Lecture Agenda and Notes Draft:

Building Software **2**
Building software, basic concepts [10 min] 2
Basic tools for compiling programs: GCC [30 min] 3
Make and makefiles [30 mins] 6
CMake and CMakeLists [10 mins] 10
Reading 12

Lecture Agenda and Notes Draft: Building Software

Building software, basic concepts [10 min]

1. Source code and binary code, interpretable vs compiled programming languages
    a. **Source code** is any collection of code, with or without comments, written using a human-readable programming language, usually as plain text.
    b. **Source code** (also referred to as source or code) is the version of software as it is originally written (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters).
    c. **Binary code** (machine code, executables) is any low-level programming language, consisting of machine language instructions, which is used to control a computer's central processing unit (CPU).

2. **Compilers** are computer programs that translate computer code written in one programming language (the source language) into another language (the target language).
    a. Examples include GCC, clang, but also tools such as PDFLatex.

3. Take for example the following repository (https://github.com/charlesq34/pointnet2) which has introduced PointNet++, a popular system for learning with 3D point clouds. The repository includes a compilation script that seeks to provide support to the users.
    a. Observe the compilation script and name its possible flaws. What could be improved?

Basic tools for compiling programs: GCC [30 min]

- GCC (upper case) refers to the GNU Compiler Collection. This is an open source compiler suite which includes compilers for C, C++, Objective C, Fortran, Ada, Go and Java. gcc (lower case) is the C compiler in the GNU Compiler Collection. Historically GCC and gcc have been used interchangeably, but efforts are being made to separate the two terms as GCC contains tools to compile more than C.
    - g++, gcc, clang, icc are some of the other C compilers
    - g++ is equivalent to `gcc -xc++ -lstdc++ -shared-libgcc` (the 1st is a compiler option, the 2nd two are linker options).

- To install GCC on Linux, use `apt install build-essential`. To install on a Mac, use `brew install gcc`

- "Hello world!" with common command line options.

  For programs with a single source file, using gcc is simple.

```
/* File name is hello_world.c */
#include <stdio.h>

int main(void)
{
    int i;
    printf("Hello world!\n");
}
```

- To compile the file hello_world.c from the command line:

```
gcc hello_world.c
```

- When the build process fails, error messages are produced. The compiler should provide details about why the build failed.

- gcc will then compile the program and output the executable to the file `a.out`. If you want to name the executable, use the `-o` option.

```
gcc hello_world.c -o hello_world
```

- The executable will then be named `hello_world` instead of `a.out`. By default, there are not that many warnings that are emitted by gcc. gcc has many warning options and it is a good idea to look through the gcc documentation to learn what is available. Using `-Wall` is a good starting point and covers many common problems.

```
gcc -Wall hello_world.c -o hello_world
```

- Warning messages occur when a program’s syntax is not 100% clear to the compiler, but it makes an assumption and continues the build process:

```
hello_world.c: In function ‘main’:
hello_world.c:6:9: warning: unused variable ‘i’ [-Wunused-variable]
```

```
        int i;
            ^
```

Here we see we now get a warning that the variable 'i' was declared but not used at all in the function.

- If you plan to use a debugger for testing your program, you'll need to tell gcc to include debugging information. Use the `-g` option for debugging support.

```
gcc -Wall -g hello_world.c -o hello_world
```

- hello_world now has debugging information present supported by GDB. If you use a different debugger, you may need to use different debugging options so the output is formatted correctly. See the official gcc documentation for more debugging options.

- By default gcc compiles code so that it is easy to debug. gcc can optimize the output so that the final executable produces the same result but has faster performance and may result in a smaller sized executable. The `-O` option enables optimization. There are several recognized qualifiers to add after the `O` to specify the level of optimization. Each optimization level adds or removes a set list of command line options. `-O2`, `-Os`, `-O0` and `-Og` are the most common optimization levels.

```
gcc -Wall -O2 hello_world.c -o hello_world
```

- `-O2` is the most common optimization level for production-ready code. It provides an excellent balance between performance increase and final executable size.

```
gcc -Wall -Os hello_world.c -o hello_world
```

- `-Os` is similar to `-O2`, except certain optimizations that may increase execution speed by increasing the executable size are disabled. If the final executable size matters to you, try `-Os` and see if there is a noticeable size difference in the final executable.

```
gcc -Wall -g -Og hello_world.c -o -hello_world
```

- Note that in the above examples with `-Os` and `-O2`, the `-g` option was removed. That is because when you start telling the compiler to optimize the code, certain lines of code may in essence no longer exist in the final executable making debugging difficult. However, there are also cases where certain errors occur only when optimizations are on. If you want to debug your application and have the compiler optimize the code, try the `-Og` option. This tells gcc to perform all optimizations that should not hamper the debugging experience.

```
gcc -Wall -g -O0 hello_world.c -o hello_world
```

- `-O0` performs even less optimizations than `-Og`. This is the optimization level gcc uses by default. Use this option if you want to make sure that optimizations are disabled.

