第四章　模板及其应用

本章导读

C++模板具有非常强大的功能，几乎成为了C++的代名词，掌握模板使用的高级技术是现代C++程序员的必修课。模板就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， 从而实现了真正的代码可重用性。模版可以分为两类，一个是函数模版，另外一个是类模版。模板是C++实现泛型程序设计的重要手段。

模板进一步利用了参数化概念，不仅允许参数化值，还允许参数化类型。不仅仅包含C++基本类型，还包括用户定义的类。通过模板，不仅可以编写不依赖特定值的代码，还能编写不依赖数据类型的代码。

尽管模板是一个令人惊奇的语言特性，但其模板语法也常常让人费解，这对推动模板的使用带来了一定障碍。

学习目标：

1. 通过一个应用实例，认识模板；
2. 深入认识函数模板；
3. 深入认识类模板；
4. 了解模板的典型应用；

本章目录

第一节　函数模板

例程4-1的功能是两个整数类型的数相加，如果要实现其他类型的数相加，可以用函数重载的方法，定义出double相加、char相加等，甚至也可以定义出自定义类如前面经常用到的分数类相加等。但如果新出现一个类，就无能为力。Plus()是非常简单的函数，还问题不大，顺手写一个函数即可，但如果是一个非常复杂的算法函数，对于新出现的类，再用类似的方法就非常难以胜任。这种需求，在C++中可用模板函数予以解决，如例程4-2所示。

例程4-1

```
第1行  int Plus(int a,int b){
第2行      int tmp=a+b;
第3行      return tmp;
第4行  };
```

例程4-2

```
第1行  template<typename T>
第2行  T Plus(T a,T b){
第3行      T tmp=a+b;
第4行      return tmp;
第5行  };
```

在例程4-2中，原有数据类型int被T替换。如果程序代码中出现int相加，则编译器将自动把T替换成int，自动生成一个Plus()函数适应int的函数；如果出现double相加，则生成适应double的函数；如果出现UpDown(前面章节例程中分数类)，则生成适应UpDown的函数。也就是说，如果代码中有多种类型数据相加，则编译器将根据数据类型按照模板函数自动生成实际使用的函数。例程4-3是模板函数

Plus()的应用示例。

例程4-3

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  template<typename T>
第5行  T Plus(T a,T b){
第6行      T tmp=a+b;
第7行      return tmp;
第8行  }
第9行  class UpDown{
第10行 public:
第11行     UpDown(int U=1,int D=1):Up(U),Down(D){}
第12行     UpDown operator+(const UpDown &UD){
第13行         return UpDown(this->Up*UD.Down+this->Down*UD.Up,this->Down*UD.Down);
第14行     }
第15行     friend ostream &operator<<(ostream &out,const UpDown &UD);
第16行 private:
第17行     int Up,Down;
第18行 };
第19行 ostream &operator<<(ostream &out,const UpDown &UD){
第20行     return out<<"("<<UD.Up<<"/"<<UD.Down<<")";
第21行 }
第22行 int main(){
第23行     cout<<Plus<int>(2,4)<<endl;//输出6
第24行     cout<<Plus<double>(2.2,4.4)<<endl;//输出6.6
第25行     cout<<Plus<int>(2.2,4.4)<<endl;//输出6
第26行     cout<<Plus<UpDown>(UpDown(2,3),UpDown(1,5))<<endl;//输出(13/15)
第27行
第28行     cout<<Plus(2,4)<<endl;//输出6
第29行     cout<<Plus(2.2,4.4)<<endl;//输出6.6
第30行     cout<<Plus(2.2,4.4)<<endl;//输出6.6
第31行     cout<<Plus<UpDown>(UpDown(2,3),UpDown(1,5))<<endl;//输出(13/15)
第32行
第33行     return 0;
第34行 }
```

当定义模板函数时，需要在函数前增加类似template<typename T>，其中template是表明模板的关键字，typename用于修饰后续的T，表明T为模板参数，T是模板参数，相当于函数的形式参数，遵循标识符命名规则。另外，typename在此处还可以用class代替，但用typename可能更好，如果用class，易认为T为某个类，而此处的T即可代表基本数据类型，也可以代表某个class。早期的编译器，仅支持class而不支持typename，不过现在的编译器，则两者均支持。

从例程4-3可以看出，当调用模板函数时，有两种形式，形如：Plus<int>(2,4)和Plus(2,4)。前者是显式调用，用以表明T相当于int，要求模板函数Plus()中的T用int代替；后者是隐式调用，编译器根据输入参数2和4对应的T，自动推断出T相当于何种数据类型。另外，前者具有更高的优先级，如Plus<int>(2.2,4.4)，虽然输入参数是double或float，但由于T已被指定为int，因此会将2.2和4.4自动转换为int类型后相加。如果参数不能转换，则报错。在具体使用过程中，一般使用隐含模式，当不能自动推断时，则必须使用显式模式。例程4-4是有关多个模板参数的示例。

例程4-4

