# Intel® Compilers for Linux*: Compatibility with GNU Compilers

White Paper

**Om P Sachan**

Developer Products Division

## Abstract

This paper describes the compatibility between Intel® Compiler  for Linux* and GNU* compilers in terms of source, binary, and command-line compatibility. The Intel® C++ and Fortran Compiler help make your software run at top speed on Intel's platforms, including those based on IA-32, Intel® 64 architectures. The compilers also provide compatibility with commonly used Linux* software development tools.

## Table of Contents

## Introduction

Intel® C++ Composer XE 2011 is the successor to Intel® C++ Compiler Professional Edition 11.1. Intel® C++ Composer XE 2011 for Linux* provides excellent source, binary, and command-line compatibility with the GNU* gcc and g++ compilers. Intel® C++ Composer XE 2011 update 6 above includes the Intel C++ Compiler XE 12.1. Intel C++ Compiler for Linux now generates C++ code that is binary compatible with g++ and requires a Linux distribution with a released version of g++ 3.4 or higher. In addition, compared to previous versions, the Intel C++ Compiler for Linux has improved compiler option compatibility with gcc and has made general improvements related to the gcc build environment.

The motivation for improving compatibility comes from customer requests in several areas:

- Mixing and matching binary files created by g++, including third-party libraries
- Generating C++ code compatible with g++ 3.4 or higher
- Building the Linux kernel with the Intel Compiler
- Using command-line options offered in the GNU compilers

The Intel® C++ Compiler for Linux* supports ANSI and ISO C and C++ standards, most GNU* C and C++ language extensions, and the OpenMP* 3.1 standard. Intel® C++ Composer XE 12.1 supports some features that will be in the future C++0x standard, consult the compiler documentation for details. The Intel C++ Linux compiler is binary and object file compatible with C and C++ binary files created with GNU gcc and g++ versions 3.4 and later.

Building the Linux kernel with Intel C++ Linux compiler is an ongoing project at Intel with the goal to improve GNU source compatibility of the compiler, and to find opportunities to improve kernel performance.

The Intel® Fortran Composer XE 2011 for Linux supports the Fortran 95 and OpenMP* 3.1 standards. Intel® Fortran Composer XE 2011 includes the Intel Fortran Compiler XE 12.1. The Intel Fortran Compiler for Linux also supports some features from the Fortran 2003 standard., consult the compiler documentation for details.

The Intel Fortran Compiler for Linux is not binary compatible with GNU g77 or GNU gfortran compiler, nor is binary compatibility a future goal of the Intel Fortran compiler. Instead, the Intel Fortran Compiler for Linux is binary compatible with C-language object files created with either Intel C++ Linux Compiler or the GNU gcc compiler.

Intel's 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® Composer XE 2011 does not support development on or for IA-64 architecture (Intel® Itanium®) systems. The Intel® Compiler version 11.1 remains available for development of IA-64 architecture applications.

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

## C and C++ Source Compatibility

Intel's C++ Linux Compiler supports ANSI and ISO C and C++ standards. The compiler also supports most GNU* C and C++ language extensions. Source compatibility allows the definition of the following GNU predefined macros:

- `__GNUC__`
- `__GNUC_MINOR__`
- `__GNUC_PATCHLEVEL__`
- `__GNUG__`

The definition of these macros greatly reduces the amount of compiler-specific code, including the usage of system header files. You can disable these GNU predefined macros using the **-no-gcc** option; however, using this option is discouraged 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 preprocessing system includes header files, so that the code compiles correctly. You can also use the Intel C++ Compiler macro, __INTEL_COMPILER to conditionally compile code.

GNU inline-assembly format is supported on IA-32 and Intel® 64 architecture-based processors. Intel 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 **-gcc-version=<version>** option allows source code compatibility with particular versions of gcc:

- **–gcc-version=340** for gcc 3.4.x compatibility
- **–gcc-version=400** for gcc 4.0.x compatibility
- **–gcc-version=410** for gcc 4.1.x compatibility
- **–gcc-version=420** for gcc 4.2.x compatibility
- **–gcc-version=430** for gcc 4.3.x compatibility
- **–gcc-version=450** for gcc 4.5.x compatibility

In fact, icc automatically detects the version of gcc being used, so the –gcc-version option should rarely be used.  A mismatch between the gcc version option selected and the actual gcc version can result in compile-time or run-time errors.

