



89-1
90-1
90-2
90-3
90-4
91-1
91-2
92-1
92-2
93-1

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



1. 函数模板



2. 类模板与成员函数模板



3. Concepts



4. 模板相关内容



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 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。
 - 模板实例化
 - 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;
}
```

```
1 #include <iostream>
2
3 template <typename T>
4 void fun(T input)
5 {
6     std::cout << input << std::endl;
7 }
8
9 int main()
10 {
11     fun<int>(3);
12     fun<double>(3);
13 }
14
```

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) {
    //
}
```

```
11 template<>
12 void fun<int>(int input)
13 {
14     std::cout.operator<<(input).operator<<(std::endl);
15 }
16 #endif
17
18
19 /* First instantiated from: insights.cpp:12 */
20 #ifdef INSIGHTS_USE_TEMPLATE
21 #template<>
22 void fun<double>(double input)
23 {
24     std::cout.operator<<(input).operator<<(std::endl);
25 }
26 #endif
27
```



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

这两种写法都可以

- **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)`

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。

- 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

```
int main() {  
    int x = 3;  
    fun<int>(x); //我们使用显示实例化的  
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
```

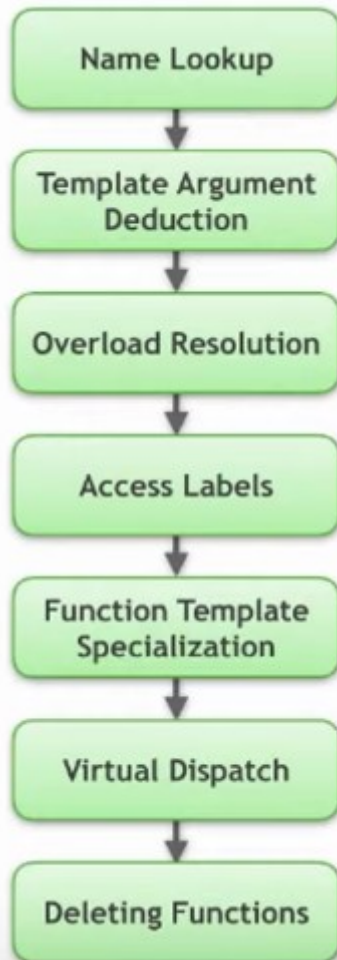


函数模板——续 4

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



```
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
}
```

```
// 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 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

- 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

友元函数

```
template<typename T2>
void fun();
```

```
template<typename 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
```

// 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 x;
```

```
};
```

```
template<typename T2>
void fun() {
```

```
    B<int> tmp1;
```

```
    tmp1.x;
```

```
    B<char> tmp1;
```

```
    tmp1.x;
```

```
}
```

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

友元函数2

```
template<typename T>
```

```
class B {
```

```
    friend void fun(B input)
    {
```

```
        std::cout << input.x;
```

```
    } //这是个普通的函数，但是
```

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。

//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> val1;
```

```
    fun(val1);
```

```
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() {}
```

C++11 支持声明模板参数为友元

```
template<typename T>
class B{
    friend T;
};
```

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
```

//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() {}
```



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
 - 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
}
```

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++的make_pair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



Concepts

- 模板的问题：没有对模板参数引入相应的限制
 - 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&>)
- (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>;
//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() {
    fun(1.3);
}
```



Concepts——续

<https://en.cppreference.com/w/cpp/language/constraints>

- requires 表达式
 - 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
- 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

```
//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

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

//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>();  
}
```

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

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

```
template <template <typename> typename T2>  
void fun() {  
}
```

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



模板相关内容——别名模板与变长模板 [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
- 变长模板（Variadic Template） [参考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

//别名模板

```
template <typename T>
using AddPointer = T*;
```

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

//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 <typename... T> class C;
```

```
template <typename T1, typename T2>
class B;
```

```
template <typename... T, typename T2>
class B<C<T...>, T2>
{};
int main() {
}
```




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>      //改进版折叠表达式, 和左边的功能是一致的
void fun(U u, T... args) {
    std::cout << u << std::endl;
    fun(args...);
}
int main() {
    fun(1, 2, "hello", 'c');
}
```

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




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

感谢聆听 !
Thanks for Listening

