VSI C++ V10.1-4 for OpenVMS x86-64

Release Notes

Publication Date: July 2026

Operating System: VSI OpenVMS x86-64 Version 9.2-2 or higher

Kit Name: VSI-X86VMS-CXX-V1001-4-1.ZIP

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

VSI C++ V10.1-4 for OpenVMS x86-64 Release Notes

Copyright © 2026 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

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

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

All other trademarks and registered trademarks mentioned in this document are the property of their respective holders.

Table of Contents

1. Kit Name 4
2. Kit Description 4
3. Version(s) of VSI OpenVMS to Which This Kit May Be Applied 5
4. Installing C++ Run-time Libraries 5
5. Main Changes from Prior Compilers 5
6. OpenVMS-Specific Pragmas 8
7. TLB and Headers 11
8. Prologue/Epilogue Files 11
    8.1. Using Prologue/Epilogue Files 11
9. Differences Between C++ on OpenVMS IA-64 and OpenVMS x86-64 12
10. SYS$STARTUP:CXX$STARTUP.COM Startup File 15
11. Known Issues 15
    11.1. Using the Clang Command Line 20
12. Supported and Not Supported DCL Qualifiers 21
13. Using Clang Command Options From DCL Command Line 23
14. Mapping Clang Command Options to DCL Command Options 24
15. Threads and Thread-Local Variables 32
16. Online Clang Documentation 32

1. Kit Name

VSI-X86VMS-CXX-V1001-4-1.ZIP

2. Kit Description

This kit includes the VSI C++ x86-64 compiler. This compiler runs on OpenVMS x86-64 and generates code for OpenVMS x86-64.

The compiler is based on the LLVM Clang compiler with additional OpenVMS features. You can learn more about Clang at https://clang.llvm.org.

The PCSI package provides two C++ compilers, the CXX demangling tool, and C++ system headers:

1. Clang – this compiler has a UNIX-style command line and is invoked as an OpenVMS foreign command. To use this compiler, you will need to set up the CLANG DCL symbol by calling the SYS$EXAMPLES:CXX$SETUP.COM command file.

2. CXX – this compiler has a DCL command line that is compatible with the C++ compiler on OpenVMS IA-64. Currently, it does not support all possible qualifiers. Unsupported qualifiers are silently ignored. See *Section 11, "Known Issues"* for a summary of qualifiers.

   The **HELP CXX** command prints out general information about the CXX compiler and lists all supported qualifiers. The supported qualifiers are listed first, followed by the qualifiers that are ignored for now.

3. CXX$DEMANGLE – a demangling tool. It is invoked as an OpenVMS foreign command. The file CXX$DEMANGLE.EXE is placed in the SYS$COMMON:[SYSLIB] directory. It is a shareable image as well as an executable file and the SYS$EXAMPLES:CXX$SETUP.COM command file will define the CXX$DEMANGLE DCL symbol. The CXX$DEMANGLE will also be used by later versions of VMS tools like LINKER, LIBRARIAN, etc. The tool is taken from the LLVM toolset known as llvm-cxxfilt and uses a UNIX-style command line. For more information enter:

   ```
   $ CXX$DEMANGLE --help
   ```

4. C++ system headers – the installation provides headers for the LIBCXX library, C++ specific versions of CRTL headers, and C++ selected versions of some OpenVMS system headers. VSI expects to stop shipping some of these C++ specific headers in the future and only provide them through the base OpenVMS distribution.

Previous C++ kits would provide C++ run-time library images (LIBCXX and LIBCXXABI) to install on OpenVMS V9.2-1 and V9.2-2 systems (possibly overriding images already on the system). However, that would lead to a confusion during OpenVMS upgrades that would cause the wrong RTL to be retained on the upgraded system.

Beginning with OpenVMS V9.2-3, the C++ RTLs (LIBCXX and LIBCXXABI) are now part of the operating system. Older C++ RTLs on V9.2-1 or V9.2-2 were bundled inside the compiler kit and were considered part of the compiler. This caused confusion when upgrading systems to V9.2-3.

To correctly track RTL versions and ownership, a PCSI kit named CXXFIXUP provides the correct RTLs and corrects the ownership. Any V9.2-3 system that was installed "clean" and not upgraded from V9.2-1 or V9.2-2 has the correct RTLs and correct PCSI ownership. See *Section 4, "Installing C++ Run-time Libraries"* and the release notes inside the CXXFIXUP kit for more detailed information. The CXXFIXUP kit is available separately from the support portal.

Any fixes to the RTLs will be provided as HOTFIX or ECO kits for OpenVMS V9.2-3 and will not be included in the compiler kit. Future OpenVMS releases will contain the latest versions.

3. Version(s) of VSI OpenVMS to Which This Kit May Be Applied

The compiler requires OpenVMS x86-64 V9.2-2 or later. VSI strongly suggests that you upgrade to V9.2-3 or later. If C++ is installed on the V9.2-2 system, you must remove C++ prior to the operating system upgrade.

4. Installing C++ Run-time Libraries

To determine if you need to install the CXXFIXUP kit, use the **PRODUCT SHOW OBJECT** command to determine the ownership/generation of the libraries on your system.

If your system shows that the LIBCXX/LIBCXXABI is provided by any version of CXX as shown:

```
$ product show object/full libcxx,libcxxabi
---------------------------------- ----------------- -------- ------------------------------ --------------
OBJECT NAME                        OBJECT TYPE (GEN) STATUS   DESTINATION ROOT DIRECTORY     PROVIDED BY
---------------------------------- ----------------- -------- ------------------------------ --------------
LIBCXX                             imgl (2147483647) OK                                      CXX V10.1-2
LIBCXX                             imgli (130060000) Conflict                                VMS V9.2-3
LIBCXXABI                          imgl (2147483647) OK                                      CXX V10.1-2
LIBCXXABI                          imgli (130060000) Conflict                                VMS V9.2-3
---------------------------------- ----------------- -------- ------------------------------ --------------
4 items found
```

Then you must take the following steps:

1. Remove your current CXX compiler with **PRODUCT REMOVE CXX**.
2. Install the CXXFIXUP kit that will correct the PCSI database and provide the correct RTLs on the target system.
3. Install this CXX kit.

The corrected system will look similar to this:

```
$ product show object/full libcxx,libcxxabi
---------------------------------- ----------------- -------- ------------------------------ -----------
OBJECT NAME                        OBJECT TYPE (GEN) STATUS   DESTINATION ROOT DIRECTORY     PROVIDED BY
---------------------------------- ----------------- -------- ------------------------------ -----------
LIBCXX                             imgli (130060000) OK                                      VMS V9.2-3
LIBCXXABI                          imgli (130060000) OK                                      VMS V9.2-3
---------------------------------- ----------------- -------- ------------------------------ -----------
2 items found
```

5. Main Changes from Prior Compilers

The CXX 10.1-4 compiler includes the following changes from the CXX 10.1-3U1 GA version:

- The full compiler command line is now stored in the `.note` section of the object file.
- New `-listing-file` flag (auto-generates an *input*`.LIS` name) and a new `-show-all` flag that enables `-show-source`, `-show-header`, `-show-include`, and `-show-expansion` together.

