Modern C++ Design

1 Policy-Based Class Design

In brief, policy-based class design fosters assembling a class with complex behavior out of many little classes (called policies), each of which takes care of only one behavioral or structural aspect.

The generic SingletonHolder class **template** (Chapter 6) uses policies for managing lifetime and thread safety. SmartPtr (Chapter 7) is built almost entirely from policies. The **double-dispatch engine** in Chapter 11 uses policies for selecting various trade-offs. The generic **Abstract Factory** implementation in Chapter 9 uses a policy for choosing a creation method.

1.1 Failure of the Do-it-all Interface

Implementing everything under the umbrella of a do-it-all interface is not a good solution, for several reasons:

- Intellectual overhead, sheer size, and inefficiency.
- Loss of static type safety. A design should enforce most constraints at compile time. (No two singleton objects)

1.2 Multiple Inheritance to the Rescue?

For example, the user would build a multi-threaded, reference-counted smart pointer class by inheriting some `BaseSmartPtr` class and two classes: `MultiThreaded` and `RefCounted`. Any experienced class designer knows that such a naive design does not work.

The problems with assembling separate features by using multiple inheritance are as follows:

- **Mechanics**. There is no boilerplate code to assemble the inherited components in a controlled manner. **The language applies simple superposition in combining the base classes and establishes a set of simple rules for accessing their members.**

- **Type information**. The base classes do not have enough type information to carry out their tasks.
- **State manipulation**. Various behavioral aspects implemented with base classes must manipulate the same state. This means that they must use virtual inheritance to inherit a base class that holds the state. This complicates the design and makes it more rigid because the premise was that user classes inherit library classes, not vice versa.

1.3 Templates

Benefits:

- Class templates are customizable in ways not supported by regular classes.
- for class templates with multiple parameters, you can use partial template specialization.

As soon as you try to implement such designs, you stumble upon several problems that are not self-evident:

- You cannot specialize structure.
- Specialization of member functions does not scale: you cannot specialize individual member functions for templates with multiple template parameters.
- The library writer cannot provide multiple **default** values.

Multiple inheritance and templates foster complementary trade-offs:

- Multiple inheritance has scarce mechanics; templates have rich mechanics
- Multiple inheritance loses type information, which abounds in templates.
- Specialization of templates does not scale, but multiple inheritance scales quite nicely.
- You can provide only one default for a template member function, but you can write an unbounded number of base classes.

1.4 Policy Classes

A **policy** defines a class interface or a class template interface. The interface consists of one or all of the following: **inner type definitions, member functions, and member**

variables. The implementations of a policy are called **policy classes**. Policy classes are not intended for stand-alone use; instead, they are inherited by, or contained within, other classes.

```
template <class T>
struct OpNewCreator{
  static T* Create(){
    return new T;
  }
};

template <class T>
struct MallocCreator{
  static T* Create(){
    void* buf = std::malloc(sizeof(T));
    if (!buf) return 0;
    return new(buf) T;
  }
};

template <class T>
struct PrototypeCreator{
  PrototypeCreator(T* pObj = 0):pPrototype_(pObj){}
  T* Create(){
    return pPrototype_ ? pPrototype_->Clone() : 0;
  }
  T* GetPrototype() { return pPrototype_; }
  void SetPrototype(T* pObj) { pPrototype_ = pObj; }
private:
  T* pPrototype_;
};
```

The classes that use one or more policies are called hosts or **host classes**.

```
template <class CreationPolicy>
class WidgetManager : public CreationPolicy{
  ...
};
typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;
```

It is the user of `WidgetManager` who chooses the creation policy. This is the gist of policy-based class design.

