模版进阶复习

1. 非类型模版参数

1.1 参数类型分类

模版参数分为类型模版参数、非类型模版参数、模版模版参数

- 类型模版参数即：出现在模版参数列表中，跟在class或在typename之类后的参数类型名称。
- 非类型模版参数即：
  - 代表一个具体的值而不是类型
  - 必须是编译时的常量
  - 允许的类型包括：
    - 整型或枚举
    - 指针或引用（指向具有静态存储期的对象或函数）
    - C++20起支持浮点类型
    - C++20起支持字面类类型

注意：

1. **浮点数、类对象以及字符串是不允许作为非类型模板参数的**。
2. **非类型的模板参数必须在编译期就能确认结果**。

代码块

```
template <int N>
class Array {
    int data[N];
};

template <auto value>
void printValue() {
    std::cout << value << std::endl;
}

template <const char* str>
class Message {
    // ...
};

```

```cpp
// C++20起
template <double Value>
struct FloatingPointWrapper {
    // ...
};
```

- 模版模版参数即：
  - 接受一个模板作为参数
  - 用于需要传递模板本身而不是具体实例化类型的场景

代码块

```cpp
template <template <typename> class Container, typename T>
class Adapter {
    Container<T> c;
};

// 使用
template <typename T>
using MyVector = std::vector<T>;

Adapter<MyVector, int> adapter;  // 内部使用 std::vector<int>
```

1.2 C++11及以后版本的扩展

- 可变参数模版

代码块

```cpp
template <typename... Args>
void printAll(Args... args);
```

- 默认模版参数

代码块

```cpp
template <typename T = int, int Size = 10>
class Buffer {
    // ...
};
```

- auto作为模版参数(C++17)

代码块

```
template <auto Value>
struct Constant {
    static constexpr auto value = Value;
};
```

- 概念约束模版参数(C++20)

代码块

```
template <std::integral T>
void process(T value);
```

2. 模版的特化

2.1 概念

> 模板特化是指为特定的模板参数提供特殊的实现。通常情况下，**使用模板可以实现一些与类型无关的代码，但对于一些特殊类型的可能　会得到一些错误的结果**，需要特殊处理，比如：实现了一个专门用来进行小于比较 的函数模板

代码块

```
// 函数模板 -- 参数匹配
template<class T>
bool Less(T left, T right) { return left < right; }

int main()
{
    cout << Less(1, 2) << endl; // 可以比较，结果正确
    Date d1(2022, 7, 7);
    Date d2(2022, 7, 8);
    cout << Less(d1, d2) << endl; // 可以比较，结果正确

    Date* p1 = &d1;
    Date* p2 = &d2;
    cout << Less(p1, p2) << endl; // 可以比较，结果错误
    return 0;
}
```

可以看到，Less绝大多数情况下都可以正常比较，但是在特殊场景下就得到错误的结果。上述示例中，p1指向的d1显然小于p2指向的d2对象，但是Less内部并没有比较p1和p2指向的对象内容，而比较的是p1和p2指针的地址，这就无法达到预期而错误。

此时，就**需要对模板进行特化。即：在原模板类的基础上，针对特殊类型所进行特殊化的实现方式**。模板特化中分为**函数模板特化**与**类模板特化**。

2.2 函数模版特化

> 函数模版的特化步骤：
>
> - 必须要先有一个基础的函数模版
> - 关键字template后面接一对空的尖括号<>
> - 函数名后面跟一对尖括号，尖括号中指定需要特化的类型
> - 函数形参表：必须要和模版函数的基础参数类型完全相同，如果不同编译器可能会报一些奇怪的错误。

代码块

```
// 函数模板 -- 参数匹配
template<class T>
bool Less(T left, T right) { return left < right; }

// 对Less函数模板进行特化
template<>
bool Less<Date*>(Date* left, Date* right) { return *left < *right; }

int main()
{
    cout << Less(1, 2) << endl;

    Date d1(2022, 7, 7);
    Date d2(2022, 7, 8);
    cout << Less(d1, d2) << endl;

    Date* p1 = &d1;
    Date* p2 = &d2;
    cout << Less(p1, p2) << endl; // 调用特化之后的版本，而不走模板生成了

    return 0;
}
```