- By default, CXX now applies `-Wno-deprecated` when compiling with the GNU++98 standard. As a result, the `deprecated-header` warning is no longer issued.
- Improved debug information, including the values of constant strings, virtual function locations, and corrected bit-field offset calculations.
- Diagnostic messages from the LLVM tools (llc, opt, and llvm-mc) now include the program name.
- Fixed various issues, including:
  - Casting a const `std::string` to a const `std::string32`
  - Compilation of `.S` files (response files are now processed in-process)
  - Default copy constructor failures for structs containing 32-bit pointers
  - Exception handling: `terminate()` is now called for C++ exceptions only
  - Incorrect `std::cout` output for (unsigned) long values
  - A crash when using `-fno-builtin`
  - A compiler crash when building code that uses certain x86 intrinsics
  - The jacket routine (`ELF$TFRADR`) now sets the frame pointer
  - The C++ standard library include path is no longer added when `-nostdinc`, `-nostdinc++`, or `-nostdlibinc` is given
  - A C++ installation kit packaging issue with header files

The CXX 10.1-3U1 compiler includes the following changes from the CXX 10.1-3 GA version:

- [NS8953] OpenVMS x86; Fixed the padding problem related to `#pragma member_alignment`, with **/NOMEMBER_ALIGNMENT** qualifier. Added member alignment guards in STL header files.
- The `ifunc()` function attribute is ignored, as it is not supported in OpenVMS.
- Fixed the conflict related to variables declared in unnamed namespaces.
- Fixed the crash caused by using ternary operator when compiling with **/POINTER_SIZE=32** qualifier.

The CXX 10.1-3 GA compiler includes the following changes from the CXX 10.1-2 GA version:

- Fixed issues with 32-bit string allocation, including problems with template functions and functions returning `std::string` with pointer-type parameters.
- Updated global allocation ::operator `new` to depend on pointer size. It now allocates from 32-bit space when compiled under `#pragma pointer_size 32` or when the **/POINTER_SIZE=32** qualifier is used.
- Corrected the MMS dependency file format generator. The **/MMS_DEPENDENCIES** qualifier now produces valid dependency files.
- Resolved multiple bugs related to 32-bit pointer size handling.

- Standardized Clang/CXX and llvm-mc messages with OpenVMS-specific prefixes: %CXX-F-FATAL, %CXX-E-ERROR, %CXX-W-WARN, %CXX-I-INFO. Conclusion messages now start with %CXX-W-ENDDIAG or %CXX-E-ENDDIAG (see *Section 9, "Differences Between C++ on OpenVMS IA-64 and OpenVMS x86-64"* for details).
- Fixed and improved debug information generation and handling in the back end.
- Improved listing file generation with prologue/epilogue files, and fixed issues with long filenames.
- Fixed various issues including:
  - Unexpected inclusion of `<math.h>`
  - Undefined symbols when compiling `.c` files
  - Incorrect results from `#pragma nomember_alignment`
  - Missing or incorrect warning messages
  - Other miscellaneous bugs and problems

The CXX 10.1-2 GA compiler includes the following changes from the CXX 10.1-1 GA version:

- By default, the operators `new` and `new[]` return 64-bit addresses that will not fit into 32-bit pointers. This is a change from the V10.1-1 compiler that always allocated memory in 32-bit heap. Compilations that use **/POINTER_SIZE=32** on the command line will allocate memory from the 32-bit heap, but compilations that do not use **/POINTER_SIZE** or those that use **/POINTER_SIZE=64** will allocate memory from the 64-bit heap. The `#pragma pointer_size` also changes the behavior of the `new()` operator and it allocates memory from the 32-bit or 64-bit space depending on the pragma option.

  For programs that revert back to 32-bit memory allocation with **/POINTER_SIZE=32**, ensure that all systems have the newest LIBCXX and LIBCXXABI libraries installed on target systems. Currently, these updated RTLs are only available with the C++ kit, so you must install C++ on all target systems, even those that will not be used for development. The updated RTLs will be included in a future update for OpenVMS V9.2-2.
- The **/LIST** qualifier is now supported. The compiler supports most of the **/SHOW** keywords (like on IA-64) and creates a listing file that has a very similar look and feel to the listing files created by the Alpha and IA-64 compilers.
- LIB$ESTABLISH built-in is fully supported by CXX.
- The allocation of objects of `std::string` is now based on the current pointer size setting. The default allocator for `std::string` will automatically be changed to the allocator appropriate for the 32-bit address space based on the current **/POINTER_SIZE** or `#pragma [required_]pointer_size` setting. When the pointer size is set to 32 bits, the object will be allocated in the 32-bit memory to ensure that `std::string::c_str()` and `std::string::data()` return valid 32-bit pointer values.
- The ability to do source-level debugging in the OpenVMS debugger is now supported. When debugging, users need to enter **SET SOURCE** before printing a line from a file.

The kit contains a log file that lists the changes in this release and in prior releases:

```
$ type SYS$HELP:CXX.CHANGELOG
```

The compiler behaves very much like the Linux version of the Clang compiler in terms of language features. The primary difference is the support for the OpenVMS-specific features, pragmas, and predefined symbols.

VSI C++ predefines common OpenVMS symbols much like the IA-64 C++ compiler (for example, VMS, __VMS, __vms, __INITIAL_POINTER_SIZE), as well as the industry standard symbols such as __x86_64 and __x86_64__. VSI C++ compiler does not define the __DECCXX_VER and __DECC_VER macros. Instead, it defines the __VSIC_VER and __VSICXX_VER macros. You can use these macros to test whether the current compiler version is newer than another version. In order to get the C++ standard version, you need to use __cplusplus value. When you compile in C mode with Clang, please use __STDC_VERSION__ to get the C version.

VSI C++ also defines the _USE_STD_STAT macro by default, which is not defined in IA-64 C++. To undefine this macro from the command line, use the **/UNDEFINE=_USE_STD_STAT** qualifier for CXX or the `-U_USE_STD_STAT` option for Clang.

Unlike a traditional UNIX compiler that "compiles and links" with a single command, the compiler on OpenVMS only compiles (i.e., the equivalent of the "`-c`" option on UNIX).

OpenVMS V9.2-3 (or the CXXFIXUP kit) provides the C++ run-time libraries in both static and shared library form. If a program needs to be linked against the static libraries, you must provide them explicitly on the **LINK** command or you can use the SYS$EXAMPLES:CXX.OPT option file.

