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.

```
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)
  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
  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
};
```

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.