第 11 章　模板

本章摘要

模板是 C++支持参数化多态性的工具之一。所谓参数化多态性，就是将程序中所处理的对象类型参数化，使一段程序可以处理多种不同类型的对象。模板中以数据类型作为参数，模板本身只涉及抽象类型的对象。

使用模板可以方便地建立起通用类型的函数库和类库，减少程序开发的重复及代码冗余，为编写大型程序提供了方便。

11．1　模板的概念

C++是一种强类型程序设计语言，因此程序设计者在定义函数时，经常需要对不同的数据类型分别定义不同的重载版本。例如，编写求最大值的函数 max()如下：

```
int    max(int x,int y)
{
    return (x>y)?x:y;
}
double    max(double x,double y)
{
    return (x>y)?x:y;
}
char    max(char x,char y)
{
    return (x>y)?x:y;
}
```

这些函数版本执行的功能都是相同的，只是参数类型和函数返回类型不同。能否为这些函数只写出一套代码呢？

C++中解决这个问题的一个方法就是使用模板。

仔细分析上面的三个函数，它们的参数个数是相同的，实现的代码也是相同的，只是形参的类型和返回值类型不同。如果将上述三个函数中的类型 int、double 和 char 进行参数化，即将 int、double 和 char 都使用一个参数 T 来代替，则可得到下述函数形式：

```
T    max(T x,T y)
{
    return (x>y)?x:y;
}
```

如果需要求两个整数的最大值或者求两个实数的最大值，只需将上述函数中的类型 T 用 int 或 double 来替换就可以了，与前面的两个分别用来求两个整数的最大值和求两个实数的最大值的函数相同。

这个以参数化表示的函数称为函数模板。函数模板就是用来定义一个通用的函数。这样做可以避免许多重复劳动，也可增加程序的灵活性。这是引进函数模板的主要原因。

因此，在 C++中，模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现代码的可重用性。

由于模板的作用就是使程序能够对不同类型的数据进行相同方式的处理。因此，在进行相同方式的处理时，只有当参加运算的数据类型不同时，才可以定义模板。

C++程序由类和函数组成，C++中的模板也分为类模板和函数模板。

在定义了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数，该函数的定义体与函数模板的函数定义体相同，称之为模板函数。

函数模板与模板函数的区别是：函数模板是模板的定义，定义中用到通用类型参数。模板函数是实实在在的函数定义，它由编译系统在遇到具体函数调用时所生成，具有程序代码。

同样，类模板是模板的定义，不是一个实实在在的类，其定义中也用到通用类型参数。在定义了一个类模板后，可以创建类模板的实例，即生成模板类。

模板、模板函数、模板类和对象之间的关系如下图 11-1 所示。

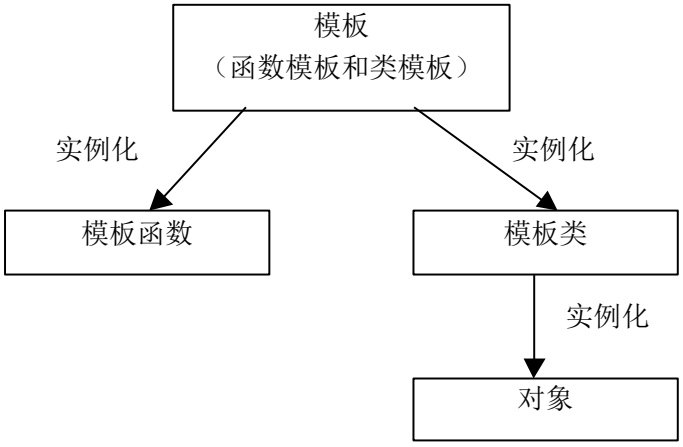

图 11-1　模板、模板函数、模板类和对象之间的关系

11．2　函数模板和模板函数

11．2．1　函数模板的定义和模板函数的生成

定义函数模板的一般形式是：

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

其中，template 是函数模板说明的关键字，它表示声明一个模板。关键字 class 后面的类型参数名是模板形参，它可以代表基本数据类型，也可以代表类。

例如，将求最大值的函数 max()定义成函数模板，如下所示：

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

其中 T 是模板形参，它既可以取系统预定义的数据类型，也可以取用户自定义的类型。

定义了上面的函数模板以后，程序中并没有得到真正的函数代码。只有用一个具体的数据类型来代替上面的类型参数 T 以后，也就是将模板具体化后，系统才会生成特定于具体数据类型的代码。

