Intel® Compilers for Linux*: Application Porting Guide

White Paper

Om P Sachan

Abstract

This paper describes application porting when using Intel® Compilers for Linux*. The Intel C/C++ compiler is compatible to the GNU* compilers in terms of source, binary and command-line compatibility. The Intel® C/C++ and Fortran Compilers help make your applications run at top speed on Intel's platforms, including those based on the IA-32, Intel® 64 and Intel® Xeon Phi architectures. The compilers also provide compatibility with commonly used Linux* software development tools**.**

Table of Contents

Introduction

The Intel® Compiler XE 2016 interoperates with GNU Compilers and tools to provide very high levels of performance across the full spectrum of Intel® processor technologies. You can rebuild your existing GNU compiler applications with the Intel compilers without having to make signficant source code changes by simply replacing the GNU compiler with the correspoinding Intel compiler. These compilers correspond to each other in the following ways:

LLanguage	Intel® Compiler	GCC* Compiler
C	cc	Gcc
C++	icpc	g++
Fortran	ifort	Gfortran

In many cases, porting applications from gcc to the Intel® C++ Compiler can be as easy as modifying your makefile to invoke the Intel® C++ Compiler instead of gcc. Using the Intel® C++ Compiler typically improves the performance of your application, especially for those applications that run on Intel processors. In many cases, your application's performance may also show improvement when running on non-Intel processors. When you compile your application with the Intel® C++ Compiler, you have access to:

- compiler options that optimize your code for the latest Intel® architecture processors.
- advanced profiling tools (PGO) similar to the GNU profiler, gprof.
- high-level optimizations (HLO).
- interprocedural optimization (IPO).
- Intel intrinsic functions that the compiler uses to inline instructions, including Intel® SSE2, Intel® SSE3, SSSE3, Intel® SSE4 Intel® AVX and Intel® AVX-512.
- a highly-optimized, highly accurate Intel® Math Library .

The Intel® C++ Compiler creates object files that are   binary compatible with object files that are created by gcc. As a result, when you are porting your applications from gcc to icc, you should not have to re-build the libraries that were built with gcc. The Intel® C++ Compiler also supports many of the same compiler options, macros, and environment variables that you already use in your gcc work.

Setting the Intel Compiler Environment

The Intel® C++ Compiler relies on environment variables to locate compiler binaries, libraries, man pages, and license files. In some cases these environment variables are different from the environment variables that gcc* uses. With the Intel® C++ compiler, unlike gcc,   these variables are not set by default after installing the compiler.. The following environment variables need to be set prior to running the Intel® C++ Compiler:

- PATH – adds the location of the compiler binaries to PATH.
- LD_LIBRARY_PATH (Linux*) – sets the location of the compiler libraries as well as the resulting binary generated by the compiler.
- DYLD_LIBRARY_PATH (OS X*) – sets the location of the compiler libraries as well as the resulting binary generated by the compiler.
- MANPATH – adds the location of the compiler man pages (**icc** and **icpc**) to MANPATH.
- INTEL_LICENSE_FILE – sets the location of the Intel® C++ Compiler license file.

To set these environment variables, run the compilervars.sh script. Setting these environment variables with compilervars.sh does not create a conflict with gcc. You should be able to use both compilers in the same shell.

Modifying makefiles

If you use makefiles to build your gcc* application, you need to change the value of the **CC** compiler variable to use the Intel® C++ compiler. You may also want to review the options specified by **CFLAGS**. A simple example follows:

Sample gcc* makefile

```
# Use gcc compiler
  CC = gcc

# Compile-time flags
  CFLAGS = -O2 -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c  $(CFLAGS) area_functions.c
```

Sample gcc* makefile

```
clean: rm -rf *o area
```

Sample makefile modified for the Intel® C/C++ Compiler

```
# Use Intel C compiler
  CC = icc

# Compile-time flags
  CFLAGS = -std=c99
all: area_app

area_app: area_main.o area_functions.o
    $(CC) area_main.o area_functions.o -o area

area_main.o: area_main.c
    $(CC) -c $(CFLAGS) area_main.c

area_functions.o: area_functions.c
    $(CC) -c  $(CFLAGS) area_functions.c

clean: rm -rf *o area
```

If your gcc* code includes features that are not supported with the Intel® C++ Compiler, such as compiler options, language extensions, macros, pragmas, and so on, you can compile those sources separately with gcc* if necessary.

