现代C++模板教程

国内众多 C++ 教程古老且过时，学校教学则更是过分。我们需要新式的，教学符合时代的知识、代码风格、思维的课程！

本教程创新性的采用学习 + 提交作业的方式，您需要视频学习后提交作业，而我们会进行批改和评论。

本教程假设读者的最低水平为：`C + class + STL`。

请确保您的编译器至少支持 C++20，优先使用 gcc13，clang16，msvc v19.latest。所有代码均测试三大编译器。

某些仓库的相对链接可能无效，这是正常现象，**可以访问远程仓库，Github、Gitee。**

函数模板

本节将介绍函数模板

初识函数模板

函数模板[1]不是函数，只有实例化[2]函数模板，编译器才能生成实际的函数定义。不过在很多时候，它看起来就像普通函数一样。

定义模板

下面是一个函数模板，返回两个对象中较大的那个：

```
template<typename T>
T max(T a,T b){
    return a > b ? a : b;
}
```

这应该很简单，即使我们还没有开始讲述函数模板的语法。

如果要声明一个函数模板，我们通常要使用：

```
template< 形参列表 > 函数声明
```

我们前面示例中的形参列表是 `typename T`，关键字 typename 顾名思义，引入了一个类型形参。

类型形参是 T，也可以使用其他标识符作为类型形参名（T 或 Ty 等，是约定的惯例），你也可以在需要的时候自定义一些有明确意义的名字。在调用函数模板 `max` 时，根据传入参数，编译器可以推导出类型形参的类型，实例化函数模板。我们需要传入支持函数模板操作的类型，如 int 或 重载了 `>` 运算符的类。注意 `max` 的 `return` 这意味着我们的模板形参 T 还需要是可复制/移动的，以便返回。

> C++17 之前，类型 T 必须是可复制或移动才能传递参数。C++17 以后，即使复制构造函数和移动构造函数都无效，因为 C++17 强制的复制消除，也可以传递临时纯右值。

因为一些历史原因，我们也可以使用 class 关键字来定义模板类型形参。所以先前的模板 `max` 可以**等价于**:

```
template<class T>
T max(T a,T b){
    return a > b ? a : b;
}
```

但是与类声明不同，在声明模板类型形参时，不能使用 struct。

使用模板

下面展示了如何使用函数模板 `max()`

```
#include <iostream>

template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

struct Test{
    int v_{};
    Test() = default;
    Test(int v) :v_(v) {}
    bool operator>(const Test& t) const{
        return this->v_ > t.v_;
    }
};

int main(){
    int a{ 1 };
    int b{ 2 };
    std::cout << "max(a, b) : " << ::max(a, b) << '\n';

    Test t1{ 10 };
    Test t2{ 20 };
    std::cout << "max(t1, t2) : " << ::max(t1, t2).v_ << '\n';

}
```

> 看起来的确和调用普通函数没区别，那么这样调用和普通函数相比，编译器会做什么呢?

编译器会**实例化两个函数**，也就是生成了一个参数为 int 的 max 函数，一个参数为 Test 的函数。

```
int max(int a, int b)
{
  return a > b ? a : b;
}

Test max(Test a, Test b)
{
  return a > b ? a : b;
}
```

我们可以使用 cppinsights 验证我们的想法。

用一句非常不严谨的话来说：

- **模板，只有你“用”了它，才会生成实际的代码。**

> 这里的“用”，其实就是指代会隐式实例化，生成代码。

并且需要注意，同一个函数模板生成的不同类型的函数，彼此之间没有任何关系。

除了让编译器自己去推导函数模板的形参类型以外，我们还可以自己显式的指明：

```
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

int main(){
    int a{ 1 };
    int b{ 2 };
    max(a, b);          // 函数模板 max 被推导为 max<int>

    max<double>(a, b);  // 传递模板类型实参，函数模板 max 为 max<double>
}
```

模板参数推导

当使用函数模板（如 max()）时，模板参数可以由传入的参数推导。如果类型 T 传递两个 int 型参数，那编译器就会认为 T 是 int 型。

然而，T 可能只是类型的“一部分”。若声明 max() 使用 `const&`：

```
template<typename T>
T max(const T& a, const T& b) {
    return a > b ? a : b;
}
```

如果我们 `max(1, 2)` 或者说 `max<int>(x,x)`，T 当然会是 int，但是函数形参类型会是 `const int&`。

不过我们需要注意，有不少情况是没有办法进行推导的：

```
// 省略 max
using namespace std::string_literals;

int main(){
    max(1, 1.2);            // Error 无法确定你的 T 到底是要 int 还是 double
    max("luse"s, "乐");     // Error 无法确定你的 T 到底是要 std::string 还是 const
char[N]
}
```

那么我们如何处理这种错误呢？可以使用前面提到的**显式指定函数模板的（T）类型。**

```
max<double>(1, 1.2);
max<std::string>("luse"s, "乐");
```

又或者说**显式类型转换**：

```
max(static_cast<double>(1), 1.2);
```

但是 std::string 没有办法如此操作，我们可以显式的构造一个无名临时对象：

```
max("luse"s, std::string("乐"));    // Error 为什么？
```

此时就不是我们的 `T` 不明确了，而是函数模板 `max` 不明确，它会和标准库的 `std::max` 产生冲突，虽然我们没有使用 `std::`，但是根据 C++ 的查找规则，（实参依赖查找）ADL，依然可以查找到。

那么我们如何解决呢？很简单，进行有限定名字查找，即使用 `::` 或 `std::` 说明，你到底要调用 “全局作用域”的 max，还是 std 命名空间中的 max。

```
::max("luse"s, std::string("乐"));
```

万能引用与引用折叠

所谓的万能引用（又称转发引用），即**接受左值表达式那形参类型就推导为左值引用，接受右值表达式，那就推导为右值引用。**

比如：

```
template<typename T>
void f(T&&t){}

int a = 10;
f(a);       // a 是左值表达式，f 是 f<int&> 但是它的形参类型是 int&
f(10);      // 10 是右值表达式，f 是 f<int> 但它的形参类型是 int&&
```

> 被推导为 `f<int&>` 涉及到了特殊的推导规则：如果 P 是到无 cv 限定模板形参的右值引用（也就是转发引用）且对应函数的调用实参是左值，那么将到 A 的左值引用类型用于 A 的位置进行推导。

通过模板或 typedef 中的类型操作可以构成引用的引用，此时适用引用折叠（reference collapsing）规则：

- **右值引用的右值引用折叠成右值引用，所有其他组合均折叠成左值引用。**

```
typedef int&  lref;
typedef int&& rref;
int n;

lref&  r1 = n; // r1 的类型是 int&
lref&& r2 = n; // r2 的类型是 int&
rref&  r3 = n; // r3 的类型是 int&
rref&& r4 = 1; // r4 的类型是 int&&
template <class Ty>
constexpr Ty&& forward(Ty& Arg) noexcept {
    return static_cast<Ty&&>(Arg);
}

int a = 10;             // 不重要
```

```
::forward<int>(a);      // 返回 int&& 因为 Ty 是 int，Ty&& 就是 int&&
::forward<int&>(a);     // 返回 int& 因为 Ty 是 int&，Ty&& 就是 int&
::forward<int&&>(a);    // 返回 int&& 因为 Ty 是 int&&，Ty&& 就是 int&&
```

有默认实参的模板类型形参

就如同函数形参可以有默认值一样，模板形参也可以有默认值。当然了，这里是“类型形参”（后面会讲非类型的）。

```
template<typename T = int>
void f();

f();            // 默认为 f<int>
f<double>();    // 显式指明为 f<double>
using namespace std::string_literals;

template<typename T1,typename T2,typename RT =
    decltype(true ? T1{} : T2{}) >

RT max(const T1& a, const T2& b) { // RT 是 std::string
    return a > b ? a : b;
}

int main(){
    auto ret = ::max("1", "2"s);
    std::cout << ret << '\n';
}
```

> #25 GCC 编译器有 **BUG**，自行注意。

以上这个示例你可能有很多疑问，我们第一次使用了多个模板类型形参，并且第三个模板类型形参给了默认值，但是这个值似乎有点难以理解，我们后面慢慢讲解。

```
max(const T1& a, const T2& b)
```

让 max 函数模板接受两个参数的时候不需要再是相同类型，那么这自然而然就会引入另一个问题了，如何确定返回类型？

```
typename RT = decltype(true ? T1{} : T2{})
```

我们从最里面开始看：

```
decltype(true ? T1{} : T2{})
```

这是一个三目运算符表达式。然后外面使用了 decltype 获取这个表达式的类型，那么问题是，为什么是 true 呢？以及为什么需要 T1{}，T2{} 这种形式？

1. 我们为什么要设置为 **true**？

   其实无所谓，设置 false 也行，true 还是 false 不会影响三目表达式的类型。这涉及到了一些复杂的规则，简单的说就是三目表达式要求第二项和第三项之间能够隐式转换，然后整个表达式的类型会是 **“公共”类型。**

   比如第二项是 int 第三项是 double，三目表达式当然会是 double。

```
using T = decltype(true ? 1 : 1.2);
using T2 = decltype(false ? 1 : 1.2);
```

T 和 T2 都是 double 类型。

2. 为什么需要 `T1{}`，`T2{}` 这种形式?

   没有办法，必须构造临时对象来写成这种形式，这里其实是不求值语境，我们只是为了写出这样一种形式，让 decltype 获取表达式的类型罢了。

   模板的默认实参的和函数的默认实参大部分规则相同。

   `decltype(true ? T1{} : T2{})` 解决了。

   事实上上面的写法都十分的丑陋与麻烦，我们可以使用 auto 简化这一切。

```
template<typename T,typename T2>
auto max(const T& a, const T2& b) -> decltype(true ? a : b){
    return a > b ? a : b;
}
```

这是 C++11 后置返回类型，它和我们之前用默认模板实参 `RT` 的区别只是稍微好看了一点吗?

不，**他们的返回类型是不一样的**，如果函数模板的形参是**类型相同** `true ? a : b` 表达式的类型是 `const T&`；如果是 `max(1, 2)` 调用，那么也就是 `const int&`；而前面的例子只是 `T` 即 `int`（前面都是用模板类型参数直接构造临时对象，而不是有实际对象，自然如此，比如 `T{}`）。

> 假设以 `max(1,1.0)` 调用，那么自然返回类型不是 `const T&`
>
> ```
> max(1, 2.0); // 返回类型为 double
> ```
>
> 下面的 `max` 定义也类似，涉及的规则不再强调。

使用 C++20 简写函数模板，我们可以直接再简化为：

```
decltype(auto) max(const auto& a, const auto& b)  {
    return a > b ? a : b;
}
```

效果和上面使用后置返回类型的写法完全一样；C++14 引入了两个特性：

1. 返回类型推导（也就是函数可以直接写 auto 或 decltype(auto) 做返回类型，而不是像 C++11 那样，只是后置返回类型。
2. `decltype(auto)` *“。如果返回类型没有使用 decltype(auto)，那么推导遵循模板实参推导的规则进行”*，我们上面的 `max` 示例如果不使用 decltype(auto)，按照模板实参的推导规则，是不会有引用和 cv 限定的，就只能推导出返回 `T` 类型。

> 大家需要注意后置返回类型和返回类型推导的区别，他们不是一种东西，后置返回类型虽然也是写的 `auto`，但是它根本没推导，只是占位。

模板的默认实参无处不在，比如标准库的 std::vector，std::string，当然了，这些都是类模板，我们以后会讲到。

非类型模板形参

既然有"类型模板形参"，自然有非类型的，顾名思义，也就是模板不接受类型，而是接受值或对象。

```
template<std::size_t N>
void f() { std::cout << N << '\n'; }

f<100>();
```

非类型模板形参有众多的规则和要求，目前，你简单认为需要参数是"常量"即可。

非类型模板形参当然也可以有默认值：

```
template<std::size_t N = 100>
void f() { std::cout << N << '\n'; }

f();     // 默认       f<100>
f<66>(); // 显式指明   f<66>
```

后续我们会有更多详细讲解和应用。

重载函数模板

函数模板与非模板函数可以重载。

这里会涉及到非常复杂的函数重载决议[3]，即选择到底调用哪个函数。

我们用一个简单的示例展示一部分即可：

```
template<typename T>
void test(T) { std::puts("template"); }

void test(int) { std::puts("int"); }

test(1);         // 匹配到test(int)
test(1.2);       // 匹配到模板
test("1");       // 匹配到模板
```

- 通常优先选择非模板的函数。

可变参数模板

和其他语言一样，C++ 也是支持可变参数的，我们必须使用模板才能做到。

老式 C 语言的变长实参有众多弊端，参见。

同样的，它的规则同样众多繁琐，我们不会说太多，以后会用到的，我们当前还是在入门阶段。

我们提一个简单的需求：

> 我需要一个函数 sum，支持 sum(1,2,3.5,x,n...) 即函数 sum 支持任意类型，任意个数的参数进行调用，你应该如何实现?

