

Introduction to GCC

Brian Gough Shakthi Kannan

Version 1.1 GNU FDL

- Introduction
- Static and shared libraries
- Compilation options
- C Language Standards
- Warning options
- Using the preprocessor
- Compiling for debugging

Sections (...)

- Compiling with optimization
- Compiling a C++ program
- Platform specific options
- Compiler related tools
- How the compiler works
- Examining compiled files
- Common error messages

Introduction

- GNU Compiler Collection
- A portable compiler
- Output for many types of processors
- Not only a native compiler
- Multiple language frontends
- Modular design
- Support to add new architectures
- GCC is Free Software

- Allow direct access to the computer's memory
- Useful for writing
 - Low-level systems software
 - High performance
 - Control over resource usage are critical
- Care is required to ensure that memory is accessed correctly
- Avoid corrupting data structures

Compiling a C program

```
/* main.c */  
#include <stdio.h>  
  
int  
main (void)  
{  
    printf ("Hello, world!\n");  
    return 0;  
}
```

Compiling a C program (...)

```
$ gcc main.c -o main
```

- To run the program, type the path name of the executable

```
$ ./main
```

```
Hello, world!
```

Compiling a C program (...)

- Compile from single source file or from multiple files
- May use system libraries and header files
- If `-o` option is omitted, the output is written to default file 'a.out'
- GCC message format file

`file:line-number:message`

Compiling a C program (...)

- To search for 'FILE.h' in current directory

```
#include "FILE.h"
```

- To search for 'FILE.h' in system header file directories

```
#include <FILE.h>
```

- With independent source files, only recompile modified files

Compiling a C program (...)

- First stage: compile an “object file” (.o) without an executable

```
/* hello.c */  
#include <stdio.h>  
#include "hello.h"  
  
void  
hello (const char *name)  
{  
    printf ("Hello, %s!\n", name);  
}
```

Compiling a C program (...)

```
/* hello.h */
```

```
void hello (const char *name);
```

```
$ gcc -Wall -c hello.c
```

Compiling a C program (...)

- Second stage: object files linked to create an executable

```
/* main.c */
#include "hello.h"

int
main (void)
{
    hello ("Everyone");
    return 0;
}
```

Compiling a C program (...)

```
$ gcc -Wall -c main.c
```

- Linker (ld) combines all the object files together

```
$ gcc main.o hello.o -o hello
```

```
$ ./hello
```

```
Hello, everyone!
```

Compiling a C program (...)

- GNU Make to automate recompilation of modified files in a project
- Implicit rules are defined in terms of make variables
 - `CC = gcc`
 - `CFLAGS = -Wall`
- For C++, the equivalent make variables are:
 - `CXX`
 - `CXXFLAGS`
 - `CPPFLAGS` for preprocessor options

Compiling a C program (...)

```
# Makefile
CC=gcc
CFLAGS=-Wall
main: main.o hello.o

clean:
    rm -f main main.o hello.o

$ make
gcc -Wall -c -o main.o main.c
gcc -Wall -c -o hello.o hello.c
gcc main.o hello.o -o main

$ ./main
Hello, world!
```

- Static libraries
 - special “archive files”
 - have extension `.a`
 - created from object files using GNU archiver `ar`
 - used by linker to resolve references to functions at compile time
- Shared libraries
 - have extension `.so`
 - preferred over static libraries

Libraries (...)

- GNU *ar* to create a static library:

```
$ gcc -Wall -c -o hello.o hello.c
$ ar cr libhello.a hello.o
```

The 'cr' stands for “create and replace”

- The “table of contents” option 't' can list the object files in an existing library

```
$ ar t libhello.a
hello_fn.o
bye_fn.o
```

```
$ gcc main.c /tmp/tmp/test/libhello.a -o main
$ ./main
Hello, everyone!
```

Libraries (...)

```
/* calc.c */
#include <math.h>
#include <stdio.h>

int
main (void)
{
    double x = sqrt (2.0);
    printf ("The square root of 2.0 is %f\n", x);
    return 0;
}
```

Libraries (...)

```
$ gcc -Wall calc.c -o calc
/tmp/ccBR60jm.o: In function 'main':
/tmp/ccBR60jm.o(.text+0x19): undefined reference
to 'sqrt'
```

- '/tmp/ccBR60jm.o' is a temporary object file created for linking
- A library should appear after any source files or object files

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

Libraries (...)

