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

C++17 provides `static_assert`.
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. **The ellipsis means the constructor accepts anything.**

```
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)
```

2.3 Local Classes

Local classes cannot define static member variables and cannot access nonstatic local variables. What makes local classes truly interesting is that you can use them in template functions. **Local classes defined inside template functions can use the template parameters of the enclosing function.**

```
class Interface{
public:
virtual void Fun() = 0;
};
template <class T, class P>
Interface* MakeAdapter(const T& obj, const P& arg){
  class Local : public Interface{
  public:
    Local(const T& obj, const P& arg): obj_(obj), arg_(arg) {}
    virtual void Fun(){
      obj_.Call(arg_);
    }
  private:
     T obj_;
     P arg_;
  };
  return new Local(obj, arg);
}
```

It can be easily proven that any idiom that uses a local class can be implemented using a template class outside the function. On the other hand, local classes can simplify implementations and improve locality of symbols.

Local classes do have a unique feature, though: They are **final**. Outside users cannot derive from a class hidden in a function. Without local classes, you'd have to add an unnamed namespace in a separate translation unit.

2.4 Mapping Integral Constants to Types

```
template <int v>
struct Int2Type{
enum { value = v };
};
```

`Int2Type` generates a distinct type for each distinct constant integral value passed. You can use `Int2Type` whenever you need to ”typify” an integral constant quickly. This way you can **select different functions, depending on the result of a compile-time calculation.** Effectively, you **achieve static dispatching on a constant integral value.**

For dispatching at runtime, you can use simple `if-else` statements or the `switch` statement. However, the `if-else` statement requires both branches to compile successfully, even when the condition tested by `if` is known at compile time.

```
template <typename T, bool isPolymorphic>
class NiftyContainer{
  void DoSomething(){
    T* pSomeObj = ...;
    if (isPolymorphic){
      T* pNewObj = pSomeObj->Clone();
      ... polymorphic algorithm ...
    }
    else{
      T* pNewObj = new T(*pSomeObj);
      ... nonpolymorphic algorithm ...
    }
  }
};
```

The polymorphic algorithm uses `pObj->Clone()`, `NiftyContainer::DoSomething` does not compile for any type that doesn't define a member function `Clone()`.
If `T` has disabled its copy constructor (by making it private), if `T` is a polymorphic type and the nonpolymorphic code branch attempts `new T(*pObj)`, the code might fail to compile.

```
template <typename T, bool isPolymorphic>
class NiftyContainer{
private:
  void DoSomething(T* pObj, Int2Type<true>){
    T* pNewObj = pObj->Clone();
    ... polymorphic algorithm ...
  }
  void DoSomething(T* pObj, Int2Type<false>){
    T* pNewObj = new T(*pObj);
    ... nonpolymorphic algorithm ...
  }
public:
  void DoSomething(T* pObj){
    DoSomething(pObj, Int2Type<isPolymorphic>());
  }
};
```

There is another solution, `if constexpr()`, the new feature provided by c++17.

2.5 Type-to-Type Mapping

```
template <class T, class U>
T* Create(const U& arg){
  return new T(arg);
}
```

If objects of type `Widget` are untouchable legacy code and must take two arguments upon construction, the second being a fixed value such as -1. How can you specialize Create so that it treats `Widget` differently from all other types with a uniform interface?

```
// Illegal code | don't try this at home
template <class U>
Widget* Create<Widget, U>(const U& arg){
  return new Widget(arg, -1);
}

// rely on overloading
template <class T, class U>
T* Create(const U& arg, T /* dummy */){
  return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Widget /* dummy */){
  return new Widget(arg, -1);
}
```

Such a solution would incur the overhead of constructing an arbitrarily complex object that remains unused.

```
template <typename T>
struct Type2Type{
  typedef T OriginalType;
};
template <class T, class U>
T* Create(const U& arg, Type2Type<T>){
  return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Type2Type<Widget>){
  return new Widget(arg, -1);
}
// Use Create()
String* pStr = Create("Hello", Type2Type<String>());
Widget* pW = Create(100, Type2Type<Widget>());
```

2.6 Type Selection

Sometimes generic code needs to select one type or another, depending on a Boolean constant.

You might want to use an `std::vector` as your back-end storage. Obviously, you cannot store polymorphic types by value, so you must store pointers. On the other hand, you might want to store nonpolymorphic types by value, because this is more efficient.

```
template <typename T, bool isPolymorphic>
struct NiftyContainerValueTraits{
  typedef T* ValueType;
};
template <typename T>
```

```
struct NiftyContainerValueTraits<T, false>{
  typedef T ValueType;
};
template <typename T, bool isPolymorphic>
class NiftyContainer{
  typedef NiftyContainerValueTraits<T, isPolymorphic> Traits;
  typedef typename Traits::ValueType ValueType;
};
```

This way of doing things is unnecessarily clumsy. Moreover, it doesn't scale: For each type selection, you must define a new traits class template.

```
template <bool flag, typename T, typename U>
struct Select{
  typedef T Result;
};
template <typename T, typename U>
struct Select<false, T, U>{
  typedef U Result;
};

template <typename T, bool isPolymorphic>
class NiftyContainer{
  typedef typename Select<isPolymorphic, T*, T>::Result ValueType;
}
```

2.7 Detecting Convertibility and Inheritance at Compile Time

In a generic function, you can rely on an optimized algorithm if a class implements a certain interface. Discovering this at compile time means not having to use `dynamic_cast`, which is costly at runtime.

Detecting inheritance relies on a more general mechanism, that of detecting convertibility. The more general problem is, How can you detect whether an arbitrary type `T` supports automatic conversion to an arbitrary type `U`?

There is a surprising amount of power in `sizeof`: You can apply `sizeof` to any expression, no matter how complex, and `sizeof` **returns its size without actually evaluating that expression at runtime.**

The idea of conversion detection relies on using `sizeof` in conjunction with overloaded functions. We provide two overloads of a function: **One accepts the type to convert to (`U`), and the other accepts just about anything else.** If the function that accepts a `U` gets called, we know that `T` is convertible to `U`.

```
typedef char Small;
class Big { char dummy[2]; };
Small Test(U);
Big Test(...);
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

Passing a C++ object to a function with ellipses has undefined results, but this doesn't matter. Nothing actually calls the function. It's not even implemented. Recall that `sizeof` does not evaluate its argument.

There is one little problem. If `T` makes its default constructor private, the expression `T()` fails to compile. Fortunately, there is a simple solution, just use a strawman function returning a `T`. `MakeT` and `Test` not only don't do anything but don't even really exist at all.

```
template <class T, class U>
class Conversion{
  typedef char Small;
  class Big { char dummy[2]; };
  static Small Test(U);
  static Big Test(...);
  static T MakeT(); // not implemented
public:
  enum { exists = sizeof(Test(MakeT())) == sizeof(Small) };
};
cout << Conversion<size_t, vector<int> >::exists << ' ';
// return 0, because that constructor is explicit.
```

We can implement one more constant inside `Conversion::sameType`, which is true if `T` and `U` represent the same type:

```
template <class T, class U>
class Conversion{
  ... as above ...
  enum { sameType = false };
};
template <class T>
class Conversion<T, T>{
public:
  enum { exists = 1, sameType = 1 };
};
#define SUPERSUBCLASS(T, U) \
(Conversion<const U*, const T*>::exists && \
!Conversion<const T*, const void*>::sameType)
```

There are only three cases in which `const U*` converts implicitly to `const T*`:

1. `T` is the same type as `U`
2. `T` is an unambiguous public base of `U`
3. `T` is `void`.

Using `const` in `SUPERSUBCLASS`, we're always on the safe side, we don't want the conversion test to fail due to `const` issues.

Why use `SUPERSUBCLASS` and not the cuter `BASE_OF` or `INHERITS`? Think with `INHERITS(T, U)` it was a constant struggle to say which way the test worked.

2.8 A Wrapper Around type_info

tandard C++ provides the std::type_info class, which gives you the ability to investigate object types at runtime. You typically use type_info in conjunction with the typeid operator. The typeid operator returns a reference to a type_info object:

```
void Fun(Base* pObj){
  // Compare the two type_info objects corresponding to the type of *pObj and Derived
  if (typeid(*pObj) == typeid(Derived)){
    ... aha, pObj actually points to a Derived object ...
  }
}
```

In addition to supporting the comparison operators operator== and operator!=, type_info provides two more functions:

1. The name member function returns a textual representation of a type, in the form of const char*.
2. he before member function introduces an implementation's collation ordering relationship for type_info objects.
3. The type_info class disables the copy constructor and assignment operator, which makes storing type_info objects impossible.
4. The objects returned by typeid have static storage, so you don't have to worry about lifetime issues.

You do have to worry about pointer identity, the standard does not guarantee that each invocation returns a reference to the same type_info object. Consequently, you cannot compare pointers to type_info objects. What you should do is to store pointers to type_info objects and compare them by applying type_info::operator== to the dereferenced pointers.

If you want to use STL's ordered containers with type_info, you must write a little functor and deal with pointers. All this is clumsy enough to mandate a wrapper class around type_info that stores a pointer to a type_info object and provides:

1. All member functions of type_info
2. Value semantics (public copy constructor and assignment operator)
3. Seamless comparisons by defining operator< and operator==

```
class TypeInfo{
public:
  // Constructors/destructors
  TypeInfo(); // needed for containers
  TypeInfo(const std::type_info&);
  TypeInfo(const TypeInfo&);
  TypeInfo& operator=(const TypeInfo&);
  // Compatibility functions
```

```
  bool before(const TypeInfo&) const;
  const char* name() const;
private:
  const std::type_info* pInfo_;
};
// Comparison operators
bool operator==(const TypeInfo&, const TypeInfo&);
bool operator!=(const TypeInfo&, const TypeInfo&);
bool operator<(const TypeInfo&, const TypeInfo&);
bool operator<=(const TypeInfo&, const TypeInfo&);
bool operator>(const TypeInfo&, const TypeInfo&);
bool operator>=(const TypeInfo&, const TypeInfo&);

void Fun(Base* pObj){
  TypeInfo info = typeid(Derived);
  if (typeid(*pObj) == info){
    ... pBase actually points to a Derived object ...
  }
}
```

The cloning factory in Chapter 8 and one double-dispatch engine in Chapter 11 put `TypeInfo` to good use.

2.9 `NullType` and `EmptyType`

```
class NullType {};
struct EmptyType {};
```

You can use `NullType` for cases in which a type must be there syntactically but doesn't have a semantic sense. You can use `EmptyType` as a default (”don't care”) type for a template.

2.10 Type Traits

Traits are a generic programming technique that allows compile-time decisions to be made based on types, much as you would make runtime decisions based on values.

2.10.1 Implementing Pointer Traits

```
template <typename T>
class TypeTraits{
private:
  template <class U>
  struct PointerTraits{
    enum { result = false };
    typedef NullType PointeeType;
  };
  template <class U>
```

```
  struct PointerTraits<U*>{
    enum { result = true };
    typedef U PointeeType;
  };
public:
  enum { isPointer = PointerTraits<T>::result };
  typedef PointerTraits<T>::PointeeType PointeeType;
};

const bool iterIsPtr = TypeTraits<vector<int>::iterator>::isPointer;
cout << "vector<int>::iterator is " << iterIsPtr ? "fast" : "smart" << '\n';
```

Similarly, TypeTraits implements an `isReference` constant and a `ReferencedType` type definition.

Detection of pointers to members is a bit different. The specialization needed is as follows:

```
template <typename T>
class TypeTraits{
private:
  template <class U>
  struct PToMTraits{
    enum { result = false };
  };
template <class U, class V>
  struct PToMTraits<U V::*>{
    enum { result = true };
  };
public:
  enum { isMemberPointer = PToMTraits<T>::result };
};
```

2.10.2 Detection of Fundamental Types

`TypeTraits<T>` implements an `isStdFundamental` compile-time constant that says whether or not `T` is a standard fundamental type.

In Section 3, we will know an `TypeList` and the expression

```
TL::IndexOf<T, TYPELIST_nn(comma-separated list of types)>::value
```

returns the zero-based position of `T` in the list, or –1 if `T` does not figure in the list.

```
template <typename T>
class TypeTraits
{
... as above ...
public:
  typedef TYPELIST_4(unsigned char, unsigned short int, unsigned int, unsigned long int) Uns
  typedef TYPELIST_4(signed char, short int, int, long int) SignedInts;
```

```
  typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
  typedef TYPELIST_3(float, double, long double) Floats;
  enum { isStdUnsignedInt = TL::IndexOf<T, UnsignedInts>::value >= 0 };
  enum { isStdSignedInt = TL::IndexOf<T, SignedInts>::value >= 0 };
  enum { isStdIntegral = isStdUnsignedInt || isStdSignedInt || TL::IndexOf <T, OtherInts>::v
  enum { isStdFloat = TL::IndexOf<T, Floats>::value >= 0 };
  enum { isStdArith = isStdIntegral || isStdFloat };
  enum { isStdFundamental = isStdArith || isStdFloat || Conversion<T, void>::sameType };
  ...
};
```

2.10.3 Optimized Parameter Types

Given an arbitrary type T, what is the most efficient way of passing and accepting objects of type T as arguments to functions? In general, the most efficient way is to pass elaborate types by reference and scalar types by value.

A detail that must be carefully handled is that C++ does not allow references to references. Thus, if T is already a reference, you should not add one more reference to it.

```
template <typename T>
class TypeTraits{
  ... as above ...
public:
  typedef Select<isStdArith || isPointer || isMemberPointer, T,ReferencedType&>::Result Para
};
```

2.10.4 Stripping Qualifiers

```
template <typename T>
class TypeTraits{
  ... as above ...
private:
  template <class U> struct UnConst{
    typedef U Result;
  };
  template <class U> struct UnConst<const U>{
    typedef U Result;
  };
public:
  typedef UnConst<T>::Result NonConstType;
};
```

2.10.5 Using TypeTraits

```
enum CopyAlgoSelector { Conservative, Fast };
// Conservative routine-works for any type
template <typename InIt, typename OutIt>
```

```
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Conservative>){
  for (; first != last; ++first, ++result)
  *result = *first;
  return result;
}
// Fast routine-works only for pointers to raw data
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Fast>){
  const size_t n = last-first;
  BitBlast(first, result, n * sizeof(*first));
  return result + n;
}
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result){
  typedef TypeTraits<InIt>::PointeeType SrcPointee;
  typedef TypeTraits<OutIt>::PointeeType DestPointee;
  enum { copyAlgo =
         TypeTraits<InIt>::isPointer &&
         TypeTraits<OutIt>::isPointer &&
         TypeTraits<SrcPointee>::isStdFundamental &&
         TypeTraits<DestPointee>::isStdFundamental &&
         sizeof(SrcPointee) == sizeof(DestPointee) ? Fast : Conservative };
  return CopyImpl(first, last, result, Int2Type<copyAlgo>);
}
```

The drawback of Copy is that it doesn't accelerate everything that could be accelerated. For example, you might have a plain C-like struct containing nothing but primitive data—a so-called plain old data, or POD, structure.

```
template <typename T>
struct SupportsBitwiseCopy{
  enum { result = TypeTraits<T>::isStdFundamental };
};
template<>
struct SupportsBitwiseCopy<MyType>{
  enum { result = true };
};
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result, Int2Type<true>){
  typedef TypeTraits<InIt>::PointeeType SrcPointee;
  typedef TypeTraits<OutIt>::PointeeType DestPointee;
  enum { useBitBlast =
         TypeTraits<InIt>::isPointer &&
         TypeTraits<OutIt>::isPointer &&
         SupportsBitwiseCopy<SrcPointee>::result &&
         SupportsBitwiseCopy<DestPointee>::result &&
         sizeof(SrcPointee) == sizeof(DestPointee) };
```

```
  return CopyImpl(first, last, Int2Type<useBitBlast>);
}
```

2.10.6 Summary

The most important point is that the compiler always find the best match of template specialization.

3 Typelists

3.1 The need for Typelists

If you want to generalize the concept of Abstract Factory and put it into a library, you have to make it possible for the user to create factories of arbitrary collections of types.

- . In the Abstract Factory case, although the abstract base class is quite simple, you can get a nasty amount of code duplication when implementing various concrete factories.
- You cannot easily manipulate the member functions of `WidgetFactory` **because virtual functions cannot be templates.**
- We wish it would be nice if we could create a `WidgetFactory` by passing a parameter list to an `AbstractFactory` template and we coul have a template-like invocation for various `CreateXxx` functions, such as `Create<Window>()`.

The definition and algorithm of `Typelist` is the same as `std::Tuple`

```
template <class T, class U>
struct Typelist{
  typedef T Head;
  typedef U Tail;
};
typedef Typelist<int, NullType> OneTypeOnly;
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
...
```

There is a lot of utility algorithms of Typelist:

- Calculating length

  ```
  template <class TList> struct Length;
  template <> struct Length<NullType>{
    enum { value = 0 };
  };
  template <class T, class U>
  struct Length< Typelist<T, U> >{
    enum { value = 1 + Length<U>::value };
  };
  ```

- Indexed Access

```
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>{
  typedef Head Result;
};
template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>{
  typedef typename TypeAt<Tail, i - 1>::Result Result;
};
```

- Searching Typelists

```
template <class T>
struct IndexOf<NullType, T>{
  enum { value = -1 };
};
template <class T, class Tail>
struct IndexOf<Typelist<T, Tail>, T>{
  enum { value = 0 };
};
template <class Head, class Tail, class T>
struct IndexOf<Typelist<Head, Tail>, T>{
private:
  enum { temp = IndexOf<Tail, T>::value };
public:
  enum { value = temp == -1 ? -1 : 1 + temp };
};
```

- Appending to Typelist

```
template <> struct Append<NullType, NullType>{
  typedef NullType Result;
};
template <class T> struct Append<NullType, T>{
  typedef TYPELIST_1(T) Result;
};
template <class Head, class Tail>
struct Append<NullType, Typelist<Head, Tail> >{
  typedef Typelist<Head, Tail> Result;
};
template <class Head, class Tail, class T>
struct Append<Typelist<Head, Tail>, T>{
  typedef Typelist<Head,typename Append<Tail, T>::Result> Result;
};
```

- Erasing a type from Typelist

```
template <class T>
struct Erase<NullType, T>{
  typedef NullType Result;
};
template <class T, class Tail>
struct Erase<Typelist<T, Tail>, T>{
  typedef Tail Result;
};
template <class Head, class Tail, class T>
struct Erase<Typelist<Head, Tail>, T>{
  typedef Typelist<Head,typename Erase<Tail, T>::Result> Result;
};
```

- Erasing Duplicates

```
template <> struct NoDuplicates<NullType>{
typedef NullType Result;
};
template <class Head, class Tail>
struct NoDuplicates< Typelist<Head, Tail> >{
private:
typedef typename NoDuplicates<Tail>::Result L1;
typedef typename Erase<L1, Head>::Result L2;
public:
typedef Typelist<Head, L2> Result;
};
```

- Replacing a type in a Typelist

```
template <class T, class U>
struct Replace<NullType, T, U>{
  typedef NullType Result;
};
template <class T, class Tail, class U>
struct Replace<Typelist<T, Tail>, T, U>{
  typedef Typelist<U, Tail> Result;
};
template <class Head, class Tail, class T, class U>
struct Replace<Typelist<Head, Tail>, T, U>{
  typedef Typelist<Head,typename Replace<Tail, T, U>::Result> Result;
};
```

- Partially Ordering Typelist

```
template <class T>
struct MostDerived<NullType, T>{
  typedef T Result;
```

```
    };
    template <class Head, class Tail, class T>
    struct MostDerived<Typelist<Head, Tail>, T>{
    private:
      typedef typename MostDerived<Tail, T>::Result Candidate;
    public:
      typedef typename Select<SUPERSUBCLASS(Candidate, Head), Head, Candidate>::Result Resul
    };
    template <>
    struct DerivedToFront<NullType>{
      typedef NullType Result;
    };
    template <class Head, class Tail>
    struct DerivedToFront< Typelist<Head, Tail> >{
    private:
      typedef typename MostDerived<Tail, Head>::Result TheMostDerived;
      typedef typename Replace<Tail, TheMostDerived, Head>::Result L;
    public:
      typedef Typelist<TheMostDerived, L> Result;
    };
```

3.2 Class Generation with Typelists

```
template <class TList, template <class> class Unit>
class GenScatterHierarchy;

template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<Typelist<T1, T2>, Unit>
: public GenScatterHierarchy<T1, Unit>
, public GenScatterHierarchy<T2, Unit>{
public:
  typedef Typelist<T1, T2> TList;
  typedef GenScatterHierarchy<T1, Unit> LeftBase;
  typedef GenScatterHierarchy<T2, Unit> RightBase;
};

template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtomicType>{
  typedef Unit<AtomicType> LeftBase;
};

template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>{};

template <class T>
struct Holder{
```

```
  T value_;
};
typedef GenScatterHierarchy<TYPELIST_3(int, string, Widget), Holder> WidgetInfo;
WidgetInfo obj;
string name = (static_cast<Holder<string>&>(obj)).value_;
```

This cast is quite ugly.

```
template <class T, class H>
typename Private::FieldTraits<H>::Rebind<T>::Result& Field(H& obj){
  return obj;
}
```

If you call `Field<Widget>(obj)` , the compiler figures out that `Holder<Widget>` is a base class of `WidgetInfo` and simply returns a reference to that part of the compound object.

3.3 Generating Tuples

```
template <class T>
struct TupleUnit{
  T value_;
  operator T&() { return value_; }
  operator const T&() const { return value_; }
};
template <class TList>
struct Tuple : public GenScatterHierarchy<TList, TupleUnit>{};
```

4 Small-Object Allocation

4.1 Why we need smallObj allocation?

For various reasons, polymorphic behavior being the most important, these small objects cannot be stored on the stack and must live on the free store . C++ provides the operators `new` and `delete` as the primary means of using the free store. However, these operators are general purpose and perform badly for allocating small objects.

For occult reasons, the default allocator (`malloc, realloc, free`) is notoriously slow. In addition to being slow, the genericity of the default C++ allocator makes it very space inefficient for small objects. Usually, the bookkeeping memory amounts to a few extra bytes (4 to 32) for each block allocated with `new`, If you allocate 8-byte objects, the per-object overhead becomes 50% to 400%.

4.2 The workings of a memory allocator

```
struct MemControlBlock{
  std::size_t size_;
  bool available_;
};
```

For each allocation request, a linear search of memory blocks finds a suitable block for the requested size. Each deallocation incurs, again, a linear search for figuring out the memory block that precedes the block being deallocated, and an adjustment of its size.

```
struct MemControlBlock{
  bool available_ ;
  MemControlBlock* prev_;
  MemControlBlock* next_;
};
```

If you store pointers to the previous and next `MemControlBlock` in each `MemControlBlock`, you can achieve constant-time deallocation.

4.3 A Small-Object Allocator

The small-object allocator described in this chapter sports a four-layered structure:

1. `Chunk` contains and manages a chunk of memory consisting of an integral number of fixedsize blocks. `Chunk` contains logic that allows you to allocate and deallocate memory blocks

2. A `FixedAllocator` object uses `Chunk` as a building block. `FixedAllocator`'s primary purpose is to satisfy memory requests that go beyond a `Chunk`'s capacity. `FixedAllocator` does this by aggregating an array of `Chunks`.

3. `SmallObjAllocator` provides general allocation and deallocation functions. A `SmallObjAllocator` holds several `FixedAllocator` objects, each specialized for allocating objects of one size.

4. Finally, `SmallObject` wraps `FixedAllocator` to offer encapsulated allocation services for C++ classes. SmallObject overloads operator `new` and operator `delete` and passes them to a `SmallObjAllocator` object.

4.4 Chunk

```
struct Chunk{
  void Init(std::size_t blockSize, unsigned char blocks);
  void* Allocate(std::size_t blockSize);
  void Deallocate(void* p, std::size_t blockSize);
  void Reset(std::size_t blockSize, unsigned char blocks);
  void Release();
  unsigned char* pData_;
  unsigned char firstAvailableBlock_, blocksAvailable_;
};
```

`firstAvailableBlock_` holds the index of the first block available in this chunk, `blocksAvailable_` holds the number of blocks available in this chunk.

`Chunk` does not define constructors, destructors, or assignment operator. Defining proper copy semantics at this level hurts efficiency at upper level. Allocating and deallocating a block inside a Chunk takes constant time.

Why we use `unsinged char` but not `unsigned short` (2 bytes on many machines):

1. We cannot allocate blocks smaller than `sizeof(unsigned short)`, which is awkward because we're building a small-object allocator.

2. Imagine you build an allocator for 5-byte blocks. In this case, casting a pointer that points to such a 5-byte block to unsigned int engenders undefined behavior.

4.5 FixedAllocator

```
class FixedAllocator{
private:
    // Internal functions
    void DoDeallocate(void* p);
    Chunk* VicinityFind(void* p);

    std::size_t blockSize_;
    unsigned char numBlocks_;
    typedef std::vector<Chunk> Chunks;
    Chunks chunks_;
    Chunk* allocChunk_;
    Chunk* deallocChunk_;

    // For ensuring proper copy semantics
    mutable const FixedAllocator* prev_;
    mutable const FixedAllocator* next_;
public:
    explicit FixedAllocator(std::size_t blockSize = 0);
    FixedAllocator(const FixedAllocator&);
    FixedAllocator& operator=(const FixedAllocator&);
    ~FixedAllocator();
```

```
    void Swap(FixedAllocator& rhs);

    // Allocate a memory block
    void* Allocate();
    void Deallocate(void* p);
    std::size_t BlockSize() const{ return blockSize_; }
};
```

allocChunk_ holds a pointer to the last chunk that was used for an allocation. Whenever an allocation request comes, `FixedAllocator::Allocate` first checks `allocChunk_` for available space. If not, a linear search occurs.
`deallocChunk_` points to the last `Chunk` object that was used for a deallocation. Whenever a deallocation occurs, `deallocChunk_` is checked first. Then, if it's the wrong chunk, `Deallocate` performs a linear search:

1. during deallocation, a chunk is freed only when there are two empty chunks.
2. `chunks_` is searched starting from `deallocChunk_` and going up and down with two iterators.

4.6 SmallObjAllocator

```
class SmallObjAllocator{
public:
  SmallObjAllocator(std::size_t chunkSize,std::size_t maxObjectSize);
  void* Allocate(std::size_t numBytes);
  void Deallocate(void* p, std::size_t size);
private:
  std::vector<FixedAllocator> pool_;
  FixedAllocator* pLastAlloc_;
  FixedAllocator* pLastDealloc_;
  std::size_t chunkSize_;
  std::size_t maxObjectSize_;
};
```

The `chunkSize` parameter is the default chunk size (the length in bytes of each Chunk object), and `maxObjectSize` is the maximum size of objects that must be considered to be "small." `SmallObjAllocator` forwards requests for blocks larger than `maxObjectSize` directly to `::operator new`.
We store `FixedAllocators` only for sizes that are requested at least once. This way `pool_` can accommodate various object sizes without growing too much. To improve lookup speed, `pool_` is kept sorted by block. size.
When an allocation request arrives, `pLastAlloc_` is checked first. If it is not of the correct size, `SmallObjAllocator::Allocate` performs a binary search in `pool_`. Deal location requests are handled in a similar way.

4.7 Small Object

```
class SmallObject{
```

```
public:
  static void* operator new(std::size_t size);
  static void operator delete(void* p, std::size_t size);
  virtual ~SmallObject() {}
};
```

In standard C++ you can overload the default operator delete in two ways—either as

```
void operator delete(void* p);
```

or as

```
void operator delete(void* p, std::size_t size);
```

To avoid the overhead of storing the size of the actual object to which p points, the compiler does a hat trick: It generates code that figures out the size on the fly. Four possible techniques of achieving that are listed here:

1. Pass a Boolean flag to the destructor meaning "Call/don't call operator delete after destroying the object." Base's destructor is virtual, so, `delete p` will reach the right object, `Derived`. At that time, the size of the object is known statically—it's `sizeof(Derived)`, and the compiler simply passes this constant to operator `delete`.

2. You can arrange that each destructor, after destroying the object, returns `sizeof(Class)`.

3. Implement a hidden virtual member function that gets the size of an object, say `_Size()`.

4. Store the size directly somewhere in the virtual function table (vtable) of each class. This solution is both flexible and efficient, but less easy to implement.

We need a unique `SmallObjAllocator` object for the whole application. That `SmallObjAllocator` must be properly constructed and properly destroyed, which is a thorny issue on its own. we solve this problem thoroughly with its `SingletonHolder` template.

```
typedef Singleton<SmallObjAllocator> MyAlloc;
void* SmallObject::operator new(std::size_t size){
  return MyAlloc::Instance().Allocate(size);
}
void SmallObject::operator delete(void* p, std::size_t size){
  MyAlloc::Instance().Deallocate(p, size);
}
```

4.8 Multithreading issues

The unique `SmallObjAllocator` is shared by all instances of `SmallObject`. If these instances belong to different threads, we end up sharing the `SmallObjAllocator` between multiple threads.

```
template <template <class T> class ThreadingModel>
class SmallObject : public ThreadingModel<SmallObject>{
  ... as before ...
```

```
}

template <template <class T> class ThreadingModel>
void* SmallObject<ThreadingModel>::operator new(std::size_t size){
  Lock lock;
  return MyAlloc::Instance().Allocate(size);
}
template <template <class T> class ThreadingModel>
void SmallObject<ThreadingModel>::operator delete(void* p, std::size_t size){
  Lock lock;
  MyAlloc::Instance().Deallocate(p, size);
}
```

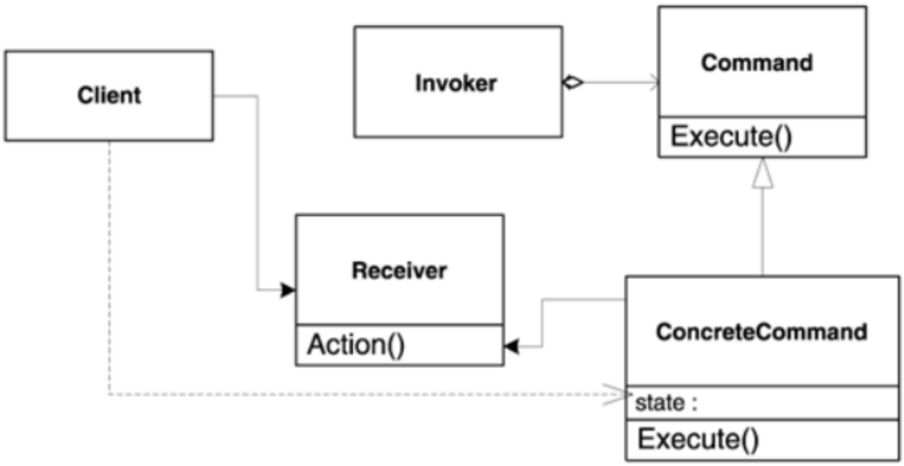
5 Generalized Functors

5.1 Why we need functors?

A generalized functor

- **Encapsulates** any processing invocation because it accepts pointers to simple functions, pointers to member functions, functors, and even other generalized functors—together with some or all of their respective arguments.
- Is **typesafe** because it never matches the wrong argument types to the wrong functions.
- Is an object with **value semantics** because it fully supports copying, assignment, and pass by value. A generalized functor can be copied freely and does not expose virtual member functions.

A typical sequence of actions is as follows:



1. The application (client) creates a `ConcreteCommand` object, passing it enough information to carry on a task.
2. The application passes the `Command` interface of the `ConcreteCommand` object to the invoker. The invoker stores this interface.
3. the invoker decides it's time to execute the action and fires `Command`'s Execute virtual member function. The virtual call mechanism dispatches the call to the `Concrete-Command` object, which takes care of the details. `ConcreteCommand` reaches the `Receiver` object (the one that is to do the job) and uses that object to perform the actual processing, such as calling its `Action` member function. Alternatively, the `ConcreteCommand` object might carry the processing all by itself. In this case, the receiver in Figure disappears.

There are two important aspects of the Command pattern:

- **Interface separation**. The invoker is isolated from the receiver.

- **Time separation**. Command stores a ready-to-go processing request that's to be started later.

From an implementation standpoint, two kinds of concrete `Command` classes can be identified:

1. All they do is call a member function for a `Receiver` object. We call them **forwarding commands**.

2. Others do tasks that are more complex. They might call member functions of other objects, but they also embed logic that's beyond simple forwarding. Let's call them active commands.

Because forwarding commands act much like pointers to functions and their C++ colleagues, functors, we call them **generalized functors**.

5.2 C++ Callable Entities

A forwarding command is a callback on steroids, a generalized callback. A callback is a pointer to a function that can be passed around and called at any time.

In addition to simple callbacks, C++ defines many more entities that support the function-call operator. Let's enumerate all the things that support `operator()` in C++:

- C-like functions

- C-like pointers to functions

- References to functions (which essentially act like const pointers to functions)

- Functors, that is, objects that define an `operator()`

- The result of applying `operator.*` or `operator->*` having a pointer to a member function

The objects that support `operator()` are known as **callable entities**.

5.3 `Functor` Class Template Skeleton

in C++ a bald pointer to a polymorphic type does not strictly have first-class semantics because of the ownership issue. To lift the burden of lifetime management from `Functor`'s clients, it's best to provide `Functor` with value semantics (well-defined copying and assignment). `Functor` does have a polymorphic implementation, but that's hidden inside it. We name the implementation base class `FunctorImpl`.

```
template <typename R, class TList>
class Functor{
public:
  Functor();
  Functor(const Functor&);
  Functor& operator=(const Functor&);
  explicit Functor(std::auto_ptr<Impl> spImpl);
private:
  // Handy type definition for the body type
```

```
  typedef FunctorImpl<R, TList> Impl;
  std::auto_ptr<Impl> spImpl_;
};
```

The purpose of `Clone` is the creation of a polymorphic copy of the `FunctorImpl` object. `FunctorImpl` defines a polymorphic interface that abstracts a function call.

```
template <typename R>
class FunctorImpl<R, NullType>{
public:
  virtual R operator()() = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};
template <typename R, typename P1>
class FunctorImpl<R, TYPELIST_1(P1)>{
public:
  virtual R operator()(P1) = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};
template <typename R, typename P1, typename P2>
class FunctorImpl<R, TYPELIST_2(P1, P2)>{
public:
  virtual R operator()(P1, P2) = 0;
  virtual FunctorImpl* Clone() const = 0;
  virtual ~FunctorImpl() {}
};
```

Constructing from `auto_ptr` is a clear statement to the outside world that `Functor` takes ownership of the `FunctorImpl` object. Users of `Functor` will actually have to type `auto_ptr` whenever they invoke this constructor; we assume that if they type `auto_ptr`, they know what `auto_ptr` is about.

5.4 Implementing the Forwarding `Functor::operator()`

```
template <typename R, class TList>
class Functor{
... as above ...
public:
  R operator()(){  return (*spImpl_)(); }
  R operator()(Parm1 p1){  return (*spImpl_)(p1); }
  R operator()(Parm1 p1, Parm2 p2){  return (*spImpl_)(p1, p2); }
};
```

The trick relies on the fact that **C++ does not instantiate member functions for templates until they are actually used**. If you try to call an overload of `operator()` that doesn't make sense, the compiler tries to generate the body of `operator()` and discovers the mismatch.

5.5 Handling Functors

```
template <class ParentFunctor, typename Fun>
class FunctorHandler: public FunctorImpl<
                       typename ParentFunctor::ResultType,
                       typename ParentFunctor::ParmList>{
public:
  typedef typename ParentFunctor::ResultType ResultType;
  FunctorHandler(const Fun& fun) : fun_(fun) {}
  FunctorHandler* Clone() const{
    return new FunctorHandler(*this);
  }
  ResultType operator()(){
    return fun_();
  }
  ResultType operator()(typename ParentFunctor::Parm1 p1){
    return fun_(p1);
  }
  ResultType operator()(typename ParentFunctor::Parm1 p1,typename ParentFunctor::Parm2 p2){
    return fun_(p1, p2);
  }
private:
  Fun fun_;
};
```

The functor is stored by value, not by pointer. This is because, in general, functors are meant to be this way—nonpolymorphic types with regular copy semantics.

Given `FunctorHandler`'s declaration, it's easy to write the templated constructor of `Functor` declared earlier in this section.

```
template <typename R, class TList>
template <typename Fun>
Functor<R, TList>::Functor(const Fun& fun) : spImpl_(new FunctorHandler<Functor, Fun>(fun)){
```

`FuncotHandler` **not only handles functor, function but also pointer and reference to functions.**

However, if the function is **overloaded**, the type of the function is no longer defined.

```
void TestFunction(int i, double d){ cout << "TestFunction << endl; }
void TestFunction(int);
```

There are two methods:

```
int main()
{
typedef void (*TpFun)(int, double);
// Method 1: use an initialization
TpFun pF = TestFunction;
Functor<void, TYPELIST_2(int, double)> cmd1(pF);
```

```
cmd1(4, 4.5);

// Method 2: use a cast
Functor<void, int, double> cmd2(static_cast<TpFun>(TestFunction));
cmd2(4, 4.5);
}
```

5.6 Argument and Return Type Conversions

In an ideal world, we would like conversions to work for `Functor` just as they work for regular function calls.

```
const char* TestFunction(double, double){
  static const char buffer[] = "Hello, world!";
// It's safe to return a pointer to a static buffer
  return buffer;
}
int main(){
  Functor<string, TYPELIST_2(int, int)> cmd(TestFunction);
  // Should print "world!"
  cout << cmd(10, 10).substr(7);
}
```

The function

```
string Functor<...>::operator()(int i, int j)
```

forwards to the virtual function

```
string FunctorHandler<...>::operator()(int i, int j)
```

whose implementation ultimately calls

```
return fun_(i, j);
```

where `fun_` has type `const char* (*)(double, double)` and evaluates to `TestFunction`. When the compiler encounters the call to `fun_`, it compiles it normally. The compiler then generates code to convert `i` and `j` to `double`, and the result to `std::string`.

5.7 Handling Pointers to Member Functions

```
class Parrot{
public:
void Eat(){ cout << "Tsk, knick, tsk...\n"; }
void Speak(){ cout << "Oh Captain, my Captain!\n"; }
};
int main(){
  typedef void (Parrot::* TpMemFun)();
  TpMemFun pActivity = &Parrot::eat;
```

```
  Parrot geronimo;
  Parrot* pGeronimo = &geronimo;
  (geronimo.*pActivity)();
  (pGeronimo->*pActivity)();
  pActivity = &Parrot::Speak;
  (geronimo.*pActivity)();
}
```

There is no C++ type for the result of `geronimo.*p-Activity` and `pGeronimo->*pActivity`. Both are binary operations all right, and they return something to which you can apply the function-call operator immediately, but that "something" does not have a type. You cannot store the result of `operator.*` or `operator->*` in any way.
Here's the implementation of `MemFunHandler`.

```
template <class ParentFunctor, typename PointerToObj, typename PointerToMemFn>
class MemFunHandler : public FunctorImpl<
                       typename ParentFunctor::ResultType,
                       typename ParentFunctor::ParmList>{
public:
  typedef typename ParentFunctor::ResultType ResultType;

  MemFunHandler(const PointerToObj& pObj, PointerToMemFn pMemFn) : pObj_(pObj), pMemFn_(pMem
  MemFunHandler* Clone() const{
    return new MemFunHandler(*this);
  }

  ResultType operator()(){
    return ((*pObj_).*pMemFn_)();
  }
  ResultType operator()(typename ParentFunctor::Parm1 p1){
    return ((*pObj_).*pMemFn_)(p1);
  }
ResultType operator()(typename ParentFunctor::Parm1 p1, typename ParentFunctor::Parm2 p2){
    return ((*pObj_).*pMemFn_)(p1, p2);
  }
private:
  PointerToObj pObj_;
  PointerToMemFn pMemFn_;
};
```

Why is `MemFunHandler` parameterized with the type of the pointer (`PointerToObj`) and not with the type of the object itself? i.e.

```
template <class ParentFunctor, typename Obj,
typename PointerToMemFn>
class MemFunHandler : public FunctorImpl<
                       typename ParentFunctor::ResultType,
                       typename ParentFunctor::ParmList>{
```

```
private:
  Obj* pObj_;
  PointerToMemFn pMemFn_;
public:
MemFunHandler(Obj* pObj, PointerToMemFn pMemFn) : pObj_(pObj), pMemFn_(pMemFn) {}
};
```

The first implementation can store any type that acts as a pointer to an object **but** the second is hardwired to store and use only simple pointers, if you want to use smart pointers, there will be wrong.
Moreover, the second version does not work for pointers to `const`. Such is the negative effect of hardwiring type.

```
int main(){
  Parrot geronimo;
  Functor<> cmd1(&geronimo, &Parrot::Eat), cmd2(&geronimo, &Parrot::Speak);
  cmd1();
  cmd2();
}
```

5.8 Binding

As soon as `Functor` is ready, new ideas come to mind. For instance, we'd like to be able to convert from a type of `Functor` to another. Think of a `Functor` as a computation, and of its arguments as the **environment** necessary to perform that computation, binding allows `Functor` to store part of the **environment** together with the computation and to reduce progressively the environment necessary at invocation time.

```
template <class Incoming>
class BinderFirst : public FunctorImpl<typename Incoming::ResultType,
                     typename Incoming::Arguments::Tail>{
  typedef Functor<typename Incoming::ResultType,Incoming::Arguments::Tail> Outgoing;
  typedef typename Incoming::Parm1 Bound;
  typedef typename Incoming::ResultType ResultType;
public:
  BinderFirst(const Incoming& fun, Bound bound): fun_(fun), bound_(bound){}
  BinderFirst* Clone() const{ return new BinderFirst(*this); }
  ResultType operator()(){ return fun_(bound_); }
  ResultType operator()(typename Outgoing::Parm1 p1){
    return fun_(bound_, p1);
  }
  ResultType operator()(typename Outgoing::Parm1 p1,typename Outgoing::Parm2 p2){
    return fun_(bound_, p1, p2);
  }
private:
  Incoming fun_;
  Bound bound_;
```

```
};

template <class Fctor>
typename Private::BinderFirstTraits<Fctor>::BoundFunctorType
BindFirst(const Fctor& fun,typename Fctor::Parm1 bound){
  typedef typename private::BinderFirstTraits<Fctor>::BoundFunctorType Outgoing;
  return Outgoing(std::auto_ptr<typename Outgoing::Impl>(
                  new BinderFirst<Fctr>(fun, bound)));
}
```

5.9 Chaining Request

`MacroCommand` class, a command that holds a linear collection (such as a list or a vector) of `Commands`. When a `MacroCommand` is executed, it executes in sequence each of the commands that it holds.

5.10 `Functor` Quick Facts

- You can initialize a `Functor` with a function, a functor, another Functor, or a pointer to an object and a pointer to a method.
- You also can initialize `Functor` with a `std::auto_ptr< FunctorImpl<R,TList> >`.
- `Functor` supports automatic conversions for arguments and return values.
- Manual disambiguation is needed in the presence of overloading.
- `Functor` fully supports first-class semantics: copying, assigning to, and passing by value.
- `Functor` is not polymorphic and is not intended to be derived from. If you want to extend `Functor`, derive from `FunctorImpl`.
- A call to `BindFirst` binds the first argument to a fixed value.
- Multiple `Functors` can be chained in a single `Functor` object by using the `Chain` function.
- `FunctorImpl` uses the small-object allocator.