第二十章　复习提纲

本章导读

C++源于C语言的面向对象编程语言，广泛支持泛型程序设计。虽然源于C语言，但已有很大变化，具有良好的工程性，常用于系统开发，引擎开发等应用领域，是至今为止最受广大程序员欢迎的最强大编程语言之一。

本章主要内容是提纲挈领地复习C++。

学习目标：

1. 回顾C++语言；
2. 回顾函数定义、函数重载等重要概念；
3. 回顾类和对象，尤其是类中的特殊函数；
4. 回顾STL对C++的工程重要性；

本章目录

第一节　数据类型

C++基本的数据类型有：void(无值型)、bool(布尔型)、short [int]/singed short [int]/unsigned short [int]、int/singed int/unsigned int、long [int]/singed long [int]/unsigned long [int]、long long、char/singed char/unsigned char、float、double、long double。另外在C++11中，针对字符串有一些新的数据类型。

指针是一种数据类型，尤其要熟悉指向函数的函数指针，如：void (*fPtr)(int,int)，是一个指向二元函数(以两个int为参数，返回类型为void)的指针，可以指向符合要求的一些列函数。另外，函数名称是指针。

C++有引用数据类型。引用有指针之实，普通名称变量之便。引用在建立之初必须指定引用关系，且不能在使用过程中变更引用关

系，和指针相比，更具有安全性。

理解左值(Location Value,lValue)、右值(Reading Value,rValue)。右值引用是C++11的重要语言特性，与move语义、纯右值、将亡值等概念相关。

类是一种用户定义类型，在地位上与普通数据类型具有极大相似性。

1、类型转换

C++中常见数据类型转换，有隐式转换，也有系列cast函数的显式转换、更有通过定义类的成员函数完成转换。例程20-1是一个数据类型的隐式转换。

例程20-1

```
第1行  #include<iostream>
第2行  #include<typeinfo>
第3行  using namespace std;
第4行
第5行  int main(){
第6行          unsigned int i=100;
第7行          cout<<-1*i<<endl;//输出：4294967196
第8行          cout<<typeid(-1*i).name()<<endl;//输出：unsigned int
第9行
第10行         int a=100;
第11行         cout<<-1*a<<endl;//输出：-100
第12行         cout<<typeid(-1*a).name()<<endl;//输出：int
第13行
第14行         return 0;
第15行 }
```

注意第7行，i=100，-1*i理当为-100，但由于i为无符号整数，因此将-1*i也变为无符号整数，-100的原码为：1000 0000 0000 0000 0000 0000 0110 0100，其反码为：11111 1111 1111 1111 1111 1111 1001 1011，其补码为：11111 1111 1111 1111 1111 1111 1001 1100。负数在内存中以补码形式存在，当将该负数视为无符号整数时，则原来最高位用于表示正负号的符号位用于表达正数或零，因此，此时输出4294967196。

对于二元运算符，C++允许两个操作数是不同类型，当操作数类型不同时，C++将自动对数据类型进行转换，这种转换称为隐式转换，当被转换的数据类型为C++基本类型时，其规则如下：

- 当其中一个操作数是long double时，则将另外一个转换为long double；
- 当其中一个操作数是double时，则将另外一个转换为double；
- 当其中一个操作数是float时，则将另外一个转换为float；
- 当其中一个操作数是unsigned long时，则将另外一个转换为unsigned long；
- 当其中一个操作数是long时，则将另外一个转换为long；
- 当其中一个操作数是unsigned int时，则将另外一个转换为unsigned int；
- 如不满足上述规则，则都转换为int类型。

对于例程20-1第7行，-1为int，i为unsigned int，因此被转换为unsigned int。

除了隐式类型转换，C++还支持显示类型转换，如例程20-2所示，其中第7、8行是C模式类型转换，第8行为C++模式的类型转换。在C++中，当需要类似转换时，建议采用static_cast。

例程20-2

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int main(){
第5行          unsigned int i=100;
```