首先就要引入一个东西：**形参包**

> 本节以 **C++14** 标准进行讲述。

模板形参包是接受零个或更多个模板实参（非类型、类型或模板）的模板形参。函数形参包是接受零个或更多个函数实参的函数形参。

```
template<typename...Args>
void sum(Args...args){}
```

这样一个函数，就可以接受任意类型的任意个数的参数调用，我们先观察一下它的语法和普通函数有什么不同。

模板中需要 typename 后跟三个点 Args，函数形参中需要用模板类型形参包后跟着三个点 再 args。

args 是函数形参包，Args 是类型形参包，他们的名字我们可以自定义。

args 里，就存储了我们传入的全部的参数，Args 中存储了我们传入的全部参数的类型。

那么问题来了，存储很简单，我们要如何把这些东西取出来使用呢？这就涉及到另一个知识：**形参包展开**。

```
void f(const char*, int, double) { puts("值"); }
void f(const char**, int*, double*) { puts("&"); }

template<typename...Args>
void sum(Args...args){  // const char * args0, int args1, double args2
    f(args...);   // 相当于 f(args0, args1, args2)
    f(&args...);  // 相当于 f(&args0, &args1, &args2)
}

int main() {
    sum("luse", 1, 1.2);
}
```

sum 的 `Args...args` 被展开为 `const char * args0, int args1, double args2`。

这里我们需要定义一个术语：**模式**。

后随省略号且其中至少有一个形参包的名字的**模式**会被展开成零个或更多个**逗号分隔**的模式实例。

`&args...` 中 `&args` 就是模式，在展开的时候，模式，也就是省略号前面的一整个表达式，会被不停的填入对象并添加 `&`，然后逗号分隔。直至形参包的元素被消耗完。

那么根据这个，我们就能写出一些有意思的东西，比如一次性把他们打印出来：

```
template<typename...Args>
void print(const Args&...args){    // const char (&args0)[5], const int & args1,
const double & args2
    int _[]{ (std::cout << args << ' ' ,0)... };
}

int main() {
    print("luse", 1, 1.2);
}
```

一步一步看：`(std::cout << args << ' ' ,0)...` 是一个包展开，那么它的模式是：`(std::cout << args << ' ' ,0)`，实际展开的时候是：

```
(std::cout << arg0 << ' ' ,0), (std::cout << arg1 << ' ' ,0),(std::cout << arg2
<< ' ' ,0)
```

很明显是为了打印，对，但是为啥要括号里加个逗号零呢？这是因为逗号表达式是从左往右执行的，返回最右边的值作为整个逗号表达式的值，也就是说：每一个 `(std::cout << arg0 << ' ' ,0)` 都会返回 0，这主要是为了符合语法，用来初始化数组。我们创建了一个数组 `int _[]` ，最终这些 0 会用来初始化这个数组，当然，这个数组本身没有用，**只是为了创造合适的包展开场所**。

> 为了简略，我们不详细说明有哪些展开场所，不过我们上面使用到的是在*花括号包围的初始化器*中展开。

- ***只有在合适的形参包展开场所才能进行形参包展开。***

> ```
> template<typename ...Args>
> void print(const Args &...args) {
> (std::cout << args << " ")...; // 不是合适的形参包展开场所 Error!
> }
> ```

▶ **细节**

```
template<typename...Args>
void print(const Args&...args){
    int _[]{ (std::cout << args << ' ' ,0)... };
}
```

我们先前的函数模板 `print` 的实现还存在一些问题，如果形参包为空呢？

也就是说，假设我如此调用：

```
print(); // Error!
```

目前这个写法在参数列表为空时会导致 `_` 为*非良构*的 **0 长度数组**，在严格的模式下造成*编译错误*。

解决方案是在数组中添加一个元素使其长度始终为正。

```
template<typename...Args>
 void print(const Args&...args){
    int _[]{ 0, (std::cout << args << ' ' ,0)... };
 }
```

大家不用感到奇怪，当形参包为空的时候，也就基本相当于

```
int _[]{ 0, };
```

也就是当做直接去掉 `(std::cout << args << ' ' ,0)...` 即可。

不用感到奇怪，花括号初始化器列表一直都允许有一个尾随的逗号。从古代 C++ 起 `int a[] = {0, };` 就是合法的定义

> ***可是我为什么要允许空参（print()）调用呢?***

这里的确完全没必要，不过我们出于教学目的，可以提及一下。

这个示例其实还有修改的余地：**我们的本意并非是创造一个局部的数组，我们只是想执行其中的副作用（打印）。**

让这个数组对象直到函数结束生存期才结束，实在是太晚了，我们可以创造一个临时的数组对象，这样它的生存期也就是那一行罢了：

```
template<typename...Args>
void print(const Args&...args) {
    using Arr = int[]; // 创建临时数组，需要使用别名
    Arr{ 0, (std::cout << args << ' ' ,0)... };
}
```

这没问题，但是还不够，在某些编译器（clang）这会造成编译器的警告，想要解决也很简单，将它变为一个*弃值表达式*，也就是转换到 `void`。

> *弃值表达式*是只用来**实施它的副作用的表达式**。从这种表达式计算的值会被舍弃。

```
template<typename...Args>
void print(const Args&...args){
    using Arr = int[];
    (void)Arr{ 0, (std::cout << args << ' ' ,0)... };
}
```

此时编译器就开心了，不再有警告。并且也很合理，我们的确只是需要实施副作用而不需要“值”。

我们再给出一个数组的示例：

```
template<typename...Args>
void print(const Args&...args) {
    int _[]{ (std::cout << args << ' ' ,0)... };
}

template<typename T,std::size_t N, typename...Args>
void f(const T(&array)[N], Args...index) {
    print(array[index]...);
}

int main() {
    int array[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    f(array, 1, 3, 5);
}
```

> 在*函数形参列表*中展开。

我们复用了之前写的 print 函数，我们看新的 **f** 函数即可。

`const T(&array)[N]` 注意，这是一个数组引用，我们也使用到了非类型模板形参 `N`；加括号，`(&array)` 只是为了区分优先级。那么这里的 `T` 是 int，`N` 是 10，组成了一个数组类型。

不必感到奇怪，内建的数组类型，其 size 也是类型的一部分，这就如同 `int[1]` 和 `int[2]` 不是一个类型一样，很正常。

`print(array[index]...);` 其中 `array[index]...` 是包展开，`array[index]` 是模式，实际展开的时候就是：

```
array[arg0], array[arg1], array[arg2]
```

到此，如果你自己写了，理解了这两个示例，那么你应该就能正确的使用形参包展开，那就可以正确的使用基础的可变参数函数。

那么回到最初的需求，实现一个 `sum`：

```
#include <iostream>
#include <type_traits>

template<typename...Args,typename RT = std::common_type_t<Args...>>
RT sum(const Args&...args) {
    RT _[]{ args... };
    RT n{};
    for (int i = 0; i < sizeof...(args); ++i) {
        n += _[i];
    }
    return n;
}

int main() {
    double ret = sum(1, 2, 3, 4, 5,6.7);
    std::cout << ret << '\n';       // 21.7
}
```

`std::common_type_t` 的作用很简单，就是确定我们传入的共用类型，说白了就是这些东西都能隐式转换到哪个，那就会返回那个类型。

`RT _[]{ args... };` 创建一个数组，形参包在它的初始化器中展开，初始化这个数组，数组存储了我们传入的全部的参数。

至于 `sizeof...` 很简单，单纯的获取形参包的元素个数。

其实也可以不写这么复杂，我们不用手动写循环，直接调用标准库的求和函数。

我们简化一下：

```
template<typename...Args,typename RT = std::common_type_t<Args...>>
RT sum(const Args&...args) {
    RT _[]{ args... };
    return std::accumulate(std::begin(_), std::end(_), RT{});
}
```

`RT{}` 构造一个临时无名对象，表示初始值，std::begin 和 std::end 可以获取数组的首尾地址。

当然了，非类型模板形参也可以使用形参包，我们举个例子：

```cpp
template<std::size_t... N>
void f(){
    std::size_t _[]{ N... }; // 展开相当于 1UL, 2UL, 3UL, 4UL, 5UL
    std::for_each(std::begin(_), std::end(_),
        [](std::size_t n){
            std::cout << n << ' ';
        }
    );
}
f<1, 2, 3, 4, 5>();
```

这很合理，无非是让模板形参存储的不再是类型形参包，而是参数形参包罢了。

在后面的内容，我们还会向你展示新的形参包展开方式：C++17 折叠表达式。不用着急。

模板分文件

新手经常会有一个想法就是，对模板进行分文件，写成 `.h` `.cpp` 这种形式。

这显然是不可以的，我们给出了一个项目示例。

> 后续会讲如何处理

在聊为什么不可以之前，我们必须先从头讲解编译链接，以及 `#include` 的知识，不然你将无法理解。

include 指令

先从预处理指令 `#include` 开始，你知道它会做什么吗？

很多人会告诉你，**它就是简单的替换**，的确，没有问题，但是我觉得不够明确，我给你几个示例：

`array.txt`

```
1,2,3,4,5
```

`main.cpp`

```cpp
#include<iostream>

int main(){
    int arr[] = {
#include"array.txt"
    };
    for(int i = 0; i < sizeof(arr)/sizeof(int); ++i)
            std::cout<< arr[i] <<' ';
    std::cout<<'\n';
}
```

> ```
> g++ main.cpp -o main
> ```

> ```
> ./main
> ```

直接编译运行，会打印出 `1 2 3 4 5`。

`#include"array.txt"` 直接被替换为了 `1,2,3,4,5`，所以 arr 是：

```
int arr[] = {1,2,3,4,5};
```

或者我们可以使用 gcc 的 `-E` 选项来查看预处理之后的文件内容：

main2.cpp

```
int main(){
    int arr[] = {
#include"array.txt"
    };
}
```

去除头文件打印之类的是因为，iostream 的内容非常庞大，不利于我们关注数组 arr。

> ```
> g++ -E main2.cpp
> ```

```
# 0 "main2.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "main2.cpp"
int main(){
    int arr[] = {
# 1 "array.txt" 1
1,2,3,4,5
# 4 "main2.cpp" 2
    };
}
```

`# 0` `# 1` 这些是 gcc 的行号更改指令。不用过多关注，不是当前的重点，明白 #include 会进行替换即可。

分文件的原理是什么?

我们通常将函数声明放在 `.h` 文件中，将函数定义放在 `.cpp` 文件中，我们只需要在需要使用的文件中 `include` 一个 `.h` 文件；我们前面也说了，`include` 就是复制，事实上是把函数声明复制到了我们当前的文件中。

```
//main.cpp
#include "test.h"

int main(){
    f();    // 非模板，OK
}
```

test.h 只是存放了函数声明，函数定义在 test.cpp 中，我们编译的时候是选择编译了 `main.cpp` 与 `test.cpp` 这两个文件，那么为什么程序可以成功编译运行呢?

是怎么找到函数定义的呢？明明我们的 main.cpp 其实预处理过后只有函数声明而没有函数定义。

这就是链接器做的事情，如果编译器在编译一个翻译单元（如 main.cpp）的时候，如果发现找不到函数的定义，那么就会空着一个符号地址，将它编译为目标文件。期待链接器在链接的时候去其他的翻译单元找到定义来填充符号。

我们的 `test.cpp` 里面存放了 `f` 的函数定义，并且具备外部链接，在编译成目标文件之后之后，和 `main.cpp` 编译的目标文件进行链接，链接器能找到函数 `f` 的符号。

不单单是函数，全局变量等都是这样，这是编译链接的基本原理和步骤。

> 类会有所不同，总而言之后续视频会单独讲解的。

那么不能模板不能分文件[4]的原因就显而易见了，我们在讲使用模板的时候就说了：

- **模板，只有你"用"了它，才会生成实际的代码。**

你单纯的放在一个 `.cpp` 文件中，它不会生成任何实际的代码，自然也没有函数定义，也谈不上链接器找符号了。

> 所以模板通常是直接放在 `.h` 文件中，而不会分文件。或者说用 `.hpp` 这种后缀，这种约定俗成的，代表这个文件里放的是模板。

总结

事实上函数模板的各种知识远不止如此，但也足够各位目前的学习与使用了。

不用着急，后面会有更多的技术和函数模板一起结合使用的，本节所有的代码示例请务必全部理解和自己亲手写一遍，通过编译，有任何不懂一定要问，提出来。

类模板

本节将介绍类模板

初识类模板

类模板不是类，只有实例化类模板，编译器才能生成实际的类。

定义类模板

下面是一个类模板，它和普通类的区别只是多了一个 `template<typename T>`

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

和函数模板一样，其实类模板的语法也就是：

```
template< 形参列表 > 类声明
```

几乎所有我们前面讲的，**函数模板中形参列表能写的东西，类模板都可以。**

同样的，我们的类模板一样可以用 `class` 引入类型形参名，一样不能用 `struct`

```
template<class T>
struct Test{};
```

使用类模板

下面展示了如何使用类模板 `Test`

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

int main(){
    Test<void> t;
    Test<int> t2;
    //Test t;       // Error!
}
```

我们必须显式的指明类模板的类型实参，并且没有办法推导，事实上这个空类在这里本身没什么意义。

或许我们可以这样：

```
template<typename T>
struct Test{
    T t;
};
```

这理所应当，类模板能使用类模板形参，声明自己的成员，那么如何使用呢？

```
// Test<void> t;  // Error!
Test<int> t2;
// Test t3;       // Error!
Test t4{ 1 };     // C++17 OK!
```

- `Test<void>` 我们稍微带入一下，模板的 `T` 是 `void` 那 `T t` 是？所以很合理
- `Test t4{ 1 };` C++17 增加了类模板实参推导，也就是说类模板也可以像函数模板一样被推导，而不需要显式的写明数据了，这里的 `Test` 被推导为 `Test<int>`。

不单单是聚合体，当然，写构造函数也可以：

```
template<typename T>
struct Test{
    Test(T v) :t{ v } {}
private:
    T t;
};
```

类模板参数推导

这涉及到一些非常复杂的规则，不过我们不用在意。

对于简单的类模板，通常可以普通的类似函数模板一样的自动推导，比如前面提到的 `Test` 类型，又或者下面：

```
template<class T>
struct A{
    A(T, T);
};
auto y = new A{1, 2}; // 分配的类型是 A<int>
```

new 表达式中一样可以。

同样的可以像函数模板那样加上许多的修饰：

```
template<class T>
struct A {
    A(const T&, const T&);
};
```

多的就不用再提。

用户定义的推导指引

举个例子，我要让一个类模板，如果推导为 int，就让它实际成为 size_t：

```
template<typename T>
struct Test{
    Test(T v) :t{ v } {}
private:
    T t;
};

Test(int) -> Test<std::size_t>;

Test t(1);      // t 是 Test<size_t>
```

如果要类模板 `Test` 推导为指针类型，就变成数组呢？

```
template<typename T>
Test(T*) -> Test<T[]>;

char* p = nullptr;

Test t(p);      // t 是 Test<char[]>
```

推导指引的语法还是简单的，如果只是设计具体类型，那么只需要：

`模板名称(类型a)->模板名称<想要让类型a被推导为的类型>`

如果涉及的是一类类型，那么就需要加上 `template`，然后使用它的模板形参。

我们提一个稍微有点难度的需求：

```cpp
template<class Ty, std::size_t size>
struct array {
    Ty arr[size];
};

::array arr{1, 2, 3, 4, 5};     // Error!
```

类模板 array 同时使用了类型模板形参与非类型模板形参，保有了一个成员是数组。

它无法被我们直接推导出类型，此时就需要我们自己**定义推导指引**。

这会用到我们之前在函数模板里学习到的形参包。

```cpp
template<typename T, typename ...Args>
array(T t,Args...) -> array<T, sizeof...(Args) + 1>;
```

原理很简单，我们要给出 array 的模板类型，那么就让模板形参单独写一个 T 占位，放到形参列表中，并且写一个模板类型形参包用来处理任意个参数；获取 array 的 size 也很简单，直接使用 sizeof... 获取形参包的元素个数，然后再 +1 ，因为先前我们用了一个模板形参占位。

标准库的 `std::array` 的推导指引，原理和这个一样。

有默认实参的模板形参

和函数模板一样，类模板一样可以有默认实参。

```cpp
template<typename T = int>
struct X{};

X x;    // x 是 X<int> C++17 起 OK
X<> x2; // x2 是 X<int>
```

必须达到 **C++17** 有 `CTAD`，才可以在全局、函数作用域声明为 `X` 这种形式，才能省略 `<>`。

但是在类中声明一个，有默认实参的类模板类型的数据成员（静态或非静态，是否类内定义都无所谓），不管是否达到 C++17，都不能省略 `<>`。

```cpp
template<typename T = int>
struct X {};

struct Test{
    X x;                // Error
    X<> x2;             // OK
    static inline X x3; // Error
};
```

> 但是 gcc13.2 有不同行为，开启 `std=c++17`，类内定义的静态数据成员省略 `<>` 可以通过编译。
> 但是，总而言之，不要类内声明中省略 `<>`。

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

struct Test{
    static inline X x3;   // OK
};

int main(){

}
```

