模板与泛型编程

条款41: 了解隐式接口和编译期多态

- classes 和 templates 都支持接口和多态
- 对 classes 而言接口是显式的，以函数签名为中心。多态则是通过 virtual 函数发生于运行期。
- 对 template 参数而言，接口是隐式的，奠基于有效表达式。多态则是通过 template 具现化和函数重载解析发生于编译期。

面向对象编程世界总是以显式接口和运行期多态解决问题。

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);
};
void doProcessing(Widget& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    Widget temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

上述代码中的 doProcessing 就体现了这两点：

- Widget 是一个显式接口
- Widget 中的 virtual 成员函数，在 doSomething 中依据动态类型而变化是运行期多态的体现。

而在 Templates 和 泛型编程的世界中，这两者重要性降低，反而 隐式接口和编译器多态 重要性升高。

```
template<typename T>
void doProcessing(T& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    T temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

例如上述代码：

- 上述的执行过程要求 T 类型必须支持 size, normalize, swap, copy, inequality comparison 等行为，这就是一组隐式接口
- 所有涉及到 w 的任何调用都有可能让 template 具象化，这种行为发生在编译期，会导致编译期多态。

条款42: 了解 typename 的双重意义

- 声明 template 参数时，前缀关键字 class 和 typename 可互换。
- 请使用关键字 typename 标识嵌套从属类型名称，但不得在 base class lists 和 member initialization list 内用它作为 base class 修饰符。

```
template<class T> class Widget;
template<typename T> class Widget;
```

上述代码中 class 与 typename 对于 c++ 而言，意义完全相同。但是推荐，当按时参数并非一定是个 class 类型时使用 typename。

但是并非任何时候 class 总是与 typename 等价。

嵌套从属类型名

```
template <typename C>
void print2nd(const C& container) {
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin());
    ++iter;
    int val = *iter;
    std::cout << value;
  }
}
```

上面这段代码接收一个 STL container，并且输出第二号元素。这里面有几个概念：

- 从属名称: template 中出现的名称如果相依于某个 template 参数，就称为从属名称，例如 C 就依赖于 C 类型。
- 嵌套从属名称: 如果丛书名称在 class 内呈现嵌套状，就称为嵌套从属名称，例如 C::const_iterator。
- 非从属名称: 不依赖任何 template 参数的名称，例如 int。

嵌套从属名称可能会导致解析困难的问题，例如

```
template <typename C>
void print2nd(const C& container) {
  C::const_iterator* x;
}
```

上面这段代码如果 C::const_iterator 是一个类，那就没有什么问题。而如果不幸的是传入的类型 C 恰好声明了一个 static 变量，变量名称叫做 const_iterator。那这就变成了一个相乘行为。这显然 和预期不符。同样的，上面的 `C::const_iterator iter(container.begin())` 也会是一个非法 语句。

想要矫正这个行为可以通过 typename 来限制它只能是一个类型。

```
template <typename C>
void print2nd(const C& container) {
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());
  }
}
```

一般性规则是：任何时候当你想要在 template 中指涉一个嵌套从属类型名称，它就必须在紧邻它的前一 个位置放上关键字 typename。

> [!warning] 这一一般性规则的例外是，typename 不剋出现在 base classes list 内的嵌套从属类型之前，也 不可以出现在 member initialization list 中作为 base class 修饰符。

```
template <typename T>
class Derived : public Base<T>::Nested { // base class list 中不允许 typename
public:
  explicit Derived(int x) : Base<T>::Nested(x) { // initialization list 中不允许修饰 base class
    typename Base<T>::Nested temp;
  }
};
```

最后一个例子是 typedef typename 连用。

```
tempalte <typename IterT>
void workWithIterator(IterT iter) {
  typedef typename std::iterator_traits<IterT>::value_type value_type; // IterT 如果是 vector<string>::iterator，那么 value_type 就是 string
  value_type temp(*iter);
}
```

条款43: 学习处理模板化基类内名称

- 可在 derived class templates 内通过 “this->” 指涉 base class templates 内的成员名称，或借由一个明白写出的 "base class 资格修饰符" 完成。

如果编译期我们有足够的信息来决定哪些信息会如何处理，就可以采用基于 template 的方法。下面是一 个例子：