The Intel® C++ Compiler uses the **O2** option by default and gcc* uses the O0 option by default; you can specify the option O0 with icc if required.

Using Optimization

The Intel® C++ Compiler is a highly optimizing compiler. With icc, there is an implicit assumption that   superior application performance on an Intel® architecture is important. Consequently, certain optimizations, such as option **O2**, are part of the default invocation settings for the Intel® C++ Compiler. Optimization is turned off in gcc by default, the equivalent of compiling with options **O** or **O0**. Other forms of the **O<*n*>** option compare as follows:

Option	Intel® C++ Compiler	gcc*
-O0	Turns off optimization.	Default. Turns off optimization.
-O1	Decreases code size with some increase in speed.	Decreases code size with some increase in speed.
-O2	**Default.** Favors speed optimization with some increase in code size. Same as option O. Intrinsics, loop unrolling, and inlining are performed.	Optimizes for speed as long as there is not an increase in code size. Loop unrolling and function inlining, for example, are not performed.
-O3	Enables option O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations.	Optimizes for speed while generating larger code size. Includes option O2 optimizations plus loop unrolling and inlining. Similar to option O2 -ip on the Intel® C++ Compiler.

Targeting Intel® Processors

While many of the same options that target specific processors are supported with both compilers, Intel includes options that utilize processor-specific instruction scheduling to target the latest Intel® processors. If you compile your gcc* application with the -march or -mtune option, consider using the Intel® C++ Compiler -x or -ax options for applications that run on IA-32 architecture or Intel® 64 architecture.

Modifying Configurations

The Intel® C++ Compiler lets you maintain configuration and response files that are part of the compilation process. Options stored in the configuration file apply to every compilation, while options stored in response files apply only when they are added on the command line. If you have several options in your makefile that apply to every build, you may find it easier to move these options to the configuration file (../bin/icc.cfg and ../bin/icpc.cfg).
In a multi-user, networked environment, options listed in the icc.cfg and icpc.cfg files are generally intended for everyone who uses the compiler. If you need a separate, single user, configuration, you can use the ICCCFG or ICPCCFG environment variable to specify the name and location of your own .cfg file, such as /my_code/my_config.cfg. Anytime you instruct the compiler to use a different configuration file, the system configuration files (icc.cfg and icpc.cfg) are ignored.

Using the Intel® Libraries

Additional libraries that contain optimized implementations of many commonly used functions are supplied with the Intel C++ Compiler. Some of these functions in the libraries are implemented using CPU dispatch. With CPU dispatch, different code may be executed when run on different processors.

Supplied libraries include the Intel® Math Library (*libm*), the Short Vector Math Library (*libsvml*), *libirc*, as well as others. These libraries are linked in by default when the compiler sees that references to them have been generated. Some library functions, such as sin or memset, may not require a call to the library, since the compiler may inline the code for the function.

Using Optimized Math Function Library (libimf)

The math function library, libimf, is an optimized math library provided with the Intel Compilers. The Intel Compilers link the libimf library first, and then the libm library. If an optimized version of a function is available, the linker will find it in libimf. Otherwise, it will use the default version of the function in the default Linux math library, libm.

Note: The Intel compiler always links to functions in the optimized libimf library if available, regardless of whether the user specifies –lm on the link command line or if the user adds both libimf and libm on the link line, regardless of the order.

Short Vector Math Library (libsvml)

When vectorization is being done, the compiler may translate some calls to the *libm* math library functions into calls to *libsvml* functions. These functions implement the same basic operations as the Intel® Math Library, but operate on short vectors of operands. This results in greater efficiency. In some cases, the *libsvml* functions are slightly less precise than the equivalent *libimf* functions.

libirc

libirc contains optimized implementations of some commonly used string and memory functions. For example, it contains functions that are optimized versions of memcpy and memset. The compiler will automatically generate calls to these functions when it sees calls to memcpy and memset. The compiler may also transform loops that are equivalent to memcpy or memset into calls to these functions.

