Snapdragon ARM LLVM Compiler for Android

User Guide

80-VB419-90 Rev. G

August 1, 2014

Submit technical questions at:
http://developer.qualcomm.com/llvm-forum

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

1 Introduction

1.1 Overview

This document describes C and C++ compilers for the ARM® processor architecture. The compilers are based on the LLVM compiler framework, and are collectively referred to as the LLVM compilers.

NOTE The LLVM compilers are commonly referred to as *Clang*.

1.2 Features

The LLVM compilers offer the following features:

- **ISO C conformance**
  Supports the International Standards Organization (ISO) C language standard
- **Compatibility**
  Supports ARM extensions and most GCC extensions to simplify porting
- **System library**
  Supports standard libraries as provided in the Android NDK
- **Processor-specific libraries**
  Provides library routines that are optimized for the Qualcomm ARM architecture
- **Intrinsics**
  Provides a mechanism for emitting ARM assembly instructions in C source code

1.3 Languages

The LLVM compilers support C, C++, and many dialects of those languages:

- C language: K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1), ISO C99 (+TC1, TC2, TC3)
- C++ language: C++98, C++11

In addition to these base languages and their dialects, the LLVM compilers support a broad variety of language extensions. These extensions are provided for compatibility with the GCC, Microsoft, and other popular compilers, as well as to improve functionality through the addition of extensions unique to the LLVM compilers.

All language extensions are explicitly recognized as such by the LLVM compilers, and marked with extension diagnostics which can be mapped to warnings, errors, or simply ignored.

1.4 GCC compatibility

The LLVM compiler driver and language features are intentionally designed to be as compatible with the GNU GCC compiler as reasonably possible, easing migration from GCC to LLVM. In most cases, code "just works".

1.5 Processor versions

The LLVM compilers can generate code for all versions of the ARM processor architecture that are supported by the standard LLVM compiler.

However, full support is provided only for ARMv7 (which includes Krait) and ARMv8 (which is ARM's newest architecture).

ARMv8 supports two instruction set architectures (ISA):

- **AArch64** – the new 64-bit ISA, which supports a larger virtual and physical address space. All general purpose registers (and many of the system registers) are 64 bits. All instructions are encoded in 32 bits.
- **AArch32** – a 32-bit ISA which provides full compatibility with the ARMv7 ISA in both ARM and Thumb modes. It also includes many aspects of AArch64 (including support for cryptography and enhanced floating point).

For more information see the *ARMv8-A Reference Manual*.

1.6 LLVM versions

The LLVM compilers are based on LLVM 3.5, as defined at http://llvm.org/.

1.7 Using the document

This document is designed as a reference for experienced C/C++ programmers. It describes the LLVM compilers and language implementations.

The document contains the following chapters:

- Chapter 1, *Introduction*, presents an overview of the compilers and the document.
- Chapter 2, *Getting Started*, explains how to compile and execute a simple C program.
- Chapter 3, *Using the Compilers*, describes the command line syntax, console messages, and input and output files.
- Chapter 4, *Porting Code from GCC*, describes issues commonly encountered while porting GCC code to ARM LLVM.
- Chapter 5, *Coding Practices*, describes recommended coding practices for ensuring the generation of efficient object code.
- Chapter 6, *Language Compatibility*, describes how the compilers implement the C language standard.
- Appendix A presents the LLVM license statements governing this document.

C language reference

This document does not describe the C or C++ languages. The suggested references are:

- *The C Programming Language (2nd Edition),* Brian Kernighan and Dennis Ritchie, Prentice Hall, 1988.
- *The C++ Programming Language (3rd Edition)*, Bjarne Stroustrup, Addison-Wesley, 1997.

Compiler references

This document does not provide detailed descriptions of the code optimizations performed by LLVM. Suggested compiler references are:

- *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall, 2006
- *Engineering a Compiler (2nd Edition)*, Keith Cooper and Linda Torczon, Morgan Kaufmann, 2011

1.8 Notation

This document uses italics for terms and document names:

The C Programming Language (2nd Edition)

Courier font is used for computer text:

```
int main()
{
   printf("Hello world\n");
   return(0);
}
```

The following notation is used to define the syntax of functions and commands:

- Square brackets enclose optional items (e.g., **`help`** [*`command`*]).
- **Bold** is used to indicate literal symbols (e.g., the brackets in *`array`***`[`***`index`***`]`**).
- The vertical bar character | is used to indicate a choice of items.
- Parentheses are used to enclose a choice of items (e.g., (**`on`**|**`off`**)).
- An ellipsis, . . . , follows items that can appear more than once.
- *Italics* are used for terms that represent categories of symbols.

Examples:

```
#define name(parameter1[, parameter2...]) definition
logging (on|off)
```

In the above examples `#define` is a preprocessor directive and `logging` is an interactive compiler command.

name represents the name of a defined symbol.

parameter1 and *parameter2* are macro parameters. The second parameter is optional since it is enclosed in square brackets. The ellipsis indicates that the macro accepts more than parameters.

`on` and `off` are bold to show that they are literal symbols. The vertical bar between them shows that they are alternative parameters of the `logging` command.

1.9 Feedback

If you have any comments or suggestions regarding the LLVM compilers (or this document), please send them to:

http://developer.qualcomm.com/llvm-forum

NOTE If you are a commercial licensee of Qualcomm, use your normal support channels for support.

2 Getting Started

2.1 Overview

This chapter explains how to build and execute a simple C program using the LLVM compiler.

The program is built in the Linux environment, and executed directly on ARMv7 hardware running Linux.

NOTE The Android NDK is assumed to be already installed on your computer. This includes the tools required for assembling and linking a compiled program.

2.2 Create source file

Create the following C source file:

```
#include <stdio.h>

int main()
{
     printf("Hello world\n");
     return(0);
}
```

Save the file as `hello.c`.

2.3 Compile program

Compile the program with the following command:

```
clang hello.c -o hello
```

This translates the C source file `hello.c` into the executable file `hello`.

2.4 Execute program

To execute the program, use the following command:

```
hello
```

The program outputs its message in the terminal:

```
Hello world
```

You have now compiled and executed a C program using the LLVM compiler. For more information on using the compiler see the following chapter.

3 Using the Compilers

3.1 Overview

The LLVM compilers translate C and C++ programs into ARM processor code.

C and C++ programs are stored in source files, which are text files created with a text editor. ARM processor code is stored in object files, which are executable binary files.

This chapter covers the following topics:

- Starting the compilers
- Input and output files
- Compiler options
- Warning and error messages
- Using code optimizations
- Using GCC cross compile environments
- Built-in functions
- Address Sanitizer
- LLVM Symbolizer
- Secure programming support

3.2 Starting the compilers

To start the C compiler from a command line, type:

```
clang [options...] input_files...
```

To start the C++ compiler from a command line, type:

```
clang++ [options...] input_files...
```

The compilers accept one or more input files on the command line. Input files can be C/C++ source files or object files. For example:

```
clang hello.c mylib.c
```

Command switches are used to control various compiler options (Section 3.4). A switch consists of a dash character (‘-’) followed by a switch name and optional parameter.

Switches are case-sensitive and must be separated by at least one space. For example:

```
clang hello.c -o hello
```

To list the available command options, use the `--help` option:

```
clang --help
clang++ --help
```

This option causes the compiler to display the command line syntax, followed by a list of the available command options.

NOTE `clang` is the name of the front end driver for the LLVM compiler framework.

3.3 Input and output files

The LLVM compilers preprocess and compile one or more source files into object files. The compilers then invoke the linker to combine the object files into an executable file.

Table 3-1 lists the input file types and the tool that processes files of each type. The compilers use the file name extension to determine how to process the file.

Table 3-1 Compiler input files

Extension	Description	Tool
.c	C source file (must be preprocessed)	C compiler
.i	C source file (should not be preprocessed)	
.h	C header file (translated into precompiled header)	
.cc .cp .cxx .cpp .CPP .c++ .C	C++ source file (must be preprocessed)	C++ compiler
.ii	C++ source file (should not be preprocessed)	
.h .hh .H	C++ header file (translated into precompiled header)	
.bc .ll	LLVM intermediate representation (IR) file	C/C++ compiler
.s	Assembly source file (should not be preprocessed)	Assembler
.S	Assembly source file (must be preprocessed)	
other	Binary object file	Linker

NOTE All file name extensions are case-sensitive literal strings. Input files with unrecognized extensions are treated as object files.

For more information on LLVM IR files see http://llvm.org.

Table 3-2 lists the output file types and the tools used to generate each file type.

Compiler options (Section 3.4) are used to specify the output file type.

Table 3-2 Compiler output files

File Type	Default File Name	Input Files
Executable file	`a.out`	The specified source files are compiled and linked to a single executable file.
Object file	*`file`*`.o`	Each specified source file is compiled to a separate object file (where *file* is the source file name).
Assembly source file	*`file`*`.s`	Each specified source file is compiled to a separate assembly source file (where *file* is the source file name).
Preprocessed C/C++ source file	`stdout`	The preprocessor output is written to the standard output.

NOTE If an output file name is not explicitly specified (with the `-o` option), the compilers use the default file names listed in Table 3-2.

3.4 Compiler options

The LLVM compilers are controlled by command-line options (Section 3.2). Many of the GCC options are supported, along with options that are LLVM-specific.

NOTE Many of the `-f`, `-m`, and `-W` options can be written in two ways: `-f`*`setting`* to enable a binary setting, or `-fno-`*`setting`* to disable the setting.

Display

See Section 3.4.1

-help
-v

Compilation

See Section 3.4.2

-###
-c -E -S -pipe
-o *file*
-Wp,`arg[,arg...]`
-Wa,`arg[,arg...]`
-Wl,`arg[,arg...]`
-x *language*

C dialect

See Section 3.4.3

-ansi -fno-asm -fgnu-runtime -fgnu89-inline
-fsigned-bitfields -fsigned-char -funsigned-char
-no-integrated-cpp -traditional -Wpointer-sign

C++ dialect

See Section 3.4.4

-cxx-isystem ***dir***
-ffor-scope -fno-for-scope -fno-gnu-keywords
-ftemplate-depth-*n* -fvisibility-inlines-hidden
-fuse-cxa-atexit -nobuiltininc -nostdinc++
-Wc++0x-compat -Wno-deprecated
-Wnon-virtual-dtor -Woverloaded-virtual
-Wreorder

Warning and error messages

See Section 3.4.5

```
-ferror-limit=n  -ftemplate-backtrace-limit=n
-ferror-warn filename  -fsyntax-only  -pedantic
-pedantic-errors  -Q-unused-arguments
-w -Wfoo -Wno-foo  -Wall  -Warray-bounds
-Wcast-align  -Wchar-subscripts
-Wcomment  -Wconversion
-Wdeclaration-after-statement  -Wno-deprecated-declarations
-Wempty-body  -Wendif-labels  -Werror
-Werror=foo  -Wno-error=foo
-Werror-implicit-function-declaration
-Weverything  -Wextra  -Wfloat-equal
-Wformat  -Wformat=2  -Wno-format-extra-args
-Wformat-nonliteral  -Wformat-security
-Wignored-qualifiers
-Wimplicit  -Wimplicit-function-declaration  -Wimplicit-int
-Wno-invalid-offsetof  -Wlong-long  -Wmain
-Wmissing-braces -Wmissing-declarations
-Wmissing-noreturn  -Wmissing-prototypes  -Wno-multichar
-Wnonnull -Wpacked  -Wpadded  -Wparentheses  -Wpointer-arith
-Wreturn-type  -Wshadow  -Wsign-compare
-Wswitch -Wswitch-enum  -Wsystem-headers
-Wtrigraphs  -Wundef  -Wuninitialized  -Wunknown-pragmas
-Wunreachable-code  -Wunused  -Wunused-function  -Wunused-label
-Wunused-parameter  -Wunused-value  -Wunused-variable
-Wno-vectorizer-no-neon  -Wwrite-strings
```

Debugging

See Section 3.4.6

```
-dumpmachine -dumpversion
-feliminate-unused-debug-symbols
-ftime-report  -g[level] -print-file-name=library
-print-libgcc-file-name  -print-multi-directory
-print-multi-lib  -print-multi-os-directory
-print-prog-name=program  -print-seh-dirs
-save-temps  -time
```

Diagnostic format

See Section 3.4.7

```
-fcaret-diagnostics  -fno-caret-diagnostics
-fdiagnostics-format=(clang|msvc|vi)
-fdiagnostics-show-option  -fno-diagnostics-show-option
-fdiagnostics-show-category=(none|id|name)
-fdiagnostics-print-source-range-info
-fno-diagnostics-print-source-range-info
-fdiagnostics-parseable-fixits
-fdiagnostics-show-note-include-stack
-fdiagnostics-show-template-tree
-fmessage-length=n
```

Individual warning groups

See Section 3.4.8

```
-Wextra-tokens  -Wambiguous-member-template
-Wbind-to-temporary-copy
```

Compiler crash diagnostics

See Section 3.4.9

```
-fno-crash-diagnostics
```

Compiler toolchain

See Section 3.4.10

```
-ccc-gcc-name
-fuse-ld=(gold|bfd)
```

Preprocessor

See Section 3.4.11

```
-A pred=ans  -A -pred=ans  -ansi  -C  -CC  -d(DMN)
-D name  -D name=definition  -fexec-charset=charset
-finput-charset=charset  -fpch-deps  -fpreprocessed
-fstrict-overflow  -ftabstop=width  -fwide-exec-charset=charset
-fworking-directory  --help  -H  -I dir  -I-  -include file
-isystem prefix  -isystem-prefix prefix
-ino-system-prefix prefix
-M  -MD  -MF file  -MG  -MM  -MMD  -MP  -MQ target  -MT target
-nostdinc  -nostdinc++  -o file  -P  -remap  --target-help
-U name  -v  -version  --version  -w  -Wall  -Wcomment
-Wcomments  -Wendif-labels  -Werror  -Wimport
-Wsystem-headers  -Wtrigraphs  -Wundef  -Wunused-macros
-Xpreprocessor option
```

Assembling

See Section 3.4.12

```
-Xassembler option
-integrated-as  -no-integrated-as
```

Linking

See Section 3.4.13

```
object_file_name  -c  -dynamic  -E
-l library  -moslib=library
-nodefaultlibs  -nostartfiles  -nostdlib
-pie  -s  -S  -shared  -shared-libgcc
-static  -static-libgcc
-symbolic  -u symbol  -Xlinker option
```

Directory search

See Section 3.4.14

```
-Bprefix
-F dir  -I dir
--gcc-toolchain=prefix
-I-
-Ldir
--sysroot=prefix
```

Processor version

See Section 3.4.15

```
-target triple
-march=version
-mcpu=version
-mfpu=version
-mfloat-abi=(soft|softfp|hard)
```

Code generation

See Section 3.4.16

```
-fasynchronous-unwind-tables
-fchar-array-precise-tbaa  -fno-char-array-precise-tbaa
-femit-all-data  -femit-all-decls
-ffp-contract=(fast|on|off)
-fno-exceptions
-fmerge-functions
-fpic  -fPIC  -fpie  -fPIE
-fshort-enums  -fno-short-enums
-fshort-wchar  -fshort-wchar
-ftrap-function=value  -ftrapv  -ftrapv-handler
-funwind-tables  -fverbose-asm
```

```
-fvisibility=[default|internal|hidden|protected]
-fwrapv
-mhwdiv=(arm|thumb|arm,thumb|none)
-mllvm -aarch64-disable-abs-reloc
-mllvm -aggressive-jt
-mllvm -arm-expand-memcpy-runtime
-mllvm -arm-memset-size-threshold
-mllvm -arm-memset-size-threshold-zeroval
-mllvm -arm-opt-memcpy
-mllvm -disable-thumb-scale-addressing
-mllvm -emit-cp-at-end
-mllvm -enable-arm-addressing-opt
-mllvm -enable-arm-peephole
-mllvm -enable-arm-zext-opt
-mllvm -enable-print-fp-zero-alias
-mllvm -enable-round-robin-RA
-mllvm -enable-select-to-intrinsics
-mllvm -favor-r0-7
-mllvm -prefetch-locality-policy=(L1|L2|L3|stream)
-mrestrict-it  -mno-restrict-it
-mtune=krait2
```

Vectorization

See Section 3.4.17

```
-fvectorize-loops  -fvectorize-loops-debug
-mllvm -print-vectorized-loops
-fprefetch-loop-arrays[=stride]  -fno-prefetch-loop-arrays
```

Parallelization

See Section 3.4.18

```
-fparallel
```

Optimization

See Section 3.4.19

```
-O  -O0  -O1  -O2  -O3  -Os
-Ofast  -Osize
```

Specific optimizations

See Section 3.4.20

-falign-functions[=*n*] **-falign-jumps**[=*n*]
-falign-labels[=*n*] **-falign-loops**[=*n*]
-falign-inner-loops **-fno-align-inner-loops**
-falign-os **-fno-align-os**
-fdata-sections **-finline** **-finline-functions**
-fnomerge-all-constants **-fomit-frame-pointer**
-foptimize-sibling-calls **-fstack-protector**
-fstack-protector-all **-fstrict-aliasing**
-funit-at-a-time **-funroll-all-loops**
-funroll-loops **-fno-zero-initialized-in-bss**

Math optimization

See Section 3.4.21

-fassociative-math **-ffast-math** **-ffinite-math-only**
-fmath-errno **-fno-math-errno** **-freciprocal-math**
-fno-signed-zeros **-fno-trapping-math**
-funsafe-math-optimizations

Link-time optimization

See Section 3.4.22

-c-lto **-flto** **-flto-scope=**(**program**|**library**)
-lto-no-inter-mod-inline **-lto-preserve** *symbol*
-lto-preserve-list *filename* **-lto-w-no-unknown-sym**
-S-lto **-Wlto** *arg,...*

Secure programming

See Section 3.4.23

--analyze **-analyzer-checker=***checker* **-analyzer-checker-help**
-analyzer-disable-checker=*checker* **--analyzer-output html**
--compile-and-analyze *dir*

3.4.1 Display

-help
Display compiler command and option summary.