```
#include<iostream>
using namespace std;

template<typename Ta,typename Tb,typename Tc>
Ta Plus(Tb a,Tc b){
    Ta tmp=a+b;
    return tmp;
}
int main(){
    //cout<<Plus(2.7,4.4)<<endl;//错误!
    cout<<Plus<int>(2.7,4.4)<<endl;//输出7
    cout<<Plus<int,int,int>(2.7,4.4)<<endl;//输出6
    cout<<Plus<double,double,double>(2.7,4.4)<<endl;//输出7.1
    cout<<Plus<double,int,double>(2.7,4.4)<<endl;//输出6.4

    return 0;
}
```

模板函数Plus()有三个模板参数，形如：template<typename Ta,typename Tb,typename Tc>Ta Plus(Tb a,Tc b)，其中Ta为函数的返回值类型，Tb是函数的第一个参数，Tc是第二个参数，注意：Ta、Tb和Tc前的typename都不能省略。在函数调用时，有多种形式，但Plus(2.7,4.4)则错误，因为函数推断不出返回值的数据类型；另外，Plus<double,int,double>(2.7,4.4)则表示Ta类型为double，为返回值类型，Tb为int，如果输入2.7不是int，将被转换为int，Tc为double。Plus后的类型顺序与template后typename申明顺序完全一致。

函数参数化，能让函数处理相同数据类型不同数值；函数模板化，能让函数处理不同类型的数据。函数重载化，让函数处理不同类型或不同顺序或不同数量的参数；函数模板化，让函数能根据需要按照模板生成不同函数。参数化、模板化、重载化，让函数具有广泛适应性。

第二节　类模板

模板，不仅能应用于函数，还能应用于类，让同样结构的类能处理不同类型的数据，如vector，可以是vector<int>也可以是vector<double>或者vector<UpDown>,例程4-5是模板应用于类的示例，是上一章“类的嵌套”中Array类定义的简化，但增加了模板功能，使之能适应不同类型。当按照上一章示例时，仅能适应一种数据类型(int)，现在则不但可以适用于int，还可以适用于double、UpDown(自定义分数类)等。示例第23行代码Array<double> myArr(50)中的<double> 表明Array用于double，将double带入Array定义中的T，会有助于理解类模板。Array这个类模板不仅适用于double，也适用于其他数据类型，在C++中，将这种类称之为容器(container)，能容纳其他各种数据类型，非常形象。

例程4-5

```
#include<iostream>
using namespace std;

template<typename T>
class Array{
public:
    Array(unsigned int size=100):count(size){
        this->addrData=new T[this->count];//注意T的使用
    }
    T *getAddrStart()const{//获得数据起始地址，注意T的使用
```

