深蓝学院
shenlanxueyuan.com

C++ 基础

第 13 章：模板

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

函数模板

- 使用 template 关键字引入模板： template<typename T> void fun(T) {...}
  - 函数模板的声明与定义
  - typename 关键字可以替换为 class ，含义相同
  - 函数模板中包含了两对参数：函数形参 / 实参；模板形参 / 实参
    函数模板不是函数。
- 函数模板的显式实例化： fun<int>(3)
  - 实例化会使得编译器产生相应的函数（函数模板并非函数，不能调用）
  - 编译期的两阶段处理
    - 模板语法检查
    - 模板实例化

    编译器还是从上到下检查。当它看到fun的时候，他先进行一下语法检查（只关注语法本身，简单的检查），
    当我们写fun<int>(3)的时候，他要回过去看模板的定义。进行模板实例化。
    并再次进行检查。
  - 模板必须在实例化时可见—翻译单元的一处定义原则

    因为模板必须在实例化时可见，因此放松了翻译单元的一处定义原则。
  - 注意与内联函数的异同

    inline表示我的函数可能会在一定情况下在调用的地方展开，来形成内联的结构。
    模板：在翻译单元的地方可见.为了保证2阶段处理
- 函数模板的重载

```
//函数模板的声明
template <typename T>
void fun(T);//可以写多次

//函数模板的定义
template <typename T> //尖括号内是模板参数
//我们现在重点讨论的是类型模板参数。我们可以把T
视为一个形式参数，它表明一种类型
void fun(T input)//表示fun要接受一个形式参数，
T是模板形参，input是函数形参。模板形参是在编译期
确定的。如果我们能为模板形参赋予相应的形参，我们就
相当于能把函数模板实例化为一个函数。
{
  std::cout << input;
}

//可以使用class代提typename
template<class T>//也是可以通过编译的
void fun(T input) {
  std::cout << input;
}

int main() {
  fun<int>(3);//int就是模板实参，我们就可以把模板函数
实例化为一个函数。3对应函数实参。
}
```

```
把template中的函数声明为内联函数
template<typename T>
inline void fun(T input) {
  std::cout << input << std::endl;
}
```

```
#include <iostream>

template <typename T>
void fun(T input)
{
    std::cout << input << std::endl;
}

int main()
{
    fun<int>(3);
    fun<double>(3);
}
```

```
#ifdef INSIGHTS_USE_TEMPLATE
template<>
void fun<int>(int input)
{
  std::cout.operator<<(input).operator<<(std::endl);
}
#endif


/* First instantiated from: insights.cpp:12 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
void fun<double>(double input)
{
  std::cout.operator<<(input).operator<<(std::endl);
}
#endif
```

右侧实例化出2个函数.

```
//函数模板重载
template<typename T>
void fun(T input) {
  std::cout << input ;
}

template<typename T, typename T2> //形参列表
void fun(T input, T2 input2) {
  //
}
```

函数模板——续 1

- 模板实参的类型推导（参考文献： https://www.youtube.com/watch?v=wQxj20X-tIU ）
  - 如果函数模板在实例化时没有显式指定模板实参，那么系统会尝试进行推导
  - 推导是基于函数实参（表达式）确定模板实参的过程，其基本原则与 auto 类型推导相似
    - 函数形参是左值引用 / 指针：
      - 忽略表达式类型中的引用
      - 将表达式类型与函数形参模式匹配以确定模板实参
    - 函数形参是万能引用
      - 如果实参表达式是右值，那么模板形参被推导为去掉引用的基本类型
      - 如果实参表达式是左值，那么模板形参被推导为左值引用，触发引用折叠
    - 函数形参不包含引用
      - 忽略表达式类型中的引用
      - 忽略顶层 const
      - 数组、函数转换成相应的指针类型

```
template<T& input>
void fun(T& input) {//函数形参是左值引用
  std::cout << input;
}
int main() {
  int y = 3; int& x = 3;
  fun(y);}
x -> int& -> int
input -> T&
```

前两个都是构造一个别名。但是如果不包含引用的话，本质上是包含一个对象的副本。那么就涉及到一个复制的过程。那么就相对复杂了

```
template<typename T>
void fun(T&& input) { // 万能引用,既可以引用左值，也可以引用右值
  std::cout << " ";
}
int main() {
  fun(3);
}

//如果实参表达式是左值，那么模板形参被推导为左值引用，触发引用折叠
int main() {
  int x = 3;
  fun(3);
}
```