```
class CompanyA {
public:
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
};

class CompanyB {
public:
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
};

class MsgInfo {};
template <typename Company>
class MsgSender {
public:
  void sendClear(const MsgInfo& info) {
    std::string msg;
    Company c;
    c.sendCleartext(msg);
  }
  void SendSecret(const MsgInfo& info) {}
}
```

在上面这段代码的基础上，如果还需要扩展功能，例如每次送出信息时都进行日志，则很容易在该代码上进 行扩充。

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    sendClear(info); // 无法通过编译
    // 传送后写 log
  }
};
```

虽然这样写非常合理，但是 sendClear 无法通过编译。因为在编译期，编译器并不清楚继承的 MsgSender 中 Company 是 CompanyA 还是 CompanyB。

另一个问题在于，如果进行了全特化，那这类继承又会产生错误。例如，CompanyZ 只能发送加密数据，于 是设计 特化版 的 MsgSender。

```
template <>
class MsgSender<CompanyZ> { // 删除了 sendClear 方法，添加了 sendSecret 方法
public:
  void sendSecret(const MsgInfo& info) {}
};
```

在具有全特化的情况下，再次考察 LoggingMsgSender 类，你会发现如果传入 CompanyZ，那么上面这 段代码注定失败，因为 MsgSender 不具备 sendClear 方法。这也是为什么编译器不允许 上述代码通过编译的原因。

为了编译成功，可以使用下列手段。

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    this->sendClear(info); // 假设 sendClear 被继承，这就会成功通过编译
    // 传送后写 log
  }
};
```

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    sendClear(info); // 可以通过，直接告诉编译器我们要调用的函数，而不让编译器自己去
base class 中寻找
    // 传送后写 log
  }
};
```

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    MsgSender<Company>::sendClear(info); // 可以通过
    // 传送后写 log
  }
};
```

第三种实现方式通常是最不恰当的实现方式，因为如果被 virtual 修饰，以明确的名称调用会关闭 “virtual 绑定行为”。

上述这三种方法实际上都是 对编译器承诺 “base class template 的任何特化版本都将支持其泛化版本 所提供的接口”。但是面对 CampanyZ 这种违背承诺的行为，编译器仍会编译失败。

条款44: 将与参数无关的代码抽离 templates

- templates 生成多个 classes 和 多个函数，所以任何 template 代码都不应该于某个造成膨胀的 template 参数产生相依关系。
- 因非类型模板参数而造成的代码膨胀，往往可以消除，做法是以函数参数或 class 成员变量替换 template 参数。
- 因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述的具现类型共享实现码。

尽管 template 可以节省时间且避免代码重复，但是有时候也可能会导致代码膨胀。想要防止这种问题， 可以进行共性与变性分析。

在编写 template 时，也要进行重复代码的判断，这种判断并不像 non-template 这样明确。

```
template<typename T, std::size_t n>
class SquareMatrix {
public:
  void insert();
};

SquareMatrix<double, 5> sm1;
sm1.insert();
SquareMatrix<double, 10> sm2;
sm2.insert();
```

上面这段代码便是存在重复，对于常量 5 和 10 来说，实际只需要一个传入参数的函数来实现即可。

```
template<typename T>
class SquareMatrixBase {
protected:
  void insert(std::size_t matrixSize);
};

template<typename T, std::size_t n>
class SquareMatrix : private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::insert;
public:
  void insert() { this->insert(n); };
};
```

使用这种结构进行设计可以尽量避免 derived classes 代码重复。这里使用 this->insert(n) 是为 了解决 模板化基类内的函数名称会被 derived class 掩盖的问题。但是上述结构仍然具有一些棘手的 问题而没有解决。那就是 SquareMatrixBase 如何修改矩阵数据呢？这部分内容只有 derived class 知道。

一个办法是，为 SquareMatrixBase::insert 添加新的指针参数，但是如果有其他函数你也要这样做。 另外一种办法是在 SquareMatrixBase 存储一个指针，指针指向矩阵所在的内存。

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T* pMem) : size(n), pData(pMem) {}
  void setDataPtr(T* ptr) { pData = ptr; }
  void insert(std::size_t matrixSize);
private:
  std::size_t size;
  T* pData;
};

template<typename T, std::size_t n>
class SquareMatrix : private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::insert;
public:
  SquareMatrix() : SquareMatrixBase<T>(n, data) {}
  void insert() { this->insert(n); }
private:
  T data[n * n];
};
```

