CS 246 Fall 2013 - Tutorial 7

November 5, 2013

1 Summary

- Inheritance
- Midterm Review

2 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
- 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 approrpiate 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)
- Sometimes we want to specify a class that cannot be instantiated (e.g. it's mainly going to be used polymorphically)
- Such as a class is called an `abstract class` or sometimes an `interface` (but typically not in C++)
- How do we make an `abstract class` in C++?
  - By having a pure virtual method

```
struct AbstractClass{
  ...
  virtual void someMethod()=0;
};
```

- A pure virtual method must be implemented in any derived classes or else those classes are also abstract
- **Note**: Pure virtual methods can have an implementation in an abstraction class
  - For example, a pure virtual destructor still needs an implementation. Otherwise, privately allocated memory could be leaked.
  - Note that in such cases the pure virtual method definitions cannot be done in the class definition but must be done outside of it.

```
class AbstractClass{
  int * data;
 protected:
  AbstractClass():data(new int[10]){}
  virtual ~AbstractClass()=0;
};
AbstractClass::~AbstractClass(){delete [] data;}
```

3 Midterm Review

Done in tutorial. No answers posted. Brief discussion about `delete` vs `delete`.