中文的C++ Template的教学指南。与知名书籍C++ Templates不同，该系列教程将C++ Templates作为一门图灵完备的语言来讲授，以求帮助读者对Meta-Programming融会贯通。(正在施工中)

Billy1900 Update ReadMe.md (#74) bd89772 · 11 months ago

.gitattributes	Add list of content	12 years ago
.gitignore	Add list of content	12 years ago
CppTemplateTutorial.cpp	更新了“名称查找”一节。	10 years ago
CppTemplateTutorial.sln	Add list of content	12 years ago
CppTemplateTutorial.vcxproj	更新了“名称查找”一节。	10 years ago
CppTemplateTutorial.vcxproj.filters	List of content changed.	12 years ago
QuickSort.cpp	Update QuickSort.cpp	9 years ago
ReadMe.md	Update ReadMe.md (#74)	11 months ago
stdafx.cpp	Add list of content	12 years ago
stdafx.h	Add list of content	12 years ago
targetver.h	Add list of content	12 years ago

C++ Template 进阶指南

章节目录由VSCode插件Markdown All in One生成。

1 前言

README

1.1. C++另类简介：比你用的复杂，但比你想的简单

C++似乎从它为世人所知的那天开始便成为天然的话题性编程语言。在它在周围有着形形色色的赞美与贬低之词。当我在微博上透露欲写此文的意愿时，也收到了很多褒贬不一的评论。作为一门语言，能拥有这么多使用并恨着它、使用并畏惧它的用户，也算是语言丛林里的奇观了。

C++之所以变成一门层次丰富、结构多变、语法繁冗的语言，是有着多层次的原因的。Bjarne在《The Design and Evolution of C++》一书中，详细的解释了C++为什么会变成如今（C++98/03）的模样。这本书也是我和陈梓瀚一直对各位已经入门的新手强烈推荐的一本书。通过它你多少可以明白，C++的诸多语法要素之所以变成如今的模样，实属迫不得已。

模板作为C++中最有特色的语言特性，它堪称玄学的语法和语义，理所应当的成为初学者的梦魇。甚至很多工作多年的人也对C++的模板部分保有充分的敬畏。在多数的编码标准中，Template俨然和多重继承一样，成为了一般程序员（非程序库撰写者）的禁区。甚至运用模板较多的Boost，也成为了“众矢之的”。

但是实际上C++模板远没有想象的那么复杂。我们只需要换一个视角：在C++03的时候，模板本身就可以独立成为一门“语言”。它有“值”，有“函数”，有“表达式”和“语句”。除了语法比较蹩脚外，它既没有指针也没有数组，更没有C++里面复杂的继承和多态。可以说，它要比C语言要简单的多。如果我们把模板当做是一门语言来学习，那只需要花费学习OO零头的时间即可掌握。按照这样的思路，可以说在各种模板书籍中出现的多数技巧，都可以被轻松理解。

简单回顾一下模板的历史。87年的时候，泛型（Generic Programming）便被纳入了C++的考虑范畴，并直接导致了后来模板语法的产生。可以说模板语法一开始就是为了在C++中提供泛型机制。92年的时候，Alexander Stepanov开始研究利用模板语法制作程序库，后来这一程序库发展成STL，并在93年被接纳入标准中。

此时不少人以为STL已经是C++模板的集大成之作，C++模板技止于此。但是在95年的《C++ Report》上，John Barton和Lee Nackman提出了一个矩阵乘法的模板示例。可以说元编程在那个时候开始被很多人所关注。自此篇文章发表之后，很多大牛都开始对模板产生了浓厚的兴趣。其中对元编程技法贡献最大的当属Alexandrescu的《Modern C++ Design》及模板程序库Loki。这一2001年发表的图书间接地导致了模板元编程库的出现。书中所使用的Typelist等泛型组件，和Policy等设计方法令人耳目一新。但是因为全书用的是近乎Geek的手法来构造一切设施，因此使得此书阅读起来略有难度。

2002年出版的另一本书《C++ Templates》，可以说是在Template方面的集大成之作。它详细阐述了模板的语法、提供了和模板有关的语言细节信息，举了很多有代表性例子。但是对于模板新手来说，这本书细节如此丰富，让他们随随便便就打了退堂鼓缴械投降。

本文的写作初衷，就是通过"编程语言"的视角，介绍一个简单、清晰的"模板语言"。我会尽可能地将模板的诸多要素连串起来，用一些简单的例子帮助读者学习这门"语言"，让读者在编写、阅读模板代码的时候，能像 `if(exp) { dosomething(); }` 一样的信手拈来，让"模板元编程"技术成为读者牢固掌握、可举一反三的有用技能。

1.2. 适宜读者群

因为本文并不是用于C++入门，例子中也多少会牵涉一些其它知识，因此如果读者能够具备以下条件，会读起来更加轻松：

- 熟悉C++的基本语法；
- 使用过STL；
- 熟悉一些常用的算法，以及递归等程序设计方法。

此外，尽管第一章会介绍一些Template的基本语法，但是还是会略显单薄。因此也希望读者能对C++ Template最基本语法形式有所了解和掌握；如果会编写基本的函数模板和类模板那就更好了。

诚如上节所述，本文并不是《C++ Templates》的简单重复，与《Modern C++ Design》交叠更少。从知识结构上，我建议大家可以先读本文，再阅读《C++ Templates》获取更丰富的语法与实现细节，以更进一步；《Modern C++ Design》除了元编程之外，还有很多的泛型编程示例，原则上泛型编程的部分与我所述的内容交叉不大，读者在读完1-3章了解模板的基本规则之后便可阅读《MCD》的相应章节；元编程部分（如Typelist）建议在阅读完本文之后再行阅读，或许会更易理解。

1.3. 版权

本文是随写随即同步到Github上，因此在行文中难免会遗漏引用。本文绝大部分内容应是直接承出我笔，但是也不定会有他山之石。所有指涉内容我会尽量以引号框记，或在上下文和边角注记中标示，如有遗漏烦请不吝指出。

1.4. 推荐编译环境

C++编译器众多，且对模板的支持可能存在细微差别。如果没有特别强调，本书行文过程中，使用了下列编译器来测试文中提供的代码和示例：

- Clang 14.0.3; 15.0 (amd64)
- Visual Studio 2022 19.2+ (amd64)

此外，部分复杂实例我们还在文中提供了在线的编译器预览以方便大家阅读和测试。在线编译器参见： gcc.godbolt.org 。

一些示例中用到的特性所对应的C++标准：

特性	标准
std::decay_t	C++ 14

1.5. 体例

1.5.1. 示例代码

```
void SampleCode() {
    // 这是一段示例代码
}
```

1.5.2. 引用

引用自C++标准：

> 1.1.2/1 这是一段引用或翻译自标准的文字

引用自其他图书：

> 《书名》 这是一段引用或翻译自其他图书的文字

1.6. 意见、建议、喷、补遗、写作计划

- 需增加：
  - 模板的使用动机。
  - 增加"如何使用本文"一节。本节将说明全书的体例（强调字体、提示语、例子的组织），所有的描述、举例、引用在重审时将按照体例要求重新组织。
  - 除了用于描述语法的例子外，其他例子将尽量赋予实际意义，以方便阐述意图。
  - 在合适的章节完整叙述模板的类型推导规则。Parameter-Argument, auto variable, decltype, decltype(auto)
  - 在函数模板重载和实例化的部分讲述ADL。
  - 变参模板处应当按照标准（Argument Packing/Unpacking）来讲解。
- 建议：
  - 比较模板和函数的差异性
  - 蓝色：C++14 Return type deduction for normal functions 的分析

2. Template的基本语法

2.1. 什么是模板(Template)

2.2. 类模板 (Class Template) 的基本语法

2.2.1. "模板类"还是"类模板"

2.2.2. Class Template的与成员变量定义

我们来回顾一下最基本的Class Template声明和定义形式：

Class Template声明：

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

Class Template定义：

```
template <typename T> class ClassA
{
    T member;
};
```

`template` 是C++关键字，意味着我们接下来将定义一个模板。和函数一样，模板也有一系列参数。这些参数都被囊括在template之后的 `< >` 中。在上文的例子中， `typename T` 便是模板参数。回顾一下与之相似的函数参数的声明形式：

```
void foo(int a);
```

`T` 则可以类比为函数形参 `a` ，这里的"模板形参" `T` ，也同函数形参一样取成任何你想要的名字； `typename` 则类似于例子中函数参数类型 `int` ，它表示模板参数中的 `T` 将匹配一个类型。除了 `typename` 之外，我们在后面还要讲到，整型也可以作为模板的参数。

在定义完模板参数之后，便可以定义你所需要的类。不过在定义类的时候，除了一般类可以使用的类型外，你还可以使用在模板参数中使用的类型 `T` 。可以说，这个 `T` 是模板的精髓，因为你可以通过指定模板实参，将T替换成你所需要的类型。

例如我们用 `ClassA<int>` 来实例化类模板ClassA，那么 `ClassA<int>` 可以等同于以下的定义：

```
// 注意：这并不是有效的C++语法，只是为了说明模板的作用
typedef class {
    int member;
} ClassA<int>;
```

可以看出，通过模板参数替换类型，可以获得很多形式相同的新类型，有效减少了代码量。这种用法，我们称之为“泛型”（Generic Programming），它最常见的应用，即是STL中的容器类模板。

2.2.3. 模板的使用

对于C++来说，类型最重要的作用之一就是用它去产生一个变量。例如我们定义了一个动态数组（列表）的类模板 `vector` ，它对于任意的元素类型都具有push_back和clear的操作，我们便可以如下定义这个类：

```
template <typename T>
class vector
{
public:
    void push_back(T const&);
    void clear();

private:
    T* elements;
};
```

此时我们的程序需要一个整型和一个浮点型的列表，那么便可以通过以下代码获得两个变量：

```
vector<int> intArray;
vector<float> floatArray;
```

此时我们就可以执行以下的操作，获得我们想要的结果：

```
intArray.push_back(5);
floatArray.push_back(3.0f);
```

变量定义的过程可以分成两步来看：第一步， `vector<int>` 将 `int` 绑定到类模板 `vector` 上，获得了一个“普通的类 `vector<int>` ”；第二步通过“vector”定义了一个变量。 与“普通的类”不同，类模板是不能直接用来定义变量的 —— 毕竟它的名字是“模板”而不是“类”。例如：

```
vector unknownVector; // 错误示例
```

这样就是错误的。我们把通过类型绑定将类模板变成“普通的类”的过程，称之为模板实例化（Template Instantiate）。实例化的语法是：

```
模板名 < [模板实参1，模板实参2，...] >
```

看几个例子：

```
vector<int>
ClassA<double>

template <typename T0, typename T1> class ClassB
{
    // Class body ...
};

ClassB<int, float>
```

当然，在实例化过程中，被绑定到模板参数上的类型（即模板实参）需要与模板形参正确匹配。 就如同函数一样，如果没有提供足够并匹配的参数，模板便不能正确的实例化。

2.2.4. 类模板的成员函数定义

由于C++11正式废弃“模板导出”这一特性，因此在类模板的变量在调用成员函数的时候，需要看到完整的成员函数定义。因此现在的类模板中的成员函数，通常都是以内联的方式实现。 例如：

```
template <typename T>
class vector
{
public:
    void clear()
    {
        // Function body
    }

private:
    T* elements;
};
```

当然，我们也可以将 `vector<T>::clear` 的定义部分放在类型之外，只不过这个时候的语法就显得蹩脚许多：

```
template <typename T>
class vector
{
public:
    void clear();  // 注意这里只有声明
private:
    T* elements;
};

template <typename T>
void vector<T>::clear()  // 函数的实现放在这里
{
    // Function body
}
```

函数的实现部分看起来略微拗口。我第一次学到的时候，觉得

```
void vector::clear()
{
    // Function body
}
```

这样不就行了吗？但是简单想就会知道， `clear` 里面是找不到泛型类型 `T` 的符号的。

因此，在成员函数实现的时候，必须要提供模板参数。此外，为什么类型名不是 `vector` 而是 `vector<T>` 呢？ 如果你了解过模板的偏特化与特化的语法，应该能看出，这里的vector在语法上类似于特化/偏特化。实际上，这里的函数定义也确实是成员函数的偏特化。特化和偏特化的概念，本文会在第二部分详细介绍。

综上，正确的成员函数实现如下所示：

```
template <typename T> // 模板参数
void vector<T> /*看起来像偏特化*/ ::clear() // 函数的实现放在这里
{
    // Function body
}
```

2.3. 函数模板 (Function Template) 入门

2.3.1. 函数模板的声明和定义

函数模板的语法与类模板基本相同，也是以关键字 `template` 和模板参数列表作为声明与定义的开始。模板参数列表中的类型，可以出现在参数、返回值以及函数体中。比方说下面几个例子

```
template <typename T> void foo(T const& v);

template <typename T> T foo();
```

```
template <typename T, typename U> U foo(T const&);

template <typename T> void foo()
{
    T var;
    // ...
}
```

无论是函数模板还是类模板，在实际代码中看起来都是“千变万化”的。这些“变化”，主要是因为类型被当做了参数，导致代码中可以变化的部分更多了。

归根结底，模板无外乎两点：

1. 函数或者类里面，有一些类型我们希望它能变化一下，我们用标识符来代替它，这就是“模板参数”；
2. 在需要这些类型的地方，写上相对应的标识符（“模板参数”）。

当然，这里的“可变”实际上在代码编译好后就固定下来了，可以称之为编译期的可变性。

这里多啰嗦一点，主要也是想告诉大家，模板其实是个很简单的东西。

下面这个例子，或许可以帮助大家解决以下两个问题：

1. 什么样的需求会使用模板来解决？
2. 怎样把脑海中的“泛型”变成真正“泛型”的代码？

```
举个例子：generic typed function ‘add’
```

在我遇到的朋友中，即便如此对他解释了模板，即便他了解了模板，也仍然会对模板产生畏难情绪。毕竟从形式上来说，模板化的类和模板化的函数都要较非模板的版本更加复杂，阅读代码所需要理解的内容也有所增多。