函数模板——续 2

- 模板实参并非总是能够推导得到
  - 如果模板形参与函数形参无关，则无法推导
  - 即使相关，也不一定能进行推导，推导成功也可能存在因歧义而无法使用
- 在无法推导时，编译器会选择使用缺省模板实参
  - 可以为任意位置的模板形参指定缺省模板实参—注意与函数缺省实参的区别
- 显式指定部分模板实参
  - 显式指定的模板实参必须从最左边开始，依次指定
  - 模板形参的声明顺序会影响调用的灵活性
- 函数模板制动推导时会遇到的几种情况
  - 函数形参无法匹配— SFINAE （替换失败并非错误）
  - 模板与非模板同时匹配，匹配等级相同，此时选择非模板的版本
  - 多个模板同时匹配，此时采用偏序关系确定选择“最特殊”的版本

```
//模板与非模板同时匹配，会选择非模板版本
template <typename T, typename T2>
void fun(T x, T2 y) {
  std::cout << "1";
}

void fun(int x, double y) {
  std::cout << 2;
}

int main() {
  fun(3, 5.0); //打印出2
}
```

```
多个模板同时匹配
template <typename T, typename T2>
void fun(T x, T2 y) {
  std::cout << 1;
}

template <typename T>
void fun(T x, float y) {
  std::cout << 2;
}
int main() {
  fun(3, 5.0f);//打印2，选择特殊的版本
}
```

函数模板——续 3

- 函数模板的实例化控制
  - 显式实例化定义： template void fun<int>(int) / template void fun(int)
  - 显式实例化声明： extern template void fun<int>(int) / extern template void fun(int)
  - 注意一处定义原则
  - 注意实例化过程中的模板形参推导
- 函数模板的 (完全) 特化： template<> void f<int>(int) / template<> void f(int)
  - 并不引入新的（同名）名称，只是为某个模板针对特定模板实参提供优化算法
  - 注意与重载的区别
  - 注意特化过程中的模板形参推导

我们希望引入实例化的定义，但是不希望调用它显示实例化定义和下面函数模板 的特化很相似，注意区别

这两种写法都可以

如果没有显示实例化声明，那么只要调用了这个翻译单元，编译器就要产生一个实例化。这会是很耗时的。然后在编译的时候又要把多余的实例化删掉，这也增加了链接级的负担。

```
int main() {
  int x = 3;
  fun<int>(x); //我们使用显示实例化的方式来调用fun()函数。这句话有两个作用： 1.显示的把模板实例化为了函数
2.调用了这个函数
}
//偶尔我们希望只是显示实例化这个函数，而不去调用它。


//显式实例化
template <typename T>
void fun(T x) {
  std::cout << x;
}

template
void fun<int>(int x); //这行和上一行表示要对模板进行显式实例化。编译器会使用原始模板的逻辑来进行实例化
这行和上一行一起叫显式实例化定义
```

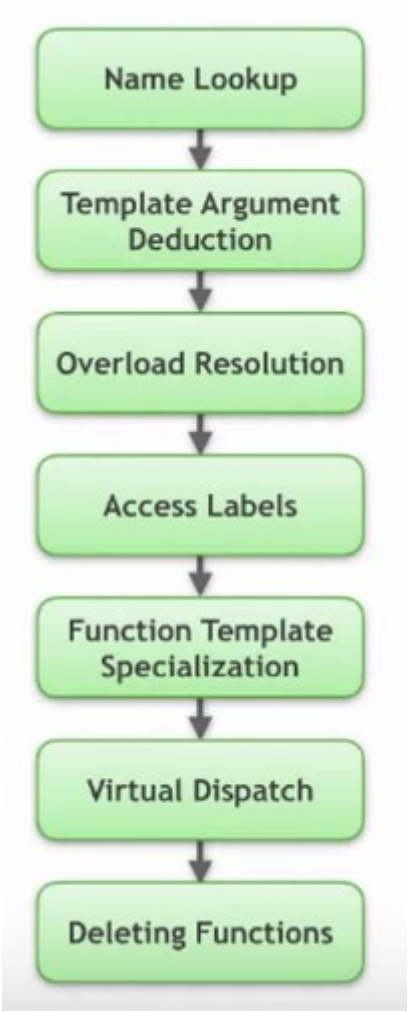
```
template <typename T>
void fun(T x)
{
}

template<> //模板函数特化
void fun(int x)
{
}
```