`MinGw clang 16.02` 与 `msvc` 均不可通过编译。

标准库中也经常使用默认实参：

`std::vector`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

`std::string`

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_string;
```

当然了，也可以给非类型模板形参以默认值，虽然不是很常见：

```
template<class T, std::size_t N = 10>
struct Arr
{
    T arr[N];
};

Arr<int> x;     // x 是 Arr<int,10> 它保有一个成员 int arr[10]
```

知道这些即可，这很合理，毕竟函数模板可以，你类模板也可以。

非类型模板形参

前面其实已经提了，像 `std::array` 都是有非类型模板形参的，这没有什么问题，类似于函数模板。

模板模板形参

类模板的模板类型形参可以接受一个类模板作为参数，我们将它称为：模板模板形参。

先随便给出一个简单的示例：

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

template<template<typename T> typename C>
struct Test {};

Test<X>arr;
```

模板模板形参的语法略微有些复杂，我们需要理解一下，先把外层的 `template<>` 去掉。

`template<typename T> typename C` 我们分两部分看就好

- 前面的 `template<typename T>` 就是我们要接受的类模板它的模板列表，是需要一模一样的，比如类模板 X 就是。
- 后面的 `typename` 是语法要求，需要声明这个模板模板形参的名字，可以自定义，这样就引入了一个模板模板形参。

下面是详细的语法形式：

```
template < 形参列表 > typename(C++17)|class 名字(可选)                (1)
template < 形参列表 > typename(C++17)|class 名字(可选) = default      (2)
template < 形参列表 > typename(C++17)|class ... 名字(可选)            (3) (C++11 起)
```

1. **可以有名字的模板模板形参。**

```
template<typename T>
struct my_array{
    T arr[10];
};

template<typename T,template<typename T> typename C >
struct Array {
    C<T>array;
};

Array<int, my_array>arr;    // arr 保有的成员是 my_array<int> 而它保有了 int arr[10]
```

2. **有默认模板且可以有名字的模板模板形参。**

```
template<typename T>
struct my_array{
    T arr[10];
};

template<typename T, template<typename T> typename C = my_array >
struct Array {
    C<T>array;
};

Array<int>arr;      // arr 的类型同（1），模板模板形参一样可以有默认值
```

3. **可以有名字的模板模板形参包。**

> 其实就是形参包的一种，能接受任意个数的类模板

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

template<typename T>
struct X2 {};

template<template<typename T>typename...Ts>
struct Test{};

Test<X, X2, X, X>t;      // 我们可以传递任意个数的模板实参
```

当然了，模板模板形参也可以和非类型模板形参一起使用，都是一样的，比如：

```
template<std::size_t N>
struct X {};

template<template<std::size_t> typename C>
struct Test {};

Test<X>arr;
```

注意到了吗？我们省略了其中 `template<std::size_t>` 非类型模板形参的名字，可能通常会写成 `template<std::size_t N>` ，我们只是为了表达这是可以省略了，看自己的需求。

对于普通的有形参包的类模板也都是同理：

```
template<typename... T>
struct my_array{
    int arr[sizeof...(T)];  // 保有的数组大小根据模板类型形参的元素个数
};

template<typename T, template<typename... T> typename C = my_array >
struct Array {
    C<T>array;
};

Array<int>arr;
```

成员函数模板

成员函数模板基本上和普通函数模板没多大区别，唯一需要注意的是，它大致有两类：

- 类模板中的成员函数模板
- 普通类中的成员函数模板

需要注意的是：

```
template<typename T>
struct Class_template{
    void f(T) {}
};
```

`Class_template` 的成员函数 f，它不是函数模板，它就是普通的成员函数，在类模板实例化为具体类型的时候，成员函数也被实例化为具体。

1. **类模板中的成员函数模板**

```
template<typename T>
struct Class_template{
    template<typename... Args>
    void f(Args&&...args) {}
};
```

`f` 就是成员函数模板，通常写起来和普通函数模板没多大区别，大部分也都支持，比如形参包。

2. **普通类中的成员函数模板**

```
struct Test{
    template<typename...Args>
    void f(Args&&...args){}
};
```

`f` 就是成员函数模板，没什么问题。

其实都是字面意思，很好理解，上面的示例都没什么实际的使用，都是语法展示，我相信明白函数模板就自然能明白这些。

可变参数类模板

形参包与包展开等知识，在类模板中是通用的。

```
template<typename ...Args>
struct X {
    X(Args...args) :value{ args... } {} // 参数展开
    std::tuple<Args...>value;           // 类型形参包展开
};

X x{ 1,"2",'3',4. };    // x 的类型是 X<int,const char*,char,double>
std::cout << std::get<1>(x.value) << '\n'; // 2
```

`std::tuple` 是一个模板类，我们用来存储任意类型任意个数的参数，我们指明它的模板实参是使用的模板的类型形参包展开，`std::tuple<Args...>` 展开后成为 `std::tuple<int,const char*,char,double>` 。

构造函数中使用成员初始化列表来初始化成员 value，没什么问题，正常展开。

需要注意的是字符串字面量的类型是 `const char[N]` ，之所以被推导为 `const char*` 在于数组之间不能“拷贝”。它隐式转换为了指向数组首地址的指针，类型自然也被推导为 `const char*` 。

```
int arr[1]{1};
int arr2[2]{1,2};
arr = arr2;           // Error!
```

```
int a = 0;
int b = a;            // OK!

int arr[1]{1};
int arr2[1] = arr;    // Error!
int arr3[1] = {arr};  // Error!
```

类模板分文件

和前面提到的函数模板分文件的原因一样，类模板也没有办法分文件。

我们给出了一个项目示例，展示类模板通常分文件的情况。

通常就是统一写到 `.h` 文件中，或者大家约定俗成了一个 `.hpp` 后缀，这个通常用来放模板。

> 我们后面会单独做一个内容处理这些情况。

总结

类模板的知识远不止如此，不过目前也足够使用了，后续还会有补充。

我们写的类模板的内容没有函数模板那么多，主要在于很多内容是和函数模板重复的，很多特性彼此之间是相通的，我们就没必要讲那么多，所以需要注意，不要跳着看。

变量模板

本节将介绍 C++14 变量模板

初识变量模板

变量模板不是变量，只有实例化的变量模板，编译器才会生成实际的变量。

变量模板实例化后简单的说**就是一个全局变量**，所以也不用担心生存期的问题。

定义变量模板

```
template<typename T>
T v;
```

就这么简单，毫无难度。

当然了，既然是变量，自然可以有各种的修饰，比如 cv 限定，比如 `constexpr` ,当然也可以有初始化器，比如 `{}` 、 `= xxx` 。

```
template<typename T>
constexpr T v{};
```

使用变量模板

```
template<typename T>
constexpr T v{};

v<int>;     // 相当于 constexpr int v = 0;
```

我们知道 `constexpr` 附带了 `const` 类型，所以其实：

```
std::cout << std::is_same_v<decltype(v<int>),const int> << '\n';
```

> std::is_same_v 其实也是个变量模板，在 C++17 引入。这里用来比较两个类型是否相同，如果相同返回 1，不相同返回 0。暂时不用纠结它是如何实现的，后续会手搓。

会打印 1，也就是 `v<int>` 的类型其实就是 `const int`。

我们再提出一个问题，`v<int>` 和 `v<double>` 有什么关系吗？

> 最早在函数模板中，我们强调了“**同一个函数模板生成的不同类型的函数，彼此之间没有任何关系**”，这句话放在类模板、变量模板中，也同样适用。

```
std::cout << &v<int> << '\n';
std::cout << &v<double> << '\n';
```

以上示例打印的地址不会相同。

有默认实参的模板形参

变量模板和函数模板、类模板一样，支持模板形参有默认实参。

```
template<typename T = int>
constexpr T v{};

int b = v<>;        // v 就是 v<int> 也就是 const int v = 0
```

与函数模板和类模板不同，即使模板形参有默认实参，依然要求写明 `<>`。

非类型模板形参

变量模板和函数模板、类模板一样，支持非类型模板形参。

```
template<std::size_t N>
constexpr int v = N;

std::cout << v<10> << '\n'; // 等价 constexpr int v = 10;
```

当然，它也可以有默认值：

```cpp
template<std::size_t N = 66>
constexpr int v = N;

std::cout << v<10> << '\n';
std::cout << v<> << '\n';
```

可变参数变量模板

变量模板和函数模板、类模板一样，支持形参包与包展开。

```cpp
template<std::size_t...values>
constexpr std::size_t array[]{ values... };

int main() {
    for (const auto& i : array<1, 2, 3, 4, 5>) {
        std::cout << i << ' ';
    }
}
```

array 是一个数组，我们传入的模板实参用来推导出这个数组的大小以及初始化。

`{values...}` 展开就是 `{1, 2, 3, 4, 5}`。

```cpp
std::cout << std::is_same_v<decltype(::array<1, 2, 3, 4, 5>), const
std::size_t[5]>; // 1
```

在 msvc 会输出 `1`，但是 gcc、clang，却会输出 `0`。***msvc 是正确的。***
gcc 与 clang 不认为 `array<1, 2, 3, 4, 5>` 与 `const std::size_t[5]` 类型相同；它们认为 `array<1, 2, 3, 4, 5>` 与 `const std::size_t[]` 类型相同，这显然是个 **bug**。

> 可以参见 llvm issues.

类静态数据成员模板

在类中也可以使用变量模板。

类静态数据成员

讲模板之前先普及一下静态数据成员的基本知识，因为**网上很多资料都是乱讲**，所以有必要重复强调一下。

```cpp
struct X{
    static int n;
};
```

n 是一个 X 类的静态数据成员，它在 X 中声明，但是却没有定义，我们需要类外定义。

```cpp
int X::n;
```

或者在 C++17 以 inline 或者 constexpr 修饰。

> 因为 C++17 规定了 **inline** 修饰静态数据成员，那么这就是在类内定义，不再需要类外定义。
> constexpr 在 C++17 修饰静态数据成员的时候，蕴含了 **inline**。

```cpp
struct X {
    inline static int n;
};

struct X {
    constexpr static int n = 1;      // constexpr 必须初始化，并且它还有 const 属性
};
```

模板

与其他静态成员一样，静态数据成员模板的需要一个定义。

```cpp
struct limits{
    template<typename T>
    static const T min; // 静态数据成员模板的声明
};

template<typename T>
const T limits::min = {}; // 静态数据成员模板的定义
```

当然，如果支持 C++17 你也可以选择直接以 `inline` 修饰。

变量模板分文件

变量模板和函数模板、类模板一样，通常写法不支持分文件，原因都一样。

总结

变量模板其实很常见，在 C++17，所有元编程库的类型特征均添加了 `_v` 的版本，就是使用的变量模板，比如 `std::is_same_v` 、`std::is_pointer_v` 等；我们在后面会详细讲解。

如果学到这里了，如果你注意到，函数模板、类模板、变量模板，很多语法是共通的，是越学越简单，代表你思考了。

后续还有很多内容是一起的，比如模板偏特化、全特化、显式实例化等。

模板全特化

本节将介绍模板全特化。

其实很多东西都能进行全特化，不过我们围绕着之前的内容：函数模板、类模板、变量模板来展开。

函数模板全特化

给出这样一个函数模板 `f`，你可以看到，它的逻辑是返回两个对象相加的结果，那么如果我有一个需求：“**如果我传的是一个 double 一个 int 类型，那么就让它们返回相减的结果**”。

```cpp
template<typename T,typename T2>
auto f(const T& a, const T2& b)  {
    return a + b;
}
```

> C++14 允许函数返回声明的 auto 占位符自行推导类型。

这种定制的需求很常见，此时我们就需要使用到模板全特化：

```
template<>
auto f(const double& a, const int& b){
    return a - b;
}
```

> 当特化函数模板时，如果模板实参推导能通过函数实参提供，那么就**可以忽略它的模板实参**。

语法很简单，只需要先写一个 `template<>` 后面再实现这个函数即可。

不过我们其实有两种写法的，比如上面那个示例，我们还可以写明模板实参。

```
template<>
auto f<double, int>(const double& a, const int& b) {
    return a - b;
}
```

个人建议写明更加明确，因为很多时候模板实参只是函数形参类型的**一部分**而已，比如上面的 `const double&`、`const int&` 只有 `double` 、`int` 是模板实参。

使用：

```
std::cout << f(2, 1) << '\n';    // 3
std::cout << f(2.1, 1) << '\n';  // 1.1
```

类模板全特化

和函数模板一样，类模板一样可以进行全特化。

```
template<typename T> // 主模板
struct is_void{
    static constexpr bool value = false;
};
template<>           // 对 T = void 的显式特化
struct is_void<void>{
    static constexpr bool value = true;
};

int main(){
    std::cout <<std::boolalpha<< is_void<char>::value << '\n';    // false
    std::cout << std::boolalpha << is_void<void>::value << '\n';  // true
}
```

我们使用全特化，实现了一个 `is_void` 判断模板类型实参是不是 `void` 类型。

虽然很简单，但我们还是稍微强调一下：同一个类模板实例化的不同的类，彼此之间毫无关系，而静态数据成员是属于类的，而不是模板类；模板类实例化的不同的类，他们的静态数据成员不是同一个，请注意。

我们知道标准库在 `C++17` 引入了 `is_xxx` 的 `_v` 的版本，就不需要再写 `::value` 了。所以我们也可以这么做，这会使用到变量模板。

```
#include <iostream>

template<typename T> // 主模板
struct is_void{
    static constexpr bool value = false;
};
template<>           // 对 T = void 的显式特化
struct is_void<void>{
    static constexpr bool value = true;
};

template<typename T>
constexpr bool is_void_v = is_void<T>::value;

int main(){
    std::cout <<std::boolalpha<< is_void_v<char> << '\n';    // false
    std::cout << std::boolalpha << is_void_v<void> << '\n';  // true
}
```

我们再给出一个简单的示例：

```
template<typename T>
struct X{
    void f()const{
        puts("f");
    }
};

template<>
struct X<int>{
    void f()const{
        puts("X<int>");
    }
    void f2()const{}

    int n;
};

int main(){
    X<void> x;
    X<int> x_i;
    x.f();         // 打印 f
    //x.f2();      // Error!
    x_i.f();       // 打印 X<int>
    x_i.f2();
}
```

我们要明白，写一个类的全特化，就相当于写一个新的类一样，你可以自己定义任何东西，不管是函数、数据成员、静态数据成员，等等；根据自己的需求。

变量模板全特化

```
#include <iostream>

template<typename T>
constexpr const char* s = "??";

template<>
constexpr const char* s<void> = "void";

template<>
constexpr const char* s<int> = "int";

int main(){
    std::cout << s<void> << '\n'; // void
    std::cout << s<int> << '\n';  // int
    std::cout << s<char> << '\n'; // ??
}
```

语法形式和前面函数模板、类模板都类似，很简单，这个变量模板是类型形参。我们特化了变量模板 `s` 的模板实参为 `void` 与 `int` 的情况，修改 `s` 的初始化器，让它的值不同。