如何才能克服这一问题，最终视模板如平坦代码呢？

答案只有一个：**无他，唯手熟尔。**

在学习模板的时候，要反复做以下的思考和练习：

1. 提出问题：我的需求能不能用模板来解决？
2. 怎么解决？
3. 把解决方案用代码写出来。
4. 如果失败了，找到原因。是知识有盲点（例如不知道怎么将 `T&` 转化成 `T` ），还是不可行（比如试图利用浮点常量特化类模板，但实际上这样做是不可行的）？

通过重复以上的练习，应该可以对模板的语法和含义都有所掌握。如果提出问题本身有困难，或许下面这个经典案例可以作为你思考的开始：

1. 写一个泛型的数据结构：例如，线性表，数组，链表，二叉树；
2. 写一个可以在不同数据结构、不同的元素类型上工作的泛型函数，例如求和；

当然和“设计模式”一样，模板在实际应用中，也会有一些固定的需求和解决方案。比较常见的场景包括：泛型（最基本的用法）、通过类型获得相应的信息（型别萃取）、编译期间的计算、类型间的推导和变换（从一个类型变换成另外一个类型，比如boost::function）。这些本文在以后的章节中会陆续介绍。

2.3.2. 函数模板的使用

我们先来看一个简单的函数模板，两个数相加：

```
template <typename T> T Add(T a, T b)
{
    return a + b;
}
```

函数模板的调用格式是：

```
函数模板名 < 模板参数列表 > ( 参数 )
```

例如，我们想对两个 `int` 求和，那么套用类的模板实例化方法，我们可以这么写：

```
int a = 5;
int b = 3;
int result = Add<int>(a, b);
```

这时我们等于拥有了一个新函数：

```
int Add<int>(int a, int b) { return a + b; }
```

这时在另外一个偏远的程序角落，你也需要求和。而此时你的参数类型是 `float` ，于是你写下：

```
Add<float>(a, b);
```

一切看起来都很完美。但如果你具备程序员的最佳美德——懒惰——的话，你肯定会这样想，我在调用 `Add<int>(a, b)` 的时候， `a` 和 `b` 匹配的都是那个 `T` 。编译器就应该知道那个 `T` 实际上是 `int` 呀？为什么还要我多此一举写 `Add<int>` 呢？ 唔，我想说的是，编译器的作者也是这么想的。所以实际上你在编译器里面写下以下片段：

```
int a = 5;
int b = 3;
int result = Add(a, b);
```

编译器会心领神会地将 `Add` 变成 `Add<int>` 。但是编译器不能面对模棱两可的答案。比如你这么写的话呢？

```
int  a = 5;
char b = 3;
int  result = Add(a, b);
```

第一个参数 `a` 告诉编译器，这个 `T` 是 `int` 。编译器点点头说，好。但是第二个参数 `b` 不高兴了，告诉编译器说，你这个 `T` ，其实是 `char` 。 两个参数各自指导 `T` 的类型，编译器就不知道怎么做了。在Visual Studio 2012下，会有这样的提示：

```
error C2782: 'T _1_2_2::Add(T,T)' : template parameter 'T' is ambiguous
```

好吧，"ambiguous"，这个提示再明确不过了。

不过，只要你别逼得编译器精神分裂的话，编译器其实是非常聪明的，它可以从很多的蛛丝马迹中，猜测到你真正的意图，有如下面的例子：

```
template <typename T> class A {};

template <typename T> T foo( A<T> v );

A<int> v;
foo(v); // 它能准确地猜到 T 是 int.
```

咦，编译器居然绕过了A这个外套，猜到了 `T` 匹配的是 `int` 。编译器是怎么完成这一"魔法"的，我们暂且不表，2.2节时再和盘托出。

下面轮到你的练习时间了。你试着写了很多的例子，但是其中一个你还是犯了疑惑：

```
float data[1024];

template <typename T> T GetValue(int i)
{
    return static_cast<T>(data[i]);
}
```

```
float a = GetValue(0);  // 出错了!
int b = GetValue(1);    // 也出错了!
```

为什么会出错呢？你仔细想了想，原来编译器是没办法去根据返回值推断类型的。函数调用的时候，返回值被谁接受还不知道呢。如下修改后，就一切正常了：

```
float a = GetValue<float>(0);
int b = GetValue<int>(1);
```

嗯，是不是so easy啊？嗯，你又信心满满的做了一个练习：

你要写一个函数模板叫 `c_style_cast` ，顾名思义，执行的是C风格的转换。然后出于方便起见，你希望它能和 `static_cast` 这样的内置转换有同样的写法。于是你写了一个use case。

```
DstT dest = c_style_cast<DstT>(src);
```

根据调用形式你知道了，有 `DstT` 和 `SrcT` 两个模板参数。参数只有一个， `src` ，所以函数的形参当然是这么写了： `(SrcT src)` 。实现也很简单， `(DstT)v` 。

我们把手上得到的信息来拼一拼，就可以编写自己的函数模板了：

```
template <typename SrcT, typename DstT> DstT c_style_cast(SrcT v)
{
    return (DstT)(v);
}

int v = 0;
float i = c_style_cast<float>(v);
```

嗯，很Easy嘛！我们F6一下...咦！这是什么意思！

```
error C2783: 'DstT _1_2_2::c_style_cast(SrcT)' : could not deduce template argument for 'DstT'
```

然后你仔细的比较了一下，然后发现 ... 模板参数有两个，而参数里面能得到的只有 `SrcT` 一个。结合出错信息看来关键在那个 `DstT` 上。这个时候，你死马当活马医，把模板参数写完整了：

```
float i = c_style_cast<int, float>(v);
```

嗯，很顺利的通过了。难道C++不能支持让参数推导一部分模板参数吗？

当然是可以的。只不过在部分推导、部分指定的情况下，编译器对模板参数的顺序是有限制的：**先写需要指定的模板参数，再把能推导出来的模板参数放在后面。**

在这个例子中，能推导出来的是 `SrcT` ，需要指定的是 `DstT` 。把函数模板写成下面这样就可以了：

```
template <typename DstT, typename SrcT> DstT c_style_cast(SrcT v)       // 模板参数 DstT 需要人肉指定，放前面。
{
    return (DstT)(v);
}

int v = 0;
float i = c_style_cast<float>(v);  // 形象地说，DstT会先把你指定的参数吃掉，剩下的就交给编译器从函数参数列表中推导啦。
```

2.4. 整型也可是Template参数

模板参数除了类型外（包括基本类型、结构、类类型等），也可以是一个整型数（Integral Number）。这里的整型数比较宽泛，包括布尔型，不同位数、有无符号的整型，甚至包括指针。我们将整型的模板参数和类型作为模板参数来做一个对比：

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

```
template <int      V> class TemplateWithValue;
```

我想这个时候你也更能理解 `typename` 的意思了：它相当于是模板参数的“类型”，告诉你 `T` 是一个 `typename` 。

按照C++ Template最初的想法，模板不就是为了提供一个类型安全、易于调试的宏吗？有类型就够了，为什么要引入整型参数呢？考虑宏，它除了代码替换，还有一个作用是作为常数出现。所以整型模板参数最基本的用途，也是定义一个常数。例如这段代码的作用：

```
template <typename T, int Size> struct Array
{
    T data[Size];
};

Array<int, 16> arr;
```

便相当于下面这段代码：

```
class IntArrayWithSize16
{
    int data[16]; // int 替换了 T, 16 替换了 Size
};

IntArrayWithSize16 arr;
```

其中有一点需要注意，因为模板的匹配是在编译的时候完成的，所以实例化模板的时候所使用的参数，也必须要在编译期就能确定。例如以下的例子编译器就会报错：

```
template <int i> class A {};

void foo()
{
    int x = 3;
    A<5> a; // 正确!
    A<x> b; // error C2971: '_1_3::A' : template parameter 'i' : 'x' : a local variable cannot be used as a non-type argument
}
```

因为x不是一个编译期常量，所以 `A<x>` 就会告诉你，x是一个局部变量，不能作为一个模板参数出现。

嗯，这里我们再来写几个相对复杂的例子：

```
template <int i> class A
{
public:
    void foo(int)
    {
    }
};
template <uint8_t a, typename b, void* c> class B {};
template <bool, void (*a)()> class C {};
template <void (A<3>::*a)(int)> class D {};

template <int i> int Add(int a) // 当然也能用于函数模板
{
    return a + i;
}

void foo()
{
    A<5> a;
    B<7, A<5>, nullptr> b; // 模板参数可以是一个无符号八位整数，可以是模板生成的类；可以是一个指针。
    C<false, &foo> c;      // 模板参数可以是一个bool类型的常量，甚至可以是一个函数指针。
    D<&A<3>::foo> d;       // 丧心病狂啊！它还能是一个成员函数指针！
    int x = Add<3>(5);     // x == 8。因为整型模板参数无法从函数参数获得，所以只能是手工指定啦。
}

template <float a> class E {}; // ERROR: 别闹！早说过只能是整数类型的啦！
```

当然，除了单纯的用作常数之外，整型参数还有一些其它的用途。这些“其它”用途最重要的一点是让类型也可以像整数一样运算。《Modern C++ Design》给我们展示了很多这方面的例子。不过你不用急着去阅读那本天书，我们会在做好足够的知识铺垫后，让你轻松学会这些招数。

2.5. 模板形式与功能是统一的

第一章走马观花的带着大家复习了一下C++ Template的基本语法形式，也解释了包括 `typename` 在内，类/函数模板写法中各个语法元素的含义。形式是功能的外在体现，介绍它们也是为了让大家能理解到，模板之所以写成这种形式是有必要的，而不是语言的垃圾成分。

从下一章开始，我们便进入了更加复杂和丰富的世界：讨论模板的匹配规则。其中有令人望而生畏的特化与偏特化。但是，请相信我们在序言中所提到的：将模板作为一门语言来看待，它会变得有趣而简单。

3. 模板元编程基础

3.1. 编程，元编程，模板元编程

技术的学习是一个登山的过程。第一章是最为平坦的山脚道路。而从这一章开始，则是正式的爬坡。无论是我写作还是你阅读，都需要付出比第一章更多的代价。那么问题就是，付出更多的精力学习模板是否值得？

这个问题很功利，但是一针见血。因为技术的根本目的在于解决需求。那C++的模板能做什么？

一个高（树）大（新）上（风）的回答是，C++里面的模板，犹如C中的宏、C和Java中的自省（restropection）和反射（reflection），是一个改变语言内涵，拓展语言外延的存在。

程序最根本的目的是什么？复现真实世界或人所构想的规律，减少重复工作的成本，或通过提升规模完成人所不能及之事。但是世间之事万千，有限的程序如何重现复杂的世界呢？

答案是“抽象”。论及具体手段，无外乎“求同”与“存异”：概括一般规律，处理特殊情况。这也是软件工程所追求的目标。一般规律概括的越好，我们所付出的劳动也就越少。

同样的，作为脑力劳动的产品，程序本身也是有规律性的。《Modern C++ Design》中的前言就抛出了一连串有代表性的问题：

```
如何撰写更高级的C++程式?
如何应付即使在很干净的设计中仍然像雪崩一样的不相干细节?
如何构建可复用组件，使得每次在不同程式中应用组件时无需大动干戈?
```

我们以数据结构举例。在程序里，你需要一些堆栈。这个堆栈的元素可能是整数、浮点或者别的什么类型。一份整型堆栈的代码可能是：

```
class StackInt
{
public:
    void push(int v);
    int pop();
    int Find(int x)
    {
        for(int i = 0; i < size; ++i)
        {
            if(data[i] == x) { return i; }
        }
    }
    // ... 其他代码 ...
};
```

如果你要支持浮点了，那么你只能将代码再次拷贝出来，并作如下修改：

```
class StackFloat
{
public:
    void push(float v);
    float pop();
    int Find(float x)
    {
        for(int i = 0; i < size; ++i)
        {
```

```
            if(data[i] == x) { return i; }
        }
    }
    // ... 其他代码 ...
};
```

当然也许你觉得这样做能充分体会代码行数增长的成就感。但是有一天，你突然发现：呀， `Find` 函数实现有问题了。怎么办？这个时候也许你只有两份这样的代码，那好说，一一去修正就好了。如果你有十个呢？二十个？五十个？

时间一长，你就厌倦了这样的生活。你觉得每个堆栈都差不多，但是又有点不一样。为了这一点点不一样，你付出了太多的时间。吃饭的时间，泡妞的时间，睡觉的时间，看岛国小电影顺便练习小臂力量的时间。

于是便诞生了新的技术，来消解我们的烦恼。

这个技术的名字，并不叫“模板”，而是叫“元编程”。

元（meta）无论在中文还是英文里，都是个很“抽象（abstract）”的词。因为它的本意就是“抽象”。元编程，也可以说就是“编程的抽象”。用更好理解的说法，元编程意味着你撰写一段程序A，程序A会运行后生成另外一个程序B，程序B才是真正实现功能的程序。那么这个时候程序A可以称作程序B的元程序，撰写程序A的过程，就称之为“元编程”。

回到我们的堆栈的例子。真正执行功能的，其实仍然是浮点的堆栈、整数的堆栈、各种你所需要的类型的堆栈。但是因为这些堆栈之间太相似了，仅仅有着些微的不同，我们为什么不能有一个将相似之处囊括起来，同时又能分别体现出不同之处的程序呢？很多语言都提供了这样的机会。C中的宏，C++中的模板，Python中的Duck Typing，广义上将都能够实现我们的思路。

我们的目的，是找出程序之间的相似性，进行“元编程”。而在C++中，元编程的手段，可以是宏，也可以是模板。

宏的例子姑且不论，我们来看一看模板：

```
template <typename T>
class Stack
{
public:
    void push(T v);
    T pop();
    int Find(T x)
    {
        for(int i = 0; i < size; ++i)
        {
            if(data[i] == x) { return i; }
        }
    }
    // ... 其他代码 ...
};

typedef Stack<int>   StackInt;
typedef Stack<float> StackFloat;
```