-v
Display compiler release version.

3.4.2 Compilation

-###
Print commands used to perform the compilation.

-c
Compile source file, but do not link it.

-E
Preprocess source file only, do not compile it.

-S
Compile source file, but do not assemble it.

-pipe
Communicate between compiler stages using pipes not temporary files.

-o *file*
Specify the name of the compiler output file.

-Wp,*arg*[**,***arg*...]
Pass the specified arguments to the preprocessor.

-Wa,*arg*[**,***arg*...]
Pass the specified arguments to the assembler.

-Wl,*arg*[**,***arg*...]
Pass the specified arguments to the linker.

-x *language*
Specify language of the subsequent source files specified on the command line.

3.4.3 C dialect

-ansi
For C, support ISO C90. For C++, remove conflicting GNU extensions.

-fno-asm
Do not recognize `asm`, `inline`, or `typeof` as keywords.

-fgnu-runtime
Generate output compatible with the standard GNU Objective-C runtime.

-fgnu89-inline
Use the gnu89 inline semantics.

-fsigned-bitfields
Define bitfields as signed.

-fsigned-char
Define `char` type as signed.

-funsigned-char
Define `char` type as unsigned.

-no-integrated-cpp
Compile using separate preprocessing and compilation stages.

-traditional
Support pre-standard C language.

-Wpointer-sign
Flag pointers when assigned or passed values with a differing sign.

3.4.4 C++ dialect

-cxx-isystem *dir*
Add specified directory to C++ SYSTEM include search path.

-ffor-scope
-fno-for-scope
Control whether the scope of a variable declared in a `for` statement is limited to the statement or to the scope enclosing the statement.

-fno-gnu-keywords
Disable recognizing `typeof` as a keyword.

-ftemplate-depth-*n*
Specify the maximum instantiation depth of a template class.

-fvisibility-inlines-hidden
Specify default visibility for inline C++ member functions.

-fuse-cxa-atexit
Register destructors with function `__cxa_atexit` (instead of `atexit`). This applies only to objects that have static storage duration.

-nobuiltininc
Disable builtin #include directories.

-nostdinc++
Disable standard #include directories for the C++ standard library.

-Wc++0x-compat
Generate warnings for C++ constructs with different semantics in ISO C++ 1998 and ISO C++ 200x.

-Wno-deprecated
Do not generate warnings when deprecated features are used.

-Wnon-virtual-dtor
Generate warning when a polymorphic class is declared with a non-virtual destructor.

-Woverloaded-virtual
Generate warning when a function hides virtual functions from a base class.

-Wreorder
Generate warning when member initializers do not appear in the code in the required execution order.

3.4.5 Warning and error messages

-ferror-limit=*n*
Stop emitting diagnostics after *n* errors have been produced. The default setting is 20. The error limit can be disabled with the option `-ferror-limit=0`.

-ftemplate-backtrace-limit=*n*
Only emit up to *n* template instantiation notes within the template instantiation backtrace for a single warning or error. The default setting is 10. The limit can be disabled with the option `-ftemplate-backtrace-limit=0`.

-ferror-warn *filename*
Convert the specified set of compiler warnings into errors.

The specified text file contains a list of warning names, with each warning name separated by whitespace in the file.

Warning names are based on the switch names of the corresponding compiler warning-message options. For example, to convert the warnings generated by the option `-Wunused-variable`, use the warning name `unused-variable`.

This option can be specified multiple times.

NOTE This option (and its associated file) can be integrated into a build system, and used to iteratively resolve the warning messages generated by a project.

-fsyntax-only
Check for syntax errors only.

-pedantic
Generate all warnings required by the ISO C and ISO C++ standards.

-pedantic-errors
Equivalent to `-pedantic`, but generate errors instead of warnings.

-Qunused-arguments
Do not generate warnings for unused driver arguments.

-w
Suppress all warnings.

-W*foo*
Enable the diagnostic *foo*.

-Wno-*foo*
Disable the diagnostic *foo*.

-Wall
Enable all `-W` options.

-Warray-bounds
Generate warning if array subscripts are out of bounds.

-Wcast-align
Generate warning if a pointer cast increases the required alignment of the target.

-Wchar-subscripts
Generate warning if array subscript is type `char`.

-Wcomment
Generate warning if a comment symbol appears inside a comment.

-Wconversion
Generate warning if an implicit conversion may alter a value.

-Wdeclaration-after-statement
Generate warning when a declaration appears in a block after a statement.

-Wno-deprecated-declarations
Do not generate warnings for functions, variables, or types assigned the attribute `deprecated`.

-Wempty-body
Generate warning if an `if`, `else`, or `do while` statement contains an empty body.

-Wendif-labels
Generate warning if an `#else` or `#endif` directive is followed by text.

-Werror
Convert all warnings into errors.

-Werror=*foo*
Convert the diagnostic *foo* into an error.

-Wno-error=*foo*
Keep the diagnostic *foo* as a warning, even if `-Werror` is used.

-Werror-implicit-function-declaration
Generate warning or error if a function is used before being declared.

-Weverything
Enable all warnings.

-Wextra
Enable selected warning options, and generate warnings for selected events.

-Wfloat-equal
Generate warning if two floating point values are compared for equality.

-Wformat
In calls to `printf`, `scanf`, and other functions with format strings, ensure that the arguments are compatible with the specified format string.

-Wformat=2
This option is equivalent to specifying the following options: "`-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`".

-Wno-format-extra-args
Do not generate warning for passing extra arguments to `printf` or `scanf`.

-Wformat-nonliteral
Generate warning if the format string is not a string literal, except if the format arguments are passed through `va_list`.

-Wformat-security
Generate warning for format function calls that may cause security risks.

-Wignored-qualifiers
Generate warning if a return type has a qualifier (for example, `const`).

-Wimplicit
Equivalent to `-Wimplicit-int` and `-Wimplicit-function-declaration`.

-Wimplicit-function-declaration
Generate warning if a function is used before it is declared.

-Wimplicit-int
Generate warning if a declaration does not specify a type.

-Wno-invalid-offsetof
Do not generate warning if macro `offsetof` is passed a non-POD type.

-Wlong-long
Generate warning if type`long long` is used.

-Wmain
Generate warning if the function `main()` has any suspicious properties.

-Wmissing-braces
Generate warning if an aggregate or union initializer is not properly bracketed.

-Wmissing-declarations
Generate warning if a global function is defined without being first declared.

-Wmissing-noreturn
Generate warning if a function does not include a `return` statement.

-Wmissing-prototypes
Generate warning if a global function is defined without a prototype.

-Wno-multichar
Do not generate warning if a multicharacter constant is used.

-Wnonnull
Generate warning if a null pointer is passed to an argument that is specified to require a non-null value (with the `nonnull` attribute).

-Wpacked
Generate warning if the memory layout of a structure is not affected after the structure is specified with the `packed` attribute.

-Wpadded
Generate warning if the memory layout of a structure includes padding.

-Wparentheses
Generate warning if the parentheses are omitted in certain cases.

-Wpointer-arith
Generate warning if any code depends on the size of `void` or a function type.

-Wreturn-type
Generate warning if a function returns a type that defaults to `int`, or a value incompatible with the defined return type.

-Wshadow
Generate warning if a local variable shadows another local variable, global variable, or parameter; or if a built-in function gets shadowed.

-Wsign-compare
Generate warning in a signed/unsigned compare if the result may be inaccurate due to the signed operand being converted to unsigned.

-Wswitch
-Wswitch-enum
Generate warning if a `switch` statement uses an enumeration type for the index, and does not specify a `case` for every possible enumeration value, or specifies a case with a value outside the enum range.

-Wsystem-headers
Generate warning for constructs declared in system header files.

-Wtrigraphs
Generate warning if a trigraph forms an escaped newline in a comment.

-Wundef
Generate warning if an undefined non-macro identifier appears in an `#if` directive.

-Wuninitialized
Generate warning if referencing an uninitialized automatic variable.

-Wunknown-pragmas
Generate warning if a `#pragma` directive is not recognized by the compiler.

-Wunreachable-code
Generate warning if code will never be executed.

-Wunused
Specifies all of the `-Wunused` options.

-Wunused-function
Generate warning if a static function is declared without being defined; or if a non-inline static function is not used.

-Wunused-label
Generate warning if a label is declared without being used.

-Wunused-parameter
Generate warning if a function argument is not used in its function.

-Wunused-value
Generate warning if the value of a statement is not subsequently used.

-Wunused-variable
Generate warning if a local or non-constant static variable is not used in its function.

-Wno-vectorizer-no-neon
Do not generate the warning “Vectorization flags ignored because armv7/armv8 and neon not set”.

Vectorization requires the target to be ARMv7 or ARMv8, and the Neon feature to be enabled. If the vectorization options are used without these required options, a warning is normally generated and the vectorization options are ignored.

-Wwrite-strings
For C, assign string constants the type `const char[`*length*`]` to ensure that a warning is generated if the string address gets copied to a non-`const char *` pointer. For C++, generate warning if converting a string constant to `char *`.

3.4.6 Debugging

-dumpmachine
Display the target machine name.

-dumpversion
Display the compiler version.

-feliminate-unused-debug-symbols
Generate debug information only for the symbols that are used. (Debug information is generated in STABS format.)

-time
-ftime-report
Display the elapsed time for each stage of the compilation.

-g[*level*]
Generate source-level debug information.

-print-file-name=*library*
Display the full library path of the specified file.

-print-libgcc-file-name
Display the library path for file `libgcc.a`.

-print-multi-directory
Display the directory names of the multi libraries specified by other compiler options in the current compilation.

-print-multi-lib
Display the directory names of the multi libraries paired with the compiler options that specified the libraries in the current compilation.

-print-multi-os-directory
Display the relative path that gets appended to the multilib search paths.

-print-prog-name=*program*
Display the absolute path of the specified program.

-print-search-dirs
Display the search paths used to locate libraries and programs during compilation.

-save-temps
Save the normally-temporary intermediate files generated during compilation.

3.4.7 Diagnostic format

The LLVM compilers aim to produce beautiful diagnostics by default, especially for new users just beginning to use LLVM. However, different users have different preferences, and sometimes LLVM may be driven by another program which needs the diagnostic output to be simple and consistent rather than user-friendly. For these cases, LLVM provides a wide range of options to control the output format of the diagnostics that it generates.

`-fcaret-diagnostics`
`-fno-caret-diagnostics`

Print source line and ranges from source code in diagnostic.

Control whether LLVM prints the source line, source ranges, and caret when emitting a diagnostic. The default setting is enabled. When enabled, LLVM will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
```

`-fdiagnostics-format=(clang|msvc|vi)`

Change diagnostic output format to better match IDEs and command line tools.

This option controls the output format of the filename, line number, and column printed in diagnostic messages. The default setting is `clang`. The effect of the setting on the output format is shown below.

`clang`

```
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int'
```

`msvc`

```
t.c(3,11) : warning: conversion specifies type 'char *' but the
argument has type 'int'
```

`vi`

```
t.c +3:11: warning: conversion specifies type 'char *' but the
argument has type 'int'
```

-fdiagnostics-show-option
-fno-diagnostics-show-option

Enable `[-Woption]` information in diagnostic line.

Control whether LLVM prints the associated warning group option name (Section 3.5.3) when outputting a warning diagnostic. The default setting is disabled. For example, given the following diagnostic output:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
```

In this case, specifying `-fno-diagnostics-show-option` prevents LLVM from printing the [`-Wextra-tokens`] information in the diagnostic output. This information indicates the option needed to enable or disable the diagnostic, either from the command line or by using the pragma `GCC diagnostic` (Section 3.5.5).

-fdiagnostics-show-category=(none|id|name)

Enable printing category information in diagnostic line.

This option controls whether LLVM prints the category associated with a diagnostic when emitting it. The default setting is `none`. The effect of the setting on the output format is shown below.

```
   none
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat]

   id
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat,1]

   name
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat,Format String]
```

Each diagnostic may or may not have an associated category; if it has one, it is listed in the diagnostic category field of the diagnostic line (in the `[]`'s).

This option can be used to group diagnostics by category, so it should be a high-level category: the goal is get dozens of categories, not hundreds or thousands of them.

-fdiagnostics-print-source-range-info
-fno-diagnostics-print-source-range-info

Print machine-parseable information about source ranges.

This option controls whether LLVM prints information about source ranges in a machine-parseable format after the file/line/column number information. The default setting is disabled. The information is a simple sequence of brace-enclosed ranges, where each range lists the start and end line/column locations. For example, given the following output:

```
exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands
to binary expression ('int *' and '_Complex float')
P = (P-42) + Gamma*4;
     ~~~~~~ ^ ~~~~~~~
```

In this case the {}'s are generated by `-fdiagnostics-print-source-range-info`.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fdiagnostics-parseable-fixits

Print Fix-Its in a machine-parseable format.

This option makes LLVM print available Fix-Its in a machine-parseable format at the end of diagnostics. The following example illustrates the format:

```
fix-it:"t.cpp":{7:25-7:29}:"Gamma"
```

In this case the range printed is half-open, so the characters from column 25 up to (but not including) column 29 on line 7 of file `t.cpp` should be replaced with the string `Gamma`. Either the range or replacement string can be empty (representing strict insertions and strict erasures, respectively). Both the file name and insertion string escape backslash (as "`\\`"), tabs (as "`\t`"), newlines (as "`\n`"), double quotes (as "`\"`"), and non-printable characters (as octal "`\xxx`").

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fdiagnostics-show-template-tree

For large templated types, this option causes LLVM to display the templates as an indented text tree, with one argument per line, and any differences marked inline.

```
   default

t.cc:4:5: note: candidate function not viable: no known conversion
from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...],
map<double, [...]>>>' for 1st argument;

   -fdiagnostics-show-template-tree

t.cc:4:5: note: candidate function not viable: no known conversion
for 1st argument;
  vector<
    map<
      [...],
      map<
        [float != float],
        [...]>>>
```

-fmessage-length=*n*

Format error messages to fit on lines with the specified number of characters.

3.4.8 Individual warning groups

-Wextra-tokens

Warn about excess tokens at the end of a preprocessor directive.

This option enables warnings about extra tokens at the end of preprocessor directives. The default setting is enabled. For example:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
```

These extra tokens are not strictly conforming, and are usually best handled by commenting them out.

-Wambiguous-member-template

Warn about unqualified uses of a member template whose name resolves to another template at the location of the use.

This option (which is enabled by default) generates a warning in the following code:

```
template<typename T> struct set{};
template<typename T> struct trait { typedef const T& type; };
struct Value {
  template<typename T> void set(typename trait<T>::type value){}
};

void foo() {
  Value v;
  v.set<double>(3.2);
}
```

C++ requires this to be an error, but because it is difficult to work around, LLVM downgrades it to a warning as an extension.

-Wbind-to-temporary-copy

Warn about an unusable copy constructor when binding a reference to a temporary.

This option enables warnings about binding a reference to a temporary when the temporary does not have a usable copy constructor. The default setting is enabled. For example:

```
struct NonCopyable {
  NonCopyable();
private:
  NonCopyable(const NonCopyable&);
};
void foo(const NonCopyable&);
void bar() {
  foo(NonCopyable());   // Disallowed in C++98; allowed in C++11.
}


struct NonCopyable2 {
  NonCopyable2();
  NonCopyable2(const NonCopyable2&);
};
void foo(const NonCopyable2&);
void bar() {
  foo(NonCopyable2());   // Disallowed in C++98; allowed in C++11.
}
```

NOTE If `NonCopyable2::NonCopyable2()` has a default argument whose instantiation produces a compile error, that error will still be a hard error in C++98 mode, even if this warning is disabled.

3.4.9 Compiler crash diagnostics

The LLVM compilers may crash once in a while. Generally, this only occurs when using the latest versions of LLVM.

LLVM goes to great lengths to assist you in filing a bug report. Specifically, after a crash it generates preprocessed source file(s) and associated run script(s). These files should be attached to a bug report to ease reproducibility of the failure. The following compiler option is used to control the crash diagnostics.

`-fno-crash-diagnostics`
Disable auto-generation of preprocessed source files during a LLVM crash.

This option can be helpful for speeding up the process of generating a delta reduced test case.

3.4.10 Compiler toolchain

`-ccc-gcc-name` *`executable`*
Specify the default program used to assemble and link the object file.

`executable` specifies the name of the executable program.

The LLVM compilers include an integrated assembler but no linker. Thus to link an object file the compiler uses the linker specified with this option.

The assembler specified with this option is used only when the integrated assembler is not being used (Section 3.4.12).

`-fuse-ld=``(`**`gold`**`|`**`bfd`**`)`
Specify an alternative linker to use in place of the default system linker.

GNU GCC sysroots (version 4.7 and later) typically include two linkers: `gold` and `bfd`. The `gold` linker provides a plugin interface which is used to support link-time optimization (Section 3.6.6), while the `bfd` linker does not.

Thus, in order to use link-time optimization, the `gold` linker must be specified as the system linker (otherwise, the optimization will fail). This option specifies the system linker that is used by the compiler.

NOTE This option is available for ARM targets only (on both Linux and Windows platforms). Although it is target-independent and available for all ARM targets, this option cannot be used for ARMv8 because ARMv8 (both AArch32 and AArch64) does not support `gold`.

For more information on the `gold` linker see llvm.org/docs/GoldPlugin.html.

3.4.11 Preprocessor

-A pred=*ans*
Assert the predicate *pred* and answer *ans*.

-A -pred=*ans*
Cancel the specified assertion.

-ansi
Use C89 standard.

-C
Retain comments during preprocessing.

-CC
Retain comments during preprocessing, including during macro expansion.

-d(**DMN**)
D Print macro definitions in -E mode in addition to normal output

M Print macro definitions in -E mode instead of normal output

N Print macro names in -E mode in addition to normal output

-D *name*
-D name=*definition*
Define the specified macro symbol.

-fexec-charset=charset
Specify the character set used to encode strings and character constants. The default character set is UTF-8.

-finput-charset=charset
Specify the character set used to encode the input files. The default is UTF-8.

-fpch-deps
Cause the dependency-output options to additionally list the files from a precompiled header's dependencies.

-fpreprocessed
Notify the preprocessor that the input file has already been preprocessed.

-fstrict-overflow
Enforce strict language semantics for pointer arithmetic and signed overflow.

-ftabstop=*width*
Specify the tab stop distance.

-fwide-exec-charset=charset
Specify the character set used to encode wide strings and character constants. The default character set is UTF-32 or UTF-16, depending on the size of `wchar_t`.

-fworking-directory
Generate line markers in the preprocessor output. The compiler uses this to determine what the current working directory was during preprocessing.

--help
Display the preprocessor release version.

-H
Display the header includes and nesting depth.

-I *dir*
Add the specified directory to the list of search directories for header files.

-I-
This option is deprecated.

-include *file*
Include the contents of the specified source file.

-isystem *prefix*
Treat an included file as a system header if it is found on the specified path (Section 3.5.6).

-isystem-prefix *prefix*
Treat an included file as a system header if it is found on the specified subpath of a defined include path (Section 3.5.6).

-ino-system-prefix *prefix*
Do not treat an included file as a system header if it is found on the specified subpath of a defined include path (Section 3.5.6).

-M
Output a `make` rule describing the dependencies of the main source file.

-MD
Equivalent to `-M -MF` *file*, except `-E` is not implied.

-MF *file*
Write dependencies to the specified file.

-MG
Add missing headers to the dependency list.

-MM
Equivalent to `-M`, except do not mention header files found in the system header directories.

-MMD
Equivalent to `-MD`, except only mention user header files, not system header files.

-MP
Create artificial target for each dependency.

-MQ *target*
Specify target to quote for dependency.

-MT *target*
Specify target for dependency.

-nostdinc
Omit searching for header files in the standard system directories.

-nostdinc++
Omit searching for header files in the C++-specific standard directories.

-o *file*
Specify the name of the preprocessor output file.

-P
Disable linemarker output when using `-E`.

-remap
Generate code for file systems that only support short file names.

--target-help
Display all command options and exit immediately.

-traditional-cpp
Emulate pre-standard C preprocessors.

-trigraphs
Preprocess trigraphs.

-U *name*
Cancel any previous definition of the specified macro symbol.

-v
Equivalent to `-help`.

-version
Display the preprocessor version during preprocessing.

--version
Display the preprocessor version and exit immediately.

-w
Suppress all preprocessor warnings.

-Wall
Enable all warnings.

-Wcomment
-Wcomments
Generate warning if a comment symbol appears inside a comment.

-Wendif-labels
Generate warning if an `#else` or `#endif` directive is followed by text.

-Werror
Convert all warnings into errors.

-Wimport
Generate warning when `#import` is used the first time.

-Wsystem-headers
Generate warning for constructs declared in system header files.

-Wtrigraphs
Generate warning if a trigraph forms an escaped newline in a comment.

-Wundef
Generate warning if an undefined non-macro identifier appears in an `#if` directive.

-Wunused-macros
Generate warning if a macro is defined without being used.

3.4.12 Assembling

-integrated-as
-no-integrated-as
Use the LLVM integrated assembler when compiling C and C++ source files.

`-no-integrated-as` explicitly disables the use of the integrated assembler.

By default, the integrated assembler is enabled.

NOTE If a program can potentially generate hardware divide instructions (`SDIV`, `UDIV`), it is strongly recommended to use the integrated assembler. Older GNU assemblers may not understand these instructions.

When directly assembling a `.s` source file, LLVM still invokes the external assembler because it cannot correctly translate all GNU assembly language constructions. As a result, not all GNU assembler options (which are passed with the `-Wa` option) will work with the integrated assembler.

The integrated assembler can process its own assembly-generated code, along with most hand-written assembly that conforms to the GNU assembly syntax.

-Xassembler *arg*
Pass the specified argument to the assembler.

3.4.13 Linking

object_file_name
Linker input file.

-c
Do not perform linking. This option is used with spec strings.

-dynamic
Link with a shared library (instead of a static library).

-E
Do not perform linking. This option is used with spec strings.

-l *library*
Search the specified library file while linking.

-moslib=*library*
Search the RTOS-specific library named `lib`*library*`.a`. The search paths for the library and include files must be explicitly specified.

-nodefaultlibs
Do not use the standard system libraries when linking.

-nostartfiles
Do not use the standard system startup files when linking.

-nostdlib
Do not use the standard system startup files or libraries when linking.

-pie
Generate a position-independent executable as the output file.

-s
Delete all symbol table information and relocation information from the executable.

-S
Do not perform linking. This option is used with spec strings.

-shared
Generate a shared object as the output file. The resulting file can be subsequently linked with other object files to create an executable.

-shared-libgcc
Link with the shared version of the library`libgcc`.

-static
Do not link with the shared libraries. Only relevant when using dynamic libraries.

-static-libgcc
Link with the static version of the library`libgcc`.

-symbolic
Bind references to global symbols when building a shared object.

-u *symbol*
Pretend the symbol *symbol* is undefined, to force linking of library modules to define it.

-Xlinker *arg*
Pass the specified argument to the linker.

3.4.14 Directory search

-B*prefix*
Specify the top-level directory of the compiler.

-F *dir*
Add the specified directory to the search path for framework includes.

--gcc-toolchain=*prefix*
Equivalent to `-B`.

-I *dir*
Add the specified directory to the include file search path.

-I-
This option is deprecated.

-L*dir*
Add the specified directory to the list of directories searched by the `-l` option.

--sysroot=*prefix*
Specify the root directory of the system tools environment (Section 3.7).

3.4.15 Processor version

LLVM defines the options `-target`, `-march`, and `-mcpu` for specifying the ARM processor version to generate code for.

If none of these options are specified, the LLVM compilers by default generate code for the lowest ARMv4t instruction set architecture, in ARM mode, for the ARM7tdmi CPU.

If the ARMv7 or ARMv8 architecture is specified using the options `-march` or `-mcpu`, but ARM mode (i.e., 32-bit-only mode) is not specified on the command line, the LLVM compilers default to generating code in Thumb2 mode. To disable Thumb mode, use the options `-mno-thumb` or `-marm`.

`-target` *`triple`*

Specify the ARM architecture, operating system, and ABI for code generation.

The `triple` argument has the following format:

`arch-platform-abi`

For example, to generate code for the ARMv7a which runs on Linux and conforms to `gnueabi`, specify the following option:

```
clang -target armv7a-linux-gnueabi foo.c
```

The best way to specify the architecture version and CPU is by using the `-march` and `-mcpu` options respectively. Even though a target triple can be used to specify the architecture, it must match the GCC tools sysroot (Section 3.7). Thus, the above command can be alternately expressed as follows:

```
clang -target arm-linux-gnueabi -mcpu=cortex-a9 foo.c
```

... where `cortex-a9` indicates ARMv7a as the CPU.

Here are some commonly-used target triples:

`arm-linux-gnueabi`

`arm-none-linux-gnueabi` (equivalent to `arm-linux-gnueabi`)

`arm-linux-androideabi` (for code conforming to Android EABI)

`aarch64-linux-gnu` (for ARMv8 AArch64 mode)

`armv8-linux-gnu` (for ARMv8 AArch* mode)

`arm-none-eabi` (for ARM bare-metal executables)

NOTE In older versions of LLVM the `-target` option was named `-triple`.

-march=*version*

Specify the ARM architecture for code generation.

This option has the following possible values:

```
armv2
armv2a
armv3
armv3m
armv4
armv4t
armv5
armv5t
armv5e
armv5te
armv5tej
armv6
armv6k
armv6j
armv6z
armv6t2
armv7
armv7-a
armv7e-m
armv7-f
armv7-s
armv7-r
armv7-m
armv8
armv8-a
```

-mcpu=*version*

Specify the ARM CPU for code generation.

For a complete list of the values defined for this option, run the following command:

```
llvm-as | </dev/null | llc -march=arm -mcpu=help
```

Here are some commonly-used CPU values:

```
cortex-a8       (ARMv7 CPUs)
cortex-a9
cortex-a15
scorpion        (Qualcomm ARMv7 CPUs)
krait2
cortex-a53      (ARMv8 CPUs)
cortex-a57
```

-mfpu=*version*

Specify the ARM architecture extensions.

For a complete list of the values defined for this option, run the following command:

```
llvm-as | </dev/null | llc -march=arm -mcpu=help
```

Here are some commonly-used FPU values:

neon

Enable the Neon single instruction, multiple data (SIMD) architecture extension for the ARM Cortex-A (`cortex-a9`) or Qualcomm ARM v7 (`krait2`) and ARMv8 processors.

vfpv4

Enable the VFPv4 architecture extensions. The VFPv4 extension enables code generation of the fused multiply add and subtract instructions (Section 3.4.16).

neon-fp-armv8

Enable ARMv8 FP and Neon extensions.

crypto-neon-fp-armv8

Enable ARMv8 FP, Neon, and Cryptography extensions.

NOTE `-mcpu=krait2` automatically enables both `neon` and `vfpv4`.

-mfloat-abi=(**soft**|**softfp**|**hard**)

Specify the floating-point ABI.

NOTE ARMv8 mandates hardware floating point.

3.4.16 Code generation

-fasynchronous-unwind-tables
Generate unwind table. The table is stored in DWARF2 format.

-fchar-array-precise-tbaa
-fno-char-array-precise-tbaa
Prevent aliasing of `char` arrays by non-`char` pointers.

This option causes the compiler to assume that no pointer other than a pointer to `char` can reference an element in a `char` array.

The default is disabled.

NOTE **-fchar-array-precise-tbaa** is enabled by default at the `-Ofast` level.

In the example below, enabling **-fchar-array-precise-tbaa** results in the statement "`d = *p`" being hoisted out, because `p` is a pointer to `int`.

```
typedef struct {
  char a;
  char b[100];
  char c;
 } S;

int *p;
S x;

void func1 (char d) {
  for (int i = 0; i < 100; i++) {
    x.b[i] += 1;
    d = *p;
    x.a += d;
  }
}
```

-femit-all-data
Emit all data, even if unused.

-femit-all-decls
Emit all declarations, even if unused.

-ffp-contract=(fast|on|off)

Fused multiply add and subtract operations (VFMLA,VFMS) are more accurate than chained multiply add and subtract operations (VMLA, VMLS) because the chained operations perform rounding both after the multiply and before the add/subtract. While rounding itself introduces only a small error, cumulatively it can have a huge impact on the final result.

While fused operations are IEEE compliant, it is not IEEE compliant for the compiler to automatically replace a multiply followed by an add/subtract (or VMLA/VMLS) with the equivalent fused operation, since the numeric result can differ so much. However, if a programmer explicitly specifies the use of a fused operation, then the substitution is considered IEEE compliant.

Fused operations are explicitly specified with the `-ffp-contract` option. It has the following possible values:

fast

Enable fused operations throughout the program.

on

Enable fused operations according to the `FP_CONTRACT` pragma (default).

off

Disable fused operations throughout the program.

NOTE This option must be used with the `-mfpu=neon-vfpv4` option.

Enabling fused operations causes the compiler to relax IEEE compliance for floating point computation.

-fno-exceptions

Do not generate code for propagating exceptions.

-finstrument-functions

Generate instrumentation calls in function entries and exits.

-fmerge-functions
-fno-merge-functions

Attempt to merge functions that are equivalent, or differ by only a few instructions (Section 3.6.5). The default setting is disabled.

This option attempts to improve code size by merging similar functions. It uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

NOTE Because this option may have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

-fpic

Generate position-independent code (PIC) for use in a shared library.

-fPIC
Generate position-independent code for dynamic linking, avoiding any limits on the size of the global offset table.

-fpie
-fPIE
Generate position-independent code (PIC) for linking into executables.

-fshort-enums
-fno-short-enums
Allocate to an enum type only as many bytes necessary for the declared range of possible values. The default is disabled.

-fshort-wchar
-fno-short-wchar
Force `wchar_t` to be `short unsigned int`. The default is disabled.

-ftrap-function=*name*
Issue a call to the specified function rather than a trap instruction.

-ftrapv
Trap on integer overflow.

-ftrapv-handler=*name*
Specify the function to be called in the case of an overflow.

-funwind-tables
Similar to `-fexceptions`, except that it only generates any necessary static data, without affecting the generated code in any other way.

-fverbose-asm
Add commentary information to the generated assembly code to improve code readability.

-fvisibility=[**default**|**internal**|**hidden**|**protected**]
Set the default symbol visibility for all global declarations.

-fwrapv
Treat signed integer overflow as two's complement.

-mhwdiv=(**arm**|**thumb**|`arm`,**thumb**|**none**)
Control the generation of hardware divide instructions in ARM or Thumb mode.

arm

Generate hardware divide instructions in Arm mode only.

thumb

Generate hardware divide instructions in Thumb mode only.

arm,thumb

Generate hardware divide instructions in ARM and Thumb modes.

none

Do not generate hardware divide instructions (default).

NOTE This option applies only to ARMv7 processors that support hardware divide.

`-mcpu=krait2` automatically sets `-mhwdiv=arm,thumb`.

-mllvm -aarch64-disable-abs-reloc

Eliminate absolute relocation by changing all global variable references to be PC-relative.

This option is commonly used with `-mllvm -emit-cp-at-end`.

-mllvm -aggressive-jt

A jump table is an efficient method to optimize switch statements by replacing them with unconditional branch instructions and simple operations to transfer program flow to them.

This option enables switch statements with small ranges to be automatically converted to jump tables.

The default is disabled.

-mllvm -arm-expand-memcpy-runtime

Set a threshold of 8 or 16 bytes for expanding (inlining) memcpy calls.

This option enables the generation of runtime checks for copy sizes 8 or 16 bytes, and inlining of memcpy calls that have copy sizes smaller than or equal to 8 or 16 bytes. For any other copy size the memcpy function is invoked.

Enabling this option causes an LLVM IR-level transformation. The resultant code might be vectorized, if Neon is enabled.

This option is effective for optimization level equal or higher than `-O1`, `Os`, and `Oz`. Otherwise it is silently ignored.

-mllvm -arm-memset-size-threshold

Control the code generation for memset library calls using NEON vector stores.

This option specifies the maximum number of bytes of data in memset call that should be implemented with NEON vector stores. A memset call with data size above the specified threshold will not be compiled into vector store operations.

The default is 128.

-mllvm -arm-memset-size-threshold-zeroval

Control the code generation for memset library calls that write 0 value using NEON vector stores.

This option specifies the maximum number of bytes of data in memset call that writes 0 value that can be implemented with NEON vector stores. A memset call that writes 0 value with data size above the specified threshold will not be compiled into vector store operations.

The default is 32.

-mllvm -arm-opt-memcpy

The optimized libc for Krait targets includes two specialized memcpy functions for copy sizes greater than 8 and 16 bytes:

```
memcpyGT8(void*, const void*, size_t)
memcpyGT16(void*, const void*, size_t)
```

When this option is set in conjunction with `-mllvm -arm-expand-memcpy-runtime`, the compiler transforms the LLVM IR by replacing memcpy calls with the runtime checks for copy size less than or equal to 8 or 16 bytes and these specialized memcpy calls. Their implementation uses vector instructions and requires Neon to be enabled.

Note the user needs to additionally set the option for copy size threshold, `-mllvm -arm-expand-memcpy-runtime`.

This option has no effect if `-mllvm -arm-expand-memcpy-runtime` is disabled.

This option is effective for optimization level equal or higher than `-O1`, `Os` and `Oz`. Otherwise it is silently ignored.

The default is disabled.

-mllvm -disable-thumb-scale-addressing

Control the code generation of scaled immediate addressing in Thumb mode.

By default scaled immediate addressing is enabled in Thumb mode, unless `-mtune=krait2` or `-mcpu=krait2` are set in the command line.

To disable it, set `-mllvm -disable-thumb-scale-addressing=true`.

-mllvm -emit-cp-at-end

Place constant pool at the end of a function.

When this option is used in conjunction with **-mllvm -aarch64-disable-abs-reloc** (which changes all global variable references to be PC-relative), the compiler places the constant pool at the end of a function.

The default is disabled.

In the following example global variable "`a`" is loaded using the default relocation code:

```
movz x8, #:abs_g3:a
movk x8, #:abs_g2_nc:a
movk x8, #:abs_g1_nc:a
movk x8, #:abs_g0_nc:a
ldr w0, [x8]
```

Enabling this option with **-mllvm -aarch64-disable-abs-reloc** changes the code to the following:

```
ldr x8, .LCPI0_0
ldr w0, [x8]
ret
.LCPI0_0:
.xword a    // address of "a"
```

-mllvm -enable-arm-addressing-opt

Promotes use of optimized address modes by merging ADD operations into the associated LOAD instruction.

The default is enabled.

-mllvm -enable-arm-peephole

Enable peephole optimizations to eliminate VMOV instructions, which can be an expensive operations.

This option controls two peephole optimizations:

- Eliminate vmovs from D to R to S

Eliminates excess VMOVs that result from copying a value from a D register to an S register.

There is no copy instruction from D to S, so the code generator inserts a VMOV from D to R and then another VMOV from R back to S. This peephole optimization eliminates the VMOVs by using D registers that alias S registers – registers D0-D15 are aliases as S0-S31. No copy is necessary to get to an S register from these D registers.

- Eliminate VMOVs from D to R for an ADD operation.

Eliminates excess VMOVs that result from an ADD instruction whose operands are defined by VMOVs from a D register. The ADD is replaced with a horizontal ADD using the VPADD instruction and a VMOV to get the result to the R register.

The default is enabled.

-mllvm -enable-arm-zext-opt

Removes redundant ZERO-EXTEND operations, for example, when preceded by a LOAD instruction that zero-extends the value to 32 bits as part of its operation.

The default is enabled.

-mllvm -enable-print-fp-zero-alias

When this option is used in conjunction with `-no-integrate-as`, the compiler prints FP compare-with-zero instructions using the alias format "fcmXY ..., #0" instead of the default LLVM format "fcmXY ..., #0.0" specified in the ARMv8 documentation.

This ensures assembly code compatibility between LLVM and GNU tools while the tools are out of sync (i.e., the 4.9 GNU assembler currently uses "#0" syntax).

-mllvm -enable-round-robin-RA

Enable a round-robin register allocation heuristic which selects registers avoiding back-to-back reuse to minimize false data dependency.

This heuristic works well for targets with limited register renaming capability, as in Krait targets.

The default is disabled, unless `-mtune=krait2` or `-mcpu=krait2` is specified.

`-mllvm -enable-select-to-intrinsics`

Expose more if-statements to be converted into LLVM IR's SELECT instruction which in turn can more easily be mapped to ARM HW instructions.

The default is disabled, unless `-mtune=krait2` or `-mcpu=krait2` is specified.

`-mllvm -favor-r0-7`

Enable a heuristic in the Greedy Register Allocator that better guides the assignment of high-order registers (R8-R15) which are currently avoided aggressively in the allocator. The allocator exploits the fact that a Thumb2 instruction that uses one of R8-15 registers must be encoded in 32 bits. So a candidate assigned to these registers has a very a high cost.

With this option, the allocator avoids this register assignment based on an additional cost, the candidate frequency in a function. The benefits are better code size reduction, better performance/power generated from better code density, and reduced spilling. This change impacts mostly Thumb code generation, but ARM code generation can also be affected because it disables R8-15 register avoidance.

The default is disabled.

NOTE Use `-falign-inner-loops` with `-favor-r0-7` to achieve the maximum benefit from loop alignment.

`-mllvm -prefetch-locality-policy`=(**`L1`**|**`L2`**|**`L3`**|**`stream`**)

Configure data prefetch to be temporal or non-temporal.

`L1`

Temporal or retained prefetch allocated in L1 cache.

`L2`

Temporal or retained prefetch allocated in L2 cache.

`L3`

Temporal or retained prefetch allocated in L3 cache.

`stream`

Streaming or non-temporal prefetch.

The default is `L1`.

NOTE This option is available only for AArch64, and only when `-fprefetch-loop-arrays` is enabled.

-mrestrict-it
-mno-restrict-it

Control the code generation of IT blocks.

In the ARMv8 architecture (AArch32) IT blocks are deprecated in Thumb mode. They can only be one instruction long, and can only contain a subset of all 16-bit instructions.

`-mrestrict-it` disallows the generation of IT blocks that are deprecated in ARMv8.

`-mno-restrict-it` allows generation of legacy IT blocks (i.e., deprecated forms in ARMv7).

The default option setting is determined by the target architecture (ARMv8 or ARMv7). For ARMv8 (AArch32) Thumb mode, `-mrestrict-it` is enabled by default, while for other targets it is disabled by default.

-mtune=krait2

Generate a common-denominator binary which runs best on the Krait micro-architecture while remaining compatible with older hardware (such as ARM v6 or other cores such as Cortex-A9).

NOTE This option can be used only with the options `-mcpu=cortex-a8`, `-mcpu=cortex-a9`, or `-mcpu=arm1136jf-s`.

This option is equivalent to specifying `-mcpu=krait2` while explicitly disabling the unsupported instructions in the older hardware.

3.4.17 Vectorization

-fvectorize-loops

Perform automatic vectorization of loop code (Section 3.6.3).

Vectorization is subject to the following constraints:

- On nested loops it is performed only on the innermost loop.
- It can be used only with code optimization level `-O2` or higher (such as `-O3` or `-Ofast`).
- It works only with the ARMv7 or ARMv8 processor architecture with the Neon extension. Neon is enabled either implicitly (by specifying a target processor such as Krait), or explicitly with `-mfpu=neon`.

NOTE `-fvectorize-loops` is enabled by default at the `-O3` and `-Ofast` levels.

`-fvectorize-loops` is an alias of the GCC option `-ftree-vectorize`.

-fvectorize-loops-debug

Equivalent to `-fvectorize-loops`, but also generates a report indicating which loops in the program were vectorized.

-mllvm -print-vectorized-loops
Generate verbose information on vectorized loops.

NOTE This option works best when used with the `-g` option to print out the precise location of the loops that get vectorized.

The GCC option `-ftree-vectorizer-verbose` is not supported in LLVM.

-fprefetch-loop-arrays[=*stride*]
-fno-prefetch-loop-arrays
Control the automatic insertion of ARM `PLD` instructions into loops that are vectorized.

The argument *`stride`* specifies the distance that the `PLD` instruction attempts to load. If the argument is omitted, the compiler automatically chooses a value.

The default is disabled.

NOTE This option must be used with the `-fvectorize-loops` option.

3.4.18 Parallelization

-fparallel
Perform automatic parallelization of loop code (Section 3.6.4).

Parallelization is subject to the following constraints:

- It must be specified (on the command line) when compiling each `.c` or `.cpp` file.
- It must additionally be specified on the command line that directs linking.
- It must be used with the option `-O2`, `-O3`, or `-Ofast`.

NOTE This option cannot be used with the AArch64 target, as parallelization is currently not supported for Android 64-bit environments.

3.4.19 Optimization

-O0

Do not optimize. This is the default optimization setting.

-O
-O1

Enable the following code optimizations:

```
-fdefer-pop
-fmerge-constants
-fthread-jumps
-floop-optimize
-fif-conversion
-fif-conversion2
-fdelayed-branch
-fguess-branch-probability
-fcprop-registers
```

-O2

Enable the code optimizations specified by `-O`, and the following additional optimizations:

```
-foptimize-sibling-calls
-fstrength-reduce
-fcse-follow-jumps  -fcse-skip-blocks
-frerun-cse-after-loop  -frerun-loop-opt
-fgcse  -fgcse-lm  -fgcse-sm  -fgcse-las
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fregmove
-fschedule-insns  -fschedule-insns2
-fsched-interblock  -fsched-spec
-fcaller-saves
-fpeephole2
-freorder-blocks  -freorder-functions
-fstrict-aliasing
-funit-at-a-time
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcrossjumping
```

-O3

The following optimizations are enabled at the -O3 level:

- -fvectorize-loops
- -mllvm -enable-merge-select=true
- -mllvm -assume-safe-shifts=true
- If -mcpu=arm1136jf, -mcpu=krait2, or -mcpu=cortex-a9 are set in the command line, the following additional options are set:
  - -mllvm -scev-reuse-gep=true
  - -mllvm aggr-basicaa=true
  - -mllvm value-track-phi=true

For details on the -mllvm options listed above, see the LLVM documentation.

When more than one optimization level among Os, O0, O1, O2, and O3 are present in the command line, the last occurrence of the -O option prevails.

-Os

Enable all of the -O2 code optimizations that do not typically increase code size, and perform additional optimizations which reduce code size.

-Ofast

The following optimizations are enabled at the -Ofast level:

- All options enabled with -O3 (including -fvectorize-loops)
- -mllvm -switch-transpose=true
- -mllvm -unroll-allow-partial=true
- -mllvm -unroll-threshold=1000
- -mllvm -inline-threshold=375
- -ffast-math and -fmath-errno
- If compiling for ARM mode:
  - -mllvm -unroll-rt-prolog=false
- If compiling for Thumb mode:
  - -mllvm -unroll-rt-prolog=true
  - -mllvm -enable-lsr-nested=true
  - -mllvm -lsr-no-outer=false
  - -mllvm -favor-r0-7=true
- If -mllvm -favor-r0-7=true is successfully set, then -falign-inner-loops=8 option is also enabled.

For details on the -mllvm options listed above, please refer to the LLVM documentation. The LLVM options -mllvm -favor-r0-7 and -falign-inner-loops are further described in this document.

If the user sets any of the above options in the command line, then the user setting prevails. For example, it the user sets `-mllvm -favor-r0-7=false` or `-fno-align-inner-loops`, then `-Ofast` will not enable favoring r0 to r7 registers nor inner loops alignment.

If `-Ofast` is combined with any other optimization level (`-Os`, `-O0` to `-O3`) in the command line, the last `-O` option prevails.

The following are the recommended options for performance and code size optimizations.

1. Performance optimizations

The LLVM compilers generate the best performing code with the `-Ofast` option.

The following combination of options are recommended for Krait cores:

`-Ofast -mcpu=krait2` (Thumb mode and `-fvectorize-loops` are enabled by default)

`-Ofast -mcpu=krait2 -marm` (ARM mode and `-fvectorize-loops` are enabled by default)

For non-Krait cores the following options are recommended:

```
-O3 -mtune=krait2 -mllvm -unroll-threshold=1000 -mllvm -unroll-allow-partial -mllvm -inline-threshold=325
```

The `-unroll-threshold` and `-inline-threshold` options increase the limits of loop unrolling and inlining respectively to exploit superscalar architectures such as Krait. In general, more aggressive loop unrolling and function inlining contributes to better performance.

2. Code-size optimizations

Currently, LLVM generates smallest code when compiled for Thumb2 mode. The following are the options recommended for generating compact code:

```
-Os -mthumb
```

`-Osize`

Enable `-Os` level optimizations and some additional options that trade off performance for best code size.

If `-Osize` is combined with any other optimization level (`-Os`, `-O0` to `-O3`) in the command line, the last `-O` option prevails.

3.4.20 Specific optimizations

`-falign-functions``[=`*`n`*`]`
Control function alignment.

Setting `-falign-functions=1` and `-fno-align-functions` are equivalent, resulting in disabling function alignment.

Setting `-falign-functions=0` or `-falign-functions` (with no value specified) enables function alignment using the target's default alignment value.

Setting `-falign-functions=`*`n`* enables function alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align functions.

To enable function alignment at the `-Os` level, an additional `falign-os` option must be set.

`-falign-jumps``[=`*`n`*`]`
Control jump alignment.

Setting `-falign-jumps=1` and `-fno-align-jumps` are equivalent, resulting in disabling jump alignment.

Setting `-falign-jumps=`*`n`* enables jump alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align jumps.

To enable jump alignment at the Os level, an additional `falign-os` option must be set.

`-falign-labels``[=`*`n`*`]`
Control label (branch target) alignment.

Setting `-falign-labels=1` and `-fno-align-labels` are equivalent, resulting in disabling label alignment.

Setting `-falign-labels =0` or `-falign-labels` (with no value specified) enables labels alignment using the target's default alignment value.

Setting `-falign-labels=`*`n`* enables label alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align labels.

To control the type of label that should be aligned, use the `-mllvm branch-target-align` option with strings "none" (do not align branch targets), "nocalls" (do not align function calls), "allcalls" (align after function calls).

To enable label alignment at the Os level, `-falign-os` must also be set.

-falign-loops[=*n*]
Control loop alignment.

Setting `-falign-loops=1` and `-fno-align-loops` are equivalent, resulting in disabling loop alignment.

Setting `-falign-loops=0` or `-falign-loops` (with no value specified) enables loop alignment using the target's default alignment value.

Setting `-falign-loops=`*n* enables loop alignment using the next power-of-two greater than *n* as the alignment value, where *n* is the number of bytes.

The default is to not align loops.

To enable loop alignment at the Os level, an additional `falign-os` option must be set.

-falign-inner-loops
-fno-align-inner-loops
Control innermost loop alignment. When enabled, only the start basic block of innermost loops is aligned.

Setting `-falign-inner-loops=1` and `-fno-align-inner-loops` are equivalent, resulting in disabling innermost loop alignment.

Setting `-falign-inner-loops=0` or `-falign-inner-loops` (with no value specified) enables innermost loop alignment using the target's default alignment value.

Setting `-falign-inner-loops=`*n* enables innermost loop alignment using the next power-of-two greater than *n* as the alignment value, where *n* is the number of bytes.

`-falign-loops` and `-falign-inner-loops` are incompatible and cannot be set simultaneously, otherwise a compiler error is generated.

The default is to not align innermost loops.

To enable innermost loop alignment at the Os level, an additional `-falign-os` must be set.

-falign-os
-fno-align-os
Control alignment in Os level.

The default is to ignore alignment options in Os level.

When enabling alignment of loops, functions and labels in Os level set `-falign-os`, otherwise the compiler generates a warning of unused option. To disable it, set `-fno-align-os`.

-fdata-sections
Assign each data item to its own section.

-finline
Specify the `inline` keyword as active.

-finline-functions
Perform heuristically-selected inlining of functions.

-fnomerge-all-constants
Do not merge constants.

-fomit-frame-pointer
Do not store the stack frame pointer in a register if it is not required in a function.

-foptimize-sibling-calls
Optimize function sibling calls and tail-recursive calls.

-fstack-protector
Generate code which checks selected functions for buffer overflows.

-fstack-protector-all
Generate code which checks *all* functions for buffer overflows.

-fstrict-aliasing
Enforce the strictest possible aliasing rules for the language being compiled.

-funit-at-a-time
Parse the entire compilation unit before beginning code generation.

-funroll-all-loops
Unroll *all* loops.

-funroll-loops
Unroll selected loops.

-fno-zero-initialized-in-bss
Assign all variables that are initialized to zero to the BSS section.

3.4.21 Math optimization

-fassociative-math
Allow the operands in a sequence of floating-point operations to be re-associated.

Because this option may reorder floating-point operations, it should be used with caution when exact results are required (with no expectation of an error cutoff).

To use this option, both `-fno-signed-zeros` and `-fno-trapping-math` must be enabled, while `-frounding-math` must *not* be enabled.

NOTE This option enables additional features of parallelization (Section 3.6.4).

-ffast-math
Enable 'fast-math' mode in the compiler front-end. This has no effect on optimizations, but defines the preprocessor macro `__FAST_MATH__` which is the same as the GCC `-ffast-math` option.

-ffinite-math-only
Enable optimizations which assume that floating-point argument and result values are never NaNs nor +-Infs.

-fno-math-errno
Do not set ERRNO after using single-instruction math functions.

-freciprocal-math
Enable optimizations which assume that the reciprocal of a value can be used instead of dividing by the value.

-fno-signed-zeros
Enable optimizations which ignore the sign of floating point zero values.

-fno-trapping-math
Enable optimizations which assume that floating-point operations cannot generate user-visible traps.

-funsafe-math-optimizations
Enable code optimizations which assume that the floating-point arguments and results are valid, and which may violate IEEE or ANSI standards.

This option enables `-fno-signed-zeros`, `-fno-trapping-math`, `-fassociative-math`, and `-freciprocal-math`.

3.4.22 Link-time optimization

-c-lto
Run the LLVM compiler up to and including the object-generation step of the link-time optimizer.

-flto
Invoke link-time optimizer (Section 3.6.6).

This option can be used when the files in a program are compiled separately. In this case the option must be specified when compiling each source file, and again when the compiler is used to link the resulting object files.

When this option is used with `-c`, it produces a bitcode file which is used during link-time optimization.

NOTE This option must be used with `-fuse-ld=gold`. The `gold` linker provides a plugin interface which is required for link-time optimization. All compile-time options must be passed to the linker for LTO to use them for optimization and code generation. To ensure that the options are passed correctly, it is strongly suggested to use `clang/clang++` to perform the linking.

This option cannot be used with the Windows version of the LLVM compilers, because the `gold` plugin is not available on those compilers.

NOTE The `gold` plugin for LLVM compiler is not distributed as part of the Linux version of the LLVM compiler. If you are using Android NDK version r10, copy the file `LLVMgold.so` from "*Android-NDK-root*`/toolchains/llvm-3.4/prebuilt/linux-x86_64/lib`" to "`$INSTALL_PREFIX/lib`".

-flto-scope=(**program**|**library**)
Specify the scope of link-time optimization.

`program`

Apply intermodular optimization across all bitcode objects and archives.

`library`

Apply intermodular optimization only at archive boundaries.

The default is `program`.

-lto-no-inter-mod-inline
Disable inlining across module boundaries during link-time optimization.

-lto-preserve *symbol*
Prevent link-time optimization from discarding symbol.

-lto-preserve-list *filename*
Specify symbols that the link-time optimizer should preserve. The specified text file contains a list of symbols, with each symbol on a separate line in the file.

-lto-w-no-unknown-sym
Disable warnings about preserving unknown symbols.

-Wlto ***arg,*** **...**
Pass the specified arguments to the link-time optimizer. The argument list follows the option, and is comma-separated.

3.4.23 Secure programming

--analyze
Invoke static program analyzer (Section 3.11.1) on the specified input files.

-analyzer-checker=*checker*
Enable the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Enabling a checker category enables all the checkers in that category.

For a complete list of checker names use `-analyzer-checker-help`.

-analyzer-checker-help
List the complete set of checkers and their categories for use in `-analyzer-checker` and `-analyzer-checker-disable`.

-analyzer-disable-checker=*checker*
Disable the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Disabling a checker category disables all the checkers in that category.

For a complete list of checker names use `-analyzer-checker-help`.

--analyzer-output html
Generate the static analyzer output report in HTML format.

The default report format is plist.

--compile-and-analyze *dir*
Compile input files while running the static analyzer. The analysis report files are written to the specified directory.

3.5 Warning and error messages

LLVM provides a number of ways to control which code constructs cause the compilers to emit errors and warning messages, and how the messages are displayed to the console.

3.5.1 Controlling how diagnostics are displayed

When LLVM emits a diagnostic, it includes rich information in the output, and gives you fine-grain control over which information is printed. LLVM has the ability to print this information. The following options are used to control the information:

- A file/line/column indicator which shows exactly where the diagnostic occurs in your code.
- A categorization of the diagnostic as a note, warning, error, or fatal error.
- A text string describing the problem.
- An option indicating how to control the diagnostic (for diagnostics that support it) [`-fdiagnostics-show-option`].
- A high-level category for the diagnostic for clients that want to group diagnostics by class (for diagnostics that support it) [`-fdiagnostics-show-category`].
- The line of source code that the issue occurs on, along with a caret and ranges indicating the important locations [`-fcaret-diagnostics`].
- "FixIt" information, which is a concise explanation of how to fix the problem (when LLVM is certain it knows) [`-fdiagnostics-fixit-info`].
- A machine-parseable representation of the ranges involved (disabled by default) [`-fdiagnostics-print-source-range-info`].

For more information on these options see Section 3.4.7.

3.5.2 Diagnostic mappings

All diagnostics are mapped into one of the following classes:

- Ignored
- Note
- Warning
- Error
- Fatal

3.5.3 Diagnostic categories

Though not shown by default, diagnostics can each be associated with a high-level category. This category is intended to make it possible to triage builds which generate a large number of errors or warnings in a grouped way.

Categories are not shown by default, but they can be turned on with the `-fdiagnostics-show-category` option (Section 3.4.7). When this option is set to "`name`", the category is printed textually in the diagnostic output. When set to "`id`", a category number is printed.

NOTE The mapping of category names to category identifiers can be obtained by invoking LLVM with the option `--print-diagnostic-categories`.

3.5.4 Controlling diagnostics with compiler options

LLVM can control which diagnostics are enabled through the use of options specified on the command line.

The `-W` options are used to enable warning diagnostics for specific conditions in a program. For instance, `-Wmain` will generate a warning if the compiler detects anything unusual in the declaration of function `main()`.

`-Wall` enables *all* the warnings defined by LLVM. `-w` disables all of them.

Warnings for a specific condition can be disabled by specifying the corresponding `-W`*cond* option as `-Wno-`*cond*. For instance, `-Wno-main` disables the warning normally enabled by `-Wmain`.

`-Werror=`*cond* changes the specified warning to an error (Section 3.5.2). `-Werror` specified without a condition changes *all* the warnings to errors. `-ferror-warn` changes just the warnings that are listed in the specified text file.

`-pedantic` and `-pedantic-errors` enable diagnostics that are required by the ISO C and ISO C++ standards.

3.5.5 Controlling diagnostics with pragmas

LLVM can also control which diagnostics are enabled through the use of pragmas in the source code. This is useful for disabling specific warnings in a section of source code. LLVM supports GCC's pragma for compatibility with existing source code, as well as several extensions.

The pragma may control any warning that can be used from the command line. Warnings can be set to ignored, warning, error, or fatal. The following example instructs LLVM or GCC to ignore the `-Wall` warnings:

```
#pragma GCC diagnostic ignored "-Wall"
```

In addition to all the functionality provided by GCC's pragma, LLVM also enables you to push and pop the current warning state. This is particularly useful when writing a header file that will be compiled by other people, because you don't know what warning flags they build with.

In the below example `-Wmultichar` is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed:

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"

char b = 'df'; // no warning.

#pragma clang diagnostic pop
```

The `push` and `pop` pragmas save and restore the full diagnostic state of the compiler, regardless of how it was set. That means that it is possible to use push and pop around GCC-compatible diagnostics, and LLVM will push and pop them appropriately, while GCC will ignore the pushes and pops as unknown pragmas.

NOTE While LLVM supports the GCC pragma, LLVM and GCC do not support the same set of warnings. Thus even when using GCC-compatible pragmas there is no guarantee that they will have identical behavior on both compilers.

3.5.6 Controlling diagnostics in system headers

Warnings are suppressed when they occur in system headers. By default, an included file is treated as a system header if it is found in an include path specified by `-isystem`, but this can be overridden in several ways.

The `system_header` pragma can be used to mark the current file as being a system header. No warnings will be produced from the location of the pragma onwards within the same file.

```
char a = 'xy'; // warning

#pragma clang system_header

char b = 'ab'; // no warning
```

The options `-isystem-prefix` and `-ino-system-prefix` can be used to override whether subsets of an include path are treated as system headers. When the name in a `#include` directive is found within a header search path and starts with a system prefix, the header is treated as a system header. The last prefix on the command-line which matches the specified header name takes precedence. For example:

```
$ clang -Ifoo -isystem bar -isystem-prefix x/
    -ino-system-prefix x/y/
```

Here, `#include "x/a.h"` is treated as including a system header, even if the header is found in `foo`, and `#include "x/y/b.h"` is treated as not including a system header, even if the header is found in `bar`.

An `#include` directive which finds a file relative to the current directory is treated as including a system header if the including file is treated as a system header.

3.5.7 Enabling all warnings

In addition to the traditional `-W` flags, *all* warnings can be enabled by specifying the option `-Weverything`.

`-Weverything` works as expected with `-Werror`, and also includes the warnings from `-pedantic`.

NOTE When this option is used with `-w` (which disables all warnings), `-w` takes priority.

3.6 Using code optimizations

This section describes the following topics:

- Optimizing for performance
- Optimizing for code size
- Automatic vectorization
- Automatic parallelization
- Merging functions
- Link-time optimization

3.6.1 Optimizing for performance

LLVM currently generates the fastest code when compiling for ARM mode.

Table 3-3 lists the options to use for optimizing code performance.

Table 3-3 Optimizing for performance

Core	Options
ARMv7	`-Ofast -mcpu=krait2`
ARMv8 (AArch32)	`-Ofast -mtune=krait2`
ARMv8 (AArch64)	`-Ofast -mcpu=cortex-a53`

For more information on `-Ofast` see Section 3.4.19.

3.6.2 Optimizing for code size

LLVM currently generates the smallest code when compiling for Thumb2 mode.

Table 3-4 lists the options to use for optimizing code size.

Table 3-4 Optimizing for code size

Core	Options
ARMv7	`-Osize -mthumb`
ARMv8 (AArch32)	
ARMv8 (AArch64)	`-Os -mcpu=cortex-a53`

`-Osize` is preferred over `-Os` because it enables additional code-size optimizations.

For more information on `-Osize` see Section 3.4.19.

3.6.3 Automatic vectorization

LLVM includes support for automatic code vectorization. By default the vectorizer is disabled – to enable it use the `-fvectorize-loops` option (Section 3.4.17).

Vectorization can be used only with code optimization level `-O2` or higher (such as `-O3` or `-Ofast`).

Vectorization works only with the ARMv7 or ARMv8 processor architecture with the Neon extension. Neon is enabled either implicitly (by specifying a target processor such as Krait), or explicitly with `-mfpu=neon`.

The following is an example of a loop that can be vectorized with `-fvectorize-loops`:

```
void foo(int * restrict A, int N) {
  for (int i = 0; i < N; i++)
    A[i] = A[i] + 1;
}
```

To see which loops in a program get vectorized, use the following option:

```
-mllvm -print-vectorized-loops
```

For vectorization of floating point computation, the GCC option `-ffast-math` should be specified. Because floating point vectorizations (reductions in particular) are not IEEE compliant, the fast math option is required to ensure maximum vectorization of floating point computations.

NOTE The vectorizer can also be enabled using the GCC option `-ftree-vectorize`, which is an alias for `-fvectorize-loops`.

The GCC option `-ftree-vectorizer-verbose` is not supported for printing out verbose information on a vector. Instead, use `-mllvm -print-vectorized-loops`.

The vectorizer currently operates only on the innermost loop of a nested loop.

3.6.4 Automatic parallelization

The Qualcomm LLVM ARM compilers include support for automatic code parallelization. By default parallelization is disabled – to enable it use the `-fparallel` option (Section 3.4.18).

Parallelization can be used only with code optimization level `-O2`, `-O3`, or `-Ofast`.

Automatic code parallelization enables selected loops to be executed in parallel for faster performance. During parallelization, if a loop is determined to be free of any data, control, or memory dependencies, it is then split into multiple loops, each of which performs part of the work from the original loop. The resulting loops are dispatched to work queues on separate cores so they can be executed in parallel.

Parallelization requires a run-time component which is linked into the final executable image. The purpose of the component is to initialize a new thread at program initialization time, and subsequently manage the work queues during parallel execution.

While automatic code parallelization can significantly improve overall performance by distributing work across multiple cores, it accomplishes this by putting otherwise underutilized cores to use. Because it uses other cores, performance is a function of the entire system, and thus not fully determinable at compile time. Consequently it is possible for performance to improve, but also for net performance to be negatively impacted. Although the threads maintain the cores in a power-saving mode when they are not working, the additional work that is done in parallel can increase the overall power usage.

For this reason automatic code parallelization is not enabled by default in the compiler, and its use must be evaluated on a case-by-case basis.

NOTE By default, automatic code parallelization enables parallel execution of loops with reduction operations on integer types (including sum, product, logical and/or/xor, min, max, etc). `-fassociative-math` enables a subset of the supported integer reductions for floating point types (Section 3.4.21).

3.6.5 Merging functions

LLVM includes support for function merging. By default this optimization is disabled – to enable it use the `-fmerge-functions` option (Section 3.4.16).

Function merging attempts to improve code size by merging functions that are equivalent or differ in only a few instructions. The optimization uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

The following example shows how function merging works:

```
int f1(int a, int b) {
   int x;
   x = a + 4;
   return x * b;
}
```

```
int f2(int a, int b) {
   int x;
   x = a + 10;
   return x * b;
}
```

Function merging determines that functions `f1` and `f2` are similar, and replaces them with the following functions:

```
int f1__merged(int a, int b, int choice) {
   int x;
   if (choice)
      x = a + 10;
   else
      x = a + 4;
   return x * b;
}

int f1(int a, int b) {
   return f1__merged(a, b, 0);
}

int f2(int a, int b) {
   return f1__merged(a, b, 1);
}
```

This example is for illustration purposes only. In practice, the optimizer would determine that functions `f1` and `f2` are too small to be worth merging.

NOTE Because function merging may have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

3.6.6 Link-time optimization

Link-time optimization (LTO) comprises a set of powerful intermodular optimizations which are performed during the linking stage of compilation.

LTO expands the scope of optimizations from individual modules to the entire program (or at least to all the modules visible at link time). This enables deeper compiler analysis (such as better alias analysis) and more effective code transformations (such as function inlining), which can result in improved performance and code size.

When used with the `-c` option, `-flto` produces a file containing the LLVM compiler's intermediate representation (also known as *bitcode*). This file can be subsequently used in a final link step which then performs inter-module code optimizations on the file contents.

LTO comprises the following elements:

- The link-time optimizer, a compiler feature (controlled with the option `-flto`) which performs the intermodular optimizations while linking the files together.
- A set of compiler options which control the link-time optimizer (Section 3.4.22).
- The LTO-specific attribute `lto_preserve`, which when applied to a C or C++ function or variable prevents it from being discarded by the link-time optimizer.

Link-time optimizer

The link-time optimizer is invoked with the following command:

```
clang -flto input_files...
```

The optimizer inputs several LLVM bitcode files or archives. It then links the specified files together, performs the specified intermodular optimizations on them as a whole, and finally generates a single assembly file containing the optimized result.

An important optimization that the optimizer performs is the aggressive removal of functions it does not think are used. To provide the optimizer with a larger context for determining if a function is used, the list of filenames may include additional non-bitcode objects and archives. The optimizer will use the symbol information in these files to determine if a function should be preserved. You can also explicitly direct the optimizer to preserve a symbol by using the optimizer options `-lto-preserve` or `-lto-preserve-list`.

The optimizer has the following restrictions:

- Archives must be homogeneous: all members of a given archive must be either bitcode files or object files.
- The optimizer’s understanding of linker information is not as complete as a production-quality linker. While the optimizer attempts to eliminate the need to explicitly preserve symbols with the options `-lto-preserve` or `-lto-preserve-list`, it may not be able to do this in all cases. This functionality should hopefully improve over time, so if you encounter a situation that the optimizer does not handle properly, please report the problem to the developers.

3.7 Using GCC cross compile environments

The LLVM compilers are stand-alone compilers which rely on an existing system tools "root" environment – also known as *sysroot* – for accessing include files and libraries (as well as an ARM cross linker). The compilers are prebuilt to work with a GCC sysroot environment: to include header files and libraries in the build, they assume a predefined directory structure anchored by a GCC system root directory.

For example, the GCC sysroot for the ARMv8 AArch64 toolchain has the following structure:

```
/aarch64-linux-gnu/
   /bin/
   /debug-root/
/include/
   /include/c++/4.8.2/
      / arm-linux-gnueabihf/
/backward/
/lib/
/libc/
   /lib/
   /usr/
      /include/
```

The top level of the GCC tools directory must have a subdirectory that matches the target triple specified on the compiler command line (Section 3.4.15). The target triple directory typically contains a `libc` directory which mimics a host compilation environment by storing the following items:

- The library files in *`GCC-top`*`/`*`target-triple`*`/libc/lib`
- The include files in *`GCC-top`*`/`*`target-triple`*`/libc/usr/include`

Thus the sysroot location is *`GCC-top`*`/`*`target-triple`*`/libc`.

The sysroot location is specified with the compile option `--sysroot`.

The LLVM compilers additionally require the location of the GNU linker (and also an assembler, if not using the LLVM integrated assembler). This location is specified with the option `-B` or `-gcc-toolchain`, and must point to the top of the GCC toolchain directory.

For example, the following two LLVM commands compile a source file and generate code for the ARMv8 AArch64 ISA:

```
clang -target aarch64-linux-gnu --sysroot=GCC-top/aarch64-
                                    linux-gnu/libc -BGCC-top foo.c

clang -target aarch64-linux-gnu --sysroot=GCC-top/aarch64-linux-
                             gnu/libc --gcc-toolchain=GCC-top foo.c
```

With C++, it may be necessary to add a set of C++ include directories so the LLVM compilers can correctly search for the header files. Note that this is required only with certain GCC toolchain sysroots – in such cases the following directories should be added to the LLVM compiler command using the `-isystem` option:

```
-isystem GCC-top/include/c++/GCC-version
-isystem GCC-top/include/c++/GCC-version/triple
-isystem GCC-top/include/c++/GCC-version/backward
```

... where *GCC-version* indicates the version number of the GCC toolchain (4.6, 4.8.1, etc.).

NOTE The target triple specified above may differ from the target triple used on the compiler command line.

3.8 Built-in functions

```
__builtin_neon_memcpy_1024(void*, const void*, size_t)
```

The header file `arm_memcpy_bias.h` contains the declaration of a specialized ARM `memcpy` builtin for copy size of 1024.

Using this built-in function in the source will result in generated code with a runtime check for copy size of 1024 and the inlining of the `memcpy` specialized implementation for copy size 1024 using vector instructions.

NOTE Neon must be enabled to use this built-in.

3.9 Address Sanitizer

The LLVM compiler release includes a tool named *Address Sanitizer* (ASan) which can be used to detect memory errors in C and C++ code.

ASan is a runtime tool which requires compile-time instrumentation of the code, and a dedicated runtime library. If ASan encounters a bug during the execution of a program, it halts the execution and displays (on stderr) an error message and stack trace.

To use ASan you must instrument your C/C++ code and generate an Android executable.

To instrument your C/C++ code with ASan, add the following options to both the compile and link options in LLVM:

```
-g -fsanitize=address
```

Build your executable using the specified Asan options and run it on the Android/Linux environment.

If an error occurs during execution, the ASan runtime invokes the LLVM Symbolizer. The runtime locates the file `llvm-symbolizer` through the system PATH or the environment variable `ASAN_SYMBOLIZER_PATH`.

NOTE If the runtime cannot find the Symbolizer (or if you did not compile in debug mode (using `-g`), the ASan stack traces will contain raw addresses instead of line numbers.

To get better stack traces add `-fno-omit-frame-pointer`. To get perfect stack traces you may need to disable both inlining (use only `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

`asan_symbolize.py` is deprecated – use the LLVM Symbolizer instead.

3.9.1 Usage on Android

Generating an Android LLVM executable requires the following items:

- The Android NDK (for its linker)
- sysroot (for building the executable)

Once the executable is built, push your executable, the ASan runtime library, and the LLVM Symbolizer (Section 3.10) to an Android device. The ASan runtime library is a shared object which must be preloaded into the executable when launched.

The shared object can be found under the LLVM release tools installation directory:

```
export INSTALL_PREFIX=LLVM_release_tools_install_dir
file $INSTALL_PREFIX/lib/clang/*/lib/linux/libclang_rt.asan-arm-
                                                    android.so
```

Example

The following example shows how to detect a variable that is being referenced after it has been deleted:

```
$ cat boom.cc
int main(int argc, char **argv) {
  int *p = new int[10];
  delete [] p;
  return p[argc];
}
```

Build C/C++ executable with ASan instrumentation:

```
$ mkdir -p out
$ $INSTALL_PREFIX/bin/clang++ -target arm-linux-androideabi -g
-fsanitize=address boom.cc -o out/boom
--sysroot=Android_ARM_sysroot
--gcc-toolchain=Android_NDK_toolchain
```

Push executable, ASan runtime library, and symbolizer to Android device (Jellybean or later)

```
$ adb push out/boom /data/data/
$ adb push $INSTALL_PREFIX/lib/clang/*/lib/linux/libclang_rt.asan-
arm-android.so /data/data/
$ adb push $INSTALL_PREFIX/arm-linux-androideabi/llvm-symbolizer
/data/data/
```

Run ASan-instrumented executable:

```
$ adb shell "ASAN_SYMBOLIZER_PATH=/data/data/llvm-symbolizer
LD_PRELOAD=/data/data/libclang_rt.asan-arm-android.so
/data/data/boom"
ERROR: AddressSanitizer: heap-use-after-free
READ of size 4 at 0x60400000dfd4 thread T0
   #0 0x4b35e1 in main boom.cc:4
   #1 0x7f0db8e2376c in __libc_start_main libc-start.c:226
   #2 0x4b31ec in _start (boom+0x4b31ec)

0x60400000dfd4 is located 4 bytes inside of 40-byte region
[0x60400000dfd0,0x60400000dff8)
freed by thread T0 here:
   #0 0x41b551 in operator delete[](void*) asan_new_delete.cc:98
   #1 0x4b3510 in main boom.cc:3
   #2 0x7f0db8e2376c in __libc_start_main libc-start.c:226

previously allocated by thread T0 here:
   #0 0x41afc1 in operator new[](unsigned long)
asan_new_delete.cc:64
   #1 0x4b34bc in main boom.cc:2
   #2 0x7f0db8e2376c in __libc_start_main libc-start.c:226
```

3.9.2 Usage on Linux

Build C/C++ executable with ASAN instrumentation:

```
$INSTALL_PREFIX/bin/clang++ -target arm-linux-gnueabi
--sysroot=Linux_ARM_sysroot
--gcc-toolchain=Linux_ARM_toolchain
-g
-fsanitize=address boom.cc
-o boom
```

Run ARM ASan-instrumented executable:

You can run the executable on an ARM Linux system or a stand-alone user-mode QEMU emulator.

```
ASAN_SYMBOLIZE_PATH=$INSTALL_PREFIX/arm-linux-gnueabi/
  llvm-symbolizer
```

3.9.3 Options

When you run your instrumented executable and ASan does not detect any errors, you will see no output. Conversely, when you see no output, it can mean either that no errors occurred, or that your executable was not instrumented with the ASan runtime.

To verify that your executable is instrumented with ASan, use the environment variable `ASAN_OPTIONS` and the flag "`verbosity=1`". Doing this directs the ASan runtime to output a startup message when your executable is launched. For example (in Bash):

```
$ ASAN_OPTIONS=verbosity=1 ./myExe
```

If no output is generated while verbose mode is enabled, this implies your executable was not instrumented with ASan. Check that the option `-fsanitize=address` was passed to `clang` for *both* for the compilation step and the linking step.

ASan offers a variety of options for controlling the runtime behavior and enabling/disabling its functionality. For example, if you are running out of memory, set "`qualantine_size=0`". This causes ASan to miss any use-after-free errors but still detect buffer-overflow errors. Similarly, if you are overflowing the stack, set "`redzone=0`" to save stack space. In this case you will miss buffer-overflow errors, but can still detect use-after-free errors.

You can specify multiple options by separating flags with a colon. For example:

```
$ ASAN_OPTIONS=log_path=my-asan-report:redzone=8
```

Table 3-5 lists the options supported in ASan.

Table 3-5 ASan options

Option	Default	Description
`verbosity`	0	Be more verbose (mostly for testing the tool itself)
`malloc_context_size`	30	Number of frames in malloc/free stack traces (0-256).
`redzone`	16	Size of minimal redzone.
`log_path`	stderr	Path to log files. If specified as `log_path=PATH`, every process will write error reports to `PATH.PID`.
`sleep_before_dying`	0	Sleep for the specified number of seconds before exiting the process on failure.
`quarantine_size`	256Mb	Size of quarantine (in bytes) for finding use-after-free errors. Lower values save memory but increase false negatives rate.
`exitcode`	1	Call `_exit(exitcode)` on error.
`abort_on_error`	0	If set to 1, on error call `abort()` instead of `_exit(exitcode)`.
`strict_memcmp`	1	If set to 1 (default), treat `memcmp("foo", "bar", 100)` as a bug.
`alloc_dealloc_mismatch`	1	If set to 1, check for mismatches between `malloc()/new/new` and `free()/delete/delete`.
`handle_segv`	1	If set to 1, ASan installs its own handler for `SIGSEGV`.
`allow_user_segv_handler`	0	If set to 1, allows user to override `SIGSEGV` handler installed by ASan.
`check_initialization_order`	0	If set to 1, detect existing initialization order problems.
`strip_path_prefix`	""	If `strip_path_prefix=PREFIX`, remove the substring `.*PREFIX` from the reported file names.

3.9.4 Notes

The ASan runtime library does not yet demangle symbols, but the LLVM symbolizer can be used to demangle symbols (Section 3.10).

The ASan runtime library cannot be statically linked on Android. The linker does not load the `libc` symbols before any others, as it does on Linux systems. ASan relies on this feature to hijack symbols before any other shared objects are loaded. Therefore, on Android it is necessary to use the `LD_PRELOAD` trick.

For more information on ASan see clang.llvm.org/docs/AddressSanitizer.html.

3.10 LLVM Symbolizer

The LLVM compiler release includes a tool named *LLVM Symbolizer*, which can be used to convert program addresses into source code locations.

Symbolizer is a command-line tool – it reads object file names and addresses from the standard input, and writes the corresponding source code locations to the standard output.

If an object file name is directly specified as a command-line argument, Symbolizer treats it as the name of the input object file, and reads only addresses from the standard input.

NOTE To perform its conversion, Symbolizer uses the symbol tables and debug info sections that are stored in the object files.

To start Symbolizer from a command line, type:

`llvm-symbolizer` *`options...`*

Command switches are used to control the symbolizer options (Section 3.10.2).

NOTE The Symbolizer normally returns 0 as a program return code. Any other code values indicate that an internal program error.