If a user specifies the -gcc-name=/usr/bin/gcc43 (or gcc44 or whatever), the compiler will automatically set the gcc-version accordingly. Most users should never need to set it. By default, icc uses the version of gcc defined in your 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.

**Table 1** summarizes GNU C-language extensions supported by Intel C/C++ Compiler   9.1, 10.0, 11.0 and 12.1, highlighting continuous source compatibility improvements. **Table** 2 shows supported GNU C++ language extensions. Note that some C extensions are not supported when compiling C++ source files. The gcc C and C++ language extensions are documented in the gcc manual, available at http://gcc.gnu.org.

Compatibility of the Intel® C++ Linux compiler with the GNU  compiler is   demonstrated by the successful build and execution of Linux kernels on IA-32 and  Intel® 64 architectures using a limited number of temporary source patches.

**Table 1. gcc C-Language Extensions Supported in the Intel® C++ Compiler for Linux***

| GNU C Language Extension | Intel C++ Compiler 9.1 | Intel C++ Compiler 10.0 | Intel C++ Compiler 11.1 | Intel C++ Compiler XE 12.1 |
|---|---|---|---|---|
| Statement Expressions | Yes | Yes | Yes | Yes |
| Locally Declared Labels | Yes | Yes | Yes | Yes |
| Labels as Values | Yes | Yes | Yes | Yes |
| Nested Functions | No | No | No | No |
| Constructing Function Calls | No | No | No | No |
| Naming an Expression's Type | Yes | Yes | Yes | Yes |
| Referring to a Type with Typeof | Yes | Yes | Yes | Yes |
| Generalized Lvalues | Yes | Yes | Yes | Yes |
| Conditionals with Omitted Operands | Yes | Yes | Yes | Yes |
| Double Word Integers | Yes | Yes | Yes | Yes |
| Complex Numbers | Yes | Yes | Yes | Yes |
| Hex Floats | Yes | Yes | Yes | Yes |
| Arrays of Zero Length | Yes | Yes | Yes | Yes |

| GNU C Language Extension | Intel C++ Compiler 9.1 | Intel C++ Compiler 10.0 | Intel C++ Compiler 11.1 | Intel C++ Compiler XE 12.1 |
|---|---|---|---|---|
| Arrays of Variable Length | Yes | Yes | Yes | Yes |
| Macros with Variable Number of Arguments | Yes | Yes | Yes | Yes |
| Looser Rules for Escaped Newlines | No | No | Yes | Yes |
| Strings Literals with Embedded Newlines | Yes | Yes | Yes | Yes |
| Non-Lvalue Arrays Subscripts | Yes | Yes | Yes | Yes |
| Arithmetic on Void Pointers | Yes | Yes | Yes | Yes |
| Arithmetic on Function Pointers | Yes | Yes | Yes | Yes |
| Non-Constant Initializers | Yes | Yes | Yes | Yes |
| Compound Literals | Yes | Yes | Yes | Yes |
| Designated Initializers | Yes | Yes | Yes | Yes |
| Cast to Union Type | Yes | Yes | Yes | Yes |
| Case Ranges | Yes | Yes | Yes | Yes |
| Mixed Declarations and Code | Yes | Yes | Yes | Yes |
| Function Attributes | Yes | Yes | Yes | Yes |
| Prototype and Old-Style Function Definitions | No | No | No | No |
| C++ Style Comments | Yes | Yes | Yes | Yes |
| Dollar Sign in Identifier Names | Yes | Yes | Yes | Yes |
| The ESC Character in Constants | Yes | Yes | Yes | Yes |
| __alignof__ (types, variables) | Yes | Yes | Yes | Yes |
| Attributes of Variables | Yes | Yes | Yes | Yes |
| Inline Function as Fast as a Macro | Yes | Yes | Yes | Yes |
| Inline ASM (IA-32) | Yes | Yes | Yes | Yes |
| Controlling Names Used in ASM Code | Yes | Yes | Yes | Yes |
| Variables in Specified Registers | Yes | Yes | Yes | Yes |
| Alternate Keywords | Yes | Yes | Yes | Yes |
| Incomplete enum Types | Yes | Yes | Yes | Yes |
| Function Name as Strings | Yes | Yes | Yes | Yes |
| Getting the Return or Frame Address of a Function (IA-32) | Yes | Yes | Yes | Yes |
| Other Built-in Functions | Yes | Yes | Yes | Yes |
| Using Vector Instructions Through Built-in Functions | No | No | No | No |
| Built-in Functions Specific to Particular Target Machines | No | No | No | No |
| Variadic templates | No | No | No | Yes |