通过模板，我们可以将形形色色的堆栈代码分为两个部分，一个部分是不变的接口，以及近乎相同的实现；另外一部分是元素的类型，它们是需要变化的。因此同函数类似，需要变化的部分，由模板参数来反映；不变的部分，则是模板内的代码。可以看到，使用模板的代码，要比不使用模板的代码简洁许多。

如果元编程中所有变化的量（或者说元编程的参数），都是类型，那么这样的编程，我们有个特定的称呼，叫“泛型”。

但是你会问，模板的发明，仅仅是为了做和宏几乎一样的替换工作吗？可以说是，也可以说不是。一方面，很多时候模板就是为了替换类型，这个时候作用上其实和宏没什么区别。只是宏是基于文本的替换，被替换的文本本身没有任何语义。只有替换完成，编译器才能进行接下来的处理。而模板会在分析模板时以及实例化模板时时候都会进行检查，而且源代码中也能与调试符号一一对应，所以无论是编译时还是运行时，排错都相对简单。

但是模板和宏也有很大的不同，否则此文也就不能成立了。模板最大的不同在于它是“可以运算”的。我们来举一个例子，不过可能有点牵强。考虑我们要写一个向量逐分量乘法。只不过这个向量，它非常的大。所以为了保证速度，我们需要使用SIMD指令进行加速。假设我们有以下指令可以使用：

```
Int8,16: N/A
Int32  : VInt32Mul(int32x4, int32x4)
Int64  : VInt64Mul(int64x4, int64x4)
Float  : VInt64Mul(floatx2, floatx2)
```

所以对于Int8和Int16，我们需要提升到Int32，而Int32和Int64，各自使用自己的指令。所以我们需要实现下的逻辑：

```
for(v4a, v4b : vectorsA, vectorsB)
{
    if type is Int8, Int16
        VInt32Mul( ConvertToInt32(v4a), ConvertToInt32(v4b) )
    elif type is Int32
        VInt32Mul( v4a, v4b )
    elif type is Float
        ...
}
```

这里的问题就在于，如何根据 `type` 分别提供我们需要的实现？这里有两个难点。首先， `if(type == xxx) {}` 是不存在于C++中的。第二，即便存在根据 `type` 的分配方法，我们也不希望它在运行时branch，这样会变得很慢。我们希望它能按照类型直接就把代码编译好，就跟直接写的一样。

嗯，聪明你果然想到了，重载也可以解决这个问题。

```
GenericMul(int8x4,  int8x4);
GenericMul(int16x4, int16x4);
GenericMul(int32x4, int32x4);
GenericMul(int64x4, int64x4);
// 其它 Generic Mul ...

for(v4a, v4b : vectorsA, vectorsB)
{
    GenericMul(v4a, v4b);
}
```

这样不就可以了吗？

唔，你赢了，是这样没错。但是问题是，我这个平台是你可没见过，它叫 `Deep Thought` ， 特别缺心眼儿，不光有 `int8` ，还有更奇怪的 `int9` ， `int11` ，以及可以代表世间万物的 `int42` 。你总不能为之提供所有的重载吧？这简直就像你枚举了所有程序的输入，并为之提供了对应的输出一样。

好吧，我承认这个例子还是太牵强了。不过相信我，在你阅读完第二章和第三章之后，你会将这些特性自如地运用到你的程序之中。你的程序将会变成体现模板"可运算"威力的最好例子。

3.2. 模板世界的If-Then-Else：类模板的特化与偏特化

3.2.1. 根据类型执行代码

前一节的示例提出了一个要求：需要做出根据类型执行不同代码。要达成这一目的，模板并不是唯一的途径。比如之前我们所说的重载。如果把眼界放宽一些，虚函数也是根据类型执行代码的例子。此外，在C语言时代，也会有一些技法来达到这个目的，比如下面这个例子，我们需要对两个浮点做加法， 或者对两个整数做乘法：

```
struct Variant
{
    union
    {
        int x;
        float y;
    } data;
    uint32 typeId;
};

Variant addFloatOrMulInt(Variant const* a, Variant const* b)
{
    Variant ret;
    assert(a->typeId == b->typeId);
    if (a->typeId == TYPE_INT)
    {
        ret.x = a->x * b->x;
    }
    else
    {
```

```
        ret.y = a->y + b->y;
    }
    return ret;
}
```

更常见的是 `void*`：

```
define BIN_OP(type, a, op, b, result) (*(type *)(result)) = (*(type const *)(a)) op (*(type const*)(b))
void doDiv(void* out, void const* data0, void const* data1, DATA_TYPE type)
{
    if(type == TYPE_INT)
    {
        BIN_OP(int, data0, *, data1, out);
    }
    else
    {
        BIN_OP(float, data0, +, data1, out);
    }
}
```

在C++中比如在 `Boost.Any` 的实现中，运用了 `typeid` 来查询类型信息。和 `typeid` 同属于RTTI机制的 `dynamic_cast`，也经常会用来做类型判别的工作。我想你应该写过类似于下面的代码：

```
IAnimal* animal = GetAnimalFromSystem();

IDog* maybeDog = dynamic_cast<IDog*>(animal);
if(maybeDog)
{
    maybeDog->Wangwang();
}
ICat* maybeCat = dynamic_cast<ICat*>(animal);
if(maybeCat)
{
    maybeCat->Moemoe();
}
```

当然，在实际的工作中，我们建议把需要 `dynamic_cast` 后执行的代码，尽量变成虚函数。不过这个已经是另外一个问题了。我们看到，不管是哪种方法都很难避免 `if` 的存在。而且因为输入数据的类型是模糊的，经常需要强制地、没有任何检查的转换成某个类型，因此很容易出错。

但是模板与这些方法最大的区别并不在这里。模板无论其参数或者是类型，它都是一个编译期分派的办法。编译期就能确定的东西既可以做类型检查，编译器也能进行优化，砍掉任何不必要的代码执行路径。例如在上例中，

```
template <typename T> T addFloatOrMulInt(T a, T b);

// 迷之代码1：用于T是float的情况

// 迷之代码2：用于T是int时的情况
```

如果你运用了模板来实现，那么当传入两个不同类型的变量，或者不是 `int` 和 `float` 变量，编译器就会提示错误。但是如果使用了我们前述的 `Variant` 来实现，编译器可就管不了那么多了。但是，成也编译期，败也编译期。最严重的“缺点”，就是你没办法根据用户输入或者别的什么在运行期间可能发生变化的量来决定它产生、或执行什么代码。比如下面的代码段，它是不成立的。

```
template <int i, int j>
int foo() { return i + j; }
int main()
{
    cin >> x >> y;
    return foo<x, y>();
}
```

这点限制也粉碎了妄图用模板来包办工厂（Factory）甚至是反射的梦想。尽管在《Modern C++ Design》中（别问我为什么老举这本书，因为《C++ Templates》和《Generic Programming》我只是囫囵吞枣读过，基本不记得了)大量运用模板来简化工厂方法；同时C++11/14中的一些机制如Variadic Template更是让这一问题的解决更加彻底。但无论如何，直到C++11/14，光靠模板你就是写不出依靠类名或者ID变量产生类型实例的代码。

所以说，从能力上来看，模板能做的事情都是编译期完成的。编译期完成的意思就是，当你编译一个程序的时候，所有的量就都已经确定了。比如下面的这个例子：

```
int a = 3, b = 5;
Variant aVar, bVar;
aVar.setInt(a);                 // 我们新加上的方法，怎么实现的无所谓，大家明白意思就行了。
bVar.setInt(b);
Variant result = addFloatOrMulInt(aVar, bVar);
```

除非世界末日，否则这个例子里不管你怎么蹦跶，单看代码我们就能知道， `aVar` 和 `bVar` 都一定会是整数。所以如果有合适的机制，编译器就能知道此处的 `addFloatOrMulInt` 中只需要执行 `Int` 路径上的代码，而且编译器在此处也能单独为 `Int` 路径生成代码，从而去掉那个不必要的 `if` 。

在模板代码中，这个“合适的机制”就是指“特化”和“部分特化（Partial Specialization）”，后者也叫“偏特化”。

3.2.2. 特化

我的高中物理老师对我说过一句令我受用至今的话：把自己能做的事情做好。编写模板程序也是一样。当你试图用模板解决问题之前，先撇开那些复杂的语法要素，用最直观的方式表达你的需求：

```
// 这里是伪代码，意思一下

int|float addFloatOrMulInt(a, b)
{
    if(type is Int)
    {
        return a * b;
    }
    else if (type is Float)
    {
        return a + b;
    }
}

void foo()
{
    float a, b, c;
    c = addFloatOrMulInt(a, b);         // c = a + b;

    int x, y, z;
    z = addFloatOrMulInt(x, y);         // z = x * y;
}
```

因为这一节是讲类模板有关的特化和偏特化机制，所以我们不用普通的函数，而是用类的静态成员函数来做这个事情（这就是典型的没事找抽型）：

```
// 这里仍然是伪代码，意思一下，too。
class AddFloatOrMulInt
{
    static int|float Do(a, b)
    {
        if(type is Int)
        {
            return a * b;
        }
        else if (type is Float)
        {
            return a + b;
        }
    }
};

void foo()
{
    float a, b, c;
    c = AddFloatOrMulInt::Do(a, b); // c = a + b;

    int x, y, z;
```

```
    z = AddFloatOrMulInt::Do(x, y); // z = x * y;
}
```

好，意思表达清楚了。我们先从调用方的角度，把这个形式改写一下：

```
void foo()
{
    float a, b, c;
    c = AddFloatOrMulInt<float>::Do(a, b); // c = a + b;

    int x, y, z;
    z = AddFloatOrMulInt<int>::Do(x, y); // z = x * y;
}
```

也许你不明白为什么要改写成现在这个样子。看不懂不怪你，怪我讲得不好。但是你别急，先看看这样改写以后能不能跟我们的目标接近一点。如果我们把 `AddFloatOrMulInt<float>::Do` 看作一个普通的函数，那么我们可以写两个实现出来：

```
float AddFloatOrMulInt<float>::Do(float a, float b)
{
    return a + b;
}

int AddFloatOrMulInt<int>::Do(int a, int b)
{
    return a * b;
}

void foo()
{
    float a, b, c;
    c = AddFloatOrMulInt<float>::Do(a, b); // c = a + b;

    int x, y, z;
    z = AddFloatOrMulInt<int>::Do(x, y); // z = x * y;
}
```

这样是不是就很开心了？我们更进一步，把 `AddFloatOrMulInt<int>::Do` 换成合法的类模板：

```
// 这个是给float用的。
template <typename T> class AddFloatOrMulInt
{
    T Do(T a, T b)
    {
        return a + b;
    }
};

// 这个是给int用的。
template <typename T> class AddFloatOrMulInt
{
    T Do(T a, T b)
    {
        return a * b;
    }
};

void foo()
{
    float a, b, c;

    // 嗯，我们需要 c = a + b;
    c = AddFloatOrMulInt<float>::Do(a, b);
    // ... 觉得哪里不对劲 ...
    // ...
    // ...
    // ...
    // 啊！有两个AddFloatOrMulInt，class看起来一模一样，要怎么区分呢！
}
```

好吧，问题来了！如何要让两个内容不同，但是模板参数形式相同的类进行区分呢？特化！特化（specialization）是根据一个或多个特殊的整数或类型，给出模板实例化时的一个指定内容。我们先来看特化是怎么应用到这个问题上的。

```
// 首先，要写出模板的一般形式（原型）
template <typename T> class AddFloatOrMulInt
{
    static T Do(T a, T b)
    {
        // 在这个例子里面一般形式里面是什么内容不重要，因为用不上
        // 这里就随便给个0吧。
        return T(0);
    }
};

// 其次，我们要指定T是int时候的代码，这就是特化：
template <> class AddFloatOrMulInt<int>
{
public:
    static int Do(int a, int b) //
    {
        return a * b;
    }
};

// 再次，我们要指定T是float时候的代码：
template <> class AddFloatOrMulInt<float>
{
public:
    static float Do(float a, float b)
    {
        return a + b;
    }
};

void foo()
{
    // 这里面就不写了
}
```

我们再把特化的形式拿出来一瞧：这货有点怪啊： `template <> class AddFloatOrMulInt<int>` 。别急，我给你解释一下。

```
// 我们这个模板的基本形式是什么？
template <typename T> class AddFloatOrMulInt;

// 但是这个类，是给T是Int的时候用的，于是我们写作
class AddFloatOrMulInt<int>;
// 当然，这里编译是通不过的。

// 但是它又不是个普通类，而是类模板的一个特化（特例）。
// 所以前面要加模板关键字template，
// 以及模板参数列表
template </* 这里要填什么？ */> class AddFloatOrMulInt<int>;

// 最后，模板参数列表里面填什么？因为原型的T已经被int取代了。所以这里就不能也不需要放任何额外的参数了。
// 所以这里放空。
template <> class AddFloatOrMulInt<int>
{
    // ... 针对Int的实现 ...
};

// Bingo!
```

哈，这样就好了。我们来做一个练习。我们有一些类型，然后你要用模板做一个对照表，让类型对应上一个数字。我先来做一个示范：

```
template <typename T> class TypeToID
{
public:
    static int const ID = -1;
};
```

```
template <> class TypeToID<uint8_t>
{
public:
    static int const ID = 0;
};
```

然后呢，你的任务就是，要所有无符号的整数类型的特化（其实就是 `uint8_t` 到 `uint64_t` 啦），把所有的基本类型都赋予一个ID（当然是不一样的啦）。当你做完后呢，可以把类型所对应的ID打印出来，我仍然以 `uint8_t` 为例：

```
void PrintID()
{
    cout << "ID of uint8_t: " << TypeToID<uint8_t>::ID << endl;
}
```

嗯，看起来挺简单的，是吧。但是这里透露出了一个非常重要的信号，我希望你已经能察觉出来了： `TypeToID` 如同是一个函数。这个函数只能在编译期间执行。它输入一个类型，输出一个ID。