For example:

```
// HW.CXX
#include <iostream>

int main()
{
  std::cout << "Hello World!\n";
  return 0;
}

$ CXX HW.CXX
$ LINK HW ! Link against installed shared libraries
$ RUN HW
Hello World!
$ LINK HW,SYS$EXAMPLES:CXX/OPT ! Link against static libraries
$ RUN HW
Hello World!
```

The OpenVMS x86-64 linker places code into the 64-bit address by default (the default on Alpha and IA-64 was to place code into 32-bit address space). Programs should not notice the difference. The linker creates small trampolines in the 32-bit address space, so the address of a routine will still fit into a 32-bit variable.

6. OpenVMS-Specific Pragmas

Compilers have been extended to support the following OpenVMS-specific pragmas:

```
  #pragma pointer_size [options]
  #pragma required_pointer_size [options]
  #pragma [no]member_alignment [options]
  #pragma extern_model [options]
  #pragma extern_prefix [options]
  #pragma include_directory <string-literal>
```

```
#pragma message [option] ("message-list")
```

Other OpenVMS pragmas are processed, however they only have partial/limited support at the moment. The `#pragma` message is supported, but the names of the error messages on IA-64 and x86-64 are different, so they need to be modified.

The `#pragma pointer_size` is active only when the DCL qualifier **`/POINTER_SIZE`** (or `-pointer-size` option for the foreign **`CLANG`** command) is given. This matches the behavior of the VSI C++ IA-64 compiler.

Neither **`/POINTER_SIZE`** (`-pointer-size`) nor `#pragma [required_]pointer_size` has any effect on vptr. The size of the virtual pointer is always 64 bit.

The option provided to `#pragma nomember_alignment [option]` determines the base alignment of the structure. If provided with the appropriate alignment, the attribute is set on the structure declaration.

There are some differences from the IA-64 CXX member alignment:

- In IA-64 CXX:

  ```
  struct
   /*
   **  This directive does not change the alignment of s1 because we have
   **  already started the struct.
   */
  #pragma nomember_alignment
    {
    ...
    } s1;
  ```

  On OpenVMS x86-64, it does change the alignment. It is best to use the pragma prior to the struct definition.

- In IA-64 CXX, the `#pragma` pack cancels the base alignment defined by `#pragma nomember_alignment <base_alignment>`, but on OpenVMS x86-64, `#pragma` pack has no effect on base alignment.

- In IA-64 CXX, if the aggregate is packed, a zero-length bit field causes the next member to start at the next byte boundary.

  On OpenVMS x86-64, the zero-length bit field pads to the next alignment boundary determined by its declared base type, regardless of whether the aggregate is packed or not.

The compiler supports `#pragma extern_model`, which means that the `strict_refdef`, `common_block`, `relaxed_refdef`, `globalvalue`, `save`, and `restore` options are supported.

The following attributes are available:

```
gbl lcl
shr noshr
wrt nowrt
ovr con
exe noexe
vec novec
```

The following alignments are available:

```
byte
word
long
quad
octa
```

The IA-64 compiler provided many header files that are not STL standard files. The x86-64 installation adds hard links with .HXX extensions for all STL standard files and .H hard links for those of them that conflict with the CRTL headers.

The files like

```
stream.h(xx)
generic.h(xx)
strstream.hxx
messages.hxx
cxxl.hxx
```

were provided for the IA-64 CXX internal use only and not for compiler users. If you include these files, you must update your code to be able to compile with the x86-64 CXX compiler based on Clang.

An OpenVMS style `#pragma message` is also available. Currently, the compiler only supports the following format:

```
#pragma message [option] ("message-list")
```

The `[option]` parameter must be one of the following keywords:

Keyword	Meaning
`enable`	Enables issuance of messages specified in the message list.
`disable`	Disables issuance of messages specified in the message list.
`error`	Sets the severity of each message in the message list to Error.
`fatal`	Sets the severity of each message in the message list to Fatal.
`informational/warning`	Sets the severity of each message in the message list to Warning.
`save`	Saves the current state of the compiler messages.
`restore`	Restores the saved state of the compiler messages.

Example:

```
// example.cpp
#pragma message enable ("unused-variable")
#pragma message disable ("return-type implicitly-unsigned-literal")
int foo() {
  long long a = 12341234123412341234;
  int b = a;
}
```

With the example above, you should get something like this:

```
warning: unused variable 'a' [-Wunused-variable]
```

The values for the `message-list` is the name following the message severity code letter.
In this example, the `unused-variable` message is enabled, while `return-type` and `implicitly-unsigned-literal` are disabled. The values in the `message-list` should be separated by spaces.

The IA-64 C++ compiler message identifiers are not supported and are silently ignored. You should use Clang's respective message IDs. Note that it is not possible to reduce message severity to warning after it was set to error or fatal, but it is possible to reduce from fatal to error.

The following pragmas are not supported on x86-64 and are silently ignored by default. However, if the `-Wall` or `-Wunknown-pragmas` options are enabled, the compiler will issue a warning:

```
#pragma [no]builtins
#pragma define_template
#pragma dictionary
#pragma environment
#pragma function
#pragma [no]inline
#pragma intrinsic
#pragma module
#pragma [no]standard
```

7. TLB and Headers

The C++ compiler allows to search and use text libraries (TLB) with the **/LIBRARY** qualifier and with automatic searches for headers performed by the compiler. However, unlike in the C++ compiler on IA-64, some of the headers on x86-64 are expanded into separate files. The installation directory for the C++ standard library headers is SYS$COMMON:[VSICXX$LIB.INCLUDE.LIB_CXX.INCLUDE].

Both the **CXX** DCL command and the CLANG DCL symbol, defined by the SYS$EXAMPLES:CXX$SETUP.COM command file, know the location of these headers.

8. Prologue/Epilogue Files

Both Clang and CXX automatically process user-supplied prologue and epilogue files just like the IA-64 compiler.

8.1. 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. Then, it checks if one or both of the following specially named files or modules exist in the same location as the included file:

   ```
   __DECC_INCLUDE_PROLOGUE.H
   __DECC_INCLUDE_EPILOGUE.H
   ```

   The location is the OpenVMS directory containing the included file or the text library file containing the included module. In case of a text library, module names should not include the .H suffix.

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 `#include`.

4. The text of the epilogue file is processed just after the text of the file specified by `#include`.

5. Subsequent `#includes` that refer to files from the same location use the saved text from any prologue/epilogue file found there.

9. Differences Between C++ on OpenVMS IA-64 and OpenVMS x86-64

- The datatypes long, size_t, nullptr_t, and ptrdiff_t are 64-bits wide on OpenVMS x86-64 but only 32-bits wide on OpenVMS IA-64.
- The default size for pointers is 64 bits on OpenVMS x86-64 but only 32 bits on OpenVMS IA-64. You can use the command-line option `-pointer-size` for Clang and the **/POINTER_SIZE** qualifier for **CXX** to change the default size of pointers.
- The compiler does not automatically upcase external names like all other OpenVMS compilers. There is a `-names` command-line option for Clang and **/NAMES** qualifier for **CXX**, and their default is `as_is`.
- The compiler does not automatically truncate external names that are longer than 31 characters. External names can be any length. There are symbol length limits in the LIBRARIAN and LINKER options files, but in general, symbol lengths do not matter in LINKER. The **/NAMES=TRUNCATED** option can be used to revert to the IA-64 C++ behavior.
- The compiler does not support passing non-trivial arguments with no corresponding parameter.