比如在程序中有如下语句：

```
int  i;
i=max（2，30）;
```

系统就会自动生成具有整型参数和返回值的函数代码：

```
int  max(int x , int y)
{
    return (x>y)?x:y;
}
```

然后将它插入到程序中，这个生成的函数称为模板函数。这样，如果再有一句对整型参数的函数调用，系统就不会再次生成函数代码，而是直接使用已经生成的函数代码了。

同样，如果程序中又有了这么一句函数调用：

```
double x=max（12.3，48.5）;
```

系统会先看有没有已经从模板生成了函数代码，如果没有，就生成具有双精度浮点型参数的函数代码：

```
double  max(double x , double y)
{
    return (x>y)?x:y;
}
```

一般来说，当编译程序看到对模板函数的调用语句时，就会根据函数模板生成对应于该特定数据类型的具体函数代码。但是，也可以通过声明函数原型来告诉编译程序实例化函数模板。例如：

```
float  max(float,float);
```

当编译器看到函数原型时，会先根据函数模板生成相应的函数代码。

例 11-1　不同数据类型数组中的元素求和

```
#include <iostream>
using namespace std;
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 main()
{
    int int_array[]={1,3,5,7,9,11,13,15,17,19};
    double double_array[]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.01};
    cout<<"The summary of integer array are "<<Sum(int_array,10)<<endl;
    cout<<"The summary of double array are "<<Sum(double_array,10)<<endl;
    return 0;
}
```

程序运行结果为：

```
The summary of integer array are 100
The summary of double array are 59.51
```

在程序中，生成了两个模板函数。其中 Sum(int_array,10)用实参 int_array 将类型参数 T 进行了实例化，int_array 为一整型数组名，编译程序生成一个形如 int Sum(int *array,int size)的模板函数；Sum(double_array,10)用实参 double_array 将类型参数 T 进行了实例化，double_array 为一双精度型数组名，编译程序生成一个形如 double Sum(double *array,int size)的模板函数。

11．2．2　函数模板的使用

虽然函数模板中的模板形参 T 可以实例化为各种类型，但实例化 T 的各模板实参之间必须保持完全一致的类型。模板类型并不具有隐式的类型转换，例如在 int 与 char 之间、float 与 int 之间、float 与 double 之间等的隐式类型转换，尽管这种转换在 C++中是比较普遍的。

例 11-2　分析下面程序中的错误

```
#include <iostream>
using namespace std;
template <class T>
T max(T a,T b)
```

```
{
    T c;
    if (a>b) c=a;
    else c=b;
    return c;
}

int main()
{
    int i1=50,i2=40;
    char c1='A',c2='B';
    double x1=45.6,x2=2.5e2;
    cout<<max(i1,i2)<<endl;
    cout<<max(c1,c2)<<endl;
    cout<<max(x1,x2)<<endl;
    cout<<max(i1,c1)<<endl;      //  错误
    cout<<max(i1,x1)<<endl;      //  错误
    cout<<max(c1,x1)<<endl;     //  错误
    return 0;
}
```

程序在编译时将出现 3 个错误（出错语句在程序中给出了注释），原因是 max 模板参数 T 的各实参之间必须保持一致的类型，但出错的这 3 个语句的实参的类型与形参不一致。如 max(i1,c1)，系统找不到与 max（int,char）相匹配的函数定义，VC++编译器给出的错误提示信息为：

```
error C2782: 'T __cdecl max(T,T)' : template parameter 'T' is ambiguous
        could be 'char'
        or       'int'
```

虽然 int 和 char 之间可以隐式转换，完全可以认为是 max（int,int），但是模板类型没有这种识别能力。

解决这个问题有以下两种方法：

（1）采用强制类型转换，例如将调用语句

```
max(i1,c1)
```

改写为

```
max(i1,int(c1))
```

（2）显式给出模板实参，强制生成对特定实例的调用。具体地说，就是在调用格式中要插入一个模板实参表：

```
cout<<max<int>(i1,x1)<<endl;
```

其中，紧跟在函数名 max 后的<int>就是模板实参表，通过它通知编译系统生成对形如 int max(inta,int b)的函数实例的调用。这样，在调用过程中，double 型的参数 x1 将被自动转换成 int 型。当然，也可以按下面的语句实例化：

```
cout<<max<double>(i1,x1)<<endl;
```

