CS 246 - Object Oriented Programming **Fall 2016**

Lecture 12: December 1, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

Recall :

```
Text t {...}'
Book &b = t;

Text &t2 = dynamic_cast<Text &> (b);
//If B "points to " a Text, then t2 is a reference to the same text.
//If not, then a exception bad_cast is raised
```

With dynamic casting, we can solve the polymorphic assignment problem.

```
Text &Text::operator=(const Book &other) { // Virtual
  Text &textother = dynamic_cast<Text &> (other); //Throws an exception if other is not a
      text, which can be solved. Thus solving the polymorphic assignment problem.
  if (this == &textother) return *this;
  Book::operator=(other);
  topic = textother.topic;
  return *this;
}
```

12.1 How Virtual Methods Work

Consider :

```
class Vec {
  int x, y;
  int doSomething();
  ...
};

class Vec2{
  int x, y;
  virtual int doSomething();
  ...
}

//Client
Vec v {1,2};
Vec w {1,2}'

cout << sizeof(v) << " " << sizeof(w) << endl; //Returns 8 for v and 16 for w
```

Note : 8 is the size of two integers, so there is no space for functions. Therefore the compiler turns methods into regular functions and stores them outside the object with other functions.

Now Recall :

```
Book *pb = new (Book/Text/or Comic)
pb->isItHeavy();
```

Since the function is virtual, the compiler bases what function to call based on the type of thw actual object, which the compiler can not know in advance. So, the correct version of the function must be chosen at runtime.

To achieve the selection of the correct virtual function, the compiler creates a table of function pointers **(The vtable)** for each class with virtual methods. Then a pointer called the **vpointer** , which points to the **vtable**, is added to each virtual class. It also worth noting that the vtable is only created once, then included in each version of the class. In addition, each different type of class will have its own **vtable**.

Lastly, each **vtable** also starts with some type of value to help indicate what class the table for. This is done to allow for dynamic casting, which is a very expensive operation.

To determine the correct function, the compiler fetches the **vpointer** and continually follows the path till the correct function is found.

Steps that occur when calling a virtual method

1. Program follows **vpointer** to **vtable**
2. Program fetches pointer to the actual method from the table
3. Program follows the function pointer to call the function.

Therefore, virtual function calls incur a small overhead cost in addition to the increased memory requirements.

Going back to the first example regarding the 8 byte difference, its evident that the addition of the **vpointer** adds an extra 8 bytes to enable the selection of the correct function.

12.1.1 How is an object concretely laid out

Concretely, in most compilers, objects are laid out in a way such that the **vpointer** is always at the top to ensure subclass objects are similar to the superclass if you trim the new values. In addition, having the **vpointer** at the top makes its easier to locate, thus avoiding any issues.

12.2 Multiple Inheritance

A class can inherit from more than one class

```
1  class A {
2    public :
3      int a;
4  };
5
6  class B {
7    public :
8      int b;
9  };
10
11 class C : public A, public B{
12   void f() { cout << a << b << endl;} //F can access A and B fields.
13 };
```

Now consider :

```
Class D : public B, public C { //Both B and C have a variable A inherited from A
  public :
    int d;
};

D d;
cout << d.a;;
```

This will not compile as there is ambiguity to which variable a should be called. To fix this we can explicitly state which a variable you want use the scope operator.

```
cout << d.B::a << " " << d.C::a << endl;
```

Now, since B and C inherit from another class A and you want only one copy A, you can utilize the pattern called the **deadly diamond pattern**. To achieve this you make the base class virtual, so in essence you invoke virtual inheritance.

```
class B : virtual public A { ... };
class C : virtual public A { ... };
```

Note : A good example of virtual inheritance within C++ is the iostream class.

Unlike regular inheritence, the distance between the actual objects and their superclass in the model of object is not consistent. So, to find the base classes, the program will go to one of the intermediate classes such as B, and obtain the super class information from there. In addition, the diagram for layout does not look like A,B,C,D, simultaneously. It looks like slices of the classes instead.

So,

- pointer assignment amoung A, B, C, D changes the address in the pointer.
- static/dynamic/const cast under multi inheritance also changes the value of the pointer
- reinterpret cast does not work.

12.3 Template Functions

Consider :

```
template <typename T> T min (T x, T y) {
  return x < y ? x : y;
}

int x = 1, y = 2;
int z = min(x, y);
```

As noted in the example, explicitly stating the type of x and y when using the function template is not necessary, as the compiler is smart enough to realize the type automatically. This does not work for certain types, thus a explicit type may be needed.

Recall : the following works as long as AbstractIterator supports *,!,=,++ and can be called as a function

```
void for_each(AbstractIterator start, abstractIterator finish, void (*f) (int)){
  while (start != finish){
    f(*start);
```

```
4       ++start;
5     }
6   }
```

We can achieve the same result using functions templates while ensuring it can work with any types that does not inherit from abtractiterator, thus resulting in more generality.

```
template <typename Iterm typename Func>
  void for_each(Iter start, Iter finish, Func f) {
    while (start != finish) {
      f(*start);
      ++start
    }
  }
```

12.3.1 STL ⟨algorithm⟩

`<algorithm>` is a suite of template functions, many of which work over iterators

Example 12.1 *foreach (as given above) is included in the algorithm library.*

Example 12.2 *Find is also included in the algorithm, which eliminates the need for loops to find values. Find returns iterator to the first item that matches val and if not found returns last.*

```
template <typename Iter, typename T>
  Iter find (Iter first , Iter last, const T &val) {
    ...
  }
```

Example 12.3 *Count is much like find, but returns the number of occurrences of the given value.*

Example 12.4 *Copy is another common template included within algorithm which copies one container range to another starting at result . It may be worth noting that copy does not allocate memory.*

```
template <typename InIter, typename OutIter>
  OutIter copy (Initer first, Initer last, outIter result) {
    ...
  }
```

Example 12.5 *Transform is another template which works alot like copy but also applies a function*

```
template <typename InIter, typename OutIter, typename Func>
  OutIter copy (Initer first, Initer last, outIter result, func f) {
    while (first != last){
      *result = f(*first);
      ++first;
      ++result;
    }
  }
```

- *func could be regular objects with an overloaded () operator , which will allow for state to be maintained.*
- *func could also be a lambda which is indicated used open/close brackets* `[](int n) {return n % 2 == 0;}`
- *You can also avoid having to manage memory by utilizing* `back_inserter(w)`,

That is all for Fall 2016. Hope these notes helped!