第3章 模　板

使用模板可以建立具有通用类型的函数库或类库，为一系列逻辑功能相同而数据类型不同的函数或类创建框架，模板提供了一种重用程序源代码的有效方法，方便了大规模的软件开发。

3.1 模板的概念

模板的本质就是将所处理的数据类型说明为参数，模板是对具有相同特性的函数或类的再抽象，它将程序所处理的数据的类型参数化，这样可使一段程序代码能用于处理多种不同类型的数据。C++ 程序由类和函数组成，类对应类模板，函数对应函数模板。

为更好地理解，我们来考察三个 Swap()函数，分别用于交换两个整型数、两个浮点实型数以及两个双精度实型数。这三个 Swap()函数的功能完全一样，只是所处理的数据的类型不同，下面是这三个函数的具体实现：

```
void Swap(int &x, int &y)                    //交换整型数 x、y
{
    int temp=x; x=y; y=temp;                 //通过循环赋值交换 x、y
}

void Swap(float &x, float &y)                //交换浮点实型数 x、y
{
    float temp=x; x=y; y=temp;               //通过循环赋值交换 x、y
}

void Swap(double &x, double &y)              //交换双精度实型数 x、y
{
    double temp=x; x=y; y=temp;              //通过循环赋值交换 x、y
}
```

上面通过函数重载实现了让函数 Swap()处理不同类型的数据，但不能处理任意类型的数据，要求交换任何一对同一种类型的数据。最好的解决方法是类型参数化，这样就得到了函数模板。使用函数模板定义如下：

```
template<class ElemType>                     //此处的 class 不是定义类的标识，只表明
                                             //ElemType 是一个类型参数
void Swap(ElemType &x, ElemType &y)          //交换 x、y
{
    ElemType temp=x; x=y; y=temp;            //通过循环赋值交换 x、y
}
```

这样得到可以将任一类型 ElemType 的两个数据进行互换的函数模板。

下面的三个类 Integer、Float 和 Double 分别用来处理整型数、浮点型实数以及双精度实型数。这三种类的处理功能完全一样，只是所处理的数据的类型不同，下面是这三个类的声明：

```
//声明整型类
class Integer
{
private:
//数据成员
    int num;                                    //数据值

public:
//公有函数
    Integer(int n=0): num(n){ }                 //构造函数
    void Set(int n) { num=n; }                  //设置数据值
    int Get() const { return num; }             //返回数据值
};

//声明浮点型实数类
class Float
{
private:
//数据成员
    float num;                                  //数据值

public:
//公有函数
    Float(float n=0): num(n){ }                 //构造函数
    void Set(float n) { num=n; }                //设置数据值
    float Get() const { return num; }           //返回数据值
};

//声明双精度实型数类
class Double
{
private:
//数据成员
    double num;                                 //数据值

public:
//公有函数
    Double(double n=0): num(n){ }               //构造函数
    void Set(double n) { num=n; }               //设置数据值
    double Get() const { return num; }          //返回数据值
```

```
};
```

上面实现的三个类的功能相同，处理不同类型的数据，还是不能处理任意类型的数据，但采用类型参数化后就可以处理任何类型，这样就得到了类模板。使用类模板定义如下：

```
template<class ElemType>
class Number
{
private:
//数据成员
    ElemType num;                                    //数据值

public:
//公有函数
    Number(ElemType n=0): num(n){}                   //构造函数
    void Set(ElemType n) { num=n; }                  //设置数据值
    ElemType Get() const { return num; }             //返回数据值
};
```

注意：函数模板或类模板是对一族函数或类的描述，模板是一种由通用代码构成，使用类型参数来产生一组函数或类的机制。

3.2 函数模板及模板函数

函数模板是对一批功能相同的函数的说明，它不是某一个具体的函数，是带有“类型参数”的一种描述。模板函数是将函数模板内的“数据类型参数”取某一个具体的数据类型后得到的具体函数。

3.2.1 函数模板的声明及生成模板函数

使用函数模板的方法是先声明函数模板，然后将其类型参数具体化形成相应的模板函数，最后才可以调用模板函数。

函数模板的一般声明格式如下：

```
template<class 类型参数名 1, class 类型参数名 2, …>
返回值类型 函数名(形参表)
{
    ⋮                                                //函数体
}
```