Compatibility with GNU Compiler

The Intel® C++ Compiler XE 2016 product is the successor to the Intel® C++ Composer XE 2015 product. The Intel® C and C++ compilers for Linux* provide excellent source, binary, and command-line compatibility with the GNU* gcc and g++ compilers. The Intel C/C++ compilers for Linux generate C/ C++ code that is binary compatible with gcc/g++ and requires a Linux

distribution with a released version of gcc/g++ 4.1 or higher. In addition, compared to previous versions, the Intel C/C++ compilers for Linux continue to improve the degree of compatibility with the gnu gcc/g++ compilers, including the degree of compiler option compatibility.
Compatibility has been improved in multiple areas, some as a result of customer requests. Some notably improved areas include:

- Improved gcc/g++/icc/icpc binary file interoperability , including third-party library interoperability with gcc/g++ builit libraries.
- Support for generating C++ code compatible with g++ 4.1 or higher
- Support for building the Linux kernel with the Intel C++ Compiler
- Improved support for command-line options and build environments availabe with the GNU compilers

The Intel® C/C++ compilers for Linux* support ANSI and ISO C and C++ standards, most GNU* C and C++ language extensions, and the OpenMP* 4.0 standard. Intel® C++ Compiler XE 16 supports many features that are in the C++11 and C++14 standard - consult the compiler documentation and C++11 Features Supported by Intel® C++ Compiler for details. The object and binary files created by the Intel C/C++ Linux compiler are compatible with the C and C++ binary files created by the GNU gcc and g++ compilers versions 4.1 and later.
Building the Linux kernel with the Intel C/C++ Linux compiler is an ongoing effort at Intel. Some of the objectives of the effort are to improve the degree of GNU source level compatibility, and to improve kernel performance.

The Intel® Fortran Compiler 16.0 supports the Fortran 2003 and OpenMP* 4.0 standards, as well as many features from Fortran 2008. For more details, consult the compiler release notes.

The Intel Fortran Compiler for Linux does not generate object files that are binary compatible with object files generated by the GNU g77 or GNU gfortran compilers, nor is binary compatibility a future goal of the Intel Fortran compiler. Instead, the Intel Fortran Compiler for Linux generates object files that are binary compatible with C-language object files generated with either the Intel C++ Compiler for Linux or the GNU gcc compiler.
Intel's C/C++ and Fortran Compilers support a large number of the commonly used GNU compiler command-line options. See the *Command Line Options* section of this document for details.

Intel® Compiler XE 2016 does not support development on or for IA-64 architecture (Intel® Itanium®) systems. Intel® Compiler version 11.1 remains available to developt applications for the IA-64 architecture..

We encourage customers to submit feature requests to improve compatibility through their Intel® Premier Support accounts. See *Customer Feedback* below.

C and C++ Source Compatibility

Intel's C/C++ Linux Compiler supports ANSI and ISO C and C++ standards. The compiler also supports most GNU* C and C++ language extensions. The Intel compiler predefines the following macros based on the version of gcc available at compilation time:

- __GNUC__
- __GNUC_MINOR__
- __GNUC_PATCHLEVEL__
- __GNUG__

You can disable these GNU predefined macros using the -no-gcc option; however, using this option is not recommended because the system header files may not compile correctly. A new option, -gcc-sys, has been added that is similar to the -no-gcc option, except that when you use the new option the GNU macros are only defined when the compiler is preprocessing system header files, so that the code compiles correctly. You can also use the Intel C/C++ Compiler macro, __INTEL_COMPILER to conditionally compile code.

GNU inline-assembly format is supported on IA-32 and Intel® 64 architecture-based processors. The Intel C/C++ Compiler implements a large number of gcc built-in functions, which are documented at http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. See the compiler documentation for information on intrinsic functions and supported gcc built-in functions.

The Intel® compiler automatically detects the version of gcc being used. If a user specifies the -gcc-name=/usr/bin/gcc43 (or gcc44, etc.), the compiler will automatically set the gcc-version accordingly. Most users should never have a need to set it. By default, the Intel® compiler uses the version of gcc defined in the PATH environment variable. Explicitly setting -gcc-name may be useful when you want to generate code compatible with different versions of gcc installed on your system.