```
第6行          cout<<-1*(int)i<<endl;//输出：-100
第7行          cout<<-1*int(i)<<endl;//输出：-100
第8行          cout<<-1*static_cast<int>(i)<<endl;//输出：-100
第9行
第10行         return 0;
第11行     }
```

C++数据类型转换除了static_cast外，还有dynamic_cast和reinterpret_cast，其中的dynamic_cast的格式如下：

dynamic_cast<typeid>(Expression)

dynamic_cast把Expression转换成typeid类型的对象。typeid必须是类的指针、类的引用或者void*。如果typeid是类指针，那么Expression也必须是一个指针，如果typeid是一个引用，那么Expression也必须是一个引用。dynamic_cast主要用于类层次间的向上转换或向下转换。当进行向上转换时，dynamic_cast和static_cast的效果相同；当进行向下转换时，dynamic_cast具有类型检查，比static_cast更安全。例程20-3是dynamic_cast与static_cast的应用示例。

例程20-3

```
第1行  #include<iostream>
第2行  #include<typeinfo>
第3行  using namespace std;
第4行  struct classRoot{
第5行          classRoot(){cout<<"I am in classRoot.\n";}
第6行          virtual void funA(){cout<<"I am in funA()@classRoot.\n";};
第7行  };
第8行  struct classBranchA:public classRoot{
第9行          int num;
第10行         classBranchA(){this->num=100;cout<<"I am in classBranchA.\n";}
第11行         void funA(){cout<<"this->num="<<this->num<<". I am in funA()@classBranchA.\n";}
第12行 };
第13行 struct classBranchB:public classRoot{
第14行         int num;
第15行         classBranchB(){this->num=99;cout<<"I am in classBranchB.\n";}
第16行         void funA(){cout<<"this->num="<<this->num<<". I am in funA()@classBranchB.\n";}
第17行 };
第18行 void funTest(classRoot &refObj){
第19行         cout<<"\n\n";
第20行
第21行         cout<<typeid(refObj).name()<<endl;
第22行         refObj.funA();
第23行
第24行         classBranchB &refB=static_cast<classBranchB&>(refObj);
第25行         classBranchA *refA=dynamic_cast<classBranchA*>(&refObj);
第26行
第27行         refB.funA();
第28行         if(refA!=NULL)refA->funA();
第29行 }
第30行 int main(){
```

```
第31行          classBranchA a;
第32行          classBranchB b;
第33行          funTest(a);funTest(b);
第34行
第35行          classBranchA *pA=static_cast<classBranchA *>(&a);//如将a修改b将导致编译错误或者语法错误
第36行          classBranchB *pB=dynamic_cast<classBranchB *>(&a);//此处a或b均可，当a时，pB为NULL
第37行
第38行          return 0;
第39行  }
```

和static_cast相比，dynamic_cast要求基类必须有虚函数，而static_cast则没有该限制。dynamic_cast有类型检查。当将第25行修改为classBranchA &refA=dynamic_cast<classBranchA&>(refObj)，则当refObj为classBranchB时，将触发std::bad_cast异常；当不进行修改时，则当refObj为classBranchB时，refA的值将为NULL，删除第28行前的if(refA!=NULL)将报错。和static_cast相比，dynamic_cast更加安全。

另外，dynamic_cast支持交叉转换，如例程20-3第36行所示。当将classBranchA类型的对象a转换为classBranchB时，其结果为null，没有语法错误；当将&a改变为&b时，则正确转换。对第35行的static而言，不支持交叉转换，即将&a改变为&b，则将出现编译错误或者语法错误。

reinterpret_cast也用于类型转换，其格式如下。reinterpret_cast可以将任意指针类型转换成任意指针类型，即便两者没有任何关系，转换结果是二进制的简单拷贝。reinterpret_cast可以将指针转换为整数，或将整数转换为指针。整数值所代表的指针与平台相关，仅仅保证指针转换为整数时有足够容量存储整数，并保证将整数转回为合法的指针。reinterpret_cast转换是一种低层次操作，其转换为该类型的二进制表达，与平台相关，跨平台性不强。例程progCast01是reinterpret_cast的使用示例。

reinterpret_cast<typeid>(Expression)

reinterpret_cast常用于将各种类型的值转换为char *形式，其本质是该类型在内存中的值转换为字符指针型，类型的长度即为该字符指针所指向内容的长度，这种特征非常适合于将各种类型包括结构或类类型的数据以二进制形态写入文件，然后以二进制形态输入到程序中，如例程20-4所示。

例程20-4