或

```
template<typename 类型参数名 1, typename 类型参数名 2, …>
返回值类型 函数名(形参表)
{
    ⋮                                                //函数体
}
```

其中，template 是一个声明模板的关键字，class 在此处并不表示类的意思，只是借用此关键字表示其后是一个类型参数。“class 类型参数名 1，class 类型参数名 2，…”称为类型形参表，类型参数名可以用任何实际的类型（包括类类型）进行具体化（也称为实例化）得到模板函数。

class 和 typename 的作用相同，都是表示“类型名”，二者可以互换。大部分 C++ 程序员都喜欢用 class，这是因为 class 更容易输入，typename 是新加入到标准 C++ 中的，使用 typename 时，含义非常清楚。

在使用函数模板时，用实际的数据类型具体化（实例化）类型形式参数，再根据实际参数类型，生成一个具体的模板函数，模板函数的函数体与函数模板的函数体完全相同，在程序中真正执行的代码是模板函数的代码。

在使用函数模板生成模板函数时，有两种使用方式：

```
函数名(实参表)
```

或

```
函数名<类型 1, 类型 2, …>(实参表)
```

第一种使用方式将根据实参类型确定类型形式参数的具体类型，第二种方式中，<类型 1，类型 2，…>称为类型实参表，用类型实参表中的类型来确定类型形式参数具体类型。

说明：类型形参表与类型实参表通常只包含一个类型，这时函数模板的一般声明格式如下：

```
template<class 类型参数名>
返回值类型 函数名(形参表)
{
    ⋮                                          //函数体
}
```

或

```
template<typename 类型参数名>
返回值类型 函数名(形参表)
{
    ⋮                                          //函数体
}
```

使用函数模板生成模板函数的两种使用方式如下：

```
函数名(实参表)
```

或

```
函数名<类型>(实参表)
```

例 3.1 函数模板定义与模板函数的调用示例。

```
//文件路径名:e3_1\main.cpp
#include<iostream>                                    //编译预处理命令
```

```
using namespace std;                                          //使用命名空间 std

template<class ElemType>
ElemType Max(ElemType x, ElemType y)                          //求 x、y 的最大值
{
    return x<y ? y : x;                                       //返回 x、y 的最大值
}

int main()                                                    //主函数 main()
{
    cout<<"2 和 3 的最大值为"<<Max(2, 3)<<endl;                 //输出 2、3 的最大值
    //cout<<"2 和 3.0 的最大值为"<<Max(2, 3.0)<<endl;
        //错,无法根据 2 和 3.0 确定类型形式参数的具体类型
    cout<<"2 和 3.0 的最大值为"<<Max<int>(2, 3.0)<<endl;  //输出 2、3.0 的最大值

    system("PAUSE");                  //调用库函数 system( ),输出系统提示信息
    return 0;                         //返回值 0, 返回操作系统
}
```

程序运行时屏幕输出如下：

```
2 和 3 的最大值为 3
2 和 3.0 的最大值为 3
请按任意键继续...
```

从上面的例题可以看出，采用第一种形式使用函数模板生成模板函数时，同一模板类型参数的各参数之间必须保持完全一致的类型。

3.2.2 重载函数模板

模板函数类似于重载函数，但是同一个函数模板类型形式参数具体化(实例化)后的所有模板函数必须执行相同的代码，而函数重载时在每个函数体中可以执行不同的代码，当遇到执行的代码有所不同时，不能简单地套用函数模板，而应像重载普通函数那样进行重载。

重载函数模板后，编译器首先匹配类型完全相同的函数，如果匹配失败，再寻求函数模板进行匹配。

例 3.2 重载函数模板与匹配过程示例。

```
//文件路径名:e3_2\main.cpp
#include<iostream>                                    //编译预处理命令
using namespace std;                                  //使用命名空间 std

template<class ElemType>
ElemType Max(ElemType x, ElemType y)                  //求 x、y 的最大值
{
    return x<y ? y : x;                               //返回 x、y 的最大值
}
```

```
char * Max(char * str1, char * str2)                    //求 str1、str2 的最大值
{
    return strcmp(str1, str2)<0 ? str2 : str1;          //返回 str1、str2 的最大值
}

int main()                                              //主函数 main()
{
    cout<<"2 和 3 的最大值为"<<Max(2, 3)<<endl;   //输出 2、3 的最大值, 匹配函数模板
    cout<<"China 与 American 的最大值为"<<Max("China", "American")<<endl;
        //输出"China","American"的最大值, 匹配函数

    system("PAUSE");                                    //调用库函数 system( ),输出系统提示信息
    return 0;                                           //返回值 0, 返回操作系统
}
```