函数模板——续 4

函数模板的特化：
因为特化和实际版本的位置不同，或者提供模板实参的方式不同，那么函数会选择不同的函数来执行。换句话来说，你稍微写错一点，就可能不是我们想要的

- 避免使用函数模板的特化（参考资料）
  - 不参与重载解析，会产生反直觉的效果
  - 通常可以用重载代替
  - 一些不便于重载的情况：无法建立模板形参与函数形参的关联
    - 使用 if constexpr 解决
    - 引入“假”函数形参
    - 通过类模板特化解决
- (C++20) 函数模板的简化形式：使用 auto 定义模板参数类型
  - 优势：书写简捷
  - 劣势：在函数内部需要间接获取参数类型信息



```cpp
template <typename T>
void fun(T x) {
  std::cout << "1\n";
}

template <typename T>
void fun(T* x) {
  std::cout << "2\n";
}

template <>
void fun(int* x) {
  std::cout << “3\n";
}

int main() {
  int x;
  fun(&x);//程序输出3
具体还是看一下视频把
}
```

```cpp
// if constexpr
template <typename Res, typename T>
Res fun(T x) {
  if constexpr(std::is_same_v<Res, int>) {
    std::cout << "1\n";
  } else {
    std::cout << "2\n";
  }
  return Res{};
}

int main() {
  int x;
  fun<int>(&x);
}


template<typename Res, typename T>
Res fun(T x, const Res&) {
  std::cout << "2\n";
  return Res{};
}

template <typename T> //这是函数重载，不是函数
特化，因为这个template里面是有东西的
int fun(T x, const int&) {
  std::cout << "1\n";
  return Res{};
}
int main() {
  int x;
  fun(&x, int{});
}
```

类模板与成员函数模板

- 使用 template 关键字引入模板： template<typename T> class B {...};
  - 类模板的声明与定义—翻译单元的一处定义原则
  - 成员函数只有在调用时才会被实例化
  - 类内类模板名称的简写
  - 类模板成员函数的定义（类内、类外）
- 成员函数模板
  - 类的成员函数模板
  - 类模板的成员函数模板
- 友元函数（模板）
  - 可以声明一个函数模板为某个类（模板）的友元
  - C++11 支持声明模板参数为友元

优点：减少编译时间，减少编译文件的大小。

回过头去看vector的定义，发现他就是个类模板
搜索vector的实现：gcc github vector: gcc/stl_vector.h at master · gcc-mirror/gcc - GitHub

```
友元函数
template<typename T2>
void fun();

template<typanme T>
class B {
  template<typename T2>
  friend void fun();//是个函数模板，这是个友元函数，不是成员函数。友元函数模板比较少见

  int x;
};

template<typename T2>
void fun() {
  B<int> tmp1;
  tmp1.x;

  B<char> tmp1;
  tmp1.x;
// 这个友元函数对于任何的B的类模板所实例化出的类都是友元。
}
int main() {
  fun<float>();
}
```

```
友元函数2
template<typename T>
class B {
  friend void fun(B<T> input)
  {
  std::cout << input.x;
  }//和左边的友元函数的区别是，这是个普通的函数，但是这个普通的函数会依赖于B的存在而存在。虽然这个fun是个普通的函数，但是这个fun()有多少种，是需要看B有多少种实例化而产生的。
例：
B<int> val;
fun(val);//fun有不同种
B<float> val2;
fun(val2);

  //常用例子：
  friend auto operator+(B input1, B input2) {
    B res;
    res.x = input1.x + input2.x;
    return res;
  }
  int x = 3;
};

int main() {
  B<int> val;
  fun(val);

  B<int> val1;
  B<int> val2;
  B<int> res = val1+val2;
}
```

类模板定义

```
template<typename T>
class B
{
public:
  void fun(T input) {
    std::cout << input;
  }
  auto fun1() {
    return B<T>{};
    //可以对上句进行简写，可以写成return B{};
  }
};

//类模板声明
template<typename T>
class B;

int main() {
  B<int> x;
  x.fun(3);//如果成员函数没有被调用，那么成员函数就不会被实例化出来。好处：1.节省编译程序的大小和时间.2. 可能有的函数需要一些特殊的支持，例如对这个<<符号的支持。那么只要我们不调用这个函数，那么程序就能编译。可以使用一些其它的函数。
}
```

