CS 213 -- Lecture #14

"Late Night Guide to C++"
Chapter 9 pg 222 - 239
MORE ABOUT CLASSES
Part I

Administrative...

- We're starting to catch up!

Interfaces in C++

- For those with Java experience, you know that Java implements the concept of an *interface.*
- An interface is a class which cannot exist by itself, its sole purpose is to provide a set of methods which *must* be implemented by any class that derives from it.
- It also provides a way for other code to work with derived classes of the interface (both currently defined and "to be defined") without needing to know any of the details of the derived class. (Assign #4)
- In Java, an interface declaration looks like this:

```
// WARNING! WARNING!  What follows is JAVA code!
public interface Comparable
{
  public int compareTo(Object b);
}
```

Interfaces in C++ (cont)

- Any Java class may *implement* the interface by the addition of one keyword and by providing definitions for each of the methods defined in the interface.

```
// WARNING! WARNING!  What follows is JAVA code!
public Rectangle implements Comparable
{
  public int compareTo(Object b)
  { … }                          // definition goes here
}
```

- So where did the Java engineers get the idea for the `interface` and `implements` keywords?
- By looking at C++, no doubt!

Interfaces in C++ (cont)

- In C++ there are no separate keywords corresponding to the `interface` and `implements` keywords in Java.
- This makes interfaces more of a concept than a language construct as far as C++ goes.
- The same concept of an "interface" is implemented in C++ by using abstract classes with pure virtual member functions.

```
class Comparable
{
  virtual int compareTo(Comparable &c)=0;
};
class Rectangle : public Comparable
{ // Incomplete definition...
  int compareTo(Comparable &c)
  { … }
}
```

Interfaces in C++ (cont)

- Since there are no keywords which allow you to specify an interface in C++, the following are *suggestions* and not *requirements*…
- Interfaces are not used as standalone objects. This implies that there is at least one pure virtual member function.
- Interfaces are used to define (abstractly) a set of member functions. Other functions/objects will be utilizing the set of member functions defined in the interface. To implement the interface, all member functions must be defined in the derived class.
- What this buys us is the ability to create truly "pluggable" objects which will require minimal recompile time.
- An example of this is a simple message box object. You can have multiple objects (graphical dialog box, text messages, etc.) which all derive from one interface. They may then be used interchangeably...

Demonstration

Interfaces in C++

Private Inheritance

- When looking at inheritance, we've always used a declaration of the following form:

```
class X : public Y
{
…
};
```

- I've asked you to take it on faith that `public Y` is simply the syntax you must use to say that the class `X` is derived from class `Y`.
- Now, consider the following partial definitions of `X` and Y...

Private Inheritance (cont)

```
class Y
{
public:
  int a,b;
protected:
  int c,d;
};

class X : public Y
{
public:
  int h,i;
};
```

- As we've gone over before, a member function in class `X` has access to member variables `a,b,c,d,h,i` from the base class `Y`.
- When working with `X` outside of the class, `a,b,h,i` are accessible.

Private Inheritance (cont)

```
class Y
{
public:
  int a,b;
protected:
  int c,d;
};

class X : private Y        // Notice the change to private
{
public:
  int h,i;
};
```

- But, it we make use of *private inheritance*, the public (and protected) members of the base class become *private members* of the derived class.

Private Inheritance (cont)

- That is to say that no members from a privately inherited base class may be accessed from outside the scope of the derived class.
- It also means that the relationship `X` *is a* `Y` or `X` *is a kind of* `Y` doesn't really hold up here.
- Why? Because if a `Y` has certain public methods and `X` is a `Y`, then `X` should have the same public methods.
- With private inheritance this is not the case, as the public methods in `Y` are not public methods in `X`.
- So, then, what is private inheritance good for?
- LNG suggests the following (from pg. 231):
  - This is what private inheritance is for. If A is to be implemented as a B, and if class B has virtual functions that A can usefully override, make A a private base class of B.
- Is this really useful? Wait… before you decide...

Private Inheritance (cont)

- A pointer to a class derived privately from a base class may not be assigned directly to a pointer to the base class.

```
X x;         // declare an instance of the class X
Y *yPtr;     // pointer to an instance of base class
yPtr = &x;   // COMPILER ERROR. access violation
```

- Ouch! We've been able to do this before but now the compiler won't let us :-(.
- Oh, wait, this is C++, it will let me do almost *anything* with a little "coercion"

```
X x;              // declare an instance of the class X
Y *yPtr;          // pointer to an instance of base class
yPtr = (Y *)&x;   // Oh, ok, this is fine
```

Private Inheritance (cont)

```
class Y
{
public:
  void doSomethingUseful(int);  // arbitrary public member func
  int a,b;
};

class X : private Y
{
public:
  int h,i;
};
```

- Since `doSomethingUseful()` is a public member of `Y`, it is not accessible outside of the scope of `X`. That is...