这样生成的函数调用将把 int 型的 i1 自动转换成 double 型。

例 11-3　采用上面两种方法对例 11-2 的一种修改

```
#include <iostream>
using namespace std;
template <class T>
T max(T a,T b)
{
    T c;
    if (a>b) c=a;
    else c=b;
    return c;
}

int main()
{
    int i1=50,i2=40;
    char c1='A',c2='B';
    double x1=45.6,x2=2.5e2;
    cout<<max(i1,i2)<<endl;
    cout<<max(c1,c2)<<endl;
    cout<<max(x1,x2)<<endl;
    cout<<max(i1,int(c1))<<endl;            //  正确
    cout<<max<int>(i1,x1)<<endl;            //  正确
    cout<<max<double>(c1,x1)<<endl;     //  正确
    return 0;
}
```

需要注意的是，对不同的数据类型在处理形式上的统一性是建立模板的基础，即同一个函数模板实例化后的所有模板函数都执行系统的动作。但是，这种统一性是相对的，个别数据类型有可能比较特殊，在处理上与大多数数据类型不一致。针对这样的特殊情况，可以通过重载模板函数进行补充。例如要比较两个字符串哪个大，就不能用上面的方法直接比较两个字符指针的大小，而需要用 strcmp（）函数来进行，所以需要重新定义函数 max() 来比较字符串。

例 11-4　重载模板函数

```
#include <iostream>
using namespace std;
template <class T>
T max(T a,T b)
{
```

```
    if (a>b) return a;
    else    return b;
}
char * max(char *a,char *b)
{
    if (strcmp(a,b)>0) return a;
    else return b;
}
int main()
{
    cout<<max(15,20)<<endl;
    cout<<max('A','B')<<endl;
    cout<<max("China","Beijing")<<endl;
    return 0;
}
```

程序运行结果为：

```
20
B
China
```

程序中，函数 char *max(char *,char *)中的名字 max 与函数模板的名字相同，但操作不同，函数体中的比较采用了字符串比较函数，所以有必要用重载的方法把它们区分开，这种情况就是重载模板函数。编译程序在处理这种情况时，首先匹配重载函数，然后再寻求模板的匹配。

编译程序遇到 max("China","Beijing")调用时，先进行重载函数匹配，结果匹配了非模板函数 char *max(char *,char *)，所以这里不会为它产生模板函数的代码。

如果将程序中的重载模板函数 char * max(char *a,char *b) 删除，程序也能运行，但运行结果不再正确（实例化后的模板函数进行字符指针的比较）。结果为：

```
20
B
Beijing
```

11．2．3　函数模板与模板函数

函数模板是对一组函数的描述，它不是一个实实在在的函数，编译系统并不产生任何执行代码。

当编译系统在程序中发现有与函数模板中相匹配的函数调用时，便生成一个重载函数，该重载函的函数体与函数模板的函数体相同。这个根据函数模板生成的重载函数称为模板函数。

函数模板与模板函数的区别如下：

（1）函数模板不是一个函数，而是一组函数的模板，在定义中使用了参数类型。

（2）模板函数是一种实实在在的函数定义，它的函数体与某个函数模板的函数体相同。

编译系统遇到模板函数调用时，将生成可执行代码。函数模板是定义重载函数的一种工具。一个函数模板只为一种原型函数生成一个模板函数，不同原型的模板函数是重载的。这样就使得一个函数只需编码一次就能用于某个范围的不同类型的对象上。因此，可以说函数模板是提供一组重载函数的样板。

11．3　类模板和模板类

类模板是一系列相关类的模型或样板，这些类的成员组成相同，成员函数的源代码形式也相同，所不同的只是所针对的类型，也就是这一系列类中的某些数据成员、某些成员函数的参数、某些成员函数的返回值能取任意数据类型。

11．3．1　类模板的定义

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，后面是用尖括号括起来的模板参数，然后是类名，其一般形式是：

```
template <class　类型参数名 1，class　类型参数名 2，…>
class 类名
{
    类声明体
};
```

