

C++程序设计基础 第十章

前言

第十章 模版

10.1 模版的概念

10.2 函数模版

10.2.1 函数模版的定义

10.2.2 函数模版实例化

1. 隐式实例化

2. 显示实例化

10.2.3 函数模版重载

10.3 类模板

10.3.1 类模版定义与实例化

10.3.2 类模版的派生

1. 类模板派生普通类

2. 类模板派生类模板

3. 普通类派生类模版

10.3.3 类模板与友元函数

1. 非模版友元函数

2. 约束模版友元函数

3. 非约束模版友元函数

10.4 模版的参数

1. 类型参数

2. 非类型参数

3. 模版类型参数

10.5 模版特化

1. 偏特化

2. 全特化

本文档由 @ItsJiale 创作，作者博客：<https://jiale.domcer.com/>，作者依据数学与大数据学院 2024 级大数据教学班的授课重点倾向编写而成。所有内容均为手动逐字录入，其中加上了不少自己的理解与思考，耗费近一周时间精心完成。

此文档旨在助力复习 C++ 程序设计基础，为后续学习数据结构筑牢根基。信计专业的同学，也可参考本文档规划复习内容。需注意，若个人学习过程中存在不同侧重点或对重难点的理解有差异，应以教材内容为准。倘若文档内容存在任何不妥之处，恳请各位读者批评指正。

By: ItsJiale

2025.5.4

第十章 模版

为什么要使用模版？因为对于相同结构但数据类型不同的函数而言，只需要关注逻辑代码的编写，而不用关注实际具体的数据类型，对于代码的维护而言大大增强。

10.1 模版的概念

模板（Template）是 C++ 中一项强大的特性（在前端开发中也有模版的相关概念），它允许你编写通用的代码，这些代码可以处理多种数据类型，而不需要为每种数据类型都单独编写一份代码。模板主要分为函数模板和类模板。函数模版生成的实例叫做**模版函数**，类模版生成的实例叫做**模板类**。

10.2 函数模版

函数模板可以创建一个通用的函数，它能够处理不同的数据类型。定义函数模板时使用 `template` 关键字，后跟一个模板参数列表，通常用 `<typename T>` 或 `<class T>` 表示，这里的 `T` 是一个类型参数，可以代表任意数据类型。

10.2.1 函数模版的定义

格式如下：

```
template<class 类型占位符>
```

```
返回值类型 函数名（参数列表）
```

```
{
```

```
    //函数体;
```

```
}
```

上述语法格式中，`template` 是声明模板的关键字，`< >`中的参数称为**模板形参**，`class` 关键字用于标识模板形参，可以用 `typename` 关键字代替 `class` 关键字，

`typename` 和 `class` 并没有区别。模板形参不能为空，但是一个函数模板中可以有多模板形参，模板形参和函数形参相似。`template` 下面就是定义的一个函数模板，函数模板与普通的函数定义方式相同，只是参数列表中的数据类型要使用`< >`中的形参名表示。

```
1  #include <iostream>
2  using namespace std;
3  template<class T>
4  T add(T t1, T t2) {
5      return t1 + t2;
6  }
7  int main()
8  {
9      cout << add(3, 2) << endl;
10     cout << add(4.3, 3.2) << endl;
11     return 0;
12 }
13
```

10.2.2 函数模版实例化

所谓实例化，就是用类型参数替换模版中的模版参数，生成具体类型的函数。

1. 隐式实例化

隐式实例化是指在使用模板时，编译器根据提供的参数类型自动推导模板参数的类型，并生成对应的模板实例。

```
1  #include <iostream>
2  using namespace std;
3  // 函数模板
4  template <typename T>
5  T maximum(T a, T b) {
6      return (a > b)? a : b;
7  }
8
9  int main() {
10     int x = 5, y = 10;
11     // 隐式实例化, 编译器根据传入的参数类型 int 自动实例化 maximum 函数
12     cout << "The maximum of " << x << " and " << y << " is " << maximum(x,
13     y) << endl;
14
15     double m = 3.14, n = 2.71;
16     // 隐式实例化, 编译器根据传入的参数类型 double 自动实例化 maximum 函数
17     cout << "The maximum of " << m << " and " << n << " is " << maximum(m,
18     n) << endl;
19     return 0;
20 }
```