如果你体味到了这一点，那么恭喜你，你的模板元编程已经开悟了。

3.2.3. 特化：一些其它问题

在上一节结束之后，你一定做了许多的练习。我们再来做三个练习。第一，给 `float` 一个ID；第二，给 `void*` 一个ID；第三，给任意类型的指针一个ID。先来做第一个：

```
// ...
// TypeToID 的模板“原型”
// ...

template <> class TypeToID<float>
{
public:
    static int const ID = 0xF10A7;
};
```

嗯， 这个你已经了然于心了。那么 `void*` 呢？你想了想，这已经是一个复合类型了。不错你还是战战兢兢地写了下来：

```
template <> class TypeToID<void*>
{
public:
    static int const ID = 0x401d;
};

void PrintID()
{
    cout << "ID of uint8_t: " << TypeToID<void*>::ID << endl;
}
```

遍译运行一下，对了。模板不过如此嘛。然后你觉得自己已经完全掌握了，并试图将所有C++类型都放到模板里面，开始了自我折磨的过程：

```
class ClassB {};

template <> class TypeToID<void ()>;      // 函数的TypeID
template <> class TypeToID<int[3]>;       // 数组的TypeID
template <> class TypeToID<int (int[3])>; // 这是以数组为参数的函数的TypeID
template <> class TypeToID<int (ClassB::*[3])(void*, float[2])>; // 我也不知道这是什么了，自己看着办吧。
```

甚至连 `const` 和 `volatile` 都能装进去：

```
template <> class TypeToID<int const * volatile * const volatile>;
```

此时就很明白了，只要 <> 内填进去的是一个C++能解析的合法类型，模板都能让你特化。不过这个时候如果你一点都没有写错的话，PrintID 中只打印了我们提供了特化的类型的ID。那如果我们没有为之提供特化的类型呢？比如说double？OK，实践出真知，我们来尝试着运行一下：

```
void PrintID()
{
    cout << "ID of double: " << TypeToID<double>::ID << endl;
}
```

嗯，它输出的是-1。我们顺藤摸瓜会看到， TypeToID 的类模板“原型”的ID是值就是-1。通过这个例子可以知道，当模板实例化时提供的模板参数不能匹配到任何的特化形式的时候，它就会去匹配类模板的“原型”形式。

不过这里有一个问题要理清一下。和继承不同，类模板的“原型”和它的特化类在实现上是没有关系的，并不是在类模板中写了 ID 这个Member，那所有的特化就必须要加入 ID 这个Member，或者特化就自动有了这个成员。完全没这回事。我们把类模板改成以下形式，或许能看的更清楚一点：

```
template <typename T> class TypeToID
{
public:
    static int const NotID = -2;
};

template <> class TypeToID<float>
{
public:
    static int const ID = 1;
};

void PrintID()
{
    cout << "ID of float: " << TypeToID<float>::ID << endl;       // Print "1"
    cout << "NotID of float: " << TypeToID<float>::NotID << endl; // Error! TypeToID<float>使用的特化的类，这个类的实现没有NotID这
    cout << "ID of double: " << TypeToID<double>::ID << endl;     // Error! TypeToID<double>是由类模板实例化出来的，它只有NotID，没
}
```

这样就明白了。类模板和类模板的特化的作用，仅仅是指导编译器选择哪个编译，但是特化之间、特化和它原型的类模板之间，是分别独立实现的。所以如果多个特化、或者特化和对应的类模板有着类似的内容，很不好意思，你得写上若干遍了。

第三个问题，是写一个模板匹配任意类型的指针。对于C语言来说，因为没有泛型的概念，因此它提供了无类型的指针 void* 。它的优点是，所有指针都能转换成它。它的缺点是，一旦转换称它后，你就再也不知道这个指针到底是指向 float 或者是 int 或者是 struct 了。

比如说 copy 。

```
void copy(void* dst, void const* src, size_t elemSize, size_t elemCount, void (*copyElem)(void* dstElem, void const* srcElem))
{
    void const* reader = src;
    void const* writer = dst;
    for(size_t i = 0; i < elemCount; ++i)
    {
        copyElem(writer, reader);
        advancePointer(reader, elemSize); // 把Reader指针往后移动一些字节
        advancePointer(writer, elemSize);
     }
}
```

为什么要提供copyElem，是因为可能有些struct需要深拷贝，所以得用特殊的copy函数。这个在C++98/03里面就体现为拷贝构造和赋值函数。

但是不管怎么搞，因为这个函数的参数只是 void* 而已，当你使用了错误的elemSize，或者传入了错误的copyElem，就必须要到运行的时候才有可能看出来。注意，这还只是有可能而已。

那么C++有了模板后，能否既能匹配任意类型的指针，同时又保留了类型信息呢？答案是显然的。至于怎么写，那就得充分发挥你的直觉了：

首先，我们需要一个 typename T 来指代“任意类型”这四个字：

```
template <typename T>
```

接下来，我们要写函数原型：

```
void copy(?? dest, ?? src, size_t elemCount);
```

这里的 `??` 要怎么写呢？既然我们有了模板类型参数T，那我们不如就按照经验，写 `T*` 看看。

```
template <typename T>
void copy(T* dst, T const* src, size_t elemCount);
```

编译一下，咦，居然通过了。看来这里的语法与我们以前学到的知识并没有什么不同。这也是语言设计最重要的一点原则：一致性。它可以让你辛辛苦苦体验到的规律不至于白费。

最后就是实现：

```
template <typename T>
void copy(T* dst, T const* src, size_t elemCount)
{
    for(size_t i = 0; i < elemCount; ++i)
    {
        dst[i] = src[i];
    }
}
```

是不是简洁了许多？你不需要再传入size；只要你有正确的赋值函数，也不需要提供定制的copy；也不用担心dst和src的类型不匹配了。

最后，我们把函数模板学到的东西，也应用到类模板里面：

```
template <typename T> // 嗯，需要一个T
class TypeToID<T*> // 我要对所有的指针类型特化，所以这里就写T*
{
public:
    static int const ID = 0x80000000;   // 用最高位表示它是一个指针
};
```

最后写个例子来测试一下，看看我们的 `T*` 能不能搞定 `float*`：

```
void PrintID()
{
    cout << "ID of float*: " << TypeToID<float*>::ID << endl;
}
```

哈哈，大功告成。嗯，别急着高兴。待我问一个问题：你知道 `TypeToID<float*>` 后，这里的T是什么吗？换句话说，你知道下面这段代码打印的是什么吗？

```
// ...
// TypeToID 的其他代码，略过不表
// ...

template <typename T> // 嗯，需要一个T
class TypeToID<T*> // 我要对所有的指针类型特化，所以这里就写T*
{
public:
    typedef T           SameAsT;
    static int const ID = 0x80000000; // 用最高位表示它是一个指针
};

void PrintID()
{
    cout << "ID of float*: " << TypeToID< TypeToID<float*>::SameAsT >::ID << endl;
}
```

别急着运行，你先猜。

---------- 这里是给勤于思考的码猴的分割线 ----------

OK，猜出来了吗，T是 `float` 。为什么呢？因为你用 `float *` 匹配了 `T *` ，所以 `T` 就对应 `float` 了。没想清楚的自己再多体会一下。

嗯，所以实际上，我们可以利用这个特性做一件事情：把指针类型的那个指针给"干掉"：

```
template <typename T>
class RemovePointer
{
public:
    typedef T Result;  // 如果放进来的不是一个指针，那么它就是我们要的结果。
};

template <typename T>
class RemovePointer<T*> // 祖传牛皮藓，专治各类指针
{
public:
    typedef T Result;  // 正如我们刚刚讲的，去掉一层指针，把 T* 这里的 T 取出来。
};

void Foo()
{
    RemovePointer<float*>::Result x = 5.0f; // 喏，用RemovePointer后，那个Result就是把float*的指针处理掉以后的结果：float啦。
    std::cout << x << std::endl;
}
```

当然啦，这里我们实现的不算是真正的 `RemovePointer` ，因为我们只去掉了一层指针。而如果传进来的是类似 `RemovePointer<int**>` 这样的东西呢？是的没错，去掉一层之后还是一个指针。 `RemovePointer<int**>::Result` 应该是一个 `int*` ，要怎么才能实现我们想要的呢？聪明的你一定能想到：只要像剥洋葱一样，一层一层一层地剥开，不就好了吗！相应地我们应该怎么实现呢？可以把 `RemovePointer` 的特化版本改成这样（当然如果有一些不明白的地方你可以暂时跳过，接着往下看，很快就会明白的）：

```
template <typename T>
class RemovePointer<T*>
{
public:
    // 如果是传进来的是一个指针，我们就剥夺一层，直到指针形式不存在为止。
    // 例如 RemovePointer<int**>, Result 是 RemovePointer<int*>::Result,
    // 而 RemovePointer<int*>::Result 又是 int，最终就变成了我们想要的 int，其它也是类似。
    typedef typename RemovePointer<T>::Result Result;
};
```

是的没错，这便是我们想要的 `RemovePointer` 的样子。类似的你还可以试着实现 `RemoveConst` , `AddPointer` 之类的东西。

OK，回到我们之前的话题，如果这个时候，我需要给 `int*` 提供一个更加特殊的特化，那么我还得多提供一个：

```
// ...
// TypeToID 的其他代码，略过不表
// ...

template <typename T> // 嗯，需要一个T
class TypeToID<T*>    // 我要对所有的指针类型特化，所以这里就写T*
{
public:
    typedef T SameAsT;
    static int const ID = 0x80000000; // 用最高位表示它是一个指针
};

template <> // 嗯，int* 已经是个具体的不能再具体的类型了，所以模板不需要额外的类型参数了
class TypeToID<int*> // 嗯，对int*的特化。在这里呢，要把int*整体看作一个类型
{
public:
    static int const ID = 0x12345678; // 给一个缺心眼的ID
};

void PrintID()
{
```

```
    cout << "ID of int*: " << TypeToID<int*>::ID << endl;
}
```

嗯，这个时候它会输出0x12345678的十进制（大概？）。 可能会有较真的人说， `int*` 去匹配 `T` 或者 `T*` ，也是合法的。就和你说22岁以上能结婚，那24岁当然也能结婚一样。 那为什么 `int*` 就会找 `int*` ， `float *` 因为没有合适的特化就去找 `T*` ，更一般的就去找 `T` 呢？废话，有专门为你准备的东西你不用，非要自己找事？这就是直觉。 但是呢，直觉对付更加复杂的问题还是没用的（也不是没用，主要是你没这个直觉了）。我们要把这个直觉，转换成合理的规则——即模板的匹配规则。 当然，这个匹配规则是对复杂问题用的，所以我们会到实在一眼看不出来的时候才会动用它。一开始我们只要把握：**模板是从最特殊到最一般形式进行匹配的** 就可以了。

3.3. 即用即推导

3.3.1. 视若无睹的语法错误

这一节我们将讲述模板一个非常重要的行为特点：那就是什么时候编译器会对模板进行推导，推导到什么程度。

这一知识，对于理解模板的编译期行为、以及修正模板编译错误都非常重要。

我们先来看一个例子：

```
template <typename T> struct X {};

template <typename T> struct Y
{
    typedef X<T> ReboundType;                               // 类型定义1
    typedef typename X<T>::MemberType MemberType;       // 类型定义2
    typedef UnknownType MemberType3;                        // 类型定义3

    void foo()
    {
        X<T> instance0;
        typename X<T>::MemberType instance1;
        WTF instance2
        大王叫我来巡山 - + &
    }
};
```

把这段代码编译一下，类型定义3出错，其它的都没问题。不过到这里你应该会有几个问题：

1. 不是 `struct X<T>` 的定义是空的吗？为什么在 `struct Y` 内的类型定义2使用了 `X<T>::MemberType` 编译器没有报错？
2. 类型定义2中的 `typename` 是什么鬼？为什么类型定义1就不需要？
3. 为什么类型定义3会导致编译错误？
4. 为什么 `void foo()` 在MSVC下什么错误都没报？

这时我们就需要请出C++11标准 —— 中的某些概念了。这是我们到目前为止第一次参阅标准。我希望能尽量减少直接参阅标准的次数，因此即便是极为复杂的模板匹配决议我都暂时没有引入标准中的描述。 然而，Template引入的“双阶段名称查找（Two phase name lookup）”堪称是C++中最黑暗的角落 —— 这是LLVM的团队自己在博客上说的 —— 因此在这里，我们还是有必要去了解标准中是如何规定的。

3.3.2. 名称查找：I am who I am

在C++标准中对于“名称查找（name lookup）”这个高大上的名词的诠释，主要集中出现在三处。第一处是3.4节，标题名就叫“Name Lookup”；第二处在10.2节，继承关系中的名称查找；第三处在14.6节，名称解析（name resolution）。

名称查找/名称解析，是编译器的基石。对编译原理稍有了解的人，都知道“符号表”的存在及重要意义。考虑一段最基本的C代码：

```
int a = 0;
int b;
b = (a + 1) * 2;
printf("Result: %d", b);
```

在这段代码中，所有出现的符号可以分为以下几类：

- `int` ： 类型标识符，代表整型；
- `a` , `b` , `printf` ： 变量名或函数名；

- `=`, `+`, `*`：运算符；
- `,`, `;`, `(`, `)`：分隔符；

那么，编译器怎么知道 `int` 就是整数类型， `b=(a+1)*2` 中的 `a` 和 `b` 就是整型变量呢？这就是名称查找/名称解析的作用：它告诉编译器，这个标识符（identifer）是在哪里被声明或定义的，它究竟是什么意思。

也正因为这个机制非常基础，所以它才会面临各种可能的情况，编译器也要想尽办法让它在大部分场合都表现的合理。比如我们常见的作用域规则，就是为了对付名称在不同代码块中传播、并且遇到重名要如何处理的问题。下面是一个最简单的、大家在语言入门过程中都会碰到的一个例子：

```
int a = 0;
void f() {
    int a = 0;
    a += 2;
    printf("Inside <a>: %d\n", a);
}
void g() {
    printf("Outside <a>: %d\n", a);
}
int main() {
    f();
    g();
}

/* ------------ Console Output -----------------
Inside <a>: 2
Outside <a>: 0
--------------- Console Output -------------- */
```

我想大家尽管不能处理所有名称查找中所遇到的问题，但是对一些常见的名称查找规则也有了充分的经验，可以解决一些常见的问题。 但是模板的引入，使得名称查找这一本来就不简单的基本问题变得更加复杂了。 考虑下面这个例子：

```
struct A  { int a; };
struct AB { int a, b; };
struct C  { int c; };

template <typename T> foo(T& v0, C& v1){
    v0.a = 1;
    v1.a = 2;
    v1.c = 3;
}
```

简单分析上述代码很容易得到以下结论：

1. 函数 `foo` 中的变量 `v1` 已经确定是 `struct C` 的实例，所以， `v1.a = 2;` 会导致编译错误， `v1.c = 3;` 是正确的代码；
2. 对于变量 `v0` 来说，这个问题就变得很微妙。如果 `v0` 是 `struct A` 或者 `struct AB` 的实例，那么 `foo` 中的语句 `v0.a = 1;` 就是正确的。如果是 `struct C` ，那么这段代码就是错误的。

因此在模板定义的地方进行语义分析，并不能**完全**得出代码是正确或者错误的结论，只有到了实例化阶段，确定了模板参数的类型后，才知道这段代码正确与否。令人高兴的是，在这一问题上，我们和C++标准委员会的见地一致，说明我们的C++水平已经和Herb Sutter不分伯仲了。既然我们和Herb Sutter水平差不多，那凭什么人家就吃香喝辣？下面我们来选几条标准看看服不服：

> 14.6 **名称解析**（Name resolution）
>
> 1) 模板定义中能够出现以下三类名称：
>
> - 模板名称、或模板实现中所定义的名称；
> - 和模板参数有关的名称；
> - 模板定义所在的定义域内能看到的名称。
>
> ...
>
> 9) ... 如果名字查找和模板参数有关，那么查找会延期到模板参数全都确定的时候。 ...
>
> 10) 如果（模板定义内出现的）名字和模板参数无关，那么在模板定义处，就应该找得到这个名字的声明。...

