第八讲 模板

模板的概念

模板（template）是 C++语言的重要特征，它能够显著提高编程效率。 C++语言引入模板技术，它使用参数化的类型创建相应的函数和类，分别称之为函数模板和类模板。

函数模板是参数化的通用函数。 类模板是能根据不同参数建立不同类型成员的类。类模板中的数据成员、成员函数的参数、成员函数的返回值可以取不同类型，在实例化成对象时，根据传入的参数类型，实例化成具体类型的对象。类模板也称模板类。

C++中的模板并不是一个实实在在的的函数或类，它们仅仅是逻辑功能相同而类型不同的函数和类的一种抽象，是参数化的函数和类。

利用 C++的函数模板和类模板，能够快速建立具有类型安全的类库集合和函数集合，进行大规模软件开发，并提高软件的通用性和灵活性。

使用函数模板进行两个数相加（隐式实例化）

```
#include<iostream>
#include <iomanip>
using namespace std;
//加法的函数模板
template <typename T>
T add(T x, T y)
{
    return x + y;
}
//测试代码
int main()
{
    cout << " sum = " << add(5, 8) << endl;
    cout << " sum = " << fixed << setprecision(2) << add(4.0, 9.0)
<< endl;
    double a = 0.0, b = 12.0;
    cout << " sum = " << add(a, b) << endl;
}

// 结果:
```

```
//  sum = 13
//  sum = 13.00
//  sum = 12.00
```

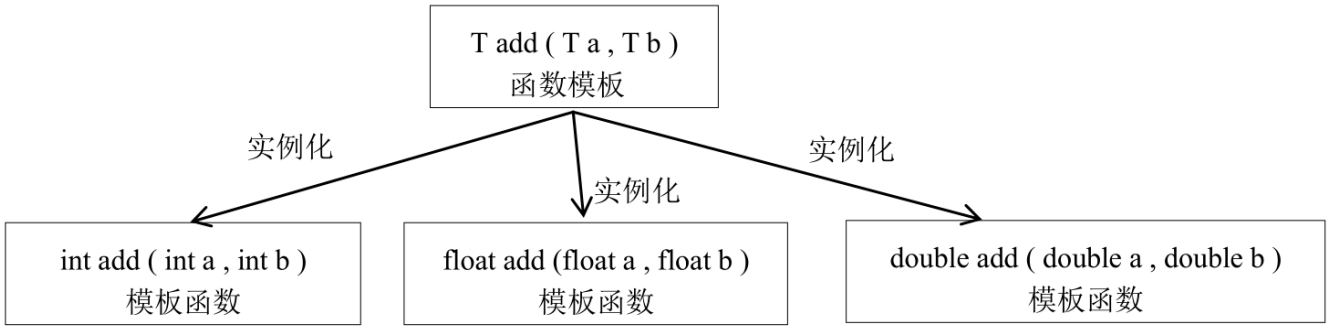
函数模板

可以用来创建一个通用功能的函数，以支持多种不同形参，进一步简化重载函数的函数体设计。

```
template <typename 类型名1, typename 类型名2, ...>
返回类型 函数名 (形参表)
{
    函数体;
}
```

意义：
对于功能完全一样，只是参数类型不同的函数，能写一段通用代码适用于多种不同的数据类型，会使代码的可重用性大大提高，从而提高软件的开发效率。

➢ template 关键字表示声明的是模板。
➢ < >中是模板的参数表，可以有一项或多项，其中的类型名称为参数化类型，是一种抽象类型或可变类型。
➢ typename 是类型关键字，也可以用 class 作为关键字。
➢ 函数返回值类型可以是普通类型，也可以是模板参数表中指定的类型。
➢ 模板参数表中的参数类型可以是普通类型。



函数模板定义后，就可以用它生成各种具体的函数（称为模板函数）。在函数调用时，用函数模板生成模板函数实际上就是将模板参数表中的参数化类型根据实参实例化（具体化）成具体类型。这个过程称为模板的实例化。

函数模板的实例化

函数模板实例化分为 隐式实例化（explicit instantiation）与 显式实例化（implicit instantiation）

隐式实例化

隐式实例化的格式为函数调用式，实例化过程是在实参与形参结合时，用实参的类型实例化形参对应的参数化类型。

优点： 节省代码

缺点： 每次调用不管类型是否相同，都会再次进行类型推演，当调用比较频繁时，会降低代码运行效率

显式实例化

显式实例化将模板参数表中的参数化类型一一实例化完成，但无须将整个函数体显式地重写，只重写声明语句即可

`template 函数返回类型 函数模板名 <实际类型列表>(实参类型列表);` 实例化： `template int add<int> (int, int);`

```
#include <iostream>
#include <iomanip>
using namespace std;
//加法的函数模板
template <typename T>
T add(T x, T y)
{
    return x + y;
}
//显式实例化方法1
template int add<int>(int, int);
//测试代码
int main()
{
    cout << " sum = " << add(5, 8) << endl;
    float c = 4.0, d = 9.0;
    //显示实例化方法2，直接用常量值4.0、9.0替换c、d也可以
    cout << " sum = " << fixed << setprecision(2) << add<float>(c,
d) << endl;
    double a = 0.0, b = 12.0;
```

```
    cout << " sum = " << add(a, b) << endl; //隐式实例化
    return 0;
}

// 结果:
//  sum = 13
//  sum = 13.00
//  sum = 12.00
```

如果函数模板的参数类型列表中有普通类型的形参，则实例化时要给出常量值

如:

```
template <typename T, int n>
T func( ) { … }
// 实例化
func <int , 100>();
```

函数模板与函数重载

模板函数与重载函数调用示例代码

```
#include <iostream>
using namespace std;
//加法的函数模板
template <typename T>
T add(T x, T y)
{
    cout << " 模板函数被调用 " << endl;
    return x + y;
}
//加法的重载函数
int add(int x, int y)
{
    cout << " 重载函数被调用 " << endl;
    return x + y;
}
//测试代码
int main()
{
```

```
    cout << add(5, 8) << endl;
    return 0;
}

// 结果:
// 重载函数被调用
// 13
```

C++对重载函数的绑定是遵循最佳匹配优先规则的:

1. 精确匹配优先；若同时精确匹配，则普通函数优先于模板函数。
2. 提升转换：向高类型转换，如 char、short 转换为 int ，float 转换为 double。
3. 标准转换：向低类型或相容类型转换，如 int 转换为 char，long 转换为 double。
4. 用户自定义的转换：如类生命中定义的转换。绑定顺序可以简单表述为：精确匹配 > 提升转换后匹配 > 标准转换后匹配 > 自定义转换

函数模板的具体化

函数模板技术是为了便于类型不同的数据共享函数结构，避免多次重写相似的函数体，函数模板的实例化可以生成数据类型不同的模板函数；而函数模板的具体化其实相当于在函数模板的基础上，添加一个专门针对特定类型的、实现方式不同的一个函数模板的特例。
函数模板的具体化有两种方式，第一种是显式具体化：函数模板显式具体化是为了将接口相同，实现不同的函数区分出来而提供的技术。函数模板显式具体化的格式如下:

```
template <> 函数返回类型 函数模板名<实际类型列表>(实参类型列表)
{
    函数体;
}
```

显式具体化与显式实例化的本质区别在于显式实例化不需要重写函数体，而显式具体化必须要重写函数体，因为具体化的目的就是提供与模板不同的函数实现。
例子:

```
#include <iostream>
using namespace std;
//交换函数的模板
template <typename T>
```

```
void swap(T &a, T &b)
{
    T t;
    t = a;
    a = b;
    b = t;
}
struct Info
{
    char ID[18];
    char NO;
    bool flag;
};
template <>
void swap<Info>(Info &a, Info &b)
{
    bool bt;
    bt = a.flag;
    a.flag = b.flag;
    b.flag = bt;
}
void swap(Info &a, Info &b)
{
    bool bt;
    bt = a.flag;
    a.flag = b.flag;
    b.flag = bt;
}
```

函数模板的重载

函数模板同样可以按照普通函数重载的规则进行重载，比如：

```
template <typename T>
T add(T &a, T &b)
{
    return a + b;
}
template <typename T>
T add(T &a, T &b, T &c)
```

```
{
    return a + b + c;
}
```

类模板

类模板的定义

类模板定义的语法为：

```
template <模板参数表>
class 类名
{
成员名;
}
```

➢ template 为模板关键字。

➢ 模板参数表中的类型为参数化(parameterized)类型，也称可变类型，类型名为 class (或 typename)；模板参数表中的类型也可包含普通类型，普通类型的参数用来为类的成员提供初值。

