第11讲函数模板

目录 CONTENTS

什么是模板？

```
class  A
{
//…..
  int add(int x,int y)
  {   return x+y;   }
  double add(double x,double y)
  {   return x+y;   }
  float add(float x, float y)
  {   return x+y;   }
};
```

```
void main()
{
    A   op;
    cout<<op.add(1,2)<<endl;
    cout<<op.add(1.2f,2.3f)<<endl;
    cout<<op.add(1.2,2.3)<<endl;
}
```

抽象上例成员函数

这三个add()函数的主体行为是一样的，只是处理的数据类型不同而已。该例的成员函数可抽象为：

```
T add(T x, T y)      //T为模板形参
{   return x+y;   }
```

使用时T为int、float、double等。

- 模板(template)的目的是实现数据类型的参数化，即将数据类型作为参数。同样一段代码，可以处理不同数据类型的参数。
- 模板提供了将代码与数据类型相脱离的机制。
- 模板可分为函数模板和类模板两种类型，在定义函数或类时，可以将数据类型作为模板的参数，模板进一步实现了代码的重用。

11.1 函数模板的概念

- 函数模板(Function Template)是指建立一个通用函数，该函数的某些形参类型或函数返回类型不具体指定，使用类型参数标识符代替。
- 在调用函数模板时，会根据实参的类型来取代函数模板中的类型参数标识符，从而生成实际的函数。
- 函数模板的实例化由编译器实现。

【例】 通过函数模板交换数据

```
template <typename T>
//也可以写成: template <class T>
void Swap(T& a, T& b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

模板的定义以关键字template开头，关键字typename (或class)表示后面的标识符（这里是T）是模板参数(数据类型参数)。

本函数模板中，变量a和b的数据类型在定义函数模板时并没有确定。

```
int main()
{
    int a = 10, b=20;
    cout<<“a=”<<a<<“,b=“<<b<<endl;
    Swap(a, b);
    cout<<“a=”<<a<<“,b=“<<b<<endl;

    double x = 12.3, y = 45.6;
    cout<<“a=”<<a<<“,b=“<<b<<endl;
    Swap(x, y);
    cout<<“a=”<<a<<“,b=“<<b<<endl;
}
```

11.2 使用方法

（1）这些在调用函数模板时生成的函数模板实例称为模板函数，此实例化的过程称为函数模板的实例化。

11.2 使用方法

（2）函数模板在实例化时，编译器根据实际参数的数据类型进行函数模板类型参数的替换，生成一个具体函数，然后再对该函数进行编译。

（3）在template语句和函数模板定义语句之间不能有其他语句。例如：下面的代码会出现编译错误。

```
template <typename T>
int j;
T min(T x, T y)
{
    return (x<y) ? x : y ;
}
```

错误：不允许在template语句和函数模板定义语句之间插入别的语句

11.2 使用方法

（4）函数模板也可以重载。

```
template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for(int i=0;i<n;i++){
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
```

```
int main()
{
    int a = 10, b=20;
    cout<<“a=”<<a<<“,b=“<<b<<endl;
    Swap(a, b);
    cout<<“a=”<<a<<“,b=“<<b<<endl;

    int x[] = {1,2,3,4,5};
    int y[] = {6,7,8,9,10}
    Swap(x, y, 5);
    //显示
}
```

11.2 使用方法

（5）显示具体化

```
class Person
{
public:
    string name;
    double salary;
public:
    Person(string n, double s)
    {
        name = n;
        salary = s;
    }
    Person()
    {
    }
};
```

```
int main()
{
    string n1 = “Tom";
    string n2 = “Sally";
    Person p1(n1, 10000);
    Person p2(n2, 20000);
    Swap(p1, p2);
    cout<<p1.name<<”,”<<p1.salary<<endl;
}
```

姓名和薪水都交换了，如果只想交换薪水，怎么办？

11.2 使用方法

对于模板，模板中的语句（函数体或者类）不一定能适应所有的类型，可能会有个别的类型没有意义，不能满足需求，或者会导致语法错误。希望模板能够针对某种具体的类型使用不同的算法，可以使用显式具体化。

template<> 返回值 函数名(参数);

```
template<> void Swap(Person& p1, Person& p2)
{
    double temp;
    temp = p1.salary;
    p1.salary = p2.salary;
    p2.salary = temp;
}
```

```
int main()
{
    string n1 = “Tom";
    string n2 = “Sally";
    Person p1(n1, 10000);
    Person p2(n2, 20000);
    Swap(p1, p2);
    cout<<p1.name<<”,”<<p1.salary<<endl;
}
```

只会交换薪水

11.2 使用方法

（6）定义函数模板时可以使用多个类型参数，其形式如下所示。

template <typename T1, typename T2, typename T3>

或

template <class T1, class T2, class T3>

但是对于多参数函数模板，编译器无法自动推导返回值类型，

【例2】 使用多个类型参数的函数模板

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
void fun(T1 x, T2 y)
{
    cout<<x<<"  "<<y<<endl;
}
int main()
{
    int j = 200;
    char str1[20] = "Hello";
    double d = 123.45;
    char c = 'a';
    fun(j, str1);
    fun(d, c);
    return 0;
}
```

程序的运行结果如下：
200 Hello
123.45 a

类型参数T1和T2被分别替换成了int和char*

类型参数T1和T2被分别替换成了double和char

【例2】 使用多个类型参数的函数模板

```
#include <iostream>
using namespace std;
template <typename T1, typename T2, typename T3>
T1 fun(T2 x, T3 y)
{
    cout<<x<<"  "<<y<<endl; T1 z; return z;
}
int main()
{
    int j = 200;
    char str1[20] = "Hello";
    double d = 123.45;
    char c = 'a';
    fun<double>(j, str1);
    fun<int>(d, c);
    return 0;
}
```

当函数模板的返回值是模板类型参数时，编译器无法通过函数调用来推断返回值的具体类型。此时，在调用函数时必须提供一个显式模板实参。

其中double表示返回值类型，其他两个参数可推断。

单选题 1分

设置

此题未设置答案，请点击右侧设置按钮

下面的模板声明，哪个是正确的？

A
```
template <typename T>
T max ( T x , y )
{ return (x>y?x：y) ;}
```

B
```
template < typename T>
T max ( T x , T y )
{ return (x>y?x：y) ;}
```

C
```
template <typename T>
int i ;
T max ( T x , T y )
{ return (x>y?x：y) ;}
```

D
```
template < typename T>
T max ( T1 x , T2 y )
{ return (x>y?x：y) ;}
```

提交

11.2 使用方法

（7）模板函数的调用，参数必须符合要求。

```
template <class T>          //函数模板的创建
T  mymax (T x, T y)         // AT为模板形参
{   return( x>y)?x:y ;    }
void main( )
{      int i1=10, i2=56;
       float f1=12.5,f2=24.5;
       double d1=50.344,d2=4.888;
       char c1='k', c2='n';
       cout<<mymax(i1,i2)<<endl;          //模板函数的创建
       cout<<mymax(f1,f2)<<endl;          //模板函数的创建
       cout<<mymax(d1,d2)<<endl;          //模板函数的创建
       cout<<mymax(c1,c2)<<endl;          //模板函数的创建
       cout<<mymax(i1,d2)<<endl;           //错误
       cout<<mymax(c1,d2)<<endl;           //错误
       cout<<mymax(i1 , (int)d2)<<endl;
       cout<<mymax(c1 , (char)d2)<<endl;
}
```

11.2 使用方法

重载解析：决定函数调用使用哪个函数定义。

函数模板与同名非模板函数重载时执行顺序为：

- （1）参数完全匹配的函数优先
- （2）显示具体化
- （3）函数模板实例化
- （4）低一级的函数重载方法：例如通过类型转换可产生参数匹配等
- （5）可以通过<>限定只使用函数模板

```
template <class T>
T  mymax (T x, T y)
{   return( x>y)?x:y ;  }
int mymax (int, int ) ;
 void main()
{
   cout<<mymax(10,22)<<endl;              //调用mymax (int, int )
   cout<<mymax<>(10,22)<<endl;            //强制调用模板函数
   cout<<mymax("hello","happy")<<endl; //调用模板函数
   cout<<mymax('f','c')<<endl;            //调用模板函数
   cout<<mymax(25.3,99.8)<<endl;          //调用模板函数
   cout<<mymax(5,5.5)<<endl;              //调用mymax (int, int )
   cout<<mymax('c',4)<<endl;              //调用mymax (int, int )
}
```

```
template <class T>
T  mymax (T x, T y)
{   return( x>y)?x:y ;  }
int mymax (int, int ) ;
double mymax (double, double ) ;
char mymax (char, char ) ;
 void main()
{
   cout<<mymax(10,22)<<endl;                          //调用mymax (int, int )
   cout<<mymax("hello","happy")<<endl;                //调用模板函数
   cout<<mymax('f','c')<<endl;                        //调用mymax (char, char )
   cout<<mymax(25.3,99.8)<<endl;                      //调用mymax (double, double )
   cout<<mymax(5,5.5)<<endl;                          //错误
   cout<<mymax('c',4)<<endl;                          //错误
}
```

当重载函数有多个参数时也会产生二义性，而且情况更加复杂。C++标准规定，如果有且只有一个函数满足下列条件，则匹配成功：

(1) 该函数对每个实参的匹配都不劣于其他函数；

(2) 至少有一个实参的匹配优于其他函数。

11.2 使用方法

C++ 标准规定，在进行重载决议时编译器应该按照下面的优先级顺序来处理实参的类型：

优先级	包含的内容	举例说明
精确匹配	不做类型转换，直接匹配	(暂无说明)
只是做微不足道的转换	从数组名到数组指针、从函数名到指向函数的指针、从非 const 类型到 const 类型。	
类型提升后匹配	整型提升	从 bool、char、short 提升为 int，或者从 char16_t、char32_t、wchar_t 提升为 int、long、long long。
小数提升	从 float 提升为 double。	
使用自动类型转换后匹配	整型转换	从 char 到 long、short 到 long、int 到 short、long 到 char。
小数转换	从 double 到 float。	
整数和小数转换	从 int 到 double、short 到 float、float 到 int、double 到 long。	
指针转换	从 int * 到 void *。	

11.2 使用方法

```
void f(char, int); //1
void f(char, int, float); //2
void f(char, long, double); //3

short s = 10;
f('A', s, 100);     //调用哪个函数？2
f('A', s, 100.0);   //调用哪个函数？错误
```

11.2 使用方法

（9）函数模板还可以应用在指针或数组。

```
template  <class T>
T sum (T *array, int size=0){
    T  total =0;
    for  (int i=0; i<size; i++)
          total+=array[i];
    return  total;
}
int i_array [ ]={ 1,2,3,4,5,6,7,8,9,10};
double d_array[ ]= { 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.10};
void main( )
{
     int itotal=sum<int> (i_array,10);
     double dtotal=sum<double> (d _array,10);
}
```

```
template  <class T, int size>
T sum (T *array){
    T  total =0;
    for  (int i=0; i<size; i++)
          total+=array[i];
    return  total;
}
int i_array [ ]={ 1,2,3,4,5,6,7,8,9,10};
double d_array[ ]= { 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.10};
void main( )
{
     int itotal=sum<int, 10> ( i_array);
     double dtotal=sum<double,10> (d _array);
}
```

size是非类型参数

11.2 使用方法

- 非类型形参在模板定义的内部是常量值，也就是说非类型形参在模板的内部是常量。
- 非类型模板的形参只能是整型，指针和引用，像double, string, string **这样的类型是不允许的。但是double &, double *，对象的引用或指针是正确的。
- 调用非类型模板形参的实参必须是一个常量表达式，即他必须能在编译时计算出结果。
- 任何局部对象，局部变量，局部对象的地址，局部变量的地址都不是一个常量表达式，都不能用作非类型模板形参的实参。

11.2 使用方法

- 全局指针类型，全局变量，全局对象也不是一个常量表达式，不能用作非类型模板形参的实参。
- 全局变量的地址或引用，全局对象的地址或引用const类型变量是常量表达式，可以用作非类型模板形参的实参。

```
void main( )
{
    int n = 10;
    const int m = 10;
    int itotal=sum<int, n> ( i_array); //错误
    double dtotal=sum<double,m> (d _array);  //正确
}
```