War Factories

So some of you know that I have a really terrible habit, a life sink that sucks away more free time than I would care to admit. Those of us who are under its terrible spell know it as *WoW*. You may know it as *World of Warcraft*.

Now, you might ask yourself, why is David bringing up *World of Warcraf*t in a legitimate programming class—even if it is pass/fail and taught by an undergrad? Well, I decided it was one of the easier ways to talk to you about Abstract Factories, which is a rather nice design pattern from Gamma, et al. Andrei Alexandrescu did a rather nice job of talking about it in *Modern C++ Design*. Hopefully I can do a reasonable job here.

Motivation

So in World of Warcraft, there are two kinds of monsters you can fight: regular and elite. Regular monsters are the ones that take about ten seconds to kill and they make up most of the long hours you spend trying to reach level sixty. They have pretty bad treasure, but you can take them out solo. Then there are the elites, which live almost exclusively in dungeons. These things take a good minute each, and only then if you have a team of people. On the plus side, they have really nice treasure, and really cool items. So we might consider coding up the system like this:

```
class AbstractWoWFactory
{
public:
     virtual Ogre * makeOgre() = 0;
     virtual Murloc * makeMurloc() = 0;
     virtual Jenkins * makeLeroyJenkins() = 0;
      // Yeah i went there.
     virtual Boss * makeBoss() = 0;
     virtual Loot * generateLoot() = 0;
     // Why do we have this?
     virtual ~AbstractWoWFactory() {};
};

class NormalWoWFactory : public AbstractWoWFactory
{
public:
     virtual Ogre * makeOgre()
     {return new BadOgre();}
     virtual Murloc * makeMurloc()
     {return new BadMurloc();}
     virtual Jenkins * makeLeroyJenkins()
     {return new StupidJenkins(); }
     virtual Boss * makeBoss()
     {return new YoungDragonBoss(); }
```

```
        virtual Loot * generateLoot()
        { return RandomLootMaker::BadLoot();}
};

// EliteWoWFactory similarly.
```

And somewhere in your code you would have logic like this:

```
AbstractFactory * factory = (ch->InDungeon())?
     new EliteWoWFactory  : new NormalWoWFactory;
```

Ok, now that we've properly motivated it, let's talk about the problems. As usual, the design is pretty OK, except that it's not very generic: every time you want to create a new factory thing, you have a lot of boilerplate to go through. As usual, we're gonna get rid of the boilerplate.

An Interface Suggestion

So let's talk about what we want out of this pattern. We want to have an interface class that quickly specifies all the functions we want (variants of say Create), and an easy way of specifying the implementation classes. I'll make this suggestion, but we'll talk about other ideas in class:

```
template <template <class T> class Creator,
          [some means of specifying types]
         >
class abstract_factory : private
                         [some way of inheriting from
                    Creator<T> for all the types we
                         specified.]
{
public:
     template <class T>
     T * create()
     {
          Creator<T> & creator = *this;
          return creator.create(/*Trickery later*/);
     }
};
```

So the idea here is it inherit from a bunch of instantiations of Creator, then select which version we want (the reference assignment), and then call create to return the correct version. So now we need a way of representing a list of types. Perhaps we'll call it a type-list:

```
template <class T, class Tail=nil>
struct type_list
```

```
{
     typedef T head;
     typedef Tail tail;
};

struct nil {} ;

// Imagine a lot of macros like this:
// TYPELIST_n(T1, T2, ... Tn) \
//   typelist<T1,TYPELIST_n-1(T2, ... Tn)>
```

“Oh God it's Lisp! You didn't say anything about Lisp!” Yeah, well, it turns out that C++ metaprogramming is very easily understandable as a dialect as Lisp, or any other functional language. Next week we'll be talking about all the features of this metalanguage. Anyway, just take a look at it for a minute. Let's define some metafunctions to get us started though:

```
template <class TL>
struct car
{
     typedef typename TL::head type;
};

template <class TL>
struct cdr
{
     typedef typename TL::tail type;
};

// Here's the neat one:
// So we need to figure out a way to inherit form
// X<T> for all types T in an typelist. In addition,
// we can't have X<T>'s inherit from each other.
// (Why? Think name hiding.)
// So we need some kind of branching scheme:
// We inherit from the car of a typelist, and then also
// recursively from our metafunction applied to the cdr:

template <class TL, template <class T> class X>
struct branch_inherit: X<typename car<TL>::type>,
                       branch_inherit<typename cdr<TL>::type>
{};

// Now we need a base case, since this is recursion.
// Partial specialization to the rescue:
template <template <class T> class X>
struct branch_inherit<nil,X> // no inheritance; we're done.
{};
```