例如，一个单链表类模板的定义如下：

```
template <class T>
class List{                    // 定义通用单链表类模板 List，有一个模板参数 T
public:
    List();                    // 声明构造函数
    void Add(T&);              // 声明增加结点的成员函数
    void Remove(T&);           // 声明删除结点的成员函数
    void PrintList();          // 声明打印链表的成员函数
    ~List();                   // 声明析构函数
protected:
    struct Node{               // 定义结点的结构类型
        Node *pNext;           // 指向下一个结点的指针成员
        T data;                // 本结点的数据成员
    };
    Node *pHead;               // 指向链表头结点的指针
};
```

在类声明体外定义成员函数时，若成员函数中有模板参数存在，则需要在函数体外进行模板声明（即成员函数定义为函数模板），并且在函数名前面的类名后缀上“<类型参数

名>”。

其一般形式为：

template <class　类型参数名 1，class　类型参数名 2，…>

函数返回值类型 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

}

例如，类模板 List 中成员函数 Add()和 Remove()在类体外定义为：

```
template<class T>
void List<T>::Add(T& t)          // 定义增加结点的成员函数
{
    Node* temp=new Node;     // 向堆内存申请一个结点的内存空间
    temp->data=t;            // 将数据 t 赋给新结点的 data 数据成员
    temp->pNext=pHead;       // 将链表头指针赋给新结点的 pNext
    pHead=temp;              // 将新结点的地址赋给头指针
}
template<class T>
void List<T>::Remove(T& t)   // 定义删除结点的成员函数
{
    Node *q=0;
    if ((pHead->data)==t)    // 判断要删除的结点是否是头结点，如是，
                             // 链首指针指向下一个结点，把第一个结点脱链
    {
        q=pHead;
        pHead=pHead->pNext;
    }
    else                     // 顺链查找
    {
        for(Node * p=pHead;p->pNext;p=p->pNext)
            if ((p->pNext->data)==t)
            {
                q=p->pNext;
                p->pNext=q->pNext;
                break;
            }
    }
    if(q)                    // 如果待删除结点在链中存在
    {
        delete q;            // 删除待删结点
```

```
    }
}
```

11．3．2　类模板的使用

类模板不代表一个具体的、实际的类，而代表着一类类，编译程序不会为类模板（包括成员函数定义）创建程序代码，但是通过对类模板的实例化可以生成一个具体的类（模板类）以及该具体类的对象。

与函数模板不同的是：函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定，其实例化的一般形式是：

类名 <实际的数据类型 1，实际的数据类型 2…> 对象名

例如，List<int> intList;

表示将类模板 List 的类型参数 T 全部替换成 int 型，从而创建一个具体的整型链表类，并生成该具体类的一个对象 intList。

List<char> charList;

表示将类模板 List 的类型参数 T 全部替换成 char 型，从而创建一个具体的字符链表类，并生成该具体类的一个对象 charList。

通过对类模板实例化可以生成模板类。类模板与模板类的区别是：类模板是模板的定义，不是一个实实在在的类，定义中用到通用类型参数。而模板类是实实在在的类定义，是类模板的实例化。类定义中参数被实际类型所代替。