- **Compiling programs that depend on external libraries.** Two ways of compiling with dependencies exist: dynamic or static linking
- **Dynamic linking (the default):** naming convention: lib*libraryname*.so (e.g. libcurl.so is a dynamic curl library)
  - Only the name of the library is copied into the final executable, not any actual code.
  - At run-time, the executable searches the `LD_LIBRARY_PATH` and standard path for the library.

  - Requires less memory and disk space; multiple binaries can share the same dynamically linked library at once.
  - By default, a linker looks for a dynamic library rather than a static one.

- **Static linking** includes all of the libraries into the executable itself.
  - Linker copies all library rou-nes into the final executable.
  - Requires more memory and disk space than dynamic linking.
  - More portable because the library does not need to be available at run-me.

```
gcc -Wl,-Bstatic -llib1 -llib2 file.c
```

- In all cases, **linking to libraries** in non-standard locations requires the following information at build time:
  - Name of library (specified with `-llibraryname` flag)
  - Location of library (specified with `-L/path/to/non/standard/location/lib`)
  - Location of header files (specified with `-I/path/to/non/standard/location/include`)
  - In this example, two libraries (gsl and gslcblas) are linked to the final executable:

```
 gcc –L/usr/local/gsl/latest/x86_64/gcc46/nonet/lib
–I/usr/local/gsl/latest/x86_64/ gcc46/nonet/include –lgsl –lgslcblas
bessel.c –Wall –O3 –o calc_bessel
```

- When linked, `ldd` tool displays which libraries have been linked to a binary executable:

```
ldd /usr/local/bin/swig
    linux-vdso.so.1 =>  (0x00007ffe80be9000)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fea0bd69000)
    libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007fea0b9e7000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fea0b7d1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fea0b406000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fea0b1e9000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fea0aee0000)
    /lib64/ld-linux-x86-64.so.2 (0x000055f48d3e2000)
```

- `otool` is the MacOS equivalent:

```
otool -L /Library/TeX/texbin/pdftex
/Library/TeX/texbin/pdftex:
          /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version
400.9.4)
     /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
1252.250.1)
```

Make and makefiles [30 mins]

1. Focus on the C/C++ compilation use case. Interpreted languages like Python, Ruby, and Javascript don't require an analogue to Makefiles.
2. **Installing make.**
   Ubuntu Linux: `apt-get install build-essential`
   Mac OS: `brew install make` or `xcode-select --install`

3. **Using a Makefile.** To use an existing Makefile, you can simply run:
   ```
   make
   ```
   even without specifying which makefile you're referring to or which target you're building.
   - A few most useful options are
     ```
     make -f makefile
     make -j jobs
     ```

4. **Writing a Makefile.** A Makefile consists of a set of rules. A rule generally looks like this:
   ```
   targets: prerequisites
      command
      command
      command
   ```
- The targets are file names, separated by spaces. Typically, there is only one per rule.
- The commands are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The prerequisites are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies.

- **Understanding dependencies:**
  - Dependency files need to exist before the commands for the target are run.
  - Unix keeps 3 timestamps for each file: mtime, ctime, and atime.
  - atime (access time) is the time when the file was last read
  - mtime is the file modification time. mtime changes when you write to the file. It is the age of the data in the file.
  - ctime is the inode change time. Whenever mtime changes, so does ctime. But ctime changes a few extra times. For example, it will change if you change the owner or the permissions on the file.
  - Make uses ctime to detect if anything changes.

- The following Makefile has three separate rules. When you run `make blah` in the terminal, it will build a program called `blah` in a series of steps:
  ```
  blah: blah.o
      cc blah.o -o blah # Runs third

  blah.o: blah.c
      cc -c blah.c -o blah.o # Runs second

  blah.c:
      echo "int main() { return 0; }" > blah.c # Runs first
  ```

- Make is given blah as the target, so it first searches for this target
- blah requires blah.o, so make searches for the blah.o target

- blah.o requires blah.c, so make searches for the blah.c target
- blah.c has no dependencies, so the echo command is run
- The cc -c command is then run, because all of the blah.o dependencies are finished
- The top cc command is run, because all the blah dependencies are finished
- That's it: blah is a compiled c program

- This makefile has a single target, called some_file. The default target is the first target, so in this case some_file will run.
```
some_file:
    echo "This line will always print"
```