Now we're ready for our factory:

```
template <class TL,
          template <class T> class Creator = SOME_DEFAULT
         >
class abstract_factory : protected
     branch_inherit<Creator,TL>
{
public:
     typedef TL product_list;
     template <class T>
     T * create()
     {
          Creator<T> & creator = *this;
          return creator.create();
     }
};

// We need an abstract creation mechanism now:
// Ideally, something we can use as a default:
// How about:
template <class T>
class abstract_creator
{
protected:
     virtual T * create(/* Trickery here too...*/) =0;
public:
     virtual ~abstract_creator(){}
}

// So now we can use things like:
typedef abstract_factory<
          TYPELIST_4(Ogre,Murloc,Boss,Loot)
        > AbstractWoWFactory;
```

Let's make things more concrete:

Now things get a bit more complicated. We want to define a concrete factory that can be passed around as a pointer to the abstract_factory. To do that, we need to implement a lot of pure virtual functions in our concrete factory. (Otherwise the class isn't very concrete is it?) Sounds like a job for branch_inherit, right? However, there's a catch: the implementation of overridden functions has to be done by a subclass, you can't just inherit another class that happens to implement a function with the right name and signature. What we need to do is to *linearly inherit* from some classes that implement these functions, and then at the base of the hierarchy, inherit from the abstract factory. Without further ado, the last metafunction of the handout:

```
template <template <class T, class B> class X,
          class TL,
          class Base>
```

```
struct linear_inherit: X<typename car<T>::type,
                  linear_inherit<X,
                                 typename cdr<TL>::type,
                                    Base>
                        >
{};

// Base case:
template <template <class T, class B> class X,
          class Base>
struct linear_inherit<X,nil,Base>: Base
{};

// X looks like this:
template <class T, class Base>
struct X: Base
{
     // functions and such.
};

// Here's one that we might want:
template <class T, class Base>
class use_new_factory_creator : public Base
{
public:
     T * create(/*Trickery goes here too...*/)
     {
          return new T();
     }
}
```

So what's the trickery? Unfortunately for us, we have to tell the compiler what version of create we want to overload. So we need to give the compiler a way of telling the functions apart. We need a type that is unique for a given type T, that isn't T itself. An easy answer is just T*:

```
template <class T, class Base>
class use_new_factory_creator : public Base
{
public:
     T * create(AppropriateBaseOfT *);
}
```

We're not quite done yet, since we have to pick the appropriate base of T, which just happens to be contained at the Nth place in the typelist. We'll use a bit of recursion. Note how we defined product_list in abstract_family. With that, we can do something like this:

```
template <class T, class Base>
class use_new_factory_creator : public Base
{
```

```
     typedef typename Base::product_list bpl;
protected:
     typedef typename car<bpl>::type AppropriateBaseOfT;
     typedef typename cdr<bpl>::type product_list;
public:
     T * create(AppropriateBaseOfT *);
}
```

Now concrete factory is rather straightforward to implement:

```
template <class AbstractFactory,
          class ConcreteProductList,
          template <class C,class B>
          class Creator=use_new_factory_creator
          >
struct concrete_factory :
                    linear_inherit<Creator,
                typename reverse<ConcreteProductList>::type,
                                   AbstractFactory>
{
     typedef ConcreteProductList product_list;
     typedef AbstractFactory  abstract_factory;
     typedef typename abstract_factory::product_list
                         abstract_product_list;
};

// And to use:
typedef concrete_factory<
                         AbstractWoWFactory,
                         TYPELIST_4(...)
                         > ConcreteWoWFactory;
```

WoW indeed. You might notice that whole reverse thing, and wonder why I promised there wouldn't be another metaprogramming algorithm in this handout. Well that's because it's the proverbial exercise left to the reader. As for why it's there, think about it for just a second. The top most element in the linear hierarchy right below the abstract_factory has the entire abstract product_list as its product_list, and so the head of that list is the the abstract_T1. However, it's the last phase of linear_inherit, so the T passed in that way is concrete_Tn. So, all you have to do is reverse the list.

That's it for this time. Next week, we'll be going deeper into this metaprogramming business, to show just how powerful it is, and what it can do for you.