| GNU C Language Extension | Intel C++ Compiler 9.1 | Intel C++ Compiler 10.0 | Intel C++ Compiler 11.1 | Intel C++ Compiler XE 12.1 |
|---|---|---|---|---|
| Unnamed struct/union fields within structs/unions | No | No | Yes | Yes |
| Decimal floating types | No | No | Yes | Yes |
| Pragmas Accepted by gcc | No | Yes | Yes | Yes |

**Table 2. GNU C++ Language Extensions Supported in the Intel® C++ Compiler for Linux***

| GNU C++ Language Extensions | Intel C++ Compiler XE 12.1 |
|---|---|
| Minimum and Maximum Operators in C++ | Yes |
| When Is a Volatile Object Accessed? | No |
| Restricting Pointer Aliasing | Yes |
| Vague Linkage | Yes |
| Declarations and Definitions in One Header | No |
| Where's the Template? | External Template Supported |
| Extracting the Function Pointer from a Bound Pointer to Member Function | Yes |
| C++-Specific Variable, Function, and Type Attributes | Yes |
| Java* Exceptions | No |
| Deprecated Features | No |
| Backward Compatibility | No |

Starting in 11.0 the Intel(R) C++ Compiler has supported some of the C++0x features. With the latest release of Intel C++ Composer XE for Windows, Linux* and Mac OS* X, and the Intel Parallel Composer 2011, more C++0x 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 successfully. 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.

Intel® Compilers version 10.0 and later are object level interoperable with GNU* compilers version 4.2 and later. Intel® Compilers version 10.0 and 10.1 provide support of OpenMP 2.5. Intel® Compilers version 11.0 and 12.0 provide support of OpenMP 3.0. Intel® Compilers 12.1 supports OpenMP 3.1.

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 12.1 provides C and C++ binary compatibility with released versions of gcc including 3.4 and later. The default C++ library implementation is the gcc provided C++ libraries. In release 12.1 the C++ code is binary compatible with g++ and due to this a Linux* system with gcc compiler 3.4 or higher is required.

Intel C++ Compiler for Linux supports the C++ ABI (Application Binary Interface), a convention for binary object code interfaces between C++ code and the implementation-provided system and runtime libraries. The gcc 3.x and 4.x compilers also conform to the C++ ABI convention, allowing Intel C++ Compiler to be binary compatible with released versions of g++ 3.4 or higher, with slight differences to account for minor ABI changes in g++. 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.

The Intel C++ Compiler XE 12.1 uses the C++ runtime libraries/files provided by gcc. The gcc C++ runtime libraries/files includes the **libstdc++** standard C++ header files, library, and language support. By default, C++ code generation is compatible with the gcc version 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 gcc that has non-standard installation.

The following examples demonstrate binary compatibility with g++ and the usage of the g++ runtime libraries. Figure 1 shows building 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 linked against 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 library.

