CS 246 Fall 2021 — Tutorial 4

September 29, 2021

Summary

1 Preprocessor

- The preprocessor runs before the compiler. It handles all preprocessor directives (any line which begins with `#`).
- Common preprocessor directives:

`#include "file.h"`	inserts the contents of `file.h` by first looking in the current directory, then in the system libraries.
`#define var val`	defines a preprocessor macro `var` with value `val`. If `val` is omitted the value is the empty string.
`#ifdef var`	includes following code if `var` is defined. Must be closed with `#endif`.
`#ifndef var`	similar to `#ifdef`, but includes code if `var` is not defined.

1.1 Include Guards

- As you learnt in CS136, we often want to program in modules that logically separate our code. When we do this, we will often end up including header files in other files so that we have access to functions declared in a module.
- However, we may end up including the same file multiple times in our program, which will likely result in compilation errors.
- To prevent including the same file multiple times, we are going to set up each header file so that it first checks if a unique macro is defined. If it has not yet been defined, we will define the macro and include the contents. If the macro has been defined, we won't include the contents.

2 Make and Makefiles

- With single-file programs, compilation is easy:

```
g++14 change.cc -o change
```

- However, when we have a project across multiple files, compilation may become a pain to type out. Also, recompiling everything becomes quite slow if there are a lot of `.cc` files.
- Separate compilation reduces compilation time by only recompiling files that need to be recompiled because of changes. It looks something like

```
g++14 -c main-vec3d.cc
g++14 -c vec3d.cc
g++14 main-vec3d.o vec3d.o -o vec3d
```

- `make` can automate the build process and avoid unnecessary compilation by keeping track of changed files based upon the last modified timestamp for each file.

  This allows us to specify the target dependencies (the files produced by the build process) and the command for producing each target in a `Makefile`.

  If a target is newer than its dependencies, then there is no need to rebuild this target.

- A `Makefile` will look something like:

```
# example1/Makefile
vec3d: main-vec3d.o vec3d.o
    g++ -std=c++14 main-vec3d.o vec3d.o -o vec3d

main-vec3d.o: main-vec3d.cc vec3d.h
    g++ -std=c++14 -c main-vec3d.cc

vec3d.o: vec3d.cc vec3d.h
    g++ -std=c++14 -c vec3d.cc
```

  Note: The whitespaces before the build command (in this case, `g++...`) **MUST** be a tab.

- On the command line, run `make`. This will build our project.
- If `vec3d.cc` changes, what happens?
- What happens when we execute the command `make`?
- **Tip:** We can also build specific targets using `make`:

```
make vec3d.o
```

- Common practice: add a phony target[1] `clean` to remove all generated files:

[1] "phony target": it is not the name of a file but a recipe to be executed when an explicit request is made. See `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`

```
clean:
    rm *.o vec3d

.PHONY: clean
```

Then to do a full rebuild: `make clean && make`

- We can use makefile variables to simplify maintenance, and `make` has built-in rules using some variables:

```
# example2/Makefile
CXX=g++                                 # compiler command
CXXFLAGS=-std=c++14 -Wall -Werror -g # compiler options to pass
EXEC=vec3d                              # name of executable
OBJECTS=main-vec3d.o vec3d.o            # object files

# This one doesn't have a generic form, so have to list it
${EXEC}: ${OBJECTS}
      ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

# Uses the recipe ${CXX} ${CXXFLAGS} -c main-vec3d.cc -o main-vec3d.o
main-vec3d.o: main-vec3d.cc vec3d.h

vec3d.o: vec3d.cc vec3d.h

clean:
    rm ${OBJECTS} ${EXEC}

.PHONY: clean
```

- The compiler can also generate the dependencies for us so that we don't need to update the Makefile every time we change them.

  Running the command `g++14 -MMD -c vec3d.cc` creates `vec3d.d` that contains

```
vec3d.o: vec3d.cc vec3d.h
```

- This is exactly what we need in our `Makefile`. We just need to include all `.d` files in our `Makefile`. This means our `Makefile` will look like:

```
# example3/Makefile
CXX=g++
CXXFLAGS=-std=c++14 -Wall -Werror -g -MMD
EXEC=vec3d
OBJECTS=main-vec3d.o vec3d.o
DEPENDS=${OBJECTS:.o=.d}

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}
```

```
-include ${DEPENDS}

clean:
    rm ${OBJECTS} ${EXEC} ${DEPENDS}

.PHONY: clean
```

- This is the final version of our `Makefile`. By changing the variables of this `Makefile`, we can use this exact `Makefile` for basically any program we want to create in this course.

3 Example: Matrix of floats

We would like to write a small C++ program that can read in and output a two-dimensional matrix of real numbers.

Let's start by breaking the problem down into what we need to do:

- we need a way to properly initialize the matrix
  - What constitutes an empty matrix?
  - How do we read in the values and store them properly?
- how are we going to output the `Matrix`?
- finally, we need to free all of our dynamically-allocated memory

4 Tips of the Week: Preprocessor Debug Messages

- While programming, we will often want to debug by using print statements to check if program is doing what we expect it to do. However, if we forget to comment out the print statment afterwards, our program will not do what we expect.
- One simple way around this it to wrap the print statements in a preprocessor macro.

```
#ifdef DEBUG
    cerr << "Testing debug mode" << endl;
#endif
```

- It can become cumbersome to wrap each statement like this. Another approach is to write a function which will be called for debugging purposes.

```
void debug(const string &s) {
#ifdef DEBUG
    cerr << s << endl;
#endif
```

```
}
```

This function could be overloaded to handle any built in class or struct you define. You will be able to turn on debugging by compiling with the `-DDEBUG` flag.