注意：一般情况下如果函数模板遇到不能处理或者处理有误的类型，为了实现简单通常都是将该函数直接给出。

代码块

```
bool Less(Date* left, Date* right) { return *left < *right; }
```

该种实现简单明了，代码的可读性高，容易书写，因为对于一些参数类型复杂的函数模板，特化时特别给出，因此函数模板不建议特化。

2.3 类模版特化

2.3.1 全特化

> 为所有模板参数指定具体类型或值，提供完全特化的实现。

代码块

```
template<class T1, class T2>
class Data
{
public:
    Data() {cout<<"Data<T1, T2>" <<endl;}
private:
    T1 _d1;
    T2 _d2;
};

template<>
class Data<int, char>
{
public:
    Data() {cout<<"Data<int, char>" <<endl;}
private:
    int _d1;
    char _d2;
};

void TestVector()
{
    Data<int, int> d1;
    Data<int, char> d2;
}
```

2.3.2 偏特化

> 偏特化：任何针对模版参数进一步进行条件限制设计的特化版本。

偏特化有以下两种表现方式：

- 部分特化：将模板参数类表中的一部分参数特化。

代码块

```
// 主模板
template <typename T, typename U>
class MyPair {
public:
    void print() { std::cout << "General template" << std::endl; }
};

// 偏特化版本 - 当两个类型相同时
template <typename T>
class MyPair<T, T> {
public:
    void print() { std::cout << "Partial specialization for same types" << std::endl; }
};

// 偏特化版本 - 当第二个类型是int时
template <typename T>
class MyPair<T, int> {
public:
    void print() { std::cout << "Partial specialization for second type int" << std::endl; }
};

// 使用
MyPair<double, char> p1;  // 使用主模板
p1.print();               // 输出: General template

MyPair<float, float> p2;  // 使用第一个偏特化
p2.print();               // 输出: Partial specialization for same types

MyPair<char, int> p3;     // 使用第二个偏特化
p3.print();               // 输出: Partial specialization for second type int
```

- 参数更进一步的限制：偏特化并不仅仅是指特化部分参数，而是针对模板参数更进一步的条件限制所设计出来的一个特化版本。

代码块

```
//两个参数偏特化为指针类型
template <typename T1, typename T2>
class Data <T1*, T2*>
{
public:
    Data() {cout<<"Data<T1*, T2*>" <<endl;}
private:
    T1 _d1;
    T2 _d2;
};

//两个参数偏特化为引用类型
template <typename T1, typename T2>
class Data <T1&, T2&>
{
public:
    Data(const T1& d1, const T2& d2)
    : _d1(d1)
    , _d2(d2)
    { cout<<"Data<T1&, T2&>" <<endl; }

private:
    const T1 & _d1;
    const T2 & _d2;
};

void test2 ()
{
    Data<double , int> d1; // 调用特化的int版本
    Data<int , double> d2; // 调用基础的模板
    Data<int *, int*> d3; // 调用特化的指针版本
    Data<int&, int&> d4(1, 2); // 调用特化的指针版本
}

```

2.3.3 类模版特化应用示例

> 有如下专门用来按照小于比较的类模板Less：

代码块

```
#include<vector>
#include<algorithm>
```

```cpp
template<class T>
struct Less
{
    bool operator()(const T& x, const T& y) const { return x < y; }
};

int main()
{
    Date d1(2022, 7, 7);
    Date d2(2022, 7, 6);
    Date d3(2022, 7, 8);

    vector<Date> v1;
    v1.push_back(d1);
    v1.push_back(d2);
    v1.push_back(d3);

    // 可以直接排序，结果是日期升序
    sort(v1.begin(), v1.end(), Less<Date>());
    vector<Date*> v2;
    v2.push_back(&d1);
    v2.push_back(&d2);
    v2.push_back(&d3);

    // 可以直接排序，结果错误日期还不是升序，而v2中放的地址是升序
    // 此处需要在排序过程中，让sort比较v2中存放地址指向的日期对象
    // 但是走Less模板，sort在排序时实际比较的是v2中指针的地址，因此无法达到预期
    sort(v2.begin(), v2.end(), Less<Date*>());

    return 0;
}
```