  Example:

  ```
  // Non-trivial type.
  struct O {
      int a;  int b;  int c;
      ~O() {}
  };

  void hn(...) {}

  int main()
  {
    O o;
    hn(o);
    return 0;
  }
  ```

  In the example above, the C++ compiler will generate an error message similar to the following:

  ```
  error: cannot pass object of non-trivial type 'O' through variadic
  function; call will abort at runtime [-Wnon-pod-varargs]
  ```

  You can suspend this error with `-Wno-non-pod-varargs`, but the program will abort at runtime.

- Some well-known function calls may be optimized away.

  Example:

  ```
  #include <stdio.h>
  #include <stdlib.h>
  extern "C" {
    double ceil(double a) { printf("User defined\n"); return 0; }
  }
  int main() {
  ```

```
  printf("%f\n", ceil(5.6)); // user-defined won't be called, ceil call
                                optimized out
  volatile double a;
  printf("%f\n", ceil(a));
  return 0;
}
```

In this case, Clang puts the result of the first `ceil` function call directly in the place where it is called. If first call never happens, you will get the next result.

Result:

```
6.000000
User defined
0.000000
```

- The x86-64 **`CXX`** command does not support comma-separated lists for source files, it only accepts one source file at a time.

- `#pragma required_pointer_size` directive affects LIBCXX's allocating (not placement) operator `new` behavior. The operator `new` allocates memory with a 32-bit or 64-bit address, depending on the pragma option.

  Example:

```
int main() {
  #pragma required_pointer_size short
  // here the operator new allocates 32bit address memory
  int*  p1 = new int;
  delete p1;

  #pragma required_pointer_size long
  // here the operator new allocates 64bit address memory
  int*  p2 = new int;
  delete p2;
}
```

- `chf64$signal_array*` thrown by LIB$SIGNAL call cannot be caught with the catch block of the `chf64$signal_array&` param type. It can be caught directly with the `catch(chf64$signal_array*)` or `catch(...)` block.

- The allocation of objects of `std::string` is now based on the current pointer size setting. The default allocator for `std::string` will automatically change based on the current **`/POINTER_SIZE`** or `#pragma [required_]pointer_size` setting. When the pointer size is set to 32 bits, the object will be allocated in 32-bit memory to ensure that `std::string::c_str()` and `std::string::data()` return valid 32-bit pointer values.

- Template instantiation happens according to the C++ standard given by the respective **`/STANDARD`** option (default is C++98 standard with GNU extensions). IA-64 CXX template instantiation features are not supported.

- With the x86-64 CXX compiler you can get the address of a label defined in the current function with the unary operator '&&'. The value has type `void *`.

  Example:

```
static const void *L[] = {&&L1, &&L2, &&L3};
```

```
goto *L[i];
```

When compiled with **/POINTER_SIZE=32** the program ends with %SYSTEM-F-ACCVIO, which is expected since array L is an array of 32-bit pointers but the addresses of those labels in the code would be in 64-bit address space. Label addresses will be 32-bits if you link with **/SEG=CODE=P0**.

IA-64 CXX gives compilation errors for taking the address of a label.

- With the OpenVMS x86-64 CXX compiler, the address space of placement new/delete operators parameters need to match. As the 32-bit pointer size is applied only to pointer types and not to array types, the following new/delete pairs don't work:

```
void* operator new(size_t s, int* i);
void operator delete(void* p, int i[5]);
```

or

```
void* operator new(size_t s, int i[5]);
void operator delete(void* p, int* i);
```

In 32-bit mode, the compiler cannot find the matching delete operator for the specified new operator as the address spaces of the second parameter are different, hence the delete operator is not called. The example works correctly if both second parameters are either of pointer type or of array type.

- The x86-64 C++ compiler is based on the clang open-source compiler. The output format for messages does not follow the style of other OpenVMS compilers. However, the final $STATUS DCL value after a compilation should be exactly the same as the IA-64 compiler, with the bottom 4 bits representing the final OpenVMS severity. A compilation with errors results in a $STATUS value of $STATUS == "%X15F61262" (this is exactly like IA-64).

  Also, the messages of Clang (as well as llvm-mc) have OpenVMS-specific prefixes such as %CXX-F-FATAL, %CXX-E-ERROR, %CXX-W-WARN, %CXX-I-INFO for appropriate fatal/error/warning/info (note, remark) messages and %CXX-W-ENDDIAG or %CXX-E-ENDDIAG before conclusion message. For example:

```
$ clang test.cpp
%CXX-W-WARN, test.cpp:7:10: warning: implicit conversion from 'double'
 to 'int' changes value from 0.022 to 0 [-Wliteral-conversion]
       b = 0.022;
             ~ ^~~~
%CXX-W-ENDDIAG, 1 warning generated.

$ cxx err.cpp
%CXX-E-ERROR, USER$:[USER]err.cpp;19:7:3: error: use of undeclared
 identifier 'bbbb'; did you mean 'bbbbb'?
  bbbb = 0.022;
  ^~~~
  bbbbb
%CXX-I-INFO, USER$:[USER]err.cpp;19:2:12: note: 'bbbbb' declared here
 extern int bbbbb;
        ^
%CXX-W-WARN, USER$:[USER]err.cpp;19:7:10: warning: implicit conversion
 from 'double' to 'int' changes value from 0.022 to 0
 [-Wliteral-conversion]
  bbbb = 0.022;
       ~ ^~~~
%CXX-E-ENDDIAG, 1 warning and 1 error generated.
```

- The **CXXLINK** is not provided and is fully replaced by **LINK**.

  CXXLINK was only required on Alpha as a "prelinker". The CXXLINK image did some minimal processing and then invoked the standard Alpha linker. For IA-64, CXXLINK only provided some formatting of mangled names from any linker-generated error messages. Later versions of the IA-64 compiler and IA-64 linker were enhanced to provide such mangled name formatting and removed the need for CXXLINK. So on IA-64, the **CXXLINK** command just spawns the system linker with the same command line. You can replace all uses of **CXXLINK** on IA-64 with **LINK** and get the same results.

  For x86-64, we simply removed the CXXLINK image and corresponding **HELP** entry. The current compiler and linker do not provide automatic name demangling like the IA-64 system, but that is something we are actively working on for a future release.

  The **HELP CXX** entry has been updated to have a CXXLINK subtopic to clearly indicate that **CXXLINK** is not provided on x86-64 and any references to it in command files can be replaced by **LINK** with no other changes required.