```
template<typename T>
constexpr bool is_void_v = false;

template<>
constexpr bool is_void_v<void> = true;

int main(){
    std::cout << std::boolalpha << is_void_v<char> << '\n';   // false
    std::cout << std::boolalpha << is_void_v<void> << '\n';   // true
}
```

上面的变量模板，模板是类型形参，我们根据类型进行全特化。我们特化了 `is_void_v` 的模板实参为 `void` 的情况，让 `is_void_v` 值 为 `true`。

细节

前面函数、类、变量模板的全特化都讲的很简单，示例也很简单，或者说语法本身大多数时候就是简单的。我们在这里进行一些更多的**补充一些细节。**

特化必须在导致隐式实例化的首次使用之前，在每个发生这种使用的翻译单元中声明：

```
template<typename T> // 主模板
void f(const T&){}

void f2(){
    f(1);  // 使用模板 f() 隐式实例化 f<int>
}

template<> // 错误 f<int> 的显式特化在隐式实例化之后出现
void f<int>(const int&){}
```

如果 f2 中的调用换成 `f(1.)` 就没问题，它隐式实例化的就是 `f<double>` 了。

只有声明没有定义的模板特化可以像其他不完整类型一样使用（例如可以使用到它的指针和引用）：

```
template<class T> // 主模板
class X;
template<>        // 特化（声明，不定义）
class X<int>;

X<int>* p;        // OK：指向不完整类型的指针
X<int> x;         // 错误：不完整类型的对象
```

函数模板和变量模板的显式特化是否为 inline/constexpr/constinit/consteval **只与显式特化自身有关，主模板的声明是否带有对应说明符对它没有影响。**模板声明中出现的属性在它的显式特化中也没有效果：

```
template<typename T>
int f(T) { return 6; }
template<>
constexpr int f<int>(int) { return 6; }   // OK，f<int> 是以 constexpr 修饰的

template<class T>
constexpr T g(T) { return 6; }            // 这里声明的 constexpr 修饰函数模板是无效的
template<>
int g<int>(int) { return 6; }             //OK，g<int> 不是以 constexpr 修饰的

int main(){
    constexpr auto n = f<int>(0);         // OK，f<int> 是以 constexpr 修饰的，可以编译期求值
    //constexpr auto n2 = f<double>(0);   // Error! f<double> 不可编译期求值

    //constexpr auto n3 = g<int>(0);      // Error! 函数模板 g<int> 不可编译期求值

    constexpr auto n4 = g<double>(0);     // OK! 函数模板 g<double> 可编译期求值
}
```

可通过编译。

如果主模板有 constexpr 属性，那么模板实例化的，如 `g<double>` 自然也是附带了 `constexpr`，但是***如果其特化没有，那么以特化为准***（如 `g<int>`）。

特化的成员

特化成员的写法略显繁杂，但是只要明白其逻辑，一切就会很简单。

> **主模板**

```
template<typename T>
struct A{
    struct B {};      // 成员类

    template<class U> // 成员类模板
    struct C {};
};
```

> **特化模板类**。`A<void>`。

```
template<>
struct A<void>{
    void f();       // 类内声明
};

void A<void>::f(){  // 类外定义
    // todo..
}
```

> **特化成员类**。设置 `A<char>` 的情况下 `B` 类的定义。

```
template<>
struct A<char>::B{      // 特化 A<char>::B
    void f();           // 类内声明
};

void A<char>::B::f(){   // 类外定义
    // todo..
}
```

> **特化成员类模板**。设置 `A<int>` 情况下模板类 `C` 的定义。

```
template<>
template<class U>
struct A<int>::C{
    void f();               // 类内声明
};
// template<> 会用于定义被特化为类模板的显式特化的成员类模板的成员
template<>
template<class U>
void A<int>::C<U>::f(){     // 类外定义
    // todo..
}
```

> **特化类的成员函数模板**

其实语法和普通特化函数模板没什么区别，类外的话那就指明函数模板是在 X 类中。

```
struct X{
    template<typename T>
    void f(T){}
```

```
    template<>                  // 类内特化
    void f<int>(int){
        std::puts("int");
    }
};

template<>                      // 类外特化
void X::f<double>(double){
    std::puts("void");
}

X x;
x.f(1);     // int
x.f(1.2);   // double
x.f("");
```

> **特化类模板的成员函数模板**

成员或成员模板可以在多个外围类模板内嵌套。在这种成员的显式特化中，对每个显式特化的外围类模板都有一个 `template<>`。

其实就是注意有几层那就多套几个 `template<>`，并且指明模板类的模板实参。下面这样：就是自定义了 `X<void>` 且 `f<double>` 的情况下的函数。

```
template<typename T>
struct X {
    template<typename T2>
    void f(T2) {}

    template<>
    void f<int>(int) {             // 类内特化，对于 函数模板 f<int> 的情况
        std::puts("f<int>(int)");
    }
};

template<>
template<>
void X<void>::f<double>(double) { // 类外特化，对于 X<void>::f<double> 的情况
    std::puts("X<void>::f<double>");
}

X<void> x;
x.f(1);    // f<int>(int)
x.f(1.2);  // X<void>::f<double>
x.f("");
```

> 视频中的代码，模板类和成员函数模板都用的 `T`，只能在 `msvc` 下运行，gcc 与 clang 有歧义，**需要注意**。

> **类内对成员函数 `f` 的特化**，在 `gcc` **无法通过编译**，根据考察，这是一个很多年前就有的 `BUG`，使用 `gcc` 的开发者自行注意。

总结

我们省略了一些内容，但是以上在我看来也完全足够各位学习使用了。如有需求，查看 cppreference。

模板全特化的语法主要核心在于 `template<>`，以及你需要注意，你到底要写几个 `template<>`。其他的都很简单。

模板偏特化

如果你认真学习了我们上一节内容，本节应当是十分简单的。

模板偏特化这个语法让**模板实参具有一些相同特征**可以自定义，而不是像全特化那样，必须是**具体的**什么类型，什么值。

比如：指针类型，这是一类类型，有 `int*`、`double*`、`char*`，以及自定义类型的指针等等，他们都属于指针这一类类型；可以使用偏特化对指针这一类类型进行定制。

- ***模板偏特化使我们可以对具有相同的一类特征的类模板、变量模板进行定制行为。***

变量模板偏特化

```
template<typename T>
const char* s = "?";            // 主模板

template<typename T>
const char* s<T*> = "pointer";  // 偏特化，对指针这一类类型

template<typename T>
const char* s<T[]> = "array";   // 偏特化，但是只是对 T[] 这一类类型，而不是数组类型，因为 int[] 和 int[N] 不是一个类型

std::cout << s<int> << '\n';            // ?
std::cout << s<int*> << '\n';           // pointer
std::cout << s<std::string*> << '\n';   // pointer
std::cout << s<int[]> << '\n';          // array
std::cout << s<double[]> << '\n';       // array
std::cout << s<int[1]> << '\n';         // ?
```

语法就是正常写主模板那样，然后再定义这个 `s` 的时候，指明模板实参。或者你也可以定义非类型的模板形参的模板，偏特化，都是一样的写法。

不过与全特化不同，全特化不会写 `template<typename T>`，它是直接 `template<>`，然后指明具体的模板实参。

它与全特化最大的不同在于，全特化基本必写 `template<>`，而且定义的时候（如 `s`）是指明具体的类型，而不是一类类型（T*、T[]）。

我们再举个例子：

```
template<typename T,typename T2>
const char* s = "?";

template<typename T2>
const char* s<int, T2> = "T == int";

std::cout << s<char, double> << '\n';       // ?
std::cout << s<int, double> << '\n';        // T == int
std::cout << s<int, std::string> << '\n';   // T == int
```

这种偏特化也是可以的，多个模板实参的情况下，对第一个模板实参为 `int` 的情况进行偏特化。

其他的各种形式无非都是我们提到的这两个示例的变种，类模板也不例外。

类模板偏特化

```
template<typename T,typename T2>
struct X{
     void f_T_T2();                 // 主模板，声明
};

template<typename T, typename T2>
void X<T, T2>::f_T_T2() {}          // 类外定义

template<typename T>
struct X<void,T>{
    void f_void_T();                // 偏特化，声明
};

template<typename T>
void X<void, T>::f_void_T() {}      // 类外定义

X<int, int> x;
x.f_T_T2();         // OK!
X<void, int> x2;
x2.f_void_T();      // OK!
```

稍微提一下类外的写法，不过其实**通常不推荐写到类外**，目前还好；很多情况涉及大量模板的时候，类内声明写到类外非常的麻烦。

我们再举一个偏特化类模板中的类模板，全特化和偏特化一起使用的示例：

```
template<typename T,std::size_t N>
struct X{
    template<typename T_,typename T2>
    struct Y{};
};

template<>
template<typename T2>
struct X<int, 10>::Y<int, T2> {     // 对 X<int,10> 的情况下的 Y<int> 进行偏特化
    void f()const{}
};
```

```
int main(){
    X<int, 10>::Y<int, void>y;
    y.f();                          // OK X<int,10> 和 Y<int>
    X<int, 1>::Y<int, void>y2;
    y2.f();                         // Error! 主模板模板实参不对
    X<int, 10>::Y<void, int>y3;
    y3.f();                         // Error! 成员函数模板模板实参不对
}
```

> 此示例无法在 gcc 通过编译，这是**编译器 BUG 需要注意**。

语法形式是简单的，不做过多的介绍。

其实和全特化没啥区别。

实现 `std::is_same_v`

我们再写一个小示例，实现这个简单的 C++ 标准库设施。

```
template <class, class> // 主模板
inline constexpr bool is_same_v = false;
template <class Ty>     // 偏特化
inline constexpr bool is_same_v<Ty, Ty> = true;
```

这是对变量模板的偏特化，逻辑也很简单，如果两个模板类型参数的类型是一样的，就匹配到下面的偏特化，那么初始化就是 `true`，不然就是 `false`。

因为没有用到模板类型形参，所以我们只是写了 `class` 进行占位；这就和你声明函数的时候，如果形参没有用到，那么就不声明名字一样合理，比如 `void f(int)`。

> 声明为 `inline` 的是因为 内联变量 (C++17 起)可以在被多个源文件包含的头文件中定义。也就是允许多次定义。

总结

我们在一开始的模板全特化花了很多时间讲解各种情况和细节，偏特化除了那个语法上，其他的各种形式并无区别，就不再介绍了。

本节我们给出了三个示例，也是最常见最基础的情况。我们要懂得变通，可能还有以此为基础的各种形式。值得注意的是，模板偏特化还可以和 `SFINAE`[1] 一起使用，这会在我们后续的章节进行讲解，不用着急。

如还有需求，查看 cppreference。

最后强调一句：***函数模板没有偏特化，只有类模板和变量模板可以。***

模板显式实例化解决模板分文件问题

前言

在前面的内容，我们一直讲的都是“**通常写法，函数模板、类模板、变量模板不能分文件**”。

并且阐述了原因，简单的说：**在于模板必须使用了才会生成实际的代码，才会有符号让链接器去链接。**

- **只有实例化模板，编译器才能生成实际的代码。**

> 需要注意，以前说的“**使用模板**”其实就是会 **隐式实例化模板**，编译器根据我们的使用，知道我们需要什么类型的模板，生成实际的代码，比如实际的函数，实际的类，实际的变量等，然后再去调用。

分文件这个问题显然是可以解决的，那就是：**显式实例化模板。**

我们自己指明，到底需要哪些具体的函数。

函数模板显式实例化

```
template 返回类型 名字 < 实参列表 > ( 形参列表 ) ;          (1)
template 返回类型 名字 ( 形参列表 ) ;                      (2)
extern template 返回类型 名字 < 实参列表 > ( 形参列表 ) ;   (3) (C++11 起)
extern template 返回类型 名字 ( 形参列表 ) ;               (4) (C++11 起)
```

1. 显式实例化定义（显式指定所有无默认值模板形参时不会推导模板实参）
2. 显式实例化定义，对所有形参进行模板实参推导
3. 显式实例化声明（显式指定所有无默认值模板形参时不会推导模板实参）
4. 显式实例化声明，对所有形参进行模板实参推导

- ***在模板分文件问题中，几乎不会使用到显式实例化声明。***

因为我们引用 `.h` 文件本身就有声明，除非你准备直接两个 `.cpp`。

> **显式实例化定义强制实例化它所指代的函数或成员函数。**它可以出现在程序中模板定义后的任何位置，而对于给定的实参列表，它在整个程序中只能出现一次，不要求诊断。

> 显式实例化声明（extern 模板）阻止隐式实例化：本来会导致隐式实例化的代码必须改为使用已在程序的别处所提供的显式实例化。(C++11 起)

> 在函数模板特化或成员函数模板特化的显式实例化中，尾部的各模板实参在能从函数参数推导时不需要指定。

类模板显式实例化

```
template 类关键词 模板名 < 实参列表 > ;             (1)
extern template 类关键词 模板名 < 实参列表 > ; (2) (C++11 起)
```

类关键词 - class，struct 或 union

1. 显式实例化定义
2. 显式实例化声明

语法不过多赘述，看使用示例。

使用示例

我们用 cmake 构建了一个简单的项目展示使用显式实例化模板解决类模板函数模板的分文件的问题。

`main.cpp`

```
#include "test_function_template.h"
#include "test_class_template.h"
```

```
int main() {
    f_t(1);
    f_t(1.2);
    f_t('c');
    //f_t("1");   // 没有显式实例化 f_t<const char*> 版本，会有链接错误

    N::X<int>x;
    x.f();
    //x.f2();     // 链接错误，没有显式实例化 X<int>::f2() 成员函数
    N::X<double>x2{};
    //x2.f();     // 链接错误，没有显式实例化 X<double>::f() 成员函数

    N::X2<int>x3; // 我们显式实例化了类模板 X2<int> 也就自然而然实例化它所有的成员，f，f2 函数
    x3.f();
    x3.f2();

    // 类模板分文件 我们写了两个类模板 X X2，他们一个使用了成员函数显式实例化，一个类模板显式实例化，进行对比
    // 这主要在于我们所谓的类模板分文件，其实类模板定义还是在头文件中，只不过成员函数定义在 cpp 罢了。
}
```

test_function_template.h

```
#pragma once

#include <iostream>
#include <typeinfo>

template<typename T>
void f_t(T);
```

test_function_template.cpp

```
#include"test_function_template.h"

template<typename T>
void f_t(T) { std::cout << typeid(T).name() << '\n'; }

template void f_t<int>(int); // 显式实例化定义 实例化 f_t<int>(int)
template void f_t<>(char);   // 显式实例化定义 实例化 f_t<char>(char)，推导出模板实参
template void f_t(double);   // 显式实例化定义 实例化 f_t<double>(double)，推导出模板实参
```

test_class_template.h

```
#pragma once

#include <iostream>
#include <typeinfo>

namespace N {
```

```
    template<typename T>
    struct X {
        int a{};
        void f();
        void f2();
    };

    template<typename T>
    struct X2 {
        int a{};
        void f();
        void f2();
    };
};
```

test_class_template.cpp

