Expanding Monostate Protopattern

Monostate

This is a technique in search of an application. Let's consider the Monostate pattern. A Monostate is a viable alternative to Singleton in many cases when trying to avoid the embarrassment of global variables.

```
class Monostate {
  public:
    int getNum( ) const { return num_; }
    void setNum( int num ) { num_ = num; }
    const std::string &getName( ) const { return name_; }
  private:
    static int num_;
    static std::string name_;
};
```

Like Singleton, a Monostate provides access to a single, shared copy of an object. Unlike a typical Singleton, the sharing is accomplished not by lazy construction of the object, but by access to static member data. Note that a Monostate differs from a traditional (and now outmoded and denigrated) use of a collection of static member data accessed through static member functions. A Monostate provides *non-static* member functions to access its static data.

```
Monostate m1;
Monostate m2;
//…
m1.setNum( 12 );
cout << m2.getNum() << endl; // shift 12
```

Every object of a particular Monostate type shares the same state. However, users of the Monostate do not have to employ any special syntax to get this result, unlike the corresponding use of a Singleton.

```
Singleton::instance().setNum( 12 );
cout << Singleton::instance().getNum() << endl;
```

Expanding Monostate

What if we'd like to be able to augment the members of a Monostate? Ideally this augmentation should take place without source code change, and if possible without recompilation of existing code that is not directly concerned with the augmentation. Let's look at a simple use of a template member function to accomplish this.

```
class Monostate {
  public:
    template <typename T>
    T &get() {
        static T member;
        return member;
    }
};
```

Note that a template member function will be instantiated as needed during compilation, and that it cannot be virtual. Fortunately, we do not require that our Monostate accessor functions be virtual. This version of Monostate implements a "lazy creation" strategy for its shared static members.

```
    Monostate m;
    m.get<int>( ) = 12; // create an int member
    Monostate m2;
    cout << m2.get<int>( ); // access previously-created member
    m2.get<std::string>( ) = "Hej!" // create a string member
```

Note that, unlike the lazy creation of a traditional Singleton, this lazy creation takes place at compile time instead of runtime.

Indexed Expanding Monostate

This approach is clearly far from ideal, at least if one wants to have more than one shared member of a particular type. One way to ameliorate the situation would be to add a second, "index," parameter to the template member function.

```
    class IndexedMonostate {
      public:
        template <typename T, int i>
        T &get( );
    };

    template <typename T, int i>
    T &IndexedMonostate::get( ) {
        static T member;
        return member;
    }
```

Now we have the ability to have more than one member of a particular type, but the ease of use of the technique clearly leaves something to be desired.

```
    IndexedMonostate im1, im2;
    im2.get<int,1066>( ) = 12;
    im2.get<double,42>( ) = im2.get<int,1066>( )+1;
```

Named Expanding Monostate

What we really need is a way to generate a mnemonic name for the user of a Monostate member that also encapsulates both a unique instantiation type for the template member function, and the actual type of the static member.

```
    template <typename T, int n>
    struct Name {
        typedef T Type;
    };
```

The `Name` template doesn't look like much, and it isn't. But it's just enough.

```
    typedef Name<int,86> grossAmount;
    typedef Name<double,007> percentage;
```

Now we can generate readable (type) names that bind together an index and a member type. Note that the actual values of the indexes are immaterial, as long as the [index,type] pair is unique. A Named Monostate makes the assumption that the type of the member can be extracted from its instantiation type.

```
    class NamedMonostate {
      public:
        template <class N>
        typename N::Type &get() {
            static typename N::Type member;
            return member;
        }
    };
```

This results in an improved user interface, without sacrificing any of the simplicity or flexibility of the original technique.

```
    NamedMonostate nm1, nm2;
    nm1.get<grossAmount>() = 12;
    nm2.get<percentage>( ) = nm1.get<grossAmount>( ) + 12.2;
    cout << nm1.get<grossAmount>() * nm2.get<percentage>() << endl;
```

Protopattern? What's That?

As we mentioned earlier, this is a technique looking for an application. As such, we really don't have the right to call it a pattern. A design pattern is an encapsulation of existing successful practice. The term "protopattern" is usually applied to a technique that has been observed in one or two contexts, but has not (yet) been shown to be widely enough employed to deserve the "pattern" appellation. I'm stretching things a bit here by calling an Expanding Monostate a protopattern, since I can't point to even a single successful application of it!