```
第11行          return this->addrData;
第12行      }
第13行      unsigned int getSize(){
第14行          return this->count;
第15行      }
第16行  private:
第17行      T *addrData;//数据类型为T
第18行      unsigned int count;//数据成员数量
第19行  };
第20行
第21行  int main(){
第22行
第23行      Array<double> myArr(50);//int可以更换其他数据类型
第24行
第25行      double *addrStart=myArr.getAddrStart();//获得数据起始地址
第26行      for(unsigned int i=0;i<myArr.getSize();++i)
第27行          *(addrStart+i)=rand()/100.0;
第28行
第29行      for(unsigned int i=0;i<myArr.getSize();++i)
第30行          cout<<*(addrStart+i)<<endl;
第31行
第32行      return 0;
第33行  }
```

类模板不仅可以应用于独立类，也可应用在继承、包含、友元和嵌套等关系中，例程4-6继承关系的示例。

例程4-6

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  using namespace std;
第5行
第6行  template<typename leftArg,typename rightArg,typename resultReturn>
第7行  struct binFunction{
第8行      binFunction(rightArg arg2):rNum(arg2){}
第9行      rightArg rNum;
第10行 };
第11行 template<typename T>
第12行 struct lessThan:public binFunction<T,T,bool>{
第13行     lessThan(const T &rNum):binFunction(rNum){}
第14行     bool operator()(const T & leftNum){return leftNum<rNum;}
第15行 };
第16行 template<typename T>
第17行 struct greaterThan:public binFunction<T,T,bool>{
第18行     greaterThan(const T &rNum):binFunction(rNum){}
第19行     bool operator()(const T & leftNum){return leftNum>rNum;}
```

```
第20行  };
第21行  template<typename T>
第22行  struct equalTo:public binFunction<T,T,bool>{
第23行      equalTo(const T &rNum):binFunction(rNum){}
第24行      bool operator()(const T & leftNum){return leftNum==rNum;}
第25行  };
第26行
第27行  int main(){
第28行      vector<int> myVec;
第29行      for(int i=0;i<20;++i)
第30行          myVec.push_back(rand()%100);
第31行      cout<<"原始数据\n";
第32行      for(int i=0;i<20;++i)
第33行          cout<<myVec[i]<<" ";
第34行
第35行      int lessThan20=count_if(myVec.begin(),myVec.end(),lessThan<int>(20));
第36行      cout<<"\n小于20的数量="<<lessThan20<<endl;
第37行
第38行      int greaterThan20=count_if(myVec.begin(),myVec.end(),greaterThan<int>(20));
第39行      cout<<"\n大于20的数量="<<greaterThan20<<endl;
第40行
第41行      int equalTo20=count_if(myVec.begin(),myVec.end(),equalTo<int>(20));
第42行      cout<<"\n等于20的数量="<<equalTo20<<endl;
第43行
第44行      return 0;
第45行  }
```

在例程4-6中，基类binFunction有三个模板参数(leftArg、rightArg和resultReturn)，派生类lessThan、greaterThan和equalTo继承自基类binFunction。基类有三个模板参数，其中前两个与派生类相同，第3个则是bool型。当派生类如lessThan被实例化时，只要确定lessThan的模板参数，则派生类的模板参数也同时确定，因此也能被正确实例化。另外，当派生类如lessThan执行构造函数，同时执行基类的构造函数如binFunction(rNum)(如不调用该构造函数，基类将不能被实例化)。

例程4-6中的类成员函数定义在类的外部，当在类外定义成员函数时，必须遵循以下规则：

- 必须以template开头且后接模板形参表；
- 必须指出成员函数的隶属关系，即属于哪个类；
- 类名必须包含模板形参；
- 当以被嵌套类作为数据类型时，该数据类型前必须使用typename，且不能以class替换(如例程4-7所示)。

例程4-6第35、38和41行，分别在count_if函数中有参数lessThan<int>(20)、greaterThan<int>(20)和equalTo<int>(20)，虽然都类似函数，但却不是函数，这3个分别在count_if中的参数，都是类lessThan、greaterThan和equalTo的匿名对象，执行其重载的圆括号运算符函数，故虽然形似函数，但却不是函数，在C++中常被称为函数对象(Function Object)、或Functor(仿函数、函数子)等。由例程4-6可以看出，如果有通用的算法和通用的函数对象，将大大提升C++程序的开发效率，这正是C++STL的设计目标，其STL中的函数对象的设计更加技巧，此处仅仅是为说明类模板。

例程4-7是含有嵌套关系的模板类，其功能与例程4-6相似，类lessThan和greaterEqual嵌套在了类binary之中。类lessThan的代码都在类体中实现，而greaterEqual类仅定义原型，其实现在第20-25行。第21行代码binary<T>::greaterEqual::greaterEqual(T rArg):rightArg(rArg){}的含义是定义binary类中嵌套类greaterEqual类里构造函数greaterEqual，binary<T>不能省略，不然编译器不知道greaterEqual类的隶属关系。binary<T>中的<T>也不能省略，表明其为模板类。