条款45: 运用成员函数模板接受所有兼容类型

- 请使用 member function templates 生成 “可接受所有兼容类型” 的函数。
- 如果你声明 member template 用于“泛化 copy 构造”或者 “泛化 assignment 操作”，你还是需要声明正常的 copy 构造函数和 copy assignment 操作符。

在使用指针时，推荐使用智能指针，它能保障自动删除 heap-based 资源。但是有时候，智能指针并不会 像真实指针那样完成工作，例如 ++ 操作。因为真实指针支持隐式转换，特别是 derived class 转换为 base class。

```
class Top {};
class Middle : public Top {};
class Bottom : public Middle {};
Top* pt1 = new Middle;
Top* pt2 = new Bottom;
const Top* pct2 = pt1;
```

```
template<typename T>
class SmartPtr {
public:
  explicit SmartPtr(T* realPtr);
};

SmartPtr<Top> pt1 = SmartPtr<Middle>(new Middle);
```

```
SmartPtr<Top> pt2 = SmartPtr<Bottom>(new Bottom);
SmartPtr<const Top> pct1 = pt1;
```

上面这两段代码的对比就有这种情况，下面这段代码是不成立的。因为同一个 template 的不同具现体之 间并没有与生俱来的关系，即，不像继承关系那样。为了能够获得我们希望的 SmartPtr class 之间的 转换能力，我们必须将他们明确编写出来。

Templates 和 泛型编程

如果仅仅通过单一的构造函数来实现，那工程量显然是巨大的。任何新增的一种类型都会需要你通过扩充构 造函数来实现。

我们实际需要的是一个构造模板，这种模板是所谓的 member function templates，其作用是为了 class 生成 copy 构造函数。

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other);
};
```

上面这段代码被称为泛化 copy 构造函数。其中的参数没有声明为 explicit，是因为原始指针类型之间 的转换是隐式转换，无需明白写出的转型动作。

但是，上面这段代码的功能与我们所需的功能有部分冲突，因此我们需要扩充并限制。首先是我们希望安全 的隐式转换发生，而非任意的转换，例如 Base class 转向 derived class 或者 int* 转向 double*。其次是，我们希望能够提供和 std::auto_ptr 等标准的智能指针相同的方法，例如 get。

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other) : heldPtr(other.get()) {}
  T* get() const { return heldPtr; }
private:
  T* heldPtr;
};
```

上面这段既提供了通用方法，又保证了合理的隐式转换的发生。

另外需要注意的一点是，在 class 内声明泛化的 copy 构造函数并不会阻止编译期生成自己的 copy 构 造函数。所以如果你想自己控制 copy 构造的方方面面，就必须同时声明泛化 copy 构造函数和 “正常” 的 copy 构造函数。

条款46: 需要类型转换时请为模板定义非成员函数

- 当我们编写一个 class template，而它所提供之 “与此 template 相关的” 函数支持 “所有参数之隐式类型转换” 时，请将那些函数定义为 “class template 内部的 friend 函数”。

在条款24中讨论了为什么只有 non-member 函数才有能力 在所有实参身上实施隐式类型转换。但是在模 板化过程中，条款24似乎就不再适用了。

```
template<typename T>
class Rational {
public:
    Rational (const T& numerator = 0, const T& denominator = 1) :
_numerator(numerator),
      _denominator(denominator) {}
    const T numerator() const { return _numerator; }
    const T denominator() const { return _denominator; }
private:
    T _numerator;
    T _denominator;
};

template <typename T>
const Rational<T> operator*(const Rational<T>& lhs, const Rational<T>& rhs) {
  return Rational(lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator());
}

Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2;
```

上述代码无法通过编译，它并没有像非模板那样按照预期运行。这主要是因为，模板的运行过程是先根据传 入的参数进行对模板类型 T 的推算，而在这个过程中并不会进行隐式类型转换。