10. SYS$STARTUP:CXX$STARTUP.COM Startup File

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

11. Known Issues

- long double

  The long double data type is not fully supported yet.

- No machine code listing file generation. The **ANALYZE/OBJECT/DISASSEMBLE** command will produce machine code with source line numbers. It does not show any static data or static constants.

- Still have some problems with 32-bit pointers.

  Previously built and linked applications may have problems using the new versions of C++ RTLs (LIBCXX and LIBCXXABI). Please report any issues related to 32/64-bit pointers.

- Calling `std::cout.operator<<` within global objects constructors brings to %SYSTEM-F-ACCVIO when linked with static libraries CXX_STATIC.OLB and CXXABI_STATIC.OLB.

- The overloading of global new/delete operators are currently not supported.

- Comma-separated list of source files is not supported by CXX, meaning you cannot compile multiple files at once.

- If you have one or more files with the same name as the standard library header name, you must specify their location with the `-iquote` option of Clang instead of the **/INCLUDE** qualifier to avoid confusion.

  Let's say the file is in the current directory. Instead of specifying

```
$ CXX /INCLUDE=[] FOO.CXX
```

use:

```
$ CXX /CLANG=("-iquote", "[]")
```

- If your C++ code calls LIB$FIND_IMAGE_SYMBOL or any other code that in turn calls LIB$FIND_IMAGE_SYMBOL, you must link with **/THREADS_ENABLE**.

- The built-in functions are not fully supported yet for OpenVMS x86-64. Currently, only the functions with the __ATOMIC_* and __PAL_* prefixes are supported:

  __ATOMIC_ADD_LONG
  __ATOMIC_ADD_LONG_RETRY
  __ATOMIC_AND_LONG
  __ATOMIC_AND_LONG_RETRY
  __ATOMIC_OR_LONG
  __ATOMIC_OR_LONG_RETRY
  __ATOMIC_INCREMENT_LONG
  __ATOMIC_INCREMENT_LONG_RETRY
  __ATOMIC_DECREMENT_LONG
  __ATOMIC_DECREMENT_LONG_RETRY
  __ATOMIC_EXCH_LONG
  __ATOMIC_EXCH_LONG_RETRY
  __ATOMIC_ADD_QUAD
  __ATOMIC_ADD_QUAD_RETRY
  __ATOMIC_AND_QUAD
  __ATOMIC_AND_QUAD_RETRY
  __ATOMIC_OR_QUAD
  __ATOMIC_OR_QUAD_RETRY
  __ATOMIC_INCREMENT_QUAD
  __ATOMIC_INCREMENT_QUAD_RETRY
  __ATOMIC_DECREMENT_QUAD
  __ATOMIC_DECREMENT_QUAD_RETRY
  __ATOMIC_EXCH_QUAD
  __ATOMIC_EXCH_QUAD_RETRY
  __PAL_INSQHIL
  __PAL_INSQTIL
  __PAL_INSQUEL
  __PAL_INSQUEL_D
  __PAL_INSQHILR
  __PAL_INSQTILR
  __PAL_REMQHIL
  __PAL_REMQTIL
  __PAL_REMQUEL
  __PAL_REMQUEL_D
  __PAL_REMQHILR
  __PAL_REMQTILR
  __PAL_INSQHIQ
  __PAL_INSQTIQ
  __PAL_INSQUEQ
  __PAL_INSQUEQ_D
  __PAL_INSQHIQR

__PAL_INSQTIQR
__PAL_REMQHIQ
__PAL_REMQTIQ
__PAL_REMQUEQ
__PAL_REMQUEQ_D
__PAL_REMQHIQR
__PAL_REMQTIQR

Information for each group of newly supported built-in functions will be added here and in the RELEASE_NOTES file of the OpenVMS x86-64 CXX kit.

- In some cases, the optimizer unexpectedly removes a user-defined operator `delete()`. This optimizer bug has already been fixed in a currently unreleased version of the LLVM backend. A future version of VSI C++ will include this newer LLVM. The workaround is to disable the optimizer at compilation time.

  Sample code:

```
$ type bug.cxx
   extern "C" int printf(const char *,...);
   class C {
     public:
       C() { throw "up"; }
   };

   void operator delete(void *ptr) throw()
   {
     printf("Called delete\n");
   }

   int main()
   {
     try {
       C *p = new C; // C() will throw exception, invoking delete
     }
     catch (...) { printf("Exception Caught\n"); }

     return 0;
   }

   $ CXX  /OPT=(LEVEL=3) bug.cxx
   $ link bug
   $ run  bug
   Exception Caught

   // But the expected output is:
   Called delete
   Exception Caught
```

- To be able to successfully use **`CXX NL:`** or **`CXX SYS$INPUT`** on ODS-5 disks, please set DECC$RENAME_NO_INHERIT to 1 as follows:

```
$ define decc$rename_no_inherit 1
```

- `std::string` becomes `std::string32` in contexts where the pointer size is 32 bits, and these two types can be implicitly converted to each other. Still, these pointers and non-const references for these types are not implicitly convertible.

Example:

```
#include <string>

#pragma required_pointer_size short
void foo32(std::string& str32);
void foo32const(const std::string& str32);
#pragma required_pointer_size long

void foo64(std::string& str64);
void foo64const(const std::string& str64);

int main () {
#pragma required_pointer_size short
  std::string str32 = "hello str32";
  std::string* str32p = &str32;
#pragma required_pointer_size long
  std::string str64 = "hello str64";
  std::string* str64p = &str64;

  str32 = str64; // Ok
  str64 = str32; // Ok
  foo32const(str64); // Ok
  foo64const(str32); // Ok

  str64p = str32p; //Failure
  str32p = str64p; //Failure
  foo32(str64); //Failure
  foo64(str32); //Failure

  return 0;
}
```

- Previously, `stdlib.h` included the `math.h` header file. Starting from the current version, this inclusion has been removed.

  As a result, you may encounter compilation errors due to undeclared symbols (such as `abs` or `logf`). To resolve this, please add `#include <math.h>` to your code.

- There could be some differences with IA-64 CXX in handling of `#pragma extern_model strict_refdef`. For example:

  ```
  #pragma extern_model strict_refdef "ABCD" lcl, shr, exe, wrt, novec,
   long
  ```

  In the map file, it is as:

  ```
  ABCD                            xxxxxxxx xxxxxxxx xxxxxxxx (
    dd.) OCTA  4 CON,REL,LCL,  SHR,NOEXE,  WRT,NOVEC,NOMOD
  ```

  But should be:

  ```
  ABCD                            xxxxxxxx xxxxxxxx xxxxxxxx (
    dd.) OCTA  4 CON,REL,LCL,  SHR,NOEXE,  WRT,NOVEC,NOMOD
  ```

- Two functions called `floatundidf()` and `floatundisf()`, from LLVM compiler_rt library (which is now loaded in LIBOTS2.EXE shared library) are not working correctly. The problem is under investigation.
- Printing a 64-bit pointer with `std::cout` (`operator<<`) shows only the low 32 bits of the address. This affects only addresses that do not fit in 32 bits, such as memory allocated in 64-bit (P2) space; pointers into the default low-memory regions print correctly.