例程4-7

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
template<typename T>
struct binary{
    struct lessThan{
        T rightArg;
        lessThan(T rArg):rightArg(rArg){}
        bool operator()(T leftArg){return leftArg<rightArg;}
    };
    struct greaterEqual{
        T rightArg;
        greaterEqual(T rArg);
        bool operator()(T leftArg);
    };
    greaterEqual NOT(lessThan LT);//仅为了说明问题
    lessThan NOT(greaterEqual GE);//仅为了说明问题
};
template<typename T>
binary<T>::greaterEqual::greaterEqual(T rArg):rightArg(rArg){}
template<typename T>
bool binary<T>::greaterEqual::operator()(T leftArg){
    return leftArg>=rightArg;
}
template<typename T>
typename binary<T>::greaterEqual binary<T>::NOT(typename binary<T>::lessThan LT){
    return binary<T>::greaterEqual(LT.rightArg);
}
template<typename T>
typename binary<T>::lessThan binary<T>::NOT(typename binary<T>::greaterEqual GE){
    return binary<T>::lessThan(GE.rightArg);
}
int main(){
    vector<int> myVec;
    for(int i=0;i<100;++i)
        myVec.push_back(rand()%100);
    cout<<"原始数据:\n";
    for(int i=0;i<100;++i)
        cout<<myVec[i]<<" ";

    int lessThan50=count_if(myVec.begin(),myVec.end(),binary<int>::lessThan(50));
    cout<<endl<<lessThan50<<endl;

```

```
第45行      int greaterEqual50=count_if(myVec.begin(),myVec.end(),binary<int>::greaterEqual(50));
第46行      cout<<endl<<greaterEqual50<<endl;
第47行
第48行      binary<int> myBin;
第49行      myBin.NOT(binary<int>::greaterEqual(50));
第50行      greaterEqual50=count_if(myVec.begin(),myVec.end(),myBin.NOT(binary<int>::lessThan(50)));
第51行      cout<<endl<<greaterEqual50<<endl;
第52行
第53行      return 0;
第54行  }
```

例程4-7第27行、第31行分别出现类似typename binary<T>::greaterEqual的用法，其功能是说明其数据类型，注意其前面的typename不能省略，用以表明greaterEqual是以模板形式依赖binary。

例程4-8是类组合模式的示例。Node用于描述一个节点。节点有dataSegment指针，用于保存用new分配T数据类型的地址；同时节点还有Node<T>类型的Next指针，用于保存下一个节点Node。即每个节点都有Next指针，指向下一个节点，由此构成单向链表，链表中每个节点存储的数据量由addNode()函数设定。

例程4-8

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   template<typename T>
第4行   class Node{
第5行   public:
第6行       Node(unsigned int memSize=100):size(memSize){
第7行           dataSegment=new T[this->size];
第8行           this->Next=NULL;
第9行       }
第10行      T * getDataStartAddr(){
第11行          return this->dataSegment;
第12行      }
第13行      T* getNextAddr(){
第14行          return this->Next;
第15行      }
第16行      void setNextAddr(Node<T>* p){
第17行          this->Next=p;
第18行      }
第19行  private:
第20行      unsigned int size;
第21行      T *dataSegment;
第22行      Node<T> *Next;
第23行  };
第24行  template<typename T>
第25行  class List{
第26行  public:
第27行      List(){this->first=NULL;this->last=NULL;}
第28行      List(int size){
```

```
第29行          this->first=new Node<T>(size);
第30行          this->last=this->first;
第31行      }

第32行      void addNode(int size=100){
第33行          if(this->first==NULL){
第34行              this->first=new Node<T>(size);
第35行              this->last=this->first;
第36行          }
第37行          else{
第38行              Node<T> *pTmp=new Node<T>(size);
第39行              this->last->setNextAddr(pTmp);
第40行              this->last=pTmp;
第41行          }
第42行      }
第43行      Node<T>* getFirst(){
第44行          return this->first;
第45行      }
第46行  private:
第47行      Node<T>*first;//第一个节点
第48行      Node<T>*last;//最后一个节点
第49行  };
第50行  int main(){
第51行      List<int> myList;
第52行      myList.addNode(20);
第53行
第54行      return 0;
第55行  }
```

从某种意义上讲，友元类和类的组合相似，差别主要是友元类可以访问含友元类的类全部数据成员和函数成员，不在此处赘述。

第三节　模板应用

在C++中，模板的用途非常广泛。数据遍历是数组、vector等系列数据管理中最常用功能之一，与其他功能相配合，用途更加广泛。以for_each()函数的演进为例，能对模板有更多了解。例程4-9第6-9行的for_each()函数的功能就是遍历，要求输入开始指针(Beg)和结束指针(End)。在函数代码中，print()函数的功能是输出每一个数据成员的值，该函数定义在第3-5行。

例程4-9

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   void print(const int &val){
第4行       cout<<val<<endl;
第5行   }
第6行   void for_each(int *Beg,int *End){//遍历数据
第7行       int *p=Beg;
第8行       while(p!=End)print(*p++);
第9行   };
第10行  int main(){
```