```
第1行   #include<iostream>
第2行   #include<typeinfo>
第3行   #include<fstream>
第4行   using namespace std;
第5行   struct XYZ{
第6行           XYZ(){};
第7行           XYZ(int inA,double inB):a(inA),b(inB){}
第8行           int a;
第9行           double b;
第10行  };
第11行  int main(){
第12行          XYZ xyz(1000,99.99),readingData;
第13行          fstream myFile("test.dat",ios::binary|ios::out);
第14行          myFile.write(reinterpret_cast<char *>(&xyz),sizeof(XYZ));
第15行          myFile.close();
第16行
第17行          myFile.open("test.dat",ios::binary|ios::in);
第18行          myFile.read(reinterpret_cast<char *>(&readingData),sizeof(XYZ));
```

```
第19行          myFile.close();
第20行
第21行          cout<<readingData.a<<" "<<readingData.b<<endl;
第22行
第23行          return 0;
第24行  }
```

const_cast用于修改类型的const或volatile属性，其格式如下，其中除const修饰符外，typeid与Expression类型一致。const_cast有两种功能，一是将常量指针转化为非常量的指针，并且仍然指向原来的对象；二是将常量引用转换为非常量引用，并且仍然指向原来的对象。例程20-5是其应用示例。从示例可以看出，const_cast只能对类、结构和指针做从const到非const的转换，对基本数据类型，虽然能完成转换，但通过指针不能修改其值(第20-23行)。

const_cast<typeid>(Expression)

例程20-5

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   struct UpDown{
第4行           UpDown(){this->Up=1;this->Down=1;}
第5行           int Up,Down;
第6行   };
第7行   int main(){
第8行           const UpDown UD;
第9行
第10行          UpDown *udA=const_cast<UpDown *>(&UD);
第11行          udA->Up=1;udA->Down=2;
第12行          cout<<"udA="<<"("<<udA->Up<<"/"<<udA->Down<<")"<<endl;//输出:udA=(1/2)
第13行          cout<<"UD="<<"("<<UD.Up<<"/"<<UD.Down<<")"<<endl;//输出:UD=(1/2)
第14行
第15行          UpDown &udB=const_cast<UpDown &>(UD);
第16行          udB.Up=2;udB.Down=3;
第17行          cout<<"udB="<<"("<<udB.Up<<"/"<<udB.Down<<")"<<endl;//输出:udB=(2/3)
第18行          cout<<"UD="<<"("<<UD.Up<<"/"<<UD.Down<<")"<<endl;//输出:UD=(2/3)
第19行
第20行          const int a=100;
第21行          int *pA=const_cast<int *>(&a);
第22行          *pA=200;
第23行          cout<<"a="<<a<<" *pA="<<*pA<<endl;//输出a=100 *pA=200
第24行
第25行          return 0;
第26行  }
```