For more recent versions of g++, contact Intel through your Intel® Premier Support account for additional information.

Starting in version 11.0, the Intel(R) C++ Compiler has supported some of the C++11 features. With the latest release of Intel C++ Compiler XE for Linux* and Mac OS* X more C++11 features are supported. The detailed supported features are listed at http://software.intel.com/en-us/articles/c0x-features-supported-by-intel-c-compiler/.

OpenMP* Compatibility and Interoperability

The term *object-level interoperability* refers to the ability to link object files generated by one compiler with object files generated by a second compiler, such that the resulting executable runs correctly. We call an OpenMP program *compatible* with a compiler if the level of OpenMP support required by the program is less than or equal to the level of OpenMP support offered by the compiler.
The Intel® Compiler, versions 11.0 and later, are object level interoperable with GNU* compiler versions 4.1 and later. Versions 11.0 and 12.0 provide support for OpenMP 3.0 and versions 12.1, 13.0 provide support for OpenMP 3.1. Intel compiler version 14.0 has added partial support for OpenMP 4.0 and Intel compiler version 16.0 supports OpempMP 4.0 completely.
You should determine the level of OpenMP support that each file of your program requires, and use a compatible and object-level interoperable compiler to compile the code, producing one or more object files. You should then link all object files together with a single compiler, to produce the executable file. The compiler you choose for the link step will determine which runtime libraries are used during execution.

Binary Compatibility

Intel® C++ Compiler XE 16.0 provides C and C++ binary compatibility with released versions of gcc including gcc 4.1 and later.   The default C++ library implementation uses the gcc provided C++ libraries. The Intel C++ Compiler for Linux and GNU compilers both conform to the C++ ABI (Application Binary Interface), a convention for binary object code interfaces between C++ code and the implementation-provided system and runtime libraries. Consult http://gcc.gnu.org/releases.html for information on the latest gcc releases. The goal is to implement a stable ABI for C++ applications and libraries, which will benefit the Linux community.
Intel C++ Compiler XE 16.0 uses the C++ runtime libraries/files provided by gcc. The gcc C++ runtime libraries/files include the libstdc++, standard C++ header files, library, and language support. By default, C++ code generation is compatible with the code generation of the version of gcc specified in the PATH environment variable. The compiler option –gcc-name allows specifying the full-path location of gcc. Use this option if you are using a version of gcc that has a non-standard installation location.
The following examples demonstrate binary compatibility with g++ and usage of the g++ runtime libraries. Figure 1 shows the building of the “Hello World” example program.

```
prompt> cat hello.cxx
  #include <iostream>
  int main(){ std::cout<<“Hi”<<std::endl; }
prompt> icpc hello.cxx
```

Figure 1.   "Hello World" Example using the default run time libraries

Figure 2 shows the default run-time libraries that are linked against when building applications on an Intel® 64 architecture-based system, including the g++ provided C++ runtime libraries. The g++ libraries are the standard GNU* C++ library, libstdc++, and C++ language support libraries.

The system utility, ldd, is used to show the dynamic libraries linked to the application, and the awk utility is used to make the output easier to read in this paper.

```
prompt> ldd a.out | awk '{print $1}'
linux-vdso.so.1
libm.so.6
libstdc++.so.6
libgcc_s.so.1
libc.so.6
libdl.so.2
/lib64/ld-linux-x86-64.so.2
```

Figure 2. Default run time libraries linked against "Hello World" example program

The example in Figure 3 demonstrates C++ binary compatibility with g++ by mixing binary files created by g++ and the Intel C++ Compiler.

```
prompt> cat main.cxx
void pHello();
int main() { pHello(); }
prompt> cat pHello.cxx
#include <iostream>
void pHello()
{ std::cout << "Hello" << std::endl; }
prompt> icpc –c main.cxx
prompt> g++ –c pHello.cxx
prompt> icpc main.o pHello.o
prompt> ./a.out
Hello
```