Private Inheritance (cont)

```
int main()
{
  X anX;

  anX.doSomethingUseful(5);     // access violation
  return 0;
}
```

- Attempting to access `doSomethingUseful()` from a variable of class `X` is an access violation.
- That's because `doSomethingUseful()` is a public method of a privately inherited base class.
- Bummer.
- Oh, wait, this is C++… Can I coerce my way around this restriction?
- You betcha!

Private Inheritance (cont)

```
class Y
{
public:
  void doSomethingUseful(int);  // arbitrary public member func
  int a,b;
};

class X : private Y
{
public:
  Y::doSomethingUseful;
  int h,i;
};
```

- Notice the bizarre syntax in the public section of `Y`.
- It is termed the *fully qualified name* of the member function `doSomethingUseful()` defined in class `Y`.

Private Inheritance (cont)

```
int main()
{
  X anX;

  anX.doSomethingUseful(5);     // now this is ok
  return 0;
}
```

- The addition of the line `Y::doSomethingUseful;` into the public section of the class definition of `X` solved our problem.
- Notice that only the name of the member function from the base class to be made public is used, there is no parameter list.
- So, back to my original question… Is it useful?
- Let's see it in action first...

Demonstration

Private Inheritance

Private Inheritance -- Is it Useful?

- Well, put a feature in a language and *someone* will find a way to use it!
- In my programming travels I have never seen it used.
- It's not for doing straight inheritance, because the relationships break down.
- It's not for doing interfaces since the pure virtual approach is much cleaner, simpler, and enforces the implementation of all members.
- The example given in the books shows a case where a generic base class is used to provide functionality to a more specific derived class.
  - the derived class is, conceptually, different from the base class
    - List vs. Stack
  - Many of the features of the base class might not be applicable to the derived class.
    - Removing the nth element of a list is not a stack-like function.
  - In this case, private inheritance may be appropriate.

Declarations in Classes: Enumerations

- Since the class definition also provides a sense of "scope", we can perform declarations and definitions within the class definition.
- Indeed, we do just that with member functions and member variables.
- But there are other types that can be declared and defined as well.
- We'll look at enumerations first.

```
class Example
{
public:
  void setCounter(int argCounter=kInitialValue);
private:
  enum CounterValues {
    kInitialValue=0,  kLastValue
  };
  int counter;
}
```

Declarations in Classes: Enumerations (cont)

- The constants defined in the enumeration may be used anywhere in the class, and in other classes by the following rules:
- If the enumerations are declared protected, they may be used only in derived classes.
- If the enumerations are declared public, they may be accessed from anywhere provided their fully qualified names are used:

```
int main()
{
  Example anExample;

  // In order to use the enumeration defined in Example
  // we must qualify it with the class name (Example::)
  anExample.setCounter(Example::kLastValue);
}
```

Declarations in Classes: Static Members

- A static member variable is a variable declared in the class definition but is treated like a global variable:
  - There is only one copy of it no matter how many instances of its class exist
  - If its value is changed by any one instance, the value is changed for all the instances
- In effect, it is a global variable but only visible outside the class if declared in the `public` section.

```
class Example
{
public:
  Example() { currentTotal = 0; }
  void addToTotal(int aValue);
private:
  static int currentTotal;
};
```

Static Members (cont)

- There's only one problem…
- Since the static member doesn't really exist in the class, separate storage must be allocated for it outside the class.
- This is done in the form of a global variable using the fully qualified name of the static member...

```
class Example
{
public:
  Example() { currentTotal = 0; }
  void addToTotal(int aValue);
private:
  static int currentTotal;
};

int Example::currentTotal = 0;  // Define the static member
```

Static Members (cont)

- If you don't define space for the static member in this manner, you will get a linker error!
- The static member may be accessed outside of the class only if it is declared in one of the class' `public` sections.
- If so, the static member may be accessed anywhere by using its fully qualified name:

```
int main()
{
  Example e1;

  Example::currentTotal = 5;        // There are two ways to
  cout << e1.currentTotal << endl;  // access the same var!

  return 0;
}
```

Static Members (cont)

- A static member function is a little different since there is already only one copy of the code used to implement a member function.
- A static member function has no `this` pointer and cannot access individual non-static member fields.
- However a static member function may be called without creating an instance of the class in which it is defined...

```
class Example
{
public:
  static void printBanner() { cout << "Hello World" << endl; }
};
int main()
{
  Example::printBanner();
  return 0;
}
```

Demonstration

Declarations in Classes

Final Thoughts

- Assignment #7 will be posted on Thursday
- Only 11 assignments will be given