程序运行时屏幕输出如下：

```
2 和 3 的最大值为 3
China 与 American 的最大值为 China
请按任意键继续...
```

函数“char * Max(char * str1, char * str2)”中的函数名与函数模板的函数名相同，但操作代码不同，函数体中的比较采用了字符串比较函数，所以要用重载的方法把它们区分开来，遵循的原则是首选函数名、参数类型都匹配的函数，再找模板。本例在调用 Max(2, 3)时，由于实参是整型，与函数“char * Max(char * str1, char * str2)”的形参类型不匹配，因此只能匹配函数模板，在调用 Max("China", "American")时，实参为字符串，与函数“char * Max(char * str1, char * str2)”的形参类型相匹配，因此匹配此函数。

例 3.3 重载函数模板示例。

```
//文件路径名:e3_3\main.cpp
#include<iostream>                                      //编译预处理命令
using namespace std;                                    //使用命名空间 std

template<class ElemType>
ElemType Max(ElemType x, ElemType y)                    //求 x、y 的最大值
{
    return x<y ? y : x;                                 //返回 x、y 的最大值
}

template<class ElemType>
ElemType Max(ElemType x, ElemType y, ElemType z)        //求 x、y、z 的最大值
{
    ElemType m=x<y ? y : x;                             //m 为 x、y 的最大值
    m=m<z ? z : m;                                      //m、z 的最大值
    return m;                                           //返回最大值
}
```

```
template<class ElemType>
ElemType Max(ElemType a[], int n)                    //求 a[0],a[1],…,a[n-1]的最大值
{
    ElemType m=a[0];                                 //假设 a[0]为最大值
    for (int i=1; i<n; i++)
        if (m<a[i]) m=a[i];                          //如 a[i]更大,则将 a[i]赋值给 m
    return m;                                        //返回最大值
}

int main()                                           //主函数 main()
{
    int a[]={1, 9, 7, 5, 6, 3};                      //定义数组 a
    cout<<"数组 a 的最大元素值为"<<Max(a, 6)<<endl;   //输出数组 a 的最大元素值
    cout<<"2 和 3 的最大值为"<<Max(2, 3)<<endl;        //输出 2、3 的最大值
    cout<<"2,3 和 8 的最大值为"<<Max(2, 3, 8)<<endl;   //输出 2、3、8 的最大值

    system("PAUSE");                                 //调用库函数 system( ),输出系统提示信息
    return 0;                                        //返回值 0, 返回操作系统
}
```

程序运行时屏幕输出如下：

```
数组 a 的最大元素值为 9
2 和 3 的最大值为 3
2,3 和 8 的最大值为 8
请按任意键继续...
```

本例中根据实参的个数与类型来匹配不同的函数模板。

3.3 类模板及模板类

类模板与函数模板类似，它可以为任意数据类型定义一种模板，使用不同的数据类型具体化(实例化)类模板生成具体的模板类，模板类可以用于生成具体的对象。

3.3.1 类模板的声明及生成模板类

定义一个类模板与定义函数模板的格式类似，必须以关键字 template 开始，类模板的一般声明形式如下：

```
template<class 类型参数名 1, class 类型参数名 2, …>
class 类名
{
    ⋮                                                //类体
};
```

或

```
template<typename 类型参数名 1, typename 类型参数名 2, …>
class 类名
{
    ⋮                                    //类体
};
```

类模板的成员函数不但可以在类模板内定义，也可以在类模板外定义。在类模板内定义时，与一般类的成员函数的定义方法完全一样；在类模板外定义时，需要采用下面的形式：

```
template<class 类型参数名 1, class 类型参数名 2, …>
返回值类型 类名<类型参数名 1, 类型参数名 2, …>::成员函数名(形参表)
{
    ⋮                                    //函数体
}
```