```
#include "test_class_template.h"

template<typename T>
void N::X<T>::f(){
    std::cout << "f: " << typeid(T).name() << "a: " << this->a << '\n';
}

template<typename T>
void N::X<T>::f2() {
    std::cout << "f2: " << typeid(T).name() << "a: " << this->a << '\n';
}

template void N::X<int>::f();    // 显式实例化定义 成员函数，这不是显式实例化类模板

template<typename T>
void N::X2<T>::f() {
    std::cout << "X2 f: " << typeid(T).name() << "a: " << this->a << '\n';
}

template<typename T>
void N::X2<T>::f2() {
    std::cout << "X2 f2: " << typeid(T).name() << "a: " << this->a << '\n';
}

template struct N::X2<int>;      // 类模板显式实例化定义
```

值得一提的是，我们前面讲类模板的时候说了类模板的成员函数不是函数模板，但是这个语法形式很像前面的"*函数模板显式实例化*"对不对？的确看起来差不多，不过**这是显式实例化类模板成员函数**，而不是函数模板。

上面的 `f` `f2` 是定义，但是别把它当成函数模板了，那个 `template<typename T>` 是属于类模板的。

类型链接的时候都不存，只需要保证当前文件有类的完整定义，就能使用模板类。

类的完整定义不包括成员函数定义，理论上只要数据成员定义都有就行了。

所以我们只需要显式实例化这个成员函数也能完成类模板分文件，如果有其他成员函数，那么我们就得都显式实例化它们才能使用，或者使用显式实例化类模板，它会实例化自己的所有成员。

总结

如你所见，解决分文件问题很简单，显式实例化就完事了。

再次我们再重复强调一些概念：

模板必须实例化才能使用，实例化就会生成实际代码；有隐式实例化和显式实例化，我们平时粗略的说的“*模板只有使用了才会生成实际代码*”，其实是指使用模板的时候，就会**隐式实例化**，生成实际代码。

分文件自然没有隐式实例化了，那我们就得显式实例化，让模板生成我们想要的代码。

显式实例化解决模板导出静态库动态库

前言

我们使用显式实例化解决模板导出动态静态库这个问题，原因什么的不再过多赘述，总结一句话：

- **模板需要实例化才能生成实际的代码，既然不能隐式实例化，那就显式实例化。**

我们以 `windows` 环境 `vs2022`，`sln` 解决方案为例。

> 不再使用 cmake 主要是本课程也不是教你 cmake 的，这些东西有点麻烦，不如直接用 VS 按钮点点点设置就是。之前用 cmake 项目主要在于 `add_executable`，比较明确，说明编译哪些文件。

我们创建了一个解决方案，其中有四个项目：

1. **测试动态库使用**
2. **测试静态库使用**
3. **用模板生成动态库**
4. **用模板生成静态库**

> 生成动态库和静态库用的代码几乎是一模一样的，只是去掉了 `__declspec(dllexport)`。

> 测试动态库和静态库使用的主要区别在于项目配置，代码上的区别是去掉 `__declspec(dllexport)`。

模板生成动态库与测试

`export_template.h`

```
#pragma once

#include <iostream>
#include <string>

template<typename T>
void f(T);

template<typename T>
struct __declspec(dllexport) X {
    void f();
};
```

`export_template.cpp`

```cpp
#include "export_template.h"

template<typename T>
void f(T) {                                     // 函数模板定义
    std::cout << typeid(T).name() << '\n';
}

template <typename T>
void X<T>::f(){                                 // 类模板中的成员函数模板定义
    std::cout << typeid(T).name() << '\n';
}

template __declspec(dllexport)  void f<int>(int);
template __declspec(dllexport)  void f<std::string>(std::string);

template struct X<int>;     // 类模板显式实例化
```

这很简单，和之前分文件写法的区别只是用了`__declspec(dllexport)`。

> 可以使用 __declspec(dllexport) 关键字从 DLL 中导出数据、函数、类或类成员函数。

以上示例中的函数模板、类模板**显式实例化**，不要放在`.h`文件中，因为**一个显式实例化定义在程序中最多只能出现一次**；如果放在`.h`文件中，被多个翻译单元使用，就会产生问题。

> 当显式实例化函数模板、变量模板 (C++14 起)、类模板的成员函数或静态数据成员，或成员函数模板时，**只需要它的声明可见。**

> **类模板、类模板的成员类或成员类模板在显式实例化之前必须出现完整定义**，除非之前已经出现了拥有相同模板实参的显式特化

我们将生成的`.dll`与`.lib`文件放在了指定目录下，配置了项目的查找路径以供使用。

`run_test.cpp.cpp`

```cpp
#include "export_template.h"
#include <string>

int main(){
    std::string s;
    f(1);
    //f(1.2); // Error!链接错误，没有这个符号
    f(s);
    X<int>x;
    x.f();
}
```

模板生成静态库与测试

没有多大的区别，原理都一样，代码也差不多，不必再讲。

总结

我们没有讲述诸如：项目如何配置，**怎么配置项目生成动态库、静态库**，这也不是我们的重点。在视频中我们会从头构建这些。但是文本的话，不必要从头教学这些基本知识。

我们关注代码上即可。

静态库也没有单独提，因为的确没啥区别。

另外如果你直接打开我们的项目，无法编译或许很正常，请自己根据当前环境，处理编码问题，以及生成动态静态库，配置，使用。

折叠表达式

C++17 折叠表达式让我们能以更加简单的语法形式，更加轻松的进行**形参包展开**。

本节也将重新复习形参包的知识，希望各位不要忘记了。

语法

```
( 形参包 运算符 ... )               (1)
( ... 运算符 形参包 )               (2)
( 形参包 运算符 ... 运算符 初值 )   (3)
( 初值 运算符 ... 运算符 形参包 )   (4)
```

1. 一元右折叠
2. 一元左折叠
3. 二元右折叠
4. 二元左折叠

折叠表达式的实例化按以下方式展开成表达式 e：

1. 一元右折叠 `(E 运算符 ...)` 成为 `(E1 运算符 (... 运算符 (EN-1 运算符 EN)))`
2. 一元左折叠 `(... 运算符 E)` 成为 `(((E1 运算符 E2) 运算符 ...) 运算符 EN)`
3. 二元右折叠 `(E 运算符 ... 运算符 I)` 成为 `(E1 运算符 (... 运算符 (EN-1 运算符 (EN 运算符 I))))`
4. 二元左折叠 `(I 运算符 ... 运算符 E)` 成为 `((((I 运算符 E1) 运算符 E2) 运算符 ...) 运算符 EN)`
   （其中 N 是包展开中的元素数量）

- ***折叠表达式是左折叠还是右折叠，取决于 `...` 是在“形参包”的左边还是右边。***

实现一个 print 函数

> - 以下代码来自01函数模板.md 中的可变参数模板的示例

```
template<typename...Args>
void print(const Args&...args){
   int _[]{ (std::cout << args << ' ' ,0)... };
}
print("luse", 1, 1.2); // luse 1 1.2
```

运用折叠表达式，我们可以简化 `print`，而不再需要创建愚蠢的数组对象 `_`。

```
template<typename...Args>
void print(const Args&...args) {
    ((std::cout << args << ' '), ...);
}
print("luse", 1, 1.2); // luse 1 1.2
```

这显然是 (1)**一元右折叠**，我们一步一步分析：

`(std::cout << args << ' ')` 就是语法中指代的**形参包**（其实说的是含有形参包的运算符表达式）。那么 `,` 逗号就是运算符，最后 `...` 。然后最外层有括号 `()` 符合语法。

函数模板实例化、折叠表达式展开，大概就是：

```
void print(const char(&args0)[5], const int& args1, const double& args2) {
    (std::cout << args0 << ' '), ((std::cout << args1 << ' '), (std::cout <<
args2 << ' '));
}
```

我不建议各位数括号，死记这个规则，知道大概的意思就行，运用逗号运算符进行这个折叠表达式还是简单的，多用用就好。

详细展示语法

折叠表达式这四种语法形式我们都需要学习，并且明白其区别，我们用一个一个示例来展示，你自然会感觉到区别，毕竟**运行结果不一样**。

你可以学不懂或用不到这所有形式，但我总得教。

一元折叠

```
template<typename...Args>
void print(const Args&...args) {
    (...,(std::cout << args << ' '));
}
print("luse", 1, 1.2); // luse 1 1.2
```

这个示例就是参考我们上面用折叠表达式实现的 `print`，只不过一开始的是“一元右折叠”，而我们这个示例是“**一元左折叠**”。

如你所见，这个 `print` 不管使用左折叠还是右折叠，运行结果是一样的，这是为什么呢？

实例化展开后是这样：

```
void print(const char(&args0)[5], const int& args1, const double& args2) {
    ((std::cout << args0 << ' '), (std::cout << args1 << ' ')), (std::cout <<
args2 << ' ');
}
```

其实这个括号根本不影响什么，我们可以得出结论：“*对于逗号运算符，一元左折叠和一元右折叠没有区别*”。

我们用一个非类型模板参数的变量模板来展示在一些情况下左折叠和右折叠是会造成不同结果的：

```cpp
template<int...I>
constexpr int v_right = (I - ...);  // 一元右折叠

template<int...I>
constexpr int v_left = (... - I);   // 一元左折叠

int main(){
    std::cout << v_right<4, 5, 6> << '\n';  //(4-(5-6)) 5
    std::cout << v_left<4, 5, 6> << '\n';   //((4-5)-6) -7
}
```

这个示例很好，那么简单总结一下：左折叠和右折叠是需要注意的，他们的效果可能不同。

其实按照以上示例效果 `(4-(5-6))` `((4-5)-6)` 还可以总结一段简单的话：***右折叠就是先算右边，左折叠就是先算左边。***

我知道你肯定有疑问了：

> 前面不是说了：“对于逗号运算符，一元左折叠和一元右折叠没有区别”，为啥这里还会有**谁先算**这种说法？

有这个想法就代表思考了，我们来讲一下。

逗号表达式其实也是右折叠先算右边，左折叠先算左边，但是但是，请注意：“`,`”，逗号不同于其他运算符；

比如 `(expr1,(expr2,expr3))` 和 `((expr1,expr2),expr3)` 因为逗号表达式的特性，从左往右顺序执行，逗号运算符本身不需要做什么运算，那么这些括号根本不影响什么，但是如果换成其他的运算符，比如 `-`，就不同了，`(expr1-(expr2-expr3))` 和 `((expr1-expr2)-expr3)` 显然不同，也就是前面的：`(4-(5-6))` `((4-5)-6)`。

“先算”这个词，不是各位想象的那种，一定要先进行运算产生结果，也可能不会有什么运算，比如逗号表达式；这个先算其实指代的是折叠表达式语法给加的括号。但是这个表达式到底是怎么样的运算，是要根据运算符的。

到此，我们讲明白了为什么逗号表达式一元左右折叠效果一样，有些情况效果不一样；以及一元折叠的内容。

> 补充：各位要明白一件事情，我们说的“逗号特殊”，不是它真的特殊，对于折叠表达式规则而言，这些运算符都是一样的处理的，根据你自己运算符的行为，彼此之间毫无区别。我为什么要说“逗号特殊”？这是一个抽象的指代，**指代的是大家看到可能不懂，奇怪，觉得特殊，而不是真的特殊。**`+` 运算符也是左折叠和右折叠效果一样，但是这个大家都懂，我就没提。另外上面提到的所谓的“逗号运算符本身什么都不做”是没有考虑你重载它的，这个大家知道就行。

二元折叠

```cpp
template<typename... Args>
void print(Args&&... args){
    (std::cout << ... << args) << '\n';
}
print("luse", 1, 1.2); // luse11.2
```

又是我们的老朋友 `print` 函数，不过这次，它又换了形式，这是一个**二元左折叠。**

判断一个折叠表达式是否是二元的，只需要看一点：`运算符 ... 运算符` 这种形式就是二元。

判断是左还是右，我们前面已经提了：

> - ***折叠表达式是左折叠还是右折叠，取决于 `...` 是在"形参包"的左边还是右边。***

根据语法套一下 `( 初值 运算符 ... 运算符 形参包 )`，我们给出的示例 print 中 `std::cout` 就是初值、 `运算符 ... 运算符` 就是 `<< ... <<` 、形参包就是 `args`。

```
// 二元右折叠
template<int...I>
constexpr int v = (I + ... + 10);    // 1 + (2 + (3 + (4 + 10)))
// 二元左折叠
template<int...I>
constexpr int v2 = (10 + ... + I);   // (((10 + 1) + 2) + 3) + 4

std::cout << v<1, 2, 3, 4> << '\n';  // 20
std::cout << v2<1, 2, 3, 4> << '\n'; // 20
```

其实和一元折叠中说的差不多不是吗?

> **右折叠就是先算右边，左折叠就是先算左边。**

不过二元折叠表达式必然有一个"**初值**"（我们这里是 `10` ），是先计算的。

总结

省略了一些，但这一节整体我写的较为满意，规则和重点都聊的很清楚，其他的任何形式，无非都是以上的去雕花罢了。

我们可以布置一个课后作业，说出以下代码使用的折叠表达式语法，以及它的效果，详细解析，使用 Markdown 语法。

```
template<class ...Args>
auto Reverse(Args&&... args) {
    std::vector<std::common_type_t<Args...>> res{};
    bool tmp{ false };
    (tmp = ... = (res.push_back(args), false));
    return res;
}
```

提交到 `homework` 文件夹中的 `08折叠表达式作业` 文件夹中（如果没有就创建），然后新建文件（命名需要是有意义的），写好后提交 pr。

待决名

本节，我们开始讲待决名

待决名在模板当中无处不在，只要你写模板，一定遇到并且处理过它，即使你可能是第一次听到"待决名"这个名字。

待决名是你学习模板中重要的一个阶段，以此就可以划分，"新手写模板"和"正常写模板"，**我们不但要知道，怎么写，不能怎么写，还要知道，为什么?**

> 在模板（类模板和函数模板）定义中，某些构造的含义可以在不同的实例化间有所不同。特别是，类型和表达式可能会取决于类型模板形参的类型和非类型模板形参的值。

循序渐进，待决名的规则非常繁杂，我们用一个一个示例为你慢慢展示（只讲常见与重要的），从简单到困难。

待决名的 `typename` 消除歧义符

> **在模板（包括别名模版）的声明或定义中，不是当前实例化的成员且取决于某个模板形参的名字不会被认为是类型，除非使用关键词 typename 或它已经被设立为类型名（例如用 typedef 声明或通过用作基类名）。**

先用一个示例引入问题。

```
template<typename T>
const T::type& f(const T&) {
    return 0;
}

struct X{
    using type = int;
};

int main(){
    X x;
    f(x);
}
```

以上代码会产生编译错误。

msvc 报错提示：

```
error C2061: 语法错误: 标识符“type”
error C2143: 语法错误: 缺少“;”(在“{”的前面)
error C2447: “{”: 缺少函数标题(是否是老式的形式表?)
error C2065: “x”: 未声明的标识符
error C3861: “f”: 找不到标识符
```

gcc：