模板成员函数的类外定义

```
template<typename T>
class B {
public:
  void fun();
};

template<typename T>
void B<T>::fun() {}
//如果可能的话，最好还是在类内定义。当然如果类很大还是定义在外面
```

```
类的成员函数模板
class B {
public:
  template<typename T>
  void fun() {
  }//类内定义
  void fun1();
};

template<typename T>
void B::fun2() {}//类外定义,注意，这里B后面没有尖括号

//类模板的成员函数模板
template <typename T1>
class A {
public:
  template <typename T2> //类内
  void fun() {}//fun内部既可以使用T1，又可以使用T2
};

int main() {
  B x;
  x.fun<int>();
  A<int> y;
  y.fun<float>();
}
```

```
类外定义
template <typename T1>
template<typename T2>
void B<T1>::fun() {}
```

C++11 支持声明模板参数为友元

```
//用途不太大
template<typename T>
class B{
  friend T;
};
```

类模板与成员函数模板——续 1

- 类模板的实例化 (https://en.cppreference.com/w/cpp/language/class_template)
  - 与函数实例化很像
  - 可以实例化整个类，或者类中的某个成员函数
- 类模板的（完全）特化 / 部分特化（偏特化）

  函数模板不支持部分特化，但是类模板是支持的
  类模板的特化是一个非常重要的技术
  - 特化版本与基础版本可以拥有完全不同的实现
- 类模板的实参推导（从 C++17 开始）
  - 基于构造函数的实参推导
  - 用户自定义的推导指引
  - 注意：引入实参推导并不意味着降低了类型限制！
  - C++ 17 之前的解决方案：引入辅助模板函数

```
//基于构造函数的实参推导
template <typename T>
struct B {
  B(T input) {}
  void fun() {
    std::cout << "1\n";
  }
};

int main() {
  int i = 3;
  B x(3); //会自动推导出T为int
}
```

```
template<typename T>
struct B {
void fun() {}
};

//完全特化
template<>
struct B<int>
{
void fun() {}
//特化版本和基础版本可以有完全
不同的实现
void fun2() {}
//我们可以在特化版本中不定义fun()
而是定义一个新的fun2()
};

int main() {
  B<int> x;
  x.fun2();
}
```

C++17虽然不支持类模板的推导，但是
支持函数模板的推导。因此可以引入辅助
模板函数来解决

```
template<typename T1, typename T2>
std::pair<T1,T2> make_pair(T1 val1, T2 val2) {
  return std::pair<T1, T2>(val1, val2);
}

int main() {
  auto x = make_pair(3, 3.14);
}
```

这个方法其实很多人在用，很普遍。c++中其实就有
make_pair这个函数，我们写的make_pair是个简化的版本
其实也是类模板的推导。不过比我们这个函数考虑更多
的情况而已

```
//部分特化
template<typename T,typename T2>
struct B {
  void fun() {}
};

//部分特化
template <typename T2>
struct B<int, T2>
{
  void fun2() {}
};

int main() {
  B<int, double> x; //会调用部分特
化的函数
  x.fun2();
}
```

g++ -c "main.cpp" -O2 -Wall --std=c++2a -DNDEBUG -o ./Release/main.cpp.o -I. -I.

Concepts

这里有个链接。是cppreference上的非常好的关于concepts的文档

- 模板的问题：没有对模板参数引入相应的限制

9:11

  - 参数是否可以正常工作，通常需要阅读代码进行理解
  - 编译报错友好性较差 (vector<int&>)

谓词的概念就是基于给定的输入，返回true或false

- （C++20） Concepts：编译期谓词，基于给定的输入，返回 true 或 false
  - 与 constraints （require 从句）一起使用限制模板参数
  - 通常置于表示模板形参的尖括号后面进行限制
- Concept 的定义与使用
  - 包含一个模板参数的 Concept
    - 使用 requires 从句
    - 直接替换 typename
  - 包含多个模板参数的 Concept
    - 用做类型 constraint 时，少传递一个参数，推导出的类型将作为首个参数

```
template<typename T>
concept IsAvail = std::is_same_v<T, int> || std::is_same_v<T, float>;
//定义约束
//是在编译器求值的

//用法
template <typename T>
  requires IsAvail<T>
void fun(T input) {}

int main() {
  std::cout << IsAvail<int> << std::endl; //结果输出1

  fun(3);//可以编译
  fun(true);//不可以编译，报错更友好
}
```

```
 //直接替换typename
 #include<iostream>
 #include<vector>
 #include<type_traits>

 template<typename T>
 concept IsIntOrFloat = std::is_same_v<T, int> || std::is_same_v<T, float>;

 template<IsIntOrFloat T>
 void fun(T input> {}

 int main() {
   fun(3); //ok
   fun(true);//error
 }
```

```
    template <typename T, typename T2>
    concept IsAvail = !std::is_same_v<T, T2>;
    //判断两个类型是否一致，不一致则通过，一致则报错

    template <typename T, typename T2>
      requires IsAvail<T, T2>
    void fun(T input1, T2 input2>
    {}

    int main() {
      fun(3, 3.5);
    }
```