  The cause is the CRTL handling of the `"%p"` conversion and will be addressed in a future release.

  As a workaround, cast the pointer to a 64-bit unsigned integer and print that value. The `long long` formatting path is not affected by this issue.

  Example:

```
#include <iostream>

int main()
{
    void *p = (void *)0x00007FFFFEEDFACEULL;

    std::cout << p << std::endl;   // prints FEEDFACE (low 32 bits only)

    // Workaround: print the full address as a 64-bit integer.
    std::cout << std::hex << std::uppercase
              << reinterpret_cast<unsigned long long>(p)
              << std::endl;        // prints 7FFFFEEDFACE

    return 0;
}
```

- Printing a `long` (or `unsigned long`) value with `printf()` using the `"%l"` length modifier (for example, `"%ld"` or `"%lu"`) produces incorrect results.

  On OpenVMS x86-64, the `long` type is 64 bits, but the C run-time library's `printf()` interprets the `"%l"` modifier as a 32-bit value, so the upper 32 bits are lost.

  Example:

```
long value = 0x100000000L;   /* 4294967296 */

printf("%ld\n", value);      /* prints 0 instead of 4294967296 */
```

  As a workaround, use the `"%ll"` length modifier (for example, `"%lld"` or `"%llu"`) when printing `long` and `unsigned long` values.

```
printf("%lld\n", value);     /* prints 4294967296 */
```

  As a side effect, you will get the warning:

```
warning: format specifies type 'long long' but the argument has type
 'long'
```

  This warning can be suppressed with the `-Wno-format` option for Clang, or the **`/WARNINGS=DISABLE="format"`**qualifier for CXX.

The same problem affects 64-bit pointer values printed with the "%p" conversion: only the low 32 bits of the address are shown.

```
void *p = (void *)0x0000000280001234;  /* a 64-bit address */

printf("%p\n", p);           /* prints 0x80001234 (low 32 bits only) */
```

As a workaround, cast the pointer to unsigned long long and print it with the "%ll" length modifier:

```
printf("%#llx\n", (unsigned long long)p);

/* prints 0x280001234 (full address) */
```

11.1. Using the Clang Command Line

Clang is sensitive to input file extensions. Clang is both a C and a C++ compiler. If you compile a file with the ".C" extension, Clang enters "C mode". If you compile with the ".CPP" (or ".CXX") extension, it will enter "C++ mode". In C-mode, C++ features and libs (even STL) are not available. But there is an -x CL command-line option that can force Clang to switch to the specified language mode. The following command will compile a.c as C++ code:

```
$ clang -x c++ a.c
```

The command-line options are also case-sensitive. However, due to the default parsing rules for upcasing arguments in DCL, you will get errors such as:

```
$ clang except.cpp -Wall -I disk2:[000000]
clang: error: unknown argument '-wall'; did you mean '-Wall'?
clang: error: unknown argument: '-i'
```

The issue here is that DCL with traditional parsing will upcase all the arguments and then the CRTL will downcase all the arguments. You can prevent DCL from upcasing by using double-quotes on the -Wall and -I command-line options.

Alternatively, you can prevent DCL from upcasing and CRTL from downcasing with these definitions (you can put them in your LOGIN.COM):

```
$ set process /parse=extended
$ define/nolog DECC$ARGV_PARSE_STYLE ENABLE
$ define/nolog DECC$EFS_CASE_PRESERVE ENABLE
$ define/nolog DECC$EFS_CHARSET ENABLE
```

Note, however, that these changes may impact other OpenVMS commands and programs.

Clang accepts multiple source files in one command line. For example:

```
$ clang a.cpp b.cpp
```

Clang allows to compile C files as well, according to C99 standard compiling, and linking the example below ends with an undefined symbol.

Example:

```
$ type test.c
#include <stdio.h>
```

```
inline int sumx( int x , int y ) { return 0; }
int main() {
  int x = 8;
  int y = 7;
  printf( "sum = %d\n" , sumx(x,y) );
}
$ clang test.c
$ link test
%ILINK-W-NUDFSYMS, 1 undefined symbol:
%ILINK-I-UDFSYM,  sumx
%ILINK-W-USEUNDEF, undefined symbol sumx referenced
```

The following are possible ways to fix this problem:

- Change `sumx` to a static inline function. This is usually the right solution if only one translation unit needs to use the function. Static inline functions are always resolved within the translation unit, so you will not have to add a non-inline definition of the function elsewhere in your program.
- Remove the `inline` keyword from this definition of `sumx`. The `inline` keyword is not required for a function to be inlined, nor does it guarantee that it will be. Clang treats it as a mild suggestion from the programmer.
- Provide an external (non-inline) definition of `sumx` somewhere else in your program. Be aware that the two definitions must be equivalent.
- Compile in the GNU C89 dialect by adding `-std=gnu89` to the set of Clang options. This option is only recommended if the program source cannot be changed or if the program also relies on additional C89-specific behavior that cannot be changed.

12. Supported and Not Supported DCL Qualifiers

The following table shows the current set of supported DCL qualifiers. Support for the remaining qualifiers will be added in a future release.

Qualifier	Ignored	Supported
/VERSION	-	+
/CLANG	-	+
/COMMENTS	-	+
/DEFINE	-	+
/DEBUG	-	+
/ERROR_LIMIT	-	+
/EXCEPTIONS	-	+
/FIRST_INCLUDE	-	+
/INCLUDE_DIRECTORY	-	+
/LIBRARY	-	+
/LIST	-	+
/L_DOUBLE_SIZE	-	+

Qualifier	Ignored	Supported
/MEMBER_ALIGNMENT	-	+
/NAMES	-	+
/OBJECT	-	+
/OPTIMIZE	-	+
/POINTER_SIZE	-	+
/PREPROCESS_ONLY	-	+
/RTTI	-	+
/SHOW	-	+
/STANDARD	-	+
/UNDEFINE	-	+
/WARNINGS	-	+
/NESTED_INCLUDE_DIRECTORY	-	+
/VERBOSE	-	+
/EXTERN_MODEL	-	+
/MMS_DEPENDENCIES	-	+
/BREAKPOINTS	+	-
/GEMDEBUG	+	-
/DUMPS	+	-
/SWITCHES	+	-
/TRACEPOINTS	+	-
/ANSI_ALIAS	+	-
/ARCHITECHTURE	+	-
/ASSUME	+	-
/BE_DUMPS	+	-
/CHECK	+	-
/DIAGNOSTICS	+	-
/FLOAT	+	-
/G_FLOAT	+	-
/IMPLICIT_INCLUDE	+	-
/ENDIAN	+	-
/EXPORT_SYMBOLS	+	-
/GRANULARITY	+	-
/IEEE_MODE	+	-
/INSTRUCTION_SET	+	-