```
<source>:2:7: error: need 'typename' before 'T::type' because 'T' is a dependent
scope
    2 | const T::type& f(const T&) {
      |       ^
      |       typename
<source>: In function 'int main()':
<source>:12:5: error: 'f' was not declared in this scope
   12 |     f(x);
      |     ^
```

总的意思也很简单，**编译器不觉得你这个 `type` 是一个类型。**

我知道此时，很多人会想到使用一个关键字：`typename`。

我们只需要在 `T::type&` 前面加上 `typename` 就能够通过编译。

```
template<typename T>
const typename T::type& f(const T&) {
    return 0;
}
```

我们使用这个函数模板，来套一下，一句一句分析我们最开始说的那些概念。

> 在模板（包括别名模版）的声明或定义中

我们函数模板 `f` 自然是在模板中，符合。

> 不是当前实例化的成员且取决于某个模板形参的名字

我们的 `T::type` 的确不是当前实例化的成员，当前实例化的是函数模板 `f`；`T::type` 的确是取决于我们的模板形参的名字，简单的说就是 `T::type` 是什么，取决于当前的函数模板。符合。

> 不会被认为是类型

是的，所以我们前面没有使用 `typename` 产生了编译错误。

> 除非使用关键词 typename 或它已经被设立为类型名（例如用 typedef 声明或通过用作基类名）

是的，我们后面的示例使用了 `typename` 就没有问题了，`T::type` 被认为是类型了。

```
int p = 1;

template<typename T>
void foo(const std::vector<T>& v){
    // std::vector<T>::const_iterator 是待决名，
    typename std::vector<T>::const_iterator it = v.begin();

    // 下列内容因为没有 'typename' 而会被解析成
    // 类型待决的成员变量 'const_iterator' 和某变量 'p' 的乘法。
    // 因为在此处有一个可见的全局 'p'，所以此模板定义能编译。
    std::vector<T>::const_iterator* p;

    typedef typename std::vector<T>::const_iterator iter_t;
    iter_t* p2; // iter_t 是待决名，但已知它是类型名
}

int main(){
    std::vector<int>v;
    foo(v); // 实例化失败
}
```

这个就不逐句解释了，就说一下最后一句注释：

iter_t 是待决名，但已知它是类型名

> 除非使用关键词 typename 或它**已经被设立为类型名**（例如用 typedef 声明或通过用作基类名）

值得一提的是，只有在添加 `foo(v)`，即进行模板实例化后 gcc/clang 才会拒绝该程序；
如果你测试过 msvc 的话，会注意到，`typedef typename std::vector<T>::const_iterator iter_t;` 这一句，即使不加 `typename` 一样可以通过编译；

msvc 搞特殊，我们知道就行；不要只测 msvc，不然代码不可跨平台。

> 关键词 typename 只能以这种方式用于限定名（例如 T::x）之前，但这些名字**不必待决**。

`v` 不是待决名，但是的确可以以 `typename` 修饰，虽然没啥用处。

```
typename std::vector<int>::value_type v;
```

到此，`typename` 待决名消除歧义符，我们已经讲清楚了所有的概念，其他各种使用无非都是从这些概念上的，不会有什么特殊。

待决名的 `template` 消除歧义符

> **与此相似，模板定义中不是当前实例化的成员的待决名同样不被认为是模板名，除非使用消歧义关键词 template，或它已被设立为模板名：**

```
template<typename T>
struct S{
    template<typename U>
    void foo() {}
};

template<typename T>
void bar(){
    S<T> s;
    s.foo<T>();          // 错误：< 被解析为小于运算符
    s.template foo<T>(); // OK
}
```

使用 `template` 消除歧义更加少见一点，不过我们一句一句分析就行。

> 模板定义中不是当前实例化的成员的待决名同样不被认为是模板名

`s.foo<T>()` 的确是在模板定义，并且不是当前实例化的成员，它只是依赖了当前的模板实参 `T`，所以不被认为是模板名。

> 除非使用消歧义关键词 template

`s.template foo<T>()` 的确。

注意：`s.foo<T>()` 在 msvc 可以被解析，通过编译，这是非标准的，知道即可。

关键词 `template` 只能以这种方式用于运算符 `::`（作用域解析）、`->`（通过指针的成员访问）和 `.`（成员访问）之后，下列表达式都是合法示例：

- `T::template foo<X>();`
- `s.template foo<X>();`
- `this->template foo<X>();`
- `typename T::template iterator<int>::value_type v;`

与 typename 的情况一样，即使名字并非待决或它的使用并未在模板的作用域中出现，也允许使用 template 前缀。

```
struct X{
    template<typename T>
    void f()const {}
};
struct C{
    using Ctype = int;
};

X x;
x.template f<void>();
C::template Ctype I;
```

没有作用，但是合法。

重复强调一下：`template` **的使用比** `typename` **少，并且** `template` **只能用于** `::`、`->`、`.` **三个运算符** ***之后。***

绑定规则

> **对待决名和非待决名的名字查找和绑定有所不同。**
>
> **非待决名在模板定义点查找并绑定。即使在模板实例化点有更好的匹配，也保持此绑定**

名字查找有非常复杂的规则，尤其还和待决名掺杂在一起，但是我们却不得不讲。

```
#include <iostream>

void g(double) { std::cout << "g(double)\n"; }

template<class T>
struct S{
    void f() const{
        g(1); // "g" 是非待决名，现在绑定
    }
};

void g(int) { std::cout << "g(int)\n"; }

int main(){
    g(1);  // 调用 g(int)

    S<int> s;
    s.f(); // 调用 g(double)
}
```

`s.f()` 中调用的是 `g(1);` 按照一般直觉会选择到 `void g(int)`，但是实际却不是如此，它调用了 `g(double)`。

> 非待决名在模板定义点查找并绑定。即使在模板实例化点有更好的匹配，也保持此绑定

查找规则

我们用 **loser homework** 第九题，引入我们的问题。

```
#include<iostream>
```

```
template<class T>
struct X {
    void f()const { std::cout << "X\n"; }
};

void f() { std::cout << "全局\n"; }

template<class T>
struct Y : X<T> {
    void t()const {
        this->f();
    }
    void t2()const {
        f();
    }
};

int main() {
    Y<void>y;
    y.t();
    y.t2();
}
```

以上代码的运行结果是：

```
X
全局
```

名字查找分为：**有限定**名字查找，**无限定**名字查找。

> 有限定名字查找指？
>
> 出现在作用域解析操作符 `::` 右边的名字是限定名（参阅有限定的标识符）。 限定名可能代表的是：
>
> - 类的成员（包括静态和非静态函数、类型和模板等）
> - 命名空间的成员（包括其他的命名空间）
> - 枚举项

如果 `::` 左边为空，那么查找过程只会考虑全局命名空间作用域中作出（或通过 using 声明引入到全局命名空间中）的声明。

```
this->f();
```

那么显然，这个表达式**不是有限定名字查找**，那么我们就去无限定名字查找中寻找答案。

我们找到**模板定义**：

> 对于在模板的定义中所使用的**非待决名**，当**检查该模板的定义时将进行无限定的名字查找**。在这个位置与声明之间的绑定并不会受到在实例化点可见的声明的影响。而对于在模板定义中所使用的**待决名**，**它的查找会推迟到得知它的模板实参之时**。此时，ADL 将同时在模板的定义语境和在模板的实例化语境中检查可见的具有外部连接的 (C++11 前)函数声明，而非 ADL 的查找只会检查在模板的定义语境中可见的具有外部连接的 (C++11 前)函数声明。（换句话说，在模板定义之后添加新的

> 函数声明，除非通过 ADL 否则仍是不可见的。）如果在 ADL 查找所检查的命名空间中，在某个别的翻译单元中声明了一个具有外部连接的更好的匹配声明，或者如果当同样检查这些翻译单元时其查找会导致歧义，那么行为未定义。无论哪种情况，**如果某个基类取决于某个模板形参，那么无限定名字查找不会检查它的作用域（在定义点和实例化点都不会）**。

很长，但是看我们加粗的就够：

- 非待决名：检查该模板的定义时将进行无限定的名字查找
- 待决名：它的查找会推迟到得知它的模板实参之时

我们这里简单描述一下：

`this->f()` 是一个待决名，这个 `this` 依赖于模板 `X`。

所以，我们的问题可以解决了吗？

1. `this->f()` **是待决名**，所以它的查找会推迟到得知它模板实参之时（届时可以确定父类是否有 f 函数）。
2. `f()` **是非待决名**，检查该模板的定义时将进行无限定的名字查找（无法查找父类的定义），按照正常的查看顺序，先类内（查找不到），然后全局（找到）。

> 补充：如果是 `msvc` 的某些早期版本，或者 C++ 版本设置在 C++20 之前，会打印 `X` `X`。这是因为 `msvc` 不支持**二阶段名字查找** `Two-phase name lookup`。

总结

我们省略了很多的规则，这很正常，着重聊了几个重点，这足够各位的使用，如果还有需求，查阅文档。

SFINAE

“代换失败不是错误” (Substitution Failure Is Not An Error)

在**函数模板的重载决议**[1]中会应用此规则：当模板形参在替换成显式指定的类型或推导出的类型失败时，从重载集中丢弃这个特化，*而非导致编译失败*。

此特性被用于模板元编程。

> 注意：**本节非常非常的重要，是模板基础中的基础，最为基本的特性和概念。**

解释

对函数模板形参进行两次代换（由模板实参所替代）：

- 在模板实参推导前，对显式指定的模板实参进行代换
- 在模板实参推导后，对推导出的实参和从默认项获得的实参进行替换

代换的实参写出时非良构[2]（并带有必要的诊断）的任何场合，都是*代换失败*。

> “对显式指定的模板实参进行代换”这里的显式指定，就比如 `f<int>()` 就是显式指明了。我知道你肯定有疑问：我都显式指明了，那下面还推导啥？对，如果模板函数 `f` 只有一个模板形参，而你显式指明了，的确第二次代换没用，因为根本没啥好推导的。

> 两次代换都有作用，是在于有多个模板形参，显式指定一些，又根据传入参数推导一些。

代换失败与硬错误

> **只有在函数类型或其模板形参类型或其 explicit 说明符 (C++20 起)的*立即语境*中的类型与表达式中的失败，才是 *SFINAE 错误*。如果对代换后的类型/表达式的*求值导致副作用*，例如实例化某模板特化、生成某隐式定义的成员函数等，那么这些副作用中的错误都被当做*硬错误*。**

> 代换失败就是指 SFINAE 错误。

以上概念中注意关键词“SFINAE 错误”、“硬错误”，这些解释不用在意，先看完以下示例再去看概念理解。

```
#include <iostream>

template<typename A>
struct B { using type = typename A::type; }; // 待决名，C++20 之前必须使用 typename 消除歧义

template<
    class T,
    class U = typename T::type,              // 如果 T 没有成员 type 那么就是 SFINAE 失败（代换失败）
    class V = typename B<T>::type>           // 如果 T 没有成员 type 那么就是硬错误 不过标准保证这里不会发生硬错误，因为到 U 的默认模板实参中的代换会首先失败
void foo(int) { std::puts("SFINAE T::type B<T>::type"); }

template<typename T>
void foo(double) { std::puts("SFINAE T"); }

int main(){
    struct C { using type = int; };

    foo<B<C>>(1);       // void foo(int)    输出: SFINAE T::type B<T>::type
    foo<void>(1);       // void foo(double) 输出: SFINAE T
}
```

全平台测试通过。

以上的示例很好的向我们展示了 SFINAE 的作用，可以影响重载决议。

`foo<B<C>>(1)`、`foo<void>(1)` 如果根据一般直觉，他们都会选择到 `void foo(int)`，然而实际却不是如此；

这是因为 `foo<void>(1);` 去尝试匹配 `void foo(int)` 的时候，模板实参类型 `void` 进行替换，就会变成：

```
template<
    class void,
    class U = typename void::type,          // SFINAE 失败
    class V = typename B<void>::type>       // 不会发生硬错误，因为 U 的代换已经失败
```

`void::type` 这一看就是非良构[2]，根据前面提到的：

> 代换的实参写出时非良构（并带有必要的诊断）的任何场合，都是代换失败。

所以这是一个代换失败，但是因为“*代换失败不是错误*”，只是从“*重载集中丢弃这个特化，而不会导致编译失败*”，然后就就去尝试匹配 `void foo(double)` 了，`1` 是 int 类型，隐式转换到 double，没什么问题。

至于其中提到的*硬错误*？为啥它是硬错误？其实最开始的概念已经说了：

> 如果对代换后的类型/表达式的求值导致副作用，例如实例化某模板特化、生成某隐式定义的成员函数等，那么这些副作用中的错误都被当做硬错误。

`B<T>` 显然是对代换后的类型求值导致了副作用，实例化了模板，实例化失败自然被当做硬错误。

> 注意，你应当关注 `B<T>` 而非 `B<T>::type`，因为是直接在实例化模板 B 的时候就失败了，被当成硬错误；如果 `B<T>` 实例化成功，而没有 `::type`，则被当成**代换失败**（不过这里是不可能）。

这节内容非常重要，提到的概念和代码需要全部掌握，后面的内容其实无非都是以本节为基础的变种、各种使用示例、利用标准库的设施让写法简单一点，但是根本的原理，就是本节讲的。

基础使用示例

请在完全读懂上一节内容再阅读本节内容。

C++ 的模板，很多时候就像拼图一样，我们带入进去想，很多问题即使没有阅读规则，也可以无师自通，猜出来。

> ***我需要写一个函数模板 `add`，想要要求传入的对象必须是支持 `operator+` 的，应该怎么写？***

利用 SFINAE 我们能轻松完成这个需求。

```
template<typename T>
auto add(const T& t1, const T& t2) -> decltype(t1 + t2){   // C++11 后置返回类型，在返回类型中运用 SFINAE
    std::puts("SFINAE +");
    return t1 + t2;
}
```

我知道你一定会有疑问

> 1. 这样有啥好处吗？使用了 SFINAE 看起来还变复杂了。我就算不用这写法，如果对象没有 `operator+` 不是一样会编译错误吗？
> 2. 虽然前面说了 SFINAE 可以影响重载决议，我知道这个很有用，但是我这个函数根本没有别的重载，这样写还是有必要的吗？

这两个问题其实是一个问题，本质上就是还是不够懂 SFINAE 或者说模板：

- 如果就是简单写一个 `add` 函数模板不使用 SFINAE，那么编译器在编译的时候，会尝试模板实例化，生成函数定义，发现你这类型根本没有 `operator+`，于是实例化模板错误。
- 如果按照我们上面的写法使用 SFINAE，根据“*代换失败不是错误*”的规则，从重载集中丢弃这个特化 `add`，然而又没有其他的 `add` 重载，所以这里的错误是“**未找到匹配的重载函数**”。

这里的重点是什么？**是模板实例化，能不要实例化就不要实例化**，我们当前的示例只是因为 `add` 函数模板非常的简单，即使实例化错误，编译器依然可以很轻松的报错告诉你，是因为没有 `operator+`。但是很多模板是非常复杂的，编译器实例化模板经常会产生一些完全不可读的报错；如果我们使用 SFINAE，编译器就是直接告诉我：“未找到匹配的重载函数”，我们自然知道就是传入的参数没有满足要求。而且实

例化模板也是有开销的，很多时候甚至很大。

总而言之：
即使不为了处理重载，使用 SFINAE 约束函数模板的传入类型，也是有很大好处的：报错、编译速度。

但是令人诟病的是 SFINAE 的写法在很多时候非常麻烦，目前各位可能还是没有感觉，后面的需求，写出的示例，慢慢的你就会感觉到了。这些问题会在下一章的约束与概念解决。

标准库支持

标准库提供了一些设施帮助我们更好的使用 SFINAE。

`std::enable_if`