- `-lname` will link object files with library file 'libNAME.a' in the standard library directories

```
$ gcc -Wall calc.c -lm -o calc
```

- If library `lglpk` uses and depends on `lm` library, it must appear before `lm`

```
$ gcc -Wall data.c -lglpk -lm
```

- Not all compilers search all libraries, so order libraries from left to right

Libraries (...)

```
$ gcc -Wall -fPIC -c hello.c
$ gcc -shared -Wl,-soname,libhello.so.1 -o \
  libhello.so.1.0 hello.o
$ ln -sf libhello.so.1.0 libhello.so
$ ln -sf libhello.so.1.0 libhello.so.1
$ gcc -Wall -I/tmp/tmp/test -L/tmp/tmp/test main.c \
  -lhello -o main

$ ./main
./main: error while loading shared libraries: \
  libhello.so.1: cannot open shared object file: \
  No such file or directory
```

Libraries (...)

```
$ export LD_LIBRARY_PATH=/tmp/tmp/test:$LD_LIBRARY_PATH
$ gcc -Wall -I/tmp/tmp/test -L/tmp/tmp/test main.c \
  -lhello -o main

$ ./main
Hello, everyone!
```

Compilation Options

- A common problem when using library header files

```
FILE.H: No such file or directory
```

```
/usr/bin/ld: cannot find LIBRARY
```

- Default directory search locations for header files

```
/usr/local/include/
```

```
/usr/include/
```

for libraries

```
/usr/local/lib/
```

```
/usr/lib/
```

Compilation Options (...)

- “include path” - list of directories for header files
- Header file in `/usr/local/include` takes precedence over `/usr/include`
- `-I` to add new directory to search path
- “library search path” or “link path” - list of directories for libraries
- Library in `/usr/local/lib` takes precedence over `/usr/lib`
- `-L` to add new directory to library search path

Compilation Options (...)

- Never use absolute paths of header files in *#include* statements
- Environment variables in shell, or `.bash_profile` in GNU bash for search paths
- Additional directories can be added to include path using environment variable:
 - `C_INCLUDE_PATH` (for C header files)
 - `CPLUS_INCLUDE_PATH` (for C++ header files).
- Several search directories can be a colon separated list

`DIR1:DIR2:DIR3:...`

Compilation Options (...)

- *libgdbm.so* shared object file is preferred over *libgdbm.a* static library
- Set load path through environment variable *LD_LIBRARY_PATH*

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib  
$ export LD_LIBRARY_PATH
```

- *LD_LIBRARY_PATH* can be set in
 - */etc/profile*
 - */etc/ld.so.conf*
- *-static* with *gcc* to avoid the use of shared libraries

C Language Standards

- The variable name *asm* is valid under the ANSI/ISO standard

```
/* ansi.c */
#include <stdio.h>

int
main (void)
{
    const char asm[] = "6502";
    printf ("the string asm is '%s'\n", asm);
    return 0;
}
```

C Language Standards (...)

- *asm* is a GNU C keyword extension for native assembly instructions to be used in C functions

```
$ gcc -Wall ansi.c
ansi.c: In function 'main':
ansi.c:6:14: error: expected identifier or '(' before 'asm'
ansi.c:7:39: error: expected expression before 'asm'
```

C Language Standards (...)

```
$ gcc -Wall -ansi ansi.c
```

- Non-standard keywords and macros defined by GNU C extensions:
 - asm
 - inline
 - typeof
 - unix
 - vax
- Use macro ‘_GNU_SOURCE’ to enable extensions in the GNU C library

- Other Macro extensions:
 - POSIX extensions ('_POSIX_C_SOURCE')
 - BSD extensions ('_BSD_SOURCE')
 - SVID extensions ('_SVID_SOURCE')
 - XOPEN extensions ('_XOPEN_SOURCE')
- *-pedantic* to write portable programs which follow ANSI/ISO standard
- *-std* option for a specific language standard

Warning Options

- *-Wcomment* (included in *-Wall*) warns about nested comments
- Safe way to “comment out” section of code containing comments

```
#if 0
```

```
#endif
```

- *-Wformat* (included in *-Wall*) incorrect use of format strings
- *-Wunused* (included in *-Wall*) unused variables

Warning Options (...)

- *-Wimplicit* (included in *-Wall*) functions used without being declared
- *-Wreturn-type* (included in *-Wall*)
 - functions defined without return type but not declared *void*
 - Empty *return* statements in functions that are not declared *void*
- *-W* common programming errors
 - functions which can return without a value
 - comparisons between signed and unsigned values

The options *-W* and *-Wall* are normally used together

Warning Options (...)

- *-Wconversion* implicit type conversions
- *-Wshadow* redeclaration of a variable name in a scope where it has already been declared
- *-Wcast-qual* pointers that are cast to remove a type qualifier, such as *const*
- *-Wwrite-strings* gives all string constants a *const* qualifier
- *-Wtraditional* warns code interpreted differently by an ANSI/ISO compiler and a “traditional” pre-ANSI compiler
- *-Werror* changes the default behavior by converting warnings into errors, stopping the compilation whenever a warning occurs

Using the preprocessor

- GNU C preprocessor *cpp* expands macros in source files before compilation