> 14.6.2 **依赖性名称（Dependent names）**

> 1) ...（模板定义中的）表达式和类型可能会依赖于模板参数，并且模板参数会影响到名称查找的作用域 ... 如果表达式中有操作数依赖于模板参数，那么整个表达式都依赖于模板参数，名称查找延期到**模板实例化时**进行。并且定义时和实例化时的上下文都会参与名称查找。（依赖性）表达式可以分为类型依赖（类型指模板参数的类型）或值依赖。

> 14.6.2.2 **类型依赖的表达式**

> 2) 如果成员函数所属的类型是和模板参数有关的，那么这个成员函数中的 `this` 就认为是类型依赖的。

> 14.6.3 **非依赖性名称（Non-dependent names）**

> 1) 非依赖性名称在**模板定义**时使用通常的名称查找规则进行名称查找。

Working Draft: Standard of Programming Language C++, N3337

知道差距在哪了吗：人家会说黑话。什么时候咱们也会说黑话了，就是标准委员会成员了，反正懂得也不比他们少。不过黑话确实不太好懂 —— 怪我翻译不好的人，自己看原文，再说好懂了人家还靠什么吃饭 —— 我们来举一个例子：

```
int a;
struct B { int v; }
template <typename T> struct X {
    B b;                  // B 是第三类名字，b 是第一类
    T t;                  // T 是第二类
    X* anthor;            // X 这里代指 X<T>，第一类
    typedef int Y;        // int 是第三类
    Y y;                  // Y 是第一类
    C c;                  // C 什么都不是，编译错误。
    void foo() {
       b.v += y;          // b 是第一类，非依赖性名称
       b.v *= T::s_mem;   // T::s_mem 是第二类
                          // s_mem的作用域由T决定
                          // 依赖性名称，类型依赖
    }
};
```

所以，按照标准的意思，名称查找会在模板定义和实例化时各做一次，分别处理非依赖性名称和依赖性名称的查找。这就是"两阶段名称查找"这一名词的由来。只不过这个术语我也不知道是谁发明的，它并没有出现的标准上，但是频繁出现在StackOverflow和Blog上。

接下来，我们就来解决2.3.1节中留下的几个问题。

先看第四个问题。为什么MSVC中，函数模板的定义内不管填什么编译器都不报错？因为MSVC在分析模板中成员函数定义时没有做任何事情。至于为啥连"大王叫我来巡山"都能过得去，这是C++语法/语义分析的特殊性导致的。 C++是个非常复杂的语言，以至于它的编译器，不可能通过词法-语法-语义多趟分析清晰分割，因为它的语义将会直接干扰到语法：

```
void foo(){
    A<T> b;
}
```

在这段简短的代码中，就包含了两个歧义的可能，一是 `A` 是模板，于是 `A<T>` 是一个实例化的类型， `b` 是变量，另外一种是比较表达式（Comparison Expression）的组合， `((A < T) > b)` 。

甚至词法分析也会受到语义的干扰，C++11中才明确被修正的 `vector<vector<int>>` ，就因为 `>>` 被误解为右移或流操作符，而导致某些编译器上的错误。因此，在语义没有确定之前，连语法都没有分析的价值。

大约是基于如此考量，为了偷懒，MSVC将包括所有模板成员函数的语法/语义分析工作都挪到了第二个Phase，于是乎连带着语法分析都送进了第二个阶段。符合标准么？显然不符合。

但是这里值得一提的是，MSVC的做法和标准相比，虽然投机取巧，但并非有弊无利。我们来先说一说坏处。考虑以下例子：

```
// ----------- X.h ------------

template <typename T> struct X {
    // 实现代码
};

// ---------- X.cpp -----------
```

```
// ... 一些代码 ...
X<int> xi;
// ... 一些代码 ...
X<float> xf;
// ... 一些代码 ...
```

此时如果X中有一些与模板参数无关的错误，如果名称查找/语义分析在两个阶段完成，那么这些错误会很早、且唯一的被提示出来；但是如果一切都在实例化时处理，那么可能会导致不同的实例化过程提示同样的错误。而模板在运用过程中，往往会产生很多实例，此时便会大量报告同样的错误。

当然，MSVC并不会真的这么做。根据推测，最终他们是合并了相同的错误。因为即便对于模板参数相关的编译错误，也只能看到最后一次实例化的错误信息：

```
template <typename T> struct X {};

template <typename T> struct Y
{
    typedef X<T> ReboundType; // 类型定义1
    void foo()
    {
        X<T> instance0;
        X<T>::MemberType instance1;
        WTF instance2
    }
};

void poo(){
    Y<int>::foo();
    Y<float>::foo();
}
```

MSVC下和模板相关的错误只有一个：

```
error C2039: 'MemberType': is not a member of 'X<T>'
          with
          [
              T=float
          ]
```

然后是一些语法错误，比如 `MemberType` 不是一个合法的标识符之类的。这样甚至你会误以为 `int` 情况下模板的实例化是正确的。虽然在有了经验之后会发现这个问题挺荒唐的，但是仍然会让新手有困惑。

相比之下，更加遵守标准的Clang在错误提示上就要清晰许多：

```
error: unknown type name 'WTF'
    WTF instance2
    ^
error: expected ';' at end of declaration
    WTF instance2
                ^
                ;
error: no type named 'MemberType' in 'X<int>'
    typename X<T>::MemberType instance1;
    ~~~~~~~~~~~~~~~^~~~~~~~~~
    note: in instantiation of member function 'Y<int>::foo' requested here
        Y<int>::foo();
                ^
error: no type named 'MemberType' in 'X<float>'
    typename X<T>::MemberType instance1;
    ~~~~~~~~~~~~~~~^~~~~~~~~~
    note: in instantiation of member function 'Y<float>::foo' requested here
        Y<float>::foo();
                 ^
4 errors generated.
```

可以看到，Clang的提示和标准更加契合。它很好地区分了模板在定义和实例化时分别产生的错误。

另一个缺点也与之类似。因为没有足够的检查，如果你写的模板没有被实例化，那么很可能缺陷会一直存在于代码之中。特别是模板代码多在头文件。虽然不如接口那么重要，但也是属于被公开的部分，别人很可能会踩到坑上。缺陷一旦传播开修复起来就没那么容易了。

但是正如我前面所述，这个违背了标准的特性，并不是一无是处。首先，它可以完美的兼容标准。符合标准的、能够被正确编译的代码，一定能够被MSVC的方案所兼容。其次，它带来了一个非常有趣的特性，看下面这个例子：

```
struct A;
template <typename T> struct X {
    int v;
    void convertTo(A& a) {
       a.v = v; // 这里需要A的实现
    }
};

struct A { int v; };

void main() {
    X<int> x;
    x.foo(5);
}
```

这个例子在Clang中是错误的，因为：

```
error: variable has incomplete type 'A'
                        A a;
                          ^
    note: forward declaration of 'A'
     struct A;
            ^
1 error generated.
```

符合标准的写法需要将类模板的定义，和函数模板的定义分离开：

> TODO 此处例子不够恰当，并且描述有歧义。需要在未来版本中修订。

```
struct A;
template <typename T> struct X {
    int v;
    void convertTo(A& a);
};

struct A { int v; };

template <typename T> void X<T>::convertTo(A& a) {
   a.v = v;
}

void main() {
    X<int> x;
    x.foo(5);
}
```

但是其实我们知道， `foo` 要到实例化之后，才需要真正的做语义分析。在MSVC上，因为函数实现就是到模板实例化时才处理的，所以这个例子是完全正常工作的。因此在上面这个例子中，MSVC的实现要比标准更加易于写和维护，是不是有点写Java/C那种声明实现都在同一处的清爽感觉了呢！

扩展阅读： The Dreaded Two-Phase Name Lookup

3.3.3. “多余的” typename 关键字

到了这里，2.3.1 中提到的四个问题，还有三个没有解决：

```
template <typename T> struct X {};

template <typename T> struct Y
{
```

```
    typedef X<T> ReboundType;                                   // 这里为什么是正确的?
    typedef typename X<T>::MemberType MemberType2;      // 这里的typename是做什么的?
    typedef UnknownType MemberType3;                            // 这里为什么会出错?
};
```

我们运用我们2.3.2节中学习到的标准，来对Y内部做一下分析：

```
template <typename T> struct Y
{
    // X可以查找到原型;
    // X<T>是一个依赖性名称，模板定义阶段并不管X<T>是不是正确的。
    typedef X<T> ReboundType;

    // X可以查找到原型;
    // X<T>是一个依赖性名称，X<T>::MemberType也是一个依赖性名称;
    // 所以模板声明时也不会管X模板里面有没有MemberType这回事。
    typedef typename X<T>::MemberType MemberType2;

    // UnknownType 不是一个依赖性名称
    // 而且这个名字在当前作用域中不存在，所以直接报错。
    typedef UnknownType MemberType3;
};
```

下面，唯一的问题就是第二个： `typename` 是做什么的?

对于用户来说，这其实是一个语法噪音。也就是说，其实就算没有它，语法上也说得过去。事实上，某些情况下MSVC的确会在标准需要的时候，不用写 `typename` 。但是标准中还是规定了形如 `T::MemberType` 这样的 `qualified id` 在默认情况下不是一个类型，而是解释为 `T` 的一个成员变量 `MemberType` ，只有当 `typename` 修饰之后才能作为类型出现。

事实上，标准对 `typename` 的使用规定极为复杂，也算是整个模板中的难点之一。如果想了解所有的标准，需要阅读标准14.6节下2-7条，以及14.6.2.1第一条中对于 `current instantiation` 的解释。

简单来说，如果编译器能在出现的时候知道它是一个类型，那么就不需要 `typename` ，如果必须要到实例化的时候才能知道它是不是合法，那么定义的时候就把这个名称作为变量而不是类型。

我们用一行代码来说明这个问题：

```
a * b;
```

在没有模板的情况下，这个语句有两种可能的意思：如果 `a` 是一个类型，这就是定义了一个指针 `b` ，它拥有类型 `a*` ；如果 `a` 是一个对象或引用，这就是计算一个表达式 `a*b` ，虽然结果并没有保存下来。可是如果上面的 `a` 是模板参数的成员，会发生什么呢?

```
template <typename T> void meow()
{
    T::a * b; // 这是指针定义还是表达式语句?
}
```

编译器对模板进行语法检查的时候，必须要知道上面那一行到底是个什么——这当然可以推迟到实例化的时候进行（比如VC，这也是上面说过VC可以不加 `typename` 的原因），不过那是另一个故事了——显然在模板定义的时候，编译器并不能妄断。因此，C++标准规定，在没有 `typename` 约束的情况下认为这里 `T::a` 不是类型，因此 `T::a * b;` 会被当作表达式语句（例如乘法）；而为了告诉编译器这是一个指针的定义，我们必须在 `T::a` 之前加上 `typename` 关键字，告诉编译器 `T::a` 是一个类型，这样整个语句才能符合指针定义的语法。

在这里，我举几个例子帮助大家理解 `typename` 的用法，这几个例子已经足以涵盖日常使用 （预览） :

```
struct A;
template <typename T> struct B;
template <typename T> struct X {
    typedef X<T> TA; // 编译器当然知道 X<T> 是一个类型。
    typedef X    TB; // X 等价于 X<T> 的缩写
    typedef T    TC; // T 不是一个类型还玩毛

    // ！！！注意我要变形了！！！
    class Y {
        typedef X<T>     TD;          // X 的内部，既然外部高枕无忧，内部更不用说了
        typedef X<T>::Y  TE;          // 嗯，这里也没问题，编译器知道Y就是当前的类型，
```

```
                                    // 这里在VS2015上会有错，需要添加 typename，
                                    // Clang 上顺利通过。
        typedef typename X<T*>::Y TF; // 这个居然要加 typename!
                                    // 因为，X<T*>和X<T>不一样哦，
                                    // 它可能会在实例化的时候被别的偏特化给抢过去实现了。
    };

    typedef A TG;                   // 嗯，没问题，A在外面声明啦
    typedef B<T> TH;                // B<T>也是一个类型
    typedef typename B<T>::type TI; // 嗯，因为不知道B<T>::type的信息，
                                    // 所以需要typename
    typedef B<int>::type TJ;        // B<int> 不依赖模板参数，
                                    // 所以编译器直接就实例化（instantiate）了
                                    // 但是这个时候，B并没有被实现，所以就出错了
};
```