```
template<bool B, class T = void>
struct enable_if {};

template<class T> // 类模板偏特化
struct enable_if<true, T> { typedef T type; };      // 只有 B 为 true，才有 type，即 ::type 才合法

template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type; // C++14 引入
```

这是一个模板类，在 C++11 引入，它的用法很简单，就是第一个模板参数为 true，此模板类就有 `type`，不然就没有，以此进行 SFINAE。

```
template<typename T,typename SFINAE =
    std::enable_if_t<std::is_same_v<T,int>>>
void f(T){}
```

函数 `f` 要求 `T` 类型必须是 `int` 类型；我们一步一步分析

`std::enable_if_t<std::is_same_v<T,int>>` 如果 T 不是 int，那么 std::is_same_v 就会返回一个 false，也就是说 `std::enable_if_t<false>` ，带入：

```
using enable_if_t = typename enable_if<false,void>::type; // void 是默认模板实参
```

但是问题在于：

- **enable_if 如果第一个模板参数为 `false`，它根本没有 `type` 成员。**

所以这里是个**代换失败**，但是因为“*代换失败不是错误*”，所以只是不选择函数模板 `f`，而不会导致编译错误。

再谈，std::enable_if 的默认模板实参是 `void`，如果我们不在乎 std::enable_if 得到的类型，就让它默认就行，比如我们的示例 `f` 根本不在乎第二个模板形参 `SFINAE` 是啥类型。

```
template <class Type, class... Args>
array(Type, Args...) -> array<std::enable_if_t<(std::is_same_v<Type, Args> &&
...), Type>, sizeof...(Args) + 1>;
```

以上示例，是显式指明了 std::enable_if 的第二个模板实参，为 `Type`。

它是我们类模板推导指引那一节的示例的**改进版本**，我们使用 std::enable_if_t 与 C++17 折叠表达式，为它增加了约束，这几乎和 libstdc++ 中的代码一样。

`(std::is_same_v<Type, Args> && ...)` 做 std::enable_if 的第一个模板实参，这里是一个一元右折叠，使用了 `&&` 运算符，也就是必须 std::is_same_v 全部为 true，才会是 true。简单的说就是要求类型形参包 Args 中的每一个类型全部都是一样的，不然就是替换失败。

这样做有很多好处，老式写法存在很多问题：

```
template<class Ty, std::size_t size>
struct array {
    Ty arr[size];
};

template<typename T, typename ...Args>
array(T t, Args...) -> array<T, sizeof...(Args) + 1>;

::array arr{1.4, 2, 3, 4, 5};        // 被推导为 array<double,5>
::array arr2{1, 2.3, 3.4, 4.5, 5.6}; // 被推导为 array<int,5>     有数据截断
```

如果不使用 SFINAE 约束，那么 array 的类型完全取决于第一个参数的类型，很容易导致其他问题。

`std::void_t`

```
template< class... >
using void_t = void;
```

如你所见，它的实现非常非常的简单，就是一个别名，接受任意个数的类型参数，但自身始终是 `void` 类型。

- 将任意类型的序列映射到类型 void 的工具元函数。
- 模板元编程中，用此元函数检测 SFINAE 语境中的非良构类型[2]。

> *我要写一个函数模板 `add`，我要求传入的对象需要支持 `+` 以及它需要有别名 `type`，成员 `value`、`f`。*

```
#include <iostream>
#include <type_traits>

template<typename T,
    typename SFINAE = std::void_t<
    decltype(T{} + T{}), typename T::type, decltype(&T::value), decltype(&T::f)
>>
auto add(const T& t1, const T& t2) {
    std::puts("SFINAE + | typename T::type | T::value");
    return t1 + t2;
}

struct Test {
    int operator+(const Test& t)const {
        return this->value + t.value;
    }
```

```
    void f()const{}
    using type = void;
    int value;
};

int main() {
    Test t{ 1 }, t2{ 2 };
    add(t, t2);  // OK
    //add(1, 2); // 未找到匹配的重载函数
}
```

- `decltype(T{} + T{})` 用 decltype 套起来只是为了获得类型符合语法罢了，std::void_t 只接受类型参数。如果类型没有 `operator+`，自然是*代换失败*。
- `typename T::type` 使用 `typename` 是因为待决名；type 本身是类型，不需要 decltype。如果 `add` 推导的类型没有 `type` 别名，自然是*代换失败*。
- `decltype(&T::value)` 用 decltype 套就不用说了，`&T::value` 是成员指针的语法，不区分是数据成员还是成员函数，如果有这个成员 `value`，`&类名::成员名字` 自然合法，要是没有，就是*代换失败*。
- `decltype(&T::f)`，其实前面已经说了，成员函数是没区别的，没有成员 `f` 就是 *代换失败*。

总而言之，这是为了使用 SFINAE。

> 那么这里 `std::void_t` 的作用是?

其实倒也没啥，无非就是给了个好的语境，让我们能这样写，最终 `typename SFINAE = std::void_t` 这里的 `SFINAE` 的类型就是 `void`；当然了，这不重要，重要的是创造这样写的语境，能够方便我们进行 `SFINAE`。

仅此一个示例，我相信就足够展示 `std::void_t` 的使用了。

> *那么如果在 C++17 标准之前，没有 std::void_t，我该如何要求类型有某些成员呢?*

其实形式和原理都是一样的。

```
template<typename T,typename SFINAE = decltype(&T::f)>
void f(T){}

struct Test {
    void f()const{}
};

Test t;
f(t);  // OK
f(1);  // 未找到匹配的重载函数
```

C++11 可用。

`std::declval`

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

将任意类型 T 转换成引用类型，*使得在 decltype 说明符的操作数中不必经过构造函数就能使用成员函数*。

- std::declval 只能用于 **不求值语境**，且不要求有定义。
- **它不能被实际调用，因此不会返回值，返回类型是 `T&&`。**

它常用于模板元编程 SFINAE，我们用一个示例展现它的必要性：

```
template<typename T, typename SFINAE = std::void_t<decltype(T{} + T{})> >
auto add(const T& t1, const T& t2) {
    std::puts("SFINAE +");
    return t1 + t2;
}

struct X{
    int operator+(const X&)const{
        return 0;
    }
};

struct X2 {
    X2(int){}   // 有参构造，没有默认构造函数
    int operator+(const X2&)const {
        return 0;
    }
};

int main(){
    X x1, x2;
    add(x1, x2);          // OK

    X2 x3{ 0 }, x4{ 0 };
    add(x3,x4);           // 未找到匹配的重载函数
}
```

错误的原因很简单，`decltype(T{} + T{})` 这个表达式中，同时**要求了 `T` 类型支持默认构造**（虽然这不是我们的本意），然而我们的 `X2` 类型没有默认构造，自然而然 `T{}` 不是合法表达式，*代换失败*。其实我们之前也有类似的写法，我们在本节进行纠正，使用 `std::declval`：

```
template<typename T, typename SFINAE = std::void_t<decltype(std::declval<T>() +
std::declval<T>())> >
auto add(const T& t1, const T& t2) {
    std::puts("SFINAE +");
    return t1 + t2;
}
```

测试。

把 `T{}` 改成 `std::declval<T>()` 即可，decltype 是不求值语境，没有问题。

部分（偏）特化中的 SFINAE

在确定一个类或变量 (C++14 起)模板的特化是由部分特化还是主模板生成的时候也会出现推导与替换。在这种确定期间，**部分特化的替换失败不会被当作硬错误，而是像函数模板一样*代换失败不是错误*，只是忽略这个部分特化。**

```
#include <iostream>

template<typename T,typename T2 = void>
struct X{
    static void f() { std::puts("主模板"); }
};

template<typename T>
struct X<T, std::void_t<typename T::type>>{
    using type = typename T::type;
    static void f() { std::puts("偏特化 T::type"); }
};

struct Test { using type = int; };
struct Test2 { };

int main(){
    X<Test>::f();       // 偏特化 T::type
    X<Test2>::f();      // 主模板
}
```

总结

到此，其实就足够了，SFINAE 的原理、使用、标准库支持（std::enable_if、std::void_t、std::declval）。

虽然称不上全部，但如果你能完全理解明白本节的所有内容，那你一定超越了至少 95% C++ 开发者。其他的各种形式无非都是这样类似的，因为我们已经为你讲清楚了 ***原理。***

- ***代换失败不是错误。***

约束与概念

类模板，函数模板，以及非模板函数（通常是类模板的成员），可以与一项约束（constraint）相关联，它指定了对模板实参的一些要求，这些要求可以被用于选择最恰当的函数重载和模板特化。

这种要求的具名集合被称为概念（concept）。每个概念都是一个谓词，它在编译时求值，并在将之用作约束时成为模板接口的一部分。

前言

在 C++20 引入了约束与概念，这一核心语言特性是所有使用模板的 C++ 开发者都期待的。

有了它，我们的模板可以有更多的静态检查，语法更加美观，写法更加容易，而不再需要利用古老的 **SFINAE。**

请务必学习完了上一章节内容；本节会一边为你教学约束与概念的语法，一边用 SFINAE 对比，让你意识到：***这是多么先进、简单的核心语言特性。***

定义*概念*（concept）与使用

```
template < 模板形参列表 >
concept 概念名 属性(可选) = 约束表达式;
```

定义概念所需要的 *约束表达式*，只需要是可以在编译期产生 `bool` 值的表达式即可。

> - 你可以先不看基本概念，关注我们的示例和下面的讲解。

> ***我需要写一个函数模板 `add`，想要要求传入的对象必须是支持 `operator+` 的，应该怎么写?***

此需求就是 `SFINAE` 中提到的，我们使用*概念*（concept）来完成。

```
template<typename T>
concept Add = requires(T a) {
    a + a; // "需要表达式 a+a 是可以通过编译的有效表达式"
};

template<Add T>
auto add(const T& t1, const T& t2){
    std::puts("concept +");
    return t1 + t2;
}
```

我们使用关键字 `concept` 定义了一个*概念*（concept），命名为 `Add`，它的*约束*是 `requires(T a) { a + a; }` 即要求 `f(T a)`、`a + a` 是合法表达式。

```
template<Add T> // T 被 Add 约束
```

语法上就是把原本的 `typename`、`class` 换成了我们定义的 `Add` *概念*（concept），语义和作用也非常的明确：

- **就是让这个概念约束模板类型形参 `T`，即要求 `T` 必须满足*约束表达式*的*要求序列* `T a` `a + a`。**
  如果不满足，则不会选择这个模板。

> "满足"：要求带入后必须是合法表达式；

最开始的概念已经说了：

> *概念*（concept）可以与一项约束（constraint）相关联，它指定了对模板实参的一些要求，这些要求可以被用于选择最恰当的函数重载和模板特化。

另外最开始的概念中还说过：

> 每个概念都是一个**谓词**，它在**编译时求值**，并在将之用作约束时成为模板接口的一部分。

也就是说我们其实可以这样：

```
std::cout << std::boolalpha << Add<int> << '\n';        // true
std::cout << std::boolalpha << Add<char[10]> << '\n';   // false
constexpr bool r = Add<int>;                            // true
```

我相信这非常的好理解，这些语法形式，合理且简单。

记得我们在第一章节函数模板中提到的："C++20 简写函数模板"吗？

```
decltype(auto) max(const auto& a, const auto& b)  { // const T&
    return a > b ? a : b;
}
```

这段代码来自函数模板那一章节。

> **我想要约束：传入的对象 a b 必须都是整数类型，应该怎么写？**。

```
#include <concepts> // C++20 概念库标头

decltype(auto) max(const std::integral auto& a, const std::integral auto& b) {
    return a > b ? a : b;
}

max(1, 2);     // OK
max('1', '2'); // OK
max(1u, 2u);   // OK
max(1l, 2l);   // OK
max(1.0, 2);   // Error! 未满足关联约束
```

如你所见，我们没有自己定义 *概念*（concept），而是使用了标准库的 `std::integral`，它的实现非常简单：

```
template< class T >
concept integral = std::is_integral_v<T>;
```

这也告诉各位我们一件事情：**定义*概念*（concept）** 时声明的约束表达式，只需要是编译期可得到 `bool` 类型的表达式即可。

> 我相信你这里一定有疑问："那么我们之前写的 requires 表达式呢？它会返回 `bool` 值吗？"对，简单的说，把模板参数带入到 `requires` 表达式中，是否符合语法，符合就返回 `true`，不符合就返回 `false`。在 `requires` 表达式 一节中会详细讲解。

它的实现是直接使用了标准库的 `std::is_integral_v<T>`，非常简单。

再谈*概念*（concept）在简写函数模板中的写法 `const std::integral auto& a`，*概念*（concept）只需要写在 `auto` 之前即可，表示此概念约束 `auto` 推导的类型必须为整数类型，语义十分明确，像是 cv 限定、引用等，不需要在乎，或许我们可以先写的简单一点先去掉那些：

```
decltype(auto) max(std::integral auto a, std::integral auto b) {
    return a > b ? a : b;
}
```

这是否直观了很多？并且概念不单单是可以用作简写函数模板中的 `auto`，还有几乎一切语境，比如：

```
int f() { return 0; }

std::integral auto result = f();
```

还是那句话，语义很明确：

- ***概念*（concept）约束了 `auto`，它必须被推导为整数类型；如果函数 `f()` 返回类型是 `double` `auto` 无法推导为整数类型，那么编译器会报错：“*未满足关联约束*”。**

类模板同理，如：

```
template<typename T>
concept add = requires(T t){  // 定义概念，通常推荐首字母小写
    t + t;
};

template<add T>
struct X{
    T t;
};
```

变量模板也同理

```
template<typename T>
concept add = requires(T t){
    t + t;
};

template<add T>
T t;

t<int>;     // OK
t<char[1]>; // “t”未满足关联约束
```

将模板中的 `typename`、`class` 换成 *概念*（concept）即可，表示约束此模板类型形参 `T`。

`requires` 子句

关键词 requires 用来引入 requires 子句，它指定对各模板实参，或对函数声明的约束。

也就是说我们多了一种使用*概念*（concept）或者说约束的写法。