Figure 3. Mixing Binary Files Created by g++ and the Intel® C++ Compiler

Figure 4 shows how initially mixing binary files created by g++ and by the Intel C++ Compiler and using g++ to link them fails. Intel recommends linking by invoking the Intel Compiler to correctly pass the Intel libraries to the system linker, ld. On IA-32 and Intel® 64 architecture-based processors, the Intel C++

Compiler calls the function __intel_new_proc_init from the main routine to determine the ability to run processor-specific code. This routine is found in the Intel library libirc. To link this example correctly, libirc needs to be added to the link command. Different compiler optimizations, such as OpenMP* and vectorization, may require additional libraries provided by the Intel C/C++ Compiler.
Linking with the Intel Compiler removes the necessity of knowing the details of which Intel libraries are required, provided the same compiler options are used when compiling and linking. Without passing the correct libraries and library location, the initial link fails. For the example in Figure 4, linking with the Intel libirc library is required.

```
prompt> icpc –c main.cxx
prompt> g++ –c pHello.cxx

# Fails without Intel provided libraries
prompt> g++ main.o pHello.o
main.o: In function `main':
main.cxx:(.text+0x17): undefined reference to `__intel_new_feature_proc_init'
collect2: ld returned 1 exit status

# Links with g++ and explicitly list
# Intel provided libraries in the default
# library location for release 13.0
prompt>  g++ main.o pHello.o -L/opt/intel/compilers_and_libraries_2016.0.089/linux/compiler/lib/intel64/ -
lirc
prompt> ./a.out
Hello

# Recommended usage
prompt> icpc main.o pHello.o
prompt> ./a.out
Hello
```

Figure 4. Example of Linking Using g++

Linking and Libraries

Linking C Language with Intel® Compiler and gcc Compiler

C-language object files can be linked with either Intel Compilers or gcc compilers. Linking with an Intel Compiler is recommended, as the Intel libraries will be correctly passed to the linker. The examples shown in Figure 5 through 10 use gcc to link the application with the file main.c, compiled with gcc, and the file calcSin.c, compiled with the Intel C/C++ Compiler.

Figure 5 illustrates code that uses the Intel Compiler to automatically vectorize the loop in calcSin.c when the C99 restrict keyword is used and enabled via a command-line option.

```
prompt> cat calcSin.c
#include <math.h>
void calcSin(double *a,double * restrict b, int N)
{
        int i;
        for (i=0; i<N; i++)
              b[i] = sin(a[i]);
}
prompt>  icc -c  -xHost  calcSin.c -qopt-report -qopt-report-phase=vec -restrict
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
prompt>cat calcSin.optrpt
Intel(R) Advisor can now assist with vectorization and show optimization report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.
Begin optimization report for: calcSin(double *, double *__restrict__, int)

 Report from: Vector optimizations [vec]
LOOP BEGIN at calcSin.c(4,3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at calcSin.c(4,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at calcSin.c(4,3)
<Remainder loop for vectorization>
LOOP END
===============================================================================
```

Figure 5. Compiling C Language Function with the Intel® C++ Compiler, using the -restrict command line option to enable usage of the C99 keyword restrict.

The main function is compiled with gcc as shown by code in Figure 6.

```
prompt> cat main.c
```

```
void calcSin(double *a,double *b,int N);
 int main() {
  const int N=100000;
  double a[N], b[N], c[N], x[N];
  int i;
  for (i=0;i<N;i++)
    a[i] = i;
  for (i=0;i<100;i++)
    calcSin( a, b, N); }
prompt> gcc -c main.c
```

Figure 6. Main Function Compiled with gcc

The application is linked with gcc and the necessary libraries from the Intel Compiler are passed to gcc. In the example in Figure 7, the short vector math library, libsvml, and the optimized math function library, libimf, both provided by Intel, are required to link with gcc.

```
prompt> gcc main.o calcSin.o -
L/opt/intel/compilers_and_libraries_2016.0.089/linux/compiler/lib/intel64/ -lirc -lsvml -limf -o
calSin
```

Figure 7. Linking With gcc Using Additional Intel Provided Libraries: libsvml and libimf