3.4. 本章小结

这一章是写作中最艰难的一章，中间停滞了将近一年。因为要说清楚C++模板中一些语法噪音和设计决议并不是一件轻松的事情。不过通过这一章的学习，我们知道了下面这几件事情：

1. **部分特化/偏特化** 和 **特化** 相当于是模板实例化过程中的 `if-then-else` 。这使得我们根据不同类型，选择不同实现的需求得以实现；
2. 在 2.3.3 一节我们插入了C++模板中最难理解的内容之一：名称查找。名称查找是语义分析的一个环节，模板内书写的 **变量声明**、typedef、**类型名称** 甚至 **类模板中成员函数的实现** 都要符合名称查找的规矩才不会出错；
3. C++编译器对语义的分析的原则是"大胆假设，小心求证"：在能求证的地方尽量求证 —— 比如两段式名称查找的第一阶段；无法检查的地方假设你是正确的 —— 比如 `typedef typename A<T>::MemberType X;` 在模板定义时因为 `T` 不明确不会轻易判定这个语句的死刑。

从下一章开始，我们将进入元编程环节。我们将使用大量的示例，一方面帮助巩固大家学到的模板知识，一方面也会引导大家使用函数式思维去解决常见的问题。

4. 深入理解特化与偏特化

4.1. 正确的理解偏特化

4.1.1. 偏特化与函数重载的比较

在前面的章节中，我们介绍了偏特化的形式、也介绍了简单的用例。因为偏特化和函数重载存在着形式上的相似性，因此初学者便会借用重载的概念，来理解偏特化的行为。只是，重载和偏特化尽管相似但仍有差异。

我们来先看一个函数重载的例子：

```
void doWork(int);
void doWork(float);
void doWork(int, int);

void f() {
    doWork(0);
    doWork(0.5f);
    doWork(0, 0);
}
```

在这个例子中，我们展现了函数重载可以在两种条件下工作：参数数量相同、类型不同；参数数量不同。

仿照重载的形式，我们通过特化机制，试图实现一个模板的"重载"：

```
template <typename T> struct DoWork;     // (0) 这是原型

template <> struct DoWork<int> {};       // (1) 这是 int 类型的"重载"
template <> struct DoWork<float> {};     // (2) 这是 float 类型的"重载"
template <> struct DoWork<int, int> {};  // (3) 这是 int, int 类型的"重载"

void f(){
```

```
    DoWork<int>      i;
    DoWork<float>    f;
    DoWork<int, int> ii;
}
```

这个例子在字面上"看起来"并没有什么问题，可惜编译器在编译的时候仍然提示出错了 goo.gl/zI42Zv ：

```
5 : error: too many template arguments for class template 'DoWork'
template <> struct DoWork<int, int> {}; // 这是 int, int 类型的"重载"
^ ~~~~
1 : note: template is declared here
template <typename T> struct DoWork {}; // 这是原型
~~~~~~~~~~~~~~~~~~~~ ^
```

从编译出错的失望中冷静一下，在仔细看看函数特化/偏特化和一般模板的不同之处：

```
template <typename T> class X      {};
template <typename T> class X <T*> {};
//                            ^^^^ 注意这里
```

对，就是这个 `<T*>` ，跟在X后面的"小尾巴"，我们称作实参列表，决定了第二条语句是第一条语句的跟班。所以，第二条语句，即"偏特化"，必须要符合原型X的基本形式：那就是只有一个模板参数。这也是为什么 `DoWork` 尝试以 `template <> struct DoWork<int, int>` 的形式偏特化的时候，编译器会提示模板实参数量过多。

另外一方面，在类模板的实例化阶段，它并不会直接去寻找 `template <> struct DoWork<int, int>` 这个小跟班，而是会先找到基本形式，`template <typename T> struct DoWork;` ，然后再去寻找相应的特化。

我们以 `DoWork<int> i;` 为例，尝试复原一下编译器完成整个模板匹配过程的场景，帮助大家理解。看以下示例代码：

```
template <typename T> struct DoWork;          // (0) 这是原型

template <> struct DoWork<int> {};            // (1) 这是 int 类型的特化
template <> struct DoWork<float> {};          // (2) 这是 float 类型的特化
template <typename U> struct DoWork<U*> {};   // (3) 这是指针类型的偏特化

DoWork<int>    i;  // (4)
DoWork<float*> pf; // (5)
```

首先，编译器分析(0), (1), (2)三句，得知(0)是模板的原型，(1)，(2)，(3)是模板(0)的特化或偏特化。我们假设有两个字典，第一个字典存储了模板原型，我们称之为 `TemplateDict` 。第二个字典 `TemplateSpecDict` ，存储了模板原型所对应的特化/偏特化形式。所以编译器在处理这几句时，可以视作

```
// 以下为伪代码
TemplateDict[DoWork<T>] = {
    DoWork<int>,
    DoWork<float>,
    DoWork<U*>
};
```

然后 (4) 试图以 `int` 实例化类模板 `DoWork` 。它会在 `TemplateDict` 中，找到 `DoWork` ，它有一个形式参数 `T` 接受类型，正好和我们实例化的要求相符合。并且此时 `T` 被推导为 `int` 。(5) 中的 `float*` 也是同理。

```
{   // 以下为 DoWork<int> 查找对应匹配的伪代码
    templateProtoInt = TemplateDict.find(DoWork, int);    // 查找模板原型，查找到(0)
    template = templatePrototype.match(int);              // 以 int 对应 int 匹配到 (1)
}

{   // 以下为DoWork<float*> 查找对应匹配的伪代码
    templateProtoIntPtr = TemplateDict.find(DoWork, float*) // 查找模板原型，查找到(0)
    template = templateProtoIntPtr.match(float*)            // 以 float* 对应 U* 匹配到 (3)，此时U为float
}
```

那么根据上面的步骤所展现的基本原理，我们随便来几个练习：

```
template <typename T, typename U> struct X           ;    // 0
                                                          // 原型有两个类型参数
                                                          // 所以下面的这些偏特化的实参列表
                                                          // 也需要两个类型参数对应
template <typename T>             struct X<T,  T  > {};    // 1
template <typename T>             struct X<T*, T  > {};    // 2
template <typename T>             struct X<T,  T* > {};    // 3
template <typename U>             struct X<U,  int> {};    // 4
template <typename U>             struct X<U*, int> {};    // 5
template <typename U, typename T> struct X<U*, T* > {};    // 6
template <typename U, typename T> struct X<U,  T* > {};    // 7

template <typename T>             struct X<unique_ptr<T>, shared_ptr<T>>; // 8

// 以下特化，分别对应哪个偏特化的实例?
// 此时偏特化中的T或U分别是什么类型?

X<float*,  int>      v0;
X<double*, int>      v1;
X<double,  double>   v2;
X<float*,  double*>  v3;
X<float*,  float*>   v4;
X<double,  float*>   v5;
X<int,     double*>  v6;
X<int*,    int>      v7;
X<double*, double>   v8;
```

在上面这段例子中，有几个值得注意之处。首先，偏特化时的模板形参，和原型的模板形参没有任何关系。和原型不同，它的顺序完全不影响模式匹配的顺序，它只是偏特化模式，如 `<U, int>` 中 `U` 的声明，真正的模式，是由 `<U, int>` 体现出来的。

这也是为什么在特化的时候，当所有类型都已经确定，我们就可以抛弃全部的模板参数，写出 `template <> struct X<int, float>` 这样的形式：因为所有列表中所有参数都确定了，就不需要额外的形式参数了。

其次，作为一个模式匹配，偏特化的实参列表中展现出来的"样子"，就是它能被匹配的原因。比如， `struct X<T, T>` 中，要求模板的两个参数必须是相同的类型。而 `struct X<T, T*>` ，则代表第二个模板类型参数必须是第一个模板类型参数的指针，比如 `X<float***, float****>` 就能匹配上。当然，除了简单的指针、 `const` 和 `volatile` 修饰符，其他的类模板也可以作为偏特化时的"模式"出现，例如示例8，它要求传入同一个类型的 `unique_ptr` 和 `shared_ptr` 。C++标准中指出下列模式都是可以被匹配的：

> N3337, 14.8.2.5/8
>
> 令 `T` 是模板类型实参或者类型列表（如 *int, float, double* 这样的， `TT` 是template-template实参（参见6.2节）， `i` 是模板的非类型参数（整数、指针等），则以下形式的形参都会参与匹配：
>
> `T , cv-list T , T* , template-name <T> , T& , T&&`
>
> `T [ integer-constant ]`
>
> `type (T) , T() , T(T)`
>
> `T type ::* , type T::* , T T::*`
>
> `T (type ::*)() , type (T::*)() , type (type ::*)(T) , type (T::*)(T) , T (type ::*)(T) , T (T::*)() , T (T::*)(T)`
>
> `type [i] , template-name <i> , TT<T> , TT<i> , TT<>`

对于某些实例化，偏特化的选择并不是唯一的。比如v4的参数是 `<float*, float*>` ，能够匹配的就有三条规则，1，6和7。很显然，6还是比7好一些，因为能多匹配一个指针。但是1和6，就很难说清楚谁更好了。一个说明了两者类型相同；另外一个则说明了两者都是指针。所以在这里，编译器也没办法决定使用那个，只好报出了编译器错误。

其他的示例可以先自己推测一下， 再去编译器上尝试一番： goo.gl/9UVzje 。

4.1.2. 不定长的模板参数

不过这个时候也许你还不死心。有没有一种办法能够让例子 `DoWork` 像重载一样，支持对长度不一的参数列表分别偏特化/特化呢?

答案当然是肯定的。

首先，首先我们要让模板实例化时的模板参数统一到相同形式上。逆向思维一下，虽然两个类型参数我们很难缩成一个参数，但是我们可以通过添加额外的参数，把一个扩展成两个呀。比如这样：

```
DoWork<int,   void> i;
DoWork<float, void> f;
DoWork<int,   int > ii;
```

这时，我们就能写出统一的模板原型：

```
template <typename T0, typename T1> struct DoWork;
```

继而偏特化/特化问题也解决了：

```
template <> struct DoWork<int,   void> {};  // (1) 这是 int 类型的特化
template <> struct DoWork<float, void> {};  // (2) 这是 float 类型的特化
template <> struct DoWork<int,    int> {};  // (3) 这是 int, int 类型的特化
```

显而易见这个解决方案并不那么完美。首先，不管是偏特化还是用户实例化模板的时候，都需要多撰写好几个 `void` ，而且最长的那个参数越长，需要写的就越多；其次，如果我们的 `DoWork` 在程序维护的过程中新加入了一个参数列表更长的实例，那么最悲惨的事情就会发生 —— 原型、每一个偏特化、每一个实例化都要追加上 `void` 以凑齐新出现的实例所需要的参数数量。

所幸模板参数也有一个和函数参数相同的特性：默认实参（Default Arguments）。只需要一个例子，你们就能看明白了 goo.gl/TtmcY9 ：

```
template <typename T0, typename T1 = void> struct DoWork;

template <typename T> struct DoWork<T> {};
template <>           struct DoWork<int> {};
template <>           struct DoWork<float> {};
template <>           struct DoWork<int, int> {};

DoWork<int> i;
DoWork<float> f;
DoWork<double> d;
DoWork<int, int> ii;
```

所有参数不足，即原型中参数 `T1` 没有指定的地方，都由T1自己的默认参数 `void` 补齐了。

但是这个方案仍然有些美中不足之处。

比如，尽管我们默认了所有无效的类型都以 `void` 结尾，所以正确的类型列表应该是类似于 `<int, float, char, void, void>` 这样的形态。但你阻止不了你的用户写出类似于 `<void, int, void, float, char, void, void>` 这样不符合约定的类型参数列表。

其次，假设这段代码中有一个函数，它的参数使用了和类模板相同的参数列表类型，如下面这段代码：

```
template <typename T0, typename T1 = void> struct X {
    static void call(T0 const& p0, T1 const& p1);        // 0
};

template <typename T0> struct X<T0> {
    static void call(T0 const& p0);                      // 1
};

void foo(){
    X<int>::call(5);               // 调用函数 1
    X<int, float>::call(5, 0.5f);  // 调用函数 0
}
```

那么，每加一个参数就要多写一个偏特化的形式，甚至还要重复编写一些可以共享的实现。

不过不管怎么说，以长参数加默认参数的方式支持变长参数是可行的做法，这也是C++98/03时代的唯一选择。

例如， `Boost.Tuple` 就使用了这个方法，支持了变长的Tuple：

```
// Tuple 的声明，来自 boost
struct null_type;

template <
  class T0 = null_type, class T1 = null_type, class T2 = null_type,
  class T3 = null_type, class T4 = null_type, class T5 = null_type,
  class T6 = null_type, class T7 = null_type, class T8 = null_type,
  class T9 = null_type>
class tuple;

// Tuple的一些用例
tuple<int> a;
tuple<double&, const double&, const double, double*, const double*> b;
tuple<A, int(*)(char, int), B(A::*)(C&), C> c;
tuple<std::string, std::pair<A, B> > d;
tuple<A*, tuple<const A*, const B&, C>, bool, void*> e;
```

此外，Boost.MPL也使用了这个手法将 `boost::mpl::vector` 映射到 `boost::mpl::vector _n_` 上。但是我们也看到了，这个方案的缺陷很明显：代码臃肿和潜在的正确性问题。此外，过度使用模板偏特化、大量冗余的类型参数也给编译器带来了沉重的负担。

为了缓解这些问题，在C++11中，引入了变参模板（Variadic Template）。我们来看看支持了变参模板的C++11是如何实现tuple的：