Qualifier	Ignored	Supported
/LINE_DIRECTIVE	+	-
/LOOKUP	+	-
/MACHINE_CODE	+	-
/MAIN	+	-
/OS_VERSION	+	-
/PENDING_INSTANTIATIONS	+	-
/PREFIX_LIBRARY_ENTRIES	+	-
/PSECT_MODEL	+	-
/PURE_CNAME	+	-
/USING_STD	+	-
/DISTINGUISH_NESTED_ENUMS	+	-
/ALTERNATIV_TOKENS	+	-
/QUIET	+	-
/REPOSITORY	+	-
/REENTRANCY	+	-
/ROUNDING_MODE	+	-
/SHARE_GLOBALS	+	-
/MODEL	+	-
/STACK_CHECK	+	-
/TEMPLATE_DEFINE	+	-
/UNSIGNED_CHAR	+	-
/XREF	+	-
/FE_DUMP	+	-
/BOTH_CASE	+	-

13. Using Clang Command Options From DCL Command Line

A new qualifier has been added that allows you to pass the Clang command-line options when there is no equivalent DCL qualifier:

`/CLANG=(`*`quoted_string`*`[,...])`

This qualifier takes Clang options and passes them to the Clang driver. It is important to specify them in double quotes and separate them with a comma.

Instead of

```
CXX /CLANG=(-Wall, "-D__macro1 -D__MACRO2") foo.cxx
```

specify

```
CXX /CLANG=("-Wall", "-D__macro1", "-D__MACRO2") foo.cxx
```

CXX is a C++ compiler, but you can force C-mode using this qualifier.

For example:

```
CXX /CLANG=("-xc") a.c
```

or

```
CXX /CLANG=("-x", "c") a.c
```

14. Mapping Clang Command Options to DCL Command Options

This section shows Clang options that are similar to **CXX** DCL qualifiers.

/DEFINE

`-D` flag, so **/DEFINE=TRUE** becomes `-DTRUE` which results to `#define TRUE 1`.

/INCLUDE_DIRECTORY

`-I <directory>` – add directory to include search path.

The compiler is extended to support files lookup at predefined logicals locations.

Example:

```
// DISK:[SOURCES]INC.CPP
#include <SYS/INC.HPP>
int main()
{
  f();
}

// DISK:[HEADERS]INC.HPP
void f() {};
```

To compile INC.CPP, it is enough to define the corresponding SYS logical:

```
$ DEFINE SYS DISK:[HEADERS]
```

And then compile INC.CPP:

```
$ SET DEF DISK:[SOURCES]
$ CXX INC.CPP
```

The CXX$SYSTEM_INCLUDE, CXX$LIBRARY_INCLUDE, and CXX$USER_INCLUDE logical names are supported.

If some of them are defined, then the compiler will also do the header search in the directories mentioned by the logicals in the same order as listed above.

Note that CXX$USER_INCLUDE will also impact angle-bracket include directives, which is NOT the same behavior as in IA-64 CXX. The **/ASSUME** qualifier is not supported yet.

/POINTER_SIZE, /NOPOINTER_SIZE

`-no-pointer-size`

- Disables processing of `#pragma pointer_size`
- Predefines the preprocessor macro __INITIAL_POINTER_SIZE to 0
- The initial default pointer size is 64-bit

On Alpha and IA-64, **/NOPOINTER_SIZE** instructs the compiler to assume that all pointers are 32-bit pointers.

`-pointer-size={long|short|64|32|argv64}`

- `argv64` means:
  - The main argument `argv` will be an array of 64-bit pointers. The default is an array of 32-bit pointers.
  - Enables processing of `#pragma pointer_size`
  - Sets the initial default pointer size to 64-bit for translation unit
  - Predefines the preprocessor macro __INITIAL_POINTER_SIZE to 64
- `long` or `64` means:
  - Enables processing of `#pragma pointer_size`
  - Sets the initial default pointer size to 64-bit for translation unit
  - Predefines the preprocessor macro __INITIAL_POINTER_SIZE to 64
- `short` or `32` means:
  - Enables processing of `#pragma pointer_size`
  - Sets the initial default pointer size to 32-bit for translation unit
  - Predefines the preprocessor macro __INITIAL_POINTER_SIZE to 32

The default is `-no-pointer-size` option.

/NAMES=(UPPERCASE,AS_IS)

`-names={uppercase|as_is}` option.

This option controls the conversion of external symbols to the specified case. Two possible options are `uppercase` and `as_is`:

- `uppercase` – uppercases all external symbols in translation unit.
- `as_is` – leaves them as they are. This is default.

/NAMES=(TRUNCATED,SHORTENED)

`-names2={truncated|shortened}` option.

This option controls whether the external names longer than 31 characters get truncated or shortened. Two possible options are:

- `truncated` – Truncates long external names to first 31 characters.
- `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 the dollar sign ("$").

By default, external names can be of any length. IA-64 C++ defaults to **/NAMES=TRUNCATED**.

/[NO]MEMBER_ALIGNMENT

`-[no-]member-alignment`

Clang is extended to support the `-[no-]member-alignment` command-line option. It directs the compiler to align the data structure members naturally. This means that the 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 `-no-member-alignment` causes the data structure members to be byte-aligned (with the exception of bit-field members).

/PENDING_INSTANTIATIONS

`-ftemplate-depth=`*n*

Sets the maximum instantiation depth for template classes to *n*.

/EXTERN_MODEL

The **/EXTERN_MODEL** qualifier takes the following options (the meaning is the same as for the `#pragma extern_model` directive):

```
COMMON_BLOCK
RELAXED_REFDEF (D)
STRICT_REFDEF=["NAME"]
GLOBALVALUE
```

Clang is extended to support the `-extern-model=option` command-line option. Options are the same as for **CXX**, by default Clang does not set any extern model.

/LIBRARY

`-text-library=<`*TEXT_LIBRARY_PATH*`>`

The Clang option to add text library to include the search path.

/LIST

`-listing-file=<LISTING_FILENAME>`

The Clang option to provide the listing filename. Clang does not add extension by default.

/L_DOUBLE_SIZE
/L_DOUBLE_SIZE=128 (D)
/L_DOUBLE_SIZE=*option*

Determines how the compiler interprets the long double type. The qualifier options are 64 and 128.

/L_DOUBLE_SIZE for Clang is analogous to `-mlong-double-[128,64,80]`. 80 is supported by Clang, but OpenVMS does not support 80 bit long double size.

/STANDARD
/STANDARD=(*option*)
/STANDARD=RELAXED (D)

The following new keywords are added to explicitly specify C++ standards: `CXX98`, `GNU98`, `CXX03`, `GNU03`, `CXX11`, `GNU11`, `CXX14`, `GNU14`.

