Development Tools 4: Compiling

1. Intro
2. Text Editors
3. IDE's
4. **Compiling**
   - **GCC compiler**
   - **Make build tool**
   - **feature test macros**
5. Debugging and Tracing
6. Advanced Development

GCC

The **GNU Compiler Collection** is the most popular C/C++ **compiler** for Linux/UNIX.

GCC is free in every sense of the word and supports a wide range of processors.

Originally GCC stood for **GNU C Compiler**, but the name was changed as support was added not only for C++ but also for other languages like Fortran and Pascal.

GCC will be available for any Linux distro, though it might not be installed by default.

GCC (contd.)

There have been fairly significant changes between some releases of GCC, so occasionally you may find software that compiles without problems on one distro but has slight issues on another.

For this class, any version of GCC should be acceptable.

Under Linux, GCC's C compiler is typically invoked with:
`gcc`

By default, `gcc` will run the **preprocessor**, C compiler, **assembler**, and **linker**, to produce an **executable** file.

Options can be supplied to limit how far the compilation process proceeds (e.g., produce only **object file**(s), no linking).

GCC (contd.)

A typical call to `gcc` to compile a single-file program would be:

```
gcc -Wall -std=gnu99 -omyprog myprog.c
```

Explanation:

- `-Wall`: turns on all compiler *warnings*, as most warnings indicate a true problem with the program
- `-std=gnu99`: uses C99 extensions plus various GNU extensions, to allow `for` index variable declaration inside `for`
- `-omyprog`: specifies an output executable file named `myprog` (by default the executable will be named `a.out`)
- `myprog.c`: the source file, should have `.c` extension for GCC

List all files with a multi-file program:

```
gcc -Wall -std=gnu99 -omyprog myprog1.c myprog2.c myprog3.c
```

GCC Options

GCC has a very large number of options, and a very long man page, so it is often easier to use the manual (HTML or PDF).

Key GCC options:

- `-o`*file* – output filename is *file*
- `-Wall` – turn on all warnings
- `-std=c99` – use C99 standard
- `-std=gnu99` – use C99 plus GNU extensions standard
- `-g` – include debugging information in output file
- `-l`*library* – use *library* when linking
- `-L`*directory* – add *directory* to search list for libraries
- `-D`*name* – define preprocessor macro *name* as 1
- `-D`*name*=*def* – define preprocessor macro *name* as *def*
- `-I`*directory* – add *directory* to search list for header files
- `-O`n – apply **optimizations** during compilation, $n \in \{1, 2, 3\}$

GCC Library Options

When including *libraries* (`-l` option):
"It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the *order they are specified*."

This means that libraries should generally be specified *last*.

If using Pthreads, the option `-pthread` is *preferred to* `-lpthread`:
"`-pthread`...also sets flags for the preprocessor, so it should be used consistently for both compilation and linking."

A basic call to `gcc` that used Pthreads and the real-time library:

```
gcc -Wall -std=gnu99 -omyprog myprog.c -pthread -lrt
```

Alternatives to GCC

In the past, the only serious competitor to GCC has been Intel's compilers (that cost lots of money and support only Intel hardware).

Recently, though, **Clang** has become a viable alternative.

Clang supports C, C++, and Objective-C.

Clang development has been supported by both Apple and Google.

Like GCC, Clang is a free/open source project (not GPL however).

Clang aims to be compatible with GCC, using largely similar command-line options.

It aims to better support incremental compilation to assist IDE's.

Build Tools: Make

Make is a utility that can be used to simplify building/re-building software.

Make uses a file called a **makefile**, which specifies how to build target files and what their dependencies are.

It analyzes what files are available and their dates to determine what actions need to be taken to produce a specified target.

E.g., if an executable relies on two object files, each of which relies on one source code file, make could see that one source file has been modified so it must be compiled and then used to produce a new executable.

Linux uses GNU's version of Make.

The standard names for makefiles are:
`Makefile` (most common) or just `makefile`.

Make (contd.)

Invoking Make can then be as simple as:

`make`

or

`make` *target*

A **target** is a name used to specify what is to be accomplished, but if no target is given the *first* target in the file will be used.

Often the target will be the name of the executable that is supposed to be built, though it could be `all`, `clean`, etc.

The primary elements of makefiles are **rules**, which have syntax:
target: [*prerequisites*]
<tab>*action_command*

Note: the action line must be introduced by a *tab*!

Make (contd.)

The simplest sort of makefile we might have, where a single source code file is used to build an executable, would look like:

```
myprog: myprog.c
<tab>gcc -Wall -std=gnu99 -omyprog myprog.c
```

A basic makefile for a program that consists of two source files:

```
myprog: myprog1.o myprog2.o
<tab>gcc -omyprog myprog1.o myprog2.o

myprog1.o: myprog1.c
<tab>gcc -Wall -c myprog1.c

myprog2.o: myprog2.c
<tab>gcc -Wall -c myprog2.c
```

Make (contd.)

Rule preprequisites represent *dependencies* among the files.

For example, the first time we make `myprog`, make will see that `myprog1.o` `myprog2.o` are required, so the rules for their creation will be used to produce them, then the action to produce `myprog` will be run.

The advantage of using make rather than a simple script is that make will check modification times on existing files and only rebuild what is necessary.

Thus, if we edit `myprog1.c` and again make `myprog`, make will see that `myprog1.o` needs updating but `myprog2.o` does not, so it will use the rule for `myprog1.o` and then run the action to create `myprog`.

Make (contd.)

GNU make understands how to build C programs, so in fact the rules for producing the object (`.o`) files are not required in the `myprog` makefile.

Note that prerequisites may also include (non-library) *header files*, since if a project header file changes, that is effectively the same as if every source file that `#include`'s that header has also changed.

Makefiles for complex, multi-file systems can get very large and complicated, and *variables* may be used to simplify the rules:

```
objects = file1.o file2.o...

myprog: $(objects)
<tab>gcc -omyprog $(objects)

$(objects): header.h
```

Feature Test Macros

A somewhat esoteric and potentially confusing aspect of Linux C code development is the use of **feature test macros**.

Since there are a number of C standards and UNIX standards, the definitions (library, system call, constants) that are made available by default may not always be what a programmer wants.

The man page says: "Feature test macros allow the programmer to control the definitions that are exposed by system header files when a program is compiled. This can be useful for creating portable applications, by preventing non-standard definitions from being exposed. Other macros can be used to expose non-standard definitions that are not exposed by default."

Feature test macros make use of the preprocessor macro facility and the `#define` directive.

Feature Test Macros (contd.)

The *man page* for a function/call will provide information about any macro definitions that must have been made in order to use the function/call.

For example, the man page for the system call `fchmod` says:

```
SYNOPSIS
    #include <sys/stat.h>

    int chmod(const char *path, mode_t mode);
    int fchmod(int fd, mode_t mode);

  Feature Test Macro Requirements for glibc:

    fchmod(): _BSD_SOURCE || _XOPEN_SOURCE >= 500
```

Feature Test Macros (contd.)

This says that in order to use `fchmod`, you must have a line such as either of the following in your code:

```
#define _BSD_SOURCE
```

or

```
#define  _XOPEN_SOURCE 500
```

These macros/constants must have been defined *prior to* the related *header includes*.

They can also be defined in a call to GCC:

```
gcc -D_BSD_SOURCE ...
```

Note that specifying standards options to compilers such as GCC will also affect the feature test macros that are defined by default:

```
gcc -std=c89 ...
```