```
template <typename... Ts> class tuple;
```

是不是一下子简洁了很多！这里的 `typename... Ts` 相当于一个声明，是说 `Ts` 不是一个类型，而是一个不定常的类型列表。同C语言的不定长参数一样，它通常只能放在参数列表的最后。看下面的例子：

```
template <typename... Ts, typename U> class X {};             // (1) error!
template <typename... Ts>             class Y {};             // (2)
template <typename... Ts, typename U> class Y<U, Ts...> {};   // (3)
template <typename... Ts, typename U> class Y<Ts..., U> {};   // (4) error!
```

为什么第(1)条语句会出错呢？(1)是模板原型，模板实例化时，要以它为基础和实例化时的类型实参相匹配。因为C++的模板是自左向右匹配的，所以不定长参数只能结尾。其他形式，无论写作 `Ts, U` ，或者是 `Ts, V, Us,` ，或者是 `V, Ts, Us` 都是不可取的。(4) 也存在同样的问题。

但是，为什么(3)中， 模板参数和(1)相同，都是 `typename... Ts, typename U` ，但是编译器却并没有报错呢？

答案在这一节的早些时候。(3)和(1)不同，它并不是模板的原型，它只是 `Y` 的一个偏特化。回顾我们在之前所提到的，偏特化时，模板参数列表并不代表匹配顺序，它们只是为偏特化的模式提供的声明，也就是说，它们的匹配顺序，只是按照 `<U, Ts...>` 来，而之前的参数只是告诉你 `Ts` 是一个类型列表，而 `U` 是一个类型，排名不分先后。

在这里，我们只提到了变长模板参数的声明，如何使用我们将在第四章讲述。

4.1.3. 模板的默认实参

在上一节中，我们介绍了模板对默认实参的支持。当时我们的例子很简单，默认模板实参是一个确定的类型 `void` 或者自定义的 `null_type` ：

```
template <
    typename T0, typename T1 = void, typename T2 = void
> class Tuple;
```

实际上，模板的默认参数不仅仅可以是一个确定的类型，它还能是以其他类型为参数的一个类型表达式。 考虑下面的例子：我们要执行两个同类型变量的除法，它对浮点、整数和其他类型分别采取不同的措施。 对于浮点，执行内置除法；对于整数，要处理除零保护，防止引发异常；对于其他类型，执行一个叫做 `CustomeDiv` 的函数。

第一步，我们先把浮点正确的写出来：

```
include <type_traits>

template <typename T> T CustomDiv(T lhs, T rhs) {
    // Custom Div的实现
}

template <typename T, bool IsFloat = std::is_floating_point<T>::value> struct SafeDivide {
```

```
    static T Do(T lhs, T rhs) {
        return CustomDiv(lhs, rhs);
    }
};

template <typename T> struct SafeDivide<T, true>{    // 偏特化A
    static T Do(T lhs, T rhs){
        return lhs/rhs;
    }
};

template <typename T> struct SafeDivide<T, false>{   // 偏特化B
    static T Do(T lhs, T rhs){
        return lhs;
    }
};

void foo(){
    SafeDivide<float>::Do(1.0f, 2.0f);  // 调用偏特化A
    SafeDivide<int>::Do(1, 2);          // 调用偏特化B
}
```

在实例化的时候，尽管我们只为 `SafeDivide` 指定了参数 `T` ，但是它的另一个参数 `IsFloat` 在缺省的情况下，可以根据 `T` ，求出表达式 `std::is_floating_point<T>::value` 的值作为实参的值，带入到 `SafeDivide` 的匹配中。

嗯，这个时候我们要再把整型和其他类型纳入进来，无外乎就是加这么一个参数 goo.gl/0Lqywt ：

```
include <complex>
include <type_traits>

template <typename T> T CustomDiv(T lhs, T rhs) {
    T v;
    // Custom Div的实现
    return v;
}

template <
    typename T,
    bool IsFloat = std::is_floating_point<T>::value,
    bool IsIntegral = std::is_integral<T>::value
> struct SafeDivide {
    static T Do(T lhs, T rhs) {
        return CustomDiv(lhs, rhs);
    }
};

template <typename T> struct SafeDivide<T, true, false>{    // 偏特化A
    static T Do(T lhs, T rhs){
        return lhs/rhs;
    }
};

template <typename T> struct SafeDivide<T, false, true>{   // 偏特化B
    static T Do(T lhs, T rhs){
        return rhs == 0 ? 0 : lhs/rhs;
    }
};

void foo(){
    SafeDivide<float>::Do(1.0f, 2.0f);                          // 调用偏特化A
    SafeDivide<int>::Do(1, 2);                                  // 调用偏特化B
    SafeDivide<std::complex<float>>::Do({1.f, 2.f}, {1.f, -2.f}); // 调用一般形式
}
```

当然，这时也许你会注意到， `is_integral` ， `is_floating_point` 和其他类类型三者是互斥的，那能不能只使用一个条件量来进行分派呢？答案当然是可以的： goo.gl/jYp5J2 ：

```
include <complex>
include <type_traits>
```

```cpp
template <typename T> T CustomDiv(T lhs, T rhs) {
    T v;
    // Custom Div的实现
    return v;
}

template <typename T, typename Enabled = std::true_type> struct SafeDivide {
    static T Do(T lhs, T rhs) {
        return CustomDiv(lhs, rhs);
    }
};

template <typename T> struct SafeDivide<
    T, typename std::is_floating_point<T>::type>{    // 偏特化A
    static T Do(T lhs, T rhs){
        return lhs/rhs;
    }
};

template <typename T> struct SafeDivide<
    T, typename std::is_integral<T>::type>{          // 偏特化B
    static T Do(T lhs, T rhs){
        return rhs == 0 ? 0 : lhs/rhs;
    }
};

void foo(){
    SafeDivide<float>::Do(1.0f, 2.0f);  // 调用偏特化A
    SafeDivide<int>::Do(1, 2);          // 调用偏特化B
    SafeDivide<std::complex<float>>::Do({1.f, 2.f}, {1.f, -2.f});
}
```

我们借助这个例子，帮助大家理解一下这个结构是怎么工作的：

1. 对 `SafeDivide<int>`

- 通过匹配类模板的泛化形式，计算默认实参，可以知道我们要匹配的模板实参是 `SafeDivide<int, true_type>`
- 计算两个偏特化的形式的匹配：A得到 `<int, false_type>` ,和B得到 `<int, true_type>`
- 最后偏特化B的匹配结果和模板实参一致，使用它。

2. 针对 `SafeDivide<complex<float>>`

- 通过匹配类模板的泛化形式，可以知道我们要匹配的模板实参是 `SafeDivide<complex<float>, true_type>`
- 计算两个偏特化形式的匹配：A和B均得到 `SafeDivide<complex<float>, false_type>`
- A和B都与模板实参无法匹配，所以使用原型，调用 `CustomDiv`

4.2. 后悔药：SFINAE

考虑下面这个函数模板：

```cpp
template <typename T, typename U>
void foo(T t, typename U::type u) {
    // ...
}
```

到本节为止，我们所有的例子都保证了一旦咱们敲定了模板参数中 `T` 和 `U`，函数参变量 `t` 和 `u` 的类型都是成立的，比如下面这样：

```cpp
struct X {
    typedef float type;
};

template <typename T, typename U>
void foo(T t, typename U::type u) {
    // ...
}
```

```
void callFoo() {
    foo<int, X>(5, 5.0); // T == int, typename U::type == X::type == float
}
```

那么这里有一个可能都不算是问题的问题 —— 对于下面的代码，你认为它会提示怎么样的错误：

```
struct X {
    typedef float type;
};

struct Y {
    typedef float type2;
};

template <typename T, typename U>
void foo(T t, typename U::type u) {
  // ...
}

void callFoo() {
    foo<int, X>(5, 5.0); // T == int, typename U::type == X::type == float
    foo<int, Y>(5, 5.0); // ???
}
```

这个时候你也许会说：啊，这个简单， `Y` 没有 `type` 这个成员自然会出错啦！嗯，这个时候咱们来看看Clang给出的结果：

```
error: no matching function for call to 'foo'
   foo<int, Y>(5, 5.0); // ???
   ^~~~~~~~~~
   note: candidate template ignored: substitution failure [with T = int, U = Y]: no type named 'type' in 'Y'
       void foo(T t, typename U::type u) {
```

完整翻译过来就是，直接的出错原因是没有匹配的 `foo` 函数，间接原因是尝试用 `[T = int, U = y]` 做类型替换的时候失败了，所以这个函数模板就被忽略了。等等，不是出错，而是被忽略了？那么也就是说，只要有别的能匹配的类型兜着，编译器就无视这里的失败了？

银河火箭队的阿喵说，就是这样。不信邪的朋友可以试试下面的代码：

```
struct X {
    typedef float type;
};

struct Y {
    typedef float type2;
};

template <typename T, typename U>
void foo(T t, typename U::type u) {
    // ...
}

template <typename T, typename U>
void foo(T t, typename U::type2 u) {
  // ...
}
void callFoo() {
    foo<int, X>(5, 5.0); // T == int, typename U::type == X::type == float
    foo<int, Y>( 1, 1.0 ); // ???
}
```

这下相信编译器真的是不关心替换失败了吧。我们管这种只要有正确的候选，就无视替换失败的做法为SFINAE。

我们不用纠结这个词的发音，它来自于 Substitution failure is not an error 的首字母缩写。这一句之乎者也般难懂的话，由之乎者 —— 啊，不，Substitution，Failure和Error三个词构成。

我们从最简单的词"Error"开始理解。Error就是一般意义上的编译错误。一旦出现编译错误，大家都知道，编译器就会中止编译，并且停止接下来的代码生成和链接等后续活动。

其次，我们再说“Failure”。很多时候光看字面意思，很多人会把 Failure 和 Error 等同起来。但是实际上Failure很多场合下只是一个中性词。比如我们看下面这个虚构的例子就知道这两者的区别了。

假设我们有一个语法分析器，其中某一个规则需要匹配一个token，它可以是标识符，字面量或者是字符串，那么我们会有下面的代码：

```
switch(token)
{
case IDENTIFIER:
    // do something
    break;
case LITERAL_NUMBER:
    // do something
    break;
case LITERAL_STRING:
    // do something
    break;
default:
    throw WrongToken(token);
}
```

假如我们当前的token是 `LITERAL_STRING` 的时候，那么第一步它在匹配 `IDENTIFIER` 时，我们可以认为它失败（failure）了，但是它在第三步就会匹配上，所以它并不是一个错误。

但是如果这个token既不是标识符、也不是数字字面量、也不是字符串字面量，而且我们的语法规定除了这三类值以外其他统统都是非法的时，我们才认为它是一个error。

大家所熟知的函数重载也是如此。比如说下面这个例子：

```
struct A {};
struct B: public A {};
struct C {};

void foo(A const&) {}
void foo(B const&) {}

void callFoo() {
    foo( A() );
    foo( B() );
    foo( C() );
}
```

那么 `foo( A() )` 虽然匹配 `foo(B const&)` 会失败，但是它起码能匹配 `foo(A const&)`，所以它是正确的； `foo( B() )` 能同时匹配两个函数原型，但是 `foo(B const&)` 要更好一些，因此它选择了这个原型。而 `foo( C() );` 因为两个函数都匹配失败（Failure）了，所以它找不到相应的原型，这时才会报出一个编译器错误（Error）。

所以到这里我们就明白了，在很多情况下，Failure is not an error。编译器在遇到Failure的时候，往往还需要尝试其他的可能性。

好，现在我们把最后一个词，Substitution，加入到我们的字典中。现在这句话的意思就是说，我们要把 Failure is not an error 的概念，推广到Substitution阶段。

所谓substitution，就是将函数模板中的形参，替换成实参的过程。概念很简洁但是实现却颇多细节，所以C++标准中对这一概念的解释比较拗口。它分别指出了以下几点：

- 什么时候函数模板会发生实参 替代（Substitute） 形参的行为；
- 什么样的行为被称作 Substitution；
- 什么样的行为不可以被称作 Substitution Failure —— 他们叫SFINAE error。

我们在此不再详述，有兴趣的同学可以参照 这里 ，这是标准的一个精炼版本。这里我们简单的解释一下。

考虑我们有这么个函数签名：

```
template <
    typename T0,
    // 一大坨其他模板参数
    typename U = /* 和前面T有关的一大坨 */
```

```
>
RType /* 和模板参数有关的一大坨 */
functionName (
    PType0 /* PType0 是和模板参数有关的一大坨 */,
    PType1 /* PType1 是和模板参数有关的一大坨 */,
    // ... 其他参数
) {
    // 实现，和模板参数有关的一大坨
}
```

那么，在这个函数模板被实例化的时候，所有函数签名上的“和模板参数有关的一大坨”被推导出具体类型的过程，就是替换。一个更具体的例子来解释上面的“一大坨”：

```
template <
    typename T,
    typename U = typename vector<T>::iterator // 1
>
typename vector<T>::value_type  // 1
foo(
    T*, // 1
    T&, // 1
    typename T::internal_type, // 1
    typename add_reference<T>::type, // 1
    int // 这里都不需要 substitution
)
{
  // 根据定义，substitution只发生在函数签名上。
  // 故而整个函数实现部分都不会存在 substitution。
  // 这是一个重点需要记住。
}
```

所有标记为 `1` 的部分，都是需要替换的部分，而它们在替换过程中的失败（failure），就称之为替换失败（substitution failure）。

下面的代码是提供了一些替换成功和替换失败的示例：