用做类型 constraint 时，少传递一个参数，推导出的类型将作为首个参数

```
template <typename T, typename T2>
concept IsAvail = !std::is_same_v<T, T2>;

template <IsAvail<int> T>//类型constraint.等价于IsAvail<T, int>, T一定是第一个参数。
void fun(T input1) {}

int main() {
  std::cout << IsAvail<int, char> << std::endl;
  fun(1.3);
}
```

Concepts—— 续

https://en.cppreference.com/w/cpp/language/constraints
https://en.cppreference.com/w/cpp/header/type_traits

- requires 表达式 这里面的例子都可以在cppreference中找到
  - 简单表达式：表明可以接收的操作
  - 类型表达式：表明是一个有效的类型
  - 复合表达式：表明操作的有效性，以及操作返回类型的特性
  - 嵌套表达式：包含其它的限定表达式

```
//特化小技巧在声明中引入A||B进行限制。系统会选择
最严格的版本

template<typename T>
requires std::is_same_v<int> || std::is_same_v
float>
class B;

template<>
class B<int> {};

template<>
class B<float> {};
```

- 注意区分 requires 从句与 requires 表达式
- requires 从句会影响重载解析与特化版本的选取
  - 只有 requires 从句有效而且返回为 true 时相应的模板才会被考虑
  - requires 从句所引入的限定具有偏序特性，系统会选择限制最严格的版本
- 特化小技巧：在声明中引入“ A||B” 进行限制，之后分别针对 A 与 B 引入特化

```
template<typename T>
requires std::is_integral_v<T> || std::
is_floating_point_v<T>
class B;
```

```
//简单表达式
template<typename T>
concept Addable =
requires (T a, T b) {
  a + b;
};

template <Addable T>
auto fun(T x, T y) {
  return x + y;
}

struct Str {};

int main() {
  Str a;
  Str b;
  fun(a, b);//系统会报错，
因为Str a和b不可加，不满足
约束
}
```

```
//类型表达式
template<typename T>
concept Avail =
requires {
  typename T::inter;
};

template <Avail T>
auto fun(T x, T y) {
  return x + y;
}

struct Str {
  using inter = int;
};
int main() {
  fun(3);//会报错，因为它要求
传入的类型必须有inter的类型定义
  fun(Str{});//不会报错，因为
满足了要求
}
```

```
//复合表达式
template <typename T>
concept Avail =
requires (T x) {
  {x + 1} -> int;
};
template <Avail T>
auto fun(T x) {}

struct Str {
  using inter = int;
};
int main() {
  fun(3);
}
```

```
//系统会选择较严格的版本
template <typename T>// 会选择这个版本，因为这个更严格
concept C1 = std::is_same_v int>;
template <typename T>
concept C2 = std::is_same_v int> || std::is_same_v float>;
template <C1 T>
void fun(T) {
std::cout  1;
}
template <C2  T>
void fun(T) {
  std::cout  2;
}
int main() {
  fun(3);
}
```

模板相关内容——数值模板参数与模板模板参数

```
template<int a>
int fun(int x) {
  return x+a;
}
int main() {
  fun<3>(5);
}
```

数值作为模板参数的要求：
1. 数值一定是要有类型的

- 模板可以接收（编译期常量）数值作为模板参数
  - template <int a> class Str;

如果我们不想显示的声明数值的类型，则用下面的方法：

  - template <typename T, T value> class Str;
  - (C++ 17) template <auto value> class Str;
  - (C++ 20) 接收字面值类对象与浮点数作为模板参数
    - 目前 clang 12 不支持接收浮点数作为模板参数
- 接收模板作为模板参数
  - template <template<typename T> class C> class Str;
  - (C++17) template <template<typename T> typename C> class Str;
  - C++17 开始，模板的模板实参考虑缺省模板实参（clang 12 支持程度有限）
    - Str<vector> 是否支持?