If the Intel libraries are not passed to gcc, the link fails with undefined references, as shown in Figure 8.

```
# Not recommended
prompt> gcc main.o calcSin.o  -o calSin
calcSin.o: In function `calcSin':
calcSin.c:(.text+0x96): undefined reference to `__svml_sin4'
calcSin.c:(.text+0xc5): undefined reference to `__svml_sin4'
calcSin.c:(.text+0xf0): undefined reference to `__svml_sin4'
collect2: ld returned 1 exit status
```

Figure 8. Failed Link Due to Missing Libraries When Linking With gcc

The nm utility can determine the names of unresolved symbols provided in the Intel libraries. Running the nm utility on the Intel libraries helps determine which libraries are required. See the Intel C++ and Fortran User's Guides, library section, for documentation on the different Intel libraries. Next, the grep utility (Figure 9) verifies that the symbols are defined in the Intel libraries. Examine the symbol file, icc-symbols.txt, to determine which library contains the missing symbols and then addthem to the link options.

```
prompt> nm
/opt/intel/compilers_and_libraries_2016.0.089/linux/compiler/lib/intel64* >
icc-symbols.txt
```

```
prompt> grep "__svml_sin4$" icc-symbols.txt
0000000000000000 T __svml_sin4
```

Figure 9. Utilities to Determine Libraries Required to Link with gcc Correctly

Link with the Intel Compiler, passing the same options (-xHost in Figure 10) used during compilation, to avoid the necessity of finding out which Intel supplied libraries are required.

```
#Recommended usage
prompt> gcc -c main.c
prompt> icc -c -xHost  calcSin.c -restrict
prompt> icc calcSin.o main.o
```

Figure 10. Avoiding Missing Libraries when Linking: Use Intel® C++ Compiler with the Same Options as Used During Compilation

Intel® Compiler Linking Conventions/Changing Default Linking Behavior

This section describes linking conventions used by the Intel Compilers for Linux and compiler options for changing the default behavior. The icc compiler driver links the C++ runtime libraries if and only if the input source files have C++ file extensions. The icpc compiler driver is meant to be used for C++ files, and it automatically links in the C++ runtime libraries, similar to the behavior of g++.

By default, Intel Compilers for Linux use the Dynamic Shared Object (DSO) versions, also known as shared libraries, of the Linux system libraries. For the Intel provided libraries, by default, the DSO versions of OpenMP and libcxaguard, the g++ compatibility support libraries are used, and static versions of all other Intel libraries are used. With user-provided libraries, the compiler searches for a DSO version first. If no DSO version is found, it searches for a static version.

The following command line options modify the default linking behavior:

- –Bstatic: Uses the static version of all libraries specified after this point or until the option "–Bdynamic" is used. This option can be used to statically link all libraries.
- –Bdynamic: Uses dynamic (DSO) version of all libraries specified after this point or until the "–Bstatic" option is used. Note that –Bstatic and –Bdynamic are toggles.
- -static-intel: Statically links all compiler libraries provided by Intel. Use this option to avoid the need to redistribute the Intel compiler libraries with your application.
- -shared-intel: Dynamically links all compiler libraries provided by Intel. In other words, use DSO versions of Intel libraries.
- –shared: Instructs the linker to create a DSO instead of an application binary.

The ldd utility lists the DSOs that an application is linked with and is useful in understanding the command-line options described previously.

Build Environment Enhancements

The Intel C++ Compiler continues to improve compatibility with different gcc build environments. The ability to use the GNU C++ library on systems with non-standard gcc configurations has been improved. The Intel Compiler installation supports the traditional installation process with root account access using rpm as well as the non-root account installation process that doesn't use the rpm package manager.

The following GNU environment variables are supported:

- CPATH: Path to include directory for C/C++ compilations.
- C_INCLUDE_PATH: Path to include directory for C compilations.
- CPLUS_INCLUDE_PATH: Path to include directory for C++ compilations.
- DEPENDENCIES_OUTPUT: If this variable is set, its value specifies how to output dependencies for make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.
- GCC_EXEC_PREFIX: This variable specifies alternative names for the linker (ld) and assembler (as).
- LIBRARY_PATH: The value of LIBRARY_PATH is a colon-separated list of directories, much like PATH.
- SUNPRO_DEPENDENCIES: This variable is the same as DEPENDENCIES_OUTPUT, except that system header files are not ignored.
- GXX_INCLUDE: Specifies the location of the gcc headers. Set this variable only when the compiler cannot locate the gcc headers when using the -gcc-name option.
- GXX_ROOT: Specifies the location of the gcc binaries. Set this variable only when the compiler cannot locate the gcc binaries when using the -gcc-name option.