1.5 Implementing Policy Classes with Template Template Parameters

The policy's template argument is redundant. In this case, we can use **template template parameters** for specifying policies, as shown in the following:

```
template <template <class> class CreationPolicy = OpNewcreator>
class WidgetManager : public CreationPolicy<Widget>{
  ...
};
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

Using template template parameters with policy classes is not simply a matter of convenience; sometimes, it is essential that the host class have access to the template so that the host can instantiate it with a different type. For example:

```
template <template <class> class CreationPolicy = OpNewcreator>
class WidgetManager : public CreationPolicy<Widget>{
  void DoSomething(){
    Gadget* pW = CreationPolicy<Gadget>().Create();
  }
};
```

Benefits of using policies:

- you can change policies from theoutside as easily as changing a template argument when you instantiate `WidgetManager`.
- you can provide your own policies that are specific to your concrete application.
- Policies allow you to generate designs by combining simple choices in a typesafe manner.
- the binding between a host class and its policies is done at compile time, the code is tight and efficient, comparable to its handcrafted equivalent.

1.6 Destructors of Policy Classes

The user can automatically convert a host class to a policy and later `delete` that pointer. Unless the policy class defines a virtual destructor, applying delete to a pointer to the policy class has undefined behavior.

Defining a virtual destructor for a policy, however, works against its static nature and hurts performance. The lightweight, effective solution that policies should use is to define a nonvirtual protected destructor:

```
template <class T>
struct OpNewCreator{
protected:
  ~OpNewCreator() {}
};
```

Because the destructor is protected, **only derived classes can destroy policy objects**, so it's impossible for outsiders to apply delete to a pointer to a policy class.

1.7 Enriched Policies

The `Creator` policy prescribes only one member function, `Create`. However, `PrototypeCreator` defines two more functions: `GetPrototype` and `SetPrototype`.

A user who uses a prototype-based Creator policy class can write the following code:

```
typedef WidgetManager<PrototypeCreator> MyWidgetManager;

Widget* pPrototype = ...;
MyWidgetManager mgr;
mgr.SetPrototype(pPrototype);
```

If the user later decides to use a creation policy that does not support prototypes, **the compiler pinpoints the spots where the prototype-specific interface was used.** This is exactly what should be expected from a sound design.

1.8 Optional Functionality Through Incomplete Instantiation

If a member function of a class template is never used, **it is not even instantiated—the compiler does not look at it at all**, except perhaps for syntax checking.

```
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>{
  void SwitchPrototype(Widget* pNewPrototype){
    CreationPolicy<Widget>& myPolicy = *this;
    delete myPolicy.GetPrototype();
    myPolicy.SetPrototype(pNewPrototype);
  }
};
```

The resulting context is very interesting:

- If the user instantiates `WidgetManager` with a Creator policy class that does not support prototypes and tries to use `SwitchPrototype`, a compile-time error occurs.
- If the user instantiates `WidgetManager` with a Creator policy class that does not support prototypes and does not try to use `SwitchPrototype`, the program is valid.

This all means that `WidgetManager` can benefit from optional enriched interfaces but still work correctly with poorer interfaces.

1.9 Compatible and Incompatible Policies

Suppose you create two instantiations of `SmartPtr : FastWidgetPtr`, a pointer with out checking, and `SafeWidgetPtr`, a pointer with checking before dereference. It is natural to accept the conversion from `FastWidgetPtr` to `SafeWidgetPtr`, but freely converting `SafeWidgetPtr` objects to `FastWidgetPtr` objects is dangerous.

The best, most scalable way to implement conversions between policies is to initialize and copy `SmartPtr` objects policy by policy, as shown below:

```
template<class T,template <class> class CheckingPolicy>
class SmartPtr : public CheckingPolicy<T>{
  template<class T1,template <class> class CP1,>
  SmartPtr(const SmartPtr<T1, CP1>& other)
    : pointee_(other.pointee_), CheckingPolicy<T>(other){ ... }
};
```

When you initialize a `SmartPtr<Widget, EnforceNotNull>` with a `SmartPtr<ExtendedWidget, NoChecking>`. The compiler tries to match `SmartPtr<ExtendedWidget, NoChecking>` to `EnforceNotNull`'s constructors.

If `EnforceNotNull` implements a **constructor** that accepts a `NoChecking` object, then the compiler matches that constructor. If `NoChecking` implements a **conversion** operator to `EnforceNotNull`, that conversion is invoked. In any other case, the code fails to compile.

Although conversions from `NoChecking` to `EnforceNotNull` and even vice versa are quite sensible, some conversions don't make any sense at all. As soon as you try to confine a pointer to another ownership policy, you break the invariant that makes reference counting work.

In conclusion, conversions that change the ownership policy should not be allowed implicitly and should be treated with maximum care.

1.10 Decomposing a Class into Policies

Two policies that do not interact with each other are orthogonal. By this definition, the Array and the Destroy policies are not orthogonal.

Nonorthogonal policies are an imperfection you should strive to avoid. **They reduce compile-time type safety and complicate the design of both the host class and the policy classes.**

If you must use nonorthogonal policies, you can minimize dependencies by passing a policy class as an argument to another policy class's template function. However, this decreases encapsulation.

2 Techniques

2.1 Compile-Time Assertions

The simplest solution to compile-time assertions works in C as well as in C++, relies on the fact that a zero-length array is illegal.