template<>尖括号
中间就是上面
模板直接粘贴上

```
//接收模板作为参数模板
template <typename T>
class C{};

template <template<typename T> class T2>
void fun() {
  T2<int> tmp;
}
//函数模板的第一个参数是一个类模板，这个
类模板只能接收一个模板实参。
在整个这个程序当中，这个T其实是没有用的，
我们只是想说明这个类模板只接收一个参数。
而这个T并不重要。因此我们可以把那个T删掉
//template <template <typename> class T2>
从c++17开始，class也可以换成typename
//template <template<typename> typename T2>

int main() {
  fun<C>();
}

fun<std::vector>();
//这个代码对上面的函数来说也是合法的。
```

```
template <typenmae T>
class C{};

template <typename T, typename T2 = int>
class C2{};

template <template <typename> typename T2>
void fun() {
}
int main() {
  fun<C2>();
}
```

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

template <typename T, typename T2>
class C2{};

template <template <typename> typename T2>
void fun()
{
    T2<int> tmp;
}

int main()
{
    fun<C2>();
}
```

这个函数会报错，因为C2和fun的参数类型不匹配
C2需要两个模板参数，fun只需要一个

模板相关内容——别名模板与变长模板

cppreference: type alias, alias template

- 可以使用 using 引入别名模板
  - 为模板本身引入别名
  - 为类模板的成员引入别名
  - 别名模板不支持特化，但可以基于类模板的特化引入别名，以实现类似特化的功能
    - 注意与实参推导的关系

```
template<class T>
struct Alloc { };
template<class T>
using Vec = vector<T, Alloc<T>>; // type-id is vector<T, Alloc<T>>
Vec<int> v; // Vec<int> is the same as vector<int, Alloc<int>>
```

基于类模板的特化引入别名，例：我们希望实现如果是int返回int的引用，其它情况返回Pointer

```
template<typename T>
struct B
{
  using type = T*;
};
template<>
struct B<int> //特化
{
  using type = int&;
}
template<typename T>
using MyPointer = typename B<T>::type;

int main() {
  MyPointer<int> x;
}
```

c++11之后

- 变长模板（Variadic Template）

cppreference，老师觉得总结的很好：parameter_pack

  - 变长模板参数与参数包 parameter pack
  - 变长模板参数可以是数值、类型或模板
  - sizeof... 操作 可以获取参数包里面的个数
  - 注意变长模板参数的位置

```
#include <iostream>

template <typename... T>
void fun(T... args)
{
    std::cout << sizeof...(T) << std::endl;
}

int main()
{
    fun(3, 5.3, 'c');
}
```

sizeof... 运算符

sizeof... 也被归类为包展开

```
template<class... Types>
struct count {
    static const std::size_t value = sizeof...(Types);
};
```

```
//别名模板
template <typename T>
using AddPointer = T*;

int main() {
  AddPointer<int> x;
}
```
AddPointer其实就是T*

```
//为类模板的成员引入别名
template<typename T>
struct B {
  using TP = T*;
};

template<typename T>
using MyPointer = typename B<T>::TP;

int main() {
  AddPointer<int> x;
}
```

```
template <int... a>
//现在是一个形参包，
表示里面可以包含0-n个int的数值
void fun()
{
}

int main() {
  fun<1,2,3>();
}

template <typename... a> //接收类型
void fun() {
}

int main() {
  fun<int, double, char>();
}
```

```
//带可选名字的函数形参包
template <typename... T>
void fun(T... args) {
}

int main() {
  fun<int, double, char>(3, 5.3, 'c');
}

//变长模板template  class C;template
typename T2>class B;template
typename T2>class BT...>, T2>{};int
main() {}
```

变参类模板可用任意数量的模板实参实例化：

```
template<class ... Types> struct Tuple {};
Tuple<> t0;           // Types 不包含实参
Tuple<int> t1;        // Types 包含一个实参: int
Tuple<int, float> t2; // Types 包含二个实参: int 与 float
Tuple<0> error;       // 错误: 0 不是类型
```

变参函数模板可用任意数量的函数实参调用（模板实参通过模板实参推导推导）：