3.10.1 Usage

```
$ cat addr.txt
a.out 0x4004f4
/tmp/b.out 0x400528
/tmp/c.so 0x710
/tmp/mach_universal_binary:i386 0x1f84
/tmp/mach_universal_binary:x86_64 0x100000f24
$ llvm-symbolizer < addr.txt
main
/tmp/a.cc:4

f(int, int)
/tmp/b.cc:11

h_inlined_into_g
/tmp/header.h:2
g_inlined_into_f
/tmp/header.h:7
f_inlined_into_main
/tmp/source.cc:3
main
/tmp/source.cc:8

_main
/tmp/source_i386.cc:8
```

```
_main
/tmp/source_x86_64.cc:8
$ cat addr2.txt
0x4004f4
0x401000
$ llvm-symbolizer -obj=a.out < addr2.txt
main
/tmp/a.cc:4

foo(int)
/tmp/a.cc:12
```

3.10.2 Options

Symbolizer is controlled by command-line options.

Table 3-5 lists the options supported in Symbolizer.

Table 3-6 Symbolizer options

Option	Description
`-obj`	Path to object file to be symbolized.
`-functions=(none|short|linkage)`	Specify how function names are printed. `none` – Omit function name `short` – Print short function name `linkage` – Print full linkage name The default is `linkage`.
`-use-symbol-table`	Favor function names stored in the symbol table over function names in debug info sections. The default is enabled.
`-demangle`	Print demangled function names. The default is enabled.
`-inlining`	If a source code location is in an inlined function, prints all the inlined frames. The default is enabled.
`-default-arch` *`arch_name`*	If a binary contains object files for multiple architectures (e.g., it is a Mach-O universal binary), symbolize the object file for the specified architecture. The architecture name is specified as a string value. The default is an empty string. The architecture can alternatively be specified by passing the string "*`binary_name`*`:`*`arch_name`*" as part of the input (Section 3.10.1). NOTE - If an architecture is not specified in either way, addresses will not be symbolized.

3.10.3 Notes

Symbolizer is used with the ASan tool (Section 3.9).

3.11 Secure programming support

To support secure programming, the following tools are provided with the LLVM compiler:

- Static analyzer
- Post processor
- Scan-build

The static analyzer is a source code analysis tool which finds potential bugs in C and C++ programs. It can be used to analyze individual files or entire programs.

The post processor creates a summary of the report that is generated by performing static analysis while compiling a program.

Scan-build is an additional tool for compiling and statically analyzing a program. It can be used with a Make-based build system.

3.11.1 Static analyzer

The static analyzer is a source code analysis tool which is integrated into the LLVM compiler. It analyzes a program for various types of potential bugs including security threats, memory corruption, and garbage values.

The static analyzer has the following features:

- It supports more than 100 distinct checkers which are organized into the categories `alpha`, `core`, `cplusplus`, `debug`, and `security`.
- Checkers can be selectively enabled or disabled from the command line (with a complete list of checkers provided by the option `-analyzer-checker-help`).
- Disabling a checker category disables all the checkers in that category.

Analyzing source files

To use the static analyzer on specific program source files, invoke the LLVM compiler on the files using the options `--analyze`, `-analyzer-output`, and `-o` (Section 3.4.23).

For example:

```
clang --analyze --analyzer-output html test.cpp -o test-dir
```

If the static analyzer detects any potential bugs, it outputs a diagnostic report which is stored in the directory specified by `-o`. If the specified directory does not exist, the compiler automatically creates it.

Example of a diagnostic report entry:

```
// @file: test.cpp
int main() {
  int* p = new int();
  return* p;
}
warning: Potential leak of memory pointed to by 'p'
```

Each potential bug flagged in a report includes the path (i.e., control and data) needed for locating the bug in the program.

Analyzing programs

To use the static analyzer on an entire program, invoke the LLVM compiler on the files using the option `--compile-and-analyze` (Section 3.4.23).

If the static analyzer detects any potential bugs, it outputs a diagnostic report which is stored in the directory specified by `--compile-and-analyze`. If the specified directory does not exist, the compiler automatically creates it.

When using a build system, specifying the same directory name throughout the build will generate all the HTML report files in the specified directory.

NOTE The filenames generated for the static analysis report use hashing functions, so the report files will not be overwritten.

Statically analyzing an entire program at once (as opposed to selected source files) is recommended for the following reasons:

- The generated analysis report files are all stored in a single location.
- The command option can be passed from the build system, which helps perform the static analysis and compilation every time the program is built.
- Because build systems are good at tracking files that have changed, and compiling only the minimal set of required files, the overall turnaround time for static analysis is relatively small, making it reasonable to run the static analyzer with every build.

Managing checkers

The static analyzer may report false positives. Several factors can contribute to the generation of false positives. For example, if the checker used to analyze dead code is enabled, then the static analyzer will flag code that has been conditionally enabled for debugging purposes.

For this reason the static analyzer by default enables a set of checkers which has been tested to identify a very high percentage of actual program bugs. If necessary, additional checkers can be individually enabled with the compiler option `-analyzer-checker` (Section 3.4.23).

NOTE `-analyzer-checker-help` lists the supported checkers. This option should be used with `-cc1`. For example: `clang -cc1 -analyzer-checker-help`

Another factor that result in false positives is non-returning functions such as assert functions. Although the static analyzer is aware of the standard library non-returning functions, if a program has its own implementation of asserts (for example), it helps to mark them with the following function attribute:

```
__attribute__((__noreturn__))
```

Using this attribute greatly improves the static analysis diagnostics and lessens the number of false positives. For example:

```
void my_abort(const char* msg) __attribute__((__noreturn__)) {
printf("%s", msg);
exit(1);
}
```

3.11.2 Post processor

The post processor is a report generator which is implemented as a stand-alone script. It creates a summary of the report that is generated by using the option `-compile-and-analyze` (Section 3.11.1).

The post processor is invoked with the following command:

```
post-process --report-dir dir --html-title MyReport
```

The post processor reads all the files from the directory specified by the option `--report-dir`, and writes in the same directory a summary report file named `index.html`.

The report title (`MyReport` in this case) is specified with the option `--html-title`.

For more information invoke `post-process --help`.

The post processor script is stored in the directory `$INSTALL_PREFIX/lib/static-analyzer`.

NOTE In some cases the static analyzer may generate multiple report files for the same bug. The post processor cleans up after multiple report files. For this reason it should be run regularly to keep the report directory clean.

3.11.3 Scan-build

`Scan-build` is a stand-alone tool for compiling and statically analyzing a program. It can be used with a Make-based build system (though it is recommended to instead use the LLVM static analyzer whenever possible – see Section 3.11.1).

`Scan-build` enables a user to run the static analyzer as part of regular build process. Here are two examples of invoking `scan-build`:

```
scan-build clang++ -c test.cpp

scan-build -v -k -o out-dir -disable-checker deadcode \
  -use-c++=clang++ --use-c=clang make -j8
```

`Scan-build` works well with a Make-based build system.

For more information invoke `scan-build --help`.

The `Scan-build` script is stored in the directory `$INSTALL_PREFIX/lib/static-analyzer`.

NOTE `Scan-build` is intended for use only on Linux systems.

4 Porting Code from GCC

4.1 Overview

This chapter describes issues commonly encountered while porting to LLVM an application that was previously built only with GCC.

It covers the following topics:

- Command options
- Errors and warnings
- Function declarations
- Casting to incompatible types
- Array sizes
- `aligned` attribute
- Reserved registers
- Inline versus extern inline

NOTE For more information on GCC compatibility see Chapter 6.

4.2 Command options

LLVM supports many but not all of the GCC command options. Unsupported options are either ignored or flagged with a warning or error message: most receive warning messages.

For more information see Section 3.4.5.

4.3 Errors and warnings

LLVM enforces strict conformance to the C99 language standard. As a result, you may encounter new errors and warnings when compiling GCC code.

To handle these messages when porting to LLVM, consider the following steps:

1. Remove the command option `-Werror` if it is being used (as it converts all warnings into errors).
2. Update the code to eliminate the remaining errors and warnings.

4.4 Function declarations

LLVM enforces the C99 rules for function declarations. In particular:

- A function declared with a non-`void` return type must return a value of that type.
- A function referenced before being declared is assumed to return a value of type `int`. If the function is subsequently declared to return some other type, it will be flagged with an error.
- A function declaration with the `inline` attribute assumes the existence of a separate definition for the function, which does not include the `inline` attribute. If no such definition appears in the program, a link-time error will occur.

To satisfy these restrictions when porting to LLVM, consider the following steps:

1. Use option `-Wreturn-type` to generate a warning whenever a function definition does not return a value of its declared type.
2. Use `-Wimplicit-function-declaration` to generate a warning whenever a function is used before being declared.
3. Update the code to eliminate the remaining errors and warnings.

For more information on inlining see http://clang.llvm.org/compatibility.html#inline.

A discussion of different inlining approaches can be found at http://www.greenend.org.uk/rjk/tech/inline.html.

4.5 Casting to incompatible types

LLVM enforces the C99 rules for *strict aliasing*.

In the C language, two pointers that reference the same memory location are said to *alias* one another. Because any store through an aliased pointer can potentially modify the data referenced by one of its pointer aliases, pointer aliases can limit the compiler's ability to generate optimized code.

In strict aliasing, pointers to different types are prevented from being aliased with one another. The compiler flags pointer aliases with an error message.

Note that strict aliasing has a few exceptions:

- Any pointer type can be cast to `char*` or `void*`.
- A `char*` or `void*` can be cast to any pointer type.
- Pointers to types that differ only by signedness (e.g., `int` versus `unsigned int`) can be aliased.

To satisfy strict aliasing when porting to LLVM, consider the following steps:

1. Use option `-Wcast-align` to generate a warning whenever a pointer alias is detected.
2. Update the code to eliminate the resulting warnings.

4.6 Array sizes

LLVM requires arrays to be defined with array sizes that are compile-time constants.

To satisfy this restriction when porting to LLVM, change the code so arrays are allocated with their largest possible array size. For example:

```
int foo ()
{
#ifndef __llvm__
    char ctx[my_size_func()];
#else
    char ctx[4];
#endif
    …
}
```

Alternatively, allocate the arrays dynamically on the heap:

```
char *ctx = (char *) alloca (sizeof(char) * my_size_func());
```

4.7 aligned attribute

LLVM does not allow the `aligned` attribute to appear inside the `__alignof__` operator.

To satisfy this restriction when porting to LLVM, create a typedef with the `aligned` attribute. For example:

```
typedef unsigned char u8;
#ifdef __llvm__
    typedef u8 __attribute((aligned)) aligned_u8;
#endif
unsigned int foo()
{
#ifndef __llvm__
    return __alignof__(u8 __attribute__ ((aligned)));
#else
    return __alignof__(aligned_u8);
#endif
}
```

4.8 Reserved registers

LLVM does not support the GCC extension to place global variables in specific registers.

To satisfy this restriction when porting to LLVM, use the equivalent LLVM intrinsics whenever possible. For example:

```
#ifndef __llvm__
    register unsigned long current_frame_pointer asm("r11");
#endif
…
#ifndef __llvm__
    fp = current_frame_pointer;
#else
    fp = (unsigned long)__builtin_frame_address(0);
#endif
```

4.9 Inline versus extern inline

LLVM conforms to the C99 language standard, which defines different semantics for the `inline` keyword than GCC. For example, consider the following code:

```
inline int add(int i, int j) { return i + j; }

int main() {
  int i = add(4, 5);
  return i;
}
```

In C99 the function attribute `inline` specifies that a function's definition is provided only for inlining, and that another definition (without the `inline` attribute) is specified elsewhere in the program.

This implies that the above example is incomplete, because if `add()` is not inlined (for example, when compiling without optimization), then `main()` will include an unresolved reference to that other function definition. This will result in the following link-time error:

```
Undefined symbols:
  "_add", referenced from:   _main in cc-y1jXIr.o
```

By contrast, GCC's default behavior follows the GNU89 dialect, which is based on the C89 language standard. C89 does not support the `inline` keyword; however, GCC recognizes it as a language extension, and treats it as a hint to the optimizer.

There are several ways to fix this problem:

- Change `add()` 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 it will not be necessary to add a non-inline definition of the function elsewhere in the program.
- Remove the `inline` keyword from this definition of `add()`. The `inline` keyword is not required for a function to be inlined, nor does it guarantee that it will be. Some compilers ignore it completely. LLVM treats it as a mild suggestion from the programmer.
- Provide an external (non-inline) definition of `add()` somewhere else in the program. Note that the two definitions *must* be equivalent.
- Compile with the GNU89 dialect by adding `-std=gnu89` to the set of LLVM options. This approach is not recommended.

5 Coding Practices

5.1 Overview

This chapter describes recommended coding practices for users of the LLVM compilers. These practices typically result in the compiler generating more optimized code.

This chapter covers the following topics:

- Use `int` types for loop counters
- Mark function arguments as `restrict` (if possible)
- Do not pass or return structures by value
- Avoid using inline assembly

5.2 Use int types for loop counters

Using an `int` type for loop counters is strongly recommended and results in the compiler generating more efficient code. If the code uses a non-`int` type, then the compiler will have to insert zero and sign-extensions to abide by C rules. For example, the following code is *not* recommended:

```
extern int A[30], B[30];
for (short int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

Use this code instead:

```
extern int A[30], B[30];
for (int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

5.3 Mark function arguments as restrict (if possible)

LLVM supports the `restrict` keyword for function arguments. Using `restrict` on a pointer passed in as a function argument indicates to the compiler that the pointer will be used exclusively to dereference the address it points at. This allows the compiler to enable more aggressive optimizations on memory accesses.

NOTE When using the `restrict` keyword, you must ensure that the restrict condition holds for all calls made to that function. If an argument is erroneously marked as `restrict`, the compiler may generate incorrect code.

5.4 Do not pass or return structs by value

It is strongly recommended that structs get passed to (and returned from) functions by reference and not by value.

If a struct is passed to a function by value, the compiler must generate code which makes a copy of the struct during application runtime. This can be extremely inefficient, and will reduce the performance of the compiled code. For this reason, it is recommended that structs be passed by pointer.

