GNU Compiler Collection

- GCC: originally referred to GNU C Compiler
  - Richard Stallman, the founder of the GNU Project
  - First release (1.00) 1987, C Compiler
  - 2.00 release in 1992, with C++ compiler …
  - 3.0 release in 2001, and 4.00 released in 2005; GCC 4.8 in CS Lab
- GCC➔GNU Compiler Collection (Now)
  - Compilers: gcc, c++, g++, gcj (java), f77, f90, etc.
  - A portable compiler
    - It runs on most platforms available today and produces executable for many types of processors
- Comprehensive online documentation:

  http://gcc.gnu.org/onlinedocs/gcc

Cross-Compiler

GCC is not only a native compiler but also a Cross-Compiler

- **Cross-Compiling**
  - Producing executable for a system different from the one used by GCC itself
- Used for embedded system development, which normally small and not capable of running a compiler

Source Code Compilation

- Compilation refers to the process of converting a program from the textual *source code* in a programming language, such as C or C++, into *machine code (binary file)*, the sequence of 1's and 0's used for the Central Processing Unit (CPU) of the computer.
- This machine code is then stored in a file known as
  - an *executable file*, sometimes also referred to as a *binary file*.

Compile C/C++ Codes

- Compiler cmd: **`gcc |g++|c++ |cc`**
- Compile a simple C/C++ program on UNIX
  - **`gcc -Wall hello.c -o hello`**
  - **`c++ -Wall hello.cpp -o HelloWorld`**
    - `-Wall`: turns on all the commonly used compiling warnings
    - `-o` : name of the binary/exe file,
    - `a.out: the default executable` in case `-o` is omitted
    - To run the compiled program: `./hello` or `./a.out`
  - The compiled program will have "`x`" permission for all (this is done by the compiler during linking process)
- Compile several files
  - **`gcc -Wall prog1.c prog2.c -o prog`**
- Compile files with external libraries
  - **`c++ -Wall Hello.cpp Main.cpp -o helloworld -lm`**

Compiling Multiple Source Files

Two stages/steps:

- Compilation: build object files, normally ending with `.o`
  - Create object files from source files with `-c` option

```
gcc -Wall -c prog1.c [ -o prog1.o ]
gcc -Wall -c prog2.c [ -o prog2.o ]
```

- Linking: link the objects and libraries to build the executables
  - Link, create executables from object files (*.o)
    - `gcc prog1.o prog2.o -o prog`
- Why using the stages to build an application?
  - Separate the compiling and linking can save time in case that an application contains many files
    - Recompiling a large number of source codes can be time-consuming
    - Only the modified file needs to be recompiled, then linked together with others

Header Files

- System header files: `#include <stdio.h>`
  - The compiler will search the system header file directories for those header files enclosed with < >
  - Default system header file path:

    `/usr/local/include/:/usr/include/`
- Customer-defined header files
  - `#include "myheader.h"`
  - The compiler will search the current directory first before searching the system header file directories.
  - You can specify the header file search path during compilation with complier's `-I` option:
    - `-I/path/to/additional/header/files/`
  - There is also a set of environmental variables for this:

    `CPATH; C_INCLUDE_PATH; CPLUS_INCLUDE_PATH; OBJC_INCLUDE_PATH`
  - **Never include path of headers in the source codes.**

Libraries

- A library is a collection of precompiled object files which can be linked into programs,
  - such as C library provides all the C functions, or the math library provides all the math functions, such as **`sin, sqrt`**, etc.
- External libraries
  - You need to specify the names of the external libraries containing the functions referenced in the source codes
    - **`-lNAME`**, such as **`-lm`** in the previous example
  - In case the libraries are not available in the standard library directories/paths, the path needs to be specified as well
- Standard library path:
  - **`/usr/local/lib/:/usr/lib/:/lib:/lib64:`**
- The order of the search path matters!!!
  - There might be dependence among the libraries.

