CS 246 Winter 2014 - Tutorial 8

November 5, 2014

1 Summary

- Makefiles
- Inheritance

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 out makefile. Altering the variables of this Makefile, we can use this exact make file for basically any program we want to create.

3 Inheritance

- Public inheritance specifies an 'is-a' relationship

```
struct Tree{
  ...
 private:
  int data;
};

struct BTree : public Tree {
  ...
};
```

- `BTree` is-a Tree and we can use it wherever we could use a Tree
  - **Warning:** What you expect to happen may not be what happens

- Remeber that subclasses can't see any private members of super classes
  - So BTree can't access the field called `data`
- What's a way to fix this?
  - `Public get method`: allows any one to access data (may be bad)
  - `Protected` visibility: only objects of the same type, friends, or derived classes can access these members

```
struct Tree{
  ...
 protected:
  int data;
};
```

- Now let us consider the following example:

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

struct Computer{
  void makeCall(){ cout << "Making call through the power of the internet\n";}
  void test(){cout << "Dialing out\n";}
};
struct Smartphone : public Computer {
  void makeCall(){ cout << "Attempting to make a call through Rogers\n";}
};
void testCall(Computer& c){
  c.test();
  c.makeCall();
}
int main(){
  Smartphone Nexus4;
  testCall(Nexus4);
  Computer * laptop = new Smartphone;
  laptop->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
SuperClass Object	SuperClass Method	SuperClass Method
SubClass Pointer	SubClass Method	SubClass Method
SuperClass Pointer to SuperClass	SuperClass Method	SuperClass Method
SuperClass Pointer to SubClass	SubClass Method	SuperClass Method

- See animal.cc example
- If the subclass method is not defined, then the superclass method is called regardless of whether or not it's an object or a pointer.
- Okay, `virtual` is useful but how useful?
- Let's make a general purpose class.

```
struct Object{
};
struct MyObject : public Object {
  int * arr;
  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.
- We use our good buddy `virtual` to do this. See `object-fixed.cpp`
- 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)