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 shows C++ binary compatibility with g++ by mixing binary files created by g++ and 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> icc -c main.cxx
prompt> g++ -c pHello.cxx
prompt> icc main.o pHello.o
prompt> ./a.out
Hello
```

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

Figure 4 shows how initially mixing binary files created by g++ and Intel C++ Compiler and using g++ to link them fails. Intel recommends linking with the Intel Compiler to correctly pass the Intel libraries to the system linker, **ld**.  On IA-32 and Intel® 64 architecture-based processors, 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 **librc**. To link this example correctly, **librc** 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++ 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 **librc** 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.o(.text+0xd): undefined reference to `__intel_new_proc_init'
collect2: ld returned 1 exit status

# Links with g++ and explicitly list
# Intel provided libraries in the default
# library location for release 12.0
prompt>  g++ main.o pHello.o -L/opt/intel/composerxe/lib/intel64/ -lirc
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 the Intel Compiler is recommended, as the Intel libraries will be correctly passed to the linker. The examples shown in Figure 5 through 10 uses gcc to link the application with the file **main.c**, compiled with gcc, and the file **calcSin.c**, compiled with the Intel C++ Compiler for Intel®64 architecture-based applications and optimized for the Intel® Core™ 2 Duo processor. The paper, *Optimizing Applications with the Intel® C++ and Fortran Compilers*, available at http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf, describes advanced compiler optimizations available with the Intel 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  -xSSSE3  calcSin.c -restrict
```

**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/composerxe/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.

```
prompt> gcc main.o calcSin.o
calcSin.o: In function `calcSin':
calcSin.c:(.text+0x7d): undefined reference to `__svml_sin2'
calcSin.c:(.text+0xd3): undefined reference to `__svml_sin2'
calcSin.c:(.text+0xfc): undefined reference to `__svml_sin2'
calcSin.c:(.text+0x132): undefined reference to `__svml_sin2'
collect2: ld returned 1 exit status
```

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

The **nm** utility can determine the location of unresolved symbols 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 add them to the link options.

```
prompt> nm /opt/intel/composerxe/lib/intel64/* > icc-symbols.txt
prompt> grep "__svml_sin2" icc-symbols.txt
```