- This file will make some_file the first time, and the second time notice it's already made, resulting in make: 'some_file' is up to date.
```
some_file:
    echo "This line will only print once"
    touch some_file
```

- Here, the target some_file "depends" on other_file. When we run make, the default target (some_file, since it's first) will get called. It will first look at its list of dependencies, and if any of them are older, it will first run the targets for those dependencies, and then run itself. The second time this is run, neither target will run because both targets exist.
```
some_file: other_file
    echo "This will run second, because it depends on other_file"
    touch some_file

other_file:
    echo "This will run first"
    touch other_file
```

- Making multiple targets and you want all of them to run? Make an all target.
```
all: one two three

one:
    touch one
two:
    touch two
three:
    touch three

clean:
    rm -f one two three
```

- Useful feature: **variables**
  - The user can define variables with:
    `objects = foo.o bar.o all.o`
  - And reference them with:
    `$(objects)` or `${objects}`

- Useful feature: **automatic variables**

- `$@` is an automatic variable that contains the target name.
- `$?` is an automatic variable that contains all prerequisites newer than the target
- `$^` is an automatic variable that contains all prerequisites

```
hey: one two
    # Outputs "hey", since this is the first target
    echo $@

    # Outputs all prerequisites newer than the target
    echo $?

    # Outputs all prerequisites
    echo $^

    touch hey

one:
    touch one

two:
    touch two

clean:
    rm -f hey one two
```

- Useful feature: **wildcard expansion:**
  - `$(wildcard *.c)` expands to all files matching `*.c` pattern
  - `%` expands differently depending on the use (see below)

- Useful feature: **static pattern expansion**
  - Take the following example code:

```
objects = foo.o bar.o all.o
foo.o: foo.c
bar.o: bar.c
all.o: all.c
all: $(objects)
```

  - There clearly is a pattern: `%.o: %.c` where `%` is the wildcard (producing the “stem”)
  - **Static pattern expansion:**

```
targets ...: target-pattern: prereq-patterns ...
   commands
```

- Target matched by `target-pattern`, yielding “stem”
- Stem substituted into `prereq-patterns`:

```
objects = foo.o bar.o all.o
$(objects): %.o: %.c
all: $(objects)
```

(In the case of the first target, foo.o, the target-pattern matches foo.o and sets the "stem" to be "foo". It then replaces the '%' in prereq-patterns with that stem, "foo".)

CMake and CMakeLists [10 mins]

1. CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

2. **Installing CMake** can generally be done using the official downloads page: https://cmake.org/download/
   Ubuntu Linux: `apt install cmake`
   Mac OS: `brew install cmake`

3. **Using a CMake configuration system.** A simple but typical use of cmake(1) with a fresh copy of software source code is to create a build directory and invoke cmake there:

```
$ cd some_software-1.4.2
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_INSTALL_PREFIX=/opt/the/prefix
$ cmake --build .
$ cmake --build . --target install
```

- It is recommended to build in a separate directory (in this example, `build`) to the source because that keeps the source directory pristine, allows for building a single source with multiple toolchains, and allows easy clearing of build artifacts by simply deleting the build directory.

- Software projects often require variables to be set on the command line when invoking CMake. Some of the most commonly used CMake variables are listed in the table below:

Variable	Meaning
CMAKE_PREFIX_PATH	Path to search for **dependent packages**
CMAKE_MODULE_PATH	Path to search for additional CMake modules
CMAKE_BUILD_TYPE	Build configuration, such as Debug or Release, determining debug/optimization flags. This is only relevant for single-configuration buildsystems such as Makefile and Ninja. Multi-configuration buildsystems such as those for Visual Studio and Xcode ignore this setting.

CMAKE_INSTALL_PREFIX	Location to install the software to with the install build target
CMAKE_TOOLCHAIN_FILE	File containing cross-compiling data such as **toolchains andsysroots**.
BUILD_SHARED_LIBS	Whether to build shared instead of static libraries for **add_library()** commands used without a type
CMAKE_EXPORT_COMPILE_COMMANDS	Generate a compile_commands.json file for use with clang-based tools

- **Setting variables on the command line.** CMake variables can be set on the command line either when creating the initial build:

```
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DCMAKE_BUILD_TYPE=Debug
```

4. Writing a CMake-based configuration system.
    a. TBD next year

Reading

- GCC Command Options: https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html
- Compiling programs (ACCRE Vanderbilt tutorial)
- Learn Makefiles with the tastiest examples: https://makefiletutorial.com
- CMake Reference: https://cmake.org/cmake/help/latest/guide/tutorial/index.html