➢ 类模板中的成员函数可以是函数模板，也可以是普通函数。

类模板的成员函数可以在类内部定义，也可以在类外部定义，若在类外部定义，则其语法如下：

```
template <模板参数表>
类型 类名 <模板参数名表>::函数名(参数表)
{
    函数体;
}
```

➢ 模板参数表与类模板的模板参数表相同。

➢ 模板参数名表列出的是模板参数表中参数名，顺序与模板参数表中的顺序一致。

类模板的实例化

一个类模板是具体类的抽象，在使用类模板建立对象时，才根据给定的模板参数值实例化(专门化)成具体的类，然后由类建立对象。与函数模板不同，类模板实例化为模板类只能采用显式方式。

类模板实例化、建立对象的语法如下:

`类模板名<模板参数值表>对象列表;`

➢ 模板参数值表的值为类型名，类型名可以是基本数据类型名，也可以是构造数据类型名，还可以是类类型名。

➢模板参数值表的值还可以是常数表达式，以初始化模板参数表中普通参数。

➢模板参数值表的值按一一对应的顺序实例化类模板的模板参数表。

```
#include <iostream>
using namespace std;
//类模板
template <typename T1, typename T2, int T3>
class A
{
private:
    int a;      //私有成员
    T1 Arr[T3]; //私有数组，T1类型的数组，数组大小为T3
public:
    T2 Func(T2 & x, T2 & y); //公有成员函数
};
//成员函数外部定义
template <typename T1, typename T2, int T3>
T2 A<T1, T2, T3>::Func(T2 &x, T2 &y) //定义格式
{
    x++;
    y++;
    return x + y;
}
int main()
{
    A<char, int, 10> obj1, obj2; //定义两个A类对象
    int a, b, w;
    cin >> a >> b;
    w = obj1.Func(a, b);
    cout << w << endl;
}
```

```
// 结果:
// 2 3
// 7
```

类模板与静态成员

类模板中同样可以定义静态成员，定义方法也是再成员类型前加 static 关键字

```
#include <iostream>
using namespace std ;
//类模板
template < typename T >
class A
{
..... //其他成员
public:
    static int count ; //整型的静态成员count
    static T visitnum ; //T型的静态成员
} ;
```

对确定类型为整型的静态成员 count，有两种初始化的方式：

- 第一种是具体化的定义，即指定 T 类型的静态成员初值；
  - `template <> int A<int >:: count = 100 ;`
- 第二种是范化的定义，定义时不指定 T 的类型

  - `template < typename T > int A< T > :: count = 200 ;`

  用第二种初始化方式时，若类模板实例化为整型，则所有整型对象共享同一个 count，初值为 200；若实例化为 float 型，则所有 float 对象共享一个 count，初值为 200.0。

  而对未明确类型的静态成员 visitnum 来说，就必须用具体化的方式定义，例如：

  `template <> floatA<float>:: visitnum = 300.0f;`

类模板与友元

普通友元

非模板的函数、类、类的成员函数都可以作为类模板的友元，实例化后的模板类也会将这些函数、类、类的成员函数作为其友元。友元函数和友元类可以访问当前类的任意成员。

```
void f1(){...} class A{...};
class B
{
public:
    void f() { ... }
};
template <class T>
class C
{
    ……
    friend void f1();
    friend class A;
    friend void B::f();
};
```

一般模板友元关系

友元可以是类模板或函数模板，友元与类可以分别使用自己的类型参数，声明为友元后，友元函数模板的任意实例都可以访问当前类的任意实例的私有成员；友元类模板的任意实例可以访问当前类的任意实例。

```
template <class T>
class A
{
    template <class T1>
    friend class B;
    template <class T2>
    friend void Func(T &);
};
```

特定的模板友元关系

类模板中的友元声明时，可以指定特定实例的访问权，即并非接受友元类模板的所有实例。

```
template <class T>
class A;
template <class T>
void Func(T);
```

```
template <class T1>
class B
{
    friend class A<int>;
    friend void Func<char>(char);
};
```

指定 A 类型参数为 int 的实例才是友元；友元函数模板 Func 的实例中，只有类型参数为 char 的实例才是友元

```
template <class T>
class A;
template <class T>
void Func(T);
template <class T1>
class B
{
    friend class A<T1>;
    friend void Func<T1>(T1);
};
```

只有与 B 实例有相同类型的 A 和 Func 的实例才是 B 的友元