```
struct X {
    typedef int type;
};

struct Y {
    typedef int type2;
};

template <typename T> void foo(typename T::type);    // Foo0
template <typename T> void foo(typename T::type2);   // Foo1
template <typename T> void foo(T);                   // Foo2

void callFoo() {
    foo<X>(5);    // Foo0: Succeed, Foo1: Failed,  Foo2: Failed
    foo<Y>(10);   // Foo0: Failed,  Foo1: Succeed, Foo2: Failed
    foo<int>(15); // Foo0: Failed,  Foo1: Failed,  Foo2: Succeed
}
```

在这个例子中，当我们指定 `foo<Y>` 的时候，substitution就开始工作了，而且会同时工作在三个不同的 `foo` 签名上。如果我们仅仅因为 `Y` 没有 `type`，匹配 `Foo0` 失败了，就宣布代码有错，中止编译，那显然是武断的。因为 `Foo1` 是可以被正确替换的，我们也希望 `Foo1` 成为 `foo<Y>` 的原型。

std/boost库中的 `enable_if` 是 SFINAE 最直接也是最主要的应用。所以我们通过下面 `enable_if` 的例子，来深入理解一下 SFINAE 在模板编程中的作用。

假设我们有两个不同类型的计数器（counter），一种是普通的整数类型，另外一种是一个复杂对象，它从接口 `ICounter` 继承，这个接口有一个成员叫做increase实现计数功能。现在，我们想把这两种类型的counter封装一个统一的调用：inc_counter。那么，我们直觉会简单粗暴的写出下面的代码：

```
struct ICounter {
    virtual void increase() = 0;
    virtual ~ICounter() {}
```

```
};

struct Counter: public ICounter {
    void increase() override {
        // Implements
    }
};

template <typename T>
void inc_counter(T& counterObj) {
    counterObj.increase();
}

template <typename T>
void inc_counter(T& intTypeCounter){
    ++intTypeCounter;
}

void doSomething() {
    Counter cntObj;
    uint32_t cntUI32;

    // blah blah blah
    inc_counter(cntObj);
    inc_counter(cntUI32);
}
```

我们非常希望它展现出预期的行为。因为其实我们是知道对于任何一个调用，两个 `inc_counter` 只有一个是能够编译正确的。“有且唯一”，我们理应当期望编译器能够挑出那个唯一来。

可惜编译器做不到这一点。首先，它就告诉我们，这两个签名

```
template <typename T> void inc_counter(T& counterObj);
template <typename T> void inc_counter(T& intTypeCounter);
```

其实是一模一样的。我们遇到了 `redefinition` 。

我们看看 `enable_if` 是怎么解决这个问题的。我们通过 `enable_if` 这个 `T` 对于不同的实例做个限定：

```
template <typename T> void inc_counter(
    T& counterObj,
    typename std::enable_if<
        std::is_base_of<ICounter, T>::value
    >::type* = nullptr );

template <typename T> void inc_counter(
    T& counterInt,
    typename std::enable_if<
        std::is_integral<T>::value
    >::type* = nullptr );
```

然后我们解释一下，这个 `enable_if` 是怎么工作的，语法为什么这么丑：

首先，替换（substitution）只有在推断函数类型的时候，才会起作用。推断函数类型需要参数的类型，所以， `typename std::enable_if<std::is_integral<T>::value>::type` 这么一长串代码，就是为了让 `enable_if` 参与到函数类型中；

其次， `is_integral<T>::value` 返回一个布尔类型的编译器常数，告诉我们它是或者不是一个 `integral type` ， `enable_if<C>` 的作用就是，如果这个 `C` 值为 `True` ，那么 `enable_if<C>::type` 就会被推断成一个 `void` 或者是别的什么类型，让整个函数匹配后的类型变成 `void inc_counter<int>(int & counterInt, void* dummy = nullptr);` 如果这个值为 `False` ，那么 `enable_if<false>` 这个特化形式中，压根就没有这个 `::type` ，于是替换就失败了。和我们之前的例子中一样，这个函数原型就不会被产生出来。

所以我们能保证，无论对于 `int` 还是 `counter` 类型的实例，我们都只有一个函数原型通过了substitution —— 这样就保证了它的“有且唯一”，编译器也不会因为你某个替换失败而无视成功的那个实例。

这个例子说到了这里，熟悉C++的你，一定会站出来说我们只要把第一个签名改成：

```
void inc_counter(ICounter& counterObj);
```

就能完美解决这个问题了，根本不需要这么复杂的编译器机制。

嗯，你说的没错，在这里这个特性一点都没用。

这也提醒我们，当你觉得需要写 `enable_if` 的时候，首先要考虑到以下可能的替代方案：

- 重载（适用于函数模板）
- 偏特化（适用于类模板）
- 虚函数

但是问题到了这里并没有结束。因为 `increase` 毕竟是个虚函数。假如 `Counter` 需要调用的地方实在是太多了，这个时候我们会非常期望 `increase` 不再是个虚函数以提高性能。此时我们会调整继承层级：

```
struct ICounter {};
struct Counter: public ICounter {
    void increase() {
        // impl
    }
};
```

那么原有的 `void inc_counter(ICounter& counterObj)` 就无法再执行下去了。这个时候你可能会考虑一些变通的办法：

```
template <typename T>
void inc_counter(ICounter& c) {};

template <typename T>
void inc_counter(T& c) { ++c; };

void doSomething() {
    Counter cntObj;
    uint32_t cntUI32;

    // blah blah blah
    inc_counter(cntObj); // 1
    inc_counter(static_cast<ICounter&>(cntObj)); // 2
    inc_counter(cntUI32); // 3
}
```

对于调用 `1`，因为 `cntObj` 到 `ICounter` 是需要类型转换的，所以比 `void inc_counter(T&) [T = Counter]` 要更差一些。然后它会直接实例化后者，结果实现变成了 `++cntObj`，BOOM！

那么我们做 `2` 试试看？嗯，工作的很好。但是等等，我们的初衷是什么来着？不就是让 `inc_counter` 对不同的计数器类型透明吗？这不是又一夜回到解放前了？

所以这个时候，就能看到 `enable_if` 是如何通过 SFINAE 发挥威力的了：

```
include <type_traits>
include <utility>
include <cstdint>

struct ICounter {};
struct Counter: public ICounter {
    void increase() {
        // impl
    }
};

template <typename T> void inc_counter(
    T& counterObj,
    typename std::enable_if<
        std::is_base_of<ICounter, T>::value
    >::type* = nullptr ){
    counterObj.increase();
}
```

```
template <typename T> void inc_counter(
    T& counterInt,
    typename std::enable_if<
        std::is_integral<T>::value
    >::type* = nullptr ){
    ++counterInt;
}

void doSomething() {
    Counter cntObj;
    uint32_t cntUI32;

    // blah blah blah
    inc_counter(cntObj); // OK!
    inc_counter(cntUI32); // OK!
}
```

这个代码是不是看起来有点脏脏的。眼尖的你定睛一瞧，咦， `ICounter` 不是已经空了吗，为什么我们还要用它作为基类呢?

这是个好问题。在本例中，我们用它来区分一个 `counter` 是不是继承自 `ICounter` 。最终目的，是希望知道 `counter` 有没有 `increase` 这个函数。

所以 `ICounter` 只是相当于一个标签。而于情于理这个标签都是个累赘。但是在C++11之前，我们并没有办法去写类似于:

```
template <typename T> void foo(T& c, decltype(c.increase())* = nullptr);
```

这样的函数签名，因为假如 `T` 是 `int` ，那么 `c.increase()` 这个函数调用就不存在。但它又不属于Type Failure，而是一个Expression Failure，在C++11之前它会直接导致编译器出错，这并不是我们所期望的。所以我们才退而求其次，用一个类似于标签的形式来提供我们所需要的类型信息。以后的章节，后面我们会说到，这种和类型有关的信息我们可以称之为 `type traits` 。

到了C++11，它正式提供了 Expression SFINAE，这时我们就能抛开 `ICounter` 这个无用的Tag，直接写出我们要写的东西:

```
struct Counter {
    void increase() {
        // Implements
    }
};

template <typename T>
void inc_counter(T& intTypeCounter, std::decay_t<decltype(++intTypeCounter)>* = nullptr) {
    ++intTypeCounter;
}

template <typename T>
void inc_counter(T& counterObj, std::decay_t<decltype(counterObj.increase())>* = nullptr) {
    counterObj.increase();
}

void doSomething() {
    Counter cntObj;
    uint32_t cntUI32;

    // blah blah blah
    inc_counter(cntObj);
    inc_counter(cntUI32);
}
```

此外，还有一种情况只能使用 SFINAE，而无法使用包括继承、重载在内的任何方法，这就是Universal Reference。比如，

```
// 这里的a是个通用引用，可以准确的处理左右值引用的问题。
template <typename ArgT> void foo(ArgT&& a);
```

假如我们要限定ArgT只能是 float 的衍生类型，那么写成下面这个样子是不对的，它实际上只能接受 float 的右值引用。

```
void foo(float&& a);
```

此时的唯一选择，就是使用Universal Reference，并增加 `enable_if` 限定类型，如下面这样：

```
template <typename ArgT>
void foo(
    ArgT&& a,
    typename std::enabled_if<
        std::is_same<std::decay_t<ArgT>, float>::value
    >::type* = nullptr
);
```

从上面这些例子可以看到，SFINAE最主要的作用，是保证编译器在泛型函数、偏特化、及一般重载函数中遴选函数原型的候选列表时不被打断。除此之外，它还有一个很重要的元编程作用就是实现部分的编译期自省和反射。

虽然它写起来并不直观，但是对于既没有编译器自省、也没有Concept的C++11来说，已经是最好的选择了。

4.3. Concept “概念”：对模板参数约束的直接描述

4.3.1. “概念” 解决了什么问题

从上一节可以看出，我们兜兜转转了那么久，就是为了解决两个问题：

1. 在模板进行特化的时候，盘算一下并告诉编译器这里能不能特化；
2. 在函数决议面临多个候选的时候，如果有且仅有其中一个原型能够被函数决议接纳，那就决定是你了！

如果语言能允许用户直接描述需求并传达给编译器，不就不用这么麻烦了么。其实在很多现代语言中，都有类似的语言要素存在，比如C的约束（constraint on type parameters）：

```
public class Employee {
  // ...
}

public class GenericList<T> where T : Employee {
  // ...
}
```

上例就非常清晰的呈现了我们对 `GenericList` 中 `T` 的要求是：它得是一个 `Employee` 或 `Employee` 的子类。

这种“清晰的”类型约束，在C++中称作概念（Concept）。最早有迹可循的概念相关工作应当从2003年后就开始了。2006年Bjarne在POPL 06上的一篇报告“Specifying C++ concepts”算是“近代”Concept工作的首次公开亮相。委员会为Concept筹划数年，在2008年提出了第一版Concepts提案，试图进入C++0x的标准中。这也是Concept第一次在C++社群当中被广泛“炒作”。不过2009年的会议，让“近代”Concept在N2617草案戛然而止。

2013年之后，Concept改头换面为Concept Lite提案（N3701)卷土重来，历经多方博弈和多轮演化，最终形成了我们在C++20里看到的Concept。有关于Concept的方法论和比较，B.S. 在白皮书中有过比较详细的交代。

总之，在concept进入标准之后，模板特化的类型约束写起来就方便与直接多了。而且这些约束之间还可以像表达式一样复用和组合。虽然因为C++类型系统自身的琐碎导致基础库中的concept仍然相当的冗长，但是比起之前起码具备了可用性。

比如我们拿上一节中最后一个例子作为对比：

```
// SFINAE
template <typename ArgT>
void foo(
    ArgT&& a,
    typename std::enabled_if<
        std::is_same<std::decay_t<ArgT>, float>::value
    >::type* = nullptr
);
// Concept
template <typename ArgT>
  requires std::same_as<std::remove_cvref<T>, float>
void foo(ArgT&& a)  {
}
```

可以看到，concept之后的表达式消除了语法噪音，显得更为简洁一些。而对于之前++的例子，concept下则更为扼要：

```
template <typename T> concept Incrementable = requires (T t) { ++t; }
template <Incrementable T>
void inc_counter(T& intTypeCounter) {
    ++intTypeCounter;
}
```

直接告诉编译器，我们对T的要求是你得有 `++` 。

当然有人会问，那能不能直接写成以下形式，不是更简单吗？

```
template <typename T> requires (T t) { ++t; }
void inc_counter(T& cnt);
```

答案是：不能。 因为 `requires` 作为关键字/保留字是存在二义性的。当它用于函数模板或者类模板的声明时，它是一个constraint，后面需要跟着concept表达式；而用于concept中，则是一个required expression，用于concept的求解。既然constraint后面跟着一个concept表达式，而requires也可以用来定义一个concept expression，那么一个风骚的想法形成了：我能不能用 `requires (requires (T t) {++t;})` 来约束模板参数呢？

当然是可以的！C++就是这么的简（有）单（病）！

```
template <typename T> requires (requires (T t) { ++t; })
void inc_counter(T& cnt);
```

总而言之，除了这些烦人的问题，“概念”的出现，使得模板的出错提示也清爽了些许 —— 虽然大佬们都在鼓吹concept让模板出错多么好调试，但是实际上模板出错，有一半是来源自类型系统本质上的复杂性，概念并不能解决这一问题。

比如这里使用SFINAE的提示：

```
<source>:23:5: error: no matching function for call to 'Inc'
    Inc(y);
    ^~~
<source>:5:6: note: candidate template ignored: substitution failure [with T = X]: cannot increment value of type 'X'
void Inc(T& v, std::decay_t<decltype(++v)>* = nullptr)
     ^                               ~~
```

而这里是使用了concept的提示。

```
<source>:25:5: error: no matching function for call to 'Inc_Concept'
    Inc_Concept(y);
    ^~~~~~~~~~
<source>:13:6: note: candidate template ignored: constraints not satisfied [with T = X]
void Inc_Concept(T& v)
     ^
<source>:12:11: note: because 'X' does not satisfy 'Incrementable'
template <Incrementable T>
          ^
<source>:10:41: note: because '++t' would be invalid: cannot increment value of type 'X'
concept Incrementable = requires(T t) { ++t; };
```

虽然在这个例子中，通过 *Concept* 获得出错提示看起来要比使用 *SFINAE* 所获得的错误描述要更长一点，但是对于更加复杂类型来说，则会友善许多。以后会找个例子给大家陈述。

4.3.2. "概念"入门

5. 未完成章节

```
# 6. 元编程下的数据结构与算法
## 6.1. 表达式与数值计算
## 6.2. 获得类型的属性—类型萃取（Type Traits）
```