2. 显示实例化

显式实例化是程序编写者**明确指定模板参数的具体类型**, 要求编译器生成该特定类型的模板实例。

```
1  #include <iostream>
2  using namespace std;
3  // 函数模板
4  template <typename T>
5  T minimum(T a, T b) {
6      return (a < b)? a : b;
7  }
8
9  // 显式实例化 minimum 函数模板为 int 类型
10 template int minimum<int>(int a, int b);
11
12 int main() {
13     int p = 15, q = 20;
14     // 可以直接使用显式实例化后的 minimum 函数
15     cout << "The minimum of " << p << " and " << q << " is " << minimum(p,
16     q) << endl;
17     return 0;
18 }
```

在上述例子中，不是 `int` 类型的数据会转换为 `int` 类型再进行计算。

10.2.3 函数模版重载

函数模版可以进行实例化，以支持不同类型的参数，不同类型的参数调用会产生一系列重载函数。

```

1  #include<iostream>
2  using namespace std;
3  int max(const int& a,const int& b){ //非模板函数，求两个int类型数据的最大者
4      return a>b ? a:b;
5  }
6  template<class T>                //定义求两个任意类型数据的最大值
7  T max(const T& t1,const T& t2){
8      return t1>t2 ? t1:t2;
9  }
10 template<class T>                //定义求三个任意类型数据的最大值
11 T max(const T& t1,const T& t2,const T&t3){
12     return max(max(t1,t2),t3);
13 }
14 int main()
15 {
16     cout<< max(1,2)<<endl;        //调用非模板函数
17     cout<< max(1,2,3)<<endl;      //调用三个参数的函数模板
18     cout<< max('a','e')<<endl;   //调用两个参数的函数模板
19     cout<< max(6,3.2)<<endl;      //调用非模板函数
20     return 0;
21 }
22

```

形参个数不同——>函数模版重载

补充：为什么这里要用常量引用

1. **复制开销**：值传递会复制实参，当传递的对象较大时，会增加内存和时间开销。
2. **函数内可能意外修改参数**：因为没有 `const` 的限制，函数内部可能会不小心修改传入的参数。

使用函数模版需要注意的问题：

1. 函数模板中的每一个类型参数在函数参数表中必须至少使用一次。（**定义了就一定要用**）

```
template<class T1, class T2>
```

```
void func(T1 t){}
```

函数模板声明了两个参数 `T1` 与 `T2`，但在使用时只使用了 `T1`，没有使用 `T2`。

2. 全局作用域中声明的与模板参数同名的对象、函数或类型，在函数模板中将被隐藏，例如：

```
int num;
```

```
template<class T>
```

```
void func(T t){ T num; }
```

在函数体内访问 `num` 是访问的 `T` 类型的 `num`，而不是全局变量 `num`。因为**局部变量优先**。

3. 函数模板中**声明的对象或类型不能与模板参数同名**，例如：

```
template<class T>
```

```
void func(T t){
```

```
    typedef float T;           //错误，定义的类型与模板参数名相同
```

```
}
```

4. 模板参数名在**同一模板参数列表中只能使用一次**，但可在多个函数模板声明或定义之间重复使用。
例如：

```
template <class T, class T>      //错误，在同一个模板中重复定义模板参数
```

```
void func1(T t1, T t2){}
```

```
template <class T>
```

```
void func2(T t1){}
```

```
template <class T>              //在不同函数模板中可重复使用相同模板参数名
```

```
void func3(T t1){}
```

5. 模板的定义和**多处声明**所使用的模板参数名**不一定要必须相同**，例如：

```
//模板的前向声明
```

```
template <class T>
```

```
void func1(T t1, T t2);
```

```
//模板的定义
```

```
template<class U>
```

```
void func1(U t1, U t2){}
```

