C/C++ – Lecture 6

AMS 595 / DCS 525

Stony Brook University – Applied Math & Statistics

Fall 2023

Review

- Dynamic memory
- Header files

Make and Makefiles

Multiple Source Files

- We may compile multiple source files using `gcc` or `g++`:
  `g++ file1.cpp ... filen.cpp -o exec_name`
  - Alternatively, to compile all C++ files in a given directory:
    `g++ *.cpp -o exec_name`
- This approach is not the most efficient nor the most convenient
  - If we only update a few files out of many there is no need to recompile the other files again
- Solution: `make` and Makefiles

make

- `make` is a tool to help build executables coming from many source files
  - Created by Stuart Feldman in 1976 while at Bell Labs
- We will be using GNU make
  - Written for Unix-like systems (e.g. macOS or Linux)
  - Can be used on Windows with a few extra steps (e.g. using Chocolatey), installation instructions can be found here
- Can be used for both C and C++ (and more)

Makefiles

- What is the purpose of a makefile?
  - To supply "rules" to `make`
  - These rules allow `make` to intelligently run commands (such as compiling)
  - Commands are only run *when needed*
- Suppose we have compiled multiple source files into a single executable and we change one of the source files. How and what should we recompile?
  - We could manually compile *all* our source files to create a new executable using g++ as we saw earlier, but this is both impractical and inefficient
  - A properly written makefile will tell `make` to only recompile those files that have been updated

Makefile Rules

- A *rule* in a makefile tells `make` when a source file needs to be recompiled and has the following syntax

```
target : prerequisites
    recipe
```

- The *target* is what we would like to create, for example:
  - Executables
  - Object files
  - Terminal commands to be carried out
- The *prerequisites* are what the target depends on, for example:
  - If the target is an executable, the prerequisite could be an object file
  - If the target is an object file, the prerequisite could be a source file

Makefile Rules

- The *recipe* contains the commands to be carried out
  - The indentation is part of the syntax and is mandatory
- The rule tells `make` when the target needs to be updated. This done when any of the following is true:
  - The target does not exist
  - The target is older than any of the prerequisites (by comparison of last-modification times)
- If the target is deemed out of date then the recipe tells `make` how to update it

Using `make` and Makefiles

- We may create a makefile by creating a file called `makefile` or `Makefile` in the same directory containing your source and object files
- To use `make` to create an executable, go to the appropriate directory in the terminal and give the command `make` when a file needs to be recompiled
- When you call `make` it will begin with the first rule
  - Thus you should put your "main" goal first
  - Other rules are then called if their target is a prerequisite of the "main goal"
  - This process is repeated until we reach a rule whose preprequistes are not itself a target of another rule
- Recall that the flag `-c` tells the compiler to output an object file

Makefile Example

```
program: file.o
    gcc file.o -o program

file.o: file.c
    gcc -c file.c
```

1. When we call `make` in the terminal, it will first check if `file.o` is more recent than `program` or if `program` doesn't exist
2. If either outcome in (1) is true, since `file.o` is the target of the rule beneath it, it checks if `file.o` exists or if it is less recent than `file.c`
3. If either outcome in (2) is true, then it will run `gcc -c file.c`
4. Finally, it will run `gcc file.o -o program` to finish step (1)

Makefile Variables

- Makefiles support *variables* to reduce errors
- Makefile variables always store a string
- Variable names are case-sensitive
  - May not include the following characters: :, #, =, or whitespace
- To define a variable: `variable_name = text`
- Variables are called by placing them in parentheses and preceded by $
  - Ex. `$(variable_name)`
- Example uses of variables include:
  - Specifying compiler flags or compiler type
  - Storing file names
- It is standard practice for every makefile to have a variable named `objects` (equivalently, `OBJECTS`, `objs`, `OBJS`, or `obj`) that contains all your object file names

Cleaning a Directory using Make

- `make` can be used for more than compiling programs
- One common use is for cleaning a directory (e.g. removing some or all files)
- We can use the Linux command `rm` to do this
- Ex:

```
clean:
    rm *.o
```

- In this example, if we call `make clean` in the terminal it will remove all object files in the current directory

Strings

Strings in C

- C does not have a native string datatype
- Instead, a string is seen as an array of characters
- To create a C style string:
  - `char subject[] = "Mathematics";`
  - `char subject[] = {'M', 'a', 't', 'h', '\0'};`
- The last character '\0' is known as a **null character**
  - Indicates when a string is terminated
  - Automatically added for string literals (i.e. strings initialized using double quotes)

String Input in C

- Suppose we create a string: `char string[10];`
- Strings can be read in with `scanf`, which will stop reading once it encounters white space, a newline, or has reached the end of the file: `scanf("%s", string);`
- To read in whitespace characters as part of your stream use `gets`: `gets(string)`
- However both of these approaches should be avoided as they can be problematic and unsafe (e.g. buffer overflow)
- Safer to read in strings using `fgets()`:
  `fgets(string, 10, stdin);`

Printing and Manipulating Strings in C

- To output a string we may use `puts()`: `puts(string)`
  - Automatically appends a newline at the end of the string
- Alternatively, we may use `printf` with specifier `"%s"`: `printf("%s", string);`
  - Argument is interpreted as a formatted string
  - Can be ambiguous if your string contains % characters
  - Can be less efficient than `puts`
  - Does not automatically append a new line
- Many helpful string related functions can be found in the standard library: `#include "string.h"`

Arrays of Strings

- Suppose we would like to store the following strings in an array: `"math", "physics", "CS"`
- How can this be done?
- Naive solution: use a 2D `char` array: `char strings[3][8]`
  - This is not ideal because our string are constrained to a maximum length
  - For strings shorter than the maximum length we are wasting memory
- Better solution is to use an array of `char` pointers: `char *string[3]`
  - Each element is a a pointer that pointers to the first character of each string
  - The strings themselves may not be contiguously stored in memory