```
#define STATIC_CHECK(expr) { char unnamed[(expr) ? 1 : 0]; }
template <class To, class From>
To safe_reinterpret_cast(From from){
  STATIC_CHECK(sizeof(From) <= sizeof(To));
  return reinterpret_cast<To>(from);
}
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

The problem with this approach is that the error message you receive is not terribly informative. Error messages have no rules that they must obey; it's all up to the compiler.

A better solution is to rely on a template with an informative name; with luck, the compiler will mention the name of that template in the error message.

```
template<bool> struct CompileTimeError;
template<> struct CompileTimeError<true> {};
#define STATIC_CHECK(expr) \
(CompileTimeError<(expr) != 0>())
```

If you try to instantiate `CompileTimeError<false>`, the compiler utters a message such as "Undefined specialization `CompileTimeError<false>`." This message is a slightly better hint that the error is intentional and not a compiler or a program bug.

Actually, the name `CompileTimeError` is no longer suggestive in the new context.

```
template<bool> struct CompileTimeChecker{
  CompileTimeChecker(...);
};
template<> struct CompileTimeChecker<false> { };
#define STATIC_CHECK(expr, msg) {\
  class ERROR_##msg {}; \
  (void)sizeof(CompileTimeChecker<(expr) != 0>((ERROR_##msg())));\
}

template <class To, class From>
To safe_reinterpret_cast(From from){
  STATIC_CHECK(sizeof(From) <= sizeof(To),Destination_Type_Too_Narrow);
  return reinterpret_cast<To>(from);
}
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

After macro preprocessing, the code of `safe_reinterpret_cast` expands to the following:

```
template <class To, class From>
To safe_reinterpret_cast(From from){
  class ERROR_Destination_Type_Too_Narrow {};
  (void)sizeof(
    CompileTimeChecker<(sizeof(From) <= sizeof(To))>(
      ERROR_Destination_Type_Too_Narrow()));
  return reinterpret_cast<To>(from);
}
```

The `CompileTimeChecker<true>` specialization has a constructor that accept anything; it's an ellipsis function. If the comparison between sizes evaluates to false, a decent compiler outputs an error message such as "Error: Cannot convert `ERROR_Destination_Type_Too_Narrow` to `CompileTimeChecker <false>`.

2.2 Partial Template Specialization

```
template <class Window, class Controller>
class Widget{
  ... generic implementation ...
};

// Partial specialization of Widget
```

```
template <class Window>
class Widget<Window, MyController>{
  ... partially specialized implementation ...
};

template <class ButtonArg>
class Widget<Button<ButtonArg>, MyController>{
  ... further specialized implementation ...
};
```

Unfortunately, partial template specialization does not apply to functions—be they member or nonmember—which somewhat reduces the flexibility and the granularity of what you can do:

- Although you can **totally specialize** member functions of a class template, you cannot **partially specialize** member functions.
- You cannot partially specialize namespace-level (nonmember) template functions. The closest thing to partial specialization for namespace-level template functions is overloading (not for changing the return value or for internally used type).

```
template <class T, class U> T Fun(U obj); // primary template
template <class U> void Fun<void, U>(U obj); // illegal partial specialization
template <class T> T Fun (Window obj); // legal (overloading)
```