6. 函数模板如果有多个模板参数，则每个模板类型前**都必须使用关键字**`class`或`typename`修饰，例如：

```
template <class T, class U>      //两个关键字可以混用
```

```
void func(T t, U u){}
```

```
template<class T, U>            //错误，每一个模板参数前都要有关键字修饰
```

```
void func(T t, U u){}
```

函数模版的注意事项要比考贵系都要繁琐呢（笑）

10.3 类模板

类也可以像函数一样被不同的类型参数化。

10.3.1 类模版定义与实例化

函数可以定义模板，对于类来说，也可以定义一个类模板，类模板是针对成员数据类型不同的类的抽象，它不是一个具体的实际的类，而是一个类型的类，一个类模板可以生成多种具体的类，定义类模板的格式如下所示：

```
template<class 类型占位符>
```

```
class 类名
```

```
{
```

```
}
```

类模版中的关键字含义与函数模版相同。类模板定义时必须有模板形参，就像给通用模具留个“空位”，有了它之后，在类里面要指定数据类型的地方，都能用这个“空位”代表的模板形参名来声明成员变量和函数。定义如下：

```
template<class T>
```

```
class A
```

```
{
```

```
public:
```

```
    T a;
```

```
    T b;
```

```
    T func(T a, T b);
```

```
};
```

定义了类模板后就要使用类模板创建对象以及实现类中的成员函数，这个过程其实也是类模板实例化的过程，实例化出的具体类称为模板类。

如果用类模板创建类的对象，例如，用上述定义类模板A创建对象，则在类模板 A 后面加上一个 `< >`，并在里面表明相应的类型。

```
A<int> a;
```

当类模板有两个模板形参时，创建对象时，类型之间要用逗号分隔开。

```
template<class T1, class T2>
```

```
class B
```

```
{
```

```
public:
```

```
    T1 a;
```

```
    T2 b;
```

```
    T1 func(T1 a, T2& b);
```

```
};
```

```
B<int,string> b;
```

使用类模板时，必须要为模板形参显式指定实参，不存在实参推演过程（*即谁叫谁得提前写出来*），也就是说不存在将整型值 10 推演为 int 类型传递给模板形参，必须要在 `< >` 中指定 int 类型，这一点与函数模板不同。

10.3.2 类模板的派生

类模板和普通类一样也可以继承和派生，以实现代码的复用。一般有三种情况：类模板派生普通类、类模板派生类模板、普通类派生类模板。

1. 类模板派生普通类

在派生过程中，类模板先实例化出一个模板类，这个模板类作为基类派生出普通类。

```

1  #include <iostream>
2  using namespace std;
3
4  // 定义一个类模板
5  template <typename T>
6  class BaseTemplate {
7  public:
8      T value;
9      BaseTemplate(T v) : value(v) {}
10     void printValue() {
11         cout << "Value: " << value << endl;
12     }
13 };
14
15 // 从类模板派生普通类
16 class DerivedClass : public BaseTemplate<int> {
17 public:
18     DerivedClass(int v) : BaseTemplate<int>(v) {}
19 };
20
21 int main() {
22     DerivedClass obj(10);
23     obj.printValue();
24     return 0;
25 }

```

`class DerivedClass : public BaseTemplate<int>` 注意这里，类模板实例化了模板类 `<int>`

`DerivedClass(int v) : BaseTemplate<int>(v)` 这里是 `DerivedClass` 类的构造函数，该构造函数接收一个整数类型的参数 `v`，并使用这个参数来初始化其基类 `BaseTemplate<int>` 的成员。