Link-Order of the Libraries

The traditional behavior of linkers is to search for external functions from left to right in the libraries specified on the command line

- The library containing the definition of a function should appear after the source file in which it is called.

  `c++ -Wall Hello.cpp -lmlib Main.cpp -o helloworld`

  (Main.cpp has a function defined in libmlib library)

  The above compilation will fail. Why?

  `c++ -Wall Hello.cpp Main.cpp -lmlib -o helloworld`

- When linking multiple libraries, same convention followed
  - A library which calls an external function defined in another library should appear before the library containing the function
    - **`gcc -Wall data.c -lglpk -lmylib`**
    - Where `libglpk` uses function defined in `libmylib`
- Rule of thumb:
  - The most general one goes to the most right (the last)

Additional Include and Library Path

- headers & lib searching path with **-I** and **-L** compiler options in-line during compilation
  - **gcc -Wall -I/opt/new/include/ -L/opt/new/lib file.c -o prog**
  - -I and -L can be repeated to add more paths for headers and libraries respectively (just separated by white space)

```
gcc -Wall -I/opt/new1/include -I/opt/new2/include/
  -L/opt/new1/libdir1 -L/opt/new2/libdir2 file.c -o prog
```

- Set the environmental variables in shell initialization files such as `.bashrc` in bash
  - `export` **`C_INCLUDE_PATH`**`=/opt/new/include (for GCC)`
  - `export` **`CPLUS_INCLUDE_PATH`**`=/opt/new/include (for G++)`
  - `export` **`LIBRARY_PATH`**`=/opt/new1/lib:/opt/new2/lib`
- These directories will be searched after those specified with –L options in command line, and before the standard library directories.

Search Order of the Lib Paths

- First
  - The in-line user-specified path: those specified with **–I** or **–L** option (gcc/g++ compilation options) in cmd line
- Second: environment variables if defined
  - **C_INCLUDE_PATH for C codes**
  - **CPLUS_INCLUDE_PATH** for C++ codes
  - **LIBRARY_PATH** for additional library paths
- Last
  - The standard library directories, such as /use/lib, /lib, /lib64, etc

C/C++ Compiler Options

- **`-Wall`** turn on all the default warnings, suggest to use always
- **`-o`** object/target file name
- **`-c`** compile without linking, create object files (*.o) only
- **`-Dfoo`** define a preprocessor macro
- **`-Idir -Ldir`**
  - User-defined paths for additional libraries & header files
- **`-lfoo`** link against library libfoo (can be libfoo.a or libfoo.so, *.so preferred)
- **`-O[n]`** optimization level (n: from 0 to 3), -O2 is mostly used,
- **`-Os`** optimization for small size of the executable, not speed
- **`-g`** include standard debugging information

https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html

Compiling with Optimization Flags

"Turning on optimization flag makes the compiler attempt to improve the performance of the executable and/or reduce the size of code at the expense of compilation time and possibly the ability to debug the program"

- Goals of compiling optimization
  - increase the speed at run-time, program runs faster,
  - or reduce the size of the executable,
  - or make the program debuggable

  Generally, it's not possible to achieve all of them the same time
- The higher the optimization level, the more complex of the compilation process, the longer compilation time, and the harder to debug a program
  - The optimization needs to be turned off for debugging compilation
- An optimization level is chosen with the command line option **`-OLEVEL`** where LEVEL ranges from 0 to 3 & s

GCC Optimization

- `-Os:` optimize for size
  - to reduce the size of the executable
- `-O0` : default
  - No optimization, compiling the code in the most straightforward way, fast compilation
  - It's the option to compile programs for debugging
- `-O1` or `-O`
  - Turn on the most common forms of optimization w/o sacrificing the speed / space trade off
  - Executable should be smaller and faster than `-O0`
  - More expensive optimizations, such as instruction scheduling are not used at this level