0000000000000000 T __svml_sin2 **Figure 9. Utilities to Determine Libraries Required to Link with gcc Correctly**

Link with the Intel Compiler, passing the same options (**-xSSE3** in Figure 10) used during compilation, to avoid the necessity of finding out which Intel supplied libraries are required.

```
prompt> gcc -c main.c
prompt> icc -c -xSSSE3  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 Intel Compilers for Linux and compiler options for changing the default behavior. The **icc** compiler driver links the C++ runtime libraries 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 Dynamic Shared Object (DSO) versions, also known as shared libraries, of the Linux system libraries. For the Intel provided libraries, by default, the DSO version of OpenMP and libcxaguard, the g++ compatibility support library, is used, and static versions of all other

Intel libraries are used. With user-provided libraries, search for a DSO version first. If no DSO version is found, search 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 to redistributing the 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.

  Note that **–shared** is not the opposite of the **–static** option.

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

### Using Optimized Math Function Library

The math function library, **libimf**, is an optimized math library provided with the Intel Compilers for Linux. The default Linux math library, **libm**, contains functions in addition to those provided in the math function library. The Intel Compilers first link to the **libimf** library, then to the **libm** library. If an optimized function is available, it will be found in the math-function library, **libimf**. Otherwise, it is linked from the Linux math library, **libm**.

**Note:** Since Intel C++ Compiler for Linux version 9.0, the 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**, regardless of the order. Also the latest Linux releases don't provide static libm anymore.

### Build Environment Enhancements

Intel C++ Compiler continues to improve compatibility with the gcc build environments. We have enhanced the ability to use the GNU C++ library on systems with non-standard gcc configurations. The Intel Compiler installation supports traditional install with root account access using **rpm** as well as non-root account install 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

Intel C++ Compiler for Linux supports a large number of common GNU* compiler command-line options. Table 3 lists the gcc compiler options for which support has been recently added in the Intel C++ Compiler for Linux. Due to the large number of supported options, the full list of supported options is not included in this document. Information on Intel C++ command options can be found in the compiler documentation, compiler man pages, and summary information via **icc –help**.

In addition to the recently added gcc options listed in Table 3, 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 includes header files, so that the code compiles correctly. The option `-pragma-optimization-level=[Intel|GCC]` enables / disables processing `#pragma optimize` using Intel (default) or gcc syntax.

The meaning of the **`-ansi`** switch was changed starting with Intel C++ Compiler 8.1, to be compatible with the gcc command-line option of the same name. The Intel Compiler can support stricter conformance of semantics to ISO C and C++. This support is implemented by using the **`-strict–ansi`** command-line option.

Note that Intel C++ and Fortran Compilers have a large number of features to optimize applications for the latest Intel processors. The paper "*Optimizing Applications with the Intel C++ and Fortran Compilers*," available at http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf, 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.

**Table 3. Recently Added gcc Compiler Command-Line Options for Intel® C++ Compiler for Linux***

| gcc Compiler Command Linux* Option | Description |
|---|---|
| -f[no-]keep-static-consts | Enable/disable (DEFAULT) emission of static const variables even when not referenced |
| -trigraphs | Support ISO C trigraphs; also enabled in ANSI and C99 modes |
| -fno-rtti | Disable RTTI support |
| -fsyntax-only | Perform syntax and semantic checking only (no object file produced) |
| -fvisibility-inlines-hidden | Do not mark inline member functions as hidden |
| `-dD` | Same as -dM, but output #define directives in preprocessed source |
| `-dN` | Same as -dD, but #define directives contain only macro names |
| `-fargument-alias` | same as -alias-args |
| `-fargument-noalias` | same as -alias-args- |
| `-fargument-noalias-global` | arguments do not alias each other and do not alias global storage |
| `-fdata-sections` | same as -ffunction-sections |
| `-f[no-]exceptions` | enable(DEFAULT)/disable exception handling |
| `-ffunction-sections` | separate functions for the linker (COMDAT) |
| `-finline` | inline functions declared with __inline, and perform C++ inlining |
| `-fno-inline` | Do not inline functions declared with __inline, and do not perform C++ inlining |
| `-finline-functions` | inline any function, at the compiler's discretion (same as -ip) |
| `-finline-limit=<n>` | set maximum number of statements a function can have and still be considered for inlining |
| `-f[no-]math-errno` | set ERRNO after calling standard math library functions |
| `-fno-builtin` | disable inline expansion of intrinsic functions |
| `-fno-builtin-<func>` | disable the <func> intrinsic |