例 11-5　类模板 List 的定义和使用

```
#include <iostream>
using namespace std;
template <class T>
class List{                    // 定义通用单链表类模板
public:
    List();                    // 声明构造函数
    void Add(T&);              // 声明增加结点的成员函数
    void Remove(T&);           // 声明删除结点的成员函数
    void PrintList();          // 声明打印链表的成员函数
    ~List();                   // 声明析构函数
protected:
    struct Node{               // 定义结点的结构类型
        Node *pNext;           // 指向下一个结点的指针成员
        T data;                // 本结点的数据成员
    };
    Node *pHead;               // 指向链表头结点的指针
};

template <class T>
```

```
List<T>::List()                    // 定义单链表的构造函数
{
    pHead=NULL;                    // 单链表头指针设为空指针
}
template<class T>
void List<T>::Add(T& t)        // 定义增加结点的成员函数
{
    Node* temp=new Node;       // 向堆内存申请一个结点的内存空间
    temp->data=t;              // 将数据 t 赋给新结点的 data 数据成员
    temp->pNext=pHead;         // 将链表头指针赋给新结点的 pNext
    pHead=temp;                // 将新结点的地址赋给头指针
}
template<class T>
void List<T>::Remove(T& t)   // 定义删除结点的成员函数
{
    Node *q=0;
    if ((pHead->data)==t)     // 判断要删除的结点是否是头结点，如是，
                              // 链首指针指向下一个结点，把第一个结点脱链
    {
        q=pHead;
        pHead=pHead->pNext;
    }
    else                      // 顺链查找
    {
        for(Node * p=pHead;p->pNext;p=p->pNext)
            if ((p->pNext->data)==t)
            {
                q=p->pNext;
                p->pNext=q->pNext;
                break;
            }
    }
    if(q)                     // 如果待删除结点在链中存在
    {
        delete q;             // 删除待删结点
    }
}

template<class T>
```

```
void List<T>::PrintList()     //定义打印链表的成员函数
{
    for (Node* p=pHead;p;p=p->pNext)
    {
        cout<<(p->data)<<" ";   // 显示 p 指针所指向结点的数据值
    }
    cout <<endl;
}
template<class T>
List<T>::~List()                    // 定义链表类的析构函数
{
    Node* p;
    while(pHead!=NULL)              // 删除链表中所有结点，释放结点数据所空间
    {
        p=pHead;
        pHead=pHead->pNext;
        delete p;
    }
}

int main()
{
    List<int> intList;          // 将 List 实例化为整型链表类，并创建对象 intList
    List<char> charList;        // 将 List 实例化为字符链表类，并创建对象 charList
    List<double> doubleList; // 将 List 实例化为双精度数链表类，并创建对象 doubleList
    int a;
    char c;
    double x;
    for(int i=0;i<10;i++)
    {
        intList.Add(i);
        c=65+i;
        charList.Add(c);
        x=i*10+0.5;
        doubleList.Add(x);
    }
    cout<<"The int List is    ";
    intList.PrintList();
    cout<<"The char List is    ";
```

```
    charList.PrintList();
    cout<<"The double List is    ";
    doubleList.PrintList();
    cout<<"Please input a integer to remove :" ;
    cin>>a;
    intList.Remove(a);
    cout<<"Please input a char to remove :" ;
    cin>>c;
    charList.Remove(c);
    cout<<"Please input a double to remove :" ;
    cin>>x;
    doubleList.Remove(x);
    cout<<"After removed :"<<endl;
    cout<<"The int List is    ";
    intList.PrintList();
    cout<<"The char List is    ";
    charList.PrintList();
    cout<<"The double List is    ";
    doubleList.PrintList();
    return 0;
}
```

程序运行情况示例如下：

```
The int List is    9 8 7 6 5 4 3 2 1 0
The char List is    J I H G F E D C B A
The double List is    90.5 80.5 70.5 60.5 50.5 40.5 30.5 20.5 10.5 0.5
Please input a integer to remove :7
Please input a char to remove :m
Please input a double to remove :90.5
After removed :
The int List is    9 8 6 5 4 3 2 1 0
The char List is    J I H G F E D C B A
The double List is    80.5 70.5 60.5 50.5 40.5 30.5 20.5 10.5 0.5
```

11．3．3　类模板的派生

可以从类模板派生出类模板，也可以从类模板派生出普通类（非模板类），还可以从一个普通类甚至一个未知的基类派生出类模板。

1．从类模板派生出类模板

从类模板派生出新的类模板的格式如下所示：

```
template <class T>
```

```
class    Base
{
      ……
};
template    <class>
class Derived:public Base <T>
{
   ……
};
```

与定义一般派生类的格式类似，只是在指出它的父类时要加上模板参数，如 Base <T>。

例 11-6　从单链表类模板派生出一个集合类模板

集合（采用单链表来保存集合中的元素）与链表的差别是，集合中没有相同的元素，因此，在进行插入操作时要先判断集合中是否存在要插入的元素，仅当集合中没有这个元素时才做插入操作，并且对集合而言，通常有判断一个元素是否在集合中的需求。这样可以从前面定义的单链表类模板中派生出一个集合类模板，在集合类模板中只需重新定义完成插入操作的成员函数，并增加一个判断一个元素是否在集合中的成员函数，其余成员函数继承单链表类模板中定义的即可。具体的代码如下：

```
#include <iostream>
using namespace std;
……  // 省略了单链表类模板的定义和实现
template <class T>
class Set:public List<T>
{
public:
      bool Contains(T& t);
      void Add(T& t);
};

template <class T>
bool Set<T>::Contains(T& t)
{
      for(Node *p=pHead;p;p=p->pNext)
      {
            if((p->data)==t)
                  return true;
      }
      return 0;
}
template <class T>
```

```
void Set<T>::Add(T& t)
{
    if (!Contains(t))
        List<T>::Add(t);
}
int main()
{
    List<int> intList;     // 将 List 实例化为整型链表类，并创建对象 intList
    Set<int> intSet;       // 将 Set 实例化为整数集合类，并创建对象 intSet
    int a;
    for(int i=0;i<10;i++)
    {
        intList.Add(i);
        intSet.Add(i);
    }
    cout<<"The int List is    ";
    intList.PrintList();
    cout<<"The int Set is    ";
    intSet.PrintList();
    cout<<"Please input a integer to Add :" ;
    cin>>a;
    intList.Add(a);
    intSet.Add(a);
    cout<<"After Add a integer :"<<endl;
    cout<<"The int List is    ";
    intList.PrintList();
    if (intSet.Contains(a))
        cout<<a<<"    is in intSet!"<<endl;
    else
        cout<<a<<"    isnot in intSet!"<<endl;
    cout<<"The int Set is    ";
    intSet.PrintList();
    return 0;
}
```