在派生过程中需要指定模版参数类型。

2. 类模板派生类模板

类模板也可以派生出一个新的类模板，它和普通类之间的派生几乎完全相同。但是，派生类模板的模版参数受基类模板的模版参数影响。

```
1  #include <iostream>
2  using namespace std;
3
4  // 定义一个类模板
5  template <typename T>
6  class BaseTemplate {
7  public:
8      T value;
9      BaseTemplate(T v) : value(v) {}
10     void printValue() {
11         cout << "Value: " << value << endl;
12     }
13 };
14
15 // 从类模板派生类模板
16 template <typename U>
17 class DerivedTemplate : public BaseTemplate<U> {
18 public:
19     DerivedTemplate(U v) : BaseTemplate<U>(v) {}
20     void printDerivedValue() {
21         cout << "Derived Value: " << this->value << endl;
22         //this->value 表示访问当前对象的 value 成员变量
23     }
24 };
25
26 int main() {
27     DerivedTemplate<double> obj(3.14);
28     obj.printValue();
29     obj.printDerivedValue();
30     return 0;
31 }
```

3. 普通类派生类模版

普通类派生类模板可以把现存类库中的类转换为通用的类模板，但是在实际编程中，这种派生方式并不常见，各位只需要了解即可。

```
1  #include <iostream>
2  using namespace std;
3
4  // 定义一个普通类
5  class BaseClass {
6  public:
7      int baseValue;
8      BaseClass(int v) : baseValue(v) {}
9      void printBaseValue() {
10         cout << "Base Value: " << baseValue << endl;
11     }
12 };
13
14 // 从普通类派生类模板
15 template <typename T>
16 class DerivedFromBaseClass : public BaseClass {
17 public:
18     T derivedValue;
19     DerivedFromBaseClass(int bv, T dv) : BaseClass(bv), derivedValue(dv) {}
20     void printDerivedFromBaseValue() {
21         cout << "Base Value: " << this->baseValue << ", Derived Value: " <<
22         < derivedValue << endl;
23     }
24 };
25 int main() {
26     DerivedFromBaseClass<char> obj(10, 'A');
27     obj.printBaseValue();
28     obj.printDerivedFromBaseValue();
29     return 0;
30 }
```

10.3.3 类模板与友元函数

在类模板中声明友元函数有三种情况：非模版友元函数、约束模版友元函数和非约束模版友元函数。

1. 非模版友元函数

非模板友元就是在类模板中声明普通的友元函数，例如，在一个类模板中声明一个友元函数：

```
template <class T>
class A{
    T _t;
public:
    friend void func();
};
```

在类模板A中声明了一个普通的友元函数 `func()`，则 `func()` 函数是类模板 A 所有实例（**模板类**）的友元函数，它可以访问全局对象，也可以使用全局指针访问非全局对象；可以创建自己的对象，也可以访问独立于对象的模板的静态数据成员。

也可以为类模板的友元函数提供模板类参数，示例代码如下：

```
template <class T>
class A
{
    T _t;
public:
    friend void show(const A<T>& a); // A<T> 类名—>数据类型
};
```

在上述代码中，`show()` 函数并不是模板函数，而**只是使用一个模板作参数**，这就要求在使用友元函数时**必须要显式具体化**，指明友元函数要引用的参数的类型，例如：

```
void show(const A<int>& a);
void show(const A<double>& a);
```

也就是说模板形参为 `int` 类型的 `show()` 函数是 `A<int>` 类的友元函数，模板形参为 `double` 类型的 `show()` 函数是 `A<double>` 类的友元函数。

复习：怎么使用友元函数？

```
1 class MyClass {
2     private:
3         int privateData;
4     public:
5         // 声明友元函数
6         friend void friendFunction(MyClass obj);
7 };
8
9 // 友元函数定义在类外部
10 void friendFunction(MyClass obj) {
11     // 能访问类的私有成员，但不是类的成员函数
12 }
```

下面是简单的例子

```
1 #include <iostream>
2 using namespace std;
3
4 class MyClass {
5     private:
6         int value;
7     public:
8         MyClass(int v) : value(v) {}
9         // 声明非模板友元函数
10        friend void printValue(MyClass obj);
11 };
12
13 // 非模板友元函数的定义
14 void printValue(MyClass obj) {
15     cout << "Value: " << obj.value << endl;
16 }
17
18 int main() {
19     MyClass obj(10);
20     printValue(obj);
21     return 0;
22 }
```