```
template<typename T>
concept add = requires(T t) {
    t + t;
};

template<typename T>
    requires std::is_same_v<T, int>
void f(T){}

template<typename T> requires add<T>
void f2(T) {}

template<typename T>
void f3(T)requires requires(T t) { t + t; }
{}
```

> `requires` 子句期待一个能够编译期产生 `bool` 值的表达式。

以上示例展示了 `requires` 子句的用法，我们一个个解释

1. `f` 的 `requires` 子句写在 `template` 之后，并空四格，这是我个人推荐的写法；它的约束是：`std::is_same_v<T, int>`，意思很明确，约束 `T` 必须是 int 类型，就这么简单。
2. `f2` 的 `requires` 子句写法和 `f` 其实是一样的，只是没换行和空格；它使用了我们自定义的*概念*（concept）`add`，约束 `T` 必须满足 `add`。
3. `f3` 的 `requires` 子句在函数声明符的末尾元素出现；这里我们连用了两个 `requires`，为什么？其实很简单，我们要区分，第一个 `requires` 是 *`requires` 子句*，第二个 `requires` 是*约束表达式*，它会产生一个编译期的 `bool` 值，没有问题。（如果 `T` 类型带入*约束表达式*是良构，那么就返回 `true`、否则就返回 `false`）。

> 类模板、变量模板等也都同理

requires 子句中，**关键词 requires 必须后随某个常量表达式。**

```
template<typename T>
    requires true
void f(T){}
```

完全可行，各位其实可以直接带入，说白了 `requires` 子句引入的约束表，必须是可以编译期返回 `bool` 类型的值的表达式，我们前面的三个例子：`std::is_same_v`、`add`、`requires 表达式` 都如此。

约束

前面我们讲的都是非常基础的*概念*（concept）使用，他们的约束也都十分简单，本节我们详细讲一下。

约束是逻辑操作和操作数的序列，它指定了对模板实参的要求。它们可以在 requires 表达式（见下文）中出现，也可以直接作为概念的主体。

有三种类型的约束：

1. 合取（conjunction）
2. 析取（disjunction）

合取

两个约束的合取是通过在约束表达式中使用 && 运算符来构成的：

```
template<class T>
concept Integral = std::is_integral_v<T>;
template<class T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;
template<class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

很合理，**约束表达式**可以使用 `&&` 运算符连接两个约束，只有在两个约束都被满足时才会得到满足

我们先定义了一个 *概念*（concept）Integral，此概念要求整形；又定义了*概念*（concept）SignedIntegral，它的约束表达式用到了先前定义的*概念*（concept）Integral，然后又加上了一个 `&&` 还需要满足 std::is_signed_v。

概念（concept）`SignedIntegral` 是要求有符号整数类型，它的*约束表达式*是：`Integral<T> && std::is_signed_v<T>`，就是这个表达式要返回 `true` 才成立，就这么简单。

```
void s_f(const SignedIntegral auto&){}
void u_f(const UnsignedIntegral auto&){}

s_f(1);   // OK
s_f(1u);  // 未找到匹配的重载函数
u_f(1);   // 未找到匹配的重载函数
u_f(1u);  // OK
```

> ***两个约束的合取只有在两个约束都被满足时才会得到满足。***合取从左到右短路求值（如果不满足左侧的约束，那么就不会尝试对右侧的约束进行模板实参替换：这样就会防止出现立即语境外的替换所导致的失败）。

```
struct X{
    int c{}; // 无意义，为了扩大 X
    static constexpr bool value = true;
};

template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
    requires (sizeof(T) > 1 && get_value<T>())
void f(T){}

X x;
f(x); // OK
```

析取

两个约束的析取，是通过在约束表达式中使用 || 运算符来构成的：

```
template<typename T>
concept number = std::integral<T> || std::floating_point<T>;
```

和 `||` 运算符本来的意思一样， `std::integral<T>`、`std::floating_point` 满足任意一个，那么整个约束表达式就都得到满足。

```
void f(const number auto&){}

f(1);      // OK
f(1u);     // OK
f(1.2);    // OK
f(1.2f);   // OK
f("1");    // 未找到匹配的重载函数
```

> 如果其中一个约束得到满足，那么两个约束的析取的到满足。析取从左到右短路求值（如果满足左侧约束，***那么就不会尝试对右侧约束进行模板实参替换***）。

```
struct X{
    int c{}; // 无意义，为了扩大 X
    //static constexpr bool value = true;
};
```

```
template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
    requires (sizeof(T) > 1 || get_value<T>())
void f(T){}

X x;
f(x);  // OK 即使 X 根本没有静态 value 成员。
```

如你所见，即使我们的 X 根本不满足右侧约束 `get_value<T>()` 的要求，没有静态 `value` 成员，不过一样可以通过编译。

`requires` 表达式

> **产生描述约束的 bool 类型的纯右值表达式。**

虽然前面聊*概念*（concept）的时候，用到了 `requires` 表达式（定义 concept 的时候），但是没有详细说明，本节我们详细展开说明。

> *注意，`requires` 表达式和 `requires` 子句，没关系。*

```
requires { 要求序列 }
requires ( 形参列表(可选) ) { 要求序列 }
```

要求序列，是以下形式之一：

- 简单要求
- 类型要求
- 复合要求
- 嵌套要求

解释

要求可以援引处于作用域内的模板形参，形参列表中引入的局部形参，以及在上下文中可见的任何其他声明。

- 将模板参数**代换**到模板化实体的声明中所使用的 requires 表达式中，*可能会导致在其要求中形成无效的类型或表达式，或者违反这些要求的语义*。在这种情况下，requires 表达式的值为 `false` 而不会导致程序非良构。
- 按照词法顺序进行**代换**和语义约束检查，当遇到决定 requires 表达式结果的条件时就停止。*如果代换（若存在）和语义约束检查成功*，则 requires 表达式的结果为 `true`。

> **简单的说，把模板参数带入到 requires 表达式中，是否符合语法，符合就返回 `true`，不符合就返回 `false`。**

```
#include <iostream>

template<typename T>
void f(T) {
    constexpr bool v = requires{ T::type; }; // 此处可不使用 typename
    std::cout << std::boolalpha << v << '\n';
```

```
}

struct X { using type = void; };
struct Y { static constexpr int type = 0; };

int main() {
    f(1);   // false 因为 int::type 不是合法表达式
    f(X{}); // false 因为 X::type   在待决名中不被认为是类型，需要添加 typename
    f(Y{}); // true  因为 Y::type   是合法表达式
}
```

> 三端测试。

简单要求

简单要求是任何不以关键词 requires 开始的表达式语句。它断言该表达式是有效的。表达式是不求值的操作数；只检查语言的正确性。

```
template<typename T>
concept Addable = requires (T a, T b) {
    a + b; // "需要表达式 a+b 是可以通过编译的有效表达式"
};

template<class T, class U = T>
concept Swappable = requires(T && t, U && u) {
    swap(std::forward<T>(t), std::forward<U>(u));
    swap(std::forward<U>(u), std::forward<T>(t));
};

template<typename T>
    requires (Addable<T> && Swappable<T, T>)
struct Test{};

namespace loser{
    struct X{
        X operator+(const X&)const{
            return *this;
        }
    };
    void swap(const X&,const X&){}
}

int main() {
    using loser::X;

    Test<X> t2; // OK
    std::cout << std::boolalpha << Addable<X> << '\n';     // true
    std::cout << std::boolalpha << Swappable<X,X> << '\n'; // true
}
```

> 以上代码利用了实参依赖查找（`ADL`），即 `swap(X{})` 是合法表达式，而不需要增加命名空间限定。

以关键词 requires 开始的要求总是被解释为嵌套要求。因此简单要求**不能以没有括号的 requires 表达式开始。**

类型要求

类型要求是关键词 `typename` 后面接一个可以被限定的**类型名称**。该要求是，所指名的类型是有效的。

可以用来验证：

1. 某个指名的嵌套类型是否存在
2. 某个类模板特化是否指名了某个类型
3. 某个别名模板特化是否指名了某个类型。

```
struct Test{
    struct X{};
    using type = int;
};

template<typename T>
struct S{};

template<typename T>
using Ref = T&;

template<typename T>
concept C = requires{
    typename T::X;      // 需要嵌套类型
    typename T::type;   // 需要嵌套类型
    typename S<T>;      // 需要类模板特化
    typename Ref<T>;    // 需要别名模板代换
};

std::cout << std::boolalpha << C<Test> << '\n'; // true
```

稍微解释一下，类 `Test` 有一个嵌套类 `X`，一个别名 `type`，所以 `typename T::X`、`typename T::type` 类型是有效的。

`typename S<T>` 因为有类模板 `S`，且它接受类型模板参数，所以 `typename S<T>` 类型是有效的。假设模板类 `S` 的模板是接受非类型模板参数的，比如 `template<std::size_t> S` ，那么 `typename S<T>` 类型自然不是有效的。

`typename Ref<T>` 因为有别名模板 `Ref`，自然没问题，类型自然是有效的。

> 其实说来说去也很简单，你就直接**带入**，把*概念*（concept）的模板实参（比如 Test）直接带入进去 `requires` 表达式，想想它是不是合法的表达式就可以了。

复合要求

复合要求具有如下形式

```
{ 表达式 } noexcept(可选) 返回类型要求(可选) ;
```

> 返回类型要求：-> 类型约束（*概念* concept）

并断言所指名表达式的属性。替换和语义约束检查按以下顺序进行：

1. 模板实参 (若存在) 被替换到 表达式 中；

2. 如果使用了`noexcept`，表达式 一定不能潜在抛出；
3. 如果*返回类型要求* 存在，则：
    - 模板实参被替换到*返回类型要求* 中；
    - `decltype((表达式))` 必须满足*类型约束* 蕴含的约束。否则，被包含的 requires 表达式是 `false`。

```
template<typename T>
concept C2 = requires(T x){
    // 表达式 *x 必须合法
    // 并且 类型 T::inner 必须存在
    // 并且 *x 的结果必须可以转换为 T::inner
    {*x} -> std::convertible_to<typename T::inner>;

    // 表达式 x + 1 必须合法
    // 并且 std::same_as<decltype((x + 1)), int> 必须满足
    // 即，(x + 1) 必须为 int 类型的纯右值
    {x + 1} -> std::same_as<int>;

    // 表达式 x * 1 必须合法
    // 并且 它的结果必须可以转换为 T
    {x * 1} -> std::convertible_to<T>;

    // 复合："x.~T()" 是不会抛出异常的合法表达式
    { x.~T() } noexcept;
};
```

我们可以写一个满足*概念*（concept）`C2` 的类型：

```
struct X{
    int operator*()const { return 0; }
    int operator+(int)const { return 0; }
    X operator*(int)const { return *this; }
    using inner = int;
};
```

```
std::cout << std::boolalpha << C2<X> << '\n'; // true
```

测试。

析构函数比较特殊，不需要我们显式声明它为 `noexcept` 的，它默认就是 `noexcept` 的。

不管编译器为我们生成的 `X` 析构函数，还是我们用户显式定义的 `X` 析构函数，默认都是有 `noexcept` 的[1]。只有我们用户定义析构函数的时候把它声明为了 `noexcept(false)` 这个析构函数才不是 `noexcept` 的，才会不满足 *概念*（concept） `C2` 的要求。

嵌套要求

嵌套要求具有如下形式

```
requires 约束表达式 ;
```

它可用于根据本地形参指定其他约束。*约束表达式* 必须由被替换的模板实参（若存在）满足。将模板实参替换到嵌套要求中会导致替换到 *约束表达式* 中，但仅限于确定是否满足 *约束表达式* 所需的程度。

```
template<typename T>
concept C3 = requires(T a, std::size_t n) {
    requires std::is_same_v<T*, decltype(&a)>;      // 要求 is_same_v          求值为 true
    requires std::same_as<T*, decltype(new T[n])>; // 要求 same_as            求值为 true
    requires requires{ a + a; };                    // 要求 requires{ a + a; } 求值为 true
    requires sizeof(a) > 4;                         // 要求 sizeof(a) > 4      求值为 true
};
std::cout << std::boolalpha << C3<int> << '\n';    // false
std::cout << std::boolalpha << C3<double> << '\n'; // true
```

> 嵌套要求的 *约束表达式*，只要能编译期产生 `bool` 值的表达式即可，*概念*（concept）、类型特征的库、`requires` 表达式，等都一样。

这里用 `std::is_same_v` 和 `std::same_as` 其实毫无区别，因为它们都是编译时求值，返回 `bool` 值的表达式。

在上面示例中 `requires requires{ a + a; }` 其实是更加麻烦的写法，目的只是为了展示 `requires` 表达式是编译期产生 `bool` 值的表达式，所以有可能会有**两个 `requires` 连用的情况**；我们完全可以直接改成 `a + a`，效果完全一样。

部分（偏）特化中使用*概念*

我们在讲 SFINAE 的时候提到了，它可以用作模板偏特化，帮助我们选择特化版本；本节的约束与概念当然也可以做到，并且写法**更加简单直观优美**：

```
#include <iostream>

template<typename T>
concept have_type = requires{
    typename T::type;
};

template<typename T>
struct X {
    static void f() { std::puts("主模板"); }
};

template<have_type T>
struct X<T> {
    using type = typename T::type;
    static void f() { std::puts("偏特化 T::type"); }
};

struct Test { using type = int; };
struct Test2 { };

int main() {
```

```
    X<Test>::f();       // 偏特化 T::type
    X<Test2>::f();      // 主模板
}
```

这个示例完全是从 SFINAE 的写法改进而来，我们不需要再写第二个模板类型参数，我们直接写作 `template<have_type T>` 就完成了，概念约束了模板类型参数 `T`。

- **只有概念被满足的时候，才会选择到这个偏特化。**

一些实际的用途，比如我以前的 `C++20 STL Cookbook` 中对 `std::formatter` 进行偏特化，也是使用的概念，`std::ranges::range`。

总结

我们先讲述了 *概念*（concept）的定义和使用，其中使用到了 `requires` 表达式，但是我们留到了后面详细讲述。

其实本章内容可以划分为两个部分

- 约束与概念
- `requires` 表达式

如果你耐心看完，我相信也能意识到他们是互相掺杂，一起使用的。语法上虽然感觉有些多，但是也都很合理，我们只需要 ***带入***，按照基本的常识判断这是不是符合语法，基本上就可以了。

`requires` 关键字的用法很多，但是划分的话其实就两类

- `requires` 子句
- `requires` 表达式

`requires` 子句和 `requires` 表达式可以连用，组成 `requires requires` 的形式。我们在 `requires 子句`讲过。

还有在 `requires` 表达式中的嵌套要求，也会有 `requires requires` 的形式。

如果看懂了，这些看似奇怪的 `requires` 关键字复用，其实也都很合理，只需要记住最重要的一句话：

> 可以连用 `requires requires` 的情况，都是因为第一个 `requires` 期待一个可以编译期产生 `bool` 值的表达式；而 `requires` **表达式就是产生描述约束的 bool 类型的纯右值表达式。**

---

1. 注：函数模板自身并不是类型、函数或任何其他实体。不会从只包含模板定义的源文件生成任何代码。模板只有实例化才会有代码出现。 ↩ ↩ ↩ ↩
2. 注：非良构（ill-formed）——程序拥有语法错误或可诊断的语义错误。遵从标准的 C++ 编译器必须为此给出诊断。 ↩ ↩ ↩ ↩
3. 注："重载决议"，简单来说，一个函数被重载，编译器必须决定要调用哪个重载，我们决定调用的是各形参与各实参之间的匹配最紧密的重载。 ↩
4. 注：这个问题可以通过显式实例化解决，在后面会讲。 ↩