```
$ cpp hello.c > hello.i
```

- *-DNAME* defines a preprocessor macro 'NAME' from the command line
- Preprocessor is integrated into compiler, although a separate *cpp* command is also provided
- Macros are generally undefined
- Reserved macros defined by the compiler begin with double-underscore prefix `__`
- Complete set of predefined macros

```
$ cpp -dM /dev/null
```

Using the preprocessor (...)

- Non-standard macros can be disabled with *-ansi*
- Surround macros by parentheses whenever they are part of an expression
- When a macro is defined with *-D* alone, *gcc* uses a default value of *1*
- *-E* option with *gcc* runs the preprocessor, displays the output without compiling the source code
- *-save-temps* option saves preprocessor output, *.s* assembly files, and *.o* object files

```
$ gcc -c -save-temps hello.c
```

- *hello.i* has preprocessed output

Compiling for debugging

- `-g` to store additional debugging information in object files and executables
- `ulimit -c` controls the maximum size of core files
- To allow core files of any size to be written:

```
$ ulimit -c unlimited
```

Note that this setting only applies to the current shell.

Compiling for debugging (...)

- To load core files into GNU Debugger *gdb*

```
$ gdb EXECUTABLE-FILE CORE-FILE
```

```
$ gdb a.out core
```

- *break function-name* to set a breakpoint on a specific function:

```
$ gdb a.out
```

```
(gdb) break main
```

```
Breakpoint 1 at 0x80483c6: file null.c, line 6
```

- To move forward without tracing calls, use the command *next*

Compiling for debugging (...)

- *set variable* to change the value of variable in a running program:

```
(gdb) set variable p = malloc(sizeof(int))
```

- *finish* continues execution up to the end of the current function
- *continue* runs until the program exits (or hits the next breakpoint)
- To debug a program interactively use tools like Emacs GDB mode (M-x gdb), DDD or Insight

Compiling for debugging (...)

- It is not possible to set a breakpoint on an inlined function

```
$ gcc -v --help
```

- Version number:
 - MAJOR-VERSION.MINOR-VERSION or
 - MAJOR-VERSION.MINOR-VERSION.MICRO-VERSION
- `-v` lists commands used to compile and link a program

```
$ gcc -v -Wall hello.c
Using built-in specs.
COLLECT_GCC=/usr/bin/gcc
...
```

Compiling with optimization

- Many source-level optimization techniques
- Common subexpression elimination (CSE)
 - increases the speed
 - reduces code size

```
x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2))
```

or

```
t = sin(u/2)
```

```
x = cos(v)*(1+t) + sin(w)*(1-t)
```


Compiling with optimization (...)

- *inline* keyword for a specific function should be inlined
- Loop unrolling
 - increases speed of executable
 - increases code size
- *-OLEVEL* optimization level
 - *LEVEL* 0-3
- *-O0* or no *-O*
 - default
 - no optimization level
 - debugging

Compiling with optimization (...)

- *-O1* or *-O*
 - common forms of optimization
 - executables are smaller and faster
- *-O2*
 - include instruction scheduling
 - longer to compile programs
 - require more memory than with *-O1*
 - best choice for development and deployment
- *-O3*
 - more expensive optimizations

Compiling with optimization (...)

- *user* time gives actual CPU time spent running the process
- *real* and *sys* provide the total real time for the process to run
 - time where other processes used the CPU
 - time spent waiting for OS calls
- Use optimization with debugging option '-g' is recommended
 - development
 - deb
- *-Wuninitialized* (included in *-Wall*) warns uninitialized variables

Compiling a C++ program

- The GNU C++ compiler is a true C++ compiler
- C++ source code file extensions:
 - .cc
 - .cpp
 - .cxx
 - .C
- *-ansi* option requests compliance with the C++ standard
- Linking C++ object files with gcc produces undefined references errors

Compiling a C++ program (...)

- *-Wall* and *-W* include extra warnings specific to C++
- *-fno-default-inline* disables default inlining of member functions
- *-Weffc++* warns about C++ code which breaks some of the programming guidelines given in the books *Effective C++* and *More Effective C++* by Scott Meyers.
- *-Wold-style-cast* option highlights any uses of C-style casts in C++ programs
- Use “include guards” to ensure header definitions are parsed only once

Platform specific options

- `-m` for platform-specific options for different types of CPUs
- `-march=CPU` will be faster but will not run on other processors
- `-mcpu=CPU` is tuned for a specific processor
- `-mmmx`, `-msse`, `-msse2`, `-msse3`, and `-m3dnow` enable the use of extra instructions
- `-msse2 -mfpmath=sse` to use SSE extensions for floating-point arithmetic
- `-m32` allows 32-bit code to be generated

Platform specific options (...)

- The possible values of *CPU*:
 - *power*
 - *power2*
 - *powerpc*
 - *powerpc64*
 - *common* for POWER/PowerPC
- *-mieee* to enable full support for IEEE arithmetic
- The IEEE-754 standard defines the bit-level behaviour of floating-point arithmetic operations

Platform specific options (...)

- *-fsigned-char* or *-funsigned-char* to set the default type of char
- *-fsigned-bitfields* and *-funsigned-bitfields* control definitions of bitfields in structs
- Functions `getc`, `fgetc` and `getchar` have a return type of `int`, not `char`

Compiler related tools

- `-pg` to be used for profiling