程序运行情况示例如下：

```
The int List is    9 8 7 6 5 4 3 2 1 0
The int Set is    9 8 7 6 5 4 3 2 1 0
Please input a integer to Add :1
After Add a integer :
```

```
The int List is   1 9 8 7 6 5 4 3 2 1 0
1   is in intSet!
The int Set is   9 8 7 6 5 4 3 2 1 0
```

从程序运行结果可以看出，增加整数 1 时，由于集合中已经有了这个元素，因此拒绝将其在插入集合中。

2．从类模板派生出普通类

从类模板派生出一个普通类的格式如下所示：

```
template <class T>
class Base
{
  ……
}
class Derived:public Base<int>
{
  ……
}
```

首先，作为新派生出来的普通类的父类，必须是类模板实例化后生成的模板类，例如上面的 Base<int>；其次，在派生类定义之前不需要模板声明语句 template <class T>。

例 11-7　从单链表类模板派生出一个整数集合类

```
#include <iostream>
using namespace std;
……  // 省略了单链表类模板的定义和实现
class ISet:public List<int>       // 派生出的整数集合类的定义
{
public:
      bool Contains(int& t);
      void Add(int& t);
};
// 整数集合类成员函数的实现
bool ISet::Contains(int& t)
{
      for(Node *p=pHead;p;p=p->pNext)
      {
            if((p->data)==t)
                  return true;
      }
      return 0;
}
```

```
void ISet::Add(int& t)
{
    if (!Contains(t))
        List<int>::Add(t);
}
int main()
{
    List<int> intList;     // 将 List 实例化为整型链表类，并创建对象 intList
    ISet intSet;           // 创建整数集合类 ISet 的对象 intSet
    int a;
    for(int i=1;i<20;i+=2)
    {
        intList.Add(i);
        intSet.Add(i);
    }
    cout<<"The int List is    ";
    intList.PrintList();
    cout<<"The int Set is    ";
    intSet.PrintList();
    cout<<"Please input a integer to Add :" ;
    cin>>a;
    intList.Add(a);
    intSet.Add(a);
    cout<<"After Add a integer :"<<endl;
    cout<<"The int List is    ";
    intList.PrintList();
    if (intSet.Contains(a))
        cout<<a<<"    is in intSet!"<<endl;
    else
        cout<<a<<"    isnot in intSet!"<<endl;
    cout<<"The int Set is    ";
    intSet.PrintList();
    return 0;
}
```

3．从普通类派生出类模板

在声明一个类模板时，可以尽可能将类模板中与虚拟类型参数无关的成员剥离出来，构成一个普通类，作为类模板的基类。因此，从普通类派生出类模板的情况也是十分常见的。例如：

```
class    Base
{
    ……
}
template <class T>
class Derived:public Base
{
    T data;
    ……
}
```

模板的派生甚至可以继承一个未知的基类，也就是说，继承哪个基类由模板参数决定，如下例所示。

例 11-8　从未知的基类派生出类模板

```
#include <iostream>
using namespace std;
class Base_1
{
public:
    Base_1() {cout<<"Construct Base_1 ……"<<endl; }
};
class Base_2
{
public:
    Base_2() {cout<<"Construct Base_2 ……"<<endl; }
};
template <class T,int val>
class Base_3
{
public:
    Base_3(T n=val):data(n)
    {
        cout<<"Construct Base_3……    "<<data<<endl;
    }
private:
    T data;
};
template <class T>
class Derived:public T
{
```

```
public:
    Derived():T() {cout<<"Construct Derived……"<<endl;}
};

int main()
{
    Derived<Base_1> x;
    Derived<Base_2> y;
    Derived<Base_3<int,10> > z;
    return 0;
}
```