More Optimization Levels

- **`-O2:`** Turns on further optimizations in addition to those used by **`-O1`**
  - Include instruction scheduling
  - Only optimizations that do not require any speed-space tradeoffs are used
    - So the size of the executables should be about the same as those created with `-O1`
    - But compilation takes longer time requires more memory
    - Generally, this is the best optimization level and it is the default optimization level for application release package
- **`-O3:`** Turns on more expensive optimizations, takes longer time to compilation
  - Increase the speed by increasing the size of the executable
  - Not a preferred optimization level, Sometimes, it could be slower than **`-O2`**
- Normally: **`-O0`** for debugging, **`-O2`** for release
  - Ex: **`c++ -Wall -O2 hello.cpp -o hello`**

Compiling with Debugging Enabled

- Normally, an executable file is simply the sequence of machine code instructions produced by the compiler.
  - No variable names, or line numbers
  - The above info are needed in tracking running errors
    - Trace back from a specific machine instruction to the corresponding line in the source file
- The debug compilation option provided by GCC (`-g`)
  - Storing the names and source code line-numbers of functions and variables in a *symbol table* in the object file or executable.
- The executable will be debuggable with a debugger, such as GDB (GNU Debugger)
  - GDB allows the values of variables to be examined while the program is running.

Two Types of Libraries

Static libraries

- Are simply collections of object files arranged by the **`ar`** (archciver) utility
- Name of library file ends with (conventionly) **`.a`**
- Created by a special "**`ar`**" program as the following

```
ar rcs libempolyee.a employee.o staff.o
```

  - **`r`** ➔Includes the object files in the library: replacing any object files already in the archive that have the same names.
  - **`c`** ➔Silently create the library if it does not already exist.
  - **`s`** ➔Maintain the table mapping symbol names to object file names.

Shared Libraries

- A shared library or shared object is a file that is intended to be shared by executable files.
  - Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.
- Ending with ".so" referred to Shared Objects, plus version number, similar to the ".dll" on Windows System
- Created by a special option of gcc during compilation

```
gcc -fPIC -Wall -O2 foo.c -o foo.o
gcc -shared -o libfoo.so foo.o
```

Programs Linked against Static Libraries

- The machine codes from the object files for any external functions used by the program are copied from the library into the final executable
- Pros
  - The program is a stand-alone program, has better portability. No need to install the external libraries on the host system
- Cons
  - Takes large space: due to large executable file
  - Whenever there is change in the library or the program, the whole application needs to be rebuilt

Programs Linked Against Shared Libraries

- The executable contains only a small table of the functions required from the dependent libraries, instead of copying the complete machine code from the object files for the external functions
- The machine code for the external functions is copied/loaded into memory from the shared libraries on disk by the OS during program execution →a process referred to as *dynamic linking*.
- Small executable files, saves disk space
- Application and share libraries can be maintained separately
  - Updating libraries without the need of recompiling the application
  - Updating the applications without the need to recompiling the libraries.

Applications built against shared libraries vs. against static libraries

- Advantages (.so)
  - Save disk space
    - Only one copy of the library file sits on the disk and shared by many programs
  - Small program, save memory, overall the system can run faster
  - No need to rebuild the program in case of library bug fixing as long as no changes in the APIs of the library
- Disadvantages
  - Not as portable as the one built against static library
    - The system (the program is going to run) must provide the exact same shared library as the ones the program was built on the build system.

Program Execution Built with External Libraries

- No problem in executing a program built against static libraries: portable
- Execute a program built against shared libraries
  - The shared libraries need to be available and ready to be loaded into memory when needed
  - Loader's search path:
    - By default, it's a predefined set of system directories, probably defined in file: `/etc/ld.so.conf, etc`
    - you need add new path for shared libraries which are not installed in those default paths
- Environmental variable: **`LD_LIBRARY_PATH`**
  - Define this in your current session of shell

```
export LD_LIBRARY_PATH=/opt/your/path/lib:$LD_LIBRARY_PATH
```

  - Define it in your shell initialization file, such as **`.bashrc`**