2、enum

3、const

const是英文单词constant的简写，其含义是“常数、常量、不变的事物、永恒值”，在C++程序中，常用于不允许变化的场所，如例程20-6所示，其第15、22行的函数名称int getUp()const和int getDown()const后均出现const关键字，其功能是在函数作用范围内不能修改类的数据成员，实现对数据成员的保护，防止函数代码修改数据成员的值。

```
#include<iostream>
using namespace std;
class UpDown{
public:
        UpDown(int Up=1,int Down=1){
                this->upNum=Up;
                this->downNum=Down;
        }
        void setUp(int Up){
                this->upNum=Up;
        }
        //const的使用表明不允许类的数据成员被修改
        //int不能修改为int &，const的修饰符的使用将const化数据成员
        //存在将const int转换为int &的错误
        int getUp()const{
                return this->upNum;
        }
        int getUp(){
                return this->upNum;
        }
        //没有const，可以修改this->downNum的值；
        int getDown()const{
                std::cout<<"...const!"<<std::endl;
                return this->downNum;
        }
        int &getDown(){
                std::cout<<"...NO const!"<<std::endl;
                return this->downNum;
        }
private:
        int upNum,downNum;
};
//注:函数名不能使用const修饰符，非成员函数不能使用类型修饰符
//参数使用const限定形参，表明函数内部不能修改该参数的值
void printUD(const UpDown &UD){
        //UD.setUp(10);//注：错误！对象包含与成员函数不兼容的类型限定符
        std::cout<<UD.getUp()<<"/"<<UD.getDown()<<std::endl;
}
int main(){
        UpDown UD0(2,3);
        std::cout<<UD0.getDown()<<std::endl;//会输出...NO const!
        printUD(UD0);//会输出...const!
        UD0.getDown()=5;//会输出...NO const!
        printUD(UD0);//会输出...const!
```

```
第45行          system("pause");
第46行          return 0;
第47行  }
```

注意观察第15-17行与第18-20行以及第22-25行与第26-29行，会发现getUp()函数除const外完全相同，其原因是const也是实现函数重载的要素。即如果函数参数完全相同，但有const区别，也是不同的函数。在观察main()函数能发现，程序能自动选择正确的函数形式。

当const出现在函数的参数前时，将保护参数不能被修改，如例程20-6第27行void printUD(const UpDown &UD)所示。在该函数中，const的应用将使形参UD在printUD函数中不能被修改，是UD值的保护。在实际编程过程中，如果输入的参数不希望被修改，加上const是良好的工程习惯。

当const与指针变量一起使用时，情况将变得略微复杂，如例程20-7所示，const可以出现符号*之前，也可以在其后。前后位置不同，其含义也不同。

例程20-7

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   int main(){
第5行           int numZ=100;
第6行           int numA=200;
第7行           int numB=300;
第8行           int numC=400;
第9行
第10行          int *pZ=&numZ;//获得numZ的地址
第11行          int *const pA=&numA;//获得numA的地址
第12行          const int *pB=&numB;//获得numB的地址
第13行          const int *const pC=&numC;//获得numC的地址
第14行
第15行          pZ=&numA;//注意：允许!pZ改指向numA
第16行          //pA=&numB;//注意：错误!pA的地址值为常量，不能修改，即不能调整为numB的地址；
第17行          pB=&numA;//注意：允许!pB的地址值不为常量，可以指向其他地址；
第18行          //pC=&numA;//注意：错误!pC的地址值为常量，不能修改，即不能调整为numA的地址；
第19行          cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第20行          cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第21行          cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第22行          cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第23行
第24行          *pZ=111;//注意：常规应用，可以修改内容，可以改变地址，在前面pZ已指向numA
第25行          *pA=222;//注意：不可改变地址，但可以改变内容
第26行          //*pB=333;//注意：可以改变地址，不可改变内容，在前面pB已指向numB
第27行          //*pC=444;//注意：不可改变地址，不可改变内容
第28行          cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第29行          cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第30行          cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第31行          cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第32行
第33行          //num系列变量值变化不受影响
```

