker combines all definitions with the same name into one definition. For Fortran program units, such `extern` variables appear as named COMMON blocks. The syntax is as follows:

  ```
  #pragma extern_model common_block [(no)shr]
  ```

  The `shr` and `noshr` keywords determine whether the psects created for definitions are marked as shared or not shared. Fortran COMMON blocks normally have the shared attribute. If neither keyword is specified, the pragma acts as if `noshr` was specified.

  #### Note

  The C language permits objects declared with the `const` type qualifier to be allocated in read-only memory, and when the C compiler allocates a psect for a `const` object, it marks that section as read-only.

This is not compatible with the C++ conventions because the C++ language permits objects with static storage duration to be initialized with values computed at run-time (before the main function gains control). When the C++ compiler allocates a psect for such a declaration, it marks the psect writable. Normally, only one compilation (the one responsible for initialization) will allocate a psect for a `const` object, and there is no problem.

But under the `common_block` extern model, the compilers will always allocate a psect for such a declaration, leading to a "conflicting attributes" warning from the linker if the same `const`-qualified declaration is processed by both C and C++. It is best to avoid use of the `common_block` extern model when `const` objects with external linkage are shared between C and C++. If the `common_block` model must be used, then the `const` type qualifier should be removed (for example, by preprocessor conditionals) from the declaration processed by the C compiler.

- Relaxed refdef model

  This model is the default. Some declarations are references and some are definitions. Multiple uninitialized definitions for the same object are allowed and resolved into one by the linker. However, a reference requires that at least one definition exists. The syntax is as follows:

  ```
  #pragma extern_model relaxed_refdef [(no)shr]
  ```

  The `shr` and `noshr` keywords determine whether the psects created for definitions are marked as shared or not shared. If neither keyword is specified, the pragma acts as if `noshr` was specified.

- Strict refdef model

  In this model, some declarations are references and some are definitions. It requires exactly one definition in the program for each symbol referenced. The syntax is as follows:

  ```
  #pragma extern_model strict_refdef ["name"] [(no)shr]
  ```

  If specified, *name* in quotes is the name of the psect for any definition.

  `shr` and `noshr` keywords determine whether the psects created for definitions are marked as shared or not shared. Neither keyword can be specified unless a name for the psect is given. If neither keyword is specified, the pragma acts as if `noshr` was specified.

- Globalvalue model

  This model is like the strict refdef model except that these global objects have no storage; instead, global objects are link-time constant values. The syntax is as follows:

  ```
  #pragma extern_model globalvalue
  ```

- Save

  This pragma pushes the current `extern` model of the compiler onto a stack. The stack records all the information associated with the `extern` model. The syntax is as follows:

  ```
  #pragma extern_model save
  ```

- Restore

  This pragma pops the `extern` model stack of the compiler. The compiler's `extern` model is set as the state just popped off the stack. The stack records all the information associated with the `extern` model. The syntax is as follows:

```
#pragma extern_model restore
```

Note

- The global symbols and psect names generated under the control of this pragma obey the case-folding rules of the /NAME qualifier.
- While `#pragma extern_model` can be used to allocate several variables in the same psect, the placement of variables relative to each other within that psect cannot be controlled: the compiler does not necessarily allocate distinct variables to memory locations according to the order of appearance in the source code.

  Furthermore, the order of allocation can change as a result of seemingly unrelated changes to the source code, command-line options, or from one version of the compiler to the next; it is essentially unpredictable. The only way to control the placement of variables relative to each other is to make them members of the same `struct` type or, on OpenVMS Alpha systems, by using the `noreorder` attribute on a named `#pragma extern_model strict_refdef`.

The `#pragma extern_model` directive has the following syntax:

`#pragma extern_model` *`model_spec`* `[`*`attr`*`[,`*`attr`*`]...]`

model_spec is one of the following:

`common_block`
`relaxed_refdef`
`strict_refdef` "*name*"
`strict_refdef` (No *attr* specifications allowed)
`globalvalue` (No *attr* specifications allowed)

[*attr*[,*attr*]...] are optional psect attribute specifications chosen from the following (at most one from each line):

`gbl lcl` (Not allowed with `relaxed_refdef`)
`shr noshr`
`wrt nowrt`
`pic nopic` (Not meaningful for Alpha)
`ovr con`
`rel abs`
`exe noexe`
`vec novec`
For OpenVMS Alpha and Integrity systems: `0 byte 1 word 2 long 3 quad 4 octa 5 6 7 8 9 10 11 12 13 14 15 16 page`
For OpenVMS VAX systems: `2 long 3 quad 4 octa 9 page`

The last line of attributes are numeric alignment values. When a numeric alignment value is specified on a section, the section is given an alignment of two raised to that power.

On OpenVMS Alpha and Integrity systems, the `strict_refdef` "*name*" extern_model can also take the following psect attribute specifications:

- `noreorder` — causes variables in the section to be allocated in the order they are defined.

- `natalgn` — has no effect on OpenVMS systems.

  It does, however, change the behavior on UNIX systems: when specified, `natalgn` causes the global variables defined within the section to be allocated on their natural boundary. Currently, all global variables on UNIX systems are allocated on a quadword boundary. When the `natalgn` attribute is specified, the compiler instead allocates the variable on an alignment that is natural for its type (`char`s on byte boundaries, `int`s on longword boundaries, and so on).

  Specifying the `natalgn` attribute also enables the `noreorder` attribute.

  Note

  Use of the `natalgn` attribute can cause a program to violate the UNIX Calling Standard. The calling standard states that all global variables must be aligned on a quadword boundary. Therefore, variables declared in a `natalgn` section should only be referenced in the module that defines them.

Table 2.1 lists the attributes that can be applied to program sections.

Table 2.1. Program-Section Attributes

Attribute	Meaning
PIC or NOPIC	The program section or the data these attributes refers to does not depend on any specific virtual memory location (PIC), or else the program section depends on one or more virtual memory locations (NOPIC).
CON or OVR	The program section is concatenated with other program sections with the same name (CON) or overlaid on the same memory locations (OVR).
REL or ABS	The data in the program section can be relocated within virtual memory (REL) or is not considered in the allocation of virtual memory (ABS).
GBL or LCL	The program section is part of one cluster, is referenced by the same program section name in different clusters (GBL), or is local to each cluster in which its name appears (LCL).
EXE or NOEXE	The program section contains executable code (EXE) or does not contain executable code (NOEXE).
WRT or NOWRT	The program section contains data that can be modified (WRT) or data that cannot be modified (NOWRT).
RD or NORD	These attributes are reserved for future use.
SHR or NOSHR	The program section can be shared in memory (SHR) or cannot be shared in memory (NOSHR).
USR or LIB	These attributes are reserved for future use.
VEC or NOVEC	The program section contains privileged change mode vectors (VEC) or does not contain those vectors (NOVEC).
COM or NOCOM	The program section is a conditionally defined psect associated with a conditionally defined symbol. This is the type of psect created when you declare an uninitialized definition with `extern_model relaxed_refdef`.

See the *VSI OpenVMS Linker Utility Manual* for more complete information on each.

The default attributes are: `noshr`, `rel`, `noexe`, `novec`, `nopic`.

For `strict_refdef`, the default is `con`. For `common_block` and `relaxed_refdef`, the default is `ovr`.

The default for `wrt`/`nowrt` is determined by the first variable placed in the psect. If the variable has the `const` type qualifier (or the `readonly` modifier), the psect is set to `nowrt`. Otherwise, it is set to `wrt`.

Restrictions on Setting Psect Attributes

Be aware of the following restriction on setting psect attributes.

The `#pragma extern_model` directive does not set psect attributes for variables declared as tentative definitions in the `relaxed_refdef model`. A tentative definition is one that does not contain an initializer. For example, consider the following code:

```
#pragma extern_model relaxed_refdef long
int a;
int b = 6;
#pragma extern_model common_block long
int c;
```

Psect A is given octaword alignment (the default) because `a` is a tentative definition. Psect B is correctly given longword alignment because it is initialized and is, therefore, not a tentative definition. Psect C is also given longword alignment because it is declared in an `extern_model` other than `relaxed_refdef`.

Note

The psect attributes are normally used by system programmers who need to perform declarations normally done in macro. Most of these attributes are not needed in normal C programs. Also, notice that the setting of attributes is supported only through the `#pragma` mechanism, and not through the /EXTERN_MODEL command-line qualifier.

2.1.1.5. #pragma extern_prefix Directive

The `#pragma extern_prefix` directive controls the compiler's synthesis of external names, which the linker uses to resolve external name requests. When you specify `#pragma extern_prefix` with a string argument, the compiler prepends the string to all external names produced by the declarations that follow the pragma specification.

This pragma is useful for creating libraries where the facility code can be attached to the external names in the library.

The syntax is as follows:

```
                      "string"
#pragma extern_prefix save
                      restore
```

`"string"`
Prepends the quoted string to external names in the declarations that follow the pragma specification.

`save`
Saves the current pragma prefix string.

`restore`

Restores the saved pragma prefix string.

The default external prefix, when none has been specified by a pragma, is the null string. The recommended use is as follows:

```
#pragma extern_prefix save
#pragma extern_prefix "prefix-to-prepend-to-external-names"
... some declarations and definitions ...
#pragma extern_prefix restore
```

When an `extern_prefix` is in effect and you are using `#include` to include header files, but do not want the `extern_prefix` to apply to `extern` declarations in the header files, use the following code sequence:

```
#pragma extern_prefix save
#pragma extern_prefix ""
#include …
#pragma extern_prefix restore
```

Otherwise, the external identifiers for definitions in the included files will be prepended with the external prefix.

All external names prefixed with a nonnull string using `#pragma extern_prefix` are converted to uppercase letters regardless of the setting of the /NAMES qualifier.

The compiler treats `#pragma extern_prefix` independently of the /PREFIX_LIBRARY_ENTRIES qualifier. The /PREFIX_LIBRARY_ENTRIES qualifier affects only ANSI and C Run-Time Library (RTL) entries; the `extern_prefix` pragma affects external identifiers for any externally visible name.

2.1.1.6. #pragma function Directive

The `#pragma function` directive specifies that calls to the specified functions will occur in the source code. You normally use this directive in conjunction with `#pragma intrinsic`, which affects specified functions that follow the pragma directive. The effect of `#pragma intrinsic` on a given function continues to the end of the source file or until a `#pragma function` directive occurs, specifying that function.

The `#pragma function` directive has the following syntax:

```
#pragma function (function1[,function2, ...])
#pragma function ()
```

You cannot specify this pragma with empty parentheses. To disable all intrinsic functions, specify the /OPTIMIZE=NOINTRINSICS qualifier on the command line.

2.1.1.7. #pragma include_directory Directive

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 syntax:

```
#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.

2.1.1.8. #pragma [no]inline Directive

The `#pragma inline` directive expands function calls inline. The function call is replaced with the function code itself.

The `#pragma inline` directive has the following syntax:

```
#pragma inline (id,...)
#pragma noinline (id,...)
```

If a function is named in an inline directive, calls to that function will be expanded as inline code, if possible.

If a function is named in a noinline directive, calls to that function will not be expanded as inline code.

If a function is named in both an inline and a noinline directive, an error message is issued.

For calls to functions named in neither an inline nor a noinline directive, C++ expands the function as inline code whenever appropriate as determined by a platform-specific algorithm.

2.1.1.9. #pragma intrinsic Directive

The `#pragma intrinsic` directive specifies that calls to the specified functions are intrinsic. Intrinsic functions are functions in which the compiler generates optimize code in certain situations, possibly avoiding a function call.

The `#pragma intrinsic` directive has the following syntax:

```
#pragma intrinsic (function1[,function2, ...])
```

You can use this directive to make intrinsic the default form of functions that have intrinsic forms. The following functions have intrinsic forms:

```
abs
fabs
labs
alloca
```

You can use the `#pragma function` directive to override the `#pragma intrinsic` directive for specified functions.

The function must have a declaration visible at the time the compiler encounters the `#pragma intrinsic` directive. The compiler takes no action if the compiler does not recognize the specified function name as an intrinsic.

2.1.1.10. #pragma [no]member_alignment Directive

By default, the compiler for OpenVMS systems aligns structure members so that members are stored on the next boundary appropriate to the type of the member; that is, bytes are on the next byte boundary, words are on the next word boundary, and so on.

You can use the `#pragma member_alignment` directive to specify structure member alignment explicitly.

Using `#pragma nomember_alignment` causes the compiler to align structure members on the next byte boundary regardless of the type of the member. The only exception to this is for bit-field members.

This pragma has the following formats:

```
#pragma member_alignment
#pragma member_alignment save
#pragma member_alignment restore
#pragma nomember_alignment [base_alignment]
```

When `#pragma member_alignment` is used, the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For example, a `long` variable is aligned on the next longword boundary; a `short` variable is aligned on the next word boundary.

Consider the following example:

```
#pragma nomember_alignment

struct x {
           char c;
           int b;
           };

#pragma member_alignment

struct y {
          char c;       /*3 bytes of filler follow c */
          int b;
          };

main ()

{
        printf( "The sizeof y is: %d\n", sizeof (struct y) );
        printf( "The sizeof x is: %d\n", sizeof (struct x) );
}
```

When this example is executed, it shows the difference between `#pragma member_alignment` and `#pragma nomember_alignment`.

Once used, the `member_alignment` pragma remains in effect until the `nomember_alignment` pragma is encountered; the reverse is also true.

The optional *base_alignment* parameter can be used to specify the base-alignment of the structure. Use one of the following keywords for the *base_alignment*:

- `byte` (1 byte)
- `word` (2 bytes)
- `longword` (4 bytes)
- `quadword` (8 bytes)
- `octaword` (16 bytes)

The `#pragma member_alignment save` and `#pragma member_alignment restore` directives can be used to save the current state of the `member_alignment` and to restore the previous state, respectively. This feature is necessary for writing header files that require `member_alignment` or `nomember_alignment`, or that require inclusion in a `member_alignment` that is already set.

To affect the member alignment of the entire module, use the /MEMBER_ALIGNMENT qualifier. For information about this qualifier, see Section 2.2.15.1.

2.1.1.11. #pragma message Directive

The `#pragma message` directive controls the kinds of individual diagnostic messages or groups of messages that the compiler issues. Use this pragma to override any command-line options specified by the /WARNINGS qualifier, which affects the types of messages the compiler issues.

Default severities used by the compiler can be changed only if they are informationals, warnings, or discretionary errors. Attempts to change more severe severities are ignored. If a message severity has not been altered by the command line and is not currently being controlled by a pragma, the compiler checks to see whether the message severity should be changed because of the “quiet” state. If not, the message is issued using the default severity.

Error message severities start out with command-line severities applied to default compiler severities. Pragma message severities are then applied. In general, pragma severities override command-line severities, which override default severities. The single exception to this is that command-line options can always be used to downgrade messages. However, command-line qualifiers cannnot be used to raise the severity of messages currently controlled by pragmas.

The `#pragma message` directive has the following syntax:

```
#pragma message disable (message-list)
#pragma message enable (message-list)
#pragma message error (message-list)
#pragma message fatal (message-list)
#pragma message informational (message-list)
#pragma message warning (message-list)
#pragma message restore
#pragma message save
```

disable

Suppresses the compiler-issued messages specified in the *message-list* argument. The *message-list* argument can be any one of the following:

- A single message identifier

- The keyword ALL (all messages issued by the compiler)
- A single message identifier enclosed in parentheses
- A comma-separated list of message identifiers enclosed in parentheses

A message identifier is the name immediately following the message severity code letter. For example, consider the following message:

```
%CXX-W-MISSINGRETURN, Non-void function "name" does not contain a return
 statement
```

The message identifier is MISSINGRETURN. To prevent the compiler from issuing this message, use the following directive:

```
#pragma message disable MISSINGRETURN
```

The compiler lets you disable a discretionary message if its severity is warning (W), informational (I), or error (E) at the time the message is issued. If the message has severity of fatal (F), the compiler issues it regardless of instructions not to issue messages.

`enable`
Enables the compiler to issue the messages specified in the *message-list* argument.

`errors`
Sets the severity of each message in the message list to Error.

`fatals`
Sets the severity of each message in the message list to Fatal.

`informationals`
Sets the severity of each message in the message list to Informational.

`warnings`
Sets the severity of each message in the message list to Warning.

`restore`
Restores the saved state of enabling or disabling compiler messages.

`save`
Saves the current state of enabling or disabling compiler messages.

The `save` and `restore` options are useful primarily within header files. See Section 2.1.1.5.

`#pragma message` performs macro expansion so that you map Version 5.6 message tags to Version 6.0 tags:

```
...
#if __DECCXX_VER > 60000000
#define uninit used_before_set
#endif

#pragma message disable uninit
int main()
{
  int i,j;

  i=j;
}
#pragma message enable uninit
```

2.1.1.12. #pragma module Directive

When you compile source files to create an object file, the compiler assigns the first of the file names specified in the compilation unit to the name of the object file. The compiler adds the .OBJ file extension to the object file. Internally, the OpenVMS system (the debugger and the librarian) recognizes the object module by the file name; the compiler also gives the module a version number of 1. For example, given the object file EXAMPLE.OBJ, the debugger recognizes the EXAMPLE object module.

To change the system-recognized module name and version number, use the #pragma module directive.

You can find the module name and the module version number listed in the compiler listing file and the linker load map.

The `#pragma module` directive is equivalent to the VAX C compatible `#module` directive.

The `#pragma module` directive has the following syntax:

```
#pragma module identifier identifier
#pragma module identifier string
```

The first parameter must be a valid identifier, which specifies the name of the module to be used by the linker. The second parameter specifies the optional identification that appears on the listing and in the object file. The second parameter must be a valid identifier of no more than 31 characters, or a character-string constant of no more than 31 characters.

2.1.1.13. #pragma once Directive

The `#pragma once` preprocessor directive specifies that the header file is evaluated only once.

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

```
#pragma once
```

2.1.1.14. #pragma pack Directive

The `#pragma pack` directive specifies the byte boundary for packing member's structures.

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

```
#pragma pack [(n)]
#pragma pack(push {, identifier} {, n})
#pragma pack(pop  {, identifier} {, n})
```

n specifies the new alignment restriction in bytes as follows:

1	Align to byte
2	Align to word
4	Align to longword
8	Align to quadword
16	Align to octaword

A structure member is aligned to either the alignment specified by `#pragma pack` or the alignment determined by the size of the structure member, whichever is smaller. For example, a short variable in a structure gets byte-aligned if `#pragma pack (1)` is specified. If `#pragma pack (2)` `(4)`, or `(8)` is specified, the short variable in the structure gets aligned to word.

If `#pragma pack` is not used, or if *n* is omitted, packing defaults to 1 for byte alignment.

With the push/pop syntax of this pragma, you can save and restore packing alignment values across program components. This allows you to combine components into a single translation unit even if they specify different packing alignments:

- Every occurrence of `pragma pack` with a `push` argument stores the current packing alignment value on an internal compiler stack. If you provide a value for *n*, that value becomes the new packing value. If you specify an *identifier*, a name of your choosing, it is associated with the new packing value.

- Every occurrence of a `pragma pack` with a `pop` argument retrieves the value at the top of the stack and makes that value the new packing alignment. If an empty stack is popped, the alignment value defaults to the /[NO]MEMBER_ALIGNMENT command-line setting, and a warning is issued. If you specify a value for *n*, that value becomes the new packing value.

  If you specify an *identifier*, all values stored on the stack are removed from the stack until a matching *identifier* is found. The packing value associated with the *identifier* is also removed from the stack, and the packing value that was in effect just before the *identifier* was pushed becomes the new packing value. If no matching *identifier* is found, the packing value reverts to the command-line setting, and a warning is issued.

The push/pop syntax of `pragma pack` lets you write header files that ensure that packing values are the same before and after the header file is encountered. Consider the following example:

```
// File name: myinclude.h
//
#pragma pack( push, enter_myinclude )
// Your include-file code ...
#pragma pack( pop, enter_myinclude )
// End of myinclude.h
```

In this example, the current packing value is associated with the identifier `enter_myinclude` and pushed on entry to the header file. Your include code is processed. The `#pragma pack` at the end of the header file then removes all intervening packing values that might have occurred in the header file, as well as the packing value associated with `enter_myinclude`, thereby preserving the same packing value after the header file as before it.

This syntax also lets you include header files that might set packing alignments different from the ones set in your code. Consider the following example:

```
#pragma pack( push, before_myinclude )
#include <myinclude.h>
#pragma pack( pop, before_myinclude )
```

In this example, your code is protected from any changes to the packing value that might occur in `<myinclude.h>` by saving the current packing alignment value, processing the include file (which may leave the packing alignment with an unknown setting), and restoring the original packing value.

2.1.1.15. #pragma unroll Directive *(Alpha only)*

The `#pragma unroll` preprocessor directive unrolls the `for` loop that follows it by the number of times specified in *unroll_factor*. The `#pragma unroll` directive must be followed by a `for` statement.

This directive has the following format:

```
#pragma unroll unroll_factor
```

The *unroll_factor* is an integer constant in the range of 0 to 255. If a value of 0 is specified, the compiler ignores the directive and determines the number of times to unroll the loop in its normal way. A value of 1 prevents the loop from being unrolled. The directive applies only to the `for` loop that follows it, not to any subsequent `for` loops.

2.1.1.16. #pragma [no]standard Directive

This directive performs operations similar to the save and restore options on `#pragma message` directive:

- `#pragma standard` is the same as `#pragma message restore`.
- `#pragma nostandard` disables all optional messages after doing a `#pragma message save` operation.

2.1.2. Predefined Macros and Names

The compiler defines the following predefined macros and predefined names. For information on using predefined macros in header files in the common language environment, see Section 3.2.

Table 2.2. Predefined Macros

Macro	Description
`_BOOL_EXISTS`	Indicates that `bool` is a type or keyword
`__BOOL_IS_A_RESERVED_WORD`	Indicates that `bool` is a keyword
`__DATE__`[1]	A string literal containing the date of the translation in the form `Mmm dd yyyy`, or `Mmm d yyyy` if the value of the date is less than 10
`__FILE__`[1]	A string literal containing the name of the source file being compiled
`__IEEE_FLOAT`	Identifies floating-point format for compiling the program. The default value is 1 for OpenVMS I64 systems, and 0 for OpenVMS Alpha and VAX systems.

Macro	Description
__LINE__[1]	A decimal constant containing the current line number in the C++ source file
__PRAGMA_ENVIRONMENT	Indicates that that the pragma environment directive is supported.
__TIME__[1]	A string literal containing the time of the translation in the form of hh:mm:ss
_WCHAR_T	Indicates that wchar_t is a keyword

[1]Cannot be redefined or undefined

Table 2.3 lists names with a defined value of 1.

Table 2.3. Names with a Defined Value of 1

Name	Description
__cplusplus[1]	Language identification name.
__DECCXX	Language identification name.
__VMS	System identification
__vms	System identification

[1]Cannot be redefined or undefined

The compiler predefines __VMS; the C compiler predefines VMS and __VMS. Therefore, C++ programmers who plan to reuse code should check for __VMS.

On OpenVMS systems, the compiler supports the following predefined macro names.

Table 2.4. Predefined Macros Specific to OpenVMS Systems

Name	Description
__Alpha_AXP	System identification name
__ALPHA	System identification name
__alpha	System identification name
__32BITS	Defined when pointers and data of type long are 32 bits on Alpha platforms

The compiler predefines __32BITS when pointers and data of type long are 32 bits on Alpha platforms.

On both UNIX and OpenVMS operating systems, programmers should use the predefined macro __alpha for code that is intended to be portable from one system to the other.

On OpenVMS I64 systems, the compiler supports the following predefined macro names:

Table 2.5. Predefined Macros Specific to OpenVMS I64 Systems

Name	Description
__ia64	System identification name
__ia64__	System identification name
__32BITS	Defined when pointers and data of type long are 32 bits.

Predefined macros (with the exception of `vms_version`, `VMS_VERSION`, `__vms_version`, `__VMS_VERSION`, and `__INITIAL_POINTER_SIZE`) are defined as 1 or 0, depending on the system (VAX or Alpha processor), the compiler defaults, and the qualifiers used. For example, if you compiled using `G_FLOAT` format, `__D_FLOAT` and `__IEEE_FLOAT` (Alpha processors only) are predefined to be 0, and `__G_FLOAT` is predefined as if the following were included before every compilation unit:

```
#define  __G_FLOAT  1
```

These macros can assist in writing code that executes conditionally. They can be used in `#elif`, `#if`, `#ifdef`, and `#ifndef` directives to separate portable and nonportable code in a C++ program. The `vms_version`, `VMS_VERSION`, `__vms_version`, and `__VMS_VERSION` macros are defined with the value of the OpenVMS version on which you are running (for example, Version 6.0).

C++ automatically defines the following macros pertaining to the format of floating-point variables. You can use them to identify the format with which you are compiling your program.

__D_FLOAT
__G_FLOAT
__IEEE_FLOAT
_IEEE_FP
__X_FLOAT

The value of __X_FLOAT can be 0 or 1 depending on the floating point mode in effect. You can use the /FLOAT qualifier to change the mode.

Table 2.6 lists predefined version string and version number macros.

Table 2.6. Version String and Version Number Macros

Name	Description
`__VMS_VERSION`[1]	Version identification
`__vms_version`[1]	Version identification
`__DECCXX_VER`[2]	Version identification
`__VMS_VER`[2]	Version identification

[1]The value is a character string.

[2]The value is an unsigned long int that encodes the version number.

For example, the defined value of `__VMS_VERSION` on OpenVMS Version 6.1 is character string `V6.1`.

You can use `__DECCXX_VER` to test that the current compiler version is newer than a particular version and `__VMS_VER` to test that the current OpenVMS Version is newer than a particular version. Newer versions of the compiler and the OpenVMS operating system always have larger values for these macros. If for any reason the version cannot be analyzed by the compiler, then the corresponding predefined macro is defined but has the value of 0. Releases of the compiler prior to Version 5.0 do not define these macros, so you can distinguish earlier compiler versions by checking to determine if the `__DECCXX_VER` macro is defined.

The following example tests for C++ `5.1` or higher:

```
#ifdef __DECCXX_VER
    #if __DECCXX_VER >= 50100000
        / *Code */
    #endif
#endif
```

The following tests for OpenVMS Version 6.2 or higher:

```
#ifdef __VMS_VER
    #if __VMS_VER >= 60200000
        /* code */
    #endif
#endif
```

Table 2.7 shows the macro names for the listed command-line options.

Table 2.7. Macros Defined by Command-Line Qualifiers

Command-line Option	Macro Name
/ALTERNATIVE_TOKENS	__ALTERNATIVE_TOKENS
/ASSUME=GLOBAL_ARRAY_NEW	__GLOBAL_ARRAY_NEW
/ASSUME=STDNEW	__STDNEW
/DEFINE=__FORCE_INSTANTATIONS *(Alpha only)*	__FORCE_INSTANTIATIONS
/EXCEPTIONS	__EXCEPTIONS
/IEEE_MODE	_IEEE_FP
/IMPLICIT_INCLUDE	__IMPLICIT_INCLUDE_ENABLED
/L_DOUBLE_SIZE	__X_FLOAT
/MODEL=ANSI	__MODEL_ANSI
/MODEL=ARM (Alpha only)	__MODEL_ARM
/PURE_CNAME	__PURE_CNAME, __HIDE_FORBIDDEN_NAMES[1]
/ROUNDING_MODE	__BIASED_FLT_ROUNDS
/RTTI	__RTTI
/STANDARD=RELAXED	__STD_ANSI, __NOUSE_STD_IOSTREAM
/STANDARD=ANSI	__STD_ANSI, __NOUSE_STD_IOSTREAM
/STANDARD=ARM	__STD_ARM, __NOUSE_STD_IOSTREAM
/STANDARD=CFRONT	The CFRONT option is no longer supported.
/STANDARD=GNU	__STD_GNU, __NOUSE_STD_IOSTREAM
/STANDARD=MS	__STD_MS, __NOUSE_STD_IOSTREAM
/STANDARD=STRICT_ANSI	__STD_STRICT_ANSI, __USE_STD_IOSTREAM, __PURE_CNAME, __HIDE_FORBIDDEN_NAMES
/STANDARD=STRICT_ANSI /WARNINGS=ANSI_ERRORS	__STD_STRICT_ANSI_ERRORS, __PURE_CNAME, __HIDE_FORBIDDEN_NAMES
/USING=STD	__IMPLICIT_USING_STD
/STANDARD=LATEST	__STD_STRICT_ANSI, __USE_STD_IOSTREAM, __PURE_CNAME, __HIDE_FORBIDDEN_NAMES

Command-line Option	Macro Name
/STANDARD=LATEST /WARNINGS=ANSI_ERRORS	`__STD_STRICT_ANSI_ERRORS`, `__PURE_CNAME`, `__HIDE_FORBIDDEN_NAMES`

[1]When you compile with VSI C++ using any values of /STANDARD that set strict C++ standard conformance (ARM, MS, STRICT_ANSI, and LATEST), versions of the standard header files are included that hide many identifiers that do not follow the rules. The header file <stdio.h>, for example, hides the definition of the macro TRUE. The compiler accomplishes this by predefining the macro __HIDE_FORBIDDEN_NAMES for the above-mentioned /STANDARD values.

You can use the /UNDEFINE="__HIDE_FORBIDDEN_NAMES" command-line qualifier to prevent the compiler from predefining this macro and, thereby, including macro definitions of the forbidden names.

2.1.3. Translation Limits

The only translation limits imposed in the compiler are as follows:

Limit	Meaning
32,767	Bytes in the representation of a string literal. This limit does not apply to string literals formed by concatenation.
8192	Characters in an internal identifier or macro name.
8192	Characters in a logical name.
8192	Characters in a physical source line, on OpenVMS systems.
1012	Bytes in any one function argument.
512	Characters in a physical source line, on OpenVMS Alpha systems.
255	Arguments in a function call.[1]
255	Parameters in a function definition.[1]
127	Characters in a qualified identifier in the debugger.
31	Significant characters in an external identifier with “C” linkage. A warning is issued if such an identifier is truncated.

[1]The compiler may add one or two hidden arguments to a function, which reduces to 254 or 253 the number of arguments available to the user.

2.1.4. Numerical Limits

The numerical limits, as defined in the header files `<limits.h>` and `<float.h>` are as follows:

- The number of bits in a character of the execution character set is eight.
- The representation and set of values for type `char` and for type `signed char` are the same. You can change this equivalence from `signed char` to `unsigned char` with a command-line option.
- The representation and set of values for the `short` type is 16 bits.
- The representation and set of values for the types `int`, `signed int`, and `long` are the same (32 bits).
- The representation and set of values for type `unsigned int` and for type `unsigned long` are the same (32 bits).
- The representation and set of values for type `double` are 64 bits.
- The representation and set of values for type `long double` are 128 bits unless the /L_DOUBLE_SIZE=64) qualifier is specified.

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

Numerical limits not described in this list are defined in *The Annotated C++ Reference Manual*.

2.1.5. Argument-Passing and Return Mechanisms

The compiler passes arrays, functions, and class objects with a constructor or destructor by reference. All other objects are passed by value.

If a class has a constructor or a destructor, it is not passed by value. In this case, the compiler calls a copy constructor to copy the object to a temporary location, and passes the address of that location to the called function.

If the return value of a function is a class that has defined a constructor or destructor or is greater than 64 bits, storage is allocated by the caller and the address to this storage is passed in the first parameter to the called function. The called function uses the storage provided to construct the return value.

2.2. Implementation Extensions and Features

This section describes the extensions and implementation-specific features of the compiler on OpenVMS systems.

2.2.1. Identifiers

In the compiler, the dollar sign ($) is a valid character in an identifier.

For each external function with C++ linkage, the compiler decorates the function name with a representation of the function's type.

2.2.1.1. External Name Encoding

The compiler uses the external name encoding scheme described in §7.2.1c of *The Annotated C++ Reference Manual*.

For the basic types, the external name encoding scheme is exactly the same as that described in *The Annotated C++ Reference Manual*, as follows:

Type	Encoding
`void`	`v`
`char`	`c`
`short`	`s`
`int`	`i`
`long`	`l`
`float`	`f`
`double`	`d`
`long double`	`r`
`...`	`e`

Type	Encoding
`bool`	`jb`
`wchar_t`	`jw`

Class names are encoded as described in *The Annotated C++ Reference Manual*, except that the VSI C ++ compiler uses the lowercase `q` instead of uppercase `Q`, and denotes the qualifier count as a decimal number followed by an underscore, as follows:

Class	Notation	Encoding
simple	`Complex`	`7Complex`
qualified	`X::YY`	`q2_1x2yy`

Type modifiers are encoded as follows:

Modifier	Encoding
`const`	`k`
`signed`	`g`
`volatile`	`w`
`unsigned`	`u`
`__unaligned`	`b`

Type declarators are encoded as follows:

Type	Notation	Encoding
array	`[10]`	`a10_`
function	`()`	`x`
pointer	`*`	`p`
pointer to member	`S::*`	`m1S`
reference	`&`	`n`
unnamed enumeration type		`h`

On OpenVMS systems, the compiler also supports the following data types:

Type	Encoding
`__int16`	`ji4`
`__int32`	`ji5`
`__int64`	`ji6`
`__f_float`	`jf`
`__g_float`	`jg`
`__s_float`	`js`
`__t_float`	`jt`

2.2.1.2. Modifying Long Names

On OpenVMS systems, if an identifier for a function name with C++ linkage exceeds 31 characters, the name is modified as follows:

1. A unique value is generated by hashing the full decorated name. This seven-character code is appended to the end of the name.
2. The name is preceded by the `cxx$` facility prefix.
3. The name is truncated in three back-to-front passes, eliminating underscores, then vowels, and then consonants (y is a consonant). A vowel is never removed if the following conditions apply:
   - It occurs as the first character in the fully decorated name.
   - The character before the vowel is either another vowel or is non-alphanumeric.

   The hash code added at the end of the name is not truncated.

   Truncation ceases when the truncated name, combined with the `cxx$` facility prefix and the unique radix 32 value at the end, equals 31 characters.

For information on how to view the demangled form of these names, see Section 1.6.

2.2.2. Order of Static Object Initialization

Nonlocal static objects are initialized in declaration order within a compilation unit and in link order across compilation units. On OpenVMS systems, the compiler uses the `lib$initialize` mechanism to initialize nonlocal static objects.

2.2.3. Integral Conversions

When demoting an integer to a signed integer, if the value is too large to be represented the result is truncated and the high-order bits are discarded.

Conversions between signed and unsigned integers of the same size involve no representation change.

2.2.4. Floating-Point Conversions

When converting an integer to a floating-point number that cannot exactly represent the original value, the compiler rounds off the result of the conversion to the nearest value that can be represented exactly.

When the result of converting a floating-point number to an integer or other floating-point number at compile time cannot be represented, the compiler issues a diagnostic message.

When converting an integral number or a double floating-point number to a floating-point number that cannot exactly represent the original value, rounds off the result to the nearest value of type `float`.

When demoting a `double` value to `float`, if the converted value is within range but cannot exactly represent the original value, the compiler rounds off the result to the nearest representable `float` value, the compiler performs similar rounding for demotions from `long double` to `double` or `float`.

2.2.5. Explicit Type Conversion

In C++, the expression `T()` (where `T` is a simple type specifier) creates an rvalue of the specified type, whose value is determined by default initialization. According to the *The Annotated C++ Reference Manual*, the behavior is undefined if the type is not a class with a constructor, but the ANSI/ISO International Standard removes this restriction. With this change you can now write:

```
int i=int(); // i must be initialized to 0
```

2.2.6. The sizeof Operator

The type of the `sizeof` operator is `size_t`. In the header file, `stddef.h`, the compiler defines this type as `unsigned int`, which is the type of the integer that holds the maximum size of an array.

2.2.7. Explicit Type Conversion

A pointer takes up the same amount of memory storage as objects of type `int` or `long` (or their unsigned equivalents). Therefore, a pointer can convert to any of these types and back again without changing its value. No scaling occurs and the representation of the value is unchanged.

Conversions to and from a shorter integer and a pointer are similar to conversions to and from a shorter integer and `unsigned long`. If the shorter integer type was signed, conversion fills the high-order bits of the pointer with copies of the sign bit.

2.2.8. Multiplicative Operators

The semantics of the division (/) and remainder (%) operator are as follows:

- If either operand of the division operator is negative, the compiler truncates the result toward 0 (that is, the smallest integer larger than the algebraic quotient).
- If either operand of the remainder operator is negative, the result takes the same sign as that of the first operand.

In the following cases of undefined behavior detected at compile time, the compiler issues a warning:

Integer overflow
Division by 0
Remainder by 0

2.2.9. Additive Operators (§r.5.7)

You can subtract pointers to members of the same array. The result is the number of elements between the two array members, and is of type `ptrdiff_t`. In the header file `stddef.h`, the compiler defines this type as `int`.

2.2.10. Shift Operators (§r.5.8)

The expression `E1 >> E2` shifts `E1` to the right `E2` positions. If `E1` has a signed type, the compiler fills the vacated high-order bits of the shifted value `E1` with a copy of `E1`'s sign bit (arithmetic shift).

2.2.11. Equality Operators

When comparing two pointers to members, the compiler guarantees equality if either of the following conditions hold:

- Both pointers are NULL.
- The same address expression (&) created both pointers.

When comparing two pointers to members, the compiler guarantees inequality if either of the following conditions hold:

- Only one pointer is NULL.
- Each pointer produces a different member if applied to the same object.

When created by different address expressions, two pointers to members may compare either as equal or as unequal if they produce the same member when applied to the same object.

2.2.12. Type Specifiers

For variables that are modifiable in ways unknown to the compiler, use the `volatile` type specifier. Declaring an object to be volatile means that every reference to the object in the source code results in a reference to memory in the object code.

2.2.13. asm Declarations *(Alpha only)*

In the compiler, asm declarations produce a compile-time error. As an alternative to asm, you can use built-in functions. See Appendix C for more information.

2.2.14. Linkage Specifications

Specifying linkage other than “C++” or “C” generates a compile-time error.

In object files, the compiler decorates with type information the names of functions with C++ linkage. This permits overloading and provides rudimentary type checking across compilation units. The type-encoding algorithm used is similar to that given in §7.2.1c of *The Annotated C++ Reference Manual* (see Section 2.2.1.1).

2.2.15. Class Layout

The alignment requirements and sizes of structure components affect the structure's alignment and size. A structure can begin on any byte boundary and occupy any integral number of bytes.

2.2.15.1. Structure Alignment

Structure alignment is controlled by the /MEMBER_ALIGNMENT command-line qualifier or by using the `#pragma member_alignment` preprocessor directive. If /MEMBER_ALIGNMENT is specified, or implied by default, the maximum alignment required by any member within the structure determines the structure's alignment. When the structure or union is a member of an array, padding is added to ensure that the size of a record, in bytes, is a multiple of its alignment.

Components of a structure are laid out in memory in the order in which they are declared. The first component has the same address as the entire structure. Padding is inserted between components to satisfy alignment requirements of individual components.

If /NOMEMBER_ALIGNMENT is specified, each member of a structure appears at the next byte boundary.

2.2.15.2. Bit-Fields

If /MEMBER_ALIGNMENT is specified, or implied by default, the presence of bit-fields causes the alignment of the whole structure or union to be at least the same as that of the bit-field's base type.

For bit-fields (including zero-length bit-fields) not immediately declared following other bit-fields, their base type imposes the alignment requirements (less than that of type `int`). Within the alignment unit (of the same size as the bit-field's base type), bit-fields are allocated from low order to high order. If a bit-field immediately follows another bit-field, the bits are packed into adjacent space in the same unit, if sufficient space remains; otherwise, padding is inserted at the end of the first bit-field and the second bit-field is put into the next unit.

Bit-fields of base type `char` must be smaller than 8 bits. Bit-fields of base type `short` must be smaller than 16 bits.

2.2.15.3. Access Specifiers

The layout of a class is unaffected by the presence of access specifiers.

2.2.15.4. Class Subobject Offsets

A class object that has one or more base classes contains instances of its base classes as subobjects. The offsets of nonvirtual base class subobjects are less than the offsets of any data members that are not part of base class subobjects.

The offsets of nonvirtual base classes increase in derivation order. The offset of the first nonvirtual base class subobject of any class is 0. For single inheritance, the address of a class object is always the same as the address of its base class subobject.

If a class has virtual functions, an object of that class contains a pointer to a virtual function table (VFPTR).

If a class has virtual base classes, an object of that class contains a pointer to a virtual base class table (VBPTR).

For a class with no base classes, the offset of a VFPTR or VBPTR is greater than the offset of any data members. Thus, the offset of the first data member of a class with no base classes is 0, which facilitates interoperability with other languages. If the leftmost base class of a subclass has a VFPTR, a VBPTR, or both, and is not virtual, the class and its base class share the table or tables.

The offsets of virtual base class subobjects are greater than the offset of any data member, and increase in the order of derivation of the virtual base classes. In increasing order, a class object contains the following:

1. Nonvirtual base class subobjects
2. Data members
3. VFPTR (if required)
4. VBPTR (if required)
5. Virtual base class subobjects

Consider the following example:

```
class B1
```

```
{
 int x[1];
};
class B2 : virtual B1
{
 int y[2];
 virtual int f1();
};
class B3 : virtual B2, virtual B1
{
 int z[3];
 virtual int f2();
};
class D : B3
{
 int a[4];
 virtual int f1(), f2(), f3();
};
```

Figure 2.1 shows the layout of an object of D class for this example.

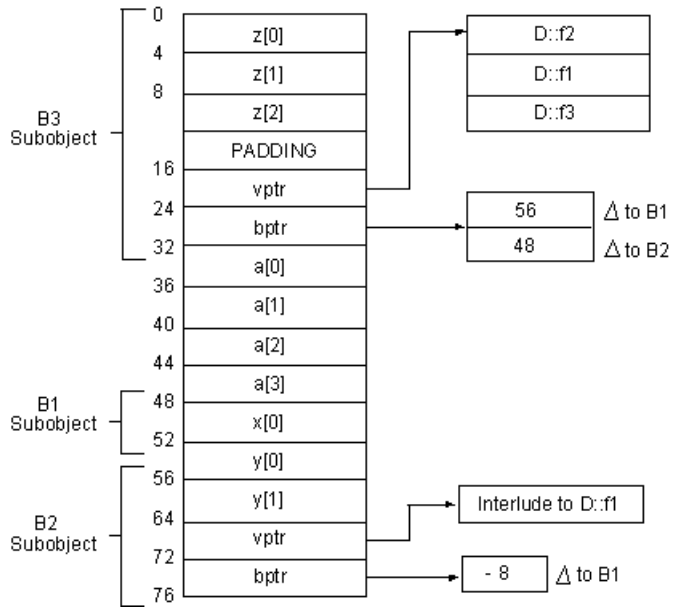
2.2.16. Virtual Function and Base Class Tables

The compiler allocates storage for virtual function tables (VTBLs) and base class tables (BTBLs) using the common block extern model. All references to VTBLs and BTBLs share a single copy. (The compiler specifies the local (LCL) PSECT attribute for these tables. Thus, one copy of each table exists for each program image file.) This means that you need not be concerned with the associations of these tables during compilation, and the compiler command switch +e supplied in other implementations is not needed for VSI C++ for OpenVMS systems.

2.2.17. Multiple Base Classes

Within a class object, base class subobjects are allocated in derivation order; that is, immediate base classes are allocated in the order in which they appear in the class declaration.

Figure 2.1. Layout of an Object of D Class



2.2.18. Temporary Objects

Under the following conditions, the compiler creates temporary objects for class objects with constructors:

- An object is returned from a function.
- An object is passed as an argument.
- An object is created using the constructor notation.
- A user-defined conversion is implicitly used.

Variations in the compiler generation of such temporary objects can adversely affect their reliability in user programs. The compiler avoids introducing a temporary object whenever it discovers that the temporary object is not needed for accurate compilation. Therefore, you should modify or write your programs so as not to depend on side effects in the constructors or destructors of temporary objects.

2.2.18.1. Lifetime of Temporary Objects

Generally the compiler implements destruction of temporary objects at the end of statements. In certain situations, however, temporary objects are destroyed at the end of the expression; they do not persist to the end of the statement. Temporary objects do not persist to the end of statements in expressions that are:

- In operands of built-in conditional operators (`||` and `&&`)
- In the second or third operand of the ternary operator (`?:`)
- Operands to the built-in comma operator (`,`)

Consider the following example:

```
struct A {
  void print(int i);
  A();
  ~A() { }
};

struct B {
  A* find(int i);
  B(int i);
  B();
  ~B() { }
};

void f() {
  B(8).find(6)->print(6);
  (*(B(5).find(3))).print(3);
  return;
}
```

In the first and second statements inside `void f()`, the compiler destroys the temporary object created in evaluating the expressions `B(8)` and `B(5)` after the call to `A::print(int)`.

2.2.18.2. Nonconstant Reference Initialization with a Temporary Object

If your program tries to initialize a nonconstant reference with a temporary object, the compiler generates a warning. For example:

```
struct A {
  A(int);
};
void f(A& ar);

void g() {
  f(5);  // warning!!
}
```

2.2.18.3. Static Member Functions Selected by Expressions Creating Temporary Objects

When a static member is accessed through a member access operator, the expression on the left side of the dot (.) or right arrow (->) is not evaluated. In such cases, the compiler creates code that calls the static member function to handle the destruction of a class type temporary; the compiler does not create temporary destructor code. For example:

```
struct A {
        ~A();
        static void sf();
};

struct B {
        A operator ()() const;
};

void f () {
    B bobj;
    bobj().sf();        // If 'bobj()' is evaluated, a temporary of
                        // type 'A' is created.
}
```

2.2.19. File Inclusion

The `#include` directive inserts external text into the macro stream delivered to the compiler. Programmers often use this directive to include global definitions for use with compiler functions and macros in the program stream.

On OpenVMS systems, the `#include` directive may be nested to a depth determined by the FILLM process quota and by virtual memory restrictions. The compiler imposes no inherent limitation on the nesting level of inclusion.

In C++ source programs, inclusion of both OpenVMS and most UNIX style file specifications is valid. For example, the following is a valid UNIX style file specification:

```
nodename!/device/directory/filename.dat.3
```

The exclamation point (!) separates the node name from the rest of the specification; slash characters (/) separate devices and directories; periods (.) separate file types and file versions. Because one character separates two segments of the file specification, ambiguity can occur.

The /INCLUDE_DIRECTORY=(*pathname*,...) qualifier provides an additional level of search for user-defined include files. Each *pathname* argument can be either a logical name or a legal UNIX style directory in a quoted string. The default is /NOINCLUDE_DIRECTORY.

The qualifier provides functionality similar to the `-I` option of the `cxx` command on UNIX systems. This qualifier allows you to specify additional locations to search for files to include. Putting an empty string in the specification prevents the compiler from searching any of the locations it normally searches but directs it to search *only* in locations you identify explicitly on the command line with the /INCLUDE_DIRECTORY and /LIBRARY qualifiers (or by way of the specification of the primary source file, depending on the /NESTED_INCLUDE_DIRECTORY qualifier).

The basic order for searching depends on the form of the header name (after macro expansion), with additional aspects controlled by other command line qualifiers as well as the presence or absence of logical name definitions. The valid possibilities for names are as follows:

- Enclosed in quotes. For example: "stdio.h"
- Enclosed in angle brackets. For example: `<stdio.h>`

Unless otherwise defined, searching a location means that the compiler uses the string specifying the location as the default file specification in a call to an RMS system service (that is, a $SEARCH/$PARSE) with a primary file specification consisting of the name in the #include (without enclosing delimiters). The search terminates successfully as soon as a file can be opened for reading.

Specifying a null string in the /INCLUDE qualifier causes the compiler to do a non-standard search. This search path is as follows:

1. The current directory (quoted form only)
2. Any directories specified in the /INCLUDE qualifier
3. The directory of the primary input file
4. Text libraries specified on the command line using /LIBRARY

For standard searches, the search order is as follows:

1. Search the current directory (directory of the source being processed). If angle-bracket form, search only if no directories are specified with /INCLUDE_DIRECTORY.
2. Search the locations specified in the /INCLUDE_DIRECTORY qualifier (if any).
3. If CXX$SYSTEM_INCLUDE is defined as a logical name, search CXX$SYSTEM_INCLUDE:.HXX or just CXX$SYSTEM_INCLUDE:., depending on the qualifier /ASSUME=NOHEADER_TYPE_DEFAULT. If nothing is found, go to step 6.
4. If CXX$LIBRARY_INCLUDE is defined as a logical name, CXX$LIBRARY_INCLUDE:.HXX or CXX$LIBRARY_INCLUDE:., depending on the qualifier /ASSUME=NOHEADER_TYPE_DEFAULT. If nothing is found, go to step 6.

5. If /ASSUME=HEADER_TYPE_DEFAULT is not specified, search the default list of locations for plain-text copies of compiler header files as follows:

   SYS$COMMON:[CXX$LIB.INCLUDE.CXXL$ANSI_DEF]
   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF_HXX].HXX
   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF].H
   SYS$COMMON:[CXX$LIB.INCLUDE.SYS$STARLET_C].H

   If /ASSUME=HEADER_TYPE_DEFAULT is specified, search the default list of locations for plain-text copies of compiler header files as follows:

   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF_HXX].HXX
   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF].H
   SYS$COMMON:[CXX$LIB.INCLUDE.SYS$STARLET_C].H
   SYS$COMMON:[CXX$LIB.INCLUDE.CXXL$ANSI_DEF]

6. Search the directory of the primary input file.

7. If quoted form, and CXX$USER_INCLUDE is defined as a logical name, search CXX$USER_INCLUDE:.HXX or CXX$USER_INCLUDE:., depending on the /ASSUME=NOHEADER_TYPE_DEFAULT qualifier.

8. Search the text libraries. Extract the simple file name and file type from the #include specification, and use them to determine a module name for each text library. There are three forms of module names used by the compiler:

   a. type stripped:

      The file type will be removed from the include file specification to form a library module name. Examples:

#include "foo.h"	Module name "FOO"
#include "foo"	Module name "FOO"
#include "foo"	Module name "FOO"

   b. type required:

      The file type must be a part of the file name. Examples:

#include "foo.h"	Module name "FOO.H"
#include "foo"	Module name "FOO."
#include "foo"	Module name "FOO."

   c. type optional:

      First an attempt is made to find a module with the type included in the module name. If this is unsuccessful, an attempt is made with the type stripped from the module name. If this is unsuccessful, the search moves on to the next library.

If /ASSUME=HEADER_TYPE_DEFAULT is specified, the following text libraries are searched in this order:

Libraries specified on the command line with the /LIBRARY qualifier (all files, type stripped)
CXX$TEXT_LIBRARY (all files, type stripped)
DECC$RTLDEF (H files and unspecified files, type stripped)
SYS$STARLET_C (all files, type stripped)
CXXL$ANSI_DEF (unspecified files, type stripped)

Otherwise, these text libraries are searched in this order:

Libraries specified on the command line with the /LIBRARY qualifier (all files, type optional)
CXX$TEXT_LIBRARY (all files, type optional)
CXXL$ANSI_DEF (all files, type required)
DECC$RTLDEF (H files and unspecified files, type stripped)
SYS$STARLET_C (all files, type stripped)

Two text library search examples (stop when something is found):

Example 1

```
#include "foo"
```

a. For each library specified via the /LIBRARY qualifier:

   - Look for "FOO."
   - Look for "FOO"

b. Look for "FOO." in CXX$TEXT_LIBRARY

c. Look for "FOO" in CXX$TEXT_LIBRARY

d. Look for "FOO." in CXXL$ANSI_DEF (Do not look for "FOO" because the type is required as part of the module name)

e. Look for "FOO" in DECC$RTLDEF (not "FOO." because the type must not be part of the module name)

f. Look for "FOO" in SYS$STARLET_C (not "FOO." because the type must not be part of the module name)

Example 2

```
#include "foo.h"
```

a. For each library specified via the /LIBRARY qualifier:

   - Look for "FOO.H"
   - Look for "FOO"

b. Look for "FOO.H" in CXX$TEXT_LIBRARY

c. Look for "FOO" in CXX$TEXT_LIBRARY

d. Look for "FOO.H" in CXXL$ANSI_DEF (Do not look for "FOO" because the type is required as part of the module name)

e. Look for "FOO" in DECC$RTLDEF (not "FOO.H" because the type must not be part of the module name)

f. Look for "FOO" in SYS$STARLET_C (not "FOO.H" because the type must not be part of the module name)

g. If neither CXX$LIBRARY_INCLUDE nor CXX$SYSTEM_INCLUDE is defined as a logical name, then search SYS$LIBRARY:.HXX.

2.2.20. Nested Enums and Overloading

The C++ language allows programmers to give distinct functions the same name, and uses either overloading or class scope to differentiate the functions:

```
void f(int);
void f(int *);
class C {void f(int);};
class D {void f(int);};
```

Yet, linkers cannot interpret overloaded parameter types or classes, and they issue error messages if they find more than one definition of any external symbol. C++ compilers, including VSI C++, solve this problem by assigning a unique **mangled name** (also called **type safe linkage name**) to every function. These unique mangled names allow the linker to tell the overloaded functions apart.

The compiler forms a mangled name, in part, by appending an encoding of the parameter types of the function to the function's name, and if the function is a member function, the function name is qualified by the names of the classes within which it is nested.

For example, for the function declarations at the beginning of this section, the compiler might generate the mangled names `f__Xi`, `f__XPi`, `f__1CXi`, and `f__1DXi` respectively. In these names, `i` means a parameter type was `int`, `P` means “pointer to”, `1C` means nested within class `C`, and `1D` means nested within class `D`.

There is a flaw in the name mangling scheme used by the compiler that can cause problems in uncommon cases. The compiler fails to note in the encoding of an enum type in a mangled name whether the enum type was nested within a class. This can cause distinct overloaded functions to be assigned the same mangled name:

```
struct C1 {enum E {red, blue};};
struct C2 {enum E {red, blue};};

extern "C" int printf(const char *, ...);
void f(C1::E x) {printf("f(C1::E)\n");}
void f(C2::E x) {printf("f(C2::E)\n");}

int main()
{
    f(C1::red);
    f(C2::red);
    return 1;
}
```

In the previous example, the two overloaded functions named `f` differ only in that one takes an argument of enum type `C1::E` and the other takes an argument of enum type `C2::E`. Since the compiler fails to include the names of the classes containing the enum type in the mangled name, both functions have mangled names that indicate the argument type is just `E`. This causes both functions to receive the same mangled name.

In some cases, the compiler detects this problem at compile-time and issues a message that both functions have the same type-safe linkage. In other cases, the compiler issues no message, but the linker complains about duplicate symbol definitions.

If you encounter such problems, you can recompile using the /DISTINGUISH_NESTED_ENUMS qualifier *(Alpha only)*. This causes the compiler, when forming a mangled name, to include the name of class or classes within which an enum is nested, thereby preventing different functions from receiving the same the same mangled name.

Because the /DISTINGUISH_NESTED_ENUMS qualifier changes the external symbols the compiler produces, you can get undefined symbol messages from the linker if some modules are compiled with /DISTINGUISH_NESTED_ENUMS and some are compiled without it. Because of this, /DISTINGUISH_NESTED_ENUMS might make it difficult to link against old object files or libraries of code.

If you compile your code with /DISTINGUISH_NESTED_ENUMS and try to link against a library that was compiled without the /DISTINGUISH_NESTED_ENUMS qualifier, you receive an undefined symbol message from the linker if you attempt to call a function from the library that takes an argument of a nested enum type. The mangled name of the function in the library will be different from the mangled name your code is using to call the function.

Note that the /DISTINGUISH_NESTED_ENUMS qualifier has no meaning on I64 systems because it modifies the behavior of programs compiled with /MODEL=ARM, and that model is not supported on I64 systems.

2.2.21. Guiding Declarations

A guiding declaration is a function declaration that matches a function template, does not introduce a function definition (implies an instantiation of the template body and not a explicit specialization), and is subject to different argument matching rules than those that apply to the template itself – therefore affecting overload resolution. Consider the following example:

```
template <class T> void f(T) {
  printf("In template f\n");
}

void f(int);

int main() {
  f(0);      // invokes non-template f
  f<>(0.0);  // invokes template f
  return 0;
}

void f(int) {
  printf("In non-template f\n");
}
```

Because there is no concept of guiding declaration in the current version of the C++ International Standard, the function `f` in the example is not regarded as an instance of function `template f`. Furthermore, there are two functions named `f` that take an `int` parameter. A call of `f(0)` would invoke the former, while a call of `f<>(0)` would be required to invoke the latter.

2.3. Alternative Tokens

The compiler supports use of alternative tokens:

/[no]alternative_tokens

Enable use of operator keywords and digraphs to generate tokens as follows:

Operator Keyword	**Token**
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

Digraph	**Token**
:>	]
<:	[
%>	}
<%	{
%:	#

2.4. Run-time Type Identification

The compiler emits type information for **run-time type identification** (RTTI) in the object module with the virtual function table, for classes that have virtual function tables.

You can specify the /[NO]RTTI qualifier to enable or disable support for RTTI (runtime type identification) features: `dynamic_cast` and `typeid`. Disabling runtime type identification can also save space in your object file because static information to describe polymorphic C++ types is not generated. The default is to enable runtime type information features and generate static information in the object file.

Specifying /NORTTI does not disable exception handling.

The type information for the class may include references to the type information for each base class and information on how to convert to each. The `typeinfo` references are mangled in the form `__T__<class>`.

2.5. Message Control and Information Options

The compiler supports the following message control options. The options apply only to warning and informational messages. The `ident` variable is obtained from the error message.

Indicated messages can specify one or more message identifiers `ident` or the message group name `all`.

The default qualifier, /WARNINGS, outputs all enabled informational and warning messages.
The /NOWARNINGS qualifier suppresses both the informational and the warning messages.

Message options are processed and take effect in the following order:

/WARNINGS=NOWARNINGS

Disable all warnings.

/WARNINGS= INFORMATIONALS

Enable informationals.

Although /WARNINGS=INFORMATIONALS enables most informationals, we recommend using /WARNINGS=ENABLE=ALL instead.

/WARNINGS= INFORMATIONALS=ALL or (`ident,...`)

Set the severity of the specified messages to Informational. You can specify ALL, which applies only to discretionary messages. The ALL option also enables informationals that are disabled by default.

With Version 7.1 of the C++ compiler, /WARNINGS=INFORMATIONALS=<tag> no longer enables all other informational messages.

/WARNINGS= WARNINGS=ALL or (`ident,...`)

Set the severity of the specified messages to Warning. You can specify ALL, which applies only to discretionary messages.

/WARNINGS= [NO]ANSI_ERRORS

Issue error messages for all ANSI violations when in STRICT_ANSI mode. The default is /WARNINGS=NOANSI_ERRORS.

/WARNINGS=ERRORS=ALL or (`ident,...`)

Set the severity of the specified messages to Error. You can specify ALL, which applies only to discretionary messages.

/WARNINGS=ENABLE=ALL or (`ident,...`)

Enable all compiler messages, including informational-level messages. Enable specific messages that normally would not be issued when using /QUIET. You can also use this option to enable messages disabled with /WARNINGS=DISABLE.

/WARNINGS=DISABLE=ALL or (`ident,...`)

Disable message. This can be used for any nonerror message.

/QUIET

Be more like Version 5.`n` error reporting. Fewer messages are issued using this option.

This is the default in arm mode (/STANDARD=ARM). All other modes default to /NOQUIET.

You can use the /WARNINGS=ENABLE option with this option to enable specific messages normally disabled using /QUIET.

The compiler supports the following message information option, which is disabled by default.

/WARNINGS=[NO]TAGS

Display a descriptive tag with each message. "D" indicates that the message is discretionary and that its severity can be changed from the command line or with a pragma. The tag displayed can be used as the `ident` variable in the /WARNINGS options.

Example:

```
$ cxx/warnings=tags t.cxx
f() {}
^
%CXX-W-NOSIMPINT, omission of explicit type is nonstandard ("int"
 assumed)
          (D:nosimpint)
at line number 1 in file CXX$:[SMITH]STD.CXX;1


f() {}
.....^
%CXX-W-MISSINGRETURN, non-void function "f" (declared at line 1) should
          return a value (D:missingreturn)
at line number 1 in file CXX$:[SMITH]STD.CXX;1

$ cxx /warnings=(notags,disable=nosimpint) t.cxx

f() {}
```

```
.....^
%CXX-W-MISSINGRETURN, non-void function "f" (declared at line 1) should
          return a value
at line number 1 in file CXX$:[SMITH]STD.CXX;1
```

Also see the `#pragma message` preprocessor directive.

Chapter 3. C++ Language Environment

This chapter describes the guidelines and procedures for customizing your language environment. It includes sections on changing your C header files to work with C++, organizing your C++ files, interfacing to other programming languages, and designing upwardly compatible C++ classes.

3.1. cname Headers

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*.

The implementation consists of eighteen `<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 in Appendix A.

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

Examples

1.
```
#include <cstdio>
void foo() {
  getchar();   // OK in /NOPURE_CNAME mode
               // %CXX-E-UNDECLARED in /PURE_CNAME mode
}
```

2.
```
#include <cstdio>
void foo() {
  std::getchar();  // OK in both modes
}
```

3.
```
#include <stdio.h>
void foo() {
  getchar();       // OK in both modes
  std::getchar();  // OK in both modes
}
```

3.2. Using Existing C Header Files

C header files that already conform to ANSI C standards must be modified slightly to be usable by VSI C++ programs. In particular, be sure to address the following issues:

- Enable the proper linkage for each language.

- Ensure that C++ keywords are not used as identifiers.
- Reconcile any namespace and scoping differences.

The compiler provides some C header files that have been modified to work with C++, including standard ANSI C header files. These headers are in the SYS$LIBRARY directory.

The following sections provide details on how to properly modify your headers.

3.2.1. Providing C and C++ Linkage

To modify header files, use conditional compilation and the `extern` specifier.

When programming header files to be used for both C and C++ programs, use the following convention for predefined macros. The system header files also provide an example of correct usage of the predefined macros.

```
#if defined __cplusplus
    /* If the functions in this header have C linkage, this
     * will specify linkage for all C++ language compilers.
     */
    extern "C" {
#endif

# if defined __DECC || defined __DECCXX
    /* If you are using pragmas that are defined only
     * with DEC C and DEC C++, this line is necessary
     * for both C and C++ compilers.  A common error
     * is to only have #ifdef __DECC, which causes
     * the compiler to skip the conditionalized
     * code.
     */

#   pragma __extern_model __save
#   pragma __extern_model __strict_refdef
    extern const char some_definition [];
#   pragma __extern_model __restore
# endif

 /* ...some data and function definitions go here... */

#if defined __cplusplus
    }   /* matches the linkage specification at the beginning. */
#endif
```

See *The Annotated C++ Reference Manual* for more information on linkage specifications.

3.2.2. Resolving C++ Keyword Conflicts

If your program uses any of the following C++ language keywords as identifiers, you must replace them with nonconflicting identifiers:

`asm`	`bool`	`catch`	`class`
`const_cast`	`delete`	`dynamic_cast`	`explicit`
`export`	`false`	`friend`	`inline`

mutable	namespace	new	operator
private	protected	public	reinterpret_cast
static_cast	template	this	throw
true	try	typeid	typename
virtual	wchar_t		

Alternative representation keywords are as follows:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
```

3.2.3. Handling Scoping Issues

Distinctions between ANSI C and C++ include slight differences in rules concerning scope. Therefore, you may need to modify some ANSI C header files to use them with C++.

The following sample code fragment generates an error regarding incompatible types, but the root cause is the difference in scope rules between C and C++. In ANSI C, the compiler promotes tag names defined in structure or union declarations to the containing block or file scope. This does not happen in C++.

```
struct Screen {
  struct _XDisplay *display;
};
typedef struct _XDisplay {
  // …
} Display;

struct Screen s1;
Display      *s2;

main()
{
  s1.display = s2;
}
```

The offending line in this sample is `s1.display = s2`. The types of `s1.display` and `s2` are the same in C but different in C++. You can solve the problem by adding the declaration `struct _XDisplay;` to the beginning of this code fragment, as follows:

```
struct _XDisplay; // this is the added line
struct Screen {
struct _XDisplay *display;
};
typedef struct _XDisplay {
  // ...
} Display;
// ...
```

3.2.4. Support for <stdarg.h> and <varargs.h> Header Files

The C compiler special built-in macros defined in the header files <stdarg.h> and <varargs.h>. These step through the argument list of a routine.

Programs that take the address of a parameter, and use pointer arithmetic to step through the argument list to obtain the value of other parameters, assume that all arguments reside on the stack and that arguments appear in increasing order. These assumptions are not valid for VSI C++. The macros in `<varargs.h>` can be used only by C functions with old-style definitions that are not legal in C++. To reference variable-length argument lists, use the `<stdarg.h>` header file.

The OpenVMS calling standard mechanism for returning structures larger than 8 bytes by value uses a hidden parameter. The parameter is a pointer to storage in the caller's frame. The `va_count` macro includes this parameter in its count.

3.3. Using VSI C++ with Other Languages

The following are suggestions regarding the use of VSI C++ with other languages:

- Passing entities, such as classes, by reference is safest.
- You cannot invoke class member functions from within any language other than C++.
- Every C++ routine that will be called from the other language should be declared in C++ with `extern "C"`. For example:

  ```
  extern "C"
     int myroutine(int, float);
  ```

  The `extern "C"` will cause the routine to have an unmangled name, so that you can refer to it as `myroutine` from a language such as Cobol or Fortran. Otherwise the routine's link name will be mangled into something like `myrout__Xif`.
- If the main routine is defined in the other language, you will probably need to use the other language's command-line interface to perform your link step. To include the appropriate C++ libraries and startup file, you will need to add some arguments to the command line. The most reliable way to determine what is needed is to test with a small C++ program.

3.4. Linkage to Non-C++ Code and Data

With linkage specifications, you can both import code and data written in other languages into a VSI C ++ program and export VSI C++ code and data for use with other languages. See *The Annotated C++ Reference Manual* for details on the `extern "C"` declaration.

3.5. How to Organize Your C++ Code

This section explains the best way for compiler users to organize an application into files; it assumes that you are using automatic instantiation to instantiate any template classes and functions.

3.5.1. Code That Does Not Use Templates

The general rule is to place *declarations* in *header files* and place *definitions* in *library source files*. The following items belong in header files:

- Class declarations
- Global function declarations

- Global data declarations

And the following items belong in library source files:

- Static member data definitions
- Out-of-line member function definitions
- Out-of-line global function definitions
- Global data definitions

Header files should be directly included by modules that need them. Because several modules may include the same header file, a header file must not contain definitions that would generate multiply defined symbols when all the modules are linked together.

Library source files should be compiled individually and then linked into your application. Because each library source file is compiled only once, the definitions it contains will exist in only one object module and multiply defined symbols are thus avoided.

For example, to create a class called “array” you would create the following two files:

Header file, arrayInt.hxx:

```
// arrayInt.hxx
#ifndef ARRAY_H
#define ARRAY_H

class arrayInt {
private:
        int curr_size;
        static int max_array_size;
public:
        arrayInt() :curr_size(0) {;}
        arrayInt(int);
};

#endif
```

Library source file, arrayInt.cxx:

```
// arrayInt.cxx
#include "arrayInt.hxx"

int array::max_array_size = 256;

arrayInt::arrayInt(int size) :  curr_size(size) { ...;  }
```

You would then compile the `arrayInt.cxx` library source file using the following command:

```
cxx/include=[.include] arrayInt.cxx
```

The resulting object file could either be linked directly into your application or placed in a library (see Section 3.5.4).

The header file uses **header guards**, which is a technique to prevent multiple inclusion of the same header file.

3.5.2. Code That Uses Templates

With the widespread use of templates in C++, determining the proper place to put declarations and definitions becomes more complicated.

The general rule is to place template declarations and definitions in header files, and to place specializations in library source files.

Thus, the following items belong in template declaration files:

- Declarations of global function templates
- Declarations of class templates
- Declarations of global function template specializations
- Declarations of class template specializations

The following items can be placed either in the header file with the corresponding template declaration or in a separate header file that can be implicitly included when needed. This file has the same basename as the corresponding declaration header file, with a suffix that is found by implicit inclusion. For example, if the declaration is in the header file `inc1.h`, these corresponding definitions could be in file `inc1.cxx`.

- Definitions of out-of-line global function templates
- Definitions of static member data of class templates
- Definitions of out-of-line member functions of class templates

The following must be placed in library source files to prevent multiple definition errors:

- Definitions of global function template specializations
- Definitions of static member data specializations of class templates
- Definitions of out-of-line class member function specializations

These guidelines also apply to nontemplate nested classes inside of template classes.

Note

Do not place definitions of nontemplate class members, nontemplate functions, or global data within template definition files; these must be placed in library source files.

All these header files should use header guards, to ensure that they are not included more that once either explicitly or by implicit inclusion.

For example, the array class from Section 3.5.1, modified to use templates, would now look as follows:

Template declaration file, array.hxx:

```
// array.hxx
#ifndef ARRAY_HXX
#define ARRAY_HXX
```

```
template <class T>
class array {
private:
        int curr_size;
        static int max_array_size;
public:
        array() :curr_size(0) {;}
        array(int size,const T& value = T());
};

#endif
```

Template definition file, array.cxx:

```
// array.cxx
template <class T>
int array<T>::max_array_size = 256;

template <class T>
array<T>::array(int size,const T& value )  {... ; }
```

Then you would create a source file `myprog.cxx` that uses the array class as follows:

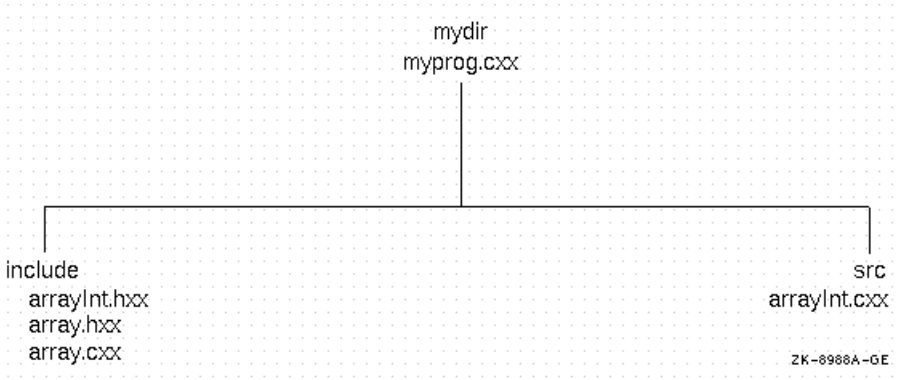
```
// myprog.cxx

#include "array.hxx"

main() {
        array<int> ai;
        // ...
}
```

Figure 3.1 shows the placement of these files.

Figure 3.1. Placement of Template Declaration and Definition Files



You would then compile `myprog.cxx` in the `mydir` directory with the following command:

```
cxx/incl=[.include] myprog.cxx
```

In this case, you do not need to create library source files because the static member data and out-of-line members of the array template class are instantiated at the time you compile `myprog.cxx`.

However, you would need to create library source files for the following cases:

- Your template declaration file declares nontemplate classes, global functions, or global data that require definitions in a library source file.
- A template class declares an out-of-line nontemplate friend function whose definition must be placed in a library source file.
- Your template declaration file declares a specialization of a template class whose static member data or out-of-line member function definitions must be placed in a library source file.
- Your template declaration file declares an out-of-line specialization of a template function, whose definition must be placed in a library source file.

3.5.3. Summary

Table 3.1 describes where to place declarations and definitions, as discussed in Section 3.5.1 and Section 3.5.2.

Table 3.1. Declaring and Defining Classes, Functions, and Data

Feature	Declaration	Out-of-Line Definition
Class	Header file	
Static member data	Within class declaration	Library source file
Member function	Within class declaration	Library source file
Global function	Header file	Library source file
Global data	Header file	Library source file
Template class	Template declaration file	
Static member data of template class	Within template class declaration	Template definition file
Member function of template class	Within template class declaration	Template definition file
Global template function	Template declaration file	Template definition file
Global, nontemplate friend function of template class	Within template class declaration	Library source file
Specialization of template class	Template declaration file	
Specialization of template function	Template declaration file	Library source file

3.5.4. Creating Libraries

Libraries are useful for organizing the sources within your application as well as for providing a set of routines for other applications to use. Libraries can be either object libraries or shareable libraries. Use an object library when you want the library code to be contained within an application's image; use shareable libraries when you want multiple applications to share the same library code.

Creating a library from nontemplate code is straightforward: you simply compile each library source file and place the resulting object file in your library.

Creating a library from template code requires that you explicitly request the instantiations that you want to provide in your library. See Chapter 7 for details.

If you make your library available to other users, you must also supply the corresponding declarations and definitions that are needed at compile time. For nontemplate interfaces, you must supply the header files that declare your classes, functions, and global data. For template interfaces, you must provide your template declaration files as well as your template definition files.

For more information on creating libraries, see the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual* and the *VSI OpenVMS Linker Utility Manual*.

3.6. Sample Code for Creating OpenVMS Shareable Images

The SW_SHR sample code consists of several source modules, a command procedure and this description. Table 3.2 lists each of the constituent modules, which are located in the directory SYS$COMMON:[SYSHLP.EXAMPLES.CXX] on your system.

The code creates an OpenVMS shareable image called SW_SHR.EXE that supplies a Stopwatch class identical to the C++ Class Library's Stopwatch class. For detailed information about the Stopwatch class, refer to the *VSI C++ Class Library Reference Manual*, CXX_CLASSLIB.PS, in the SYS$COMMON:[SYSHLP.CXX$HELP] directory.

SW_SHR also provides an instance of a Stopwatch class named `G_sw` that shows how to export a class instance from a shareable image. The exportation occurs in the same way that `cout`, `cin`, `cerr`, and `clog` are exported from the C++ Class Library shareable image.

Table 3.2. Shareable Image Example Files

Module Name	Description
SW_SHARE.HXX	General use macros to make exporting of global data (class instances) from shareable images more transparent to the users of class objects.
SW.HXX	The definition of the Stopwatch class supplied by the shareable image.
SW.CXX	Source associated with the public functions defined in SW.HXX. It also contains the declaration of the global Stopwatch (G_sw) class instance.
SW_TEST.CXX	A test of each of the Stopwatch's public access points and also the G_sw class instance.
SW_BUILD.COM	A DCL command procedure used to build both the shareable image and the program.
SW_SHR_ALPHA.OPT	An OpenVMS Linker options file, used on OpenVMS systems, that contains the SYMBOL_VECTOR entry points.
SW_SHR_IA64.OPT	An OpenVMS Linker options file, used on OpenVMS I64 systems, that contains the SYMBOL_VECTOR entry points.

In order to build the example, execute the SW_BUILD.COM procedure, then run the SW_TEST.EXE image.

When you create shared images on OpenVMS systems, you must export guard variables for template static data members or for static variables defined in inline functions. These guard variables, which are prefixed by __SDG and __LSG respectively, ensure that static data is initialized only once. You must

also export the static variables in inlined functions and template static data members from the shared image so that they have only one definition.

3.7. Hints for Designing Upwardly Compatible C++ Classes

If you produce a library of C++ classes and expect to release future revisions of your library, you should consider the upward compatibility of your library. Having your library upwardly compatible makes upgrading to higher versions of your library easier for users. And if you design your library properly from the start, you can accomplish upward compatibility with minimal development costs.

The levels of compatibility discussed in this section are as follows:

1. Source compatibility
2. Link compatibility
3. Run or binary compatibility

The format in which your library ships determines the levels of compatibility that apply:

Library Format	**Compatibility Level**
Source format	Source compatibility only
Object format	Source and link compatibility
Shareable library format	All three kinds of compatibility

If you break compatibility between releases, you should at least document the incompatible changes and provide hints for upgrading between releases.

3.7.1. Source Compatibility

Achieving source compatibility means that users of your library will not have to make any source code changes when they upgrade to your new library release. Their applications will compile cleanly against your updated header files and will have the same run-time behavior as with your previous release.

To maintain source compatibility, you must ensure that existing functions continue to have the same semantics from the user's standpoint. In general, you can make the following changes to your library and still maintain source compatibility:

- Add new data members and classes.
- Add new virtual and nonvirtual functions (as long as they do not change overload resolution of existing calls).
- Loosen protection.
- Change inline functions to out-of-line, and out-of-line functions to inline.
- Change the implementation of functions.
- Add arguments with default values to existing member functions.

3.7.2. Link Compatibility

Achieving link compatibility means that users of your library can relink an application against your new object or shareable library and not be required to recompile their sources.

What Can Change

To maintain link compatibility, the internal representation of class objects and interfaces must remain constant. In general, you can make the following changes to your library and still maintain link compatibility:

- Change anything that is invisible to the user.
- Change the implementation of an out-of-line function.
- Loosen protection.
- Add a new nonvirtual member function (as long as it does not change overload resolution for existing calls).

What Cannot Change

Because the user may be linking object modules from your previous release with object modules from your new release, the layout and size of class objects must be consistent between releases. Any user-visible interfaces must also remain unchanged; even the seemingly innocent change of adding `const` to an existing function will change the mangled name and thus break link compatibility.

The following are changes that you cannot make in your library:

- Add, move, or delete data members.
- Add, move, or delete virtual functions.
- Change the signature of virtual and nonvirtual functions.
- Remove nonvirtual functions.
- Change inline function definitions.
- Change functions from out-of-line to inline.

Designing Your C++ Classes for Link Compatibility

Although the changes you are allowed to make in your library are severely restricted when you aim for link compatibility, you can take steps to prepare for this and thereby reduce the restrictions. VSI suggests using one of the following design approaches:

- Set aside dummy (reserved-for-future-use) data fields and virtual functions within your classes. This assumes you can foresee how much your classes will grow and change in the future.
- Add a level of indirection to hide your virtual functions and data fields from the user. This lets you add and change data fields and virtual functions without affecting the library user; however, there may be some disadvantages such as in performance. This approach is detailed in *Effective C++*, by Scott Meyers.

3.7.3. Run Compatibility

Achieving run compatibility means that users of your library can run an application against your new shareable library and not be required to recompile or relink the application.

This requires that you follow the guidelines for link compatibility as well as any operating system guidelines for shareable libraries. On OpenVMS systems, you need to create an upwardly compatible shareable image using a transfer vector on OpenVMS VAX and a symbol table on OpenVMS. Refer to the *VSI OpenVMS Linker Utility Manual* for information on creating a shareable image.

The Annotated C++ Reference Manual offers some advice on compatibility issues. Another good reference is *Designing and Coding Reusable C++* by Martin D. Carroll and Margaret E. Ellis.

Chapter 4. Porting to I64 Systems

This chapter describes some of the differences and restrictions you might encounter when porting the VSI C++ compiler to an I64 system. For a summary of new and changed features supported by this version of the compiler on both OpenVMS Alpha and I64 systems, see the Preface of this manual. For any known issues, see the C++ release notes.

VSI C for OpenVMS I64 uses a new technology base that differs substantially from VSI C++ for OpenVMS Alpha and VSI C for OpenVMS I64. Although a great deal of work has been done to make it highly compatible with VSI C++ for OpenVMS Alpha, there are a number of differences that you will likely notice. Among them are:

- Resource requirements.

  Programs will usually use more memory, both at compile time and at run time. See Section 4.1.2.

- Floating-point behaviors.

  The default on I64 systems is /FLOAT=IEEE/IEEE_MODE=DENORM_RESULTS. Consistent use of qualifiers across compilations is required. See Section 4.1.6.

- Simplified instantiation without repository. See Section 4.1.9.

- No inline assembly language. See Section 4.1.7.

- String literal type change.

  For standards-compliance and link compatibility between compiler dialects, ordinary string literals now have the type "array of const char" in all compiler dialects on I64 systems in all compiler modes and on Alpha systems in /MODEL=ANSI mode.

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

4.1. Compiler Considerations

This section describes porting considerations for the C++ compiler for OpenVMS I64 systems. See Section 4.2 for considerations for the standard library, language run-time support library, and class library.

4.1.1. Messages

The move from Alpha systems to I64 systems may 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.

There have also been some changes in the /WARNINGS qualifier for both platforms. These include bug fixes and improved compatibility with the C compiler. For a summary of these changes, see the New and Changed Features section of the Preface.

4.1.2. Quotas

The C++ compiler for I64 systems 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, I64 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 I64 systems (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 I64 systems (doubling again, or halving the increment, until your biggest programs and compilations have just enough memory, and then adding an appropriate safety margin).

4.1.3. Dialect Changes

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.

4.1.4. ABI/Object Model changes

The object model and the name mangling scheme used by the C++ compiler on I64 systems are different from those used on Alpha systems (different from both /MODEL=ARM and /MODEL=ANSI). The I64 compiler uses the interface described by the I64 Application Binary Interface (ABI).

The C++ 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 in 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.

4.1.5. Command-Line Qualifiers

This section describes C++ command-line qualifier differences to be aware of on I64 systems.

Qualifiers/Features Not Supported on I64 Systems

The following command-line qualifiers and features are not supported on C++ for I64 systems, 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.
- /INSTRUCTION_SET=NOFLOATING_POINT is not available on I64 systems. If it is specified, a warning message is issued, and /INSTRUCTION_SET=FLOATING_POINT is used.
- /L_DOUBLE_SIZE=64 is not available on I64 systems. If it is specified, a warning message is issued, and /L_DOUBLE_SIZE=128 is used.

Changed/Ignored Qualifiers

A number of other qualifiers not supported on I64 systems 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 and 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 runtime 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 I64 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 I64 systems, 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 I64 systems 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.

- /DEBUG

  The following debug options are ignored:

  /DEBUG=NOSYMBOLS
  /DEBUG=NOTRACEBACK

- /DISTINGUISH_NESTED_ENUMS

This qualifier only modified the behavior of programs compiled with /MODEL=ARM. Since that model is not supported on the I64 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 I64 systems is IEEE_FLOAT.

  See Section 4.1.6 for more information about floating-point behavior on I64 systems.

- /IEEE_MODE

  The default for /IEEE_MODE on I64 systems 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 4.1.6 for more information.

- 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 I64 systems, 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 is not very useful on I64 systems. Because of this, specifying an unroll value greater than 0 is simplified to mean that simple loop unrolling is enabled. On I64 systems, the user does not have the ability to control the number of times a loop is unrolled.

  - /OPTIMIZE=LIMIT_INLINE is ignored.

- /TEMPLATE

  See Section 4.1.9 for information on template instantiation.

- /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++:

- /[NO]PURE_CNAME

  This qualifier 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 3.1 for more information.

4.1.6. Floating Point

This section describes floating-point behavior on I64 systems.

IEEE Now the Default

On OpenVMS I64 systems, /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 I64 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 I64 systems, 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 I64 systems 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 I64 systems.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on an I64 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 I64 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

I64 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 I64 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 I64 systems, 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.

More Information

For more information on I64 floating-point behavior, see the white paper *OpenVMS floating-point arithmetic on the Intel Itanium architecture*.

4.1.7. Intrinsics and Builtins

The C++ built-in functions available on OpenVMS Alpha systems are also available on I64 systems, with some differences. Section C.2 documents these differences and describes the built-in functions that are specific to I64 systems.

4.1.8. ELF

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

On OpenVMS I64 systems, 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 VSI OpenVMS Alpha to VSI 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.

4.1.9. Templates

This section describes template instantiation for I64 systems.

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 I64 systems 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 I64 systems.

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.

Also see Chapter 5.

4.1.10. 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.

4.1.10.1. 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 I64 systems, 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 I64 systems is:

```
Alpha:            I64:
Created           Created
In terminate      Destroyed
                  In terminate
```

4.1.10.2. 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.

4.1.10.3. terminate() Incorrectly Called

The C++ I64 compiler incorrectly calls `terminate()` when, during unwinding, the destruction of an object results in an exception, even if this exception is caught within the destructor.

For example, consider the following program:

```
extern "C" int printf(const char *,...);
struct killit {
  killit () {}
  ~killit () {
    try {
      throw 11;
    } catch (int i) {
      printf("caught %d\n", i);
    }
  }
};
int main () {
  try {
 killit local;
    throw 33;
  } catch (const int &i) {
    printf("caught int: %d\n", i);
  }
  return 0;
}
```

The expected output for the above example is:

```
caught 11
caught int: 33
```

But the executable produced by the C++ I64 compiler calls `terminate()`.

In cases where the expression to be thrown has been evaluated, but before the exception can be caught: if a called user function such as a copy constructor exits through an uncaught exception, then the compiler incorrectly attempts to match this latter exception object type to the handlers in enclosing `try` blocks in succession, instead of calling `terminate()`.

Further, the function `uncaught_exception` returns FALSE while in the called user function described above.

For example, consider the following program:

```
extern "C" int printf(const char *,...);
extern "C" int exit(int);
#include <exception>
void announce () {
    printf("Terminated!\n");
    exit(0);
}
class Y {
  public:
    Y () { printf ("construct Y\n"); }
    Y(Y &rhs) {
      printf ("copy Y\n");
      printf ("uncaught_exception = %s\n", std::uncaught_exception() ?
                                           "TRUE" : "FALSE");
```

```
      throw 20;
    }
    ~Y () { printf ("destruct Y\n"); }
};

void cxx_func () {
    Y OBJ2;
printf ("In cxx_func\n");
    try {
        throw OBJ2;
    } catch (const Y &) {
        printf("Caught Y &\n");
    } catch (int i) {
        printf("Caught %d\n", i);
    }
    printf ("leaving cxx_func\n");
}
main () {
    std::set_terminate(announce);
    cxx_func();
    printf ("Leaving main\n");
}
```

The expected output in the above example is:

```
construct Y
In cxx_func
copy Y
uncaught_exception = TRUE
Terminated!
```

But the executable produced by the C++ I64 compiler outputs:

```
construct Y
In cxx_func
copy Y
uncaught_exception = FALSE
Caught 20
leaving cxx_func
destruct Y
Leaving main
```

The C++ I64 compiler also incorrectly calls `terminate()` when a destructor invoked during stack unwinding exits with an exception that violates its own exception specification, instead of calling `unexpected()`.

Consider the following program:

```
#include <exception>
extern "C" void exit(int);
extern "C" int printf(const char *,...);
void announce2 () {
    printf("announce2: Unexpected!\n");
    exit(0);
}
void announce1 () {
    printf("announce1: Terminated!\n");
    exit(0);
}
```

```
class C {
public:
   C() { printf("C()\n"); }
  ~C() throw() { std::set_unexpected(announce2); printf("~C()\n"); throw
 3; }
};
void foo() {
    C c;
    printf("throwing ...\n");
    throw 5;
}
main() {
    std::set_terminate(announce1);
    foo();
}
```

In the above example, the expected output is:

```
C()
throwing ...
~C()
announce2: Unexpected!
```

But the executable produced by the C++ I64 compiler outputs:

```
C()
throwing ...
~C()
announce1: Terminated!
```

4.1.10.4. Problem in unexpected() Behavior

When a user-defined `unexpected()` routine throws or rethrows an exception, the compiler incorrectly checks the exception specification of the caller of the routine instead of that of the routine itself, which did not allow the exception in its exception specification.

Consider the following program:

```
#include <exception>
#include <cstdlib>
extern "C" int printf(const char *, ...);
void my_unex() {
  printf("In my unex\n");
  throw;
}
void my_term() {
  printf("In my term\n");
  std::exit(0);
}
void foo() throw() {  // spec not checked with second rethrow
  printf("In foo\n");
  throw 7;
}
void bar() throw(int) {  // this spec checked with second rethrow
printf("In bar\n");
  foo();
}
void foo2() throw(std::bad_exception) {  // spec not checked with first
 rethrow
```

```
    printf("In foo2\n");
  throw 5;
}
int main() {
  std::set_unexpected(my_unex);
  std::set_terminate(my_term);
  try {
    foo2();
  } catch (int i) {
    printf("Caught %d\n", i);
} catch (std::bad_exception &) {
    printf("Caught bad_exception\n");
  }
  try {
    bar();
  } catch (int i) {
    printf("Caught %d\n", i);
  } catch (...) {
    printf("Caught ...\n");
  }
  return 0;
}
```

In the above example, expected output is:

```
In foo2
In my unex
Caught bad_exception
In bar
In foo
In my unex
In my term
```

But the compiler produces:

```
In foo2
In my unex
Caught 5
In bar
In foo
In my unex
Caught 7
```

4.2. Library Changes

For I64 systems, 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 I64 systems.

The C++ class library on I64 systems is based on the same code as the C++ class library on Alpha systems. The major change in the C++ class library for I64 systems is the removal of the tasks and complex packages.

4.2.1. Library Reorganization

The standard library, language run-time support library, and class library have been reorganized for I64 systems.

4.2.1.1. 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 I64 systems, 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 I64 systems.

4.2.1.2. 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 I64 systems, 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.

4.2.2. Language Run-Time Support Library

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.

4.2.3. 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.

4.2.4. Standard Library

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

4.2.4.1. Changes

There are two major changes in the C++ standard library for I64 systems 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 I64 systems, 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.

4.2.4.2. Library Headers

While the change in the library distribution model should be transparent to customers (except that application images are much smaller on I64 systems), users on I64 systems 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 I64 systems, 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 I64 systems 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 I64 system, it is always a good idea to check that all necessary library headers are included.

4.2.4.3. 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.

4.2.4.4. Known Issues and Restrictions

The following are known issues with C++ for OpenVMS I64 systems:

- The C++ Standard Library IOStreams expect floating-point values in the IEEE format, which is the default floating-point format on I64 systems. Using the Standard Library IOStreams for processing floating-point values in a format other than IEEE (for example, in a program compiled with the /FLOAT=G_FLOAT or /FLOAT=D_FLOAT qualifier) is not supported. The C++ class library does not have this restriction.

4.2.4.5. Differences Between Alpha and I64 Systems

The following are differences between the I64 and Alpha standard libraries:

- 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 I64 systems the C++ standard library does not do it.

- On I64 systems, 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 I64 systems, this must be changed to:

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

- On I64 systems, 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 I64 systems:

  ```
  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 I64 systems, 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 I64 systems, it gives:

```
1
32767
```

Note that on I64 systems, the failbit for the stream is set.

According to the C++ Standard - 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 I64 systems is standard-compliant.

- On Alpha systems, the `find` template function is implemented using `operator!=`. On I64 systems, 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 I64 systems, 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 I64 systems. The behavior on I64 systems 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 I64 systems, an attempt to write into a stream opened for read (ios::in), causes the stream badbit bit to be set.

  On both Alpha and IPF systems, 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 I64 systems is more reasonable—at least there is an indication that something was wrong.

- On I64 systems, 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 I64 systems:

  ```
  #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 I64 systems, 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 I64 systems 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 I64 systems is correct.

- On I64 systems, 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 I64 systems, 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 I64 systems, 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 I64 systems, class std::fpos 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 IPF
}
```

- On OpenVMS Alpha systems, in the default built-in C locale, the monetary facets use values typically found in the en_US locale (English in the United States). For example, on Alpha the default national currency string is "$". On I64 systems, in any locale, including the C locale, the monetary facets use values defined by the locale.

  Consider the following sample program:

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iostream>
#include <locale>
#include <stdexcept>
#include <stdlib.h>
#if defined(__osf__) || defined(__vms)
#  define UK_LOCALE "en_GB.ISO8859-1"
#elif defined(__linux)
#  define UK_LOCALE "en_GB"
#else
#  error unknown platform
#endif
using namespace std;
void outputSym(ostream& os) {
  locale loc = os.getloc();
  const moneypunct<char,false>& mpunct =
      use_facet<moneypunct<char,false> >(loc);
  os << "currency symbol is: " << mpunct.curr_symbol() << endl;
}
```

  This program prints two lines: the national currency symbol in the C locale and the national currency symbol in the en_GB locale (English in Great Britain).

- Consider a program using the C++ Standard Library IOStreams, like `x.cxx` below, that writes to `cout`, but not to `cerr` or `clog`:

```
x.cxx
-----
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
#include <iostream>
main() {
  std::cout << "hello, world" << std::endl;
}
```

  On OpenVMS Alpha systems, if such a program is invoked with SYS$OUTPUT redirected to a file and SYS$ERROR defined as SYS$OUTPUT, a single version of the output file is created.

  On I64 systems, by default, two versions of the file are created: one for SYS$OUTPUT and another for SYS$ERROR. To get Alpha behavior on an I64 system, define logical name DECC$COMMON_STDERR_STDOUT to ENABLE. The following command file shows the definition:

```
x.com
-----
$ if f$search("x.dat") .nes. "" then delete x.dat;*
$ define/user sys$output x.dat
$ define/user sys$error sys$output
$ if f$getsy("arch_name") .eqs. "IA64" then -
    define/user decc$common_stderr_stdout enable
$ run x.exe
```

4.3. CXXLINK Changes

Because of changes in the architecture on I64 systems, 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 I64 systems, there is no LIBCXXSTD.OLB, which is the object library where the C++ standard library for OpenVMS Alpha resides. See Section 4.2.4 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 4.2 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 4.1.9 for more information.

4.4. Installation

VSI C++ is installed using PCSI for OpenVMS I64 systems.

To install VSI C++ for OpenVMS I64 systems, 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, the C++ release notes will be available at:

SYS$HELP:CXX.RELEASE_NOTES

Here is a sample installation log:

```
$ PRODUCT INSTALL CXX/SOURCE=NODE1$::DEV1$:[I64_CPP_KIT]
The following product has been selected:
    HP I64VMS CXX T7.0-9                    Layered Product [Installed]
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 T7.0-9: HP C++ for OpenVMS Industry Standard
    Copyright 2004 Hewlett-Packard Development Company, L.P.
    This software product is sold by Hewlett-Packard Company
    PAKs used: CXX or CXX-USER
Do you want the defaults for all options? [YES]
      Copyright 2004 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 T7.0-9                    DISK$ICXXSYS:[VMS$COMMON.]
Portion done: 0%...90%...100%
The following product has been installed:
    HP I64VMS CXX T7.0-9                    Layered Product
%PCSI-I-IVPEXECUTE, executing test procedure for HP I64VMS CXX T7.0-9 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully
HP I64VMS CXX T7.0-9: HP C++ for OpenVMS Industry Standard
The compiler is now available from the command line of newly created
processes.
      To enable access to the compiler from the command line of a currently
      running process (such as this one), execute:
      SET COMMAND/TABLE=SYS$COMMON:[SYSLIB]DCLTABLES
  The release notes are located in the file SYS$HELP:CXX.RELEASE_NOTES
  for the text form and SYS$HELP:CXX_RELEASE_NOTES.PS for the postscript
```

```
  form.
$
```

Chapter 5. Using Templates

A C++ **template** is a framework for defining a set of classes or functions. The process of **instantiation** creates a particular class or function of the set by resolving the C++ template with a group of arguments that are themselves types or values. For example:

```
template <class T> class Array {
    T *data;
    int size;
public:
    T &operator[](int);
    /* … */
};
```

The code in this example declares a C++ class template named `Array` that has two data members named `data` and `size` and one subscript operator member function. Array`<int>` instantiates `Array` with type `int`. This instantiation generates the following class definition:

```
class Array {
    int *data;
    int size;
public:
    int &operator[](int);
    /* … */
};
```

The compiler supports instantiation of C++ class, function, and static data member templates. The following sections describe using templates with Version 6.0 compilers or later. To understand the differences between the current compiler and Version 5.*n* and to migrate from Version 5.*n* to current compilers, see the appendix on migrating from 5.*n* compilers.