2. 约束模版友元函数

约束模板友元函数本身就是一个模板，但它的实例化类型取决于类被实例化时的类型，类实例化时会产生匹配的具体化友元函数。（**又是不讲人话**）

改写一下：

有一个友元函数本身是个模板函数，它的具体实例化类型由所在类的实例化类型来决定。当类被实例化时，会自动生成与之匹配的特定友元函数版本。

在使用约束模版友元函数时，**首先需要再类模板定义的前面声明函数模版。**

```
template<class T>
```

```
void func();
```

```
template<class T>
```

```
void show(T &t);
```

下面是简单例子

```

1  #include <iostream>
2  using namespace std;
3
4  // 前置声明类模板
5  template <typename T>
6  class MyTemplateClass;
7
8  // 前置声明模板友元函数
9  template <typename T>
10 void printTemplateValue(MyTemplateClass<T> obj);
11
12 // 定义类模板
13 template <typename T>
14 class MyTemplateClass {
15 private:
16     T value;
17 public:
18     MyTemplateClass(T v) : value(v) {}
19     // 声明约束模板友元函数
20     friend void printTemplateValue<T>(MyTemplateClass<T> obj);
21 };
22
23 // 定义模板友元函数
24 template <typename T>
25 void printTemplateValue(MyTemplateClass<T> obj) {
26     cout << "Template Value: " << obj.value << endl;
27 }
28
29 int main() {
30     MyTemplateClass<int> obj(20);
31     printTemplateValue(obj);
32     return 0;
33 }

```

`friend void printTemplateValue<T>(MyTemplateClass<T> obj);` 中参数列表 `MyTemplateClass<T> obj` 的意思是 `printTemplateValue` 函数接收一个 `MyTemplateClass` 类的对象作为参数，并且这个对象的类型参数和函数的模板参数 `T` 是一致的。

3. 非约束模版友元函数

非约束模版友元函数是将函数模版声明为类模版的友元函数，但函数模版的模版参数不受类模板影响，即友元函数模版的模版参数与类模版的模版参数是不同的。（这就是防自学教材吗（笑））

改写一下：

非约束模板友元函数指的是，把函数模板声明成类模板的友元函数，此时函数模板的模板参数不会受类模板的影响，二者的模板参数相互独立。

下面是简单的例子：

```
1  #include <iostream>
2  using namespace std;
3
4  // 定义类模板
5  template <typename T>
6  class MyNonConstrainedTemplateClass {
7  private:
8      T value;
9  public:
10     MyNonConstrainedTemplateClass(T v) : value(v) {}
11     // 声明非约束模板友元函数
12     template <typename U>
13     friend void printNonConstrainedValue(MyNonConstrainedTemplateClass<U>
obj);
14 };
15
16 // 定义非约束模板友元函数
17 template <typename U>
18 void printNonConstrainedValue(MyNonConstrainedTemplateClass<U> obj) {
19     cout << "Non - Constrained Template Value: " << obj.value << endl;
20 }
21
22 int main() {
23     MyNonConstrainedTemplateClass<double> obj(3.14);
24     printNonConstrainedValue(obj);
25     return 0;
26 }
```

在 `MyNonConstrainedTemplateClass` 类模板中，使用 `template <typename U> friend void printNonConstrainedValue(MyNonConstrainedTemplateClass<U> obj);` 声明了非约束模板友元函数。这意味着任何类型的 `printNonConstrainedValue` 函数都可以访问 `MyNonConstrainedTemplateClass` 的私有成员。在 `main` 函数中创建了 `MyNo`

`nConstrainedTemplateClass<double>` 的对象 `obj`，并调用 `printNonConstrainedValue` 函数输出其私有成员的值。

10.4 模版的参数

模版的参数有三种类型：类型参数、非类型参数和模板类型参数。

1. 类型参数

模板参数是由 `class` 或者 `typename` 标记，称为类型参数，类型参数是使用模板的主要目的。例如下列模板声明：

```
template<class T>
T add(T t1, T t2);
```