```
第11行      int myArr[]={10,20,1,7,9,88,99,12,209};
第12行      int count=sizeof(myArr)/sizeof(myArr[0]);
第13行
第14行      for_each(myArr,myArr+count);//执行for_each()函数
第15行
第16行      return 0;
第17行  }
```

对于for_each()函数而言，仅能支持int数据类型，不支持其他数据类型，经过函数模板的改造后，就能支持各种数据类型，如例程4-10所示。print()函数也做类似的改动。

例程4-10

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   template<typename T>
第5行   void print(const T &val){
第6行       cout<<val<<endl;
第7行   }
第8行
第9行   template<typename T>
第10行  void for_each(T *Beg,T *End){//遍历数据
第11行      T *p=Beg;
第12行      while(p!=End)print(*p++);
第13行  };
第14行  int main(){
第15行      int myArr[]={10,20,1,7,9,88,99,12,209};
第16行      int count=sizeof(myArr)/sizeof(myArr[0]);
第17行
第18行      for_each(myArr,myArr+count);//执行for_each()函数
第19行
第20行      double hisArr[]={1.23,20.12,1.98,7.87,9.1,88.01,99.98,12.87,209.876,0.98,5.25};
第21行      int size=sizeof(hisArr)/sizeof(hisArr[0]);
第22行
第23行      for_each(hisArr,hisArr+count);//执行for_each()函数
第24行
第25行      system("pause");
第26行      return 0;
第27行  }
```

在template<typename T>void for_each(T *Beg,T *End)中，T是数据类型的代称，在源代码编译过程中，会根据总体代码使用for_each()的情况，替换为相应的数据类型，如果是int则T被替换为int，如果是double，则被替换为double，如果是class或者结构，也同样替换为定义class或者struct的名称。template在英语中被称为模板，即本意即为“照此刻画”。在编译过程中，将根据使用for_each()的实际，依照定义，自动翻译成一个或多个对应的for_each()函数。

for_each()功能常用，有通用化需求，从其定义来看，Beg和End是数组的起点和终点，完全适应数组成员数量的变化，而处理数据的print()没有参数化，因此很没有灵活性。例程4-11将for_each()的定义变为template<typename T,typename T1>void for_each(T *Beg,T *End,T1 Exec)，其中增加了typename T1，其中T1用于修饰Exec。函数体内的while(p!=End)print(*p++)也变化为

while(p!=End)Exec(*p++)，其中print变化为Exec。第20行、22行是for_each()函数的应用，由此可以看出，实现了函数名称的参数化，其原因是函数名称实际上是函数指针，T1在此处相当于代表函数指针类型。

例程4-11

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  void printA(const int &val){
第5行      cout<<val<<endl;
第6行  };
第7行  void printB(const int &val){
第8行      cout<<val*val<<endl;//平方
第9行  };
第10行
第11行 template<typename T,typename T1>
第12行 void for_each(T *Beg,T *End,T1 Exec){//遍历数据
第13行     T *p=Beg;
第14行     while(p!=End)Exec(*p++);
第15行 };
第16行 int main(){
第17行     int myArr[]={10,20,1,7,9,88,99,12,209};
第18行     int count=sizeof(myArr)/sizeof(myArr[0]);
第19行
第20行     for_each(myArr,myArr+count,printA);//执行for_each()函数
第21行
第22行     for_each(myArr,myArr+count,printB);//执行for_each()函数
第23行
第24行     system("pause");
第25行     return 0;
第26行 }
```

例程4-11仍有不足，Exec如果能模板化，则能适应更多数据类型，不然将局限于整型，修改如例程4-12。

例程4-12

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  template<typename T>
第4行  void print(const T &val){
第5行      cout<<val<<endl;
第6行  };
第7行
第8行  template<typename T,typename T1>
第9行  void for_each(T *Beg,T *End,T1 Exec){//遍历数据
第10行     T *p=Beg;
第11行     while(p!=End)Exec(*p++);
第12行 };
```

```
第13行  int main(){
第14行      int myArr[]={10,20,1,7,9,88,99,12,209};
第15行      int count=sizeof(myArr)/sizeof(myArr[0]);
第16行
第17行      for_each(myArr,myArr+count,print<int>());//执行for_each()函数
第18行
第19行      double hisArr[]={1.23,20.12,1.98,7.87,9.1,88.01,99.98,12.87,209.876,0.98,5.25};
第20行      int size=sizeof(hisArr)/sizeof(hisArr[0]);
第21行
第22行      for_each(hisArr,hisArr+count,print<double>());//执行for_each()函数
第23行
第24行      return 0;
第25行  }
```