| `-fno-gnu-keywords` | do not recognize 'typeof' as a keyword |
| `-fno-operator-names` | disable support for operator name keywords |
| `-f[no-]omit-frame-pointer` | negative version same as –fp |

| | |
|---|---|
| `-fpack-struct` | pack structure members together |
| `-fpermissive` | allow for non-conformant code |
| `-freg-struct-return` | return struct and union values in registers when possible |
| `-ftemplate-depth-<n>` | control the depth in which recursive templates are expanded |
| `-funroll-loops` | unroll loops based on default heuristics |
| `-I-` | any directories you specify with '-I' options before the '-I-' options are searched only for the case of '#include "FILE"; they are not searched for '#include <FILE>' |
| `-imacros <file>` | treat <file> as an #include file, but throw away all preprocessing while macros defined remain defined |
| `-iprefix <prefix>` | use <prefix> with -iwithprefix as a prefix |
| `-iquote <dir>` | add directory <dir> to the front of the include file search path for files included with quotes, but not brackets |
| `-iwithprefix <dir>` | append <dir> to the prefix passed in by -iprefix and put it on the include search path at the end of the include directories |
| `-iwithprefixbefore <dir>` | similar to -iwithprefix except the include directory is placed in the same place as -I command line include directories |
| `-malign-double` | same as –align |
| `-msse` | generate code for Intel® Pentium® III and compatible Intel processors (IA-32 only) |
| `-msse2` | generate code for Intel processors that support SSE2 extensions (IA-32 only, enabled by default on Intel®64) |
| `-msse3` | generate code for Intel processors that support SSE3 extensions |
| `-mtune=<cpu>` | optimize for a specific CPU |
| `-W[no-]abi` | warn if generated code is not C++ ABI compliant (DEFAULT) |
| `-W[no-]deprecated` | print warnings related to deprecated features |
| `-Winline` | enable inline diagnostics |
| `-W[no-]comment[s]` | warn when /* appears in the middle of a /* */ comment |
| `-W[no-]main` | warn if return type of main is not expected |
| `-W[no-]missing-prototypes` | warn for missing prototypes |
| `-W[no-]pointer-arith` | warn for questionable pointer arithmetic |
| `-W[no-]return-type` | warn when a function uses the default int return type and warn when a return statement is used in a void function |
| `-W[no-]uninitialized` | warn if a variable is used before being initialized |
| `-W[no-]unknown-pragmas` | warn if an unknown #pragma directive is used (DEFAULT) |
| `-W[no-]unused-function` | warn if declared function is not used |
| `--version` | display gcc style version information |

## Intel® Fortran Compiler

Intel Fortran Compiler supports the Fortran 95 and has almost complete support for Fortran 2003. It has support for a large subset of Fortran 2008 including Coarray-Extension. It supports OpenMP* 3.1 standards. Please see Intel Fortran Composer XE release notes for details. 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 Intel C++ Compiler for Linux or the GNU gcc compiler. The *Intel Fortran Compiler* documentation has further details on calling C language functions from within Fortran code.

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 http://www.intel.com/software/products/compilers 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.

## Conclusion

Intel C++ Compilers for Linux provide outstanding source-language, binary, and command-line compatibility with the GNU C and C++ Compilers. Intel encourages users to submit feature requests for compatibility enhancements.  Many enhancements have been made based on customer feedback. This compatibility gives significant advantages to developers by increasing flexibility while enabling software to run at top speeds on IA-32 and  Intel® 64 architecture-based systems.

## Appendix A: Linux Distribution Support Information

The following table lists Linux* OS support information for recent releases of the Intel C++ Compiler for Linux. These are for the IA-32 and Intel® 64 architecture-based processors. Consult the product release notes for the most recent product information.

**Note:** Intel C++ Compiler XE 12.1 for Linux requires a Linux distribution with a released gcc 3.4 or higher.

| Intel® C++ Compiler for Linux* version | Processor Architecture | Supported glibc versions | Supported kernel versions | Supported g++ versions |
|---|---|---|---|---|
| XE 12.1 | IA-32 | , 2.3.4, 2.3.5, 2.8 | 2.4.x, 2.6.x | 3.4.x, 3.3.x, 3.4.x, 4.0.x, 4.2.x, 4.3.x ,4.4.x, 4.5.x |
| XE 12.1 | Intel® 64 | 2.3.4, 2.3.5, 2.8 | 2.4.x, 2.6.x | 3.4.x, 3.3.x, 3.4.x, 4.0.x, 4.2.x, 4.3.x ,4.4.x, 4.5.x |
| 11.1 | IA-32 | , 2.3.4, 2.3.5, 2.8 | 2.4.x, 2.6.x | 3.4.x, 4.0.x, 4.2.x, 4.3.x |
| 11.1 | Intel® 64 | , 2.3.4, 2.3.5, 2.8 | 2.4.x, 2.6.x | 3.4.x, 4.0.x, 4.2.x, 4.3.x |
| | | | | |
| 10.0 | IA-32 | , 2.3.4, 2.3.5 | 2.4.x, 2.6.x | 3.4.x, 4.0.x, 4.1.x, 4.2.x |
| 10.0 | Intel® 64 | , 2.3.4, 2.3.5 | 2.4.x, 2.6.x | , 3.4.x, 4.0.x, 4.1.x, 4.2.x |
| | | | | |
| 9.1 | IA-32 | 2.2.5, 2.3.2, 2.3.3, 2.3.4 | 2.4.x, 2.6.x | , 3.4.x, 4.0.x, 4.1.x |
| 9.1 | Intel® 64 | 2.3.2, 2.3.3, 2.3.4 | 2.4.x, 2.6.x | , 3.4.x, 4.0.x, 4.1.x |

## References

- General information on Intel® software development tools, including the Intel C++ and Fortran Compilers: http://www.intel.com/software/products
- Intel® C++ Compiler documentation is available at http://www.intel.com/software/products/compilers/> 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://www.codesourcery.com/cxx-abi/

For product and purchase information visit:
www.intel.com/software/products