通过观察上述程序的结果发现，对于日期对象可以直接排序，并且结果是正确的。但是如果待排序元素是指针，结果就不一定正确。因为：sort最终按照Less模板中方式比较，所以只会比较指针，而不是比较指针指向空间中内容，此时可以使用类版本特化来处理上述问题：

代码块

```cpp
// 对Less类模板按照指针方式特化
template<>
struct Less<Date*>
{
    bool operator()(Date* x, Date* y) const { return *x < *y; }
};
```

特化之后，在运行上述代码，就可以得到正确的结果

3. 模版分离编译

3.1 什么是分离编译

> 一个程序（项目）由若干个源文件共同实现，而每个源文件单独编译生成目标文件，最后将所有目标文件链接起来形成单一的可执行文件的过程称为分离编译模式。

3.2 模板的分离编译

🧐 **模版分离编译问题本质**

在C++中，模板不是普通的函数或类，而是"生成"函数或类的"模具"。编译器需要看到模板的完整定义（不仅仅是声明）才能为特定类型实例化模板代码。这导致了传统的声明/定义分离（.h和.cpp文件分离）在模板中不适用。

为什么模版不能直接分离编译

- 编译模型限制：模版实例化需要在编译时完成，编译器必须能看到完整的模版定义
- 延迟实例化：模版代码只有在被使用时才会实例化
- 跨编译单元问题：不同.cpp文件可能会对同一模版进行不同的实例化

分析：

C/C++程序要运行，一般要经历一下步骤：
预处理 ---> 编译 ---> 汇编 ---> 链接

编译：对程序按照语言特性进行词法、语法、语义分析，错误检查无误后生成汇编代码
注意头文件不参与编译 编译器对工程中的多个源文件是分离开单独编译的。

链接：将多个obj文件合并成一个，并处理没有解决的地址问题

a.h
```
template<class T>
T Add(const T& left, const T& right);
```

20180925Test_manifest.rc
a.obj
main.obj

a.cpp
```
#include "a.h"
template<class T>
T Add(const T& left, const T& right)
{
    return left + right;
}
```

在a.cpp中，编译器没有看到对Add模板函数的实例化，因此不会生成具体的加法函数

main.cpp
```
#include "a.h"
int main()
{
    Add(1, 2);   call Add<int>
    Add(1.0, 2.0); call Add<double>
    return 0;
}
```

在main.obj中调用的Add<int>与Add<double>，编译器在链接时才会找其地址，但是这两个函数没有实例化没有生成具体代码，因此链接时报错。

```
无法解析的外部符号：??$Add@N@@YANABN0@Z)，该符号在函数 _main 中被引用
无法解析的外部符号：??$Add@H@@YAHABH0@Z)，该符号在函数 _main 中被引用
```

解决方案

a. 包含模式(最常见)

将模板的声明和定义都放在头文件中：

代码块

```
// mytemplate.h
#ifndef MYTEMPLATE_H
#define MYTEMPLATE_H

template <typename T>
class MyTemplate {
public:
    void doSomething(T param);
};

// 模板成员函数定义也在头文件中
template <typename T>
void MyTemplate<T>::doSomething(T param) {
    // 实现代码
}

#endif
```

优点：

- 简单直接
- 保证编译器能看到完整定义

缺点：

- 增加了头文件大小
- 暴露实现细节

b. 显示实例化 (这种方法不实用，不推荐使用)

在.cpp文件中显式声明需要的实例化：

代码块

```
// mytemplate.h
template <typename T>
class MyTemplate {
public:
    void doSomething(T param);
```

```
};

// mytemplate.cpp
#include "mytemplate.h"

template <typename T>
void MyTemplate<T>::doSomething(T param) {
    // 实现代码
}

// 显式实例化
template class MyTemplate<int>;
template class MyTemplate<double>;
```

c. **C++20引入的模块系统部分解决了模版分离编译问题:**

代码块

```
// mytemplate.ixx
export module mytemplate;

export template <typename T>
class MyTemplate {
public:
    void doSomething(T param) {
        // 实现代码
    }
};
```

模块提供了更好的封装性和编译效率，但目前编译器支持仍在完善中。