观察第11行代码while(p!=End)Exec(*p++)，Exec()一般都认为是函数，但是括号运算符重载也能实现，如例程4-13所示。括号运算符重载意义重大，括号运算符是类中函数成员，而类还可以包含其他数据成员和函数成员，为拓展其功能提供方便。

例程4-13

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   class OP{
第5行   public:
第6行       void operator()(int i){
第7行           cout<<i<<endl;
第8行       }

第9行   };

第10行  template<typename T,typename T1>
第11行  void for_each(T *Beg,T *End,T1 Exec){//遍历数据
第12行      T *p=Beg;
第13行      while(p!=End)Exec(*p++);
第14行  };
第15行  int main(){
第16行      int myArr[]={10,20,1,7,9,88,99,12,209};
第17行      int count=sizeof(myArr)/sizeof(myArr[0]);
第18行
第19行      for_each(myArr,myArr+count,OP());//执行for_each()函数
第20行
第21行      system("pause");
第22行      return 0;
第23行  }
```

例程4-13中的OP仅能支持整数，类也可以进行模板化改造，以支持各种类型的数据，如例程4-14所示。

例程4-14

```
第1行   #include<iostream>
```

```
using namespace std;
template<typename T>
class OP{
public:
    void operator()(T i){
        cout<<i<<endl;
    }
};
template<typename T,typename T1>
void for_each(T *Beg,T *End,T1 Exec){//遍历数据
    T *p=Beg;

    while(p!=End)Exec(*p++);
};

int main(){
    int myArr[]={10,20,1,7,9,88,99,12,209};
    int count=sizeof(myArr)/sizeof(myArr[0]);

    for_each(myArr,myArr+count,OP<int>());//执行for_each()函数

    double hisArr[]={1.23,20.12,1.98,7.87,9.1,88.01,99.98,12.87,209.876,0.98,5.25};
    int size=sizeof(hisArr)/sizeof(hisArr[0]);

    for_each(hisArr,hisArr+size,OP<double>());//执行for_each()函数

    return 0;
}
```

例程4-13和例程4-14都是用对象的括号运算符重载，实现类似函数功能，其书写形式也很像函数，因此将这类对象称之为函数对象，或者称为仿函数、函数子等，其实质就是类实例化为对象，并在该类中重载括号运算符。使用函数对象，函数更加灵活。如例程4-15所示。

例程4-15

```
#include<iostream>
using namespace std;
template<typename T>
class OP{
public:
    int Up,Down;
    OP(int U,int D):Up(U),Down(D){};
    void operator()(T i){//在Up和Down之间的数据输出IN，否则输出OUT
        if(i<Up || i>Down)
            cout<<"IN"<<endl;
        else
            cout<<"OUT"<<endl;
    }
```

```
第14行  };
第15行  template<typename T,typename T1>
第16行  void for_each(T *Beg,T *End,T1 Exec){//遍历数据
第17行      T *p=Beg;
第18行      while(p!=End)Exec(*p++);
第19行  };
第20行  int main(){
第21行      int myArr[]={10,20,1,7,9,88,99,12,209};
第22行      int count=sizeof(myArr)/sizeof(myArr[0]);
第23行
第24行      for_each(myArr,myArr+count,OP<int>(10,20));//执行for_each()函数
第25行
第26行      system("pause");
第27行      return 0;
第28行  }
```

例程4-15第24行for_each(myArr,myArr+count,OP(10,20))中的OP(10,20)看起来很想函数，但实际上是OP类的匿名对象，OP(10,20)只不过是调用该类的构造函数，具体的处理代码有OP类的括号运算付重载完成。可以想象，构造函数可以多种重载方式，其括号运算付重载也可以完成各种工作，因此，采用函数对象作为参数，是很好的选择。

for_each()函数的参数类型用T代替，这就意味着各种数据类型都有可能，比如：自己定义的各种链表是否也都适用for_each()函数呢？还是从函数代码说起。

例程4-16