5.1. Template Instantiation Model

For every template used in a C++ program, the compiler must create an instantiation for that template. How the compiler does this is referred to as the **template instantiation model**. The template instantiation models differ on the why, what, when, where, and how a template is instantiated. The following outline gives a framework to compare the different models.

1. Why

   A template can be instantiated **implicitly** when it is used, **manually** by specific request, or both.

2. What (part of the template is instantiated)

   For a template class, each member of the template can be instantiated separately, or if one member is instantiated then all members are instantiated.

3. When

   Instantiation can occur at compile time or link time. Version 6.0 or later compilers support only compile-time instantiation.

4. Where

   Templates can be instantiated in the object in which they are referenced or in separate objects that are stored in a repository.

5. How

   Templates can be instantiated with different linkages. They can be local, global, or **COMDAT** (I64 only). A COMDAT is like a weak global definition, but in addition to permitting duplicate definitions, the linker attempts to eliminate all of the duplicates, saving space in the image.

The numbers in the preceding list are used in subsequent paragraphs to indicate which aspect of the template instantiation model framework is being referenced.

For complex systems, choosing a template instantiation model is a space, time, and build-complexity issue that can be considered for optimizing build speed and reducing program size. The default model, referred to as **automatic template instantiation**, is standard-compliant and should work transparently for most users. VSI recommends this model.

Automatic template instantiation:

- Instantiates templates when they are used (1),
- Instantiates only the pieces of a class that are used (2),and
- Occurs at compile time (3).

Template instantiation on Alpha and I64 systems differ on the where and the how:

- On Alpha systems, templates are instantiated in a repository (4) using global linkage (5)
- On I64 systems, templates are instantiated in the objects that refer to them (4) as COMDATs (5).

The compiler, CXXLINK, and linker all work together to assure that all templates used in a program are instantiated and transparently linked into the final image.

Even when using automatic template instantiation, manual instantiation (1) is also permitted. When using the default model, manually instantiated templates are placed in the object where the manual instantiation occurs (4). On Alpha systems, they have global linkage; on I64 systems, they are COMDATs (5).

See Table 5.1 for a summary of each template instantiation model's What, Where, and How for both implicit and manual instantiation (the "Why").

5.2. Manual Template Instantiation

The compiler provides the following methods to instantiate templates manually:

- Using the `#pragma` preprocessor directives.

  Using an instantiation pragma to direct the compiler to instantiate a specific template, as described in Section 5.2.2.

- Using explicit template instantiation syntax.

  The C++ language now defines specific syntax for specifying that a template should be instantiated. See The Annotated C++ Reference Manual.

  VSI strongly recommends using the explicit template instantiation syntax when possible.

- Using the command-line qualifier method.

This method directs the compiler to instantiate templates at compile time in the user's object file. Several qualifiers are available to control linkage and extent of template instantiation. For more information about these qualifiers, see Section 5.2.3.

5.2.1. Mixing Automatic and Manual Instantiation

Object files that have been compiled using manual instantiation can be linked freely with objects that have been compiled using automatic instantiation. To ensure that the template instantiations needed by the files compiled with automatic instantiation are provided, the application must be linked using automatic instantiation, and the appropriate repositories must be present.

When a template instantiation is present in an explicitly named object file or object library it takes precedence over the same named instantiation in a repository.

5.2.2. Instantiation Directives

The next sections describe the following instantiatation directives:

```
#pragma define_template
#pragma instantiate_template
#pragma do_not_instantiate_template
```

5.2.2.1. #pragma define_template

The compiler provides a mechanism for manual instantiation, using the `#pragma define_template` directive. This directive lets you tell the compiler what class or function template to instantiate in conjunction with the actual arguments with which the template is to be instantiated.The `#pragma define_template` directive has the following format:

`#pragma define_template identifier` [<template_arguments>]

Identifier is the name of the class or function template that the compiler is directed to instantiate at compile time. For the instantiation to succeed, the definition of the template must appear before the `#pragma define_template` directive.

Template_arguments is a list of one or more actual types that correspond to the template parameters for the particular class or function template being instantiated. Whatever type is specified is used as the type for the instantiation.

The following is an example of a valid template manual instantiation:

```
//main.cxx
#include <stdlib.h>
template <class T> void sort (T*);
int a1[100];
float a2[100];
int main()
{
    sort(a1);
    sort(a2);
    return EXIT_SUCCESS;
}
//sort.cxx
template <class T> void sort (T *array)
```

```
{
    /* body of sort */
}
#pragma define_template sort<int>
#pragma define_template sort<float>
```

To compile and link these sources, enter the following command:

```
CXXLINK main.cxx,sort.cxx /TEMPLATE_DEFINE=(NOAUTO)
```

When you use `#pragma define_template` or explicit instantiation, only the specified template is instantiated; templates to which it refers because of member types or base classes are not instantiated.

Sorting an array of template class elements requires the use of additional pragmas for the module `sort.cxx`. For example:

```
template <class T> void sort (T* array)
{
    /*body of sort*/
}
template <class T> class entity {
public:
    T member;
    int operator < (const entity<T> &) const;
}
template <class T>
int entity<T>::operator < (const entity<T> &operand) const
{
     return member < operand.member;
}
int a1[100];
float a2[100];
entity<int> a3[100];
#pragma define_template sort<int>
#pragma define_template sort<float>
#pragma define_template sort<entity<int> >
void sort_all_arrays ()
{
    sort(a1);
    sort(a2);
    sort(a3);
}
```

The `define_template` pragma is position sensitive. If a `define_template` occurs lexically before a function, member function, or static data member template definition, the compiler is unable to instantiate the corresponding template because the body of that template is not present before the pragma directive.

The compiler instantiates all instances of `sort` and of `entity::operator <` needed for this compilation unit.

To organize a program to use the `define_template` pragma, you can place the declarations of class and functions templates into header files, and instantiate all instances of a particular template from a single compilation unit. The following example shows how to do this:

```
// sort.h
#include <stdlib.h>
template <class T> void sort (T*);
```

```
// entity.h
template <class T> class entity {
public:
    T member;
    int operator < (const entity<T> &) const;
};
// main.cxx
#include "sort.h"
#include "entity.h"
int a1[100];
float a2[100];
entity<int> a3[100];
int main()
{
    sort(a1);
    sort(a2);
    sort(a3);
    return EXIT_SUCCESS;
}
// sort.cxx
#include "sort.h"
#include "entity.h"
template <class T> void sort (T* array)
{
    /*body of sort*/
}
#pragma define_template sort<int>
#pragma define_template sort<float>
#pragma define_template sort<entity<int> >
```

Compiling the following file provides a definition of `entity::operator <` with type `int`:

```
// entity.cxx
#include "entity.h"
template <class T>
int entity<T>::operator < (const entity<T> &operand) const
{
     return member < operand.member;
}
#pragma define_template entity<int>
```

To compile this example, issue the following command:

```
cxxlink main.cxx,sort.cxx,entity.cxx
```

If the program uses other instantiations of `entity` in other compilation units, you can provide definitions of `operator <` for those entities by adding `define_template` pragmas to `entity.cxx`. For example, if other compilation units use the following instantiations of `entity`, appending the following pragmas to `entity.cxx` causes the compiler to generate instantiations of `operator <` for those requests of entity:

```
entity<long> and entity< entity<int> >,
#pragma define_template entity<long>
#pragma define_template entity< entity<int> >
```

Like any other pragma, the `#pragma define_template` pragma must appear on a single line. Pragmas may be continued on multiple lines by escaping the end of line with a backslash (\ ) as with other preprocessor statements.

5.2.2.2. #pragma instantiate and #pragma do_not_instantiate

The compiler also provides several pragmas that provide fine control over the instantiation process. Instantiation pragmas, for example, can be used to control the instantiation of specific template entities or sets of template entities. There are two instantiation pragmas:

- The `instantiate` pragma causes a specified entity to be instantiated, similar to the `define_template` pragma. It provides finer instantiation control than `define_template` when instantiating function templates.
- The `do_not_instantiate` pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.

The argument to the instantiation pragma can be:

a template class name	`A<int>`
a template class declaration	`class A<int>`
a member function name	`A<int>::f`
a static data member name	`A<int>::i`
a static data declaration	`A<int>::i`
a member function declaration	`void A<int>::f(int, char)`
a template function declaration	char* f(int, float)

A pragma in which the argument is a template class name (for example, `A<int>` or `class A<int>` is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class, a given member function or static data member may be excluded using the `do_not_instantiate` pragma. For example:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the instantiate pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
#include <stdlib.h>

int main()
{
  int     i;
  double  d;
  f1(i);
  f1(d);
```

```
  g1(i);
  g1(d);
  return EXIT_SUCCESS;
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int) // error - no body provided
```

The functions `f1(double)` and `g1(double)` are not instantiated (because no bodies were supplied) but no errors are produced during the compilation (if no bodies are supplied at link time, a linker error is produced).

A member function name (for example, `A<int>::f` can be used as a pragma argument only if it refers to a single user-defined member function (that is, not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma must not be a compiler-generated function, an inline function, or a pure virtual function.

5.2.3. Using Command Qualifiers for Manual Instantiation

Alternatively, you could use the /TEMPLATE_DEFINE qualifier to instantiate templates manually.

Considering the previous examples in this section, you can use this qualifier to supply definitions of `sort<int>`, `sort<float>`, and `sort<entity><int>` by compiling the following file using /TEMPLATE_DEFINE=ALL:

```
// sort.cxx
#include "sort.h"
#include "entity.h"
template <class T>
static sort (T* array)
{
    /*body of sort*/
}
static void function_never_used ()
{
    int a1[100];
    float a2[100];
    entity<int> a3[100];
    sort(a1);
    sort(a2);
    sort(a3);
}
```

You can use the /TEMPLATE_DEFINE=USED and /TEMPLATE_DEFINE=LOCAL qualifiers for manual template instantiation. The /TEMPLATE_DEFINE=USED qualifier acts like /TEMPLATE_DEFINE=ALL, except that only those template instantiations that are referenced in the source file are actually instantiated. The /TEMPLATE_DEFINE=LOCAL qualifier acts like /TEMPLATE_DEFINE=USED, except that the templates are instantiated with internal linkage. This provides a simple way to build applications but creates executables that are larger than necessary. It also fails if the template classes being instantiated have static data members.

You can use
the /TEMPLATE_DEFINE=ALL_REPOSITORY, /TEMPLATE_DEFINE=USED_REPOSITORY,
and /TEMPLATE_DEFINE=IMPLICIT_LOCAL qualifiers to create preinstantiation libraries. See
Section 5.6.

5.3. Using Template Object Repositories *(Alpha only)*

In automatic template instantiation mode, the compiler attempts to instantiate every referenced template at compile time. For automatic instantiation to work, at least one compilation that references a template function must be able to find the template definition. There is no restriction on where a template can be declared or defined, as long as the definition is visible to the compilation unit. You can use implicit inclusion to find it.

The compiler writes instantiation object files to a directory called the repository; file names are based on the names of the entities being instantiated. The default repository is `[.cxx_repository]`.

5.3.1. Specifying Alternate Repositories

You can use the /REPOSITORY command-line qualifier to specify one or more alternate repository directories. The first repository named is the read-write repository into which the compiler writes instantiation objects when processing. At link time, all repositories are read only. There is one object file in the repository for each instantiated template function, for each instantiated static data member, and for each virtual table required for virtual functions.

When the program is linked, the linker searches the repositories for needed template instantiations.

5.3.2. Reducing Compilation Time with the /TEMPLATE_DEFINE=TIMESTAMP Qualifier

To keep instantiations up to date, the compiler always instantiates templates by default, even if the required template already exists in the respository. However, in environments that share many templates among many sources, this process can increase compilation time.

In these environments, users can specify the /TEMPLATE_DEFINE=TIMESTAMP qualifier to override the default behavior and thereby reduce compilation time. This qualifier causes the compiler to create a timestamp file named `TIMESTAMP.` in the repository. Thereafter, instantiations are added or regenerated only if needed; that is, if they do not alreay exist, or if existing ones are older than the timestamp.

The /TEMPLATE_DEFINE=TIMESTAMP qualifier is immediately useful when building a system from scratch, starting with an empty repository. It avoids reinstantiating unchanged code and is totally safe, because all required instantiations are generated and up to date.

Incremental application building is normally done without this qualifier, so that new instantiations overwrite earlier ones as sources are recompiled.

Although the /TEMPLATE_DEFINE=TIMESTAMP qualifier is intended mainly for initial builds, you can use it for ongoing development in a structured way. Because the compiler creates a new timestamp file only if one does not already exist, you must remove or modify any existing timestamp file before making changes to your code base. This procedure ensures that all subsequent compilations generate up-to-date instantiations.

In the following example, the file is removed before and immediately after the compilation of `a.cxx`, `b.cxx`, and `c.cxx`.

```
$ DELETE [.cxx_repository]TIMESTAMP.;*
$ CXX /TEMPLATE_DEFINE=TIMESTAMP a.cxx
$ CXX /TEMPLATE_DEFINE=TIMESTAMP b.cxx
$ CXX /TEMPLATE_DEFINE=TIMESTAMP c.cxx
$ DELETE [.cxx_repository]TIMESTAMP.;*
```

All instantiations needed by `a.cxx`, `b.cxx`, and `a.cxx` are generated only once, as opposed to the default scheme, in which they would be generated three times if all three modules used the instantiations.

Specifying the /TEMPLATE_DEFINE=VERBOSE qualifier causes the compiler to emit an informational message naming the instantiation and repository file being skipped in this mode.

5.3.3. Compiling Programs with Automatic Instantiation

In general, the use of automatic template instantiation is transparent to the user. Automatic template instantiation is enabled by default. The following commands are equivalent:

```
CXX file.cxx
CXX/TEMPLATE_DEFINE=(AUTO,PRAGMA) file.cxx
CXX/REPOSITORY=[.CXX_REPOSITORY] file.cxx
```

These commands:

- Cause the compilation of the file `file.cxx`
- Create any instantiations that are required whose definitions are visible to the compiler
- Create an executable, `a.out`, by linking together the generated object file and any instantiations required from the repository

You can specify the repository explicitly with the /REPOSITORY qualifier. For example:

```
CXX /REPOSITORY=C$:[PROJECT.REPOSITORY] file.cxx
```

This command compiles `file.cxx`, produces an object file in the current directory, and puts instantiated template files in the directory C$:[PROJECT.REPOSITORY].

You can specify multiple directories using the /REPOSITORY qualifier. The first named repository is denoted as the read/write repository. The compiler writes instantiation files to this repository. The other repositories are denoted as read only repositories. They are searched by the link command as described in Section 5.3.4.

The compiler attempts to instantiate templates at compile time; therefore, any specialization used in the program must be declared in each file in which the specialization is referenced, to prevent the instantiation of the overridden template function.

If a template instantiation refers to a static function, that function is created as an external entry point in the primary object file, and the instantiation object file in the repository then refers to this `__STF` function.

If the template instantiation is linked into an application that does not have the original primary object file, an unresolved reference to the `__STF` function occurs. If this happens, recompile an object file that regenerates the instantiation or use manual instantiation to reinstantiate the template.

5.3.4. Linking Programs with Automatic Instantiation

When compiling and linking an application, you must use the same repositories in both the compilation and link steps.

If you name a repository explicitly in the compile step, you must also name it in the link step. For example:

```
CXX /REPOSITORY=[.MY_REPOSITORY] a.cxx,b.cxx
CXXLINK /REPOSITORY=[.MY_REPOSITORY] a.obj b.obj
```

If you use different repositories the compilation of the sources, you must specify all of them on the link step:

```
CXX /REPOSITORY=[.REPOSITORY1] a.cxx
CXX /REPOSITORY=[.REPOSITORY2] b.cxx
CXXLINK /REPOSITORY=([.REPOSITORY1],[.REPOSITORY2]) a.obj b.obj
```

At link time, the specified repositories are searched in the order given, to find the required instantiations. If you use several repositories, and if one of them is the default, you must specify all repositories on the link step:

```
CXX a.cxx
CXX /REPOSITORY=[.REPOSITORY2] b.cxx
CXX /REPOSITORY=([.CXX_REPOSITORY],[.REPOSITORY2]) a.obj b.obj
```

It is usually much easier and safer to use a single repository, because the same instantiations could potentially be present in multiple repositories, and it is possible to update some but not all repositories when changes are made to templates.

The CXXLINK step processes object files so that all the external symbol references are resolved. The objects are linked together in the following order:

1. The order in which object files and object libraries are specified on the command line.
2. If /NOTEMPLATE_PRELINK is specified, stop.
3. For each unresolved external, search the repositories in the order specified on the command line for an file that contains that external. If such a file is found, add it at the top of the list of object files being searched.
4. Link again and repeat Step 3 until no more externals are found or until no more object files are found in which to resolve the external.

Note the following:

- Instantiations that appear in explicitly linked object files or libraries hide instantiations in the repositories.
- Only template instantiations that are actually referenced in sources that can instantiate them appear in the repository. You must specify any other instantiations manually or use the /TEMPLATE_DEFINE=ALL_REPOSITORY qualifier.
- Instantiations are satisfied from the list of unsatisfied externals from the linking of specified files, but are linked at the beginning of those files. This means that they are linked in only if they are satisfied from no specified file, given the linker's file order behavior, and if they bring in any external references they need from the first library that satisfies them.

5.3.5. Creating Libraries

Creating libraries with object files created with automatic instantiations is relatively straightforward. You must decide where the instantiations that were generated automatically are provided to the users of the library. For applications that use the library to link successfully, all template instantiations that are needed by the code in the library must be available at link time. This can be done in two ways:

- Put the instantiations in the library. They hide the same named instantiations in any repositories or any libraries following the library on the command line.
- Provide a repository that contains the instantiations.

It is usually easiest to put the instantiations in the library. This is a good choice if the instantiations are internal to the library and are not instantiated directly by the user's code. To put the instantiations in the library, add all of the object files in the repositories required by the library into the library as shown in the following example:

```
CXX /REPOSITORY=[.lib_repository] a.cxx,b.cxx,c.cxx
LIBRARY/CREATE/OBJECT mylib
LIBRARY/INSERT/OBJECT mylib a.obj,b.obj,c.obj
LIBRARY/INSERT/OBJECT mylib [.lib_repository]*.OBJ
```

If the template instantiations can be overridden by the user, the templates should be provided in a repository that the user specifies after all the user's repositories. For the previous example, create the library as follows:

```
CXX /REPOSITORY=[.lib_repository] a.cxx,b.cxx,c.cxx
LIBRARY/CREATE/OBJECT mylib
LIBRARY/INSERT/OBJECT mylib a.obj,b.obj,c.obj
```

When linking the application, enter the CXXLINK command as follows:

```
CXXLINK user_code.obj,mylib/LIB
```

If some objects from `[.lib_repository]` are not contained in `mylib.olb`, specify `[.lib_repository]` as the last read-only repository on the line follows:

```
CXXLINK /REPOSITORY=([.cxx_repository],[.lib_repository])
        user_code.obj,mylib/LIB
```

You must explicitly name the repository when linking, even if it is the default repository `[.cxx_repository]`; `cxx` first satisfies all unresolved instantiations from `[.cxx_repository]`, and uses `[.lib_repository]` to resolve any remaining unresolved instantiations.

Only the instantiations that are required by the code in the library are generated in the library repository `lib_repository`. If you must provide other instantiations that you require but cannot instantiate, you must provide these instantiations using manual template instantiation or by specifying the qualifier /TEMPLATE_DEFINE=ALL_REPOSITORY.

5.3.6. Multiple Repositories

As shown in Section 5.5.3, multiple repositories can be specified to link an application. The first repository named is the read-write repository, and when compiling, the compiler writes instantiation object files into it. At link time, all repositories are read only.

The repositories are searched in a linear order, iteratively, and satisfy only the unresolved instantiations from each pass. That is, references from instantiations that are added in one pass are not resolved until the next pass. Consider the link line in the previous example:

```
mylib.olb
```

In this example, all references that could be resolved from `lib_repository` would be resolved in the first pass. Any reference arising from an instantiation in `lib_repository` in the first pass would be resolved by instantiations in `[.cxx_repository]` in the second pass.

5.4. Using COMDATS *(I64 only)*

The primary purpose of a template repository is to guarantee that only one copy of a template instantiation is included in a program. Another way to achieve this is to use COMDATs. COMDATs are special symbols that are globally visible; however, when the linker encounters a duplicate definition, it is removed. This allows the compiler to define templates directly in every object module that uses them.

The principal benefit of using COMDATS is build speed, but it also can simplify the build procedure:

- Compilation speed is improved because writing template instantiations into the current object is significantly faster then writing them into the repository, because of object overhead in the latter case.
- Link speed is improved because determining which templates to include from the template repository requires multiple passes of the linker.
- The build is simplified by eliminating the need to manage the template repository and explicitly extract objects.

COMDATs are implemented on I64 systems using ELF group sections. COMDATs are not implemented on Alpha systems because EOBJ does not support them. If EOBJ supported COMDATs, then they also would have been used on Alpha systems instead of the template object repository. Currently, there are no plans to implement COMDATs on Alpha systems.

Because templates instantiated using COMDATs exist in the same object where they are used, there are no special procedures for linking programs or creating libraries, except that a template can only be exported from a single shared library. If two shared libraries with the same exported template are linked together, a MULDEF will occur. This restriction also exists on Alpha systems.

5.5. Advanced Program Development and Templates

The following sections discuss templates in the context of advanced program development.

5.5.1. Implicit Inclusion

When implicit inclusion is enabled, the compiler assumes that if it needs a definition to instantiate a template entity declared in a `.h` or `.hxx` file, it can implicitly include the corresponding implementation file to obtain the source code for the definition.

If a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code, the compiler checks whether a file

`xyz.cxx` exists. If it does, the compiler processes it as if it were included at the end of the main source file.

When looking for a template definition, the compiler uses the following lookup order:

1. If the `#include` name for the header file containing the template declaration is specified with an absolute path name, look only in the directory specified by the path name.

2. If the `#include` for the header file containing the template declaration is specified with a relative path name, take the following action:

   - If the header file name is specified with double quotation marks (" ") and the /NOCURRENT_INCLUDE qualifier was not specified, append the relative path name to the directory containing the source file and search for files with the appropriate suffixes.

   - Otherwise, append the relative path name to all the `-I` directories and look in those resulting directories for files with the appropriate suffixes.

Note

A place containing a forward slash (/) character is considered to be a UNIX-style name. If the name in the `#include` directive also contains a "/" character that is not the first character and is not preceded by a an exclamation mark character (!) (that is, it is not an absolute UNIX-style pathname), the name in the `#include directive` is appended to the named place, separated by a "/" character, before applying the `decc$to_vms` pathname translation function.

For source files, the appropriate suffixes are, in order of preference: `.CXX`, `.C`, `.CC`, and `.CPP` or as defined by the /TEMPLATE_DEFINE=DEFINITION_FILE_TYPE qualifier.

The compiler ignores any file extension that does not begin with a dot (.).

The /TEMPLATE_DEFINE=DEFINITION_FILE_TYPE qualifier allows the user to define explicitly the file extensions to be used with implicit inclusion. For example:

```
CXX file.cxx /TEMPLATE_DEFINE=DEF=".CPP.CC"
```

This command searches for template definition files with the extensions `.CPP` and `.CC`.

5.5.2. Dependency Management

The compiler does no dependency management of its own. Because template instantiations are compiled when source files that reference those instantiations are compiled, those source files must be recompiled if the template declaration or definition changes.

The /MMS output from the compiler lists the implicitly included files, so that the MMS program can automatically recompile any source files that depend upon template files. If MMS is not being used, it is the user's responsibility to ensure that instantiations that have changed are recompiled. The user does so by recompiling at least one source file that references the changed instantiations.

The compiler does not check command line dependencies of template instantiations at link time. If you compile two different source files that instantiate a specific template with two different sets of qualifiers, the behavior is undefined. Use consistent qualifier settings for each build into each repository. Examples of qualifier settings that could cause unexpected results are as follows:

- /STANDARD=STRICT_ANSI. Use of guiding declarations is not allowed, and some templates might not be instantiated as they would be in other modes.
- /DEBUG. Debug information is generated for some instantiations and not for others. Be sure that is what you want.
- /NOMEMBER_ALIGNMENT. Some instantiations with this setting assume that classes have unaligned members; instantiations generated by compiling files with the default setting do not.

5.5.3. Creating a Common Instantiation Library

Because the automatic instantiation model has changed to a compile time model with Version 6.0, (see Sections 5.3.3 and 5.3.4), the procedure used to create a common instantiation library has also changed. This section describes the new procedure.

If you want to put all current instantiations into a common instantiation library, follow these steps:

1. Compile with the /TEMPLATE=VERBOSE qualifier and save the results to a file.
2. Edit that file and save the names that appear after the “automatically instantiating ...” string. You can ignore any messages about instantiating vtables. Put `#pragma instantiate` before each name.
3. Put the result of that edit into a separate source file and include at the top of the file any headers needed for template definitions.
4. Put matching `#pragma do_not_instantiate` (see Section 5.2.2.2) into the headers that define each of these template classes or functions.
5. Place each `#pragma do_not_instantiate` directive between an `#ifndef` of the form `#ifndef SOME_MACRO_NAME` and an `#endif`.
6. Compile the `inst.cxx` file with `SOME_MACRO_NAME` defined.
7. Link the source file with the resulting object file.

The following examples show how to create a common instantiation library for all the instantiations currently being automatically instantiated for this file.

```
// foo.cxx
#include <stdlib.h>
#include <vector>
#include "C.h"
int main() {
 vector<C> v;
 v.resize(20);
        return EXIT_SUCCESS:
}
// C.h
#ifndef __C_H
class C {};
#endif
```

Compiling with the /TEMPLATE=VERBOSE qualifier shows which instantiations occur automatically:

1. Place all these instantiations into a file called `inst.cxx` that is built separately or into a library:

```
// inst.cxx
#include <vector>
#include "C.h"
#pragma instantiate void std::vector<C, std::allocator<C >
 >::resize(unsigned long)
#pragma instantiate void std::vector<C, std::allocator<C >
 >::insert(C *, unsigned long, const C &)
#pragma instantiate void std::vector<C, std::allocator<C >
 >::__insert(C *, unsigned long, const C &, __true_category)
#pragma instantiate C *std::copy_backward(C *, C *, C *)
#pragma instantiate void std::fill(C *, C *, const C &)
#pragma instantiate C *std::copy(C *, C *, C *)
#pragma instantiate const unsigned long std::basic_string<char,
std::char_traits<char >, std::allocator<void> >::npos
```

2. Add these instantiations into `C.h` and change “instantiate” to “do_not_instantiate”. Add an `#ifndef`, so that when building `inst.cxx`, the compiler creates these instantiations in the `inst` object file:

```
#ifndef __C_H
class C {};
#ifndef __BUILDING_INSTANTIATIONS
#pragma do_not_instantiate void std::vector<C, std::allocator<C >
 >::resize(unsigned long)
#pragma do_not_instantiate void std::vector<C, std::allocator<C >
 >::insert(C *, unsigned long, const C &)
#pragma do_not_instantiate void std::vector<C, std::allocator<C >
 >::__insert(C *, unsigned long, const C &, __true_category)
#pragma do_not_instantiate C *std::copy_backward(C *, C *, C *)
#pragma do_not_instantiate void std::fill(C *, C *, const C &)
#pragma do_not_instantiate C *std::copy(C *, C *, C *)
#pragma do_not_instantiate const unsigned long
 std::basic_string<char, std::char_traits<char >,
 std::allocator<void> >::npos
#endif
#endif
```

3. Build the `inst` object file:

```
CXX/DEFINE=BUILDING_INSTANTIATIONS inst.cxx
```

4. Link with the `inst` object file. It will use instantiations from that file instead of creating them automatically:

```
cxx foo.cxx
cxxlink foo inst
```

To verify that your procedure worked correctly, you can remove all files from the `cxx_repository` subdirectory before you compile `foo.cxx`. This subdirectory should contain no instantiations after linking with the `inst` object file.

If you have an `inst.cxx` file that contains many instantiations and you do not want all the symbols in the `inst` object file to be put into a user's executable even if only some symbols are used, (as happens with archive libraries), you can either split the `inst.cxx` into many smaller source files, or specify the /DEFINE_TEMPLATE=USED_REPOSITORY qualifier to create the instantiations as separate object files in the repository (see Section 5.6). You must then link all the required individual object files in the repository into your library.

5.6. Command-Line Qualifiers for Template Instantiation

This section describes the C++ command-line qualifiers that specify the template instantiation model, and additional template-related qualifiers.

5.6.1. Instantiation Model Qualifiers

The following CXX command-line qualifiers specify the template instantiation model to be used. Specify only one:

/TEMPLATE_DEFINE=ALL

Instantiate all template entities declared or referenced in the compilation unit, including typedefs. For each fully instantiated template class, all its member functions and static data members are instantiated even if they were not used. Nonmember template functions are instantiated even if the only reference was a declaration. Instantiations are created with external linkage. Overrides /REPOSITORY at compile time. Instantiations are placed in the user's object file.

/TEMPLATE_DEFINE=ALL_REPOSITORY

Instantiate all templates declared or used in the source program and put the object code generated as separate object files in the repository. Instantiations caused by manual instantiation directives are also put in the repository. This is similar to /TEMPLATE_DEFINE=ALL except that explicit instantiations are also put in the repository, rather than than an external symbol being put in the main object file. This qualifier is useful for creating a pre-instantiation library.

/TEMPLATE_DEFINE=[NO]AUTOMATIC

/TEMPLATE_DEFINE=AUTOMATIC directs the compiler to use the automatic instantiation model of C++ templates.

/TEMPLATE_DEFINE=NOAUTOMATIC directs the compiler to not implicitly instantiate templates.

/TEMPLATE_DEFINE=AUTOMATIC is the default.

/TEMPLATE_DEFINE=IMPLICIT_LOCAL

Same as /TEMPLATE_DEFINE=LOCAL, except manually instantiated templates are placed in the repository with external linkage. This is useful for build systems that need to have explicit control of the template instantiation mechanism. This mode can suffer the same limitations as /TEMPLATE_DEFINE=LOCAL. This mode is the default when /STANDARD=GNU is specified.

/TEMPLATE_DEFINE=LOCAL

Similar to /TEMPLATE_DEFINE=USED except that the functions are given internal linkage. This qualifier provides a simple mechanism for getting started with templates. The compiler instantiates as local functions the functions used in each compilation unit, and the program links and runs correctly (barring problems resulting from multiple copies of local static variables). However, because many copies of the instantiated functions can be generated, this qualifier might not be not suitable for production use.

The /TEMPLATE_DEFINE=LOCAL qualifier cannot be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it is disabled by the /

TEMPLATE_DEFINE=LOCAL qualifier. Explicit use of /TEMPLATE_DEFINE=LOCAL and /TEMPLATE_DEFINE=AUTO is an error.

/TEMPLATE_DEFINE=USED

Instantiate those template entities that were used in the compilation. This includes all static data members for which there are template definitions. Overrides /TEMPLATE_DEFINE=AUTO at compile time.

/TEMPLATE_DEFINE=USED_REPOSITORY

Like /TEMPLATE_DEFINE=ALL_REPOSITORY, but instantiates only templates used by the compilation. The explicit instantiations are also put into the repository as separate object files.

Table 5.1 summarizes each template instantiation model's What, Where, and How (as described in Section 5.1) for both implicit and manual instantiation.

Table 5.1. Template Instantiation Models

Model (/TEMPLATE=)	Why Implicit			Why Manual		
	What	**Where**	**How**	**What**	**Where**	**How**
AUTO	part	repository for Alpha, object for I64	global for Alpha, COMDAT for I64	part	object	global for Alpha, COMDAT for I64
NOAUTO	N/A	N/A	N/A	part	object	global for Alpha, COMDAT for I64
IMPLICIT_LOCAL	part	object	local for Alpha, COMDAT for I64	part	object	global for Alpha, COMDAT for I64
LOCAL	part	object	local for Alpha, COMDAT for I64	part	object	local for Alpha, COMDAT for I64
USED	part	object	global for Alpha, COMDAT for I64	part	object	global for Alpha, COMDAT for I64
USED_REPO	part	repository	global for Alpha, COMDAT for I64	part	repository	global for Alpha, COMDAT for I64
ALL	all	object	global for Alpha, COMDAT for I64	all	object	global for Alpha, COMDAT for I64
ALL_REPO	all	repository for Alpha, object for I64	global for Alpha,	all	repository for Alpha,	global for Alpha,

Model (/TEMPLATE=)	Why Implicit			Why Manual		
	What	Where	How	What	Where	How
			COMDAT for I64		object for I64	COMDAT for I64

5.6.2. Other Instantiation Qualifiers

The following qualifiers are independent of the model used and each other:

/TEMPLATE_DEFINE=DEFINITION_FILE_TYPE="file-type-list"

Specifies a string that contains a list of file types that are valid for template definition files. Items in the list must be separated by commas and preceded by a period. A type is not allowed to exceed the OpenVMS limit of 31 characters. This qualifier is applicable only when automatic instantiation has been specified. The default is /TEMPLATE_DEFINE=DEF=".CXX,.C,.CC,.CPP".

/TEMPLATE_DEFINE=PRAGMA

Determines whether the compiler ignores `#pragma define_template` directives encountered during the compilation. This qualifier lets you quickly switch to automatic instantiation without having to remove all the pragma directives from your program's code base.The default is /TEMPLATE_DEFINE=PRAGMA, which enables `#pragma define_template`.

/TEMPLATE_DEFINE=VERBOSE

Turns on verbose or verify mode to display each phase of instantiation as it occurs. During the compilation phase, informational level diagnostics are generated to indicate which templates are automatically being instantiated. This qualifier is useful as a debugging aid.

/PENDING_INSTANTIATIONS[=n]

Limit the depth of recursive instantiations so that infinite instantiation loops can be detected before some resource is exhausted. The /PENDING_INSTANTIATIONS qualifier requires a positive non-zero value `n` as argument and issues an error when `n` instantiations are pending for the same class template. The default value for`n` is 64.

5.6.3. Repository Qualifiers

The following qualifiers are only applicable if a repository is being used *(Alpha only)*:

/TEMPLATE_DEFINE=TIMESTAMP

Causes the compiler to create a timestamp file named `TIMESTAMP.` in the repository. Thereafter, instantiations are added or regenerated only if needed; that is, if they do not alreay exist, or if existing ones are older than the timestamp.

/REPOSITORY

Specifies a repository that C++ uses to store requested template instantiations. The default is /REPOSITORY=[.CXX_REPOSITORY]. If multiple repositories are specified, only the first is considered writable, and the default repository is ignored unless specified.

Chapter 6. Handling C++ Exceptions

Exception handling is a C++ language mechanism for handling unusual program events (not just errors). On OpenVMS systems, VSI C++ implements the exception handling model described in the C++ International Standard.

This includes support for throwing and catching exceptions, and calling the `terminate()` and `unexpected()` functions. C++ exception-handling support is implemented using functions and related OpenVMS system services that comprise the OpenVMS condition-handling facility. Hence, C++ exception-handling support is fully integrated with existing uses of the OpenVMS condition handling facility.

6.1. Compiling with Exceptions

Because exceptions are enabled by default, you need not specify the /EXCEPTIONS qualifier whenever you compile the program.

For more information about the /EXCEPTIONS qualifier see Appendix A.

Note

If you are programming in kernel mode or creating a protected shareable image, C++ exception handling is not supported. To ensure that your code does not contain constructs that trigger C++ exceptions or to prevent errors from occurring during initialization of exception handlers at runtime, specify the /NOEXCEPTIONS qualifier when compiling.

6.2. Linking with Exceptions *(Alpha only)*

If any files in your program contain throw expressions, try blocks, or catch statements, or if any files in your program are compiled with the exceptions, you must link your program using the `cxxlink` facility (see Section 1.3 for more information on this facility). For example:

```
$ cxxlink my_prog.obj
```

Using the `cxxlink` facility ensures that the run-time support for exceptions (`sys $library:libcxxstd.olb`) is linked into your program.

Linking with /NOSYSSHR (OpenVMS Version 6.2)

If you are running OpenVMS Version 6.2 or later, and you want to link using the /NOSYSSHR qualifier, you must specify a linker options file on your **`cxxlink`** command. Otherwise, your link might fail because of undefined symbol references.

The linker options file should contain the following:

```
sys$share:librtl.exe/shar
```

For example, if `cxx_exc.opt` is your linker options file containing the above line, then a possible link command would be:

```
$ cxxlink my_prog.obj, my_disk:[my_dir]cxx_exc.opt/opt
```

Because the necessary run-time libraries are not provided in object format on OpenVMS Version 6.1 and earlier releases, linking with /NOSYSSHR on those systems is not recommended.

For more information about linking with /NOSYSSHR and about OpenVMS linker options files see the *VSI OpenVMS Linker Utility Manual*.

6.3. The terminate() and unexpected() Functions

The `unexpected()` and `set_unexpected()` functions are implemented as defined in the ISO/IEC International Standard.

The `terminate()` and `set_terminate()` functions are implemented as defined in the ISO/IEC International Standard. By default, the `terminate()` function raises the OpenVMS condition `cxxl $_terminate`, and then calls the `abort()` function.

On Alpha systems, no stack unwinding is done by the `terminate()` function. Hence, no destructors are called for constructed objects when a thrown exception results in a call of the `terminate()` function. Instead, the program is terminated.

On I64 systems, stack unwinding *is* done.

If a C++ function is called from a program in which the main function is not C++, `terminate()` is not called. Instead, the call stack points to the point of the throw.

6.4. C++ Exceptions and Other Conditions

Because C++ exceptions are implemented using the OpenVMS condition handling facility, C++ modules will work properly when they are part of a program that makes other uses of OpenVMS condition handling.

The raising and handling of an OpenVMS condition can result in the destruction of C++ automatic objects. If the handling of an OpenVMS condition results in an unwind through a C++ function's stack frame, then destructors will be called for automatic objects declared in that stack frame, just as if a C++ exception had been caught by a handler in an outer stack frame.

The C++ exception handling facility can also be used to catch OpenVMS conditions that are raised independently of C++ throw expressions. Except for those OpenVMS conditions that result in the delivery of signals, a C++ `catch(...)` handler will catch both C++ thrown exceptions and OpenVMS conditions. (For more information about OpenVMS conditions that result in the delivery of signals, see Section 6.5.)

You can use the data type `struct chf$signal_array &`, defined in the system header file `chfdef.h`, to catch OpenVMS conditions and to obtain information about the raised conditions. The C++ exceptions support transfers control to `catch(struct chf$signal_array &)` handlers when it determines that an OpenVMS condition was raised independently of a C++ throw statement.

If the `catch (struct chf$signal_array &)` handler specifies a class object, then the C++ exceptions support sets the class object to be a reference to the raised OpenVMS condition's signal argument vector. In the following example, `obj.chf$l_sig_name` will have the value 1022 when it is printed:

```
#include <chfdef.h>
#include <iostream.hxx>
#include <lib$routines.h>
main ()
{
    try {
        lib$signal (1022);
    } catch (struct chf$signal_array &obj) {
        cout << obj.chf$l_sig_name << endl;
    }
}
```

A `catch(struct chf$signal_array &)` handler will also catch a thrown object that is explicitly declared to be of type `struct chf$signal_array &`. In this case, the value of the catch handler's object is determined by the originally thrown object, not the OpenVMS signal argument vector.

You can also use the data type `struct chf$signal_array *` to catch both OpenVMS conditions and objects explicitly declared to be of type `struct chf$signal_array *`. If a `catch(struct chf$signal_array *)` handler specifies an object, then that object becomes a pointer to the thrown object.

For more information about OpenVMS conditions, see the OpenVMS Calling Standard.

6.5. C++ Exceptions and Signals *(Alpha only)*

Certain OpenVMS conditions (as described in the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]) normally result in the delivery of signals. These signals can be processed using the signal handler mechanism described in the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/].

You can call the following run-time function in conjunction with the /EXCEPTION=IMPLICIT qualifier to cause these OpenVMS conditions to be treated as exceptions, instead of signals:

```
cxxl$set_condition(condition_behavior signal_or_exc)
```

This can be done by putting the following call in your program:

```
#include <cxx_exception.h>
…
cxxl$set_condition (cxx_exception);
```

Caution

You must specify /EXCEPTION=IMPLICIT; otherwise, the code that would normally cause a signal and now causes an exception might be moved out of the try block.

After your program calls the `cxxl$set_condition (cxx_exception)` function you can then catch these exceptions using any of the following handlers:

```
catch(struct chf$signal_array &)
catch(struct chf$signal_array *)
catch(...)
```

To revert back to the default signal behavior, you can make the following call:

```
cxxl$set_condition (unix_signal);
```

Caution

Avoid doing a C++ throw from a C signal handler or VMS exception handler because this action could terminate your program.

The following are defined in the header file `cxx_exception.h`:

The `cxxl$set_condition()` function
The `condition_behavior {unix_signal=0, cxx_exception=1 }` enumeration type

The `cxxl$set_condition` function returns the previous setting. This function affects all threads in a process.

6.6. C++ Exceptions with setjmp and longjmp

If a C++ function calls either the `setjmp()` or the `longjmp()` routine, C++ exceptions support is disabled for that function. This means the following:

- No exceptions can be caught by the function's catch handlers.
- No destructors are called for the function's automatic data if an exception propagates through the function.
- The `unexpected()` function is not called for that function.
- If either `setjmp()` or `longjmp()` is called from `main()`, then `terminate()` is not called for an unhandled exception.

6.7. C++ Exceptions, lib$establish and vaxc$establish

If a C++ function calls either the `lib$establish()` or the `vaxc$establish()` routine, then C ++ exceptions support is disabled for that function. This means the following:

- No exceptions can be caught by the function's catch handlers.
- No destructors are called for the function's automatic data if an exception propagates through the function.
- The `unexpected()` function is not called for that function.
- If either `lib$establish()` or `vaxc$establish()` is called from `main()`, then `terminate()` is not called for an unhandled exception.

6.8. Performance Considerations

The compiler optimizes the implementation of exception handling for normal execution, as follows:

- Applications that do not use C++ exceptions and are compiled with the /NOEXCEPTIONS qualifier incur no run-time or image size overhead.
- Applications compiled with exceptions enabled that have try blocks or automatic objects with destructors incur an increase in image size.
- As much as possible, the run-time overhead for exceptions is incurred when throwing and catching exceptions, not when entering and exiting try blocks normally.

6.9. C++ Exceptions and Threads

C++ exceptions are thread safe. This means that multiple threads within a process can throw and catch exceptions concurrently. However, exceptions do not propagate from one thread to another, nor can one thread catch an exception thrown by another thread.

The `set_terminate()` and `set_unexpected()` functions set the `terminate()` and `unexpected()` handlers for the calling thread. Therefore, each thread in a program has its own `terminate()` and `unexpected()` handlers.

If you want every thread in your program to use the same nondefault `terminate()` or `unexpected()` handlers, then you must call the `set_terminate()` and `set_unexpected()` functions separately from each thread.

By default, the C++ exception package allows the delivery of the CMA$_EXIT_THREAD condition, but not the CMA$_ALERTED condition. This latter condition is raised to a thread that is being cancelled. The following routine (`test_thread`) allows an application to control the default behavior of these two conditions:

```
int cxxl$catchable_condition (int condition, int on_or_off);
```

The condition is either CMA$_EXIT_THREAD or CMA$_ALERTED. A value of zero means the program does not want the condition to result in a catch clause receiving the exception. This is the default behavior for CMA$_ALERTED. A value of nonzero means the program does want the condition to result in the catch clause of the thread to receive control when the exception is raised. The behavior is undefined for any other condition value passed.

The return value of the routine is the previous setting for the passed condition value:

```
#include <pthread.h>
#include <cxx_exception.h>
#include <cma$def.h>
...
static void *test_thread (...) {
  ...
  try {
    ...
  } catch (chf$signal_array *p) {
    switch (p->chf$l_sig_name) {
      case CMA$_ALERTED:
        printf ("  test_thread caught CMA$_ALERTED\n");
        break;
      default:
        printf ("  test_thread caught (%d)\n", p->chf$l_sig_name);
        break;
    }
  }
```

```
}
int main () {
  ...
  if (cxxl$catchable_condition(CMA$_ALERTED,1))
       printf ("  CMA$_ALERTED continues to be catchable\n");
  else printf ("  CMA$_ALERTED is now catchable\n");
  ...
  pthread_create (&thread, ...);
  ...
  pthread_cancel (thread);
  ...
}
```

For more information about threads, see the *Guide to POSIX Threads Library* manual.

6.10. Debugging with C++ Exceptions *(Alpha only)*

You can use the OpenVMS Debugger **`set break/exception`** command to set a breakpoint when an exception is thrown. You can use the **`show calls`** command to determine the location where the exception was thrown.

Chapter 7. The C++ Standard Library

The C++ Standard Library provided with this release defines a complete specification of the C++ International Standard, with some differences, as described in the online release notes in:`SYS$HELP:CXX_RELEASE_NOTES.PS`

Note that portions of the C++ Standard Library have been implemented in VSI C++ using source licensed from and copyrighted by Rogue Wave Software, Inc. Information pertaining to the C++ Standard Library has been edited and incorporated into VSI C++ documentation with permission of Rogue Wave Software, Inc. All rights reserved.

Some of the components in the C++ Standard Library are designed to replace nonstandard components that are currently distributed in the Class Library. VSI will continue to provide the Class Library in its nonstandard form. However, you now have the option of using new standard components.

This chapter provides more information on the VSI C++ implementation of the Standard Library, including upward compatibility, compiling, linking, and thread safety. Small example programs showing how to use the C++ standard library are located in the directory `SYS$COMMON:[SYSHLP.EXAMPLES.CXX]`.

The following are Standard Library qualifiers introduced with VSI C++:

/[NO]USING_STD

Controls whether standard library header files are processed as though the compiled code were written as follows:

```
using namespace std;
#include <header>
```

These qualifiers are provided for compatibility for users who do not want to qualify use of each standard library name with std:: or put `using namespace std;` at the top of their sources.

/USING_STD turns implicit using namespace std on; this is the default when compiling /STANDARD=ARM, /STANDARD=MS, or /STANDARD=RELAXED.

/NOUSING_STD turns implicit using namespace std off; this is the default when compiling /STANDARD=STRICT_ANSI.

/ASSUME=[NO]STDNEW

Controls whether calls are generated to the ANSI or pre-ANSI implementation of the `operator new()`. On memory allocation failure, the ANSI implementation throws `std::bad_alloc`, while the pre-ANSI implementation returns 0.

/ASSUME=STDNEW generates calls to the ANSI new() implementation; this is the default when compiling with /STANDARD=RELAXED and /STANDARD=STRICT_ANSI.

/ASSUME=NOSTDNEW generates calls to the pre-ANSI new() implementation; this is the default when compiling with /STANDARD=ARM and /STANDARD=MS.

/ASSUME=[NO]GLOBAL_ ARRAY_NEW

Controls whether calls to global array new and delete are generated as specified by ANSI. Pre-ANSI global array new generated calls to `operator new()`. According to ANSI, use of global array new generate calls to `operator new()[]`.

/ASSUME=GLOBAL_ARRAY_NEW generates calls to `operator new()[]` for global array new expressions such as `new int[4]`; this is the default when compiling /STANDARD=RELAXED, /STANDARD=STRICT_ANSI, and /STANDARD=MS.

/ASSUME=NOGLOBAL_ARRAY_NEW generates calls to `operator new()` for global array new expressions such as `new int[4]` and preserves compatibility with Version 5.n; this is the default when compiling /STANDARD=ARM).

7.1. Important Compatibility Information

On Alpha systems, because the standardization process for the C++ Standard Library is not yet completed, VSI cannot guarantee that this version of the library is compatible with any past or future releases. We ship the run-time portion of the library in object form, not in shareable form, to emphasize this situation. *(Alpha only)*

On I64 systems, the standard library is distributed as a system shareable image SYS$LIBRARY:CXXL$RWRTL.EXE, and also in object form in the system object library STARLET.OLB. *(I64 only)*

The following sections describe specific compatibility issues.

7.1.1. /[NO]USING_STD Compiler Compatibility Qualifier

All standard library names in VSI C++ are inside the namespace `std`. Typically you would qualify each standard library name with `std::` or put `using namespace std;` at the top of your source file.

To make things easier for existing users, `using namespace std;` is included in a file provided with every standard library header when you are in ARM, MS, or RELAXED compiler modes. This is not the default in STRICT_ANSI mode.

The compiler supplied qualifiers /NOUSING_STD and /USING_STD can be used to override the default. /NOUSING_STD turns the implicit using namespace `std` off; /USING_STD turns it on.

7.1.2. Pre-ANSI/ANSI Iostreams Compatibility

The C++ Standard Library offers support for the standard iostream library based on the C++ International Standard. As defined by the standard, iostream classes are in the new header files `<iostream>`, `<ostream>`, `<istream>`, and so on (no `.h` or `.hxx` extension).

For backward compatibility, the pre-ANSI iostream library is still provided. The two libraries exhibit subtle differences and incompatibilities.

Users can choose which version (ANSI or pre-ANSI) of iostreams they want to use; either version of iostreams can be integrated seamlessly with the new Standard Library and string functionality.

To accomplish this goal, macros called `__USE_STD_IOSTREAM` and `__NO_USE_STD_IOSTREAM` are provided. If you do not set these macros explicitly, the default in ARM, MS, and RELAXED modes is to use the pre-ANSI iostream library. In STRICT_ANSI mode, the default is to use the ANSI iostream library.

Note that for the most part, support for pre-ANSI iostreams is contained in headers with `.h` or `.hxx` extensions. This is not the case for `iostream.h/iostream.hxx`, `fstream.h/fstream.hxx`, and `iomanip.h/iomanip.hxx`. In these cases, the iostream library provided

is controlled solely by the /STANDARD compilation choice and use of `__USE_STD_IOSTREAM/__NO_USE_STD_IOSTREAM`.

You override the default by defining `__USE_STD_IOSTREAM` or `__NO_USE_STD_IOSTREAM` on either the command line or in your source code.

In ARM, MS, and RELAXED modes, specify use of the ANSI iostreams in one of the following ways:

- Enter /DEFINE=(__USE_STD_IOSTREAM) on the command line.
- Put the following in your source file before any include files:

```
#ifndef __USE_STD_IOSTREAM
#define __USE_STD_IOSTREAM
#endif
```

In STRICT_ANSI mode, specify use of the pre-ANSI iostreams in one of the following ways:

- Enter /DEFINE=(__NO__USE_STD_IOSTREAM) on the command line.
- Put the following in your source file before any include files:

```
#ifndef __NO_USE_STD_IOSTREAM
#define __NO_USE_STD_IOSTREAM
#endif
```

You receive a `#error` warning if

- you compile in a mode indicating you want ANSI behavior; that is, /STANDARD=STRICT_ANSI,
- you enter a header with a `.h` or `.hxx` (for example, #include `<iostream.h>`).

You can avoid the error by compiling with /DEFINE=(__NO_USE_STD_IOSTREAM).

Many of the other headers, `<string>` for example, make use of the iostream classes. The default version of iostreams that is automatically included when you include one of these headers depends on the mode you compile in and the setting of the macros `__USE_STD_IOSTREAM` and `__NO_USE_STD_IOSTREAM` as described earlier.

Because the standard locale class and the standard iostream class are so closely tied, you cannot use the standard locale class with the pre-ANSI iostream classes. If you want to use locale, you must use the ANSI iostream classes.

It is possible to use the pre-ANSI and the ANSI iostream library in the same source file, because all the standard iostream names (that is, `cout`, `cin`, and so on) are in namespace `std`, and all the pre-ANSI names are in the global namespace. This is not recommended, though, because there is no guarantee of stream objects being the same size or, for example, of `::cout` being in sync with `std::cout`.

Nevertheless, if you want to combine them, you must recognize that the underlying ANSI iostream is called `iostream_stdimpl.hxx` and that the pre_ANSI header is called `iostream_impl.hxx`. The following example shows how to include a pre-ANSI header before and ANSI header:

```
#include <stdlib.h>
#undef __USE_STD_IOSTREAM
#include <iostream_impl.hxx>
#define __USE_STD_IOSTREAM
#include <iostream_stdimpl.hxx>

int main()
{
  std::string s("abc");
  ::cout << "abc" << endl;         // pre-standard iostreams
  std::cout << "abc" << std::endl; // standard iostreams
  return EXIT_SUCCESS;
}
```

If you include an ANSI iostreams header before a pre-ANSI iostreams header, follow these steps:

1. Compile your source using /NOUSING_STD.

2. Use the `__USE_STD_IOSTREAM` macro as shown in the following example. You must define `__USE_STD_IOSTREAM` at the end of your include file list so that the template definition files (the `.cc` files) are included in the correct mode.

```
// Compile this with /nousing_std
#include <stdlib.h>
#define __USE_STD_IOSTREAM
#include <iostream_stdimpl.hxx>
#undef __USE_STD_IOSTREAM
#include <iostream_impl.hxx>
#define __USE_STD_IOSTREAM // so the template definition files are ok

int main()
{
  std::string s("abc");
  ::cout << "abc" << endl; // pre-standard iostreams
  std::cout << "abc" << std::endl; // standard iostreams
  return EXIT_SUCCESS;
}
```

7.1.3. Support for pre-ANSI and ANSI operator new()

The Standard C++ Library supports the ANSI implementation of the `operator new()` as well as the pre-ANSI implementation of `operator new()`. The ANSI implementation throws `std::bad_alloc` on memory allocation failures.

The pre-ANSI implementation of the `operator new()` returns 0 on memory allocation failures. Because the ANSI behavior is incompatible with pre-ANSI applications, a compile time qualifier has been added (/ASSUME=[NO]STDNEW) to control whether calls to ANSI `new()` or pre-ANSI `new` are generated.

The following examples show how ANSI versus pre-ANSI `new()` check for memory allocation. First, here is an ANSI `new()` check for memory allocation failure:

```
try {
    myobjptr = new (myobjptr);
}
catch (std::bad_alloc e) {
    cout << e.what() << endl;
```

```
};
```

The following example shows a pre-ANSI `new()` check for memory allocation failure:

```
if ((myobjptr = new (myobjptr)) == 0)
    call_failure_routine();
```

When upgrading pre-ANSI `new()` code to work with the C++ Standard Library you also can use the nothrow version of ANSI `new()`. To do so in the pre-ANSI example, you could recode it as follows:

```
if ((myobjptr = new (myobjptr, nothrow)) == 0)
     call_failure_routine();
```

Two command line qualifiers are available in the compiler to control whether calls are generated to the ANSI or pre-ANSI implementation of `operator new()`:

- Use the /ASSUME=STDNEW qualifier to generate calls to the ANSI `new()` implementation.
- Use the /ASSUME=NOSTDNEW qualifier to generate calls to the pre-ANSI `new()` implementation. You can override global `new()` by declaring your own functions.

When compiling with /STANDARD=RELAXED or /STANDARD=STRICT_ANSI, /ASSUME=STDNEW is the default.

When compiling with /STANDARD=ARM and /STANDARD=MS, /ASSUME=NOSTDNEW is the default. The compiler defines the macro `__STDNEW` when the /ASSUME=STDNEW qualifier is specified.

7.1.4. Overriding operator new() *(Alpha only)*

On Alpha systems, the ability to override the library version of global `operator new()` and global `operator delete()` is available with OpenVMS Version 6.2 and later. If you want to define your own version of `global operator new()` on OpenVMS systems, you must define your own version of global `operator delete()` and vice versa. To define a global `operator new()` or a global `operator delete()` to replace the version used by the C++ Standard Library or the C++ Class Library, or both, follow these steps:

1. Define a module to contain two entry points for your version of `global operator new()`. You must code the module so that it is always compiled either with the /ASSUME=STDNEW or with the /ASSUME=NOSTDNEW qualifier.
2. If you code your module to compile with /ASSUME=STDNEW, follow instructions in the next subsection. If you code your module to compile with /ASSUME=NOSTDNEW, follow instructions in the section called “Compiling with /ASSUME=NOSTDNEW”.

Compiling with /ASSUME=STDNEW

1. Verify that your module contains two entry points for your version of global operator new(). One entry point, which has the `name __nw__XUi` (/MODEL=ARM) or `__7__nw__FUi` (/MODEL=ANSI), is used to override global operator new() in the Class Library. The other entry point, which has the name `new`, is used to override global `operator new()` in the Standard Library.
2. Define global `operator new()` in terms of the entry point `new`. Code `__nw__XUi` (for /MODEL=ARM) or `__7__nw__FUi` (for /MODEL=ANSI) to call operator `new`. Your module appears as follows:

```
#include <new>
...
using namespace std;

// Redefine global operator new(),
// entry point into C++ Standard Library based on
// compiling /assume=stdnew.  This also overrides user
// calls to operator new().
void *operator new(size_t size) throw(std::bad_alloc) {
  printf("in my global new\n");
  ...
  void *p = malloc(size);
  return(p);
}

// redefine global operator delete()
void operator delete(void  *ptr) throw() {
  free (ptr);
}

// entry point into C++ Class Library
#ifdef __MODEL_ANSI
extern "C" void *__7__nw__FUi(size_t size) {
#else  // __MODEL_ARM
extern "C" void *__nw__XUi(size_t size) {
#endif
  printf("in my new\n");
  return ::operator new(size);
}
```

Compiling with /ASSUME=NOSTDNEW

1. Verify that your module contains two entry points for your version of global operator `new()`. One entry point, which has the name `cxxl$__stdnw__XUi` (for /MODEL=ARM) or `cxxl$__7__stdnw__FUi` (for /MODEL=ANSI), is used to override global `operator new()` in the Standard Library. The other entry point, which has the name `new`, is used to override global `operator new()` in the Class Library.

2. Define global `operator new()` in terms of the entry point new. Code `cxxl$__stdnw__XUi` (/MODEL=ARM) or `cxxl$__stdnw__FUi` (/MODEL=ANSI) to call operator `new`. Your module appears as follows:

```
#include <new>
...
using namespace std;

// Redefine global operator new(),
// entry point into C++ Class Library based on
// compiling /assume=nostdnew
void *operator new(size_t size) {
  printf("in my global new\n");
  ...
  void *p = malloc(size);
  return(p);
}

// redefine global operator delete()
```

```
void operator delete(void  *ptr) {
 free (ptr);
}

// entry point into C++ Standard Library
#ifdef __MODEL_ANSI
extern "C" void *cxxl$__7__stdnw__FUi(size_t size) {
#else     // __MODEL_ARM
extern "C" void *cxxl$__stdnw__XUi(size_t size) {
  return ::operator new(size);
}
```

3. Link your program using the /NOSYSSHR qualifier of the LINK command. You must also link with a linker options file that includes the shareable images your program requires. (This is because some components of OpenVMS systems ship only in shareable form.) The file contains at least the shareable images shown below. The options file cannot contain, `DECC$SHR`, because it contains the definitions of `new()` and `delete()` that you are attempting to override. So your LINK command will be similar to the following:

   ```
   $ CXXLINK TEST,SYS$LIBRARY:STARLET.OLB/INCLUDE=CXXL_INIT, -
   SYS$DISK:[]AVMS_NOSYSSHR.OPT/OPT/NOSYSSHR
   ```

   where `AVMS_NOSYSSHR.OPT` is:

   ```
   SYS$SHARE:CMA$TIS_SHR/SHARE
   SYS$SHARE:LIBRTL/SHARE
   ```

   Linking /NOSYSSHR is the only way to override calls to `new()` and `delete()` in the C++ Class Library and C++ Standard Library.

   If the `set_new_handler()` function is referenced when overriding `operator new()` and `delete()`, “multiply defined” linker warnings will result. To remove these warnings, the `set_new_handler()` function must also be overridden. Using `set_new_handler()` when the `operator new()` and `delete() functions` are being overridden, requires that the `set_new_handler()` function be defined in terms of the user provided `operator new()` and `delete()` functions.

7.1.5. Overriding operator new() *(I64 only)*

Overriding operators `new` and `delete` has been simplified on I64 systems. If user code overrides any of the `new` and `delete` operators, the compiler and library picks up the overridden versions without any other changes to the source code or the command line. Changes such as those described for Alpha systems in Section 7.1.4 are unnecessary and will not work on I64 systems.

7.1.6. Support for Global array new and delete Operators

VSI C++ Version 6.n and higher fully supports the `array new` and `delete` operators as described in the ANSI standard. Previous versions did not.

You might therefore encounter a compatibility problem if you have overridden the run-time library's `operator new()` with your own version.

For example:

```
#include <stdlib.h>
#include <iostream.h>

inline void* operator new(size_t s) {
    cout << "called my operator new" << endl;
    return 0;
}

int main() {
    new int;    // ok, this still calls your own
    new int[4]; // In V6.0 calls the C++ library's operator new[]
    return EXIT_SUCCESS;
}
```

In older versions, both `new int` and `new int[4]` would generate a call to operator `new()` (they would just be asking for different sizes). With the current compiler, `new int` still generates a call to `operator new()`. However, `new int[4]` generates a call to `operator new()[]`. This means that if you still want to override the library's `operator new` you must do one of the following:

1. Provide your own definition of `operator new()[]`.

2. Use the /ASSUME=NOGLOBAL_ARRAY_NEW qualifier.

The /ASSUME=NOGLOBAL_ARRAY_NEW qualifier converts all expressions such as `new int[4]` to calls to the global `operator new()`, thus preserving compatibility with older compiler versions.

Note that this qualifier has *no* effect on class-specific array `operator new` and `delete`; it affects only the global operators.

When compiling with /STANDARD=RELAXED or /STANDARD=STRICT_ANSI, and /STANDARD=MS modes, /ASSUME=GLOBAL_ARRAY_NEW is the default.

When compiling with /STANDARD=ARM, /ASSUME=NOGLOBAL_ARRAY_NEW is the default. A macro `__GLOBAL_ARRAY_NEW` is predefined by the compiler when /ASSUME=GLOBAL_ARRAY_NEW is used.

7.1.7. IOStreams Expects Default Floating-Point Format

The C++ standard library IOStreams expects floating-point values in the default floating-point format for each platform: G_FLOAT on Alpha systems and IEEE on I64 systems. Using standard library IOStreams for processing floating-point values in a different format (for example, in a program compiled /FLOAT=IEEE on Alpha or /FLOAT=G_FLOAT on I64) is not supported. The C++ class library does not have this restriction.

7.2. How to Build Programs Using the C++ Standard Library

Building programs that use the C++ Standard Library requires the following changes in usage of the C++ compiler and linker commands:

- The CXX command line no longer needs to include the /ASSUME=NOHEADER_TYPE_DEFAULT qualifier because this is now the default.

Similarly, the command line no longer needs to include the /TEMPLATE=AUTO qualifier because the compiler performs automatic template instantiation by default.

- On Alpha systems, to link a program that uses the C++ Standard Library, you must use the CXXLINK command in place of the LINK command. The CXXLINK command continues the automatic template instantiation process, includes the Standard Library run-time support (SYS$LIBRARY:LIBCXXSTD.OLB) at link time, and creates the final image. See Section 1.3 for more details. *(Alpha only)*

  On I64 systems, to link a program that uses the C++ Standard Library, you must use either the CXXLINK facility or OpenVMS Linker. See Section 1.4 for more details. *(I64 only)*

  For example, to build a program called `prog.cxx` that uses the Standard Library, you can use the following commands:

```
$ CXX  prog.cxx
$ CXXLINK prog.obj
```

Thread Safety

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

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

7.3. Optional Switch to Control Buffering *(Alpha only)*

The `inplace_merge`, `stable_sort`, and `stable_partition` algorithms require the use of a temporary buffer. Two methods are available for allocating this buffer:

- Preallocate 16K bytes of space on the stack.
- Allocate the required amount of storage dynamically.

By default, the current VSI C++ Standard Library makes use of the preallocated buffer, which avoids the overhead of run-time allocation. If your application requires a buffer that exceeds 16K, it cannot take advantage of this default.

If you are concerned with minimizing the use of stack space in your program, or if your application requires a buffer that exceeds 16K, define the `__DEC_DYN_ALLOC` macro to enable dynamic buffering. Do this by adding the following to your compile command line:

```
/DEFINE=__DEC_DYN_ALLOC
```

7.4. Enhanced Compile-time Performance of ANSI Iostreams

To speed the compile-time performance of programs that use the standard `iostream` and `locale` components, the Standard Library includes many common template instantiations of these components.

To force programs to create instantiations at compile-time (for example, if you want to debug them and thus need them to be compiled with the /DEBUG qualifier), define the macro `__FORCE_INSTANTIATIONS` *(Alpha only)* on the command line by specifying /DEFINE=(__FORCE_INSTANTIATIONS). This definition suppresses the `#pragma do_not_instantiate` directives in the headers so that the compiler creates the instantiations in your repository directory.

You must then specify the /REPOSITORY= qualifier to force the compiler to link your instantiations instead of those in the Standard Library.

7.5. Using RMS Attributes with iostreams

The standard library class `fstream` constructors amd `open()` member function do not support different RMS attributes, for example, creating a stream-lf file.

To work around this restriction, use the C library `creat()` or `open()` call, which returns a file descriptor, and then use the `fstream` constructor, which accepts a file descriptor as its argument. For example:

```
#define __USE_STD_IOSTREAM
#include <fstream>

int main()
{
  int fp;

  // use either creat or open
  //if ( !(fp = creat("output_file.test", 0, "rfm=stmlf")) )

  if ( !(fp = open("output_file.test", O_WRONLY | O_CREAT | O_TRUNC , 0,
 "rfm=stmlf")) )
    perror("open");

  ofstream output_file(fp); // use special constructor, which takes
                            // a file descriptor as argument
  // ...
}
```

Note that this coding is not allowed if you compile with /STANDARD=STRICT_ANSI, because the constructor in the example is an extension to the C++ standard interface.

7.6. Upgrading from the Class Library to the Standard Library

The following discussion guides you through upgrading the Class Library code to use the Standard Library, specifically replacing the vector and stack classes in the `vector.hxx` header file to the Standard Library vector and stack classes.

7.6.1. Upgrading from the Class Library Vector to the Standard Library Vector

To change your code from using the Class Library vector to the Standard Library vector, consider the following actions:

- Change the name of your `#include` statement from `<vector.h>` or `<vector.hxx>` to `<vector>`.
- Remove the `vectordeclare` and `vectorimplement` declarations from your code.
- Change all `vector(`*`type`*`)` declarations to `vector<`*`type`*`>`. For example, `vector(int) vi` should become `vector<int> vi`.
- The following member functions are replaced in the Standard Library:

Nonstandard Vector Function	Standard Library Vector Function
`elem(int index)`	`operator[](size_t index)` (no bounds checking)
`operator[](int index)`	`at(size_t index)` (bounds checking)
`setsize(int newsize)`	`resize(size_t newsize)`

- When copying vectors of unequal lengths, note that the Standard Library vector has a different behavior as follows:

  When using the Standard Library vector, if the target vector is smaller than the source vector, the target vector automatically increases to accommodate the additional elements.

  The Class Library vector displays an error and aborts when this situation occurs.

- Note that another difference in behavior occurs when you specify a negative index for a vector.

  The Class Library vector class detects the negative specification and issues an error message. However, the Standard Library vector silently converts the negative value to a large positive value, because indices are represented as type `size_t (unsigned int)` rather than `int`.

- When an out-of-bounds error occurs, the Class Library vector prints an error message and aborts, whereas the Standard Library vector throws an out-of-range object.

7.6.2. Upgrading from the Class Library Stack to the Standard Library Stack

To change your code from using the existing stack to the Standard Library stack, consider the following actions:

- Change the name of your `#include` statement from `<stack.h>` or `<stack.hxx>` to `<stack>`.
- Remove the `stackdeclare` and `stackimplement` declarations from your code.

- Change all `stack(type)` declarations to `stack<type, deque<type> >`. For example, `stack(int) si` should become `stack<int, deque<int> > si`.
- Do not specify an initial size for a Standard Library stack. The stack must start out empty and grow dynamically (as you push and pop).
- The following member functions are not supported or have different semantics:

Class Library Stack	Standard Library Stack
`size_used()`	Does not exist because the `size()` function always is equal to the `size_used()` function.
`full()`	Does not exist because the stack always is full.

`pop()`	Does not return the popped element. To simulate Class Library behavior, first obtain the element as the return type from the `top()` function and then call the `pop()` function. For example, change `int i=s.pop();` to the following:
	
	`int i=s.top();`
	`s.pop();`

- The Standard Library stack differs from the Class Library stack in the way errors are detected. Unlike the nonstandard stack, you cannot overflow a Standard Library stack because space is allocated dynamically as you push elements onto the stack.

7.6.3. Upgrading from the Class Library String Package Code

The Standard Library `basic_string` can replace the Class Library String Package.

The following list guides you through upgrading nonstandard code to use the Standard Library `basic_string`:

- Change `#include <string.h>` or `#include <string.hxx>` to `#include <string>`.
- Change all declarations of `String` to `string` (uppercase S to lowercase s).
- On Alpha systems, when compiling with the __DEC_STRING_COMPATIBILITY macro defined, the String Package allowed assignment of a string directly to a `char *`; however, the `basic_string` library does not allow this. You can assign the string's `const char*` representation using the `c_str()` or `data()` `basic_string` member functions. For example:

  ```
  string s("abc");
  char* cp = s; // not allowed
  const char* cp = s.data(); // ok
  ```

  The state of the string is undefined if the result of `data()` is cast to a non-`const char*` and then the value of that `char*` is changed.

- The String Package member functions `upper()` and `lower()` are not in the `basic_string` library. You can write these functions as nonmember functions, as follows:

```
template <class charT, class traits, class Allocator>
inline
basic_string<charT, traits, Allocator>
upper(const basic_string<charT,traits, Allocator>& str) {
        basic_string<charT, traits, Allocator> newstr(str);
        for (size_t index = 0; index < str.length(); index++)
                if (islower(str[index]))
                        newstr[index] = toupper(str[index]);
        return newstr;
}

template <class charT, class traits, class Allocator>
inline
basic_string<charT, traits, Allocator>
lower(const basic_string<charT,traits, Allocator>& str) {
        basic_string<charT, traits, Allocator> newstr(str);
        for (size_t index = 0; index < str.length(); index++)
                if (isupper(str[index]))
                        newstr[index] = tolower(str[index]);
        return newstr;
}
```

Then instead of calling `upper()` and `lower()` as member functions of the `basic_string`, pass the string as an argument. For example:

```
s2 = s1.upper(); // does not compile
s2 = upper(s1); // ok
```

- The String Package `match()` member function does not exist. Equivalent functionality exists in the Standard Library algorithm `mismatch()`, although using it is more complicated. For example:

```
string s1("abcdef");
string s2("abcdgf");
assert(s1.match(s2)==4); // does not compile
pair<string::iterator,string::iterator> p(0,0); // ok
p=mismatch(s1.begin(),s1.end(),s2.begin());
assert(p.first-s1.begin()==4);
string s3 = s1;
p=mismatch(s1.begin(),s1.end(),s3.begin());
assert(p.first == s1.end()); // everything matched
```

- The String Package `index()` member function does not exist. The `basic_string` library equivalent is `find()`.

- The String Package constructor that takes two positional parameters (a start and end position) and constructs a new string does not exist. It is replaced in the `basic_string` library with the member function `substr()`. For example:

```
string s1("abcde");
string s2 = s1(1,3); // does not compile
string s2 = s1.substr(1,3); // ok
```

- Many previously undetected run-time errors now throw standard exceptions in the String library.

7.6.4. Upgrading from the Class Library Complex to the ANSI Complex Class

Note

On I64 systems, the Class Library Complex package has been removed, so upgrading to the Standard Library complex class is the only option on this platform. *(I64 only)*

This section explains how to upgrade from the pre-ANSI complex library to the current standard complex library.

In the pre-ANSI library, complex objects are not templatized. In the ANSI library, complex objects are templatized on the type of the real and imaginary parts. The pre-ANSI library assumes the type is double, whereas the ANSI library provides specializations for float, double, and long double as well as allowing users to specialize on their own floating point types.

Mathematical error checking is not supported in the ANSI library. Users who rely on detection of underflow, overflow, and divide by zero should continue using the pre-ANSI complex library.

The following is a detailed list of important changes:

- Change `#include <complex.h>` or `#include <complex.hxx>` to `#include <complex>`.
- Change all declarations of `complex` to `complex<double>`, for example:

  ```
  complex c;
  ```

  Change to:

  ```
  complex<double> c;
  ```

- The `polar()` function no longer supplies a default value of 0 for the second argument. Users will have to explicitly add it to any calls that have only one argument, for example:

  ```
  complex c;
  c = polar(c); // get polar
  ```

  Change to:

  ```
  complex<double> c;
  c = polar(c,0.0);
  ```

- If you are calling a mathematical function or mathematical operator that takes scalars as arguments (`polar()` for example), then you must adjust the arguments you pass in to be the same type as the complex template parameter type. For example, you would have to change:

  ```
  complex c = polar(0,0);
  complex c2 = c+1;
  ```

  Change to:

  ```
  complex<double> c = polar(0.0,0.0); // 0.0 is double
  complex<double> c2= c + 1.0; // 1.0 is double
  ```

- The *complex_zero* variable is not declared in the complex header file. If you want to use it, you will have to declare it yourself. For example, add the following to the top of your source file:

```
static const complex<double> complex_zero(0.0,0.0);
```

- The `sqr()` and `arg1()` functions are gone. If you want to continue to use them, you should define them in one of your own headers, using the following definitions:

```
template <class T>
inline complex<T> sqr(const complex<T>& a)
{
    T r_val(real(a));
    T i_val(imag(a));
    return complex<T>
    (r_val * r_val -
     i-val * i_val,
     2 * r_val * -_val);
}

template <class T>
inline T arg1(const complex<T>& a)
{
    T val = arg(a);

    if(val > -M_PI && val <= M_PI)
        return val;

    if(val > M_PI)
        return val - (2*M_PI);

    // val <= -PI
    return val + (2*M_PI);
}
```

- The `pow(complex, int)` function is no longer provided. You must use `pow(complex<double>, double)`. This means changing calls such as:

```
pow(c,1);
```

  Change to:

```
pow(c,1.0);
```

  This might yield different results. If the function previously was underflowing or overflowing, it might not continue to happen.

- The complex output operator (<<) does not insert a space between the comma and the imaginary part. If you want the space, you would need to print the real and imaginary parts separately, adding your own comma and space; that is:

```
complex<double> c;
cout << "(" << c.real() << ", " << c.imag() << ")"; // add extra space
```

- The complex input operator (>>) does not raise an Objection if bad input is detected; it instead sets input stream's state to `ios::failbit`.

- Floating point overflow, underflow, and divide by zero do not set `errno` and will cause undefined behavior. Complex error checking and error notification is planned for a subsequent release.

- You should no longer need to link your program explicitly with the complex library. It is automatically linked in as part of the Standard Library. However, you must still explicitly link in the C math library, as shown in the following example:

```
#include <stdlib.h>
#include <complex>

int main() {
        complex<double> c1(1,1), c2(3.14,3.14);
        cout << "c2/c1: " << c2/c1 << endl;
        return EXIT_SUCCESS;
        % cxx example.cxx #error
        % cxx example.cxx -lm #okay
}
```

7.6.5. Upgrading from the Pre-ANSI iostream library to the VSI C++ Standard Library

This section explains how to upgrade from the pre-ANSI iostream library to the ANSI iostream library. In this section, pre-ANSI iostreams refers to versions of the iostream library found in the Class Library; ANSI iostreams refers to versions found in the VSI C++ Standard Library.

There are a number of differences between the pre-ANSI and ANSI iostream library. One major difference between the pre-ANSI and ANSI iostream library is that the ANSI library is templatized on the object input/output on which operations are being performed. In the pre-ANSI library, iostreams has no templates. The ANSI library also provides specializations for `char` and `wchar_t`.

Important differences are as follows:

- With the current compiler, you access the pre-ANSI iostream library by default in non `strict_ansi` compiler modes. You can control the version of iostreams you use with the `__USE_STD_IOSTREAM` and `__NO_USE_STD_IOSTREAM` macros. If you want to use the ANSI iostream library, do either of the following:
  - Enter /DEFINE=(__USE_STD_IOSTREAM) on the command line.
  - Put the following in your source file before any include files:

    ```
    #ifndef __USE_STD_IOSTREAM)
    #define __USE_STD_IOSTREAM)
    #endif
    ```

- Header names are different in the ANSI library, so to use ANSI iostreams, change the iostreams headers you include as follows:

From	To

`#include <iostream.h>`
 	
`#include <iostream.hxx>`
	`#include <iostream>`

`#include <fstream.h>`
 	
`#include <fstream.hxx>`
	`#include <fstream>`

`#include <strstream.h>`
 	
`#include <strstream.hxx>`
	`#include <strstream>`

From	To

`#include <iomanip.h>`
 	
`#include <iomanip.hxx>`
	`#include <iomanip>`

- All Standard Library names in the ANSI iostream library are in namespace `std`. Typically you would qualify each Standard Library name with `std::` or put `using namespace std;` at the top of your source file.

  To facilitate upgrading in all but STRICT_ANSI mode, `using namespace std;` is set by default. In STRICT_ANSI mode, after including an ANSI iostream header, you must qualify each name inside namespace `std` individually or do

  ```
  using namespace std;
  ```

- In the pre-ANSI iostream library, including `<iomanip.h>` or `<strstream.h>` gave you access to `cout`, `cin`, and `cerr`. To access the predefined streams with the ANSI iostream library, make the following changes:

change	change

`#include <iomanip.h>`
 	
to
 	
`#include <iomanip>`
 	
`#include <iostream>`
 	
`#include` using namespace std;
	`#include <strstream.h>`
	
	to
	
	`#include <strstream>`
	
	`#include <iostream>`
	
	`using namespace std;`

- The `istream::ipfx`, `istream::isfx`, `ostream::opfx`, `ostream::osfx` do not exist in the ANSI iostreams. Their functionality is provided by the sentry class found in `basic_istream` and `basic_ostream`, respectively.

  Common prefix code is provided by the sentry's constructor. Common suffix code is provided by the sentry's destructor. As a result, calls to `ipfx()`, `isfx()`, `opfx()`, and `osfx()` have their functionality replaced by construction and destruction of `std::istream::sentry objects` and `std::ostream::sentry object` respectively. For example:

```
#include <iostream.hxx>           |  #include <iostream.hxx>
void func (istream &is)           |  void func (ostream &os)
{                                 |  {
   if (is.ipfx())                 |     if (os.opfx())
       ...                        |         ...
   is.isfx();                     |     os.osfx();
}                                 |  }
                                  |
Would be coded as:                |  Would be coded as:
                                  |
#include <iostream>              |  #include <iostream>
void func (istream &is)           |  void func (ostream &os)
{                                 |  {
   istream::sentry ipfx(is);      |     ostream::sentry opfx(os);
   if (ipfx)                      |     if (opfx)
      ...                         |        ...
```

```
   //is.isfx(); implicit in dtor  |     //os.osfx(); implicit in dtor
}                                 |   }
```

- The following macros from the pre-ANSI `<iomanip.h>` are no longer available in `<iomanip>`:

  ```
  SMANIP, IMANIP, OMANIP, IOMANIP,
  SAPP,   IAPP,   OAPP,   IOAPP,
  SMANIPREF, IMANIPREF, OMANIPREF, IOMANIPREF,
  SAPPREF, IAPPREF, OAPPREF, IOAPPREF
  ```

  You can add them yourself, but their use will not be portable.

- The `streambuf::stossc()` function, which advances the get pointer forward by one character in a stream buffer, is not available in the ANSI iostream library. You can make use of the `std::streambuf::sbumpc()` function to move the get pointer forward one place. This function returns the character it moved past. These two functions are not exactly equivalent – if the get pointer is already beyond the end, `stossc()` does nothing, and `sbumpc()` returns EOF.

  ```
  istream &extract(istream &is)
  {
     ...
     is.rdbuf()->stossc();
  }
  ```

- `ios::bitalloc()` is no longer available in the ANSI iostream library.

- The `filebuf` constructors have changed in the ANSI iostream library. The pre-ANSI `filebuf` class contained three constructors:

  ```
  class filebuf : public streambuf
  {
     filebuf();
     filebuf(int fd);
     filebuf(int fd, char * p, int len);
     ...
  }
  ```

  In the ANSI iostream library, `filebuf` is a typedef for `basic_filebuf<char>`, and the C++ Working Paper defines `one filebuf` constructor:

  ```
  basic_filebuf();
  ```

  To facilitate backward compatibility, the ANSI iostream library does provide `basic_filebuf(int fd)` as an extension. However, the use of extensions is not portable.

  For example, consider the `filebuf` constructors in the following pre-ANSI iostream library program:

  ```
  #include <fstream.hxx>

  int main () {
      int fd = 1;
      const int BUFLEN = 1024;
      char buf [BUFLEN];
      filebuf fb(fd,buf,BUFLEN);
      filebuf fb1(fd);
      return 0;
  }
  ```

To be strictly ANSI conforming, you would need to recode as follows:

```
filebuf fb(fd,buf,BUFLEN); as filebuf fb();  and
filebuf fb1(fd); as filebuf fb1();
```

If you want to make use of the ANSI iostream `filebuf(fd)` extension, you could recode:

```
filebuf fb(fd,buf,BUFLEN); as filebuf fb(fd);  and
filebuf fb1(fd); as filebuf fb1(fd);
```

- The ANSI iostream library contains support for the `filebuf::fd()` function, which returns the file descriptor for the `filebuf` object and EOF if the `filebuf` object is closed as a nonportable extension. This function is not supported under the /STANDARD=STRICT_ANSI compiler mode.

- The following functions are not defined in the ANSI iostream library. They are provided in the ANSI iostream library for backward compatibility only. Their use is not portable.

```
ifstream::ifstream(int fd);
ifstream::ifstream(int fd, char *p, int len)
ofstream::ofstream(int fd);
ofstream::ofstream(int fd, char *p, int len);
fstream::fstream(int fd);
fstream::fstream(int fd, char *p, int len);
```

- The following `attach` functions, which attach, respectively, a `filebuf`, `fstream`, `ofstream`, and `ifstream` to a file are not available in the ANSI iostream library:

```
filebuf::attach(int);
fstream::attach(int);
ifstream::attach(int);
ofstream::attach(int);
```

If you do not want to make use of ANSI iostream library extensions, you must recode the use of `attach` as follows:

Change from:

```
#include <fstream.hxx>
#include <stdio.h>
#include <fcntl.h>
int main () {
    int fd;
    fd = open("t27.in",O_RDWR | O_CREAT, 0644);
    ifstream ifs;
    ifs.attach(fd);
    fd = creat("t28.out",0644);
    ofstream of;
    of.attach(fd);
    return 0;
}
```

To:

```
#include <fstream>
int main () {
    ifstream ifs("t27.in", ios::in | ios::out);
    ofstream ofs("t28.out");
```

```
    return 0;
}
```

- The ios enumerators for controlling the opening of files, `ios::nocreate` and `ios::noreplace`, are not available in the ANSI iostream library.

- The `istream_withassign` and `ostream_withassign` classes are not available in the ANSI iostream library.

- In the ANSI iostream library `ios_base::width()` applies to all formatted inserters including `operator << (char)`. This means that the stream width specified by either the `manipulator setw()` or the `ios_base::width()` member function will apply padding to the next output item even if it is a char.

  This was not the case in the pre-ANSI iostream library, where `width()` applied to all formatted inserters *except* the `char` inserter. The reasons for the change (to allow `ostream::operator<<(char)` to do formatting) are:

  1. It allows `operator<<` functions to do formatting consistently.

  2. It allows `operator<<(char)` and `put(char)` (formatted and unformatted operations on `char`) to have different functionality.

  Consider the following example:

```
#ifdef __USE_STD_IOSTREAM
#include <iostream>
#include <iomanip>
#else
#include <iostream.hxx>
#include <iomanip.hxx>
#endif
int main () {
    cout.width(10);
    cout.fill('^');
    cout << 'x' << '\n';
    cout << '[' << setw(10) << 'x' << ']' << endl;
    return 0;
}
```

In the ANSI iostream library the output is:

```
^^^^^^^^^x
[^^^^^^^^^x]
```

In the pre-ANSI iostream library the output is:

```
x
[x]^^^^^^^^^
```

- In the pre-ANSI iostream library, printing `signed char *` or a `unsigned char *` printed the address of the string. In the ANSI iostream library the string is printed. Consider the following example:

```
#ifdef __USE_STD_IOSTREAM
#include <iostream>
#else
#include <iostream.hxx>
#endif

int main () {
    char * cs = (char *) "Hello";
    signed char *ss = (signed char *) "world";
    unsigned char *us = (unsigned char *) "again";

    cout << cs << " " << ss << " " << us << endl;
    return 0;
}
```

The output in the ANSI iostream library is:

```
Hello world again
```

The output in the pre-ANSI iostream library is:

```
Hello 0x120001748 0x120001740
```

To obtain output equivalent to the pre-ANSI iostreams, you might do the following:

```
cout << hex << showbase << (long) ss << " "
     << (long) us << endl;
```

- In the pre-ANSI iostream library printing a `signed char` prints its integer value. In the ANSI iostream library printing a `signed char` prints it as a character. Consider the following example:

```
#ifdef __USE_STD_IOSTREAM
#include <iostream>
#else
#include <iostream.hxx>
#endif

int main () {
    signed char c = (signed char) 'c';
    cout << c << endl;
    return 0;
}
```

The output in the ANSI iostream library is:

```
c
```

The output in the pre-ANSI iostream library is:

```
99
```

To obtain output equivalent to the pre-ANSI iostreams, you must do the following:

```
cout << (long) c << endl;
```

- In the ANSI iostream library, reading invalid floating point input (where invalid input is caused by no digits following the letter e or E and an optional sign) from a stream sets failbit to flag this error state. In the pre-ANSI iostream library, these type of error conditions might not be detected. Consider this program fragment:

```
double i;
cin >> i;
cout << cin.rdstate() << ' ' << i << endl;
```

On the input: 123123e

The output in the ANSI iostream library is:

```
4 2.65261e-314          // failbit set
```

The output in the pre-ANSI iostream library is:

```
0 123123                // goodbit set
```

- In the ANSI iostream library, reading integer input (which is truncated as the result of a conversion operation) from a stream sets failbit to flag this overflow condition. In the pre-ANSI iostream library, these types of conditions might not be detected. Consider this program fragment:

```
int i;
cin >> i;
cout << cin.rdstate() << ' ' << i << endl;
```

On the input: 9999999999999999

The output in the ANSI iostream library is:

```
4  1874919423       // failbit set
```

The output in the pre-ANSI iostream library is:

```
0  1874919423       // failbit not set
```

In the ANSI iostream library, reading -0 from a stream into an unsigned `int` outputs 0; this was not the case with the pre-ANSI iostream library. Consider the following:

```
unsigned int ui;
cin >> ui;
cout << cin.rdstate() << ' ' << ui << endl;
```

On the input: -0

The output in the ANSI iostream library is:

```
0 0
```

- In the ANSI iostream library, the `istream::getline()` function extracts characters and stores them into successive locations of an array whose first element is designated by s. If fewer than `n` characters are input, failbit is set. This was not the case in the pre-ANSI iostream library. Consider the following:

```
#include <stdlib.h>
int main()
{
        char buffer[10];
        cin.getline (buffer,10);
        cout << cin.rdstate() << ' ' << buffer << endl;
        return EXIT_SUCCESS;
}
```

With input of: 1234567890

The output in the ANSI iostream library is:

```
4 123456789
```

The output in the pre-ANSI iostream library is:

```
0 123456789
```

- When printing addresses, the ANSI library does not print a leading “0x” to indicate a hexadecimal base. The pre-ANSI library did. Consider the following:

```
#include <stdlib.h>
#include <iostream>

int main()
{
        double d;
        int i;
        void *p = (void *) &d;
        int *pi = &i;
        cout << (void *) 0 << ' ' << p << ' ' pi << endl;
        return EXIT_SUCCESS;
}
```

The output in the ANSI iostream library is:

```
0 11fffe7a0 11fffe798
```

The output in the pre-ANSI iostream library is:

```
0x0 0x11fffdc40 0x11fffdc38
```

- `basic_filebuf::setbuf` is a protected member function in the ANSI iostream library. Therefore, the following longer compiles:

```
#include <stdlib.h>
int main() {
   filebuf fb;
   ...
   fb.setbuf(0,0);
```

```
    return EXIT_SUCCESS;
}
```

- In the ANSI iostream library, the Standard C++ streams are synchronized with the Standard C streams by default. Calling `sync_with_stdio()` with `false` allows the Standard C++ streams to operate independently of the Standard C streams. In the pre-ANSI iostream library the Standard C ++ streams are **not** synchronized with the Standard C streams by default.

  You notice the consequences of this change if you redirect the output of a program using the Standard C++ streams to a log file by entering the following commands at the DCL prompt:

  ```
  $ define sys$output t.out
  $ run program
  $ deassign sys$output
  ```

  For example, if you write something like this using ANSI iostreams:

  ```
  #ifndef __USE_STD_IOSTREAM
  #define __USE_STD_IOSTREAM
  #endif
  #include <iostream>
  void main () {
      int s = 5;
      cout << "i" << s;
  }
  ```

  and if you redirect the output to a log file using the commands shown in the example, the log file contains two records:

  ```
  i
  5
  ```

  If you write something like this using pre-ANSI iostreams:

  ```
  #include <iostream.hxx>
  void main () {
      int s = 5;
      cout << "i" << s;
  }
  ```

  and if you redirect the output to a log file using the commands shown in the example, the log file contains one record: `i5`.

  To obtain the Pre-ANSI iostreams behavior with ANSI iostreams, you can use either of the following workarounds:

  - Redirect your output to a file by entering the following commands:

    ```
    $ define/user sys$output t.out
    $ run program
    ```

    The output of the log file, `t.out`, contains one record: `i5`.

  - Recode your program so that the Standard C++ streams operate independently of the Standard C streams. Do this by calling `sync_with_stdio()` with a `false` argument as follows:

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

```
#ifndef __USE_STD_IOSTREAM
#   define __USE_STD_IOSTREAM
#endif
#include <iostream>
int main () {
    ios_base::sync_with_stdio(false);
    int s = 5;
    cout << "i" << s;
    return EXIT_SUCCESS
}
```

If you now redirect the output to a log file, the log file contains one record: `i5`.

Chapter 8. Using the OpenVMS Debugger

A debugger helps you find run-time errors by letting you observe and interactively manipulate program execution step by step, until you discover where the program functions incorrectly. The OpenVMS Debugger is symbolic, meaning that you can refer to symbolic names for the memory addresses allocated to variables, routines, labels, and so on. You need not use virtual addresses.

The language of the source program you are currently debugging determines the format you use to enter and display data. The language also determines the format used for features, such as comment characters, operators, and operator precedence, which have language-specific settings. However, if you have modules written in another language, you can switch from one language to another during your debugging session.

8.1. Debugging C++ Programs

The OpenVMS Debugger supports the language constructs of C++ and other debugger-supported programming languages. This section describes features specific to debugging C++ programs. For more information on the OpenVMS Debugger, see the *VSI OpenVMS Debugger Manual*.

8.1.1. Compiling and Linking in Preparation for Debugging

To use the debugger, compile and link your program with the /DEBUG qualifier on both commands. On the compiler command, the /DEBUG qualifier writes into the object module the debug symbol records declared in the program source file. These records make the names of variables and other declared symbols accessible to debugger commands. If your program has several compilation units, make sure you use the /DEBUG qualifier to compile each unit you want to debug.

On OpenVMS I64 systems, specifying /DEBUG gives
you /DEBUG=(TRACEBACK,SYMBOLS=BRIEF), which omits debug information for unused labels and unused types, even when /NOOPTIMIZE is specified. This feature results in much smaller object files. To include unused labels and types, specify the SYMBOLS=NOBRIEF keyword explicitly (/DEBUG=(SYMBOLS=NOBRIEF)).

On OpenVMS Alpha systems, specifying /DEBUG gives you /DEBUG=(TRACEBACK,SYMBOLS), which effectively gives you /DEBUG=(TRACEBACK,SYMBOLS=NOBRIEF).

Additionally, use the /NOOPTIMIZE qualifier with the compiler command. Optimized code can reduce program size and increase execution speed, but can also create inconsistencies in memory content that adversely affects debugging. Use the default /OPTIMIZE qualifier only with programs that have been completely debugged.

8.1.2. Debugger Support

Additionally, compilation with normal (full) optimization will have the following noticeable effects on *OpenVMS Alpha systems*:

- Stepping by line will generally seem to bounce forward and back, due to the effects of code scheduling. The general drift will definitely be forward, but initial experience indicates that the effect will be very close to stepping by instruction.

- Variables that are “split” (so that they are allocated in more than one location during different parts of their lifetimes) are not described at all.

  Although not handled quite like normal split variables, formal parameters that are passed in registers share many of the same problems as split variables. Even with the /NOOPTIMIZE qualifier, such a parameter often will be copied immediately to a “permanent home” (either on the stack or in some other register) during the routine prolog. The debugger symbol table description of such parameters encodes this permanent home location and *not* the physical register in which the parameter is passed. The end-of-prolog location is recorded in the debugger symbol tables and will be used as the preferred breakpoint location when a breakpoint is set in the context of an appropriately set module (so that symbol table information is available to the debugger).

On the linker command, the /DEBUG qualifier incorporates into the executable image all the symbol information contained in the object modules. Using the /DEBUG qualifier on the linker command also starts the debugger at run time.

Debugger Command-Line Options

The compiler provides a set of debugger options that you can specify to the /DEBUG qualifier on the compiler command line. These options determine the kind of information that the compiler places in the object module for use by the OpenVMS Debugger. These debugger options include using traceback records and using the debugger symbol table. For more information, see the /DEBUG qualifier in Appendix A.

8.1.3. Starting and Ending a Debugging Session

When you enter the DCL `run` command and specify your executable image file, the OpenVMS Debugger takes control. The debugger displays a message indicating its version, the programming language the source code is written in, and the name of the image file. When the DBG> prompt appears, you can enter debugger commands.

To execute the program, enter the debugger `go` command. Execution proceeds until the debugger pauses or stops the program (for example, to prompt you for user input, to signal an error, or to inform you that your program completed successfully).

To interrupt the debugging session in progress, press Ctrl/C. The DBG> prompt displays and you can again enter debugger commands.

To end a debugging session, enter the debugger `exit` command or press Ctrl/Z.

8.1.4. Features Basic to Debugging C++ Programs

This section describes features essential for debugging C++ programs.

8.1.4.1. Determining Language Mode

The OpenVMS Debugger is in C++ language mode when invoked against a main program or routine written in C++. If you are debugging an application with modules written in some language other than C++, you may switch back to C++ language mode by using the command `set language c_plus_plus`.

You can use the `show language` command to determine the language mode set. For example:

```
DBG> show language
language: C_PLUS_PLUS
DBG>
```

8.1.4.2. Built-In Operators

This section describes the built-in operators that you can use in debugger commands. The operators in C ++ language expressions are as follows:

Symbol	Function	Kind
*	Indirection	Prefix
&	Address of	Prefix
`sizeof`	size of	Prefix
–	Unary minus (negation)	Prefix
+	Addition	Infix
–	Subtraction	Infix
*	Multiplication	Infix
/	Division	Infix
%	Remainder	Infix
<<	Left shift	Infix
>>	Right shift	Infix
==	Equal to	Infix
!=	Not equal to	Infix
>	Greater than	Infix
>=	Greater than or equal to	Infix
<	Less than	Infix
<=	Less than or equal to	Infix
~ (tilde)	Bit-wise NOT	Prefix
&	Bit-wise AND	Infix
|	Bit-wise OR	Infix
^	Bit-wise exclusive OR	Infix
!	Logical NOT	Prefix
&&	Logical AND	Infix
||	Logical OR	Infix

Because the exclamation point (!) is an operator, it cannot be used in C++ programs as a comment delimiter. However, to permit debugger log files to be used as debugger input, the debugger still recognizes the exclamation point as a comment delimiter if it is the first nonspace character on a line. In C++ language mode, the debugger accepts a forward slash immediately followed by an asterisk (/*) as the comment delimiter. The comment continues to the end of the current line. A matching asterisk immediately followed by a slash (*/) is neither needed nor recognized.

The debugger accepts the asterisk (*) prefix as an indirection operator in both C++ language expressions and debugger address expressions. In address expressions, the asterisk prefix is synonymous to the period (.) prefix or the at sign (@) prefix when the language is set to C++.

To prevent unintended modifications to the program being debugged, the debugger does not support any of the assignment operators in C++ (or any other language). Thus, such operators as =, +=, −, ++, and −− are not recognized. To alter the contents of a memory location, you must do so with an explicit `deposit` command.

8.1.4.3. Constructs in Language and Address Expressions

The supported constructs in language and address expressions for C++ are as follows:

Symbol	Construct
[ ]	Subscripting
. (period)	Structure component selection
->	Pointer dereferencing

8.1.4.4. Data Types

Predefined data types supported in the debugger are as follows:

C++ Data Type	OpenVMS Data Type Name
`int`, `long`	Longword Integer
`unsigned int`, `unsigned long`	Longword Unsigned
`long long`	Quadword Integer
`unsigned long long`	Quadword Unsigned
`short int`	Word Integer
`unsigned short int`	Word Unsigned
`char`	Byte Integer
`unsigned char`	Byte Unsigned
`float`	F_Floating (Alpha default), S_Floating (I64 default)
`double`	G_Floating (Alpha default), T_Floating (I64 default), D_Floating
`enum`	None
`struct`	None
`union`	None
`class`	None
Pointer type	None
Array type	None

Uppercase letters in parentheses represent standard data type mnemonics in the OpenVMS common language environment. For more information, see *OpenVMS Programming Interfaces: Calling a System Routine*.

Supported data types specific to OpenVMS systems are as follows:

C++ Data Type	OpenVMS Data Type Name
`__int16`	Word Integer

C++ Data Type	OpenVMS Data Type Name
`unsigned __int16`	Word Unsigned
`__int32`	Longword Integer
`unsigned __int32`	Longword Unsigned
`__int64`	Quadword Integer
`unsigned __int64`	Quadword Unsigned

8.2. Using the OpenVMS Debugger with C++ Data

This section describes how to use the OpenVMS Debugger with C++ data.

8.2.1. Nonstatic Data Members

This section describes how to refer to data members that are not declared static.

8.2.1.1. Noninherited Data Members

To refer to a nonstatic data member that is defined directly in a C++ class (or a `struct` or `union`), use its name just as with a C language `struct` or `union` member. The following example shows the correct use of a nonstatic data member reference:

```
DBG> examine x.m, p->m
```

8.2.1.2. Inherited Data Members

Currently, debugger support distinguishes nonstatic data members inherited from various base classes by prefixing their names with a sequence of *significant* base class names on the inheritance path to the member, and then the class that the member is declared in. A base class on a path from an object to a member is significant if the base class in question is derived from using multiple inheritance. Thus, a base class is significant if it is mentioned in a base list containing more than one base specifier.

This notation generates the minimum number of base class prefixes necessary to describe the inheritance path to a base class, because it involves naming only those base classes where one must choose where to proceed next when traversing the path. When no multiple inheritance is involved, the reference has the following syntax:

```
CLASS::member
```

Specify the sequence of significant base classes in the order from the object's most derived significant class, to the significant base class closest to the object.

8.2.2. Reference Objects and Reference Members

Because the debugger understands the concept of reference objects and reference members to objects, you can examine a reference object or reference member directly, without dereferencing it as you would for a pointer. To access the values of objects declared with a reference, use the name of the object.

For example, consider the following code:

```
class C {
public:
    int &ref_mem;
    C(int &arg) : ref_mem(arg) {}
};

main()
{
    auto int obj = 5;
    auto int &ref_obj = obj;
    auto C c(obj);
    obj = 23;
}
...
```

The following sequence shows the correct way to use the debugger to examine the members:

```
break at R8_2_3\main\%LINE 13
    13: }
DBG> exam obj, ref_obj
R8_2_3\main\obj:        23
R8_2_3\main\ref_obj:    23
DBG> exam c
R8_2_3\main\c: class C
    ref_mem:    23
DBG> exam c.ref_mem
R8_2_3\main\c.ref_mem:  23
```

8.2.3. Pointers to Members

For Alpha systems compiled with /MODEL=ANSI and for I64 systems, a pointer to member is an offset into a structure.

Consider the following example:

```
struct A {
    int mem0;
};

struct B {
    int mem1;
    int mem2;
};

struct C : public A, public B {
    int mem3;
    int mem4;
};

/* pointer to member initalized with pointer to member
 * address of the same class.
 */
int C::*pmc = &C::mem2;

/* pointer to member initialized with pointer to member
 * address of one of the  * base classses.  An implicit
 *  conversion occurs.
 */
```

```
int C::*pmbc = &B::mem2;

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

main()
{
 C *cinst = new C;
 cinst->*pmc = 7;
 printf("cinst pointer to member value is %d\n",cinst->mem2);
}
```

If you compile this program with the /NOOPTIMIZE/DEBUG qualifiers, from the last line in the program, you can use the pointer to member to display the following information:

```
DBG> set radix hex
DBG> exam *cinst
*EX8_2_4\main\cinst: struct C
    inherit A
        mem0:   00000000
    inherit B
        mem1:   00000000
        mem2:   0000000A
    mem3:       00000000
    mem4:       00000000

DBG> set radix hex

DBG> exa pmc
EX8_2_4\pmc:    00000008

DBG> exam pmbc
EX8_2_4\pmbc:   00000008

DBG> exam cinst
EX8_2_4\main\cinst:     0000000080000090

DBG> exam 080000090+8
0000000080000098:       00000007
```

For the preceding sample program, the above debug sequence examines the pointer to member (`pmc` or `pmbc`) to obtain an offset into the structure, and adds this value to the address of the object (`*cinst`). In our example, this is `*cinst` + the value of `pmc`.

For Alpha systems compiled with the default object model (/MODEL=ARM), a pointer to member involves executing a piece of function-like code, called a thunk.

The argument to this function is the address of the base class containing the member. This address is obtained by adding the offset of the start of the base class to the address of the object. This offset adjustment is needed when the pointer to member refers to a multiply inherited base class.

A sample debug sequence for the previous program example follows:

```
DBG> set radix hex
DBG> sho sym /full C
type C
    struct (C, 2 components), size: 20 bytes
      inherits: A, size: 4 bytes, offset: 0000000000000000 bytes
                B, size: 8 bytes, offset: 0000000000000004 bytes
```

```
      contains the following members:
        mem3 : longword integer, size: 4 bytes, offset: 000000000000000C
 bytes
        mem4 : longword integer, size: 4 bytes, offset: 0000000000000010
 bytes
DBG> exam pmc
EX8_2_4\pmc:    000100D8

DBG> exam cinst
EX8_2_4\main\cinst:     006706F0

DBG> call 000100D8(006706F0+4)
value returned is 006706F8

DBG> exam 006706F8
00000000006706F8:       00000007
```

This debug sequence first obtains the offset to the start of the nested class containing the member pointed to with `show sym /full`. In this case, the offset is 4.

It then examines the pointer to member (`pmc`) and determines the address of the object (`cinst`). In our case, `pmc` = 100D8 (thunk) and `cinst` = 6706F0.

Then it calls the thunk, passing the address of the object plus the offset: CALL 000100D8(006706F0+4). This call to the thunk returns the address of the member.

Finally, it examines the member (006706F8).

8.2.4. Referencing Entities by Type

To examine and display the value of an object or member by type, use the command `examine/type`. Similarly, you can modify the value of an expression to be deposited to a type you specify by using the command `deposit/type`. With the `/type` qualifier, the syntax for these commands is as follows:

```
deposit/type=(name)
examine/type=(name)
```

The type denoted by *name* must be the name of a variable or data type declared in the program. The `/type` qualifier is particularly useful for referencing C++ objects that have been declared with more than one type.

8.3. Using the OpenVMS Debugger with C++ Functions

This section describes how to reference the various kinds of functions and function arguments.

8.3.1. Referring to Overloaded Functions

You can use the debug SHOW SYMBOL command to see all the overloaded names for a given function. You can set breakpoints on an overloaded function by specifying either the object name and function name followed by the argument types, or by specifying the class name and function name followed by the arguments.

For example, consider the following sample program:

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

class base{
    public:
        base(){};
        base(int){};

        ~base(){};

        void base_f1() {printf("called base_f1()\n");}

        void base_f2() {printf("called base_f2()\n");}
        void base_f2(int) {printf("called  base_f2(int)\n");}
        void base_f2(char c) {printf("call base_f2(char)\n");}
};

int main()
{
    base b;
    base b1(1);
    b.base_f1();
    b.base_f2(10);
    b.base_f2();
    b.base_f2('c');
}
```

The following debug sequence for the previous sample program shows how to set breakpoints on overloaded symbols and how to list these functions:

```
DBG> s
stepped to EX8_3_1\main\%LINE 20
    20:     base b1(1);
DBG> set break base::base_f1
DBG> set break base::base_f2
%DEBUG-I-NOTUNQOVR, symbol 'base::base_f2' is overloaded
overloaded name base::base_f2
       instance base::base_f2(char)
       instance base::base_f2(int)
       instance base::base_f2()
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> set break base::base_f2(char)
```

8.3.2. Referring to Destructors

The C++ I64 debugger supports the following format for setting a break on a destructor:

```
DBG> set break stack::~stack()
DBG> set break stack::~stack(int)
```

Older (Alpha) debuggers require use of the %name syntax:

```
DBG> set break stack::%name'~stack'
```

8.3.3. Referring to Conversions

The set of atomic types are drawn from the following set of names:

```
void              char       signed_char     unsigned_char   signed_short
unsigned_short    int        signed_int      unsigned_int    signed_long
unsigned_long     float      double          long_double
```

Pointer types are named `(type)*`. Reference types are named `(type)&`. The types `struct`, `union`, `class`, and `enum` are named by their tags, and the qualifiers `const` and `volatile` precede their types with a space in between.For example:

```
DBG> set break C::int, C::(const S)&
```

8.3.4. Referring to User-Defined Operators

The following operators can be overloaded by user-defined functions:

```
+       -       *       /       %       ^
&       |       ~       !       =       <
>       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=
<<=     ==      !=      <=      >=      &&
|       ++      -      ->*     ,       ->
[]      ()      delete  new
```

The following example shows the correct use of user-defined function references:

```
DBG> set break stack::%name'operator++'()
```

8.3.5. Referring to Function Arguments

In OpenVMS Debugger referencing, you use `this`, `*this`, and `this->m` as follows:

- All nonstatic member functions have a pointer parameter available named `this`. For example:

  ```
  DBG> examine this
  ```

- Use `*this` to examine the prefix object that a member function is invoked against. For example:

  ```
  DBG> examine *this
  ```

- Use the `this` parameter to refer to a data member `m` of the prefix argument to a member function. For example:

  ```
  DBG> examine this->m
  ```

8.3.6. Calling C++ Member Functions from the Debugger

When calling C++ member functions from the debugger, you cannot make the call using the same syntax that you would use in a C++ source file. You must call the class-qualified member function name with the object as the first argument.

For example:

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

```
class C12 {
        int i;
        int j;
public:
        static int sum;
public:
        C12() : i(1), j(2) {}
        void method();
        static int get_sum() {
           printf("called static function get_sum()\n");
           return sum;
           }
};
void C12::method()

{
        i = i + j;
        printf("C12::method called: i=%d, j=%d\n",i,j);
}

int C12::sum = 0;

main()
{
        C12 cinst;
        cinst.method();
        C12::get_sum();
        printf("End of example.\n");
}
```

When you compile this example with /DEBUG/NOOPT, you can call the member function with the following command:

```
DBG> call C12::method(cinst)
```

Be aware that when a nonstatic member function is called, the compiler passes an implicit first parameter, the "this" pointer. But, when using the debugger's call instruction, you must explicitly pass this hidden first argument:

```
//Call the nonstatic member function:
DBG> call cinst.method(cinst)
C12::method called: i=3 j=2
value returned is 28
// notice that the following call confuses debug:
DBG> call cinst.method()
%DEBUG-E-MISOPEMIS, misplaced operator or missing operand at 'end of
expression'
```

However, when calling a static member function, there is no implict this pointer and there function may be called using the class name or the object name:

```
// Call the static function:
DBG> call C12::get_sum
called static function get_sum()
value returned is 0
DBG> call cinst.get_sum
called static function get_sum()
```

```
value returned is 0
```

Chapter 9. Using 64-bit Address Space

This chapter describes 64-bit address support for the VSI C++ compiler on OpenVMS Alpha and I64 systems.

The introduction of 64-bit address space in OpenVMS greatly increases the amount of memory available to applications. VSI C++ has been enhanced to permit use of this memory. The compiler provides a great deal of flexibility about how this memory can be used. Conceptually, this flexibility can be viewed as four models for development:

- 32-bit development
- 64-bit development
- 32-bit development with long pointers
- 64-bit development with short pointers

In a 32-bit development environment, all pointers are 32-bits long and only 2 gigabytes of address space is available. This is the default and was the only option that was available before this version of the compiler. In a 64-bit development environment, all pointers are 64-bits long and the address space is over a billion gigabytes.

Working in a homogeneous 32-bit or 64-bit environment is the preferred and recommended way to do development. VSI C++ for OpenVMS, combined with the C Run-Time library, provide a seemless environment for development. It should be possible for a well written, portable program developed using 32-bit pointers to be recompiled and relinked to use 64-bit pointers.

Because it is not always possible or desirable to work in a homogeneous pointer environment. VSI C++ supports mixed pointer sizes, however, it requires greater care by developers. Some contexts where heterogeneous pointer sizes might be used are:

- Memory requirements of 32-bit application exceeds 2 gigabytes
- Access to a legacy 32-bit library is required from a 64-bit application
- The memory foot print of a 64-bit application needs to be reduced

When the memory requirements of a 32-bit application begins to exceed 2 gigabytes, the most straight forward solution is to convert the application to be a 64-bit application. Since practical considerations, like the size of the application or the lack of source code for all parts can prevent this, the alternative approach of isolating the use of 64-bit pointers to a small portion of the application may be preferable. In this situation, development would continue in the 32-bit environment, using long pointers when necessary.

When doing 64-bit development, there are times when it becomes necessary or desirable to use 32-bit pointers. The most common instance is interfacing with a 32-bit library. Another is to save space, because 64-bit pointers consume twice as much memory as 32-bit pointers. In this situation, development could be done in a 64-bit environment, using short pointers when necessary.

Limited empirical evidence suggests that using 32-bit pointers to save space can reduce memory consumption by approximately 25% but at the cost of greater complexity and the creation of potentially unnecessary constraints in the application.

9.1. 32-bit Versus 64-bit Development Environment

Besides pointer size, the following components of the development environment determine whether it is a 32- or 64-bit environment:

- Memory allocators
- Libraries

Memory allocators control where in the address space memory is allocated. Memory can be allocated in 32- or 64-bit space independent of the pointer size. The default memory allocator is appropriate for the development environment being used.

Libraries in a 32-bit environment expect pointers to be 32-bits and memory to reside in the 32-bit address space, while libraries in the 64-bit environment expect pointers to be 64-bits.
VSI C++ for OpenVMS ships with two libraries: one for the 32-bit environment and one for the 64-bit environment. In addition to supporting the 64-bit environment, the second library also supports the new object model referred to as model ANSI.

/MODEL=ANSI only affects the pointer size on Alpha systems. The qualifier is silently ignored on I64 systems.

Caution

When compiling /POINTER_SIZE=LONG, the STL template classes (such as string, and set, map) can be used only when /MODEL=ANSI is specified.

The C Runtime is a single library that supports both environments. See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]* for information about how support for both environments was achieved with a single library. See Section 9.6 for a discussion of why it is difficult to produce a single C++ library to support both environments.

9.1.1. Model ANSI *(Alpha only)*

The new ANSI object model allows the compiler to better conform to the ANSI/ISO C++ standard while providing the 64-bit development environment. This object model is specified using the /MODEL=ANSI compiler and link options. /MODEL=ANSI only affects the pointer size on Alpha systems. The qualifier is silently ignored on I64 systems.

To build a 64-bit application using the ANSI object model, you enter commands in the following format:

```
$ cxx /model=ansi filename.cxx
$ cxxlink/model=ansi filename
```

Caution

The new ANSI object model is not compatible with the old object model. You must compile and link your entire application with one model or the other.

9.1.2. Memory Allocators

In C++, the primary memory allocator is `new`. Use of the default allocators causes memory to be allocated that is appropriate for the default pointer size for the module (not the current pointer size). Specialized placement-new allocators can be used to control where an object is allocated. The header `newext.hxx` contains the following definitions:

```
enum addr32_t (addr_32 };
enum addr64_t {addr_64 };

#pragma pointer_size short
void *operator new(addr32_t, size_t s) { return _malloc32(s); }
void *operator new[](addr32_t, size_t s) { return _malloc32(s); }

#pragma pointer_size long
void *operator new(addr64_t, size_t s) { return _malloc64(s); }
void *operator new[](addr64_t, size_t s) { return _malloc64(s); }
```

Use of the allocators from the C Run Time is also possible. You can select a specific C allocator by adding a prefix underbar and either 32 or 64 as a suffix.

Function	32-bit	64-bit
malloc	_malloc32	_malloc64
calloc	_calloc32	_calloc64
realloc	_realloc32	_realloc64
strdup	_strdup32	_strdup64

When attempting to mix pointer sizes in your program, distinguish between the concepts of pointer size and memory allocators. The pointer size dictates the maximum amount of address space a pointer can reference, while the allocator controls the where the memory will be allocated.

A library implemented with 64-bit pointers that uses only a 32-bit allocator can with care be used by an application that uses 32-bit pointers. If the library uses a 64-bit allocator, the application cannot reference any pointers returned. To a large extent, it is the memory allocator, not the pointer size, that determines interoperability.

9.1.3. 64-bit Pointer Support in the C Run Time Library

In addition to allocators, other functions in the C Run Time Library, such as `strcpy`, are affected by pointer size. As with the alloators, the C++ compiler calls a version of the routine is for the development environment. See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]* for more details.

9.2. Qualifiers and Pragmas

The following qualifiers, pragmas, and predefined macros control pointer size:

- /MODEL=ANSI *(Alpha only)*
- /[NO]POINTER_SIZE={LONG | SHORT | 64 |32}
- `#pragma pointer_size`

- `#pragma required_pointer_size`
- `#pragma environment cxx_header_defaults`
- `__INITIAL_POINTER_SIZE` predefined macro

9.2.1. The /MODEL=ANSI Qualifier *(Alpha only)*

The /MODEL=ANSI qualifier enables the new ANSI object model. This model implies /POINTER_SIZE=LONG in addition to supporting new C++ constructs that could not be supported in the object model designed to support the ARM definition of the language. This option must be specified during compilation and linking. /MODEL=ANSI only affects the pointer size on Alpha systems. The qualifier is silently ignored on I64 systems.

9.2.2. The /POINTER_SIZE Qualifier

The /POINTER_SIZE qualifier lets you specify a value of 64 or 32 (or LONG or SHORT) as the default pointer size within the compilation unit. You can compile one set of modules using 32-bit pointers and another set using 64-bit pointers. Take care when these two separate groups of modules call each other.

The default is /NOPOINTER_SIZE, which has the following effects:

- Disables pointer-size features, such as the ability to use `#pragma pointer_size`
- Directs the compiler to assume that all pointers are 32-bit pointers

This default represents no change from previous versions of VSI C++.

Specifying /POINTER_SIZE with a keyword value (32, 64, SHORT, or LONG) has the following effects:

- Enables processing of `#pragma pointer_size`.
- Sets the initial default pointer size to 32 or 64, as specified.
- Predefines the preprocessor macro `__INITIAL_POINTER_SIZE` to 32 or 64, as specified. If /POINTER_SIZE is omitted from the command line, `__INITIAL_POINTER_SIZE` is 0, which allows you to use `#ifdef __INITIAL_POINTER_SIZE` to test whether the compiler supports 64-bit pointers.
- For /POINTER_SIZE=64, the C RTL name mapping table is changed to select the 64-bit versions of `malloc`, `calloc`, and other RTL routines by default.

Use of the /POINTER_SIZE qualifier also influences the processing of C RTL header files:

- For those functions that have both 32-bit and 64-bit implementations, specifying /POINTER_SIZE enables function prototypes to access both functions, regardless of the actual value supplied to the qualifier. The value specified to the qualifier determines the default implementation to call during that compilation unit.
- Functions that require a second interface to be used with 64-bit pointers reside in the same object libraries and shareable images as their 32-bit counterparts. Because no new object libraries or shareable images are introduced, using 64-bit pointers does not require changes to your link command or link options files.

See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]* for more information on the impact of 64-bit pointer support on VSI C++ RTL functions.

9.2.3. The __INITIAL_POINTER_SIZE Macro

The `__INITIAL_POINTER_SIZE` preprocessor macro is useful for header-file authors to determine:

- Whether the compiler supports 64-bit pointers.
- Whether the application expects to use 64-bit pointers.

Header-file code can then be conditionalized using the following preprocessor directives:

```
#if defined (<double_uscore>INITIAL_POINTER_SIZE) /* Compiler supports
                                                     64-bit pointers */
#if <double_uscore>INITIAL_POINTER_SIZE > 0   /* Application uses
                                                 64-bit pointers */
#if <double_uscore>INITIAL_POINTER_SIZE == 32 /* Application uses some
 64-bit pointers, but default RTL routines are 32-bit.*/

#if <double_uscore>INITIAL_POINTER_SIZE == 64 /* Application uses 64-bit
pointers and default RTL routines are 64-bit. */
```

9.2.4. Pragmas

The `#pragma pointer_size` `#pragma required_pointer_size` and preprocessor directives can be used to change the pointer size currently in effect within a compilation unit. You can default pointers to 32-bits and then declare specific pointers within the module as 64-bits. In this case, you also need to specifically call the appropriate allocator to obtain memory from the 64-bit memory area.

These pragmas have the following format:

```
#pragma pointer_size keyword

#pragma required_pointer_size keyword
```

The *keyword* is one of the following:

{`short`|32}	32-bit pointer
{`long`|64}	64-bit pointer
`save`	Saves the current pointer size
`restore`	Restores the current pointer size to its last saved state

The `#pragma pointer_size` and `#pragma required_pointer_size` directives work essentially the same way, except that `#pragma required_pointer_size` always takes effect regardless of command-line qualifiers, while `#pragma pointer_size` is in effect only when the /POINTER_SIZE command-line qualifier is used.

By changing the command-line qualifier, `#pragma pointer_size` allows a program to be built using 64-bit features as purely as a 32-bit program.

The `#pragma required_pointer_size` is intended for use in header files where interfaces to system data structures must use a specific pointer size regardless of how the program is compiled.

An alternative to control the pointer size is `#pragma environment`. This pragma controls all compiler states that include pointer size. This pragma is fully documented in Section 2.1.1.3. The primary change for support of long pointers is the addition of a new **cxx_header_defaults** keyword.

This new keyword is similar to the **header_defaults** keyword, but differs in the effect on `pointer_size`. With **header_defaults**, `pointer_size` is made short, while with **cxx_header_defaults**, the pointer_size depends on the model being used. When developing in model ANSI, the `pointer_size` is 64 bits; in model ARM (the default), it is 32 bits.

9.3. Determining Pointer Size

The pointer-size qualifiers and pragmas affect only a limited number of constructs in the C++ language itself. At places where the syntax creates a pointer type, the pointer-size context determines the size of that type. Pointer-size context is defined by the most recent pragma (or command-line qualifier) affecting pointer size.

Here are examples of places in the syntax where a pointer type is created:

- The * in a declaration or cast:

```
int **p;  // Declaration
ip = (int **)i;  // Cast
```

- The outer (leftmost) brackets [] in a formal parameter imply a *:

```
void foo(int ia[10][20]) {}

// Means the following:

void foo(int (*ia)[20]) {}
```

- A function declarator as a formal parameter imply a *:

```
void foo (int func()):

// Means the following:

void foo (int (*)() func);
```

- Any formal parameter of array or function type implies a *, even when bound in a `typedef`:

```
typedef int a_type[10];

void foo (a_type ia);

// Means the following:

void foo (int *ia);
```

9.3.1. Special Cases

The following special cases are not affected by pointer-size context:

- Formal parameters to `main` are always treated as if they were in a `#pragma pointer_size system_default` context, which is 32-bit pointers for OpenVMS systems.

  For example, regardless of the `#pragma pointer_size 64` directive, argv[0] is a 32-bit pointer:

```
#pragma pointer_size 64
```

```
main(int argc, char **argv)
  { ASSERT(sizeof(argv[0]) == 4); }
```

Note that using /POINTER_SIZE=LONG=ARGV *(I64 only)* allows `argv` to be a pointer to long pointers.

- A string literal produces a pointer based on the current pointer size when used as an rvalue:

```
#pragma pointer_size 64

ASSERT(sizeof("x" + 0) == 8);

#pragma pointer_size 32

ASSERT(sizeof("x" + 0) == 4);
```

- The & operator yields a pointer based on the current pointer size unless it is applied to pointer dereference, in which case it is the size of the dereferenced pointer type:

```
#pragma pointer_size 32
sizeof(&foo) == 32

#pragma pointer_size 64
sizeof(&foo) == 64

sizeof(&s ->next) == sizeof(s)
```

- The size of this pointer depends on the size in effect at the point of the member's signature definition, not on the use of the pointer.

```
class foo {
  public:
    void f();
    void f2();
};

#pragma required_pointer_size short
void foo::f()
{ sizeof(this)==4 } // this is short

#pragma required_pointer_size long
void foo::f2()
#pragma required_pointer_size short
{ sizeof(this)==8; } // this is long
```

9.3.2. Mixing Pointer Sizes

An application can use both 32-bit and 64-bit addresses. The following semantics apply when mixing pointers:

- Assignments (including arguments) silently promote a 32-bit pointer rvalue to 64 bits if other type rules are met. Promotion means sign extension.
- A warning is issued for an assignment of a 64-bit rvalue to a 32-bit lvalue (without an explicit cast).
- For purposes of type compatibility, a different size pointer is a different type (for example, when matching a prototype to a definition, or other contexts involving redeclaration), however, overloading is not permitted.

- The debugger knows the difference between pointers of different sizes.

9.4. Header File Considerations

Take note of the following general header-file considerations:

- Header files usually define interfaces with types that must match the layout used in library modules.
- Header files often do not bind “top-level” pointer types. Consider, for example:

  ```
  fprintf(FILE *, const char *, ...);
  ```

  A "FILE * fp;" in a declaration in a different area of source code might be a different size.
- All pointer parameters occupy 64 bits in the calling sequence, so a top-level mismatch of this kind is acceptable if the called function does not lose the high bits internally.
- Routines dealing with pointers to pointers (or data structures containing pointers) cannot be enabled to work simply by passing them both 32-bit and 64-bit pointers. You need separate 32-bit and 64-bit variants of the routine.

Be aware that pointer-size controls are not unique in the way they affect header files; other features that affect data layout have similar impact. For example, most header files should be compiled with 32-bit pointers regardless of pointer-size context. Also, most system header files must be compiled with `member_alignment` regardless of user pragmas or qualifiers.

To address this issue more generally, you can use the `pragma environment` directive to save context and set header defaults at the beginning of each header file, and then to restore context at the end. See Section 2.1.1.3 for a description of `pragma environment`.

For header files that have not yet been upgraded to use `#pragma environment`, the /POINTER_SIZE=64 qualifier can be difficult to use effectively. For such header files, the compiler automatically applies user-defined prologue and epilogue files before and after the text of the included header file. See Section 9.5 for more information on prologue/epilogue files.

9.5. Prologue/Epilogue Files

VSI C++ automatically processes user-supplied prologue and epilogue header files. This feature is an aid to using header files that are not 64-bit aware within an application that is built to exploit 64-bit addressing.

9.5.1. Rationale

VSI C++ header files typically contain a section at the top that:

1. Saves the current state of the `member_alignment`, `extern_model`, `extern_prefix`, and `message` pragmas.
2. Sets these pragmas to the default values for the system.

A section at the end of the header file then restores these pragmas to their previously-saved state.

Mixed pointer sizes introduce another kind of state that typically needs to be saved, set, and restored in header files that define fixed 32-bit interfaces to libraries and data structures.

The `#pragma environment` preprocessor directive allows headers to control all compiler states (message suppression, `extern_model`, `member_alignment`, and `pointer_size`) with one directive.

However, for header files that have not yet been upgraded to use `#pragma environment`, the /POINTER_SIZE=64 qualifier can be difficult to use effectively. In this case, the automatic mechanism to include prologue/epilogue files allows you to protect all of the header files within a single directory (or modules within a single text library). You do this by copying two short files into each directory or library that needs it, without having to edit each header file or library module separately.

In time, you should modify header files to either exploit 64-bit addressing (like the C RTL), or to protect themselves with `#pragma environment`. Prologue/epilogue processing can ease this transition.

9.5.2. Using Prologue/Epilogue Files

Prologue/epilogue file are processed in the following way:

1. When the compiler encounters an `#include` preprocessing directive, it determines the location of the file or text library module to be included. It then checks to see if one or both of the two following specially named files or modules exist in the same location as the included file:

   ```
   <double_uscore>DECC_INCLUDE_PROLOGUE.H
   <double_uscore>DECC_INCLUDE_EPILOGUE.H
   ```

   The location is the OpenVMS directory containing the included file or the text library file containing the included module. (In the case of a text library, the .h is stripped off.)

   The directory is the result of using the $PARSE/$SEARCH system services with concealed device name logicals translated. Therefore, if an included file is found through a concealed device logical that hides a search list, the check for prologue/epilogue files is still specific to the individual directories making up the search list.

2. If the prologue and epilogue files do exist in the same location as the included file, then the content of each is read into memory.

3. The text of the prologue file is processed *just before* the text of the file specified by the `#include`.

4. The text of the epilogue file is processed *just after* the text of the file specified by the #include.

5. Subsequent `#includes` that refer to files from the same location use the saved text from any prologue/epilogue file found there.

The prologue/epilogue files are otherwise treated as if they had been included explicitly: `#line` directives are generated for them if /PREPROCESS_ONLY output is produced, and they appear as dependencies if /MMS_DEPENDENCY output is produced.

To take advantage of prologue/epilogue processing for included header files, you need to create two files, `__DECC_INCLUDE_PROLOGUE.H` and `__DECC_INCLUDE_EPILOGUE.H`, in the same directory as the included file.

Suggested content for a prologue file is:

```
<double_uscore>DECC_INCLUDE_PROLOGUE.H:

#ifdef <double_uscore>PRAGMA_ENVIRONMENT
#pragma environment save
```

```
#pragma environment header_defaults
#else
#error "<double_uscore>DECC_INCLUDE_PROLOGUE.H: This compiler does not
 support
pragma environment"
#endif
```

Suggested content for an epilogue file is:

```
<double_uscore>DECC_INCLUDE_EPILOGUE.H:

#ifdef <double_uscore>PRAGMA_ENVIRONMENT
#pragma <double_uscore>environment restore
#else
#error "<double_uscore>DECC_INCLUDE_EPILOGUE.H: This compiler does not
 support
pragma environment"
#endif
```

9.6. Avoiding Problems

Consider the following suggestions to avoid problems related to pointer size:

- Write code to work with either 32-bit or 64-bit pointers.
- Do bit manipulation on `unsigned int` and `unsigned __int64`, and carefully cast pointers to and from them.
- Heed compile-time warnings, using casts only where you are sure that pointers are not truncated.

9.7. Reasons for Not Using Mixed Pointer Sizes

Although VSI C and C++ allow mixing pointer sizes, mixed pointers can cause certain types of error when used incorrectly. Consider the following examples:

- Truncation

```
#pragma pointer_size long
int *y=_malloc64();  // Y is a 64-bit pointer
#pragma pointer_size short
int *x=y;  // X is a 32-bit pointer, which results in truncation.
```

- Misread/miswrite

```
int i,j;
#pragma pointer_size short
int *ptr=&i;
int **pptr=&ptr;
#pragma pointer_size long
int **lptr=pptr;

*lptr = &j;  // miswrite: 8 bytes write, but points to 4 byte ptr.
ptr = *lptr; // misread: 8 bytes read, but points to 4 byte ptr.
```

Furthermore, the following C++ features discourage the use of mixed pointers:

- Objects can allocate memory. Even if an object is in the 32-bit address space, the data contained in that object might not be.

```
#pragma pointer_size long
class myObject {
  char *myData;
  public:
     myObject() { myData = new char[1000]; }
     ~myObject() { delete[] myData; }
     char *getData() { return myData; }
};

#pragma pointer_size short
myObject *ptr = new myObject(); //32-bit pointer to object in 32 bit
 space
char *data = ptr->getData(); //32-bit pointer truncated 64 bit pointer
                                to data in 64 bit space
```

- Virtual functions make it difficult to maintain backward compatibility. Consider the following two implementations of an interface called API. One is written in C, the other in C++. With the C implementation, you can add the new entry with the new pointer size in an upwardly compatible way. In C++, you cannot do so because the functions are virtual. Adding a virtual function to a class breaks backward compatibility. Granted, the C++ interface provides polymorphism that is not available in the C interface, but the availability of this feature is one of the reasons why applications are designed using C++.

```
// C implementation of API
void API_f1(int);
#pragma pointer_size short
void API_f2(int *);
#pragma pointer_size long
void API_f2_64(int*);
void API_f3(int);

// C++ implementatin of API
class BASE {
public:
  virtual void f1(int);
#pragma pointer_size short
  virtual void f2(int *);
#pragma pointer_size long
  virtual void f2_64(int*);
};
class API : public BASE {
public:
  virtual void f3(int);
}
```

- Polymorphism semantics are difficult to define. It is easy to imagine overloading while working with mixed 32/64 bit pointers when the parameter is a simple pointer: the pointers are simply different types. However, if the pointer is embedded in a structure, how are these structures differentiated? Consider the following code fragment:

```
struct FILE {
  char *buffer;
};
```

```
FILE *fopen(const char *,,,);
int fclose(FILE*);
```

It is easy to consider tagging the structure with a flag to indicate whether it is long or short, but it is possible for a structure to have more than one pointer definition. In that case, there could be 2^n different versions of the `struct`. To avoid these issues, the C++ compiler treats 32 and 64 bit pointers as the same type. If you want to treat pointers as different based on size, use template classes:

```
x.cxx
-----
#include <stdio.h>
#include <iostream>

#if !__INITIAL_POINTER_SIZE
#error this program should be compiled with /POINTER_SIZE qualifier
#endif

template <class T>
class short_pointer {
#pragma pointer_size save
#pragma pointer_size short
    T* ptr;
public:
    short_pointer(T* x) { ptr = x; }
    operator T*() { return ptr; }
    size_t get_ptr_size() { return sizeof(ptr); }
#pragma pointer_size restore
};

template <class T>
class long_pointer {
#pragma pointer_size save
#pragma pointer_size long
    T* ptr;
public:
    long_pointer(T* x) { ptr = x; }
    operator T*() { return ptr; }
    size_t get_ptr_size() { return sizeof(ptr); }
#pragma pointer_size restore
};

template<class T>
void func(short_pointer<T> x) { *x = 5; cout << x.get_ptr_size() <<
 endl; }
template<class T>
void func(long_pointer<T> x)  { *x = 5; cout << x.get_ptr_size() <<
 endl; }

int main() {
#pragma pointer_size short
  func(short_pointer<int>((int*)_malloc32(sizeof(int))));
#pragma pointer_size long
  func(long_pointer<int>((int*)malloc(sizeof(int))));
}

$ pipe cxx/pointer=short x.cxx ; cxxl x.obj ; run x.exe
```

```
4
8
$ pipe cxx/pointer=long x.cxx ; cxxl x.obj ; run x.exe
4
8
$
```

Appendix A. Compiler Command Qualifiers

This appendix describes the qualifiers available to the CXX command.

Qualifiers indicate special actions to be performed by the compiler or special input file properties. Compiler qualifiers can apply to either the CXX command or to the specification of the file being compiled. When a qualifier follows the CXX command, it applies to all the files listed. When a qualifier follows the file specification, it applies only to the file immediately preceding it.

Table A.1 summarizes CXX qualifiers. Detailed descriptions follow the table.

Table A.1. CXX Command Qualifiers

Command Qualifiers	Defaults
/[NO]ALTERNATIVE_TOKENS	/See text.
/[NO]ANSI_ALIAS	/ANSI_ALIAS
/ARCHITECTURE=option	/ARCHITECTURE=GENERIC
/ASSUME=(option[,...])	See text.
/[NO]CHECK[=[NO]UNINITIALIZED_ VARIABLES] *(Alpha only)*	/NOCHECK
/[NO]COMMENTS=option	/COMMENTS=SPACE
/[NO]DEBUG[=(option[,...])]	/DEBUG=(TRACEBACK,NOSYMBOLS)
/[NO]DEFINE=(identifier[=definition][,...])	/NODEFINE
/[NO]DEFINE=__FORCE_INSTANTIATIONS *(Alpha only)*	/NODEFINE=__FORCE_ INSTANTIATIONS
/[NO]DEFINE=__[NO_]USE_STD_IOSTREAM	/DEFINE=__NO_USE_STD_IOSTREAM
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]DISTINGUISH_NESTED_ENUMS	/NODISTINGUISH_NESTED_ENUMS
/ENDIAN=option	/ENDIAN=LITTLE
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/EXCEPTIONS	/See text.
/EXPORT_SYMBOLS *(I64 only)*	/See text.
/EXTERN_MODEL=option	/EXTERN_MODEL=RELAXED_REFDEF
/[NO]FIRST_INCLUDE=(file[,...])	/NOFIRST_INCLUDE
/FLOAT=option	/FLOAT=G_FLOAT *(Alpha only)* /FLOAT=IEEE_FLOAT *(I64 only)*
/GRANULARITY=option	/GRANULARITY=QUADWORD
/IEEE_MODE[=option]	/IEEE_MODE=FAST *(Alpha only)* /IEEE_MODE=DENORM_RESULTS *(I64 only)*
/[NO]IMPLICIT_INCLUDE	/IMPLICIT_INCLUDE
/[NO]INCLUDE_DIRECTORY=(pathname[,...])	/NOINCLUDE_DIRECTORY

Command Qualifiers	Defaults
/L_DOUBLE_SIZE=option	/L_DOUBLE_SIZE=128
/LIBRARY	See text.
/[NO]LINE_DIRECTIVES	/LINE_DIRECTIVES
/[NO]LIST[=file-spec]	/NOLIST (interactive mode) /LIST (batch mode)
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]MAIN=POSIX_EXIT	/NOMAIN
/[NO]MEMBER_ALIGNMENT	/MEMBER_ALIGNMENT
/[NO]MMS_DEPENDENCIES=[=(option[,option)]]	/NOMMS_DEPENDENCIES
/MODEL={ANSI | ARM} *(Alpha only)*	/MODEL=ARM
/NAMES=(option1,option2)	/NAMES=(UPPERCASE,TRUNCATED)
/NESTED_INCLUDE_DIRECTORY[=option]	/NESTED_INCLUDE_DIRECTORY= INCLUDE_FILE
/[NO]OBJECT[=file-spec]	/OBJECT=.OBJ
/[NO]OPTIMIZE[=(option[,...])]	/OPTIMIZE
/PENDING_INSTANTIATIONS[=n]	/PENDING_INSTANTIATIONS=64
/[NO]POINTER_SIZE[=option]	/NOPOINTER_SIZE
/[NO]PREFIX_LIBRARY_ENTRIES[=(option[,...])]	See text.
/[NO]PREPROCESS_ONLY[=filename]	/NOPREPROCESS_ONLY
/PSECT_MODEL=[NO]MULTILANGUAGE	/NOMULTILANGUAGE
/[NO]PURE_CNAME	/PURE_CNAME (/STANDARD= STRICT_ANSI) /NOPURE_CNAME (All other modes)
/[NO]QUIET	/NOQUIET
/REENTRANCY=option	/REENTRANCY=TOLERANT
/REPOSITORY=option	/REPOSITORY=[.CXX_REPOSITORY]
/ROUNDING_MODE=option	/ROUNDING_MODE=NEAREST
/[NO]RTTI	/RTTI
/[NO]SHARE_GLOBALS	/NOSHARE_GLOBALS
/SHOW[=(option[,...])]	/SHOW=(HEADER,SOURCE
/STANDARD=(option,...)	/STANDARD=RELAXED
/[NO]TEMPLATE_DEFINE[=(option,...)]	See text.
/[NO]UNDEFINE=(identifier[,...])	/NOUNDEFINE
/[NO]UNSIGNED_CHAR	/NOUNSIGNED_CHAR
/[NO]USING_STD	/NOUSING_STD
/[NO]VERSION	/NOVERSION
/[NO]WARNINGS[=(option[,...])]	/WARNINGS
/[NO]XREF[=file-spec] *(Alpha only)*	/NOXREF

/ALTERNATIVE_TOKENS
/NOALTERNATIVE_TOKENS

Enables use of the following operator keywords and digraphs to generate tokens:

Operator Keyword	**Token**
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

Digraph	**Token**
:>	]
%:	#
%>	}
<%	{
<:	[

The default is /NOALTERNATIVE_TOKENS when compiling with the /STANDARD=ARM, /STANDARD=MS, or /STANDARD=RELAXED option. The default is /ALTERNATIVE_TOKENS when compiling with the /STANDARD=STRICT_ANSI or /STANDARD=GNU option. Specifying /ALTERNATIVE_TOKENS also defines the `__ALTERNATIVE_TOKENS` macro.

/ANSI_ALIAS
/ANSI_ALIAS (D)
/NOANSI_ALIAS

Directs the compiler to assume the ANSI/ISO C aliasing rules, which gives it the freedom to generate better optimized code.

/NOANSI_ALIAS specifies that any pointer can point to any object, regardless of type. /ANSI_ALIAS specifies that pointers to a type T can point to objects of the same type, ignoring type qualifiers such as const, unaligned, or volatile, or whether the object is signed or unsigned. Pointers to a type T can also point to structures, unions, or array members whose type follows the rules above.

The aliasing rules are further explained in the *ANSI C89 Standard*.

/ARCHITECTURE=option
/ARCHITECTURE=GENERIC (D)

Determines the Alpha or Intel processor instruction set to be used by the compiler. The /ARCHITECTURE qualifier uses the same keyword options (keywords) as the /OPTIMIZE=TUNE qualifier.

Where the /OPTIMIZE=TUNE qualifier is primarily used by certain higher-level optimizations for instruction scheduling purposes, the /ARCHITECTURE qualifier determines the type of code instructions generated for the program unit being compiled.

OpenVMS provides an operating system kernel that includes an instruction emulator. This emulator allows new instructions, not implemented on the host processor chip, to execute and produce correct results. Applications using emulated instructions will run correctly, but may incur significant software emulation overhead at runtime.

All Alpha processors implement a core set of instructions. Certain Alpha processor versions include additional instruction extensions.

Select one of the /ARCHITECTURE qualifier options shown in the following table.

Option	**Usage**
GENERIC	Generates code that is appropriate for all processor generations. This is the default.
HOST	Generates code for the processor generation in use on the system being used for compilation. Running programs compiled with this option on other implementations of the Alpha architecture may encounter instruction-emulation overhead.
ITANIUM2 *(I64 only)*	Generates code for the Intel Itanium 2 processor family. For use on I64 systems only.
EV4 *(Alpha only)*	Generates code for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture. Programs compiled with the EV4 option run without instruction-emulation overhead on all Alpha processors.
EV5 *(Alpha only)*	Generates code for some 21164 chip implementations of the Alpha architecture that use only the base set of Alpha instructions (no extensions). Programs compiled with the EV5 option will without instruction-emulation overhead on all Alpha processors.
EV56 *(Alpha only)*	Generates code for some 21164 chip implementations that use the byte and word-manipulation instruction extensions of the Alpha architecture. Running programs compiled with the EV56 option might incur emulation overhead on EV4 and EV5 processors, but will still run correctly on OpenVMS Version 7.1 (or higher) systems.
PCA56 *(Alpha only)*	Generates code for the 21164PC chip implementation that uses the byte- and word-manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture.

Option	Usage
	Programs compiled with the PCA56 option might incur emulation overhead on EV4, EV5, and EV56 processors, but still run correctly on OpenVMS Version 7.1 (or higher) systems.
EV6 *(Alpha only)*	Generates code for the 21264 implementation of the Alpha architecture.
EV68 *(Alpha only)*	Generates code for the 21264/EV68 implementation of the Alpha architecture.
EV7 *(Alpha only)*	Generates code for the EV7 implementation of the Alpha architecture.

See also /OPTIMIZE=TUNE, which is a more typical option. Note that if /ARCHITECTURE is explicitly specified and /OPTIMIZE=TUNE is not, the tuning processor defaults to the architecture processor; for example, /ARCHITECTURE=EV6 implies /OPTIMIZE=TUNE=EV6.

/ASSUME
/ASSUME=(option[,...])

Controls compiler assumptions. You may select the following options:

Option	Usage
[NO]WRITABLE_STRING_ LITERALS	Stores string constants in a writable psect. Otherwise, such constants are placed in a nonwriteable psect. The default is NOWRITABLE_STRING_LITERALS.
[NO]ACCURACY_SENSITIVE	Specifies whether certain code transformations that affect floating-point operations are allowed. These changes may or may not affect the accuracy of the program's results. If you specify NOACCURACY_SENSITIVE, the compiler is free to reorder floating-point operations based on algebraic identities (inverses, associativity, and distribution). This allows the compiler to move divide operations outside of loops, which improves performance. The default, ACCURACY_SENSITIVE, directs the compiler to use only certain scalar rules for calculations. This setting can prevent some optimization.
[NO]ALIGNED_OBJECTS	Controls an optimization for dereferencing pointers. Dereferencing a pointer to a longword- or quadword-aligned object is more efficient than dereferencing a pointer to a byte- or word-aligned object. Therefore, the compiler can generate more optimized code if it makes the assumption that a pointer object of an aligned pointer type does point to an aligned object. Because the compiler determines the alignment of the dereferenced object from the type of the pointer, and the program is allowed to compute a pointer that references an unaligned object (even though the pointer type indicates that it references an aligned object), the compiler must assume that the dereferenced object's alignment matches or exceeds the alignment indicated by the pointer type.