程序运行结果如下：

```
Construct Base_1 ……
Construct Derived……
Construct Base_2 ……
Construct Derived……
Construct Base_3……    10
Construct Derived……
```

本章小结

在 C++中，模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现代码的可重用性。

C++中的模板也分为类模板和函数模板。

定义函数模板的一般形式是：

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

定义了函数模板后，当编译系统在程序中发现有与函数模板中相匹配的函数调用时，便生成一个重载函数，该重载函的函数体与函数模板的函数体相同。这个根据函数模板生成的重载函数称为模板函数。

定义一个类模板的一般形式是：

```
template <class   类型参数名 1，class   类型参数名 2，…>
class 类名
{
    类声明体
```

};

类模板不代表一个具体的、实际的类，而代表着一类类，编译程序不会为类模板（包括成员函数定义）创建程序代码，但是通过对类模板的实例化可以生成一个具体的类（模板类）以及该具体类的对象。

与函数模板不同的是：函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定，其实例化的一般形式是：

类名 <实际的数据类型 1，实际的数据类型 2…> 对象名

习　　题

1．什么是模板？函数模板与模板函数、类模板与模板类之间的关系如何？

2．已知

```
void Sort(int a[],int size);
void Sort(double a[],int size);
```

是一个函数模板的两个实例，其功能是将数组 a 中的前 size 个元素按从小到大顺序排列。试设计这个函数模板。

3．设有如下的类声明：

```
class   Test
{
public:
   void SetData1(int val) { data1=val; }
   void SetData2(double val) {data2=val: }
   int GetData1()   { return data1; }
   double GetData2() { return data2; }
private:
   int data1;
   double   data2;
}
```

试将此类声明改为类模板声明，使得数据成员 data1 和 data2 可以是任何类型。

4．栈是一种重要的数据结构，它是一种只允许在表的一端进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为栈顶。表的另一端称为栈底。栈顶的当前位置是动态的，对栈顶当前位置的标记称为栈顶指针。当栈中没有数据元素时，称之为空栈。栈的插入操作通常称为进栈或入栈，栈的删除操作通常称为退栈或出栈。下面是一个整型栈类的定义：

```
const   int SIZE= 100；    // 栈中能保存的最多元素个数
class   IStack
{
public:
```

```
    Istack();                // 栈的构造函数
    void Push(int n);        // 往栈顶增加元素
    int Pop();               // 从非空栈的栈顶删除一个元素
    int GetTop();            // 返回非空栈的栈顶元素
    bool Empty();            // 判断栈是否为空
    int Size();              // 返回栈中元素的个数
    void ClearStack();       // 将栈清空
    ~Istack();               // 栈的析构函数
private:
    int stack[SIZE];         // 保存栈中各元素的数组
    int top;                 // 保存栈顶的当前位置
}
```

试编写一个栈的类模板（包括其成员函数的实现），以便为任何类型的对象提供栈结构的数据操作。

5．队列也是一种重要的数据结构，它是一种只允许在表的一端进行插入操作而在另一端进行删除操作的线性表。表中允许进行插入操作的一端称为队尾，允许进行删除操作的一端称为队头。队头和队尾分别由队头指针和队尾指针指示。当队列中没有数据元素时，称之为空队列。队列的插入操作通常称为进队列或入队列，队列的删除操作通常称为退队列或出队列。下面是一个整型队列类的定义：

```
const int SIZE=100;          // 队列中元素的最大个数
class Iqueue
{
public:
    Iqueue();                // 队列的初始化构造函数
    void Insert(int n);      // 新元素入队
    int Delete();            // 元素出队
    bool Empty();            //  判断队列是否为空
    int Length();            // 返回队列中元素的个数
    int Front();             // 返回队头元素
    void ClearQueue();       // 将队列清空
    ~Iqueue();               // 队列的析构函数
private:
    int Qlist[SIZE];         // 用于保存队列元素的数组
    int front,rear;          // 队头指针、队尾指针的位置
}
```

试编写一个队列的类模板（包括成员函数的实现），以便为任何类型的对象提供队列结构的数据操作。