```
第1行  template<typename T,typename T1>
第2行  void for_each(T *Beg,T *End,T1 Exec){//遍历数据
第3行      T *p=Beg;
第4行      while(p!=End)Exec(*p++);
第5行  };
```

在代码中有p!=End，意味着自定义类必须重载支持运算符!=；Exec(*p++)中p++则需要自定义类重载++运算符；*p则需要重载*运算符。当自定义数据类型进行了上述工作后，for_each()函数就可以支持其数据类型，例程4-17是用于遍历vector数据、数组数据。

例程4-17

```
第1行  #include<iostream>
第2行  #include<vector>

第3行  using namespace std;
第4行
第5行  template<typename T>
第6行  class levelScore{
第7行  public:
第8行      void operator()(T i){//在Up和Down之间的数据输出IN，否则输出OUT
第9行          if(i<60)cout<<i<<"分 不及格"<<endl;
第10行         if(i>=60 && i<85)cout<<i<<"分 良好"<<endl;
第11行         if(i>=85)cout<<i<<"分 优秀"<<endl;
第12行     }
第13行 };
第14行 template<typename T,typename T1>
```

```
第15行  void for_each(T Beg,T End,T1 Exec){//遍历数据
第16行      T p=Beg;
第17行      while(p!=End)Exec(*p++);
第18行  };
第19行  int main(){
第20行      cout<<"\nVector模式:\n";
第21行      vector<int> myVec;
第22行      for(int i=0;i<50;++i)
第23行          myVec.push_back(rand()%100);//0-99之间的数据，假定是成绩分布
第24行
第25行      for_each(myVec.begin(),myVec.end(),levelScore<int>());//执行for_each()函数
第26行
第27行      cout<<"\n数组模式:\n";
第28行      int myScore[50];
第29行      for(int i=0;i<50;++i)myScore[i]=rand()%100;
第30行      int count=sizeof(myScore)/sizeof(myScore[0]);
第31行      for_each(myScore,myScore+count,levelScore<int>());//执行for_each()函数
第32行
第33行      return 0;
第34行  }
```

观察例程4-17第14-18行代码，会发现Beg和End前都没有了*，但代码同样有效，这是什么原因呢？原因在于T是一个类型参数。在第25行代表myVec.begin()返回的数据类型，在第31行代表整型指针。

通过上述代码的学习，可以认识到，在模板化支持下，如果有通用的算法、函数对象、数据类型，以及遍历数据的指针，C++编程效率将大为提高，而这正是STL的目标。在STL体系中，提供最常用且经过优化的算法，提供组织数据的容器(类似Array，vector是其中一种)，提供遍历数据的迭代器，以及通用的函数对象(当然，还可以根据需要自定义)。

早期的C++没有模板，可以认为仅仅是C with class；有了模板的C++，实现大飞跃。

例程4-18是一个稍复杂些的模板应用，是上一章“类的嵌套”中Array的模板化改造，支持各种数据类型，例程4-18是其应用示例。从代码for(Array<int>::Iterator P=myArr.begin();P!=myArr.end();++P)可以看出，实现了类似vector<int>::Iterator的效果。Iterator是迭代器的意思，有些类似指针。例程4-18第9行可以看出，所谓Iterator是嵌套在Array中的一个类，其具体实现在第19-28行之间。由此可以推断，vector中的Iterator，实际上也是vector中的一个类(可以不是嵌套类)。

例程4-18中Array的函数成员和数据成员，不少在类体外定义，此时要注意表明其隶属关系的表达方式。如Iterator类是Array的嵌套类，在20行说明该类在Array之中，且该类是模板类，类型参数是T，因此要写为：Array<T>::Iterator，另外，T来源于模板，因此template<typename T>同样不能省略。不管是函数成员还是数据成员，采用类体外定义时，都必须采用类似的方式。

观察例程4-18第45行代码typename Array<T>::Iterator Array<T>::begin()，并对照第11行代码Iterator begin()，会发现返回的数据类型与往常很有不同。由于Iterator是定义在Array之中，因此也必须标明其隶属关系。可以想象，假定程序中有一个独立的名为Iterator的函数、类或者变量名称等会发生什么情况呢？肯定发生名称冲突。另外，由于Iterator类依赖于Array类，为了标明这种关系，需要在类型前增加typename关键字。注意：此处的typename不能用class替换。例程4-18第15-18行的Print()函数，是类型依赖另外一个应用示例，此处的typename同样不能省略。

例程4-18

```
第1行  //用模板实现数组
第2行  #ifndef ARRAYTEMPLATE_H
第3行  #define ARRAYTEMPLATE_H
第4行  template<typename T>
```