For instance, the following code is inefficient:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct S arg1) {
   …
}

int baz() {
  struct S;
   … populate elements of S …
   bar(S);
}
```

While this code is much more efficient:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct *S arg1) {
  /* Access z here using ‘arg1->z’ (instead of ‘arg1.z’) */
  …
}

int baz() {
  struct S;
   … populate elements of S …
   bar(&S);
}
```

Alternatively, in C++, the efficient code can be simplified by using reference parameters:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct &S arg1) {
   …
}

int baz() {
  struct S;
   … populate elements of S …
   bar(S);
}
```

5.5 Avoid using inline assembly

Using inline assembly snippets in C files is strongly discouraged for two reasons:

- Inline assembly snippets are extremely difficult to write correctly. For instance, omitting the input, output, or clobber parameters frequently leads to incorrect code. The resulting failure can be extremely difficult to debug.
- Inline assembly is not portable across processor versions. If you need to emit a specific assembly instruction, it is recommended to use a compiler intrinsic instead of inline assembly.

Intrinsics are easy to insert in a C file, and are portable across processor versions. If intrinsics are insufficient, then you should add a new function written in assembly which contains the desired functionality. The assembly function should be called from C code.

6 Language Compatibility

LLVM strives to both conform to current language standards (C99, C++98) and also to implement many widely-used extensions available in other compilers, so that most correct code will "just work" when compiler with LLVM. However, LLVM is more strict than other popular compilers, and may reject incorrect code that other compilers allow.

This chapter describes common compatibility and portability issues with LLVM to help you understand and fix the problem in your code when LLVM emits an error message.

It covers the following topics:

- C compatibility
- C++ compatibility

6.1 C compatibility

6.1.1 Differences between various standard modes

LLVM supports the `-std` option, which changes what language mode LLVM uses. The supported modes for C are c89, gnu89, c94, c99, gnu99 and various aliases for those modes. If no `-std` option is specified, LLVM defaults to gnu99 mode.

The c* and gnu* modes have the following differences:

- c* modes define `__STRICT_ANSI__`.
- Target-specific defines not prefixed by underscores (such as "`linux`") are defined in gnu* modes.
- Trigraphs default to being off in gnu* modes; they can be enabled by the `-trigraphs` option.
- The parser recognizes `asm` and `typeof` as keywords in gnu* modes; the variants `__asm__` and `__typeof__` are recognized in all modes.
- Arrays that are VLA's according to the standard, but which can be constant folded by the compiler front end are treated as fixed size arrays. This occurs for things such as "`int X[(1, 2)];`", which is technically a VLA. c* modes are strictly compliant and treat these as VLAs.

The *89 and *99 modes have the following differences:

- The *99 modes default to implementing `inline` as specified in C99, while the *89 modes implement the GNU version. This can be overridden for individual functions with the `__gnu_inline__` attribute.
- Digraphs are not recognized in c89 mode.
- The scope of names defined in a `for`, `if`, `switch`, `while`, or `do` statement is different. (example: "`if ((struct x {int x;}*)0) {}`".)
- `__STDC_VERSION__` is not defined in *89 modes.
- `inline` is not recognized as a keyword in c89 mode.
- `restrict` is not recognized as a keyword in *89 modes.
- Commas are allowed in integer constant expressions in *99 modes.
- Arrays which are not lvalues are not implicitly promoted to pointers in *89 modes.
- Some warnings are different.

c94 mode is identical to c89 mode except that digraphs are enabled in c94 mode.

6.1.2 GCC extensions not implemented yet

LLVM tries to be compatible with GCC as much as possible, but the following GCC extensions are not yet implemented in LLVM:

- `#pragma weak` – This is likely to be implemented at some point in the future, at least partially.
- **Decimal floating (`_Decimal32`, etc.) and fixed-point types (`_Fract`, etc.)** – No one has expressed interest in these yet, so it is currently unclear when they will be implemented.
- **Nested functions** – This is a complex feature which is infrequently used, so it is unlikely to be implemented anytime soon.
- **Global register variables** – This is unlikely to be implemented soon as it requires additional LLVM back end support.
- **Static initialization of flexible array members** – This appears to be a rarely used extension, but could be implemented pending user demand.
- `__builtin_va_arg_pack and __builtin_va_arg_pack_len` – This is used rarely, but in some potentially interesting places such as the `glibc` headers, so it may be implemented pending user demand. Note that because LLVM pretends to be like GCC 4.2, and this extension was introduced in 4.3, the `glibc` headers will currently not try to use this extension with LLVM.
- **Forward-declaring function parameters** – This has not showed up in any real-world code yet, though, so it might never be implemented.

6.1.3 Intentionally unsupported GCC extensions

LLVM intentionally does not implement the following GCC extensions:

- **Variable-length arrays in structures** – This is not implemented for several reasons: it is tricky to implement, the extension is completely undocumented, and the extension appears to be rarely used. Note that LLVM *does* support flexible array members (arrays with a zero or unspecified size at the end of a structure).
- **An equivalent to GCC's "fold"** – This implies that LLVM does not accept some constructs GCC might accept in contexts where a constant expression is required, such as "`x-x`" where `x` is a variable.
- `__builtin_apply` **and related attributes** – This extension is extremely obscure and difficult to implement reliably.

6.1.4 Microsoft extensions

- LLVM has some experimental support for extensions from Microsoft Visual C++; to enable it, use the option `-fms-extensions command-line`. This is the default for Windows targets. Note that the support is incomplete: enabling the Microsoft extensions will silently drop certain constructs (including `__declspec` and Microsoft-style `asm` statements).
- `-fms-compatibility` causes the compiler to accept enough invalid C++ to be able to parse most Microsoft headers. This flag is enabled by default for Windows targets.
- `-fdelayed-template-parsing` lets LLVM delay all template instantiation until the end of a translation unit. This flag is enabled by default for Windows targets.
- LLVM allows setting `_MSC_VER` with `-fmsc-version=n`. This option defaults to the value 1300, which is the same as in Visual C/C++ 2003. Any value is supported, and the value chosen can greatly affect what Windows SDK and `c++stdlib` headers LLVM can compile. This option will be removed when LLVM supports the full set of MS extensions required for these headers.
- LLVM does not support the Microsoft extension where anonymous record members can be declared using user defined typedefs.
- LLVM supports the Microsoft `pack` pragma for controlling record layout. GCC also contains support for this feature. However, in cases where MSVC and GCC are incompatible, LLVM follows the MSVC definition.
- LLVM defaults to C++11 for Windows targets..

6.1.5 Lvalue casts

Old versions of GCC permit casting the left-hand side of an assignment to a different type. LLVM produces an error for code like this:

```
lvalue.c:2:3: error: assignment to cast is illegal, lvalue casts
are not supported
(int*)addr = val;
^~~~~~~~~~ ~
```

To fix this problem, move the cast to the right-hand side. In this example, one could use:

```
 addr = (float *)val;
```

6.1.6 Inline assembly

In general, LLVM is highly compatible with the GCC inline assembly extensions, allowing the same set of constraints, modifiers and operands as GCC inline assembly.

6.2 C++ compatibility

6.2.1 Variable-length arrays

GCC and C99 allow an array's size to be determined at run time. This extension is not permitted in standard C++. However, LLVM supports such variable length arrays in very limited circumstances for compatibility with GNU C and C99 programs:

- The element type of a variable length array must be a "plain old data" (POD) type, which means that it cannot have any user-declared constructors or destructors, any base classes, or any members of non-POD type. All C types are POD types.
- Variable length arrays cannot be used as the type of a non-type template parameter.

If your code uses variable length arrays in a manner that LLVM does not support, several ways are available to fix your code:

1. Replace the variable length array with a fixed-size array if you can determine a reasonable upper bound at compile time; sometimes this is as simple as changing `int size = ...;` to `const int size = ...;` (if the initializer is a compile-time constant);
2. Use `std::vector` or some other suitable container type; or
3. Allocate the array on the heap instead using `new Type[]` – just remember to `delete[]` it.

6.2.2 Unqualified lookup in templates

Some versions of GCC accept the following invalid code:

```
template <typename T> T Squared(T x) {
  return Multiply(x, x);
}

int Multiply(int x, int y) {
  return x * y;
}

int main() {
  Squared(5);
}
```

LLVM flags this code with the following messages:

```
my_file.cpp:2:10: error: call to function 'Multiply' that is
neither visible in the template definition nor found by argument-
dependent lookup

   return Multiply(x, x);
          ^


my_file.cpp:10:3: note: in instantiation of function template
specialization 'Squared<int>' requested here

  Squared(5);
  ^


my_file.cpp:5:5: note: 'Multiply' should be declared prior to the
call site

  int Multiply(int x, int y) {
      ^
```

The C++ standard states that unqualified names such as "`Multiply`" are looked up in two ways:

- First, the compiler performs an *unqualified lookup* in the scope where the name was written. For a template, this means the lookup is done at the point where the template is defined, not where it's instantiated. Because `Multiply` has not been declared yet at this point, unqualified lookup will not find it.
- Second, if the name is called like a function, then the compiler also does *argument-dependent lookup* (ADL). In ADL the compiler looks at the types of all the arguments to the call. When it finds a class type, it looks up the name in that class's namespace; the result is all the declarations it finds in those namespaces, plus the declarations from unqualified lookup. However, the compiler does not do ADL until it knows all the argument types.

In the example code above, `Multiply` is called with dependent arguments, so ADL isn't done until the template is instantiated. At that point the arguments both have type `int`, which does not contain any class types, and so ADL does not look in any namespaces. Since neither form of lookup found the declaration of `Multiply`, the code does not compile.

Here's another example, this time using overloaded operators, which obey very similar rules.

```
#include <iostream>

template<typename T>

void Dump(const T& value) {
  std::cout << value << "\n";
}

namespace ns {
  struct Data {};
}

std::ostream& operator<<(std::ostream& out, ns::Data data) {
  return out << "Some data";
}

void Use() {
  Dump(ns::Data());
}
```

Again, LLVM flags this code with the following messages:

```
my_file2.cpp:5:13: error: call to function 'operator<<' that is
neither visible in the template definition nor found by argument-
dependent lookup

  std::cout << value << "\n";
            ^


my_file2.cpp:17:3: note: in instantiation of function template
specialization 'Dump<ns::Data>' requested here

  Dump(ns::Data());
  ^


my_file2.cpp:12:15: note: 'operator<<' should be declared prior to
the call site or in namespace 'ns'

  std::ostream& operator<<(std::ostream& out, ns::Data data) {
                ^
```

Just as before, unqualified lookup did not find any declarations with the name `operator<<`. Unlike before, the argument types both contain class types:

- One of them is an instance of the class template type `std::basic_ostream`
- The other is the type `ns::Data` that is declared in the example above

Therefore, ADL will look in the namespaces `std` and `ns` for an `operator<<`. Because one of the argument types was still dependent during the template definition, ADL is not done until the template is instantiated during `Use`, which means that the `operator<<` it should find has already been declared. Unfortunately, it was declared in the global namespace, not in either of the namespaces that ADL will look in!

Two ways exist to fix this problem:

1. Make sure the function you want to call is declared before the template that might call it. This is the only option if none of its argument types contain classes. You can do this either by moving the template definition, or by moving the function definition, or by adding a forward declaration of the function before the template.
2. Move the function into the same namespace as one of its arguments so that ADL applies.

6.2.3 Unqualified lookup into dependent bases of class templates

Some versions of GCC accept the following invalid code:

```
template <typename T> struct Base {
  void DoThis(T x) {}
  static void DoThat(T x) {}
};

template <typename T> struct Derived : public Base<T> {
  void Work(T x) {
    DoThis(x); // Invalid!
    DoThat(x); // Invalid!
  }
};
```

LLVM correctly rejects this code with the following errors (when `Derived` is eventually instantiated):

```
my_file.cpp:8:5: error: use of undeclared identifier 'DoThis'

  DoThis(x);
  ^
  this->

my_file.cpp:2:8: note: must qualify identifier to find this
declaration in dependent base class

void DoThis(T x) {}
     ^


my_file.cpp:9:5: error: use of undeclared identifier 'DoThat'

DoThat(x);
^
this->

my_file.cpp:3:15: note: must qualify identifier to find this
declaration in dependent base class

static void DoThat(T x) {}
```

As noted in Section 6.2.2, unqualified names such as `DoThis` and `DoThat` are looked up when the template `Derived` is defined, not when it's instantiated. When looking up a name used in a class, we usually look into the base classes. However, we can't look into the base class `Base<T>` because its type depends on the template argument `T`, so the standard says we should just ignore it.

The fix, as LLVM indicates, is to tell the compiler that we want a class member by prefixing the calls with `this->`:

```
void Work(T x) {
 this->DoThis(x);
 this->DoThat(x);
}
```

Alternatively, you can tell the compiler exactly where to look:

```
void Work(T x) {
  Base<T>::DoThis(x);
  Base<T>::DoThat(x);
}
```

This works whether the methods are static or not, but be careful: if `DoThis` is virtual, calling it this way will bypass virtual dispatch!

6.2.4 Incomplete types in templates

The following code is invalid, but compilers are allowed to accept it:

```
class IOOptions;

template <class T> bool read(T &value) {
  IOOptions opts;
  return read(opts, value);
}

class IOOptions { bool ForceReads; };
bool read(const IOOptions &opts, int &x);
template bool read<>(int &);
```

The standard says that types which don't depend on template parameters must be complete when a template is defined if they affect the program's behavior. However, the standard also says that compilers are free to not enforce this rule. Most compilers enforce it to some extent; for example, it would be an error in GCC to write `opts.ForceReads` in the code above. In LLVM, the decision to enforce the rule consistently provides a better experience, but unfortunately it also results in some code getting rejected that other compilers accept.

6.2.5 Templates with no valid instantiations

The following code contains a typo: the programmer meant `init()` but wrote `innit()` instead.

```
template <class T> class Processor {
  ...
  void init();
  ...
};

...

template <class T> void process() {
  Processor<T> processor;
  processor.innit();      // <-- should be 'init()'
  ...
}
```

Unfortunately, the compiler can't flag this mistake as soon as it detects it: inside a template, we're not allowed to make assumptions about "dependent types" such as `Processor<T>`. Suppose that later on in this file the programmer adds an explicit specialization of `Processor`, like so:

```
template <> class Processor<char*> {
  void innit();
};
```

Now the program will work – but only if the programmer ever instantiates `process()` with `T = char*`! This is why it's hard, and sometimes impossible, to diagnose mistakes in a template definition before it's instantiated.

The standard states that a template with no valid instantiations is ill-formed. LLVM tries to do as much checking as possible at definition-time instead of instantiation-time: not only does this produce clearer diagnostics, but it also substantially improves compile times when using pre-compiled headers. The downside to this philosophy is that LLVM sometimes fails to process files because they contain broken templates that are no longer used. The solution is simple: since the code is unused, just remove it.

6.2.6 Default initialization of const variable of a class type

The default initialization of a `const` variable of a class type requires a user-defined default constructor.

If a `class` or `struct` has no user-defined default constructor, C++ does not allow you to default-construct a `const` instance of it. For example:

```
class Foo {
public:
  // The compiler-supplied default constructor works fine, so we
  // don't bother with defining one.
  ...
}
void Bar() {
  const Foo foo; // Error!
  ...
}
```

To fix this, you can define a default constructor for the class:

```
class Foo {
public:
  Foo() {}
  ...
};

void Bar() {
  const Foo foo; // Now the compiler is happy.
  ...
}
```

6.2.7 Parameter name lookup

Due to a bug in its implementation, GCC allows the redeclaration of function parameter names within a function prototype in C++ code, e.g.

```
void f(int a, int a);
```

LLVM diagnoses this error (where the parameter name has been redeclared). To fix this problem, rename one of the parameters.

A Acknowledgements

A.1 Overview

We would like to thank the LLVM community for their many contributions to the LLVM Project.

This document was derived from the LLVM Project documentation under the terms of the LLVM Release License.

This chapter presents the following license statements:

- LLVM Release License
- Copyrights and Licenses for Third Party Software Distributed with LLVM
- Block Implementation Specification

A.2 LLVM Release License

Portions of this Snapdragon LLVM Compiler are subject to the below list of conditions and disclaimers:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.
- Neither the names of the LLVM Team, University of Illinois at Urbana-Champaign, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

A.3 Copyrights and Licenses for Third Party Software Distributed with LLVM

The LLVM software contains code written by third parties. Such software will have its own individual `LICENSE.TXT` file in the directory in which it appears. This file will describe the copyrights, license, and restrictions which apply to that code.

The disclaimer of warranty in the University of Illinois Open Source License applies to all code in the LLVM Distribution, and nothing in any of the other licenses gives permission to use the names of the LLVM Team or the University of Illinois to endorse or promote products derived from this Software.

The following pieces of software have additional or alternate copyrights, licenses, and/or restrictions:

Program	Directory
Autoconf	`llvm/autoconf`
	`llvm/projects/ModuleMaker/autoconf`
	`llvm/projects/sample/autoconf`
CellSPU backend	`llvm/lib/Target/CellSPU/README.txt`
Google Test	`llvm/utils/unittest/googletest`
OpenBSD regex	`llvm/lib/Support/{reg*, COPYRIGHT.regex}`
pyyaml tests	`llvm/test/YAMLParser/{*.data, LICENSE.TXT}`

A.4 Block Implementation Specification

Block Implementation Specification