VSI C++ V10.1-3 for OpenVMS x86-64

Release Notes

Publication Date: November 2025

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

Kit Name: VSI-X86VMS-CXX-V1001-3-1.ZIP

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

VSI C++ V10.1-3 for OpenVMS x86-64 Release Notes

Legal Notice

Table of Contents

1. Kit Name

VSI-X86VMS-CXX-V1001-3-1.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 6, "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.

OpenVMS V9.2-3 contains up-to-date versions of LIBCXX and LIBCXXABI. The C++ kit no longer attempts to place these RTLs on the system. To avoid confusion during the upgrade, this kit will also not provide the RTLs on older OpenVMS V9.2-1 and V9.2-2 systems. In order to correctly track the RTL versions and ownership, the package also includes a second PCSI kit that provides the correct RTLs but makes sure that they are correctly tracked by the PCSI utility.

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

The C++ package includes a CXXFIXUP kit that provides the correct run-time libraries if needed.

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 10.1-3 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.
- 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. 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.

6.1. 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.