Example:

- **/STANDARD=CXX03** is translated to `--std=c++03`
- **/STANDARD=GNU03** is translated to `--std=gnu++03`

The value `LATEST` is equivalent to `STRICT_ANSI`, which is equivalent to `CXX98`. The default standard for CXX is set to `GNU98`. The equivalent Clang option is `-std=<value>` – language standard to compile for.

/ARCHITECTURE=*option*

Determines the processor instruction set to be used by the compiler. Select one of the **/ARCHITECTURE** qualifier options shown below.

`X86=option`	Similar to Clang's `-march` option. For example: `$ cxx /arch=x86="skylake" hw.cxx`
`GENERIC`	Ignored.
`HOST`	Ignored.
`ITANIUM2`	Ignored.
`EV4`	Ignored.
`EV5`	Ignored.
`EV56`	Ignored.
`PCA56`	Ignored.
`EV6`	Ignored.
`EV68`	Ignored.

`EV7`	Ignored.

/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. This is not supported for x86-64.
INCLUDE_FILE	Directs the compiler to search the directory containing the file in which the `#include` directive 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 effect, **/ASSUME** is not supported for x86-64). 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.

/FIRST_INCLUDE

`-include <`*`file`*`>` – include file before parsing.

/UNDEFINE

`-U <`*`macro`*`>` – undefine macro <*macro*>.

/LINE_DIRECTIVES

`-P` – disable linemarker output in preprocessing mode.

/UNSIGNED_CHAR

For the compiler, **/UNSIGNED_CHAR** is respective to:

- `-fno-signed-char` – char is unsigned.
- `-fsigned-char` – char is signed.

/COMMENTS

`-C` – includes comments in preprocessed output.

/ERROR_LIMIT
/ERROR_LIMIT[=*number*]
/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=20**, which specifies that compilation terminates after issuing 20 error messages.

`-ferror-limit=`*n* – stops emitting diagnostics after **n** errors have been produced.

/OBJECT

`-o <`*file*`>` – writes output to the file. Default object file has .OBJ extension.

/PREPROCESS_ONLY

`-E` – only run the preprocessor.

/RTTI
/RTTI (D)
/NORTTI

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.

Clang has similar options:

`-fno-rtti` – disable generation of rtti information.

`-frtti` – enable generation of rtti information (default).

Clang does not define the __RTTI macro.

/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]EXPANSION	Expand preprocessor `#if` and `#elif` macros
[NO]STATISTIC	Ignored

Clang's corresponding options are:

- `-show-header`
- `-show-include`
- `-show-source`
- `-show-expansion`

There are no equivalent `-show` options that correspond to the **/SHOW=NOHEADER** or **/SHOW=NOSOURCE**. None of these options are enabled by default for Clang.

/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 warnings. The **/NOWARNINGS** qualifier suppresses warning messages.

The message-list in the following table of options can be any one of the following:

- A single message identifier in double quotes (within parentheses, or not).
- A comma-separated list of message identifiers in double quotes, enclosed in parentheses.
- The keyword "all".

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

NOWARNINGS	Suppresses warnings. Similar to Clang's `-w` option.
NOINFORMATIONALS	Has no effect.
ENABLE=*message-list*	Enables issuance of the specified messages. Can be used to enable specific messages that normally would not be issued when messages disabled with **/WARNINGS=DISABLE**. Specify `all` to enable all warnings. Similar to Clang's `-W<warning>` option.
DISABLE=*message-list*	Disables issuance of the specified messages. Specify `all` to suppress all warnings. Similar to Clang's `-Wno-<warning>` option.
INFORMATIONALS=*message- list*	Has no effect.

WARNINGS=*message-list*	Do not error out on the specified warnings. Similar to Clang's `-Wno-error=<warning>` option.
[NO]ANSI_ERRORS	Has no effect.
[NO]TAGS	Has no effect.
ERRORS=*message-list*	Sets the severity of the specified messages to Error. Similar to Clang's `-Werror=<warning>` option.

Important

CXX uses Clang message IDs. For example, if Clang issues a warning such as:

```
warning: non-void function does not return a value [-Wreturn-type]
```

The ID of the message is `return-type`.

All message IDs should be given to CXX as they are in double quotes to preserve the case. This is true for the keyword `all` as well.

There is a list of warnings in Clang that are turned off by default and are enabled with their specific ID or with the keyword `all`. The VMS-specific warning `cast from long pointer to short pointer will lose data` is off by default. It can be enabled with the ID `may-lose-data` or with `all`.

/EXCEPTIONS

`-fexceptions` – enable support for exception handling.

`-fno-exceptions` – disable support for exception handling.

/OPTIMIZE[=*option*]
/OPTIMIZE=(LEVEL=*n*)
/NOOPTIMIZE (D)

LEVEL=*n* – Selects the level of code optimization. Specify an integer from 0 (no optimization) to 5 (full optimization). Similar to Clang's `-On` option, where `n=[0,4]` and `LEVEL=5` is translated to `-Ofast`.

[NO]INLINE – Similar to Clang's `-finline`/`-fno-inline` options.

Clang's options:

`-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4`

`-O0`	Means "no optimization": this level compiles the fastest and generates the most debuggable code.
`-O1`	Somewhere between `-O0` and `-O2`.
`-O2`	Moderate level of optimization which enables most optimizations.
`-O3`	Like `-O2`, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

`-Ofast`	Enables all the optimizations from `-O3` along with other aggressive optimizations that may violate strict compliance with language standards.
`-Os`	Like `-O2` with extra optimizations to reduce code size.
`-Oz`	Like `-Os` (and thus `-O2`), but reduces code size further.
`-Og`	Like `-O1`. In future versions, this option migh disable different optimizations in order to improve debuggability.
`-O`	Equivalent to `-O2`.
`-O4` and higher	Currently equivalent to `-O3`

/VERBOSE

With this qualifier, the driver first prints out the fully generated command line which is going to Clang, and then runs the compiler.

/PREFIX_LIBRARY_ENTRIES

Only the **/NOPREFIX_LIBRARY_ENTRIES** option of this qualifier has an effect. Clang similar option for that option is `-noprefix-lib-entries`. Other options of this qualifier have no effect on OpenVMS x86-64.

15. Threads and Thread-Local Variables

OpenVMS does not currently support the `thread_local/__thread/_Thread_local` declaration specifiers provided by the Clang compiler. As an alternative, use the POSIX API functions, such as `pthread_key_create`, `pthread_setspecific`, `pthread_getspecific`, or `pthread_key_delete`.

16. Online Clang Documentation

The full documentation about the Clang compiler is available at the links below:

https://releases.llvm.org/10.0.0/tools/clang/docs/index.html
https://releases.llvm.org/10.0.0/tools/clang/docs/ClangCommandLineReference.html