其中，`T` 就是一个类型形参，类型形参的名字由用户自行确定，表示的是一个未知类型，模板类型形参可以作为类型说明符用在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同。

可以为模板定义多个类型参数，也可以为类型参数指定默认值，示例代码如下所示：

```
template<class T, class U = int>    //只能在最右边初始化定义
class A{
public:
    void func(T, U);
};
```

在上述代码中，把U默认设置成为int类型，类模板类型形参和函数默认参数规则一致。

2. 非类型参数

非类型参数也就是内置类型形参，例如，定义如下模板：

```
template<class T, int a>
class A{};
```

在上述代码中，`int a` 就是非类型的模板形参，非类型模板形参为函数模板或类模板预定义一些常量，在生成模板实例时，也要求必须是常量，即整型、枚举、指针和引用。

非类型模板参数在模板所有实例中都具有相同的值，而类型模板参数在不同的模板实例中拥有不同的值。

使用非类型模板参数时，有以下几点需要注意：

- (1) 调用非类型模板形参的实参必须是常量表达式，即必须能在编译时计算出结果。
- (2) 任何局部对象、局部变量的地址都不是常量表达式，不能用作非类型模板的实参，全局指针类型、全局变量也不是常量表达式，也不能用作非类型模板的实参。
- (3) `sizeof()` 表达式结果是一个常量表达式，可以用作非类型模板的实参。
- (4) 非类型模板形参一般不用于函数模板。

(上面的看看就行了，谁是超人吗能记这么多东西。正常开发时候是可以查资料的，写不对就查呗)

3. 模版类型参数

模版类型参数就是模版的参数为另一个模版。声明如下：

```
template<class T, template<class U, class Z> class A>
class Parameter{
    A<T, T> a;
};
```

class T：这是一个普通的模板类型参数。在使用 `Parameter` 类模板时，你需要为 `T` 提供一个具体的类型，比如 `int`、`double`、自定义类等。这个类型参数将用于后续 `A<T, T>` 中对 `A` 模板类的实例化。

template<class U, class Z> class A：这是一个模板模板参数。`A` 代表一个模板类，这个模板类本身需要有两个类型参数 `U` 和 `Z`。在实例化 `Parameter` 类模板时，你需要传入一个满足该要求的模板类。

A<T, T> a;：在 `Parameter` 类内部，定义了一个名为 `a` 的成员变量，其类型为 `A<T, T>`。这意味着在使用 `Parameter` 类模板时，`a` 会根据传入的 `T` 和 `A` 进行具体的实例化。具体来说，`T` 会被同时作为 `A` 模板类的两个类型参数 `U` 和 `Z` 的实例化类型。

10.5 模版特化

特化就是将泛型的东西具体化，模版特化就是为已经有的模版参数进行具体化的制定，使得不受任何约束的模版参数受到特定约束或完成被指定。

1. 偏特化

偏特化就是模板中的模板参数没有被全部确定，需要编译器在编译时进行确定。例如，定义一个类模板，如下所示：

```
template<class T, class U>
class A{};
```

将其中一个模板形参特化为 `int` 类型，另一个参数由用户指定，示例代码如下：

```
template<class T>
class A<T, int>{};
```

2. 全特化

全特化就是模板中的模板参数全部被指定为确定的类型，其标志就是产生出完全确定的东西。例如，有类模板定义，如下所示：

```
template<class T>
class Compare{
public:
    bool isEqual(const T& var1, const T& var2){
        return var1==var2;
    }
};
```

将其特化为对 `float` 类型数据的比较，定义如下所示：

```
template<>
class Compare<float>{
```

```
public:
```

```
    bool IsEqual(const float& var1,const float& var2){
```

```
        return abs(var1-var2)<10e-3;
```

```
    }
```

```
};
```

上述定义中 `template<>` 就是告诉编译器这是一个特化的模板，模板特化之后就可以处理特殊情况，需要注意的是函数模板支持全特化。