Discussion on some items from Effective C++

37: Redefining inherited function’s default values

- This is talking about the scenario where a class overrides a virtual function defined in one of its base classes, ie, not necessarily the immediate parent, but any parent class in its inheritance hierarchy.

- The point here is that you should not provide a different value for any default parameter values of this inherited (and now overridden) virtual method.

- This is because the default values of parameters are bound at compile time(static binding, or early binding), although the call to the virtual function is bound at run time (dynamic binding, or late binding)

37

- Now, this implies that your overridden virtual method in the derived class should use the same default value for the parameter(s) that its base class uses.

- That is fine, but this can lead to a maintenance issue, where at some point in the future, the default value of the parameter in the base class method is changed, and <u>**this value in the derived class(es)**</u> also needs to be changed.

- There is a way to avoid this problem.

37

- Looking at the example in the book:

- A non-virtual method called Draw is defined in the base class.
- Draw specifies default parameter value(s).
- Draw is just a wrapper for the pure virtual function DoDraw that a derived class will override.
- DoDraw is not defined as taking default parameters.
- So, the maintenance issue for default parameters along the class inheritance hierarchy is taken care of.

```
Base::Draw( ParamType param1 = param1DefaultValue )
{
        DoDraw( param1 );
}
Derived1::DoDraw( ParamType param1 ) { … }

Derived2::DoDraw( ParamType param1 ) { … }
```

42

- typename vs class

- When using these keywords inside templates, you can use either one.
- Example:

```
template <typename T>
class MyClass { … }
```

- OR

```
template <classT>
class MyClass { … }
```

- However, sometimes you have to define a variable of a *dependent type* inside the class or class methods.
- This is when you have to use typename (and cannot use class).

42

Example from book:

```
template <typename Container>          // or can say: template < class Container >
void Foo( const Container & c )    {
          Container::iterator  iter( c.begin() );
}
```

OR

```
template <typename Container>
void Foo( const Container & c )    {
           // intent is to define a pointer of type iterator
          Container::iterator * iter;
}
```

But the confusion is because Container class could have a static data member named **iterator**, and this would look like you are multiplying that static data member with **iter**. Compilers have to support the general case.

42

- So the solution is to use **typename** keyword here to tell the compiler that the identifier following it refers to a type.

```
template <typename Container>      // or can say: template < class Container >
void Foo( const Container & c )    {
          typename Container::iterator iter( c.begin() );
}
```

OR

```
template <typename Container>
void Foo( const Container & c )   {
           // intent is to define a pointer of type iterator
           typename Container::iterator * iter;
}
```

42

- You don't have to use typename in the base class list (when you specify your class inheritance).

- You also don't have to use typename in the initialization list of a constructor.

44

Factoring parameter independent code out of templates.

- This is what we covered in the "Templates" lesson (we called it template hoisting), and we exercised this concept in the Queue assignment.

- We also talked about this in the "Template specialization" lecture, where we looked at an example where:
  - a class template was fully specialized for void *,
  - and then we defined a partial specialization for pointers (i.e., T *) in terms of this full specialization for void *.

- The idea here is to reduce the code bloat that template instantiations can and cause.

45

Use member function templates to accept “all compatible types”

- We talked about this in the “Templates” lesson as well as the assignment where you had to define a member template assignment operator.
- This item talks about the same concept, using an example of smart pointers on polymorphic types.

45

- Example from book:
- Consider a class hierarchy Top ← Middle ← Bottom
- So, Top is the root class, and Middle derives from Top, and Bottom derives from Middle.

- This means that we should be able to do the following assignments:

```
Top  * t;      Middle * m;      Bottom * b;

t = m;                                 // since Middle "is a" Top

t = b;                                // since Bottom "is a" Top

m = b;                                // since Bottom "is a" Middle
```

- All of these can happen without any casting required (since up casting is always ok, unlike down casting)

45

- Now lets say there is a template class called SmartPointer.

```
SmartPointer < Top >        smT;
SmartPointer < Middle >     smM;
SmartPointer < Bottom >     smB;
```

We would want to be able to do the following assignments:

Using raw pointers	Using smart pointers
t = m;	smT = smM;
t = b;	smT = smB;
m = b;	smM = smB;

- But, as we talked about in an earlier lesson, the three instantiations of SmartPointer above are all different classes.
- There is no relationship among them.

45

- This means that even though the raw pointer assignments are ok, the smart pointer assignments will not compile(because they are different types).

- You would need to add member template methods (like we saw in the lecture and in the assignment).

- In this example, you would need to add template constructors and assignment operators that would take another type T2 and do the assignment of the raw pointer.

i.e., smT = smB
Will have code instantiated which will semantically do the following:

smT.rawPointer = smB.rawPointer; // semantically that is t = b in our example.
Note: This code will actually look different since it would be inside a member function:
So, something like: m_RawPointer = other.m_RawPointer;

45

- Now, you don't want the following types of assignments to be allowed just because you added a member template constructor:

```
smB = smT;          // this would be equivalent of b = t, which is downcasting.
                    // We don't want this statement to compile.
```

However, as we saw in the stack/queue example in an earlier lecture, this statement will only compile if the internals of the member template instantiated would compile.

Now, we know that the following statement will not compile:

```
b = t;              // downcasting error.
```

And hence, smB = smT would not compile either, because in that member template, you would be assigning the raw pointers

(like b.m_RawPointer = t.m_RawPointer)

50

- Replacing new and delete.
  - Detecting underflow or overflow
  - Analyzing memory allocation patterns
  - Double deletion bugs?
  - Better performance for allocation/deletion
  - Etc.

Eff STL

- #2: Container independent code.
  - Not reasonable to expect to write this, unless the code is not doing a whole lot.

- #4: Call empty() instead of checking size()==0
  - Eg: For lists empty() is likely faster than the size()==0 check.
  - Consider the case where a list is spliced into another list.
  - The size may not be recomputed, and so a call to empty() would then be faster.

Eff STL

- #7 Consider container of pointers instead of objects.
  - Need to delete the objects in the container, as the container does not own these objects.

```
template <typename T>
void DeleteFunc( T * t )   {
            delete t;
}
void Foo {
            vector < MyClass * >   v1;
            v1.push_back( new MyClass() );  //… add a bunch of objects to v1
            // deletion time
            for ( int ii = 0; ii != v1.size(); ++ ii )
                        delete v1[ ii ];
            OR
            for_each( v1.begin(),  v1.end(), DeleteFunc<int> );
}                       // vector v1 will get deleted at end of function since it is on stack.
```

Eff STL

- #14: Use reserve to avoid unnecessary reallocations
- #15: Be aware of variations in string implementations
- #17: Use "the swap trick" to trim excess capacity

```
vector < MyClass > v1;
// .. Code to fill up v1, delete some entries, etc.

// Time to trim capacity to what is actually needed
vector < MyClass >( v1 ).swap( v1);
```