一种解决这种问题的思路是，利用 friend 来将 non-member 函数声明在 Rational 内，这样就 不再需要根据后面的参数进行推算了，这样也就支持了混合式调用了。

```
template<typename T>
class Rational {
public:
  friend const Rational operator*(const Rational& lhs, const Rational& rhs) {
    return Rational(
      lhs.numerator() * rhs.numerator(),
      lhs.denominator() * rhs.denominator()
    );
  }
};

Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2;
```

主义上面并没有将 operator* 定义在 Rational 外部，因为那会导致链接失败，这种情况只会发生在 template 领域中，c 和 OOC 中不会因此而链接失败。为了解决这个问题，我们选在将函数声明和定义 合并在 Rational 内部。这种 friend 的使用方式和往常不同，这里是为了在 class 内部声明一个 non-member 函数而可选的唯一办法。

写在内部，则意味着也就成为了一个 inline 函数。按照 条款30 所说的那样，如果函数内部过于复杂， 那可以通过调用辅助函数来实现（本案例中函数已经非常简单了，其实不用辅助函数也可以）。

```
template <typename T> class Rational;

template <typename T>
const Rational<T> doMultiply(const Rational<T>& lhs, const Rational<T>& rhs) {
  return Rational<T>(
    lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator()
  );
}

template<typename T>
class Rational {
public:
  friend const Rational operator*(const Rational& lhs, const Rational& rhs) {
    return doMultiply(lhs, rhs);
  }
};
```

作为一个 template，doMultiply 就不再支持混合式乘法了，但是 friend operator* 已经支持混合 式调用了，那也就无需担心 doMultiply 是否支持混合式乘法的问题了。

条款47: 请用 traits classes 表现类型信息

- traits classes 使得 “类型相关信息” 在编译期可用。它们以 templates 和 “template 特化” 来实现。
- 整合重载技术后，traits classes 有可能在编译期对类型执行 if else 测试

使用 traits class 是一种非常常见的手段，以 STL 中的 工具性 templates——advance 为例。

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d); // 将迭代器像前移动 d 单位。
```

在 advance 的实现上，需要区别 random access 迭代器，或者其他类型迭代器。因为只有前者可以 进行 += 操作。

STL 迭代器分类

- input 迭代器：只能向前移动，一次一步，可读但只可读一次。代表：istream_iterators。
- output 迭代器：只能向前移动，一次一步，可写但只可写一次。代表：ostream_iterators。

- forward 迭代器：可以同时完成 input/output 迭代器工作，且可读可写多次。
- Bidirectional 迭代器：除了向前移动还可以向后移动。
- random access 迭代器：在 Bidirectional 迭代器的基础上，可以执行“迭代器算术”。

这五类迭代器提供了专属的卷标结构(tag struct)，并且之间的继承关系是：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

现在回到 advance 之上，我们已经清楚的知道了 random access 迭代器可以直接进行运算，因此不再 需要像其他类型迭代器一样进行多遍递增递减操作。那么如何来判断一个 iter 是否为 random access 迭代器呢？使用 traits。

traits 并不是 c++ 关键字或者一个预先定义好的构件，而是一个技术。按照这种技术写成的 templates 在标准库中有若干个，其中针对迭代器的被命名为 iterator_traits。

```
template <typename IterT>
struct iterator_traits;
```

如何让自己所实现的迭代器能够运用这个 traits 呢？只需如下编写代码。

```
template <>
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
  };
};
```

然后，iterator_traits 会将 iterator class 中嵌入的 typedef 进行读取。

```
template <typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator::iterator_category iterator_category;
};
```

虽然上面这种方式对于用户自定义的 iterator 是成立的，但是对指针却行不通。为此，iterator_traits 特别针对指针提供了一个偏特化版本。

```
template<typename IterT>
struct iterator_traits<IterT*> {
  typedef random_access_iterator_tag iterator_category; // 指针 和 random access
的行为类似
};
```

如何实现一个 traits class

- 确认若干希望将来可以取得的类型相关信息，例如对 iterator 而言，希望将来可以取得分类。
- 为该信息选择一个名称。
- 提供一个 template 和一组特化版本，内涵你希望支持的类型和相关信息。

有了 iterator_traits 如何实现一个 advance。可能你会选择使用 if-else 来进行逐一判断。但是 这种效率并不高。更好的方法是使用和 template 一样在编译期进行判断的方法来实现。一种可行的做法 是重载。

```
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::random_access_iterator_tag /*忽略变量名，
因为后续用不到*/) {
  iter += d;
};
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::bidirectional_iterator_tag /*忽略变量名，
因为后续用不到*/) {
  if (d >= 0) { while(d--) ++iter; }
  else { while(d++) --iter; }
};
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::input_iterator_tag /*忽略变量名，因为后续
用不到*/) {
  if (d < 0)
    throw std::out_of_range("Negative distance");
  while(d--) ++iter;
};


template <typename IterT, typename DistT>
void advance(IterT& iter, DistT d) {
  doAdvance(iter, d, typename std::iterator_traits<IterT>::iterator_category());
}
```

现在，我们了解了如何使用一个 traits class。

- 建立一组重载函数或者函数模板，彼此之间的差异只在于各自的 traits 参数。令每个函数实现码与其接受之 traits 信息相应和。
- 建立一个控制函数或者函数模板，它调用上述这些"劳工函数"并传递 traits class 所提供的信息。

Traits 广泛用于标注程序库。包括 iterator_traits，它不仅提供了上述类型功能，还提供了四份迭 代器相关的信息，最有用的是 value_type。此外还有 char_traits 用来保存字符类型的相关信息，以 及 numeric_limits 用来保存数值类型的相关信息。

条款48: 认识 template 元编程

- TMP 可将工作由运行期移往编译期，因而得以实现早期错误侦测和更高的执行效率。
- TMP 可被用来生成 “基于政策选择组合” 的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码。

模板元编程 (Template metaprogramming, TMP) 是编写 template-based c++ 程序并执行于编译 期的过程。它是以 c++ 写成、执行于 c++ 编译器内的程序。

TMP有两个伟大的效力：

1. 它让事情变得更容易。
2. 由于 TMP 执行于 c++ 编译期，因此可将工作从运行期转移到编译期。

一个好的例子是上一条款所设计的 advance 函数，使用 TMP 将原本通过运行时 if-else 的方法，通过 重载的方法来让其在编译期实现。

针对 TMP 而设计的程序库 (Boosts's MPL) 提供更高层级的语法。

再从循环角度来看看 TMP 如何运作。TMP 并不提供真正的循环，而是使用递归方案来实现。TMP 的递归 甚至不是正常种类，因为 TMP 循环并不涉及递归函数调用，而是涉及 “递归模板具现化”。

使用 TMP 实现阶乘

```
template <unsigned n>
struct Factorial {
  enum { value = n * Factorial<n - 1>::value };
};

template <>
struct Factorial<0> { // 全特化 0! = 1
  enum { value = 1 };
};

int main() {
  std::cout << "5! = " << Factorial<5>::value << std::endl;
  std::cout << "10! = " << Factorial<10>::value << std::endl;
}
```

使用 Factorial::value 就可以直接得到 n 阶阶乘。

下面给出三个通过 TMP 能实现的目标实例:

- 确保量度单位正确。例如质量、距离、时间、速度的关系。将一个质量赋值给速度是不正确的，但是距离除以时间赋值给速度则是正确的。使用 TMP 可以在编译期保证这种约束。
- 优化矩阵运算。原有的矩阵乘法使用 operator* 来执行，必须返回新对象。而在多个矩阵连乘的过程中则会创建多个临时对象。通过 TMP 来实现，就有可能消除这些临时对象，并合并循环。

- 可以生成客户定制之设计模式实现品。设计模式例如 Strategy, Observer, Visitor 等等都有多种实现方式。运用所谓的 policy-based design 之 TMP-based 技术，有可能产生一些 templates 用来表述独立的设计选项，然后可以任意结合它们，导致模式实现品带着客户指定的行为。这项技术已经被用来让若干 templates 实现出只能指针的行为策略，用以编译期生成数以百计不同的只能指针类型。