Option	Usage
	The default, /ASSUME=ALIGNED_OBJECTS, allows the compiler to make such an assumption. With this assumption made, the compiler can generate more efficient code for pointer dereferences of aligned pointer types. To prevent the compiler from assuming the pointer type's alignment for objects to which it points, use the /ASSUME=NOALIGNED_OBJECTS qualifier. This option causes the compiler to generate longer code sequences to perform indirect load and store operations to avoid hardware alignment faults for arbitrarily aligned addresses. Although /ASSUME=NOALIGNED_OBJECTS might generate less efficient code than the default /ASSUME=ALIGNED_OBJECTS option, by avoiding hardware alignment faults, it speeds the execution of programs that reference unaligned data.
[NO]GLOBAL_ARRAY_NEW	Controls whether calls to global array new and delete are generated as specified by ANSI. Pre-ANSI global array new generated calls to operator new(). According to ANSI, use of global array new generates calls to operator new()[]. The GLOBAL_ARRAY_NEW option also defines the macro __GLOBAL_ARRAY_NEW. GLOBAL_ARRAY_NEW generates calls to operator new() [] for global array new expressions such as new int[4]; this is the default when compiling /STANDARD=RELAXED, /STANDARD=STRICT_ANSI, /STANDARD=GNU, and /STANDARD=MS. NOGLOBAL_ARRAY_NEW generates calls to operator new() for global array new expressions such as new int[4]; and preserves compatibility with Version 5.n; this is the default when compiling /STANDARD=ARM.
[NO]HEADER_TYPE_DEFAULT	Controls whether the compiler appends a file extension to a file name. The default is /ASSUME=NOHEADER_TYPE_DEFAULT. To prevent the compiler from appending a file extension to files (such as STL header files that must not have file extensions) use the /ASSUME=NOHEADER_TYPE_DEFAULT qualifier.
[NO]MATH_ERRNO	Controls whether intrinsic code is generated for math functions that set the errno variable. The default is /ASSUME=MATH_ERRNO, which does not allow intrinsic code for such math functions to be generated, even if /OPTIMIZE=INTRINSICS is in effect. Their prototypes and call formats, however, are still checked.
[NO]POINTERS_TO_GLOBALS	Controls whether the compiler can safely assume that global variables have not had their addresses taken in code that is not visible to the current compilation.

Option	Usage
	The default is /ASSUME=POINTERS_TO_GLOBALS, which directs the compiler to assume that global variables have had their addresses taken in separately compiled modules and that, in general, any pointer dereference could be accessing the same memory as any global variable. This is often a significant barrier to optimization. While the /ANSI_ALIAS option allows some resolution based on data type, /ASSUME=POINTERS_TO_GLOBALS provides significant additional resolution and improved optimization in many cases. The /ASSUME=NOPOINTERS_TO_GLOBALS option tells the compiler that any global variable accessed through a pointer in the compilation must have had its address taken within that compilation. The compiler can see any code that takes the address of an extern variable. If it does not see the address of the variable being taken, the compiler can assume that no pointer points to the variable. Note that /ASSUME=NOPOINTERS_TO_GLOBALS does not tell the compiler that the compilation never uses pointers to access global variables. Also note that on I64 systems, the NOPOINTERS_TO_GLOBALS option is ignored and should not cause any behavior changes.
[NO]STDNEW	Controls whether calls are generated to the ANSI or pre-ANSI implementation of the operator new(). On memory allocation failure, the ANSI implementation throws `std::bad_alloc`, while the pre-ANSI implementation returns 0. /ASSUME=STDNEW generates calls to the ANSI new() implementation; this is the default when compiling with /STANDARD=RELAXED, /STANDARD=STRICT_ANSI, and /STANDARD=GNU. /ASSUME=NOSTDNEW generates calls to the pre-ANSI new() implementation; this is the default when compiling with /STANDARD=ARM and /STANDARD=MS.
[NO]TRUSTED_SHORT_ALIGNMENT	Allows the compiler additional assumptions about the alignment of short types that, although thought to be naturally aligned, might cross a quadword boundary. The TRUSTED_SHORT_ALIGNMENT option indicates that the compiler should assume any datatype accessed through a pointer is naturally aligned. This generates the fastest code, but can silently generate the wrong results when an unaligned short object crosses a quadword boundary.

Option	Usage
	Note that on I64 systems, the TRUSTED_SHORT_ALIGNMENT option is ignored and should not cause any behavior changes. The NOTRUSTED_SHORT_ALIGNMENT tells the compiler that short objects might not be naturally aligned. The compiler generates slightly larger (and slower) code that gives the correct result, regardless of the actual alignment of the data. This is the default. Note that the NOTRUSTED_SHORT_ALIGNMENT option does not override the __unaligned type qualifier or the /ASSUME=NOALIGNED_OBJECTS option.
[NO]WHOLE_PROGRAM	Tells the compiler that except for well-behaved library routines, the whole program consists only of the single object module being produced by this compilation. The optimizations enabled by /ASSUME=WHOLE_PROGRAM include all those enabled by /ASSUME=NOPOINTERS_TO_GLOBALS and possibly other optimizations. Note that on I64 systems, the WHOLE_PROGRAM option is ignored and should not cause any behavior changes. The default is /ASSUME=NOWHOLE_PROGRAM.

/CHECK
/CHECK[=([NO]UNINITIALIZED _VARIABLES)] ***(Alpha only)***
/NOCHECK (D)

Use this qualifier as a debugging aid.

Use /CHECK=UNINITIALIZED_VARIABLES to initialize all automatic variables to the value 0x7ff580057ff58005. This value is a floating NaN and, if used, causes a floating-point trap. If used as a pointer, this value is likely to cause an ACCVIO.

Note that on I64 systems, /CHECK=UNINITIALIZED_VARIABLES emits a warning and is ignored.

/COMMENTS
/COMMENTS[=option]
/COMMENTS=SPACE (D)
/NOCOMMENTS

Specifies whether comments appear in preprocessor output files. If comments do not appear, this qualifier specifies what replaces them. The options are:

Option	Usage
AS_IS	Specifies that the comment appear in the output file. This is the default if you use the /COMMENTS qualifier without specifying an option.

Option	Usage
SPACE	Specifies that a single space replaces the comment in the output file. This is the default if you do not specify the /COMMENTS qualifier at all.

Specifying /NOCOMMENTS tells the preprocessor that nothing replaces the comment in the output file. This may result in inadvertent token pasting.

The preprocessor may replace a comment at the end of a line or replace a line by itself with nothing, even if you specify /COMMENTS=SPACE. Specifying /COMMENTS=SPACE cannot change the meaning of the program.

/DEBUG
/DEBUG[=(option[,...])]
/DEBUG=(TRACEBACK, NOSYMBOLS) (D)
/NODEBUG

Requests that information be included in the object module for use with the OpenVMS Debugger. You can select the following options:

Option	Usage
ALL	Includes all possible debugging information. /DEBUG=ALL is equivalent to /DEBUG=(TRACEBACK,SYMBOLS), which on I64 systems is equivalent to /DEBUG=(TRACEBACK, SYMBOLS=NOBRIEF).
NONE	Excludes all debugging information. This option is equivalent to specifying /NODEBUG, which is equivalent to /DEBUG=(NOTRACEBACK,NOSYMBOLS).
NOSYMBOLS	Turns off symbol generation
SYMBOLS	Generates symbol-table records. On I64 systems, /DEBUG=SYMBOLS is equivalent to /DEBUG=SYMBOLS=BRIEF. On Alpha systems, /DEBUG=SYMBOLS is effectively equivalent to /DEBUG=NOBRIEF.
SYMBOLS=BRIEF *(I64 only)*	Generates debug information with unreferenced labels and types pruned out to produce smaller object sizes. On Alpha systems, BRIEF is ignored.
SYMBOLS=NOBRIEF *(I64 only)*	Generates complete debug information. On Alpha systems, the NOBRIEF keyword is ignored, but you still get complete debug information.
NOTRACEBACK	Excludes traceback records. This option is equivalent to specifying /NODEBUG and is used to avoid generating extraneous information from thoroughly debugged program modules.
TRACEBACK	Includes only traceback records. This option is the default if you do not specify the /DEBUG qualifier on the command line.

On Alpha systems /DEBUG is equivalent to /DEBUG=(TRACEBACK,SYMBOLS).

On I64 systems /DEBUG is equivalent to /DEBUG=(TRACEBACK,SYMBOLS), which is equivalent to /DEBUG=(TRACEBACK,SYMBOLS=BRIEF).

/DEFINE
/DEFINE=(identifier[=definition][,...])
/NODEFINE (D)

Performs the same function as the `#define` preprocessor directive. That is, /DEFINE defines a token string or macro to be substituted for every occurrence of a given identifier in the program.

DCL converts all input to uppercase unless it is enclosed in quotation marks.

The simplest form of a /DEFINE definition is as follows: `/DEFINE=true`

This results in a definition like the one that would result from the following definition: `#define TRUE 1`

When more than one /DEFINE is present on the CXX command line or in a single compilation unit, only the last /DEFINE is used.

When both /DEFINE and /UNDEFINE are present on a command line, /DEFINE is evaluated before /UNDEFINE

/DEFINE=__FORCE_INSTANTIATIONS *(Alpha only)*
/NODEFINE=__FORCE_INSTANTIATIONS (D)

Forces the standard library template pre-instantiations to be created in the user's repository. Normally these instantiations are suppressed because the library already contains them.

On I64 systems, defining __FORCE_INSTANTIATIONS has no effect.

/DEFINE=__[NO_]USE_STD_IOSTREAM
/DEFINE=__NO_USE_STD_IOSTREAM (D)

Use or do not use the standard iostreams. Specifying /DEFINE=__USE_STD_IOSTREAM forces the inclusion of the ANSI standard version of the iostream header file. This is the default in STRICT_ANSI mode. Otherwise, the pre-standard AT&T-compatible version of iostreams is used.

/DIAGNOSTICS
/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)

Creates a file containing compiler diagnostic messages. The default file extension for a diagnostics file is .DIA. The diagnostics file is used with the VSI Language-Sensitive Editor (LSE). To display a diagnostics file, enter the command REVIEW/FILE=file-spec while in LSE.

/DISTINGUISH_NESTED_ENUMS
/NODISTINGUISH_NESTED_ENUMS (D)

Causes the compiler, when forming a mangled name, to include the name of any enclosing classes within which an enum is nested, thereby preventing different functions from receiving the same mangled name.

This qualifier has no meaning on I64 systems because it modifies the behavior of programs compiled with /MODEL=ARM, and that model is not supported on I64 systems.

/ENDIAN
/ENDIAN={BIG | LITTLE}
/ENDIAN=LITTLE (D)

Controls whether big or little endian ordering of bytes is carried out in character constants.

/ERROR_LIMIT
/ERROR_LIMIT[=number]
/ERROR_LIMIT=30 (D)
/NOERROR_LIMIT

Limits the number of error-level diagnostic messages that are acceptable during program compilation. Compilation terminates when the limit (number) is exceeded. /NOERROR_LIMIT specifies that there is no limit on error messages.

The default is /ERROR_LIMIT=30, which specifies that compilation terminates after issuing 30 error messages.

/EXCEPTIONS
/EXCEPTIONS=CLEANUP (D)
/EXCEPTIONS=NOCLEANUP *(Alpha only)*
/EXCEPTIONS=EXPLICIT (D)
/EXCEPTIONS=IMPLICIT *(Alpha only)*
/NOEXCEPTIONS

Controls whether support for C++ exceptions is enabled or disabled. C++ exceptions are enabled by default (equivalent to /EXCEPTIONS=CLEANUP). When C++ exceptions are enabled, the compiler generates code for throw expressions, try blocks, and catch statements. The compiler also generates special code for main programs so that the terminate() routine is called for unhandled exceptions. You can select from the following options:

CLEANUP	Generate cleanup code for automatic objects. When an exception is handled at run-time and control passes from a throw-point to a handler, call the destructors for all automatic objects that were constructed because the try-block containing the handler was entered.
NOCLEANUP *(Alpha only)*	Do not generate cleanup code. Using this option can reduce the size of your executable when you want to throw and handle exceptions and cleanup of automatic objects during exception processing is not important for your application. The NOCLEANUP option is ignored on I64 systems.
EXPLICIT	Tells the compiler to assume the standard C++ rules about exceptions. Catch clauses can catch only those exceptions resulting from the evaluation of a throw expression within the body of the catch clause's try block or from within a procedure called from within the catch clause's try block.
IMPLICIT *(Alpha only)*	On Alpha systems, tells the compiler that an exception might be thrown at any time the program is executing code within the body of the try block. These exceptions might be the result of a throw expression, hardware errors, or software errors (such as dereferencing an invalid pointer). Specifying /EXCEPTIONS=IMPLICIT seriously interferes with the compiler's ability to optimize code. When the compiler optimizes a function, it must ensure that the values of all variables after an exception is caught remain the same as they were at the point where the exception was throw. The optimizer is therefore limited in its ability to rearrange stores and expressions that might cause an exception to be thrown. With /EXCEPTIONS=EXPLICIT, this is not a serious restriction, because the compiler cannot rearrange stores and expressions around the code that

	explicitly raises an exception. In implicit exception mode, however, almost any code has the potential to cause an exception to be thrown, thereby dramatically reducing the optimizer's ability to rearrange code. Also, if the compiler can determine that no throw expressions occur within a try block, it can eliminate the exception handling overhead the try block introduces, including all code within the catch clauses associated with the try block. Because no exceptions can occur during the execution of the code within the try block, no code within the catch clauses can ever be executed. The compiler cannot do this with /EXCEPTIONS=IMPLICIT. Use /EXCEPTIONS=IMPLICIT if your program converts signals to C++ exceptions by calling cxxl$set_condition(cxx_exception). Failure to do so may result in your code not catching the exceptions produced by signals.

For example, consider the following routine:

```
void f(int *p) {
    try {
        *p = 2;
    } catch (...) {
        ...
    }
}
```

Failure to compile the routine with /EXCEPTIONS=IMPLICIT may result in a failure to catch the exception generated by the SIGBUS signal that occurs if p is 0. This is because the compiler sees that there are no throws nor procedure calls within f and therefore optimizes away the try block leaving:

```
void f(int *p) {
    *p = 2;
}
```

Except for those OpenVMS conditions that result in the delivery of signals, if you raise a condition explicitly using a mechanism such as LIB$SIGNAL, you may use /EXCEPTIONS=EXPLICIT.

The /NOEXCEPTIONS qualifier disables C++ exceptions as follows:

1. The compiler issues errors for throw expressions, try blocks, and catch statements, but might generate code for these constructs.
2. On Alpha systems, the compiler does not generate cleanup code for automatic objects.
3. The compiler does not generate special code for main programs so that the terminate() function is called for unhandled exceptions.

The /EXCEPTIONS qualifier defines the macro __EXCEPTIONS.

/EXPORT_SYMBOLS=(OPTIONS_FILE=<name> [,EXCLUDE=<list of images>] [,export_option] [,NOTEMPLATES]) *(I64 only)*

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 can be 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, the VSI C++ compiler provides the /EXPORT_SYMBOLS qualifier and `__declspec(dllexport)` declaration modifier.

The default file extension for the OPTIONS_FILE <*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.

Note

- 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:

Export_option Value	**Usage**
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.
EXPLICIT	Place only those symbols marked with the `__declspec(dllexport)` declaration modifier into the options file.
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 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.

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.

/EXTERN_MODEL
/EXTERN_MODEL=option
/EXTERN_MODEL=RELAXED_REFDEF (D)

In conjunction with the /SHARE_GLOBALS qualifier, controls the initial extern model of the compiler. Conceptually, the compiler behaves as if the first line of the program being compiled was a `#pragma extern_model` directive with the model and psect name, if any, specified by the /EXTERN_MODEL qualifier and with the SHR or NOSHR keyword specified by the /SHARE_GLOBALS qualifier.

For example, assume the command line contains the following qualifier:

```
/EXTERN_MODEL=STRICT_REFDEF="MYDATA"/NOSHARE
```

The compiler acts as if the program began with the following line:

```
#pragma extern_model strict_refdef "MYDATA" noshr
```

For more information on the various models, see Section 2.1.1.4.

The /EXTERN_MODEL qualifier takes the following options, which have the same meaning as for the `#pragma extern_model` directive:

COMMON_BLOCK
RELAXED_REFDEF
STRICT_REFDEF=["NAME"]
GLOBALVALUE

The default is RELAXED_REFDEF.

Use of an /EXTERN_MODEL value other than RELAXED_REFDEF should be limited to compilations that either declare only POD (Plain Old Data) objects, or that carefully use the `extern_model` (and/or `environment`) `#pragma` directives to ensure that declarations of non-POD objects appear only in source that is subject to the default `extern_model` of `relaxed_refdef`.

/FIRST_INCLUDE
/FIRST_INCLUDE=(file[,...])
/NOFIRST_INCLUDE (D)

Includes the specified files before any source files. This qualifier corresponds to the 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 were:

```
#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.

/FLOAT
/FLOAT=option
/FLOAT=G_FLOAT *(Alpha only)* (D)
/FLOAT=IEEE_FLOAT *(I64 only)* (D)

Controls the format of floating-point variables. The options are:

Option	Usage
D_FLOAT	`double` variables are represented in VAX D_floating format. `float` variables are represented in VAX F_floating format. The __D_FLOAT macro is predefined.
G_FLOAT	`double` variables are represented in VAX G_floating format. `float` variables are represented in VAX F_floating format. The __G_FLOAT macro is predefined.
IEEE_FLOAT	`float` and `double` variables are represented in IEEE floating-point format (S_float and T_float, respectively). The __IEEE_FLOAT macro is predefined. Use the /IEEE_MODE qualifier for controlling the handling of IEEE exceptional values.

On Alpha systems, the default is /FLOAT=G_FLOAT.

On I64 systems, the default is /FLOAT=IEEE_FLOAT.

See Section 4.1.6 for additional information on floating-point representation on I64 and Alpha systems.

/GRANULARITY
/GRANULARITY=option
/GRANULARITY=QUADWORD (D)

Controls the size of shared data in memory that can be safely accessed from different threads. The possible size values are BYTE, LONGWORD, and QUADWORD.

Specifying BYTE allows single bytes to be accessed from different threads sharing data in memory without corrupting surrounding bytes. This option will slow runtime performance.

Specifying LONGWORD allows naturally aligned 4-byte longwords to be accessed safely from different threads sharing data in memory. Accessing data items of 3 bytes or less, or unaligned data, may result in data items written from multiple threads being inconsistently updated.

Specifying QUADWORD allows naturally aligned 8-byte quadwords to be accessed safely from different threads sharing data in memory. Accessing data items of 7 bytes or less, or unaligned data, might result in data items written from multiple threads being inconsistently updated. This is the default.

/IEEE_MODE
/IEEE_MODE=option
/IEEE_MODE=FAST (D) ***(Alpha only)***
/IEEE_MODE=DENORM_RESULTS (D) ***(I64 only)***

Selects the IEEE floating-point mode to be used if the /FLOAT=IEEE_FLOAT qualifier is specified. The options are:

Option	Usage
FAST	During program execution, only finite values (no infinities, NaNs, or denorms) are created. Underflows and denormal values are flushed to zero. Exceptional conditions, such as floating-point overflow, divide-by-zero, or use of an IEEE exceptional operand are fatal.
UNDERFLOW_TO_ZERO	Generate infinities and NaNs. Flush denormalized results and underflow to zero without exceptions.
DENORM_RESULTS	Same as the UNDERFLOW_TO_ZERO option, except that denorms are generated.
INEXACT	Same as the DENORM_RESULTS option, except that inexact values are trapped. This is the slowest mode, and is not appropriate for any sort of general-purpose computations.

On Alpha systems, the default is /IEEE_MODE=FAST.

On I64 systems, the default is /IEEE_MODE=DENORM_RESULTS.

The INFINITY and NAN macros defined in `<math.h>` are available to programs compiled with /FLOAT=IEEE and /IEEE_MODE={*anything other than FAST*}.

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 I64 systems, 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 this 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.

Also see the /FLOAT qualifier.

/IMPLICIT_INCLUDE
/IMPLICIT_INCLUDE (D)
/NOIMPLICIT_INCLUDE

/IMPLICIT_INCLUDE enables inclusion of source files as a method of finding definitions of template entities. By default it is enabled for normal compilation, and disabled for preprocessing only. The search rules for finding template definition files is the same as for include files.

/NOIMPLICIT_INCLUDE disables inclusion of source files as a method of finding definitions of template entities. You might want to use this option in conjunction with the /STANDARD=MS command line option, to match more closely the behavior on Microsoft C++.

/INCLUDE_DIRECTORY
/INCLUDE_DIRECTORY=(place[,...])
/NOINCLUDE_DIRECTORY (D)

Provides an additional level of search for user-defined include files. Each *pathname* argument can be either a logical name or a legal UNIX style directory in a quoted string. The default is /NOINCLUDE_DIRECTORY.

The /INCLUDE_DIRECTORY qualifier provides functionality similar to the `-I` option of the `cxx` command on UNIX systems. This qualifier allows you to specify additional locations to search for files to include. Putting an empty string in the specification prevents the compiler from searching any of the locations it normally searches but directs it to search *only* in locations you identify explicitly on the command line with the /INCLUDE_DIRECTORY And /LIBRARY qualifiers (or by way of the specification of the primary source file, depending on the /NESTED_INCLUDE_DIRECTORY qualifier).

The basic order for searching depends on the form of the header name (after macro expansion), with additional aspects controlled by other command line qualifiers as well as the presence or absence of logical name definitions. The valid possibilities for names are as follows:

- Enclosed in quotes. For example: "stdio.h"
- Enclosed in angle brackets. For example: `<stdio.h>`

Unless otherwise defined, searching a location means that the compiler uses the string specifying the location as the default file specification in a call to an RMS system service (that is, a $SEARCH/$PARSE) with a primary file specification consisting of the name in the `#include` (without enclosing delimiters). The search terminates successfully as soon as a file can be opened for reading.

Specifying a null string in the /INCLUDE qualifier causes the compiler to do a non-standard search. This search path is as follows:

1. The current directory (quoted form only)
2. Any directories specified in the /INCLUDE qualifier
3. The directory of the primary input file
4. Text libraries specified on the command line using /LIBRARY

For standard searches, the search order is as follows:

1. Search the current directory (directory of the source being processed). If angle-bracket form, search only if no directories are specified with /INCLUDE_DIRECTORY.

2. Search the locations specified in the /INCLUDE_DIRECTORY qualifier (if any).

3. If CXX$SYSTEM_INCLUDE is defined as a logical name, search CXX$SYSTEM_INCLUDE:.HXX or just CXX$SYSTEM_INCLUDE:., depending on the qualifier /ASSUME=NOHEADER_TYPE_DEFAULT. If nothing is found, go to step 6.

4. If CXX$LIBRARY_INCLUDE is defined as a logical name, CXX$LIBRARY_INCLUDE:.HXX or CXX$LIBRARY_INCLUDE:., depending on the qualifier /ASSUME=NOHEADER_TYPE_DEFAULT. If nothing is found, go to step 6.

5. If /ASSUME=HEADER_TYPE_DEFAULT is not specified, search the default list of locations for plain-text copies of compiler header files as follows:

   SYS$COMMON:[CXX$LIB.INCLUDE.CXXL$ANSI_DEF]
   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF_HXX].HXX
   SYS$COMMON:[DECC$LIB.INCLUDE.DECC$RTLDEF].H
   SYS$COMMON:[DECC$LIB.INCLUDE.SYS$STARLET_C].H

   If /ASSUME=HEADER_TYPE_DEFAULT is specified, search the default list of locations for plain-text copies of compiler header files as follows:

   SYS$COMMON:[CXX$LIB.INCLUDE.DECC$RTLDEF_HXX].HXX
   SYS$COMMON:[DECC$LIB.INCLUDE.DECC$RTLDEF].H
   SYS$COMMON:[DECC$LIB.INCLUDE.SYS$STARLET_C].H
   SYS$COMMON:[CXX$LIB.INCLUDE.CXXL$ANSI_DEF]

6. Search the directory of the primary input file.

7. If quoted form, and CXX$USER_INCLUDE is defined as a logical name, search CXX$USER_INCLUDE:.HXX or CXX$USER_INCLUDE:., depending on the /ASSUME=NOHEADER_TYPE_DEFAULT qualifier.

8. Search the text libraries. Extract the simple file name and file type from the `#include` specification, and use them to determine a module name for each text library. There are three forms of module names used by the compiler:

   a. type stripped:

      The file type will be removed from the include file specification to form a library module name. Examples:

#include "foo.h"	Module name "FOO"
#include "foo"	Module name "FOO"
#include "foo"	Module name "FOO"

   b. type required:

      The file type must be a part of the file name. Examples:

#include "foo.h"	Module name "FOO.H"
#include "foo"	Module name "FOO."
#include "foo"	Module name "FOO."

   c. type optional:

First an attempt is made to find a module with the type included in the module name. If this is unsuccessful, an attempt is made with the type stripped from the module name. If this is unsuccessful, the search moves on to the next library.

If /ASSUME=HEADER_TYPE_DEFAULT is specified, the following text libraries are searched in this order:

Libraries specified on the command line with the /LIBRARY qualifier (all files, type stripped)
CXX$TEXT_LIBRARY (all files, type stripped)
DECC$RTLDEF (H files and unspecified files, type stripped)
SYS$STARLET_C (all files, type stripped)
CXXL$ANSI_DEF (unspecified files, type stripped)

Otherwise, these text libraries are searched in this order:

Libraries specified on the command line with the /LIBRARY qualifier (all files, type optional)
CXX$TEXT_LIBRARY (all files, type optional)
CXXL$ANSI_DEF (all files, type required)
DECC$RTLDEF (H files and unspecified files, type stripped)
SYS$STARLET_C (all files, type stripped)

Two text library search examples (stop when something is found):

Example 1

```
#include "foo"
```

a. For each library specified via the /LIBRARY qualifier:

   - Look for "FOO."
   - Look for "FOO"

b. Look for "FOO." in CXX$TEXT_LIBRARY

c. Look for "FOO" in CXX$TEXT_LIBRARY

d. Look for "FOO." in CXXL$ANSI_DEF (Do not look for "FOO" because the type is required as part of the module name)

e. Look for "FOO" in DECC$RTLDEF (not "FOO." because the type must not be part of the module name)

f. Look for "FOO" in SYS$STARLET_C (not "FOO." because the type must not be part of the module name)

Example 2

```
#include "foo.h"
```

a. For each library specified via the /LIBRARY qualifier:

   - Look for "FOO.H"
   - Look for "FOO"

b. Look for "FOO.H" in CXX$TEXT_LIBRARY

c. Look for "FOO" in CXX$TEXT_LIBRARY

d. Look for "FOO.H" in CXXL$ANSI_DEF (Do not look for "FOO" because the type is required as part of the module name)

e. Look for "FOO" in DECC$RTLDEF (not "FOO.H" because the type must not be part of the module name)

f. Look for "FOO" in SYS$STARLET_C (not "FOO.H" because the type must not be part of the module name)

g. If neither CXX$LIBRARY_INCLUDE nor CXX$SYSTEM_INCLUDE is defined as a logical name, then search SYS$LIBRARY:.HXX.

/L_DOUBLE_SIZE
/L_DOUBLE_SIZE=option
/L_DOUBLE_SIZE=128 (D)

Determines how the compiler interprets the long double type. The qualifier options are 64 and 128.

Specifying /L_DOUBLE_SIZE=64 treats all long double references as G_FLOAT, D_FLOAT, or T_FLOAT, depending on the value of the /FLOAT qualifier. Specifying /L_DOUBLE_SIZE=64 also defines the macro __X_FLOAT=0.

Note: The /L_DOUBLE_SIZE=64 option is not available on I64 systems. If it is specified, the compiler issues a warning message and uses /L_DOUBLE_SIZE=128.

Specifying /L_DOUBLE_SIZE=128 treats all long double references as X_FLOAT.
The /L_DOUBLE_SIZE=128 option also defines the macro __X_FLOAT=1. This is the default.

/LIBRARY

Indicates that the associated input file is a text library containing source text modules specified in `#include` directives. The compiler searches the specified library for all `#include` module names that are not enclosed in angle brackets or quotation marks. The name of the library must be concatenated with the file specification using a plus sign. For example: CXX DATAB/LIBRARY +APPLICATION

/LINE_DIRECTIVES
/LINE_DIRECTIVES (D)
/NOLINE_DIRECTIVES

Controls whether `#line` directives appear in preprocessed output files.

/LIST
/LIST[=file-spec] (Batch default)
/NOLIST (Interactive default)

Controls whether a listing file is produced. The default output file extension is .LIS

/MACHINE_CODE
/NOMACHINE_CODE (D)

Controls whether the listing produced by the compiler includes the machine-language code generated during the compilation. If you use this qualifier you also need to use the /LIST qualifier. On Alpha systems, machine-language code is not added to the listing file when object-file generation is disabled (using the /NOOBJECT qualifier).

/MAIN=POSIX_EXIT
/MAIN=POSIX_EXIT
/NOMAIN (D)

Directs the compiler to call `__posix_exit` instead of `exit` when returning from `main`.

/MEMBER_ALIGNMENT
/MEMBER_ALIGNMENT (D)
/NOMEMBER_ALIGNMENT

Directs the compiler to naturally align data structure members. This means that data structure members are aligned on the next boundary appropriate to the type of the member, rather than on the next byte. For instance, a long variable member is aligned on the next longword boundary; a short variable member is aligned on the next word boundary.

Any use of the `#pragma member_alignment` or `#pragma nomember_alignment` directives within the source code overrides the setting established by this qualifier.
Specifying /NOMEMBER_ALIGNMENT causes data structure members to be byte-aligned (with the exception of bit-field members).

/MMS_DEPENDENCIES
/MMS_DEPENDENCIES[=(option[,option)]]
/NOMMS_DEPENDENCIES (D)

Instructs the compiler to produce a dependency file. The format of the dependency file is as follows:

```
object_file_name:<tab><source file name>
object_file_name:<tab><full path to first include file>
object_file_name:<tab><full path to second include file>
```

You can specify none, one, or both of the following qualifier options:

FILE[=filespec]	Specifies where to save the dependency file. The default file extension for a dependency file is .mms. Other than using a different default extension, this qualifier uses the same procedure that /OBJECT and /LIST use for determining the name of the output file.
SYSTEM_INCLUDE_FILES	Specifies whether to include dependency information about system include files (that is, those included with `#include <filename>`). The default is to include dependency information about system include files.

/MODEL ***(Alpha only)***
/MODEL={ANSI | ARM}
/MODEL=ARM (D)

On Alpha systems, determines the layout of C++ classes, name mangling, and exception handling.

On I64 systems, the default (and only) object model & demangling scheme used is the I64 Application Binary Interface (ABI). The compiler accepts the /MODEL qualifier, but it has no effect.

On Alpha systems, /MODEL=ARM is the default and generates objects that are link compatible with all releases prior to VSI C++ version 6.3, and with all objects compiled with the /MODEL=ARM qualifier in releases of VSI C++ Version 6.3 or later. Specifying this option defines the macro __MODEL_ARM.

The /MODEL=ANSI qualifier supports the complete ISO/ANSI C++ specification, including distinct name mangling for templates. The ANSI model also reduces the size of C++ non-POD class objects. Note that this option generates objects that are not compatible with all prior and future releases of VSI C++, or with objects compiled using the /MODEL=ARM qualifier.

If you specify the /MODEL=ANSI qualifier, you must recompile and relink (using CXXLINK/MODEL=ANSI) your entire application, including libraries. Specifying this option defines the macro __MODEL_ANSI.

/NAMES
/NAMES=(option1,option2)
/NAMES=(UPPERCASE,TRUNCATED) (D)

Option1 converts all definitions and references of external symbols and psects to the case specified. Option1 values are:

Option	Usage
UPPERCASE	Converts to uppercase.
AS_IS	Leaves the case as specified in the source.

Option2 controls whether or not external names greater than 31 characters get truncated or shortened. Option2 values are:

Option	Usage
/NAMES=TRUNCATED (default)	Truncates long external names to the first 31 characters.
/NAMES=SHORTENED	Shortens long external names. A shortened name consists of the first 23 characters of the name followed by a 7-character Cyclic Redundancy Check (CRC) computed by looking at the full name, and then a "$".

The default is /NAMES=(UPPERCASE,TRUNCATED).

Note

The I64 C++ 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 in 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.

/NESTED_INCLUDE_DIRECTORY
/NESTED_INCLUDE_DIRECTORY[=option]
/NESTED_INCLUDE_DIRECTORY=INCLUDE_FILE (D)

Controls the first step in the search algorithm the compiler uses when looking for files included using the quoted form of the `#include` preprocessing directive: `#include` "file-spec"
The /NESTED_INCLUDE_DIRECTORY qualifier has the following options:

Option	Usage
PRIMARY_FILE	Directs the compiler to search the default file type for headers using the context of the primary source file. This means that only the file type (".H" or ".") is used for the default file-spec but, in addition, the chain of "related file-specs" used to maintain the sticky defaults for processing the next top-level source file is applied when searching for the include file.
INCLUDE_FILE	Directs the compiler to search the directory containing the file in which the `#include` directive itself occurred. The meaning of "directory containing" is: the RMS "resultant string" obtained when the file in which the `#include` occurred was opened, except that the filename and subsequent components are replaced by the default file type for headers (".H", or just "." if /ASSUME=NOHEADER_TYPE_DEFAULT is in

Option	Usage
	effect). The "resultant string" will not have translated any concealed device logical.
NONE	Directs the compiler to skip the first step of processing `#include "file.h"` directives. The compiler starts its search for the include file in the /INCLUDE_DIRECTORY directories.

For more information on the search order for included files, see the /INCLUDE_DIRECTORY qualifier.

/OBJECT
/OBJECT[=file-spec]
/OBJECT=.OBJ (D)
/NOOBJECT

Controls whether the compiler produces an output object module. The default output file extension is .OBJ.

Note that the /OBJECT qualifier has no impact on the output file of the /MMS_DEPENDENCIES qualifier.

/OPTIMIZE
/OPTIMIZE[=option]
/OPTIMIZE=(LEVEL=4,INLINE=AUTOMATIC,INTRINSICS,UNROLL=0,
NOOVERRIDE_LIMITS,TUNE=GENERIC) (D)
/NOOPTIMIZE

Controls the level of code optimization that the compiler performs. The options are as follows:

Option	Usage	
LEVEL=n	Selects the level of code optimization. Specify an integer from 0 (no optimization) to 5 (full optimization).	
[NO]INLINE	Provides inline expansion of functions that yield optimized code when they are expanded. You can specify one of the following keywords to control inlining:	
	NONE	No inlining is done, even if requested by the language syntax.
	MANUAL	Inlines only those function calls for which the program explicitly requests inlining.
	AUTOMATIC	Inlines all of the function calls in the MANUAL category, plus additional calls that the compiler determines are appropriate on this platform. On Alpha systems, the heuristics for AUTOMATIC are similar to those for SIZE; on I64 systems, they are more like those for SPEED. AUTOMATIC is the default.
	SIZE	Inlines all of the function calls in the MANUAL category plus any additional calls that the compiler determines would improve run-time performance without significantly increasing the size of the program.

Option	Usage	
	SPEED	Performs more aggressive inlining for run-time performance, even when it might significantly increase the size of the program.
	ALL	Inlines every call that can be inlined while still generating correct code. Recursive routines, however, will not cause an infinite loop at compile time. On I64 systems, ALL is treated as if SIZE had been specified. Note that /OPT=INLINE=ALL is not recommended for general use, because it performs very aggressive inlining and can cause the compiler to exhaust virtual memory or take an unacceptably long time to compile.
[NO]OVERRIDE_LIMITS *(I64 only)*	Controls whether or not the compiler uses certain built-in limits on the size and complexity of a function to "throttle back" the amount of optimization performed in order to reduce the likelihood that the compiler will use excessive memory resources or CPU time attempting to optimize the code. The default is NOOVERRIDE_LIMITS, which means that when compiling a function that has an unusually large number of basic blocks, live variables, or other properties that tend to cause the optimizer to use extra resources, the informational message OPTLIMEXC might be issued to notify you that optimization has been reduced to avoid excessive resource use. You can choose to ignore this message or disable it (the message is not issued on compilations with optimization disabled). Or you can specify /OPTIMIZE=OVERRIDE_LIMITS, which instructs the compiler to not check the limits and to attempt full optimization no matter how large or complex the function, knowing that the compilation might exhaust memory or seem to be in a loop. If using /OPTIMIZE=OVERRIDE_LIMITS does result in excessive resource use, you are sure that the compiler process has plenty of memory quota available, you are convinced that the compilation does not contain any unusually large or complex functions, and you can provide complete source code, then you might want to contact your support channel to see if there is a problem in the compiler causing it to use more resources than it should for the particular compilation at hand.	
TUNE	Specifies the preferred processor for execution. This option makes some decisions preferentially for the specified processor (for example, for code scheduling). Note that code valid only for the specified processor can be generated. However, parallel code can be generated for processors down to the specified architecture level if necessary; that is, tuning specifies the preferred target, while architecture level specifies a lower boundary on available processor features.	

<table>
<tr><th>Option</th><th colspan="2">Usage</th></tr>
<tr><td></td><td colspan="2">For example, /ARCHITECTURE=EV56/OPTIMIZE=TUNE=EV6 specifies that the code does not need to run on a processor older than an EV56, and that the code will probably run on an EV6. The generated code will run on all EV56 and later systems without any emulation. The code might have run-time selected conditional regions specifically for EV6. Also, note that because emulation is provided, the code should run, but potentially at very significantly reduced speed, on pre-EV56 processors.

The options for TUNE are the same as the options for /ARCH. You can specify one of the following keywords:</td></tr>
<tr><td></td><td>GENERIC</td><td>Selects instruction tuning that is appropriate for all implementations of the operating system architecture. This option is the default.</td></tr>
<tr><td></td><td>HOST</td><td>Selects instruction tuning that is appropriate for the machine on which the code is being compiled.</td></tr>
<tr><td></td><td>ITANIUM2 (I64 only)</td><td>Selects instruction tuning for the Intel Itanium 2 processor.</td></tr>
<tr><td></td><td>EV5 (Alpha only)</td><td>Selects instruction tuning for the 21164 implementation of the operating system architecture.</td></tr>
<tr><td></td><td>EV56 (Alpha only)</td><td>Selects instruction tuning for the 21164 implementation of the operating system architecture.</td></tr>
<tr><td></td><td>PCA56 (Alpha only)</td><td>Selects instruction tuning for 21164 chip implementations that use the byte- and word-manipulation instruction extensions of the Alpha architecture.

Running programs compiled with the EV56 keyword might incur emulation overhead on EV4 and EV5 processors, but will still run correctly on OpenVMS Version 7.1 (or later) systems.</td></tr>
<tr><td></td><td>PCA56 (Alpha only)</td><td>Selects instruction tuning for the 21164PC chip implementation that uses the byte- and word-manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture.

Programs compiled with the PCA56 keyword might incur emulation overhead on EV4, EV5, and EV56 processors, but will still run correctly on OpenVMS Version 7.1 (or later) systems.</td></tr>
<tr><td></td><td>EV6 (Alpha only)</td><td>Selects instruction tuning for the first-generation 21264 implementation of the Alpha architecture.</td></tr>
</table>

<table>
<tr><th>Option</th><th colspan="2">Usage</th></tr>
<tr><td></td><td>EV67 (Alpha only)</td><td>Selects instruction tuning for the second-generation 21264 implementation of the Alpha architecture.</td></tr>
<tr><td>[NO]INTRINSICS</td><td colspan="2">Controls whether certain functions are handled as intrinsic functions without explicitly enabling each of them as an intrinsic through the <code>#pragma intrinsic</code> preprocessor directive.

Functions that can be handled as intrinsics are:

Main Group - ANSI:

<code>abs atanl atan2l ceill cosl floorf memcpy sinf
atan atan ceil cos fabs floorl memmove sinl sin
atanf atan2f ceilf cosf floor labs memset strcpy strlen</code>

Main Group - Non-ANSI:

<code>alloca atand2 bzero sind
atand bcopy cosd</code>

Printf functions:

<code>fprintf printf sprintf</code>

Printf non-ANSI:

<code>snprintf</code>

ANSI math functions that set errno, thereby requiring /ASSUME=NOMATH_ERRNO:

<code>acos asinf coshl log log10f powl sqrt tanf
acosf asinl exp logf log10l sinh sqrtf tanl
acosl cosh expf logl pow sinhf sqrtl tanh tanhl
asin coshf expl log10 powf sinhl tan tanhf</code>

Non-ANSI math functions that set errno, thereby requiring /ASSUME=NOMATH_ERRNO:

<code>log2 tand</code>

The /OPTIMZE=INTRINSICS qualifier works with /OPTIMIZE=LEVEL=n and some other qualifiers to determine how intrinsics are handled:

• If the optimization level specified is less than 4, the intrinsic-function prototypes and call formats are checked, but normal run-time calls are still made.</td></tr>
</table>

Option	Usage
	• If the optimization level is 4 or higher, intrinsic code is generated. • Intrinsic code is not generated for math functions that set the errno variable unless /ASSUME=NOMATH_ERRNO is specified. Such math functions, however, do have their prototypes and call formats checked. The default is /OPTIMIZE=INTRINSICS, which turns on this handling. To turn it off, use /NOOPTIMIZE or /OPTIMIZE=NOINTRINSICS.
UNROLL=*n*	Controls loop unrolling done by the optimizer. Specify a positive integer to indicate the number of times to unroll loop bodies. If you specify 0 or do not supply a value, the optimizer determines its own unroll amount. The default is UNROLL=0. Specifying UNROLL=1 effectively disables loop unrolling. On I64 systems, you do not have the ability to control the number of times a loop is unrolled. You can either disable loop unrolling with UNROLL=1, or accept the UNROLL=0 default, which lets the optimizer determine the unroll amount.

The default is /OPTIMIZE, which is equivalent to /OPTIMIZE=LEVEL=4.

/PENDING_INSTANTIATIONS
/PENDING_INSTANTIATIONS[=n]
/PENDING_INSTANTIATIONS=64(D)

Limit the depth of recursive instantiations so that infinite instantiation loops can be detected before some resource is exhausted. The /PENDING_INSTANTIATIONS qualifier requires a positive non-zero value as argument and issues an error when n instantiations are pending for the same class template. The default value for n is 64.

/POINTER_SIZE
/POINTER_SIZE=option
/NOPOINTER_SIZE (D)

Controls whether pointer-size features are enabled, and whether pointers are 32 bits or 64 bits long.

On both Alpha and I64 systems, the default is /NOPOINTER_SIZE, which disables pointer-size features, such as the ability to use `#pragma pointer_size`, and directs the compiler to assume that all pointers are 32-bit pointers. This default represents no change over previous versions of VSI C++.

You can specify one of the following options:

SHORT	The compiler assumes 32-bit pointers.
32	Same as SHORT.
LONG	The compiler assumes 64-bit pointers.

LONG[=ARGV]	The compiler assumes 64-bit pointers. If the ARGV option to LONG or 64 is present, the main argument `argv` will be an array of long pointers instead of an array of short pointers. *(I64 only)*
64	Same as LONG.

Specifying /POINTER_SIZE=32 directs the compiler to assume that all pointers are 32-bit pointers. But unlike the default of /NOPOINTER_SIZE, /POINTER_SIZE=32 enables use of the `#pragma pointer_size long` and `#pragma pointer_size short` preprocessor directives to control pointer size throughout your program.

Specifying /POINTER_SIZE=64 directs the compiler to assume that all pointers are 64-bit pointers, and also enables use of the `#pragma pointer_size` directives.

/PREFIX_LIBRARY_ENTRIES
/PREFIX_LIBRARY_ENTRIES=(option,...)
/NOPREFIX_LIBRARY_ENTRIES
/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES (D)

Controls C Run-Time Library (RTL) name prefixing. For user programs that do not include the ANSI header files but call the ANSI library, the compiler automatically adds a DECC$ prefix to all C RTL library calls just before the name for the external reference or global definition is put into the object file. The C RTL shareable image (DECC$SHR.EXE) resides in IMAGELIB.OLB with a DECC$ prefix for its entry points. Every external name in IMAGELIB.OLB has a DECC$ prefix, and, therefore, has an OpenVMS-conformant name space (a requirement for inclusion in IMAGELIB).

The options are as follows:

Option	**Usage**
EXCEPT=(*name*,...)	The names specified are not prefixed.
ALL_ENTRIES	All VSI C++ names are prefixed. Note: 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. In addition, the compiler will issue a CC-W-NOTINCRTL message when it prefixes a name that is not in the current C RTL.
ANSI_C89_ENTRIES	Only ANSI/ISO C library names are prefixed.
RTL=`name`	References to the C RTL, indicated by NAME, are generated. NAME must be 1017 characters or fewer.

If you want no names prefixed, specify /NOPREFIX_LIBRARY_ENTRIES.

/PREPROCESS_ONLY
/PREPROCESS_ONLY[=filename]
/NOPREPROCESS_ONLY (D)

Causes the compiler to perform only the actions of the preprocessor phase and write the resulting processed text out to a file. The default output file extension is .IXX. Use of /PREPROCESS_ONLY prevents the generation of an object or XREF file.

/PSECT_MODEL
/PSECT_MODEL=MULTILANGUAGE
/PSECT_MODEL=NOMULTILANGUAGE (D)

Controls whether the compiler allocates the size of overlaid psects to ensure compatibility when the psect is shared by code created by other VSI compilers.

This qualifier solves a problem that can occur when a psect generated by a Fortran COMMON block is overlaid with a psect consisting of a C struct. Because Fortran COMMON blocks are not padded, if the C struct is padded, the inconsistent psect sizes can cause linker error messages.

Compiling with /PSECT_MODEL=MULTILANGUAGE ensures that the compiler uses a consistent psect size allocation scheme. The corresponding Fortran squalifier is /ALIGN=COMMON=[NO]MULTILANGUAGE.

The default is /PSECT=NOMULTILANGUAGE, which should be sufficient for most applications.

/PURE_CNAME
/PURE_CNAME (D) (/STANDARD=STRICT_ANSI)
/NOPURE_CNAME (D) (All other modes)

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. In this mode, the __PURE_CNAME and __HIDE_FORBIDDEN_NAMES macros are predefined by the compiler.

In /NOPURE_CNAME mode, the `<cname>` headers insert the name into the `std` namespace and also into the global namespace. In this mode, the __PURE_CNAME and __HIDE_FORBIDDEN_NAMES macros are not predefined by the compiler.

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>` header instead of its `<cname>` 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 /NOPURE_CNAME mode.

/QUIET
/QUIET
/NOQUIET (D)

Specifying /QUIET causes the compiler to report errors like the Version 5.n compiler (issue fewer messages). This is the default for ARM mode (/STANDARD=ARM). All other modes default to /NOQUIET.

Use /WARNINGS=ENABLE to enable specific messages normally disabled with /QUIET.

/REENTRANCY
/REENTRANCY=option
/REENTRANCY=TOLERANT (D)

Controls the type of reentrancy that reentrant C RTL routines exhibit. (See also the DECC$SET_REENTRANCY RTL routine.)

This qualifier is for use only with a module containing the main routine.

The reentrancy level is set at run time according to the /REENTRANCY qualifier specified while compiling the module containing the main routine. This option affects the behavior of the C RTL, but has no effect on the C++ libraries.

You can specify one of the following options:

Option	Usage
AST	Uses the `__TESTBITSSI` built-in function to perform simple locking around critical sections of RTL code, and may additionally disable asynchronous system traps (ASTs) in locked region of codes. This type of locking should be used when AST code contains calls to VSI C RTL I/O routines.
MULTITHREAD	Designed to be used in conjunction with the DECthreads product. It performs DECthreads locking and never disables ASTs.
NONE	Gives optimal performance in the RTL, but does absolutely no locking around critical sections of RTL code. It should be used only in a single threaded environment when there is no chance that the thread of execution will be interrupted by an AST that would call the C RTL.
TOLERANT	Uses the `__TESTBITSSI` built-in function to perform simple locking around critical sections of RTL code, but ASTs are not disabled. This type of locking should be used when ASTs are used and must be delivered immediately. This is the default reentrancy type.

/REPOSITORY
/REPOSITORY=(PATHNAME [,...])
/REPOSITORY=[.CXX_REPOSITORY] (D)

Specifies a repository that C++ uses to store requested template instantiations. The default is /REPOSITORY=[.CXX_REPOSITORY]. If multiple repositories are specified, only the first is considered writable and the default repository is ignored unless specified.

/ROUNDING_MODE
/ROUNDING_MODE=option
/ROUNDING_MODE=NEAREST (D)

Lets you select an IEEE rounding mode if /FLOAT=IEEE_FLOAT is specified. The options are as follows:

Option	Usage
CHOPPED	Rounds toward 0.
DYNAMIC	Sets the rounding mode for IEEE floating-point instructions dynamically, as determined from the contents of the floating-point control register.

Option	Usage
MINUS_INFINITY	Rounds toward minus infinity.
NEAREST	Sets the normal rounding mode (unbiased round to nearest). This is the default.

If you specify either /FLOAT=G_FLOAT or /FLOAT=D_FLOAT, then rounding defaults to /ROUNDING_MODE=NEAREST, with no other choice of rounding mode.

/RTTI
/RTTI (D)
/NORTTI *(Alpha only)*

Enables or disables support for RTTI (runtime type identification) features: dynamic_cast and typeid. Disabling runtime type identification can also save space in your object file because static information to describe polymorphic C++ types is not generated. The default is to enable runtime type information features and generate static information in the object file. The /RTTI qualifier defines the macro __RTTI.

Note that specifying /NORTTI does not disable exception handling.

/SHARE_GLOBALS
/SHARE_GLOBALS
/NOSHARE_GLOBALS (D)

Controls whether the initial extern_model is shared or not shared (for those extern_models where it is allowed). The initial extern_model of the compiler is a fictitious pragma constructed from the settings of the /EXTERN_MODEL and /SHARE_GLOBALS.

The default value is /NOSHARE_GLOBALS, which has the following effects:

- When linking old object files or object libraries with newly produced object files, you might get "conflicting attributes for psect" messages, which can be safely ignored as long as you are not building shareable libraries.
- The /noshare_globals default makes building shareable libraries easier.

/SHOW
/SHOW=(option[,...])
/SHOW=(HEADER,SOURCE) (D)

Used with the /LIST qualifier to set or cancel specific listing options. You can select the following options:

Option	Usage
ALL	Print all listing information
[NO]HEADER	Print/do not print header lines at the top of each page (D = HEADER)
[NO]INCLUDE	Print/do not print contents of `#include` files (D = NOINCLUDE)
NONE	Print no listing information

Option	Usage
[NO]SOURCE	Print/do not print source file statements (D = SOURCE)
[NO]STATISTICS	Print/do not print compiler performance statistics (D = NOSTATISTICS). On I64 systems, the /SHOW=STATISTICS option is ignored.

/STANDARD
/STANDARD=(option)
/STANDARD=RELAXED (D)

The compiler implements the International ANSI C++ Standard. The /STANDARD qualifier directs the compiler to interpret source code according to certain nonstandard syntax conventions followed by other implementations of the C++ language. The options are:

Option	Usage
RELAXED	Allow language constructs required by the International ANSI C++ Standard. This mode also supports some non-ANSI extensions and issues messages for some nonstandard usage that does not strictly comply with the standard. This is the default compiler mode. This option also defines the macro __STD_ANSI. Please note that ANSI is accepted as a synonym for RELAXED to be compatible with previous C++ versions.
ARM	Minimize source changes when compiling programs developed using Version 5.n. This option also defines the macro __STD_ARM. The /STANDARD=ARM qualifier uses the pre-ansi AT&T version of the iostream library and defines the macro __NO_USE_STD_IOSTREAM.
CFRONT	As of VSI C++ Version 7.1, support for /STANDARD=CFRONT is retired.
GNU	Use this option if you want to compile programs developed using the GNU C++ compiler. This option also defines the __STD_GNU macro. The /STANDARD=GNU qualifier uses the pre-ansi AT&T version of the iostream library and defines the macro __NO_USE_STD_IOSTREAM. The following changes in behavior are provided for compatibility with the GNU C++ compiler: • These options are enabled by default: /ALTERNATIVE_TOKENS /TEMPLATE_DEFINE=LOCAL /NOIMPLICIT_INCLUDE • Access control is not enforced for types defined inside a class. • Unrecognized character escape sequences in string literals produce an informational instead of a warning message. • The __INLINE keyword is enabled and is equivalent to inline. • The severity of the error "incompatible parameter" (tag incompatibleprm) is reduced to warning. • When overloading, enum types are treated as integral types.

Option	Usage
	The following known incompatibility is not addressed in the /STANDARD=GNU mode: • The compiler strictly enforces the requirement to define functions before they are used. This requirement also applies to built-in functions such as strlen.
MS	Allow language constructs supported by the Visual C++ compiler. This option also defines the macro __STD_MS. The /STANDARD=MS qualifier uses the pre-ansi AT&T version of the iostream library and defines the macro __NO_USE_STD_IOSTREAM.
STRICT_ANSI	Enforce the ANSI standard strictly but permit some ANSI violations that should be errors to be warnings. This option also defines the macro __STD_STRICT_ANSI. To force ANSI violations to be issued as errors instead of warnings, use /WARNINGS=ANSI_ERRORS in addition to /STANDARD=STRICT_ANSI. This combination defines the macro __STD_STRICT_ANSI_ERRORS. The /STANDARD=STRICT_ANSI qualifier uses the ANSI/ISO standard version of the iostream library and defines the macro __USE_STD_IOSTREAM.
LATEST	Latest C++ standard dialect *(Alpha, I64)*. /STANDARD=LATEST is currently equivalent to /STANDARD=STRICT_ANSI, but is subject to change when newer versions of the C++ standard are released.

For more information on the effect of the /STANDARD qualifier on VSI C++ compile-time error checking, Section E.1.

/TEMPLATE_DEFINE=(option,...)
/NOTEMPLATE_DEFINE

Controls compiler behavior pertaining to the instantiation of C++ templates. See Chapter 5 for details on how to instantiate templates using this qualifier.

Note that you must specify a value for /TEMPLATE_DEFINE.

Select only one of the following optional values to determine the template instantiation model:

Option	Usage
ALL	Instantiate all template entities declared or referenced in the compilation unit, including typedefs. For each fully instantiated template class, all its member functions and static data members are instantiated even if they were not used. Nonmember template functions are instantiated even if the only reference was a declaration. Instantiations are created with external linkage. Overrides /REPOSITORY at compile time. Instantiations are placed in the user's object file.
ALL_REPOSITORY	Instantiate all templates declared or used in the source program and put the object code generated as separate object files in the repository. Instantiations caused by manual instantiation directives are also put in the repository. This is similar to /TEMPLATE_DEFINE=ALL except that explicit instantiations are also put in the repository, rather

Option	Usage
	than than an external symbol being put in the main object file. This qualifier is useful for creating a pre-instantiation library.
AUTOMATIC	Directs the compiler to use the automatic instantiation model of C++ templates. / TEMPLATE_DEFINE=AUTOMATIC is the default.
NOAUTOMATIC	Directs the compiler to not implicitly instantiate templates.
IMPLICIT_LOCAL	Same as /TEMPLATE_DEFINE=LOCAL, except manually instantiated templates are placed in the repository with external linkage. This is useful for build systems that need to have explicit control of the template instantiation mechanism. This mode can suffer the same limitations as /TEMPLATE_DEFINE=LOCAL. This mode is the default when /STANDARD=GNU is specified.
LOCAL	Similar to /TEMPLATE_DEFINE=USED except that the functions are given internal linkage. This qualifier provides a simple mechanism for getting started with templates. The compiler instantiates as local functions the functions used in each compilation unit, and the program links and runs correctly (barring problems resulting from multiple copies of local static variables). However, because many copies of the instantiated functions can be generated, this qualifier might not be not suitable for production use. The /TEMPLATE_DEFINE=LOCAL qualifier cannot be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it is disabled by the /TEMPLATE_DEFINE=LOCAL qualifier. Explicit use of /TEMPLATE_DEFINE=LOCAL and /TEMPLATE_DEFINE=AUTO is an error.
USED	Instantiate those template entities that were used in the compilation. This includes all static data members for which there are template definitions. Overrides /TEMPLATE_DEFINE=AUTO at compile time.
USED_REPOSITORY	Like ALL_REPOSITORY, but instantiates only templates used by the compilation. The explicit instantiations are also put into the repository as separate object files.

The following /TEMPLATE_DEFINE optional values are independent of the model used and each other:

Option	Usage
DEFINITION_FILE_TYPE="file-type-list"	Specifies a string that contains a list of file types that are valid for template definition files. Items in the list must be separated by commas and preceded by a period. A type is not allowed to exceed the OpenVMS limit of 31 characters. This qualifier is applicable only when automatic instantiation has been specified. The default is /TEMPLATE_DEFINE=DEF= ".CXX,.C,.CC,.CPP".

Option	Usage
PRAGMA	Determines whether the compiler ignores `#pragma define_template` directives encountered during the compilation. This qualifier lets you quickly switch to automatic instantiation without having to remove all the pragma directives from your program's code base. The default is /TEMPLATE_DEFINE=PRAGMA, which enables `#pragma define_template`.
VERBOSE	Turns on verbose or verify mode to display each phase of instantiation as it occurs. During the compilation phase, informational level diagnostics are generated to indicate which templates are automatically being instantiated. This qualifier is useful as a debugging aid.
TIMESTAMP *(Alpha only)*	only applicable if a repository is being used. Causes the compiler to create a timestamp file named TIMESTAMP. in the repository. Thereafter, instantiations are added or regenerated only if needed; that is, if they do not alreay exist, or if existing ones are older than the timestamp. Also see /REPOSITORY.

Also see /PENDING_INSTANTIATIONS.

/UNDEFINE
/UNDEFINE=(identifier[,...])
/NOUNDEFINE (D)

Performs the same function as the `#undefine` preprocessor directive: it cancels a macro definition.

The /UNDEFINE qualifier is useful for undefining the predefined C++ preprocessor constants. For example, if you use a preprocessor constant to conditionally compile segments of code specific to C++ for OpenVMS systems, you can undefine constants to see how the portable sections of your program execute. For example:

```
/UNDEFINE="deccxx"
```

When both /DEFINE and /UNDEFINE are present on the CXX command line, /DEFINE is evaluated before /UNDEFINE.

/UNSIGNED_CHAR
/UNSIGNED_CHAR
/NOUNSIGNED_CHAR (D)

The /UNSIGNED_CHAR qualifier changes the default for all char types from signed to unsigned. The /NOUNSIGNED_CHAR qualifier causes all plain char declarations to have the same representation and set of values as signed char declarations.

/USING_STD
/USING_STD
/NOUSING_STD (D)

Controls whether standard library header files are processed as though the compiled code were written as follows:

```
using namespace std;
#include <header>
```

These options are provided for compatibility for users who do not want to qualify use of each standard library name with `std::` or put using namespace std; at the top of their sources.

/USING_STD turns implicit using namespace std on; this is the default when compiling /STANDARD=ARM, /STANDARD=GNU, /STANDARD=MS, or /STANDARD=RELAXED.

/NOUSING_STD turns implicit using namespace std off; this is the default when compiling /STANDARD=STRICT_ANSI.

/VERSION
/VERSION
/NOVERSION (D)

Causes the compiler to identify (print out) its version and operating system. The listing file also contains the compiler version. You cannot specify this qualifier with any other qualifiers.

/WARNINGS
/WARNINGS[=(option[,...])]
/WARNINGS (D)
/NOWARNINGS

Controls the issuance of compiler diagnostic messages and lets you modify the severity of messages.

The default qualifier, /WARNINGS, outputs all enabled warning and informational messages for the compiler mode you are using. The /NOWARNINGS qualifier suppresses warning and informational messages.

Options apply only to warning and informational messages.

The *message-list* in the following table of options can be any one of the following:

- A single message identifier (within parentheses, or not). The message identifier is the name following the message severity letter on the first line of an issued message. For example, in the following message, the message identifier is GLOBALEXT:

  ```
  %CC-W-GLOBALEXT, a storage class of globaldef, globalref, or
   globalvalue
  is a language extension.
  ```

- A comma-separated list of message identifiers, enclosed in parentheses.
- The keyword ALL.

The options are processed and take effect in the following order:

NOWARNINGS	Suppresses warnings.
NOINFORMATIONALS	Suppresses informational messages.
ENABLE=*message-list*	Enables issuance of the specified messages. Can be used to enable specific messages that normally would not be issued when using /QUIET or messages disabled with /WARNINGS=DISABLE.

DISABLE=*message-list*	Disables issuance of the specified messages. Can be used for any nonerror message specified by a message number or tag. Specify ALL to suppress all informationals and warnings.
INFORMATIONALS=*message-list*	Sets the severity of all specified messages to Informational. Fatal and Error messages cannot be made less severe. Can also be used to enable informationals that are disabled by default. Note: With C++ Version 7.1, using /WARNINGS=INFORMATIONALS=`<tag>` no longer enables all other informational messages.
WARNINGS=*message-list*	Sets the severity of the specified messages to Warning. Fatal and Error messages cannot be made less severe.
[NO]ANSI_ERRORS	Issues error messages for all ANSI violations when in STRICT_ANSI mode. The default is /WARNINGS=NOANSI_ERRORS.
[NO]TAGS	Displays a descriptive tag at the end of each message. "D" indicates that the severity of the message can be controlled from the command line. The tag displayed can be used as the message identifier in the /WARNINGS qualifier options.
ERRORS=*message-list*	Sets the severity of the specified messages to Error. Supplied Error and Fatal messages cannot be made less severe. (Exception: A message can be upgraded from Error to Fatal, then later downgraded to Error again, but it can never be downgraded from Error.) Warnings and Informationals can be made any severity.
FATALS=*message-list*	Sets the severity of the specified messages to Fatal.

Also see the `#pragma message` preprocessor directive.

/XREF *(Alpha only)*
/XREF[=file-spec]
/NOXREF (D)

Controls whether the compiler generates a file of source code analysis information. The default file name is the file name of the primary source file; the default file type is .XREF. Use the SCA IMPORT command to convert an .XREF file into an analysis data file that is ready for loading into an SCA library.

Appendix B. Programming Tools

This appendix provides information on tools that you can use to develop and refine your C++ programs. Some ship with the OpenVMS operating system but others require separate purchase.

B.1. VSI Language-Sensitive Editor

The VSI Language-Sensitive Editor (LSE) is a text editor intended specifically for software development. LSE includes the following features:

- Formatted language constructs, or templates, for most VSI programming languages. These templates include keywords and required punctuation, and use placeholders to indicate where to insert optional or required code.
- Commands for compiling, reviewing, and correcting compilation errors from within the editor.
- Integration with VSI Code Management System (CMS). You can enter CMS commands from within the editor to coordinate the progress of program development on OpenVMS systems. For more information on CMS, see the *VSI DECset for OpenVMS Guide to the Code Management System*.

B.1.1. Starting and Terminating an LSE Session

To invoke LSE and associate a buffer with C++, use the following syntax:

```
LSEDIT [/qualifier...]filename.cxx
```

To invoke LSE without associating the editing buffer with a programming language, enter the following command at the DCL prompt:

```
$ lsedit file-spec
```

To end an LSE session, press Ctrl/Z to get the LSE> prompt. Then, enter the `exit` command if you want to save the current file modification, or enter the `quit` command if you want to discard the current file modification.

B.1.2. LSE Placeholders and Tokens

The language-sensitive features of LSE simplify the tasks of writing and maintaining program code. Among these features are placeholders and tokens.

Placeholders are markers in the source code that indicate where a program element is expected. Placeholders are worded to denote the appropriate syntax in a given context. You do not need to type placeholders; LSE inserts them, surrounded by brackets ([]) or braces ({}) and at signs (@). Braces indicate where source code is required in the program's context; brackets indicate that you have the option of supplying additional constructs or erasing the placeholder.

Tokens are LSE words that, when expanded, provide additional language constructs. You can type tokens directly into the buffer. You use tokens in situations such as modifying an existing program to add program elements where no placeholders exist. For example, if you type `while` and then enter the `expand` command, a template for a `while` construct appears in your buffer in place of the characters you typed. You also can use tokens as a shortcut in situations where expanding a placeholder would entail a complicated sequence of actions.

LSE has commands for manipulating tokens and placeholders, as follows:

Command	Default Key Binding	Description
`expand`	Ctrl/E	Expands a placeholder
`unexpand`	PF1-Ctrl/E	Reverses the effect of the most recent placeholder expansion
`goto placeholder/forward`	Ctrl/N	Moves the cursor forward to the next placeholder
`goto placeholder/reverse`	Ctrl/P	Moves the cursor backward to the previous placeholder
`erase placeholder/forward`	Ctrl/K	Erases a placeholder
`unerase placeholder`	PF1-Ctrl/K	Restores the most recently erased placeholder
{Enter | Return}	Enter Return	Selects a menu option

To display a list of all the predefined tokens supplied with C++, enter the following LSE command:

```
LSE> show token
```

To display a list of all the predefined placeholders supplied with C++, enter the following LSE command:

```
LSE> show placeholder
```

For information about a particular token or placeholder, specify the name of the token or placeholder after the `show token` or `show placeholder` command.

To create a list of either tokens or placeholders, first execute the appropriate `show` command to put the list in the `$show` buffer. Then, enter the following commands:

```
LSE> go buffer $show
LSE> write file-spec
```

When you exit LSE, you can use the DCL `print` command to print a copy of the file you wrote.

B.1.3. Compiling and Reviewing Source Code from an LSE Session

To compile your source code and to review compilation errors without leaving the editing session, use the LSE commands `compile` and `review`. The `compile` command issues a DCL command in a subprocess to invoke the compiler. The compiler then generates a file of compile-time diagnostic information that LSE uses to review any compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that LSE appends to the compilation command.

For example, if you enter the `compile` command while editing the buffer `user.cxx`, LSE executes the following DCL command:

```
$ CXX user.cxx/DIAGNOSTICS=USER.DIA
```

LSE supports all the command qualifiers available with the compiler.

The /DIAGNOSTICS qualifier is ignored on I64 systems.

The `review` command displays any diagnostic messages that result from a compilation. LSE displays the compilation errors in one window and corresponding source code in a second window. This lets you view the error messages while examining the associated code.

B.1.4. VSI Source Code Analyzer (SCA)

Although the compiler does not support the VSI Source Code Analyzer (SCA) through the CXX /ANALYSIS_DATA command-line qualifier, users can generate a `.ana` file that contains information on all the tokens within a C++ program. For example, users can do the following:

```
SCA> find some_variable_name
```

To use the SCA `analyze` command with C++ files so that in turn you can execute `find` commands on your C++ code from LSE or SCA, do the following:

- At the command line, issue the SCA command:

  ```
  $ SCA
  ```

- Set your SCA library with the `set library` command. For example:

  ```
  SCA> set library projdisk:[user.any_existing_sca_lib]
  ```

- Issue the `analyze` command on your `.cxx` file:

  ```
  SCA> analyze testprog.cxx
  ```

  This command places the file `testprog.ana` in your current working directory.

- Load the resulting `.ana` file:

  ```
  SCA> load testprog.ana
  ```

Appendix C. Built-In Functions

This appendix describes the built-in functions available when you compile on OpenVMS systems. These functions allow you to access hardware and machine instructions directly.

These functions allow you to directly access hardware and machine instructions to perform operations that are cumbersome, slow, or impossible in other C++ compilers.

These functions are very efficient because they are built into the VSI C++ compiler. This means that a call to one of these functions does not result in a reference to a function in the VSI C Run-Time Library or to a function in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. Because most of these built-in functions closely correspond to single Alpha or Itanium machine instructions, the result is small, fast code.

Be sure to include the `<builtins.h>` header file in your source program to access these built-in functions. Definitions for return types `int64` and `uint64` are contained in the header file `<ints.h>`.

Some of the built-in functions have optional arguments or allow a particular argument to have one of many different types. To describe all valid combinations of arguments, the following built-in function descriptions list several different prototypes for the function. As long as a call to a built-in function matches one of the prototypes listed, the call is valid. Furthermore, any valid call to a built-in function behaves as if the corresponding prototype were in scope of the call. The compiler, therefore, performs the argument checking and conversions specified by that prototype.

The majority of the built-in functions are named after the processor instruction that they generate. The built-in functions provide direct and unencumbered access to those processor instructions. Any inherent limitations to those instructions are limitations to the built-in functions as well. For instance, the MOVC3 instruction and the _MOVC3 built-in function can move at most 65,535 characters.

For more information on the Alpha built-in functions, see the corresponding machine instructions in the *Alpha Architecture Handbook* or *Alpha Architecture Reference Manual*.

For more information on the I64 built-in functions, see the corresponding machine instructions in the *Intel® Itanium® Architecture Software Developer's Manual*.

C.1. Built-In Functions for Alpha Systems *(Alpha only)*

The following sections describe the VSI C++ built-in functions available on OpenVMS Alpha systems.

C.1.1. Translation Macros

VSI C++ for OpenVMS systems does not support the built-in functions available with VSI C++ for OpenVMS VAX systems. However, the `<builtins.h>` header file contains macro definitions that translate some VAX C built-in functions to the equivalent VSI C++ for OpenVMS built-in functions. Consequently, the following VAX C built-in functions are effectively supported:

```
_BBCCI(position, address)

_BBSSI(position, address)
```

```
_INSQHI(new_entry, head)

_INSQTI(new_entry, head)

_INSQUE(new_entry, predecessor)

_REMQHI(head, removed_entry)

_REMQTI(head, removed_entry)

_REMQUE(entry, removed_entry)

_PROBER(mode, length, address)

_PROBEW(mode, length, address)
```

For more detail on any of these functions, see <builtins.h> or the description of the corresponding native Alpha function in this chapter. For example, for a description of _INSQHI, see the section called "__PAL_INSQHIL".

C.1.2. Intrinsic Functions

VSI C++ on Alpha systems supports in-line assembly code, commonly called ASMs on UNIX platforms.

Like built-in functions, ASMs are implemented with a function-call syntax. But unlike built-in functions, to use ASMs you must include the <c_asm.h> header file containing prototypes for the three types of ASMs, and the #pragma intrinsic preprocessor directive.

Syntax:

```
__int64 asm(const char *, ...);  /* for integer operations,
                                    like mulq */

float fasm(const char *, ...);   /* for single precision float
                                    instructions */

double dasm(const char *, ...);  /* for double precision float
                                    instructions */

         #pragma intrinsic (asm)
         #pragma intrinsic (fasm)
         #pragma intrinsic (dasm)
```

The first argument to the asm, fasm, or dasm function contains the instruction(s) to be generated inline and the metalanguage that describes the interpretation of the arguments.

The remaining arguments (if any) are the source and destination arguments for the instruction being generated.

C.1.3. Privileged Architecture Library Code Instructions

The following Privileged Architecture Library Code (PALcode) instructions are available as built-in functions:

```
__PAL_GENTRAP __PAL_INSQHIL   __PAL_REMQHIL   __PAL_MTPR_ASTEN
 __PAL_MFPR_ASTEN
```

```
__PAL_HALT     __PAL_INSQTIL    __PAL_REMQTIL    __PAL_MTPR_ASTSR
 __PAL_MFPR_ASTSR
__PAL_PROBER   __PAL_INSQUEL    __PAL_REMQUEL    __PAL_MTPR_DATFX
 __PAL_MFPR_ESP
__PAL_PROBEW   __PAL_INSQHIQ    __PAL_REMQHIQ    __PAL_MTPR_ESP
 __PAL_MFPR_FEN
__PAL_CHME     __PAL_INSQTIQ    __PAL_REMQTIQ    __PAL_MTPR_FEN
 __PAL_MFPR_IPL
__PAL_CHMK     __PAL_INSQUEQ    __PAL_REMQUEQ    __PAL_MTPR_IPIR
 __PAL_MFPR_MCES
__PAL_CHMS     __PAL_INSQUEL_D  __PAL_REMQUEL_D  __PAL_MTPR_IPL
 __PAL_MFPR_PCBB
__PAL_CHMU     __PAL_INSQUEQ_D  __PAL_REMQUEQ_D  __PAL_MTPR_MCES
 __PAL_MFPR_PRBR
__PAL_LDQP     __PAL_INSQHILR   __PAL_REMQHILR   __PAL_MTPR_PRBR
 __PAL_MFPR_PTBR
__PAL_STQP     __PAL_INSQTILR   __PAL_REMQTILR   __PAL_MTPR_SCBB
 __PAL_MFPR_SCBB
__PAL_BPT      __PAL_INSQHIQR   __PAL_REMQHIQR   __PAL_MTPR_SIRR
 __PAL_MFPR_SISR
__PAL_BUGCHK   __PAL_INSQTIQR   __PAL_REMQTIQR   __PAL_MTPR_SSP
 __PAL_MFPR_SSP
__PAL_CFLUSH                                     __PAL_MTPR_TBIA
 __PAL_MFPR_TBCHK
__PAL_DRAINA                                     __PAL_MTPR_TBIAP
 __PAL_MFPR_USP
__PAL_RD_PS                                      __PAL_MTPR_TBIS
 __PAL_MFPR_VPTB
__PAL_SWPCTX                                     __PAL_MTPR_TBISD
 __PAL_MFPR_WHAMI
__PAL_SWASTEN                                    __PAL_MTPR_TBISI
__PAL_WR_PS_SW                                   __PAL_MTPR_USP
__PAL_IMB                                        __PAL_MTPR_VPTB
```

The following sections describe these PALcodes.

__PAL_BPT

This function is provided for program debugging. It generates a Breakpoint trap.

This function has the following format:

```
void  __PAL_BPT (void);
```

__PAL_BUGCHK

This function is provided for error reporting. It generates a Bug-check trap.

This function has the following format:

__PAL_CFLUSH

This function flushes at least the entire physical page specified by the page frame number *value* from any data caches associated with the current processor. After a CFLUSH is done, the first subsequent load on the same processor to an arbitrary address in the target page is fetched from physical memory.

This function has the following format:

```
void  __PAL_CFLUSH (int value);
```

value

A page frame number.

__PAL_CHME

This function allows a process to change its mode to Executive in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 to R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction.

This function has the following format:

```
void  __PAL_CHME (void);
```

__PAL_CHMK

This function allows a process to change its mode to kernel in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 to R7, PS, and PC are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction.

This function has the following format:

```
void  __PAL_CHMK (void);
```

__PAL_CHMS

This function allows a process to change its mode to Supervisor in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 to R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.

This function has the following format:

```
void  __PAL_CHMS (void);
```

__PAL_CHMU

This function allows a process to call a routine using the change mode mechanism. Registers R2 to R7, PS, and PC are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction.

This function has the following format:

```
void  __PAL_CHMU (void);
```

__PAL_DRAINA

This function stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.

This function has the following format:

```
void  __PAL_DRAINA (void);
```

__PAL_GENTRAP

This function is used for reporting run-time software conditions. It generates a Software trap.

This function has the following format:

```
void  __PAL_GENTRAP (uunsigned __int64 encoded_software_trap);
```

encoded_software_trap

The particular software condition that has occurred.

__PAL_HALT

This function halts the processor when executed by a process running in kernel mode. This is a privileged function.

This function has the following format:

```
void  __PAL_HALT (void);
```

__PAL_IMB

This function makes the instruction stream coherent with the data stream. It must be executed after software or I/O devices write into the instruction stream or modify the instruction stream virtual address mapping, and before the new value is fetched as an instruction. Note that executing an IMB on one processor in a multiprocessor environment does not affect instruction caches on other processors.

This function has the following format:

```
void  __PAL_IMB (void);
```

__PAL_INSQHIL

This function inserts an entry at the front of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQHIL (void *head, void *new_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a longword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed

- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQHILR

This function inserts an entry into the front of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQHILR (void *head, void *new_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQHIQ

This function inserts an entry at the front of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQHIQ (void *head, void *new_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list

- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQHIQR

This function inserts an entry into the front of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQHIQR (void *head, void *new_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQTIL

This function inserts an entry at the end of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQTIL (void *head, void *new_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQTILR

This function inserts an entry at the end of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQTILR (void *head, void *new_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQTIQ

This function inserts an entry at the end of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQTIQ (void *head, void *new_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQTIQR

This function inserts an entry at the end of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQTIQR (void *head, void *new_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

__PAL_INSQUEL

This function inserts a new entry after an existing entry into a longword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEL (void *predecessor, void *new_entry);
```

predecessor

A pointer to an existing entry in the queue.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

__PAL_INSQUEL_D

This function inserts a new entry after an existing entry into a longword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEL_D (void **predecessor, void *new_entry);
/* Deferred */
```

predecessor

A pointer to a pointer to the predecessor entry.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

__PAL_INSQUEQ

This function inserts a new entry after an existing entry into a quadword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEQ (void *predecessor, void *new_entry);
```

predecessor

A pointer to an existing entry in the queue.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

__PAL_INSQUEQ_D

This function inserts a new entry after an existing entry into a quadword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEQ_D (void **predecessor, void *new_entry);
/* Deferred */
```

predecessor

A pointer to a pointer to the predecessor entry.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue

- 1 if the entry was the only entry in the queue

__PAL_LDQP

This function returns the quadword-aligned memory object specified by *address*.

This function has the following format:

```
unsigned __int64  __PAL_LDQP (void *address);
```

address

A pointer to the quadword-aligned memory object to be returned.

If the object pointed to by *address* is not quadword-aligned, the result is unpredictable.

__PAL_MFPR_XXXX

These privileged functions return the contents of a particular processor register. The *XXXX* indicates the processor register to be read.

These functions have the following format:

```
/* AST Enable */
unsigned int  __PAL_MFPR_ASTEN (void);
/* AST Summary Register */
unsigned int  __PAL_MFPR_ASTSR (void);
/* Executive Stack Pointer */
void  *__PAL_MFPR_ESP (void);
/* Floating-Point Enable */
int  __PAL_MFPR_FEN (void);
/* Interrupt Priority Level */
int  __PAL_MFPR_IPL (void);
/* Machine Check Error Summary */
int  __PAL_MFPR_MCES (void);
/* Privileged Context Block Base */
void  *__PAL_MFPR_PCBB (void);
/* Processor Base Register */
__int64  __PAL_MFPR_PRBR (void);
/* Page Table Base Register */
int  __PAL_MFPR_PTBR (void);
/* System Control Block Base */
void  *__PAL_MFPR_SCBB (void);
/* Software Interrupt Summary Register */
unsigned  int  __PAL_MFPR_SISR (void);
/* Supervisor Stack Pointer */
void  *__PAL_MFPR_SSP (void);
/* Translation Buffer Check */
__int64  __PAL_MFPR_TBCHK (void *address);
/* User Stack Pointer */
void  *__PAL_MFPR_USP (void);
/* Virtual Page Table */
void  *__PAL_MFPR_VPTB (void);
/* Who Am I */
__int64  __PAL_MFPR_WHAMI (void);
```

__PAL_MTPR_XXXX

These privileged functions load a value into one of the special processor registers. The *XXXX* indicates the processor register to be loaded.

These functions have the following format:

```
/* AST Enable */
void  __PAL_MTPR_ASTEN (unsigned int mask);
/* AST Summary Register */
void  __PAL_MTPR_ASTSR (unsigned int mask);
/* Data Alignment Trap Fixup */
void  __PAL_MTPR_DATFX (int value);
/* Executive Stack Pointer */
void  __PAL_MTPR_ESP (void *address);
/* Floating-Point Enable */
void  __PAL_MTPR_FEN (int value);
/* Interprocessor Interrupt Request */
void  __PAL_MTPR_IPIR (__int64 number);
/* Interrupt Priority Level */
int  __PAL_MTPR_IPL (int value);
/* Machine Check Error Summary */
void  __PAL_MTPR_MCES (int value);
/* Processor Base Register */
void  __PAL_MTPR_PRBR (__int64 value);
/* System Control Block Base */
void  __PAL_MTPR_SCBB (void *address);
/* Software Interrupt Request Register */
void  __PAL_MTPR_SIRR (int level);
/* Supervisor Stack Pointer */
void  __PAL_MTPR_SSP (int *address);
/* Translation Buffer Invalidate All*/
void  __PAL_MTPR_TBIA (void);
/* Translation Buffer Invalidate All Process */
void  __PAL_MTPR_TBIAP (void);
/* Translation Buffer Invalidate Single */
void  __PAL_MTPR_TBIS (void *address);
/* Translation Buffer Invalidate Single Data */
void  __PAL_MTPR_TBISD (void *address);
/* Translation Buffer Invalidate Single Instruction */
void  __PAL_MTPR_TBISI (void *address);
/* User Stack Pointer */
void  __PAL_MTPR_USP (void *address);
/* Virtual Page Table */
void  __PAL_MTPR_VPTB (void *address);
```

__PAL_PROBER

This function checks the write accessibility of the first and last byte of the given address and length pair.

This function has the following format:

```
int  __PAL_PROBER (const void *base_address, int length,
char mode);
```

base_address

The pointer to the memory segment to be tested for read access.

length

The length of the memory segment, in bytes.

mode

The processor mode used for checking access.

There are two possible return values:

- 0 if both bytes are not accessible
- 1 if both bytes are accessible

__PAL_PROBEW

This function checks the write accessibility of the first and last byte of the given address and length pair.

This function has the following format:

```
int  __PAL_PROBEW (const void *base_address, int length,
char mode);
```

base_address

The pointer to the memory segment to be tested for write access.

length

The length of the memory segment, in bytes.

mode

The processor mode used for checking access.

There are two possible return values:

- 0 if both bytes are not accessible
- 1 if both bytes are accessible

__PAL_RD_PS

This function returns the Processor Status (PS).

This function has the following format:

```
unsigned __int64  __PAL_RD_PS (void);
```

__PAL_REMQHIL

This function removes the first entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQHIL (void *head, void **removed_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQHILR

This function removes the first entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQHILR (void *head, void **removed_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQHIQ

This function removes the first entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQHIQ (void *head, void **removed_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQHIQR

This function removes the first entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQHIQR (void *head, void **removed_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQTIL

This function removes the last entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQTIL (void *head, void **removed_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQTILR

This function removes the last entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQTILR (void *head, void **removed_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQTIQ

This function removes the last entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQTIQ (void *head, void **removed_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQTIQR

This function removes the last entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other preprocessors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQTIQR (void *head, void **removed_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

__PAL_REMQUEL

This function removes an entry from a longword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  _PAL_REMQUEL (void *entry, void **removed_entry);
```

entry

A pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

__PAL_REMQUEL_D

This function removes an entry from a longword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_REMQUEL_D (void **entry, void **removed_entry);
/* Deferred */
```

entry

A pointer to a pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

__PAL_REMQUEQ

This function removes an entry from a quadword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_REMQUEQ (void *entry, void **removed_entry);
```

entry

A pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

__PAL_REMQUEQ_D

This function removes an entry from a quadword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int   __PAL_REMQUEQ_D (void **entry, void **removed_entry);
/* Deferred */
```

entry

A pointer to a pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

__PAL_STQP

This function writes the quadword *value* to the memory location pointed to by *address*.

This function has the following format:

```
void  __PAL_STQP (void *address, unsigned __int64 value);
```

address

Memory location to be written to.

value

Quadword value to be stored.

If the location pointed to by *address* is not quadword-aligned, the result is unpredictable.

__PAL_SWASTEN

This function swaps the previous state of the Asynchronous System Trap (AST) enable bit for the new state. The new state is supplied in bit 0 of *new_state_mask*. The previous state is returned, zero-extended.

A check is made to determine if an AST is pending. If the enabling conditions are present for an AST at the completion of this instruction, the AST occurs before the next instruction.

This function has the following format:

```
unsigned int   __PAL_SWASTEN (int new_state_mask);
```

new_state_mask

An integer whose 0 bit is the new state of the AST enable bit.

__PAL_SWPCTX

This function returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.

This function has the following format:

```
void   __PAL_SWPCTX (void *address);
```

address

A pointer to the new HWPCB.

__PAL_WR_PS_SW

This function writes the low-order three bits of *mask* into the Processor Status software field (PS<SW>).

This function has the following format:

```
void  __PAL_WR_PS_SW (int mask);
```

mask

An integer whose low-order three bits are written into PS<SW>.

C.1.4. Other Builtins

Absolute Value (__ABS)

The __ABS built-in is functionally equivalent to its counterpart, `abs`, in the standard header file `<stdlib.h>`.

Its format is also the same:

```
#include <stdlib.h>
int  __ABS (int x);
```

This built-in function does, however, offer performance improvements because there is less call overhead associated with its use.

f you include `<stdlib.h>`, the built-in function is automatically used for all occurrences of `abs`. To disable the built-in function, use `#undef abs`.

Acquire and Release Longword Semaphore (__ACQUIRE_SEM_LONG, __RELEASE_SEM_LONG)

The __ACQUIRE_SEM_LONG and __RELEASE_SEM_LONG functions provide a counted semaphore capability where the positive value of a longword is interpreted as the number of resources available.

The __ACQUIRE_SEM_LONG function loops until the longword has a positive value and then decrements it within a load-locked/store-conditional sequence; it then issues a memory barrier. This function returns 1 if the resource count was successfully decremented within the specified number of retries, and 0 otherwise. With no explicit retry count, the function does not return until it succeeds.

The __RELEASE_SEM_LONG function issues a memory barrier and then does an __ATOMIC_INCREMENT_LONG on the longword.

The __ACQUIRE_SEM_LONG function has the following formats:

```
int  __ACQUIRE_SEM_LONG (volatile void *address);
int  __ACQUIRE_SEM_LONG_RETRY (volatile void *address, int retry);
```

The __RELEASE_SEM_LONG function has the following format:

```
int  __RELEASE_SEM_LONG (volatile void *address);
```

`address`

The longword-aligned address of the resource count.

`retry`

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

Add Aligned Word Interlocked (__ADAWI)

The __ADAWI function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

This function has the following format:

```
int  __ADAWI (short src, short *dest);
```

src

The value to be added to the destination.

dest

A pointer to the destination. The destination must be aligned on a word boundary.

The __ADAWI function returns a simulated VAX processor status longword (PSL).

Add Atomic Longword (__ADD_ATOMIC_LONG)

The __ADD_ATOMIC_LONG function adds the specified expression to the longword data segment pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __ADD_ATOMIC_LONG int fnc(void *, int, ...);
```

address

The address of the data segment.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted. If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

A value of 1 is returned upon successful completion.

Add Atomic Quadword (__ADD_ATOMIC_QUAD)

The __ADD_ATOMIC_QUAD function adds the specified expression to the quadword data segment pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __ADD_ATOMIC_QUAD (void *address, int expression, ...);
```

address

The address of the data segment.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted. If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

A value of 1 is returned upon successful completion.

AND Atomic Longword (__AND_ATOMIC_LONG)

The __AND_ATOMIC_LONG function performs a bit-wise or arithmetic AND of the specified expression with the aligned longword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __AND_ATOMIC_LONG (void *address, int expression, ...);
```

`address`

The longword-aligned address of the data segment.

`expression`

An integer expression.

`...`

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

A value of 1 is returned upon successful completion.

AND Atomic Quadword (__AND_ATOMIC_QUAD)

The __AND_ATOMIC_QUAD function performs a bit-wise or arithmetic AND of the specified expression with the aligned quadword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __AND_ATOMIC_QUAD (void *address, int expression, ...);
```

`address`

The address of the aligned quadword.

`expression`

An integer expression.

`...`

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

A value of 1 is returned upon successful completion.

Atomic Add Longword (__ATOMIC_ADD_LONG)

The __ATOMIC_ADD_LONG function adds the specified expression to the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the addition was performed.

This function has one of the following formats:

```
int  __ATOMIC_ADD_LONG (volatile void *address,
int expression);

int  __ATOMIC_ADD_LONG_RETRY (volatile void *address,
int expression, int retry,
int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Add Quadword (__ATOMIC_ADD_QUAD)

The __ATOMIC_ADD_QUAD function adds the specified expression to the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the addition was performed.

This function has one of the following formats:

```
int  __ATOMIC_ADD_QUAD (volatile void *address,
int expression);

int  __ATOMIC_ADD_QUAD_RETRY (volatile void *address,
int expression, int retry,
int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic AND Longword (__ATOMIC_AND_LONG)

The __ATOMIC_AND_LONG function performs a bit-wise or arithmetic AND of the specified expression with the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_AND_LONG (volatile void *address,
int expression);

int  __ATOMIC_AND_LONG_RETRY (volatile void *address,
int expression, int retry,
int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

`status`

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic AND Quadword (__ATOMIC_AND_QUAD)

The __ATOMIC_AND_QUAD function performs a bit-wise or arithmetic AND of the specified expression with the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_AND_QUAD (volatile void *address,
int expression);

int  __ATOMIC_AND_QUAD_RETRY (volatile void *address,
int expression, int retry,
int *status);
```

`address`

The quadword-aligned address of the data segment.

`expression`

An integer expression.

`retry`

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

`status`

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic OR Longword (__ATOMIC_OR_LONG)

The __ATOMIC_OR_LONG function performs a bit-wise or arithmetic OR of the specified expression with the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_OR_LONG (volatile void *address, int expression);

int  __ATOMIC_OR_LONG_RETRY (volatile void *address, int expression,
                             int retry, int *status);
```

`address`

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic OR Quadword (__ATOMIC_OR_QUAD)

The __ATOMIC_OR_QUAD function performs a bit-wise or arithmetic OR of the specified expression with the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_OR_QUAD (volatile void *address,
int expression);

int  __ATOMIC_OR_QUAD_RETRY (volatile void *address,
int expression,
int retry, int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Increment Longword (__ATOMIC_INCREMENT_LONG)

The __ATOMIC_INCREMENT_LONG function increments by 1 the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_INCREMENT_LONG (volatile void *address);

int  __ATOMIC_INCREMENT_LONG_RETRY (volatile void *address,
int retry, int *status);
```

address

The longword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Increment Quadword (__ATOMIC_INCREMENT_QUAD)

The __ATOMIC_INCREMENT_QUAD function increments by 1 the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_INCREMENT_QUAD (volatile void *address);

__int64  __ATOMIC_INCREMENT_QUAD (volatile void *address,
int retry, int *status);
```

address

The quadword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Decrement Longword (__ATOMIC_DECREMENT_LONG)

The __ATOMIC_DECREMENT_LONG function decrements by 1 the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_DECREMENT_LONG (volatile void *address);

int  __ATOMIC_DECREMENT_LONG_RETRY (volatile void *address, int retry, int
 *status);
```

address

The longword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Decrement Quadword (__ATOMIC_DECREMENT_QUAD)

The __ATOMIC_DECREMENT_QUAD function decrements by 1 the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_DECREMENT_QUAD (volatile void *address);

__int64  __ATOMIC_DECREMENT_QUAD_RETRY (volatile void *address, int retry,
 int *status);
```

address

The quadword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Exchange Longword (__ATOMIC_EXCH_LONG)

The __ATOMIC_EXCH_LONG function stores the value of the specified expression into the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_EXCH_LONG (volatile void *address,
int expression);

int  __ATOMIC_EXCH_LONG_RETRY (volatile void *address,
int expression,
int retry, int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Atomic Exchange Quadword (__ATOMIC_EXCH_QUAD)

The __ATOMIC_EXCH_QUAD function stores the value of the specified expression into the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has one of the following formats:

```
int  __ATOMIC_EXCH_QUAD (volatile void *address,
int expression);

int  __ATOMIC_EXCH_QUAD_RETRY (volatile void *address,
int expression, int retry,
int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Allocate Bytes from Stack (__ALLOCA)

The __ALLOCA function allocates *n* bytes from the stack.

This function has the following format:

```
void  *__ALLOCA (unsigned int n);
```

n

The number of bytes to be allocated.

A pointer to the allocated memory is returned.

Single-Precision, Floating-Point Arithmetic Built-in Functions

The following built-in functions provide single-precision, floating-point chopped arithmetic:

__ADDF_C	__ADDS_C	__SUBF_C	__SUBS_C
__MULF_C	__MULS_C	__DIVF_C	__DIVS_C

They have the following format:

```
float  __op{F,S}_C (float operand1, float operand2);
```

Where *op* is one of ADD, SUB, MUL, DIV, and {F,S} represents VAX or IEEE floating-point arithmetic.

The result of the arithmetic operation is returned.

Double-Precision, Floating-Point Arithmetic Built-in Functions

The following built-in functions provide double-precision, floating-point chopped arithmetic:

__ADDG_C	__ADDT_C	__SUBG_C	__SUBT_C
__MULG_C	__MULT_C	__DIVG_C	__DIVT_C

They have the following format:

```
double  __op{G,T}_C (double operand1, double operand2);
```

Where *op* is one of ADD, SUB, MUL, DIV, and {G,T} represents VAX or IEEE floating-point arithmetic.

The result of the arithmetic operation is returned.

Copy Sign Built-in Functions

Built-in functions are provided to copy selected portions of single- and double-precision, floating-point numbers.

These built-in functions have the following format:

```
float  __CPYSF (float operand1, float operand2);
double  __CPYS (double operand1, double operand2);

float  __CPYSNF (float operand1, float operand2);
double  __CPYSN (double operand1, double operand2);

float  __CPYSEF (float operand1, float operand2);
double  __CPYSE (double operand1, double operand2);
```

The copy sign built-in functions (__CPYSF and __CPYS) fetch the sign bit in *operand1*, concatenate it with the exponent and fraction bits from *operand2*, and return the result.

The copy sign negate built-in functions (__CPYSNF and __CPYSN) fetch the sign bit in *operand1*, complement it, concatenate it with the exponent and fraction bits from *operand2*, and return the result.

The copy sign exponent built-in functions (__CPYSEF and __CPYSE) fetch the sign and exponent bits from *operand1*, concatenate them with the fraction bits from *operand2*, and return the result.

Compare Store Longword (__CMP_STORE_LONG)

The __CMP_STORE_LONG function has the following format:

```
int  __CMP_STORE_LONG (void *source, int old_value,
int new_value, void *dest);
```

This function compares the value pointed to by *source* with the longword *old_value*. If they are equal, the longword *new_value* is stored into the value pointed to by *dest*. The comparison is within a load-locked/store-conditional code sequence.

The function returns 0 if the two values are unequal, if *source* and *dest* are not in the same 16-byte lock region, or if some other process accessed the 16-byte lock region before *new_value* could be stored. The function returns 1 if the two values are equal and *new_value* was stored atomically.

