CS 246 Spring 2017 - Tutorial 6

June 14, 2017

1 Summary

- Makefiles
- Copy Constructors
- Destructors
- Copy Assignment Operators
- Lvalues and Rvalues
- Move Constructors
- Move Assignment Operators
- The Rule of Five

2 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. Surely there is a better way to compile a project without typing all `.cc` files.
- We've told you that you should use separate compilation which looks something like

```
g++14 -c main.cc
g++14 -c book.cc
...
g++14 book.o main.o textbook.o ... -o main
```

- When we do this, we only have to recompile the modules that change. This means less time compiling but more time remembering what we have recently compiled.
- Surely there must be a better way to keep track of changes. This is a bigger issue in a group when we would constantly be recompiling everything when we don't have to.

- Linux can help with the `make` command. Create a `Makefile` that outlines which files depend on each other. It will look something like:

```
#means main depends on these
main: main.o book.o textbook.o comic.o
	#specifies how to build main
	g++ -std=c++14 main.o book.o textbook.o comic.o -o main
book.o: book.cc book.h
	g++ -std=c++14 -c book.cc
textbook.o: textbook.cc textbook.h book.h
	g++ -std=c++14 -c textbook.cc
comic.o: comic.cc comic.h book.h
	g++ -std=c++14 -c comic.cc
main.o: book.h textbook.h comic.h main.cc
	g++ -std=c++14 -c main.cc
```

- The whitespaces at the beginning of the line **MUST** be a tab.
- On the command line, run `make`. This will build our project.
- If `book.cc` changes, what happens?
  - compile `book.cc`
  - relink `main`
- What happens when we execute the command `make`?
  - Builds first target in our Makefile, in this case `main`.
  - What does `main` depend on?
    - `book.o textbook.o comic.o main.o`
  - If `book.cc` changes:
    - `book.cc` is newer (timestamp) than `book.o`; rebuilds `book.o`
    - `book.o` is newer (timestamp) than `main`; rebuilds `main`
- **Tip**: We can build specific targets

```
make textbook.o
```

- Common practice: put a `clean` target at the end of a makefile to remove all binaries[1]

```
...
clean:
	rm *.o main
# clean is a ``phony target'': it is not name of a file but
# a recipe to be executed when an explicit request is made
.PHONY: clean
```

[1]The description is found in `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`

- To do a full rebuild:

```
make clean && make
```

- We can also write phony targets to redo all tests (i.e. produceOutputs and runSuite)
- We can generalize a `Makefile` with variables:

```
CXX=g++                          #compiler name
CXXFLAGS=-std=c++14 -Wall -Werror -lX11   #options to pass
...
book.o: book.cc book.h
    ${CXX} ${CXXFLAG} -c book.cc
...
```

- **Shortcut:** For any rule of the form `x.o: x.cc a.h b.h`, we can leave out the build command. `make` will guess that it is `${CXX} ${CXXFLAGS} -c book.cc -o book.o`
- Issue: how to track dependencies and updating them as they change
  - g++ can help. `g++ -std=c++14 -MMD -c comic.cc` will create `comic.o comic.d`.
  - What will `comic.d` contain?

    ```
    comic.o: comic.cc book.h comic.h
    ```

- Looking at this `.d` file, we can see it 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

```
CXX=g++
CXXFLAGS=-std=c++14 -Wall -Werror -MMD
OBJECTS=main.o book.o textbook.o comic.o
DEPENDS=${OBJECTS:.o=.d}
EXEC=main

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

-include ${DEPENDS}

clean:
    rm *.o main
.PHONY: clean
```

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

3 Copy Constructors

- From the last tutorial, we have created some constructors for the `Node` class. But there are some memory issues that still needs to be addressed.
- What happens when we run the following code?

```
Node *n1 = new Node{7};
n1->add(3);
n1->add(5);
n1->add(10);
Node *n2 = new Node{*n1};

delete n1;
n1 = nullptr;
cout << n2 << endl;
delete n2;
```

- When running this code, we see that `n2` now has garbage values in some of its nodes. Why?
- The pointers are now dangling pointers because the pointers were deleted when we deleted `n1`.
- By default, the copy constructor copies all fields *by value*. This means that when pointers are copied, the address stored in the original pointer is copied to the new pointer and **the objects share memory**. This is known as a *shallow copy*.
- We do not want these objects to be sharing memory. We want the copy constructor to copy all Nodes and not just the first Node (This is known as a *deep copy*).
- To do this, we need to implement our own copy constructor.

```
Node(const Node &n): value{n.value},
                     next{n.next ? new Node{*(n.next)} : nullptr} {
    if (next)
        next->prev = this;
}
```

- The copy constructor is called in three situations:
  - When initializing an object from another object.
  - When passing an object by value.
  - When returning an object by value.

