

CS 246 Fall 2018 — Tutorial 3

October 3, 2018

Summary

1	Operator Overloading	1
2	Preprocessor	2
3	Make and Makefiles	2
4	Tips of the Week: Preprocessor Debug Messages	5

1 Operator Overloading

- We can overload many of the built in operators including the arithmetic operators.
- This is often useful when implementing our own structures.
- When overloading operators, we should give our operators logical meanings
 - The meanings should be intuitive, e.g. `operator+` should do something that resembles addition.
 - If an operator is overloaded, consider overloading other operators associated with it, e.g. if `operator+` is defined then consider also defining `operator+=`
 - If an operator is commutative, don't forget to define the alternative arrangement (see `lectures/c++/operators/vectors.cc`)

1.1 Cascading

- We've gotten used to seeing code that looks like the following:

```
int num = 5;
cout << "The magic number is " << num * num - num << endl;
```

- We see that `num * num - num` execute `num * num` and the result will have `num` subtracted from it. What is happening here is that `operator*` is being called which returns an int, then `operator-` is being called and returns the int which is printed.
- In order for cascading to allow modifications to the previously returned value, the result must be returned by reference
- Thus, `operator<<` returns a reference to the ostream (for `operator>>`, returns a reference to the istream) it was given, which is used later in the expression.

- When we write operators, we want our operators to behave in a similar fashion. This means that our operators will return the result of the operation.

2 Preprocessor

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

<code>#include "file.h"</code>	inserts the contents of <code>file.h</code>
<code>#define var val</code>	defines a preprocessor macro <code>var</code> with value <code>val</code> . If <code>val</code> is omitted the value is the empty string
<code>#ifdef var</code>	includes following code if <code>var</code> is defined. Must be closed with <code>#endif</code>
<code>#ifndef var</code>	similar to <code>#ifdef</code> , but includes code if <code>var</code> is not defined

2.1 Include Guards

- As you learnt in CS136, we often want to program in modules which 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 setup 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.
- See `includeGuards` directory for this tutorial for an example.

3 Make and Makefiles

- With single-file programs, compilation is a breeze:

```
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 building process and avoid unnecessary compilation by keeping track of changed files based on last modified time.

It allows us to specify the dependencies of targets (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?
 - compile `vec3d.cc`
 - relink `vec3d`
- What happens when we execute the command **make**?
 - Tries to build the first target in our **Makefile**, in this case `vec3d`.
 - What does `vec3d` depend on? `main-vec3d.o vec3d.o`
 - Since `vec3d.cc` changes:
 - `vec3d.cc` is newer (timestamp) than `vec3d.o`; rebuilds `vec3d.o`
 - `vec3d.o` is newer (timestamp) than `vec3d`; rebuilds `vec3d`
- **Tip:** We can also build specific targets using **make**:


```
make vec3d.o
```

- Common practice: add a phony target¹ `clean` to remove all generated files:

¹“phony target”: it is not 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` will create `vec3d.d` which 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.

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 statement afterwards, our program will not do what we expect.
- One simple way around this is 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.