或

```
template<typename 类型参数名 1, typename 类型参数名 2, …>
返回值类型 类名<类型参数名 1, 类型参数名 2, …>::成员函数名(形参表)
{
    ⋮                                    //函数体
}
```

其中，“class 类型参数名 1，class 类型参数名 2，…”与“typename 类型参数名 1，typename 类型参数名 2，…”为类型形参表，与函数模板中的类型形参表意义一样。

类模板必须用实际的数据类型具体化(实例化)类型形式参数，再根据实际参数类型，生成一个具体的模板类，然后才能用来生成具体对象。一般语法格式如下：

```
类模板名<类型 1, 类型 2, …>对象名;
```

例 3.4 使用类模板的示例。

```
//文件路径名:e3_4\main.cpp
#include<iostream>                                    //编译预处理命令
using namespace std;                                  //使用命名空间 std

//声明数组类模板
template<class ElemType>
class Array
{
private:
//数据成员
    ElemType *elem;                                   //存储数据元素值
    int size;                                         //数组元素个数

public:
//公有函数
    Array(int sz): size(sz) { elem=new ElemType[size]; }//构造函数
```

```
    ~ Array(){ delete elem; }                          //析构函数
    void SetElem(ElemType e, int i);                   //设置元素值
    ElemType GetElem(int i) const;                     //求元素值
};

template<class ElemType>
void Array<ElemType>::SetElem(ElemType e, int i)       //设置元素值
{
    if (i<0 || i >=size)
    {
        cout<<"元素位置错!"<<endl;
        exit(1);                                       //退出程序的运行,返回到操作系统
    }
    elem[i]=e;                                         //设置元素值为 e
}

template<class ElemType>
ElemType Array<ElemType >::GetElem(int i) const     //求元素值
{
    if (i<0 || i >=size)
    {
        cout<<"元素位置错!"<<endl;
        exit(2);                                       //退出程序的运行,返回到操作系统
    }
    return elem[i];                                    //返回元素值 elem[i]
}

int main()                                             //主函数 main()
{
    int a[]={1, 9, 7, 5, 6, 3};                        //定义数组 a
    int n=6;                                           //数组元素个数
    Array<int>obj(n);                                  //定义数组对象
    int i;                                             //定义临时变量
    for (i=0; i<n; i++)
        obj.SetElem(a[i], i);                          //设置数组元素值
    for (i=0; i<n; i++)
        cout<<obj.GetElem(i)<<" ";                     //输出元素值
    cout<<endl;                                        //换行

    system("PAUSE");                                   //调用库函数 system( ),输出系统提示信息
    return 0;                                          //返回值 0, 返回操作系统
}
```

程序运行时屏幕输出如下：

```
1 9 7 5 6 3
```

请按任意键继续...

怎样选择是将成员函数在类模板内定义还是在类模板外定义呢？作者建议对于函数体实现较简单，并且行数较少的类模板的成员函数都在类模板内定义；对于函数体实现较复杂，并且行数较多的类模板的成员函数在类模板外定义。

3.3.2 在类型形参表中包含常规参数的类模板

在声明类模板的类型形参表中还可以包含常规参数，常规参数经常是数值。对模板类进行具体化(实例化)为模板类时，给这些参数所提供的表达式必须是常量表达式。

例 3.5 使用在类型形参表中包含常规参数的示例。

```
//文件路径名:e3_5\main.cpp
#include<iostream>                                    //编译预处理命令
using namespace std;                                  //使用命名空间 std

//声明数组类模板
template<class ElemType, int size>
class Array
{
private:
//数据成员
    ElemType elem[size];                              //存储数据元素值

public:
//公有函数
    void SetElem(ElemType e, int i);                  //设置元素值
    ElemType GetElem(int i) const;                    //求元素值
};

template<class ElemType, int size>
void Array<ElemType, size>::SetElem(ElemType e, int i)     //设置元素值
{
    if (i<0 || i >=size)
    {
        cout<<"元素位置错!"<<endl;
        exit(1);                                      //退出程序的运行,返回到操作系统
    }
    elem[i]=e;                                        //设置元素值为 e
}

template<class ElemType, int size>
ElemType Array<ElemType, size>::GetElem(int i) const    //求元素值
{
    if (i<0 || i >=size)
    {
```