4 Destructor

- The destructor is a method which is called when a object is destroyed.
- A default destructor is provided for us by the compiler. This destructor will call the destructors for all fields that are objects. However, it does not call `delete` on fields which are pointers.
- This means that in our example above, the `data` which `next` points at will not be freed when the `Node` is destroyed.
- We need to write our own destructor in this case. For the `Node` class:

```
~Node(){
    delete next;
}
```

- Why are we not deleting `prev` here?
- Why don't we set next to `nullptr`?
- Note: similar to constructors, destructors do not have a return type.
- Note: Destructors also take no parameters

5 Copy Assignment Operator

- What happens when we run the following code?

```
    Node* n1 = new Node{7};
    n1->add(3);
    n1->add(5);
    n1->add(10);
    Node* n2 = new Node{8};
    n2->add(12);
    *n2 = *n1;

    delete n1;
    n1 = nullptr;
    cout << *n2 << endl;
delete n2;
```

- Similar to the copy constructor, we get garbage values printed.
- The reason is similar to the case with the copy constructor: The compiler gives a default version of the copy assignment operator but this method also performs a shallow copy, not a deep copy.
- If we wanted a deep copy for the copy assignment operator, we need to define our own:

```
Node& operator=(const Node &n) {
    if ( this != &n ){
        Node copy = n;

        // swap the value of variables
        // defined in <utility>
        std::swap(next, copy.next);

        value = copy.value;
        next->prev = this;
    }
    return *this;
}
```

- The way this assignment operator is written is referred to the *copy-and-swap* idiom.
  - We are first creating a local **copy** of the node we are copying. This calls the copy constructor.
  - We then **swap** our `next` pointer with the `next` pointer of the Node `copy`. This means that when `copy` goes out of scope, our old data will be deleted by the destructor.

6 Lvalues and Rvalues

- An *lvalue* is any entity which has an address explicitly accessible by the program. They get their name because an lvalue is a value which can occur on the left side of an expression.
- An lvalue reference (often just called a *reference*): &
- An *rvalue* is anything which is not an lvalue. They get their name because an rvalue can only occur on the right side of an expression.
- An rvalue reference: &&
- Note: a rvalue may also be used to initialize a const lvalue reference.

7 Move Constructor

- Suppose we have the following function:

```
Node plus(Node n, int inc){
    for (Node *m = &n; m != nullptr; m = m->next) {
        m->value += inc;
    }
    return n;
}
```

- When we run this function, the copy contructor will be run to make a copy of the node which we pass as a parameter and then every node after that. It then returns a temporary object.

- Now, if we have something like `Node n2 = plus(n1,2)`, what happens?
  - The copy constructor will run to create `n2` from the object returned from `plus`.
  - However, this is a temporary object (rvalue) which will be destroyed as soon as we are done copying it.
- **Idea:** we should steal the data which is about to be thrown away instead of creating a copy of the data.
- How can we do that? Since we know we have an rvalue, we should write a constructor which takes an rvalue reference and then moves the value in (instead of copying).

```
Node(Node &&n): value{n.value}, next{n.next} {
    if (next) next->prev = this;
    n.next = nullptr;
}
```

- If a copy constructor (i.e. with argument of const lvalue reference) and a move constructor are both present in a class definition, passing a rvalue reference of an instance of the class into the constructor call will invoke the move constructor.
- **Important note:** we must set all pointers which will be deleted by the destructor to be `nullptr` or the destructor will delete the data we stole when the temporary object goes out of scope.

8 Move Assignment Operator

- Similar to the move copy constructor, we may want to have the following:

```
Node n1{3,7};
Node n2{1,9};

n2 = plus{n1, 2};
```

- We want to make an assignment operator which take rvalues as well.

```
    Node &operator=(Node &&n) {
        value = n.value;
        swap(n.next, next);
        next->prev = this;
        return *this;
}
```

9 Rule of Five

The Rule of Five: If you have to write one of the following:

- copy constructor
- move constructor
- copy assignment operator
- move assignment operator
- destructor

You **should probably** write all five. [2]

Why? You typically have to write each if you are dealing with non-contiguous memory.

[2] Think about when it's not neccesary to write all five.