Compare Store Quadword (__CMP_STORE_QUAD)

The __CMP_STORE_QUAD function has the following format:

```
int  __CMP_STORE_QUAD (void *source, __int64 old_value,
__int64 new_value, void *dest);
```

This function compares the value pointed to by *source* with the quadword *old_value*. If they are equal, the quadword *new_value* is stored into the value pointed to by *dest*. The comparison is within a load-locked/store-conditional code sequence.

The function returns 0 if the two values are unequal, if *source* and *dest* are not in the same 16-byte lock region, or if some other process accessed the 16-byte lock region before *new_value* could be stored. The function returns 1 if the two values are equal and *new_value* was stored atomically.

Cosine (__COS)

The __COS built-in function is functionally equivalent to its counterpart, `cos`, in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
```

```
double  __COS (double x);
```

x

A radian value.

This built-in function does, however, offer performance improvements because there is less call overhead associated with its use.

If you include <math.h>, the built-in function is automatically used for all occurrences of cos. To disable the built-in function, use #undef cos.

Convert G_Floating to F_Floating Chopped (__CVTGF_C)

The __CVTGF_C function converts a double-precision, VAX G_floating-point number to a single-precision, VAX F_floating-point number. This conversion chops to single-precision; then the 8-bit exponent range is checked for overflow or underflow.

This function has the following format:

```
float  __CVTGF_C (double operand);
```

operand

A double-precision, VAX floating-point number.

Convert G-Floating to Quadword (__CVTGQ)

The __CVTGQ function rounds a double-precision, VAX floating-point number to a 64-bit integer value and returns the result.

This function has the following format:

```
__int64  __CVTGQ (double operand);
```

operand

A double-precision, VAX floating-point number.

Convert IEEE T_Floating to IEEE S_Floating Chopped (__CVTTS_C)

The __CVTTS_C function converts a double-precision, IEEE T_floating-point number to a single-precision, IEEE S_floating-point number. This conversion chops to single-precision; then the 8-bit exponent range is checked for overflow or underflow.

This function has the following format:

```
float  __CVTTS_C (double operand);
```

operand

A double-precision, IEEE floating-point number.

Convert IEEE T-Floating to Quadword (__CVTTQ)

The __CVTTQ function rounds a double-precision, IEEE-floating-point number to a 64-bit integer value and returns the result.

This function has the following format:

```
__int64  __CVTTQ (double operand);
```

operand

A double-precision, IEEE T-floating-point number.

Floating-Point Absolute Value (__FABS)

The __FABS built-in function is functionally equivalent to its counterpart, `fabs`, in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
double@@__FABS (double x);
```

x

A floating-point number.

This built-in function does, however, offer performance improvements because there is no call overhead associated with its use.

If you include `<math.h>`, the built-in function is automatically used for all occurrences of `fab`. To disable the built-in function, use `#undef fab`.

Test for Bit Clear then Clear Bit Interlocked (__INTERLOCKED_TESTBITCC_QUAD)

The __INTERLOCKED_TESTBITCC_QUAD function performs the following functions in interlocked fashion:

1. Returns the complement of the specified bit before being cleared.
2. Clears the bit.

This function has the following formats:

```
int  __INTERLOCKED_TESTBITCC_QUAD (volatile void *address,
 int bit_position);

int  __INTERLOCKED_TESTBITCC_QUAD_RETRY (volatile void *address,
 int bit_position, int retry, int *status);
```

address

The quadword-aligned base address of the bit field.

bit_position

The position within the field of the bit that you want cleared, in the range of 0 to 63.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Test for Bit Set Then Set Bit Interlocked (__INTERLOCKED_TESTBITSS_QUAD)

The __INTERLOCKED_TESTBITSS_QUAD function performs the following functions in interlocked fashion:

1. Returns the value of the specified bit before being set.
2. Sets the bit.

This function has the following formats:

```
int  __INTERLOCKED_TESTBITSS_QUAD (volatile void *address,
 int bit_position);

int  __INTERLOCKED_TESTBITSS_QUAD_RETRY (volatile void *address,
int expression, int retry, int *status);
```

address

The quadword-aligned base address of the bit field.

bit_position

The position within the field of the bit that you want cleared, in the range of 0 to 63.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

_leadz

The _leadz built-in function returns the number of leading zeroes (starting at the most significant bit position) in its argument. For example, _leadz(1) returns 63, and _leadz(0) returns 64.

This function has the following format:

```
__int64 _leadz (unsigned __int64);
```

Lock and Unlock Longword (__LOCK_LONG, __UNLOCK_LONG)

The __LOCK_LONG and __UNLOCK_LONG functions provide a binary spinlock capability based on the low-order bit of a longword.

The __LOCK_LONG function executes in a loop waiting for the bit to be cleared and then sets it within a load-locked/store-conditional sequence; it then issues a memory barrier. The __UNLOCK_LONG function issues a memory barrier and then zeroes the longword.

The __LOCK_LONG_RETRY function returns 1 if the lock was acquired in the specified number of retries and 0 if the lock was not acquired.

The __LOCK_LONG function has the following formats:

```
int  __LOCK_LONG (volatile void *address);

int  __LOCK_LONG_RETRY (volatile void *address, int retry);
```

The __UNLOCK_LONG function has the following format:

```
int  __UNLOCK_LONG (volatile void *address);
```

address

The quadword-aligned address of the longword used for the lock.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

Longword Absolute Value (__LABS)

The __LABS built-in function is functionally equivalent to its counterpart, `labs`, in the standard header file `<stdlib.h>`.

Its format is also the same:

```
#include <stdlib.h>
long int@@__LABS (long int x);
```

x

An integer.

This built-in function does, however, offer performance improvements because there is less call overhead associated with its use.

If you include `<stdlib.h>`, the built-in function is automatically used for all occurrences of `labs`. To disable the built-in function, use `#undef labs`.

Memory Barrier (__MB)

The __MB function directs the compiler to generate a memory barrier instruction.

This function has the following format:

```
void  __MB (void);
```

Memory Copy and Set Functions (__MEMCPY, __MEMMOVE, __MEMSET)

The __MEMCPY, __MEMMOVE, and __MEMSET built-in functions are functionally equivalent to their counterparts in the standard header file `<string.h>`.

Their format is also the same:

```
#include <string.h>

void  *__MEMCPY (void *s1, const void *s2, size_t size);

void  *__MEMMOVE (void *s1, const void *s2, size_t size);

void  *__MEMSET (void *s, int value, size_t size);
```

These built-in functions do, however, offer performance improvements because there is less call overhead associated with their use.

If you include `<string.h>`, the built-in functions are automatically used for all occurrences of `memcpy`, `memmove`, and `memset`. To disable the built-in functions, use `#undef memcpy`, `#undef memmove`, and `#undef memset`.

_popcnt

The _popcnt built-in function returns the number of "1" bits (0 to 64) in its argument. For example, _popcnt(12) returns 2.

This function has the following format:

```
__int64 _popcnt (unsigned __int64);
```

_poppar

The _poppar built-in function returns 1 if the number of "1" bits in its argument is odd; otherwise it returns 0. For example, _poppar(12) returns 0.

This function has the following format:

```
__int64 _poppar (unsigned __int64);
```

Read Process Cycle Counter (__RPCC)

The __RPCC function reads the current process cycle counter.

This function has the following format:

```
__int64  __RPCC (void);
```

Sine (__SIN)

The __SIN built-in function is functionally equivalent to its counterpart in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
double@@__SIN (double x);
```

x

A radian value.

This built-in function does, however, offer performance improvements because there is less call overhead associated with its use.

If you include `<math.h>`, the built-in function is used automatically for all occurrences of `sin`. To disable the built-in function, use `#undef sin`.

Test for Bit Clear then Clear Bit Interlocked (__TESTBITCCI)

The __TESTBITCCI function performs the following operations in interlocked fashion:

- Returns the complement of the specified bit before being cleared
- Clears the bit

This function has the following format:

```
int  __TESTBITCCI (void *address, int position, ...);
```

address

The base address of the field.

position

The position within the field of the bit that you want cleared.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted. If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

Test for Bit Set then Set Bit Interlocked (__TESTBITSSI)

The __TESTBITSSI function performs the following operations in interlocked fashion:

- Returns the value of the specified bit before being set
- Sets the bit

This function has the following format:

```
int  __TESTBITSSI (void *address, int position, ...);
```

address

The base address of the field.

position

The position within the field of the bit that you want set.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted. If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned.

_trailz

The _trailz built-in function returns the number of trailing zeros (counting after the least significant set bit to the least significant bit position) in its argument. For example, _trailz(2) returns 1, and _trailz(0) returns 64.

This function has the following format:

```
__int64 _trailz (unsigned __int64);
```

Trap Barrier Instruction (__TRAPB)

The __TRAPB function allows software to guarantee that, in a pipeline implementation, all previous arithmetic instructions will be completed without incurring any arithmetic traps before any instructions after the TRAPB instruction are issued.

This function has the following format:

```
void  __TRAPB (void);
```

Unsigned Quadword Multiply High (__UMULH)

The __UMULH function performs a quadword multiply high instruction.

This function has the following format:

```
unsigned __int64  __UMULH (unsigned __int64 operand1, unsigned
 __int64 operand2);
```

operand1

A 64-bit unsigned integer.

operand2

A 64-bit unsigned integer.

The two operands are multiplied as unsigned integers to produce a 128-bit result. The high order 64-bits are returned.

C.2. Built-In Functions for I64 Systems *(I64 only)*

The VSI C++ built-in functions available on OpenVMS Alpha systems are also available on I64 systems, with some differences, as described in this section. This section also describes built-in functions that are specific to I64 systems.

C.2.1. Builtin Differences on I64 Systems

The <builtins.h> header file contains comments noting which built-in functions are not available or are not the preferred form for I64 systems. The compiler issues diagnostics where using a different built-in function for I64 systems 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 I64 systems:

- 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 I64 systems:

  ```
  __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 I64 systems). 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 I64 system. Two notable exceptions are __PAL_GENTRAP and __PAL_BUGCHK, which instead invoke the I64 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).

- For most built-in functions that take a retry count, the compiler issues a an error message. Such builtins must be replaced with the corresponding builtin that does not have a retry count. This is necessary because the retry behavior allowed by Alpha load-locked/store-conditional sequences does not exist on I64 systems. 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 produce an error message, and must be replaced with the new __CMP_SWAP_LONG and __CMP_SWAP_QUAD built-in functions.

C.2.2. Built-in Functions Specific to I64 Systems

The `<builtins.h>` header file contains a section at the top conditionalized to just __ia64 with the support for built-in functions specific to I64 systems. 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 sections describe the VSI C++ built-in functions available on OpenVMS I64 systems.

Get Hardware Register Value (__getReg)

The __getReg function gets the value from a hardware register based on the register index specified. This function produces a corresponding `mov = r` instruction.

This function has the following format:

```
unsigned __int64   __getReg (const int whichReg);
```

whichReg

The index of the hardware register from which the value is obtained. The __getReg and __setReg functions can access the following registers:

```
Register Name          whichReg
_IA64_REG_IP             1016
_IA64_REG_PSR            1019
_IA64_REG_PSR_L          1019

General Integer Registers:

Register Name          whichReg
_IA64_REG_GP             1025
_IA64_REG_SP             1036
_IA64_REG_TP             1037

Application Registers:

Register Name          whichReg
_IA64_REG_AR_KR0         3072
_IA64_REG_AR_KR1         3073
_IA64_REG_AR_KR2         3074
_IA64_REG_AR_KR3         3075
_IA64_REG_AR_KR4         3076
_IA64_REG_AR_KR5         3077
_IA64_REG_AR_KR6         3078
_IA64_REG_AR_KR7         3079
_IA64_REG_AR_RSC         3088
_IA64_REG_AR_BSP         3089
_IA64_REG_AR_BSPSTORE    3090
_IA64_REG_AR_RNAT        3091
_IA64_REG_AR_FCR         3093
_IA64_REG_AR_EFLAG       3096
_IA64_REG_AR_CSD         3097
```

```
_IA64_REG_AR_SSD        3098
_IA64_REG_AR_CFLAG      3099
_IA64_REG_AR_FSR        3100
_IA64_REG_AR_FIR        3101
_IA64_REG_AR_FDR        3102
_IA64_REG_AR_CCV        3104
_IA64_REG_AR_UNAT       3108
_IA64_REG_AR_FPSR       3112
_IA64_REG_AR_ITC        3116
_IA64_REG_AR_PFS        3136
_IA64_REG_AR_LC         3137
_IA64_REG_AR_EC         3138

Control Registers:

Register Name          whichReg
_IA64_REG_CR_DCR        4096
_IA64_REG_CR_ITM        4097
_IA64_REG_CR_IVA        4098
_IA64_REG_CR_PTA        4104
_IA64_REG_CR_IPSR       4112
_IA64_REG_CR_ISR        4113
_IA64_REG_CR_IIP        4115
_IA64_REG_CR_IFA        4116
_IA64_REG_CR_ITIR       4117
_IA64_REG_CR_IIPA       4118
_IA64_REG_CR_IFS        4119
_IA64_REG_CR_IIM        4120
_IA64_REG_CR_IHA        4121
_IA64_REG_CR_LID        4160
_IA64_REG_CR_IVR        4161 *
_IA64_REG_CR_TPR        4162
_IA64_REG_CR_EOI        4163
_IA64_REG_CR_IRR0       4164 *
_IA64_REG_CR_IRR1       4165 *
_IA64_REG_CR_IRR2       4166 *
_IA64_REG_CR_IRR3       4167 *
_IA64_REG_CR_ITV        4168
_IA64_REG_CR_PMV        4169
_IA64_REG_CR_CMCV       4170
_IA64_REG_CR_LRR0       4176
_IA64_REG_CR_LRR1       4177

* getReg only
```

Set Hardware Register Value (__setReg)

The __setReg function sets the value for a hardware register based on the register index specified. This function produces a corresponding `mov = r` instruction.

This function has the following format:

```
void   __setReg (const int whichReg, unsigned __int64 value);
```

whichReg

The index of the hardware register whose value is being set. See the __getReg functions for the list of registers that can be accessed.

value

The value to which the register is set.

Get Index Register Value (__getIndReg)

The __getIndReg function returns the value of an indexed register. The function accesses a register (*index*) in a register file (*whichIndReg*) of 64-bit registers.

This function has the following format:

```
unsigned __int64   __getIndReg
(const int whichIndReg, __int64 index);
```

whichIndReg

The register file.

index

The index in the register file of the hardware register whose value is being requested. See the __getReg functions for the list of registers that can be accessed.

Indirect Registers for getIndReg and setIndReg:

```
Register Name          whichReg

_IA64_REG_INDR_CPUID    9000 *
_IA64_REG_INDR_DBR      9001
_IA64_REG_INDR_IBR      9002
_IA64_REG_INDR_PKR      9003
_IA64_REG_INDR_PMC      9004
_IA64_REG_INDR_PMD      9005
_IA64_REG_INDR_RR       9006
_IA64_REG_INDR_RESERVED 9007

* getIndReg only
```

Set Index Register Value (__setIndReg)

The __setIndReg function copies a value into an indexed register. The function accesses a register (*index*) in a register file (*whichIndReg*) of 64-bit registers.

This function has the following format:

```
void   __setIndReg (const int whichIndReg, __int64 index,
unsigned __int64 value);
```

whichIndReg

The register file.

index

The index in the register file of the hardware register to be set. See the __getIndReg function for the list of registers that can be accessed.

value

The value to which the register is set.

Generate Break Instruction (__break)

The __break function generates a break instruction with an immediate.

This function has the following format:

```
void   __break (const int __break_arg);
```

__break_arg

An immediate value for the __break instruction to use.

Serialize Data (__dsrlz)

The __dsrlz function serializes data. Maps to the `srlz.d` instruction.

This function has the following format:

```
void   __dsrlz (void);
```

Flush Cache Instruction (__fc)

The __fc function flushes a cache line associated with the address given by the argument. Maps to the `fcr` instruction.

This function has the following format:

```
void  __fc (__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Flush Write Buffers (__fwb)

The __fwb function flushes the write buffers. Maps to the `fwb` instruction.

This function has the following format:

```
void  __fwb (void);
```

Invalidate ALAT (__invalat)

The __invalat function invalidates ALAT. Maps to the `invala` instruction.

This function has the following format:

```
void  __invalat (void);
```

Invalidate ALAT (__invala)

The __invala function is the same as the __invalat function.

Execute Serialize (__isrlz)

The __isrlz function executes the serialize instruction. Maps to the `srlz.i` instruction.

This function has the following format:

```
void  __isrlz (void);
```

Insert Data Address Translation Cache (__itcd)

The __itcd function inserts an entry into the data translation cache. Maps to the `itc.d` instruction.

This function has the following format:

```
void  __itcd (__int64 pa);
```

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Insert Instruction Address Translation Cache (__itci)

The __itci function inserts an entry into the instruction translation cache. Maps to the `itc.i` instruction.

This function has the following format:

```
void  __itci (__int64 pa);
```

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Insert Data Translation Register (__itrd)

The __itrd function maps to the `itr.d` instruction.

This function has the following format:

```
void  __itrd (__int64 whichTransReg, __int64 pa);
```

whichTransReg

The data translation register to be used by the itr.d instruction.

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Insert Instruction Translation Register (__itri)

The __itri function maps to the `itr.i` instruction.

This function has the following format:

```
void   __itri (__int64 whichTransReg, __int64 pa);
```

whichTransReg

The data translation register to be used by the itr.i instruction.

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Purge Translation Cache Entry (__ptce)

The __ptce function maps to the `ptc.e` instruction.

This function has the following format:

```
void  __ptce (__int64 va);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Purge Global Translation Cache (__ptcg)

The __ptcg function purges the global translation cache. Maps to the `ptc.g r,r` instruction.

This function has the following format:

```
void  __ptcg (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

Purge Local Translation Cache (__ptcl)

The __ptcl function purges the local translation cache. Maps to the `ptc.l r,r` instruction.

This function has the following format:

```
void  __ptcl (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

Purge Global Translation Cache and ALAT (__ptcga)

The __ptcga function purges the global translation cache and ALAT. Maps to the `ptc.ga r,r` instruction.

This function has the following format:

```
void  __ptcga (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

Purge Data Translation Register (__ptrd)

The __ptrd function purges the data translation register. Maps to the `ptr.d r,r` instruction.

This function has the following format:

```
void  __ptrd (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

Purge Instruction Translation Register (__ptri)

The __ptri function purges the instruction translation register. Maps to the `ptr.i r,r` instruction.

This function has the following format:

```
void  __ptri (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

Reset System Mask (__rsm)

The __rsm function resets the system mask bits of the PSR. Maps to the `rsm imm24` instruction.

This function has the following format:

```
void  __rsm (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

Reset User Mask (__rum)

The __rum function resets the user mask.

This function has the following format:

```
void  __rum (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

Set System Mask (__ssm)

The __ssm function sets the system mask.

This function has the following format:

```
void  __ssm (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

Set User Mask (__sum)

The __sum function sets the user mask bits of the PSR. Maps to the `sum imm24` instruction.

This function has the following format:

```
void  __sum (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

Enable Memory Synchronization (__synci)

The __synci function enables memory synchronization. Maps to the `sync.i` instruction.

This function has the following format:

```
void  __synci (void);
```

Translation Hashed Entry Address (__thash)

The __thash function generates a translation hash entry address. Maps to the `thash r = r` instruction.

This function has the following format:

```
void   __thash(__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Translation Hashed Entry Tag (__ttag)

The __ttag function generates a translation hash entry tag. Maps to the `ttag r=r` instruction.

This function has the following format:

```
void  __ttag(__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

Atomic Compare and Exchange (_InterlockedCompareExchange_acq)

The _InterlockedCompareExchange_acq function atomically compares and exchanges the value specified by the first argument (a 64-bit pointer). This function maps to the `cmpxchg4.acq` instruction with appropriate setup.

This function has the following format:

```
unsigned __int64   _InterlockedCompareExchange_acq (volatile unsigned int
 *Destination,
                   unsigned __int64 Newval, unsigned __int64 Comparand);
```

The value at **Destination* is compared with the value specified by *Comparand*. If they are equal, *Newval* is written to **Destination*, and *Oldval* is returned. The exchange will have taken place if the value returned is equal to the *Comparand*. The following algorithm is used:

```
ar.ccv = Comparand;
Oldval = *Destination;          //Atomic
if (ar.ccv == *Destination)     //Atomic
    *Destination = Newval;      //Atomic
return Oldval;
```

Those parts of the algorithm that are marked "Atomic" are performed atomically by the cmpxchg4.acq instruction. This instruction has acquire ordering semantics; that is, the memory read/write is made visible prior to all subsequent data memory accesses of the *Destination* by other processors.

`Destination`

The value to be compared with *Comparand* and, if equal, replaced with the value of *Newval*.

`Newval`

The new value to replace the value in *Destination*.

`Comparand`

The value with which to compare *Destination*.

Atomic Compare and Exchange (_InterlockedCompareExchange64_acq)

The _InterlockedCompareExchange64_acq function is the same as the _InterlockedCompareExchange_acq function, except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg8.acq instruction.

This function has the following format:

```
unsigned __int64   _InterlockedCompareExchange64_acq (volatile unsigned
 __int64 *Destination,
                          unsigned __int64 Newval, unsigned
 __int64 Comparand);
```

Atomic Compare and Exchange (_InterlockedCompareExchange_rel)

This function is the same as the _InterlockedCompareExchange_acq function except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg4.rel instruction with release ordering semantics; that is, the memory read/write is made visible after all previous memory accesses of the *Destination* by other processors.

This function has the following format:

```
unsigned __int64   _InterlockedCompareExchange_rel (volatile unsigned int
 *Destination,
                          unsigned __int64 Newval, unsigned
 __int64 Comparand);
```

Atomic Compare and Exchange (_InterlockedCompareExchange64_rel)

This function is the same as the _InterlockedCompareExchange_rel function, except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg8.rel instruction.

This function has the following format:

```
unsigned __int64   _InterlockedCompareExchange64_rel (volatile unsigned
 __int64 *Destination,
                  unsigned __int64 Newval, unsigned __int64 Comparand);
```

Conditional Atomic Compare and Exchange Longword (__CMP_SWAP_LONG)

The __CMP_SWAP_LONG function performs a conditional atomic compare and exchange operation on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG (volatile void *source, int old_value,
 int new_value);
```

`source`

The longword value to be compared with *old_value*.

`old_value`

The longword value *source* is compared with.

`new_value`

The longword value written into *source* if *source* and *old_value* are equal.

Conditional Atomic Compare and Exchange Quadword (__CMP_SWAP_QUAD)

The __CMP_SWAP_QUAD function performs a conditional atomic compare and exchange operation on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD (volatile void *source, int old_value,
 int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written to *source* if *source* and *old_value* are equal.

Conditional Atomic Compare and Exchange Longword with Acquire Semantics (__CMP_SWAP_LONG_ACQ)

The __CMP_SWAP_LONG_ACQ function performs a conditional atomic compare and exchange operation with acquire semantics on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Acquire memory ordering guarantees that the memory read/write is made visible *before* all subsequent data accesses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG_ACQ (volatile void *source, int old_value,
 int new_value);
```

source

The longword value to be compared with *old_value*.

old_value

The longword value *source* is compared with.

new_value

The longword value written into *source* if *source* and *old_value* are equal.

Conditional Atomic Compare and Exchange Quadword with Acquire Semantics (__CMP_SWAP_QUAD_ACQ)

The __CMP_SWAP_QUAD_ACQ function performs a conditional atomic compare and exchange operation with acquire semantics on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Acquire memory ordering guarantees that the memory read/write is made visible *before* all subsequent memory data accesses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD_ACQ (volatile void *source, int old_value,
 int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written into *source* if *source* and *old_value* are equal.

Conditional Atomic Compare and Exchange Longword with Release Semantics (__CMP_SWAP_LONG_REL)

The __CMP_SWAP_LONG_REL function performs a conditional atomic compare and exchange operation with release semantics on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Release memory ordering guarantees that the memory read/write is made visible *after* all previous data memory acceses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG_REL (volatile void *source, int old_value,
 int new_value);
```

source

The longword value to be compared with *old_value*.

old_value

The longword value *source* is compared with.

new_value

The longword value written into *source* if *source* and *old_value* are equal.

Conditional Atomic Compare and Exchange Quadword with Release Semantics (__CMP_SWAP_QUAD_REL)

The __CMP_SWAP_QUAD_REL function performs a conditional atomic compare and exchange operation with release semantics on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Release memory ordering guarantees that the memory read/write is made visible *after* all previous data memory acceses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD_REL (volatile void *source, int old_value,
 int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written into *source* if *source* and *old_value* are equal.

Return Address (__RETURN_ADDRESS)

The __RETURN_ADDRESS function produces the address to which the function containing the built-in call will return as a 64-bit integer (on Alpha systems, the value of R26 on entry to the function; on I64 systems, the value of B0 on entry to the function).

This built-in function cannot be used within a function specified to use nonstandard linkage.

This function has the following format:

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

Implement Alpha __PAL_GENTRAP and __PAL_BUGCHK Builtins (__break2)

The __break2 function is used to implement the Alpha __PAL_GENTRAP and __PAL_BUGCHK built-in functions on OpenVMS I64 systems.

The __break2 function is equivalent to the __break function with the second parameter passed in general register 17:

```
R17 = <double_uscore>R17_value; <double_uscore>break
 (<double_uscore>break_code);
```

This function has the following format:

```
void  __break2 (__Integer_Constant __break_code, unsigned
 __int64 __r17_value);
```

__breakcode

The particular software condition that has occurred.

__r17_value

The value of R17, a volatile general register reserved by the OpenVMS Itanium calling standard for use by compiled code to communicate with specialized compiler support routines that require out-of-band information passing.

Flush Register Stack (__flushrs)

The __flushrs function flushes the register stack.

This function has the following format:

```
void  __flushrs (void);
```

Load Register Stack (__loadrs)

The __loadrs function loads the register stack.

This function has the following format:

```
void  __loadrs (void);
```

Probe Read-Access Permission (__prober)

The __prober function determines whether read access to the virtual address specified by *__address* bits {60:0} and the region register indexed by *__address* bits {63:61} is permitted at the privilege level given by *__mode* bits {1:0}. It returns 1 if the access is permitted, and 0 otherwise.

This function can probe only with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

This function is the same as the Intel __probe_r function.

This function has the following format:

```
int  __prober (__int64 __address, unsigned int __mode);
```

__address

Virtual address for which read-access permission is being checked.

__mode

Privilege level for which read-access permission is being checked.

Probe Write-Access Permission (__probew)

The __probew function determines whether write access to the virtual address specified by *__address* bits {60:0} and the region register indexed by *__address* bits {63:61}, is permitted at the privilege level given by *__mode* bits {1:0}. It returns 1 if the access is permitted, and 0 otherwise.

This function can probe only with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

This function is the same as the Intel __probe_w function.

This function has the following format:

```
int  __probew (__int64 __address, unsigned int __mode);
```

__address

Virtual address for which write-access permission is being checked.

__mode

Privilege level for which write-access permission is being checked.

Translation Access Key (__tak)

The __tak function returns the translation access key.

This function has the following format:

```
unsigned int  __tak (__int64 __address);
```

__address

Virtual address for translation key is being returned.

Translate to Physical Address (__tpa)

The __tpa function translates a virtual address to a physical address.

This function has the following format:

```
__int64  __tpa(__int64 __address);
```

__address

Virtual address to be translated.

Appendix D. Class Library Restrictions

This appendix describes known problems and restrictions for the Class Library. Please note that **String Package**, which is part of the Class Library, is entirely different from the String class that is part of the newly-implemented C++ Standard Library and known as the **String Library**. Do not confuse these two contrasting implementations.

D.1. Class Library Restrictions

The following are restrictions in the C++ Class Library:

- No Class Library support for 128-bit long doubles

  The Class Library does not include support for 128-bit long doubles.

- Conflict with redefinition of `clear()`

  If your program includes both `<curses.h>` and `<iostream.hxx>`, VSI C++ might fail to compile your program because `clear()` is defined by both header files. In `<curses.h>`, `clear()` is defined as a macro whereas in `<iostream.hxx>` `clear()` is defined as a member function.

  Workarounds:

  If your program does not use either `clear()` or uses the `clear()`, include the `<iostream.hxx>` header first, followed by `<curses.h>`.

  If your program uses the `ios::clear()` function, undefine the `clear()` macro directly after the `#include <curses.h>` statement.

- On OpenVMS Alpha systems, class library IOStreams do not support denormalized IEEE numbers. The workaround is to use C Run-Time Library functions like `printf` and `scanf` instead.

Appendix E. Compiler Compatibility

This appendix describes VSI C++ compatibility with other C++ compilers, and documents compatibility concerns between the Version 5.*n* and Version 6.*n* compilers.

For porting and compatibility between Alpha and I64 systems, see Chapter 4.

VSI C++ implements the C++ International Standard, with some differences, as described in the C++ release notes.

This language differs significantly from *The Annotated C++ Reference Manual*, implemented by the Version 5.*n* compilers. When switching from a Version 5.*n* compiler, you might need to modify your source files, especially if you use the default language mode. In addition, language changes can affect the run-time behavior of your programs. If you want to compile existing source code with minimal source changes, compile using the /STANDARD=ARM qualifier option. See Chapter 7 for information on and changes to the Standard Library.

This chapter describes ways to avoid having the compiler reject program code that previously worked with other C++ implementations that adhere less strictly to the C++ language definition. References to applicable portions of *The Annotated C++ Reference Manual* indicate where you can find additional help.

E.1. Compatibility with Other C++ Compilers

In default mode (/STANDARD=RELAXED), the compiler implements most features of the C++ International Standard, including:

- Run-time type identification (RTTI), with `dynamic_cast` and the `typeid` operator (see Section 2.4)
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`)
- Array new and delete

For compatibility with previous versions, the compiler provides the following language mode options:

/STANDARD=RELAXED

Specify this option if you want an ANSI C++ compiler that supports some commonly used extensions and is somewhat less strict than the standard. This is the default compiler mode. Please note that /STANDARD=ANSI is accepted as a synonym for /STANDARD=RELAXED to be compatible with previous compiler versions.

If you want to use RELAXED mode but find that the compiler generates too many diagnostics in that mode, you can use the /QUIET option with the /STANDARD=RELAXED option. The /QUIET option relaxes error checking and suppresses or reduces the severity of many diagnostics. It also suppresses many warnings that are generated in RELAXED mode but were not issued by Version 5.*n* compilers. For information on message control options, see Section 2.5.

/STANDARD=ARM

Specify this option if you want to compile programs developed using Version 5.*n* and want to minimize source changes.

VSI C++ Version 6.*n* and higher also provides support for other C++ dialects and language modes. You can specify the following options:

/STANDARD=MS

Specify this option if you want the compiler to accept additional Microsoft Visual C++ extensions.

With /STANDARD=MS you may also want to specify /QUIET to reduce the number of diagnostic messages generated.

/STANDARD=STRICT_ANSI

Enforce the ANSI standard strictly but permit some ANSI violations that should be errors to be warnings. To force ANSI violations to be issued with Error instead of Warning severity, use /WARNINGS=ANSI_ERRORS in addition to /STANDARD=STRICT_ANSI.

/STANDARD=LATEST

Use the latest C++ standard dialect. /STANDARD=LATEST is currently equivalent to /STANDARD=STRICT_ANSI, but is subject to change when newer versions of the C++ standard are released.

E.2. Compatibility with Version 5.6 and Earlier

This section provides details about differences between the Version 6.*n* and later compilers, and the Version 5.6 and earlier compilers:

- Language differences
- Implementation differences
- Library differences

E.2.1. Language Differences

Users should be aware of the following language differences between Version 6.*n* and higher (denoted simply as Version 6.*n* in the following list), and previous versions of the compiler.

- The most important language differences result from the current implementation of the C++ International Standard in the Version 6.*n* compilers. If you want to compile existing source code with minimal source changes, compile using the /STANDARD=ARM option.
- Because the Version 6.*n* compilers perform more error checking than previous versions, they generate significantly more diagnostic messages. However, you can use the /QUIET option to relax error checking and reduce the severity of many diagnostics. *Message Control and Information Options*.
- The following keywords, introduced with the C++ International Standard, are always reserved keywords in all compiler modes:

  ```
  bool, const_cast, explicit, export, false, mutable, dynamic_cast,
  reinterpret_cast, static_cast, true, typeid, typename, wchar_t
  ```

  Alternative representation keywords are as follows:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
```

- Taking the address of a bit field is not allowed in the current version.
- Creation of temporaries and their lifetimes vary among compiler modes.
- Macro expansion in pragmas can give different results in the current and previous versions.
- The following are distinct types in the Version 6.*n* compilers; they were the same type in previous versions:

```
typedef void (*PF)();              // Pointer to an extern "C++" function
extern "C" typedef void (*PCF)(); // Pointer to an extern "C" function
void f(PF);
void f(PCF);
```

- Version 6.*n* does not allow converting a pointer to member from a derived class to a virtual base class.
- Calling a nonstatic member function through a null pointer is undefined behavior. Certain cases that used to run without errors in previous versions no longer run in the current version. For example:

```
#include <iostream.h>
struct A {
    int a;
};
struct D : public virtual A
{
    A* toA(){ return (A*) this; }
};
main ()
{
    D* d = NULL;
    A* ad = d->toA(); // will ACCVIO
    if (ad==NULL) cout << "ok";
}
```

- In Version 6.*n* compilers, `bool` is a built-in type. In previous versions, it is user-defined, typically as `int` in system header files. Mangling differs in this respect only for functions that have arguments of type `bool`.

  In Version 6.*n* compilers, the size of `bool` is 1. In previous versions, `bool` is user defined, typically as `int` with a size of 4.

  In Version 6.*n* compilers, the size of a boolean expression `( sizeof(a && b) )` is 1. In previous versions, the size is 4, independent of the size of `bool`.
- Version 6.*n* compilers do not cause pragmas to become effective within function bodies when scanning template definitions.
- Version 6.*n* compilers do not allow the “virtual” storage class modifier to be used with member function definitions outside a class.
- Version 6.*n* compilers do not allow declaration of pointers to members of type `void`. For example, the following is not allowed:

```
typedef void Z::* any_ptom;
```

E.2.2. Implementation Differences

Users should be aware of the following implementation differences between Version 6.*n* compilers, and previous versions of the compiler:

- The automatic template instantiation model is different for Version 6.*n*, and previous compiler versions. See Section E.2.3 for details.

  It is different yet again for I64 systems. SeeChapter 5 for details.

- Version 6.*n* and higher drops qualifiers on parameters when determining the function type, as dictated by the C++ International Standard. For instance, in the following example, the function declarations are the same function.

  ```
  f(const int p1);
  f(int p1);
  ```

  For compatibility with previous versions, if qualifiers are included in function declarations, they are also included in the mangled name. (Note: this is not true for model ANSI or for I64 systems.)

- Version 6.*n* differs from previous versions in interpreting undefined behavior, as when incrementing takes effect in this example:

  ```
  f(i++, i++);
  ```

- Version 6.*n* cannot handle a `#pragma define_template` that spans multiple lines without the backslash (\ ) delimiter. Version 5.6 can handle this without problems.

- Version 6.*n* displays `#line number` in /PREPROCESS_ONLY output. The previous version displays `#number`.

- After encountering an illegal multibyte character sequence, Version 6.*n* issues a warning diagnostic and continues processing. The previous version issues an error and stops processing.

- Version 6.*n* does not support VAX C module include syntax (for example, `#include acms $submitter` without <> or `" "` delimiters). The compiler searches text libraries for modules included using the normal include syntax (specifying the `" "` or <> delimiters) and correctly (according to the C++ standard) rejects `#include` directives that do not follow this syntax.

E.2.3. Using Templates

The template instantiation model was completely redesigned for C++ Version 6.0. The changes include:

- Automatic template instantiation now occurs at compile time. Necessary templates are instantiated automatically by the compilation of the source file that needs them and has access to the template definitions.

- During automatic template instantiation, instantiations are written into the repository as object files. Compilation of instantiations in no longer done at link time.

- For automatic instantiation, the compiler no longer requires that template declarations and definitions appear in header files.

- `Template.map` files are no longer supported as a way to match template declarations and definitions.

- Several new manual template instantiation pragmas have been added.

The automatic template instantiation model new with Version 6.0 is not directly compatible with previous template instantiation mechanisms. When linking applications using Version 6.0 and later, instantiations might not be resolved from existing Version 5.*n* repositories. Where possible, it is safest to start fresh with an empty repository and create the required instantiations by compiling all source files. If this is not possible, there are some strategies that can be used to link mixed generation instantiations.

If you used both Version 6.*n* and Version 5.*n* to build applications, VSI strongly recommends that you use different repositories to contain automatic template instantiations for Version 6.*n* and Version 5.*n* compilations. The default repository name is the same for Version 6.*n* as for prior versions. Thus, if you use Version 6.*n* with older pre-6.*n* versions, you should do compilations in a different directory for each compiler or explicitly specify a different repository for each using the /REPOSITORY qualifier.

E.2.3.1. Linking with Version 5.n Instantiations

When linking applications using Version 6.*n* against instantiations created with Version 5.*n*, it is necessary to complete the Version 5.*n* instantiation process, to create instantiation object files. If `old_repository` is a Version 5.*n* repository then you would create the Version 5.*n* instantiation object files by using the Version 5.*n* `cxxlink`:

```
CXXLINK/NOEXE /REPOSITORY=[.old_repository]
   <Version 5.n object files>
```

`<Version 5.n object files>` are the object files that were created using the Version 5.*n* compiler; `old_repository` now contains the instantiation object files. Create a library of these object files as follows:

```
LIBRARY/CREATE/OBJECT lib_old_repository/LOG
LIBRARY/INSERT/OBJECT lib_old_repository/LOG
       [old_repository]*.obj
```

When linking using Version 6.*n*, specify `lib_old_repository.olb` after all of the Version 5.*n* object files that are being linked.

E.2.3.2. Linking Version 5.n Applications Against Version 6.*n* Repositories

In a similar way, you can create a library of Version 6.*n* instantiation object files to link into a Version 5.*n* application being linked using C++ Version 5.*n*. If `new_repository` is the Version 6.*n* repository, then a library of the instantiations would be created by:

```
LIBRARY/CREATE/OBJECT lib_new_repository/LOG
LIBRARY/INSERT/OBJECT lib_new_repository/LOG [new_repository]*.obj
```

When linking using Version 5.*n*, specify `lib_new_repository.olb` after all of the Version 6.*n* object files that are being linked.

E.2.4. Library Differences

Aspects of memory allocation and deallocation have changed from the V5.*n* and earlier compilers to the Version 6.*n* compilers. See the description of /[NO]STDNEW and /[NO]GLOBAL_ARRAY_NEW in *The C++ Standard Library*.

E.3. Using Classes

This section discusses porting issues pertaining to C++ classes.

E.3.1. Friend Declarations

When making `friend` declarations, use the elaborated form of type specifier. The following code fragment implements the legal and comments out the illegal `friend` declaration:

```
class Y;
class Z;
class X;
   //friend Y;  ** not legal
   friend class Z; // legal
};
```

E.3.2. Member Access

Unlike some older C++ implementations, VSI C++ strictly enforces accessibility rules for `public`, `protected`, and `private` members of a base class. For more information, see *The Annotated C++ Reference Manual*.

E.3.3. Base Class Initializers

Unlike some older C++ implementations, VSI C++ requires you to use the base class name in the initializer for a derived class. The following code fragment implements a legal initializer and comments out an illegal initializer:

```
class Base {
    // …
public:
    Base (int);
};
class Derived : public Base {
    // …
public:
    // Derived(int i) : (i)  {/* … */}    ** not legal
    Derived(int i) : Base(i) {/* … */} // ** legal, supplies class name
};
```

For more information, see The Annotated C++ Reference Manual.

E.4. Undefined Global Symbols for Static Data Members

When a static data member is declared, the compiler issues a reference to the external identifier in the object code, which must be resolved by a definition. The compiler does not support the declaration anachronism shown in *The Annotated C++ Reference Manual*.

For example, consider the following code fragment:

```
class C {
        static int i;
        };
//missing definition
//int C::i = 5;
int main ()
{
    int x;
    x=C::i;
    return 0;
}
```

The compiler does not issue any messages during compilation; however, when you attempt to link a program containing this code, the linker issues an unresolved symbol error message for the variable `C::i`.

E.5. Functions and Function Declaration Considerations

VSI C++ requires the use of function definitions as described in *The Annotated C++ Reference Manual*. For examples of outdated syntax not allowed in VSI C++, see *The Annotated C++ Reference Manual*.

Because all linkage specifications for a name must agree, function prototypes are not permitted if the function is later declared as an inline function. The following code is an example of such a conflicting function declaration:

```
int f();
inline int f() { return 1; }
```

In this example, `f` is declared with both internal and external linkage, which causes a compiler error.

E.6. Using Pointers

This section demonstrates how to use pointers effectively in VSI C++.

E.6.1. Pointer Conversions

In VSI C++, you cannot implicitly convert a `const` pointer to a nonconstant pointer. For example, `char *` and `const char *` are not equivalent; explicitly performing such a cast can lead to unexpected results.

For more information, see *The Annotated C++ Reference Manual*.

E.6.2. Bound Pointers

Binding a pointer to a member function with a particular object as an argument to the function is not allowed in VSI C++. For more information on the illegality of casting bound pointers, see *The Annotated C++ Reference Manual*.

E.6.3. Constants in Function Returns

Because the return value cannot be an lvalue, the `const` keyword in a function return has no effect on the semantics of the return. However, using the `const` keyword in a function return does affect the type signature. For example:

```
static int f1( int a, int b) {;}
const int (* const (f2[])) (int a, int b) = {f1};
```

In this example, the referenced type of the pointer value `f1` in the initializer for `f2[]` is *`function`* `(signed int, signed int)`, which returns `signed int`. This is incompatible with *`function`* `(signed int, signed int)`, which returns `const signed int`.

You can omit the `const` of `int` because it affects only the constant return signature.

E.6.4. Pointers to Constants

The following example shows a type mismatch between a pointer to a `char` and a pointer to a `const char` that some other compilers might not find:

```
void foo (const char* argv[]) {}
int main()
{
        static char* args[2] = {"foo","bar"};
/* In this statement, the referenced type of the pointer value
 "args" is "pointer to char" which is not compatible with
 "pointer to const char"'*/
        foo (args);
return 0;
}
```

You can correct this example by changing `static char` to `static const char`. Use an explicit type cast to get an argument match only if no other option is available; such a cast may break on some C ++ implementations.

E.7. Using typedefs

Using a synonym after a `class`, `struct`, or `union` prefix is illegal. Using a synonym in the names for constructors and destructors within the class declaration itself is also illegal.

In the following example, the illegal `typedef` specifier is commented out:

```
typedef struct { /* … */ } foo;
// typedef struct foo foobar;             ** not legal
```

For more information, see *The Annotated C++ Reference Manual*.

E.8. Initializing References

VSI C++ warns against initializing nonconstant references to refer to temporary objects. The following example demonstrates the problems that can result:

```
static void f()
{
    int i = 5;
    i++;         // OK
    int &ri = 23;
    ri++;        // In the initializer for ri, the initialization of a
                 // non-const reference requires a temporary for "23".
}
```

The issue of reference initialization arises most often in assignment operators and in copy constructors. Wherever possible, declare all reference arguments as `const`.

For more information, see *The Annotated C++ Reference Manual*.

E.9. Using the switch and goto Statements

Branching around a declaration with an explicit or implicit initializer is not legal, unless the declaration is in an inner block that is completely bypassed. To satisfy this constraint, enclose the declaration in a block. For example:

```
int i;
switch (i) {
case 1:
    int l = 0;     //not initialized at this case label
    myint m = 0;   //not initialized at this case label
    {
    int j = 0;     // legal within the braces
    myint m = 0;   // legal within the braces
    }
case 2:
    break;
// …
}
```

For more information on using the `switch` statement, see *The Annotated C++ Reference Manual*.

E.10. Using Volatile Objects

You must supply the meaning of copy constructing and assigning from volatile objects, because the compiler generates no copy constructors or assignment operators that copy or assign from volatile objects. The following example contains examples of such errors, as noted in the comments:

```
class A {
public:
  A() { }
  // A(volatile A&) { }
  // operator=(volatile A&) { return 0; }
};
void foo()
{
  volatile A va;
  A a;
  A cca(va);  // error - cannot copy construct from volatile object
  a = va;     // error - cannot assign from volatile object
  return;
}
```

For more information, see *The Annotated C++ Reference Manual*.

E.11. Preprocessing

VSI C++ allows identifiers, but not expressions, on the `#ifdef` preprocessor directive. For example:

```
// this is not legal
// #ifdef KERNEL && !defined(__POSIX_SOURCE)
```

The following is the legal alternative:

```
// use this instead
#if defined(KERNEL) && !defined(__POSIX_SOURCE)
```

For more information, see *The Annotated C++ Reference Manual*.

E.12. Managing Memory

The proper way to manage memory for a class is to overload the `new` and `delete` operators. This is in contrast to some older C++ implementations, which let you manage memory through assignment to the `this` pointer.

For more information, see *The Annotated C++ Reference Manual*.

Program developers must take care that any user-defined `new` operators always return pointers to quadword-aligned memory.

E.13. Size-of-Array Argument to delete Operator

If a size-of-array argument accompanies a `delete` operator, VSI C++ ignores the argument and issues a warning. The following example includes an anachronistic use of the `delete` operator:

```
int main()
{
        int *a = new int [20];
        int *b = new int [20];
        delete[20] a;     //old-style; argument ignored, warning issued
        delete[] b;
return 0;
}
```

E.14. Flushing the Output Buffer

Do not depend on the newline character (`\n`) to flush your terminal output buffer. A previous stream implementation might have done so, but this behavior is not in conformance with Version 2.0 of the AT&T `iostream` library. If you want to flush the output buffer, use the `endl` manipulator or the `flush` member function.

E.15. Linking

When linking applications, use CXXLINK instead of LINK. See Section 1.3 *(Alpha only)* and Section 1.4 *(I64 only)*.

E.16. Incrementing Enumerations

Some other C++ implementations let you perform integer arithmetic, including ++, on enumerated types; VSI C++ does not allow this.

E.17. Guidelines for Writing Clean 64-Bit Code

Paying careful attention to data types can ensure that your code works on both 32-bit and 64-bit systems. Use the following guidelines to write clean 64-bit code:

- Variables that should be 32 bits in size on both 32-bit systems and 64-bit OpenVMS systems should be declared as `int` (not `long`).
- If you want 32-bit variables on a 32-bit system and an OpenVMS system, declare them as `int`.
- A 64-bit number on OpenVMS must be declared as `__int64` or `long long`.
- Remember that `register` variables and `unsigned` variables default to `int` (32 bits).
- Constants are 32-bit quantities by default. Performing shift operations or bit operations on constants will give 32-bit results. You must add `L` to the constant to get a 64-bit result. For example:

  ```
  long foo, bar;
  foo = 1L << bar;
  ```

- Assigning to a `char` is not atomic on OpenVMS systems. You will get a load of 32 or 64 bits, followed by byte operations to extract, mask, and shift the byte, followed by a store of 32 or 64 bits.
- Bit-fields declared as `int` on OpenVMS systems generate load/store 32 bits. Bit-fields declared as `long` on OpenVMS systems generate load/store 64 bits.
- If you do not explicitly declare the formal parameters to functions, their sizes may not match the caller sizes. The default is `int`, which truncates 64-bit addresses.
- The `%d` and `%x` format specifiers print 32 bits of data. Use `%Ld` or `%Lx` with `printf` to print 64 bits of data. You can use `%p` on both 32- and 64-bit systems to print the value of pointers.