Command Line Options

The Intel C++ Compiler for Linux supports a large number of common GNU* compiler command-line options. Information on Intel C++ command options can be found in the compiler documentation, compiler man pages, and summary information via icc –help.

The following options have been added that deal with GNU compatibility. The option -gcc-sys is similar to -no-gcc (which tells the compiler not to predefine the __GNUC__, __GNUC_MINOR__, and __GNUC_PATCHLEVEL__ macros), except that the GNU macros are only defined when preprocessing system header files. The option -pragma-optimization-level=[Intel|GCC] enables / disables processing #pragma optimize using Intel (default) or gcc syntax.

Intel compiler supports strict ansi checking; -ansi provides compatibility with gcc's –ansi switch and –strict-ansi provides more strict ansi checking than is available from gcc.

Note that the Intel C++ and Fortran Compilers have a large number of features to optimize applications for the latest Intel processors. The topic “**Optimization and Programming Guide” in “Intel® C++ Compiler 16.0 User and Reference Guide**” explains how to use Intel Compilers to optimize for the latest Intel processors.
Intel encourages customers to submit feature requests for command-option compatibility through their Intel Premier Support account. See *Customer Feedback* below.

Intel® Fortran Compiler

The Intel® Fortran Compiler supports all of the Fortran 2003 standard as well as a large part of Fortran 2008, including coarrays and DO CONCURRENT. The Intel Fortran Compiler for Linux is not binary compatible with the GNU g77 or gfortran compiler. In general, Fortran compilers are not binary compatible because they use different runtime libraries. Intel Fortran Compiler is binary compatible with C-language object files created with either the Intel C++ Compiler for Linux or the GNU gcc compiler. Your application can contain both C and Fortran source files. If your main program is a Fortran source file (myprog.for) that calls a routine written in C (cfunc.c), you can use the following sequence of commands to build your application.

```
#Recommended usage
prompt> icc -c cfunc.c
prompt> ifort -c myprog.for
prompt> ifort -o myprog myprog.o cfunc.o
```

Figure 11. Linking: C Compiler and Fortran compiler generated objects files

Use the -nofor_main compiler option if your C/C++ program contains the main() entry point and is calling an Intel® Fortran subprogram. The *Intel Fortran Compiler* documentation has further details on calling C language functions from within Fortran code.

The Intel Fortran Compiler for Linux uses a different name-mangling scheme than the GNU Fortran compiler. Intel does not recommend mixing object files created by the Intel Fortran Compiler and the GNU Fortran compiler.

Customer Feedback

Intel is committed to providing compilers that deliver the highest Linux-based application performance for applications running on platforms that use the latest Intel processors. Intel Premier Support is included with every purchased compiler; see https://software.intel.com/en-us/intel-parallel-studio-xe-support for more information.

Intel strongly values customer feedback and is interested in suggestions for improved compatibility with the GNU compilers. If your applications require additional compatibility features, please submit a feature request by creating a customer-support issue through your Intel Premier Support account that explains your request and its impact.

Developers should register for an Intel Premier Support account to obtain technical support and product updates. The product-release notes describe how to register.

References

- General information on Intel® software development tools, including the Intel C++ and Fortran Compilers: https://software.intel.com/en-us/intel-parallel-studio-xe
- Intel® C++ and Fortran Compiler documentation is available at https://software.intel.com/en-us/intel-parallel-studio-xe-support/documentation
- The OpenMP standard: http://www.openmp.org
- ANSI C and C++ standards: http://www.ansi.org
- The GNU project including GNU gcc and gfortran compilers and glibc, the GNU C library: http://www.gnu.org
- Conventions for object code interfaces between C++ code and implementation-provided system and libraries: http://mentorembedded.github.com/cxx-abi/