```
第5行  class Array{
第6行  public:
第7行      Array(int size=100);
第8行      ~Array();
第9行      class Iterator;//定义一个Iterator类
第10行     T &operator[](unsigned int i);

第11行     Iterator begin();
第12行     Iterator end();
第13行     int size(){return this->count;}
第14行 private:
第15行     T *addrStart;//指针起点
第16行     T *addrTail;//指针终点
第17行     int count;//成员数量
第18行 };
第19行 template<typename T>
第20行 class Array<T>::Iterator{
第21行     public:
第22行         Iterator(T *p){this->currentPos=p;}
第23行         T &operator*(){return *currentPos;}
第24行         Iterator &operator++();
第25行         bool operator!=(const Iterator &itr);
第26行     private:
第27行         T *currentPos;
第28行 };
第29行 //注意：参数默认值不能在此设置
第30行 template<typename T>
第31行 Array<T>::Array(int size):count(size){
第32行         this->addrStart=new T[this->count];
第33行         this->addrTail=this->addrStart+this->count;
第34行 }
第35行 template<typename T>
第36行 Array<T>::~Array(){
第37行         delete []this->addrStart;
第38行 }
第39行 template<typename T>
第40行 T &Array<T>::operator[](unsigned int i){
第41行         return *(this->addrStart+i);

第42行 }
第43行 //Iterator是定义在Array中，因此必须加上域运算符
第44行 template<typename T>
第45行 typename Array<T>::Iterator Array<T>::begin(){//通过Array中begin()成员函数建立起与其内的Iterator之间的关系
第46行     return Array<T>::Iterator(this->addrStart);
第47行 }
```

```
template<typename T>
typename Array<T>::Iterator Array<T>::end(){//通过Array中end()成员函数建立起与其内的Iterator之间的关系
    return Array<T>::Iterator(this->addrTail);
}
template<typename T>
typename Array<T>::Iterator &Array<T>::Iterator::operator++(){
    this->currentPos++;
    return *this;
}
template<typename T>
bool Array<T>::Iterator::operator!=(const Iterator &itr){
    return this->currentPos!=itr.currentPos;
}

#endif
```

例程4-19

```
//应用模板数组
#include<iostream>
#include"ArrayTemplate.h"
using namespace std;
class UpDown{
public:
    UpDown(int U=1,int D=1):Up(U),Down(D){}

    friend ostream &operator<<(ostream &out,const UpDown & UD);
private:
    int Up,Down;
};
ostream &operator<<(ostream &out,const UpDown & UD){
    return out<<"("<<UD.Up<<"/"<<UD.Down<<")";
}
template<typename T>
void Print(typename Array<T>::Iterator &P){
    cout<<*P<<endl;
}
//与上面的Print函数同名，存在函数重载
template<typename T>
void Print(const T& val){
        cout<<val<<endl;
}
int main(){
    Array<int> myArr;

    for(int i=0;i<myArr.size();++i)
```

```
第28行            myArr[i]=rand()%100;//需重载Array类的方括号即下标运算符
第29行
第30行        for(Array<int>::Iterator P=myArr.begin();P!=myArr.end();++P)//需重载Array中Iterator的!=和++运算符
第31行            cout<<*P<<endl;//需重载Array中Iterator的*运算符
第32行
第33行        Array<UpDown> arrUD(20);
第34行        for(int i=0;i<arrUD.size();++i)
第35行            arrUD[i]=UpDown(rand()%100,rand()%100);//需重载Array类的方括号即下标运算符
第36行
第37行
第38行        for(Array<UpDown>::Iterator P=arrUD.begin();P!=arrUD.end();++P)//需重载Array中Iterator的!=和++运算符
第39行            cout<<*P<<endl;//需重载Array中Iterator的*运算符
第40行        cout<<"\n\n";
第41行
第42行        Print<UpDown>(++++arrUD.begin());
第43行        Print(arrUD[0]);
第44行
第45行        return 0;
第46行    }
```

第四节　再识模板

第五节　小结

模板是C++重要特性，是STL的基础，是实现算法、数据结构、函数对象的重要基础，是泛型程序设计的重要手段。模板分为函数模板和类模板。模板是独立于类型的母版，编译器根据应用函数或类的具体场景，产生具体类型的实例。

函数模板是STL算法库的基础；类模板是STL容器和迭代器的基础，也是函数对象的基础。

第六节　阅读材料