```
template<class ... Types> void f(Types ... args);
f();       // OK: args 不包含实参
f(1);      // OK: args 包含一个实参: int
f(2, 1.0); // OK: args 包含二个实参: int 与 double
```

在主裘模板中，模板形参包必须是模板形参列表的最后一个形参。在函数模板中，模板参数包可以在列表中稍早出现，只要其后的所有形参均可从函数实参推导或拥有默认实参即可：

```
template<typename... Ts, typename U> struct Invalid; // 错误: Ts.. 不在结尾

template<typename ...Ts, typename U, typename=void>
void valid(U, Ts...);      // OK: 能推导 U
// void valid(Ts..., U);   // 不能使用: Ts... 在此位置是非推导语境

valid(1.0, 1, 2, 3);       // OK: 推导 U 为 double, Ts 为 {int,int,int}
```

主类模板意味着不是一个特化的类模板，就是普通的模板
如果是特化模板，就不需要满足上面说的 形参包必须是模板形参列表的最后一个形参

模板相关内容——包展开与折叠表达式

- (C++11) 通过包展开技术操作变长模板参数
  - 包展开语句可以很复杂，需要明确是哪一部分展开，在哪里展开
- (C++17) 折叠表达式 (cpp reference)
  - 基于逗号的折叠表达式应用
  - 折叠表达式用于表达式求值，无法处理输入（输出）是类型与模板的情形

```
template <typename U, typename... T>
void fun(U u, T... args) {
  std::cout << u << std::endl;
  fun(args...);
}
int main() {
  fun(1,2,"hello", 'c');
}
```

```
//改进版折叠表达式,和左边的功能是一致的
template <typename... T>
void fun(T... args) {
  ((std::cout << args << std::endl), ...);
}
```

包展开

模式后随省略号，其中至少有一个形参包的名字至少出现一次，其被*展开*成零或更多个逗号分隔的模式实例，其中形参包的名字按顺序被替换成包中的各个元素。

```
template<class ...Us> void f(Us... pargs) {}
template<class ...Ts> void g(Ts... args) {
    f(&args...); // “&args...” 是包展开
                 // “&args” 是其模式
}
g(1, 0.2, "a"); // Ts... args 展开成 int E1, double E2, const char* E3
                // &args... 展开成 &E1, &E2, &E3
                // Us... 展开成 int* E1, double* E2, const char** E3
```

模板相关内容——完美转发与 lambda 表达式模板

- (C++11) 完美转发： std::forward 函数
  - 通常与万能引用结合使用
  - 同时处理传入参数是左值或右值的情形　右值引用的变量是左值
- (C++20) lambda表达式模板

模板相关内容——消除歧义与变量模板

- 使用 typename 与 template 消除歧义
  - 使用 typename 表示一个依赖名称是类型而非静态数据成员
  - 使用 template 表示一个依赖名称是模板
  - template 与成员函数模板调用
- (C++14) 变量模板
  - template <typename T> T pi = (T)3.1415926;
  - 其它形式的变量模板

之前的视频中引入typename也是为了消除歧义

```
//使用template表示一个依赖名称是模板
struct Str {
  template <typename T>
  static void internal() {}
};

template<typename T>
void fun() {
  T::internal<int>();
}
int main() {
  fun<Str>();
}
1.internal是T内部的一个模板，int是模板参数
2.internal是一个数，<符号是一个小于号，这个表达式是用来判断Internal是不是小于某一个值的
编译器会认为是2
如果加入template，编译器会认为是1
T::template internal<int>()
```

```
//消除歧义
template<typename T>
void fun() {
  T::internal* p;}
//两种方法理解这句话
1.internal是T所关联的类型，*表明我要声明这个类型的指针根据这个指针，我定义了一个变量P
2. internal是T的静态数据成员它可能是个Int值。*在这里面表示乘法。
编译器会选择第二种方法
 如果加入typename，编译器就会认为是第一种方法 typename T::internal* p;
```

```
struct Str {
  inline const static int internal=3;
};
int p = 5;
//接左边的template

int main() {
   fun<Str>();
}
//是可以编译的
```

```
template <typename T>
T pi = (T)3.1415
int main() {
  pi<float>;//float类型的数，这个数是3.1415
  pi<int>;//int类型的数，是3.1415进行int类型
转化之后的结果

template<typename T>
unsigned MySize = sizeof(T);

int main() {
  std::cout << MySize<float>;
  std::cout << MySize<int>;
}
```

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening