# CS 246 Fall 2015 - Tutorial 9

November 12, 2015

## 1 Summary

- Makefiles
- Inheritance and Virtual Methods

## 2 Makefiles

- By now, you've probably realized that you recompile code...a lot
- We've told you that you should use separate compilation which looks something like

```
g++ -c main.cc
g++ -c book.cc
...
g++ 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 than mentally or else when working in a group, we'd constantly be recompiling code 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:

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

- This whitespace **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
- Command make - builds first target in our Makefile, in this case main.
- What does main depend on?
  - book.o, textbook.o, comicbook.o, main.o
- If book changes:
  - book.cc is newer (timestamp) than book.o, rebuilds book.o

  - book.o is newer (timestamp) than main, rebuilds main

- Tip: can build specific targets: make textbook.o
- Common practice: put a clean target at the end of a makefile to remove all binaries

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

- To do a full rebuild:

```
make clean
make
```

We can generalize a makefile with variables:

```
CXX=g++             #compiler name
CXXFLAGS= -Wall -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: tracking dependencies and updating them as they change. G++ can help with `g++ -MMD -c comicbook.cc` will create `comicbook.o comicbook.d`. What will comicbook.d contain?

```
comicbook.o: comicbook.cc book.h comicbook.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 file in our Makefile. This means our makefile will look like

```
CXX=g++
CXXFLAGS=-Wall -MMD
OBJECTS=main.o book.o textbook.o comicbook.o
DEPENDS=${OBJECTS:.o=.d}
EXEC=main
${EXEC}: ${OBJECTS}
        ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}
-include ${DEPENDS}
clean:
        ...
```

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

# 3 Inheritance

- Why do we need inheritance?
  - We want an object which solves a task, but are happy to let the user of our code modify the object as long as it still implements the basic functionality.
  - Similar classes may duplicate a great deal of code, whereas inheritance can minimize this by instead causing them to inherit shared logic from a common ancestor.
  - We don't care about implementation details as long as the class is able to give us the answer we need.

- Different uses for inheritance
  - Inherit from a class which does almost exactly what we want, but change a method or two
  - Create an abstract interface to a number of similar classes that solve the same problem in different ways: ADTs (for instance, we could use a binary tree, an array, or an association list to store data indexed by integers)
  - Create a specialized version of a class that solves a problem.
  - Allow one person on a team of programmers to specify a set of rules that a class ought to follow, while others implement actual classes which follow these rules and solve the desired problems.
- Example: Trees

```
class Tree{
  private:
    int data;

    ...
};

class BTree : public Tree {
  private:
    ...
};
```

- `BTree` is-a Tree and we can use it wherever we could use a Tree
  - **Warning:** Its behaviour may not be what we expect
- Remeber that subclasses can't see any private members of super classes, so BTree can't access the `data` field.
- How can we fix this?
  - `Public get method`
  - `Friend class`
  - `Protected` visibility: only objects of the same type, friends, or derived classes can access these members

```
class Tree{
  private:
    int data:

  public:
    int getData();
};
```

- Now let us consider the following example:

```
#include <iostream>
using namespace std;

class Computer {
  public:
    void makeCall() { cout << "Making call through the power of the internet" << endl; }
    void test() { cout << "Dialing out" << endl; }
}

class Smartphone : public Computer {
  void makeCall() { cout << "Attempting to make a call through Wind Mobile" << endl; }
}
```

```
void testCall(Computer& c){
  c.test();
  c.makeCall();
}
int main(){
  Smartphone Nexus6;
  testCall(Nexus6);
  Computer * phone = new Smartphone;
  phone->makeCall();
  Nexus4.makeCall();
  Nexus4.test();
}
```

- The wrong `makeCall` is being called! Why?
- Okay, we can use `virtual` to fix this!
  - Just need to make `makeCall` `virtual` in Computer base class
  - Once a method is `virtual` then it is virtual in any derived classes
  - Though it is often useful to include virtual in definitions of derived classes
- To summarize:

| | Virtual Method | Non-Virtual Method |
|---|---|---|
| Base Class Object | Base Class Method | Base Class Method |
| Derived Class Pointer | Derived Class Method | Derived Class Method |
| Base Class Pointer to Base Class | Base Class Method | Base Class Method |
| Base Class Pointer to Derived Class | Derived Class Method | Base Class Method |

- See animal.cc example
- If the subclass method is not defined, then the superclass method is called regardless of the static or dynamic type of the variable
- Okay, `virtual` is useful but how useful?
- Let's make a general purpose class.

```
class Object{
};
class MyObject : public Object {
  int * arr;
  public:
    MyObject(): arr(new int[20]){}
    ~MyObject(){ delete [] arr;}
};
int main(){
  Object * o = new MyObject;
  // Use o
  // ...
  // Clean up
  delete o;
}
```

- This compiles and runs fine. Except for one thing, what is it?
- So we need to ensure the appropriate destructor is called through a polymorphic pointer.
- Once again, we use `virtual` to do this.
- Whenever we want to allow the usage of a base class as a polymorphic pointer then we **need** to make the destructor virtual

- Otherwise, we could cause memory leaks
- **Note/Foreshadowing**: This is why you should not inherit from STL containers (`vector`, `list`, etc)