```
第34行          numZ=199;
第35行          numA=299;
第36行          numB=399;
第37行          numC=499;
第38行          cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第39行          cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第40行          cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第41行          cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第42行
第43行          return 0;
第44行      }
```

当const用于修饰函数的返回值时，如const int f1()相当于const int value=f1()，即函数的返回值为常量，不能被修改。当const用于修饰的函数返回值带有指针时，形如const int *f1()、int *const f2()或const int *const f3()对应相当于const int *p1=f1()、int *const p2=f2()或const int *const p3=f3()，其含义与变量相似，在例程20-7中，其地址来源于变量，用于修饰函数返回值时，则地址来源函数返回值。

当const用于修饰函数的参数时，其含义与例程20-7相似，都是变量的地址。

4、auto

5、decltype

6、typedef

7、typename

8、sizeof

9、static

10、mutable

11、typeid

第二节　运算符

C++运算符与C语言几乎有相同的运算符，但由于C++支持运算符重载，因此需要更加深入了解运算符。重载之后的运算符其优先级和结合性都不会改变，并且尽量保持与原运算符有相同或近似的功能。在C++中，不能重载的运算符有5个，分别是：成员运算符“.”、指针运算符“*”、作用域运算符“::”、“sizeof”、条件运算符“?:”，另外，new和delete在C++中也是运算符。

运算符重载只针对类数据类型(struct也是类的一种形式)，要求至少要有一个操作对象是类类型。即便没有重载等号运算符，但对于自定义的类类型，也同样可以适用，但与拷贝构造函数一样，同样要考虑类似浅拷贝和深拷贝。

运算符重载形式有两种，重载为类的成员函数和重载为类的友元函数，如果既不为成员函数和友元函数，则视之为普通函数。有些运算符只能定义为成员函数，如：[]、()、->、=(还包括赋值一系列运算符，如：+=、*=等)。另外，当运算符重载为成员函数时，默认第一个参数为对象本身。

在进行运算符重载时，要考虑返回值类型，尤其是要准确判断是返回值、指针还是引用。这与该运算符的一般使用场景密切相关，如：=、[]、*等运算符。另外，还要考虑运算符的参数个数，一般说来，运算符有确定的参数个数，一元运算符有一个参数，二元运算符有两个参数，三元运算符有三个参数。另：()运算符的个数可以是一个、二个也可以是多个，且该运算符与函数对象等密切相关，值得重点关注。

第三节　控制语句

C++的控制语句与C语言近乎完全相同，不过在C++11中，增加了序列for循环。使得在C++中可以使用类似java、C#的简化for循环，可以用于遍历数组，容器，string以及由begin和end函数定义的序列(即有Iterator)，示例代码如下：

例程20-8

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
```

```
int main(){
        int arrList[] = { 1, 4, 12, 56, 12, 454 };
        for (auto member : arrList)
                cout << member << "\t";

        return 0;
}
```

使用序列for循环，无需考虑越界问题，将更加具有安全性。

例程20-9

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

int main(){
        map<string, int> m{ { "a", 1 }, { "b", 2 }, { "c", 3 } };
        for (auto p : m){
                cout<< p.first << " : " << p.second << endl;
        }
}
```

第四节　名称空间

第五节　函数

函数是高级的语言的重要特征，也是代码复用的重要措施。在C/C++中，函数具有三个要素：函数名称、函数参数(可以使零参数)和函数返回值类型(函数代码中用return返回函数值，可以是void类型)。和C语言不同，在C++中，多个功能相同或相近的函数可以共享一个名称，但必须能通过函数参数进行区分，可以参数个数不同、类型不同或者类型顺序不同，此为函数重载。

申明函数时的参数为形式参数，调用函数时传入的参数为实参。实参传递有三种方式：数值传递、地址传递、引用传递。其中地址传递和引用传递都是使用同一内存地址中的值，因此，函数内部的变化将影响到函数外部。

在函数定义中，需注意const的使用。const的基本含义是不变。当参数前有const时，即说明在函数体内不能修改该参数的值。如果返回类型前含有const，则其返回值不能函数体外的代码修改。

1、函数模板

模板是C++最重要特征之一，包括函数模板和类模板，例程20-10是函数模板。在该模板中，可以实现两种类型的数相加，这两种类型的数可以是int类型的数相加，也可以是double类型的数相加，也可以是分数类型的数相加(注意，如果是分数类型，需要定义分数类，且需要重载加法运算符)。在例程20-10中，函数的返回类型为auto，其真实返回类型有decltype(a+b)确定。如果是两个int类型相加，则返回类型为int；如果是double类型相加，则为double类型；如果是分数类型相加，则为分数类型。

例程20-10