```
$ gcc -Wall -c -pg collatz.c
```

```
$ gcc -Wall -pg collatz.o
```

- `-pg` should be used
 - with each source file
 - during linking
- `gmon.out` contains profiling data in the current directory

Compiler related tools (...)

- To enable coverage testing

```
$ gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

- Data written to several files
 - *.bb*
 - *.bbg*
 - *.da*
- Lines which were not executed are marked with hashes '#####'

How the compiler works

- Preprocessed files are given the file extension:
 - *.i* for C programs
 - *.ii* for C++ programs

```
$ cpp hello.c > hello.i
```

- **-S** command-line option converts C source code into assembly without creating an object file

```
$ gcc -Wall -S hello.i
```

- Assembler converts assembly language into machine code and generates an object file

Examining compiled files

- **file** command determines file type

```
$ file a.out
```

```
a.out: ELF 32-bit LSB executable, Intel 80386,  
version 1 (SYSV), dynamically linked (uses  
shared libs), not stripped
```

- ELF = Executable and Linking Format
- COFF = Common Object File Format

```
$ file hello.o
```

```
hello.o: ELF 64-bit LSB relocatable, x86-64,  
version 1 (SYSV), not stripped
```

Examining compiled files (...)

- **nm** command lists symbols from object files

```
$ nm hello.o
0000000000000000 T hello
                 U printf
```

- **ldd** prints shared library dependencies

```
$ ldd hello
linux-vdso.so.1 => (0x00007fff73dff000)
libc.so.6 => /lib64/libc.so.6 (0x00000032a3a00000)
/lib64/ld-linux-x86-64.so.2 (0x00000032a3600000)
```

Common error messages

- *No such file or directory*
- *macro or #include recursion too deep or #include nested too deeply*
 - two or more files trying to include each other
- *invalid preprocessing directive #...*
- *warning: This file includes at least one deprecated or antiquated header*
 - C++ programs which include old-style library header files
- *variable undeclared (first use in this function)*

Common error messages (...)

- *parser error before '...' or syntax error*
- *parse error at end of input*
 - compiler encounters end of a file unexpectedly
- *warning: implicit declaration of function '...'*
 - function used without a prototype being declared
- *unterminated string or character constant*
- *character constant too long*
 - single quotes are used to enclose more than one character

Common error messages (...)

- *warning: initialization makes integer from pointer without a cast*
 - indicates a misuse of a pointer (NULL, for example) in an integer context
- *dereferencing pointer to incomplete type*
 - access elements of struct before struct declaration
- *warning: suggest parentheses around assignment used as truth value*
 - use of assignment operator '=' instead of the comparison operator '=='
- *warning: control reaches end of non-void function*
 - missing return value for all cases or not well-defined

Common error messages (...)

- *warning: assignment of read-only location warning: cast discards qualifiers from pointer target type warning: assignment discards qualifiers ... warning: initialization discards qualifiers ... warning: return discards qualifiers ...*
 - pointer is used incorrectly, violating a type qualifier, such as `/const/`
- *initializer element is not a constant*
 - global variables initialized with non-constant value
- GCC cannot recognize the file type
- *undefined reference to 'foo' collect2: ld returned 1 exit status*
 - function or variable not found in any object files or libraries used with linker

Common error messages (...)

- *error while loading shared libraries: cannot open shared object file: No such file or directory*
- *Segmentation fault Bus error* These runtime messages indicate a memory access error. Common causes include:
 - dereferencing a null pointer or uninitialized pointer
 - out-of-bounds array access
 - incorrect use of *malloc*, *free*, and related functions
- *floating point exception*
 - is caused by an arithmetic exception:
 - division by zero
 - overflow
 - underflow
 - invalid operation (taking square root of -1)
- *Illegal instruction*

- An Introduction to GCC.
<http://www.network-theory.co.uk/gcc/intro/>
- GNU Compiler Collection. <http://gcc.gnu.org/>