```
template<typename T1,typename T2>
auto Add(T1 a, T2 b)->decltype(a + b){
        return a + b;
}
```

注意观察例程20-11，定义了两个Add()函数，其中一个为模板函数，一个为常规函数。此时，优先匹配常规函数，因此可以将特殊要求的函数设定为常规函数。如：Add('a',2)的目的是想得到a后面第二个字符，则可以写成常规函数。编译器会根据场景优先选择精确匹配的常规函数。

例程20-11

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  template<typename T1,typename T2>
第4行  auto Add(T1 a, T2 b)->decltype(a + b){
第5行          cout << "AAA" << endl;
第6行          return a + b;
第7行  }
第8行  char Add(char a, int b){
第9行          cout << "BBB" << endl;
第10行         return a + b;
第11行 }
第12行 int main(){
第13行         cout << Add('a', 2) << endl;//输出：c
第14行         cout << Add(2,'a') << endl;//输出：99
第15行
第16行         return 0;
第17行 }
```

2、lambda函数

lambda函数是C+11新增的重要特征，如例程20-12第9、10、11以及15行所示，均是lambda函数的应用示例。在C++中，lambda函数以[]为标志，其后的括号中为参数列表，与常规函数的约定一致。如果采用引用模式，则lambda函数内外共享或指向相同的内存空间，如第10、11行所示。另外，[]内可以设置捕获列表，可以将外部变量的值传入lambda函数内容，同时也可以采用引用等模式传递参数。

例程20-12

```
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  int main(){
第7行          vector<int> myVec;
第8行          for (int i = 0; i < 100; ++i)myVec.push_back(i);
第9行          for_each(myVec.begin(), myVec.end(), [](int i){if (i % 2 == 0)cout << i << "\t"; });//仅仅输出偶数
第10行         for_each(myVec.begin(), myVec.end(), [](int i){i += 1; cout << i << "\t"; });//每个值都被加1；
第11行         for_each(myVec.begin(), myVec.end(), [](int &i){i += 1; cout << i << "\t"; });
第12行         //与上行显示效果一致，但myVec数据同时被更改
第13行
第14行         int a = 10;
第15行         for_each(myVec.begin(), myVec.end(), [a](int &i){i += a; cout << i << "\t"; });
第16行         //数据更改量与传入的a值相关，但myVec数据同时被更改
第17行
第18行         return 0;
第19行 }
```

lambda函数对于短小的处理代码具有很好的工程友好性。

3、函数对象

请参见类和对象

第六节　类和对象

类和对象是面向对象程序设计的基础，类是对象的抽象，对象是类的实例。在C++中，类的关键字是class，同时C语言的struct在C++中也是类。在C++，类的封装模式有public、protected、private。当省略封装模式时，C++默认为private，struct默认为public。在C++中，类中可以封装数据成员和函数成员，这和C语言struct有很大的不同。

1、特殊函数

构造函数和析构函数是类的特殊函数。构造函数的名称与类名称完全相同，而析构函数的名称则是在类名称前增加~符号。构造函数是类实例化为对象，即对象从无到有的过程首先执行的函数，即便没有定义构造函数，编译器也会自动生成无参构造函数。析构函数是对象消亡过程中自动执行的函数，有且仅有一个。

根据需求不同，一个类可以有多个构造函数，如例程20-13所示。在该例程中，有三个构造函数一个析构函数。注意：构造函数没有返回值，或者可以认为生成的对象就是构造函数的返回类型和返回值。

例程20-13

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class UD{
第4行  public:
第5行          UD() :U(0), D(1){ ; }//无参构造函数
第6行          UD(int u, int d) :U(u), D(d){ ; }//双参构造函数
第7行          UD(int u) :U(u), D(1){ ;}//单参构造函数
第8行          ~UD(){ ; }//析构函数
第9行  private:
第10行         int U, D;
第11行 };
第12行 int main(){
第13行         UD udA =1;
第14行         UD udB = { 1, 2 };
第15行         UD udC;
第16行
第17行         return 0;
第18行 }
```

关键字explicit常被应用在构造函数之前也只能引用在构造函数之前，禁止隐式转换，必须采用显示调用。当没有explicit时，如果等号后面仅有一个值，编译器会自动查找仅有一个参数的构造函数，如果是两个值，则查找双参数的构造函数。

例程20-14

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class UD{
第4行  public:
第5行          UD() :U(0), D(1){ ; }//无参构造函数
第6行          explicit UD(int u, int d) :U(u), D(d){ ; }//双参构造函数
第7行          explicit UD(int u) :U(u), D(1){ ; }//单参构造函数
第8行          ~UD(){ ; }//析构函数
第9行  private:
第10行         int U, D;
第11行 };
第12行 int main(){
```

```
第13行          UD udA (1);//写为UD udA=1为错误；
第14行          UD udB(1, 2);//写为UD udB = { 1, 2 }为错误；
第15行          UD udC;
第16行
第17行          return 0;
第18行  }
```

有时，隐式转换未必是开发人员所需，此时就最好使用explicit参数，如例程20-15所示。如果第7行构造函数之前没有explicit，则第16行可以写为Array<int> arrData=10，此时的10到底是这个数组的成员值为10呢，还是开辟10个能包含10成员的空间，此时容易带来的歧义，最好使用explicit。

例程20-15

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   template<typename T>
第5行   class Array{
第6行   public:
第7行           explicit Array(unsigned int N) :pData(new T[N]){ ; }
第8行           ~Array(){
第9行                   delete[]pData;
第10行          }
第11行  private:
第12行          T* pData=nullptr;
第13行  };
第14行
第15行  int main(){
第16行          Array<int> arrData(10);
第17行
第18行          return 0;
第19行  }
```

析构函数不一定显式申明，当没有显式申明时，编译器将自动生成析构函数。但如例程20-15所示情况，则必须申明析构函数，否则将导致内存泄露，即开辟在堆的内存没有被正确收回，且不能被其他程序使用。

构造函数分为：常规构造函数、拷贝构造函数、转移构造函数。常规构造函数比较好理解，此处不再阐述。拷贝构造函数是根据一个已有的对象复制生成一个新的对象，如例程第8-12行所示。拷贝构造函数的参数类型必须为当前类，且一般要求为引用并加上const修饰符。有些编译器如果省略const将提示warning或者error。

在例程第36行，将不会执行拷贝构造函数，原因是UD(1,2)生成的匿名对象将很快消失，鉴于此，编译器会直接让udC接管该匿名对象。注：运行该程序，仔细观察输出，将有助于理解构造函数、拷贝构造函数、参数传递等过程。

2、move语义与类

转移构造函数与将亡值、右值引用密切相关。如果类的设计含有用new在堆中分配内存块，合理使用转移构造函数、转移赋值函数将大大提升效率，如例程20-16所示，其Array是一个模板类，实现简单数组功能，用new在堆中分配内存。当执行第70行时，Array<int>(20)将生成一个匿名对象，执行完该语句后，将消失不用，也就是所谓的将亡值。与常规赋值函数相比，转移赋值函数效率将大大提高。

第72行将执行Test()函数。在函数体内，将生成临时变量tmp，当函数执行完毕返回值时，将调用转移构造函数。和拷贝构造函数相比，转移构造函数效率明显高。返回值将赋值给itsArr，且是将亡值，因此调用转移赋值函数。

move()将强制调用转移构造函数，如第75行所示。

例程20-16

```
#include<iostream>
using namespace std;
template<typename T>
class Array{
public:
	Array() :Count(0), pData(nullptr){ cout<<"Array() :Count(0), pData(nullptr)\n"; }//无参构造函数
	explicit Array(unsigned int N) :Count(N), pData(new T[N]){
		cout<<"explicit Array(unsigned int N) :Count(N), pData(new T[N])\n" ;
		for (unsigned int i = 0; i < N; ++i)*(this->pData + i) = N + i;
	}//单参构造函数

	//拷贝构造函数
	Array(const Array & argRight){
		cout << "Array(const Array & argRight)\n";
		this->Count = argRight.Count;
		if (this->Count == 0){
			this->pData = nullptr;
		}
		else{
			this->pData = new T[this->Count];
			for (unsigned int i = 0; i < this->Count; ++i)*(this->pData + i) = *(argRight.pData + i);
		}
	}
	//赋值函数
	Array& operator=(const Array &argRight){
		cout << "Array& operator=(const Array &argRight)\n";
		if (this->Count != 0)delete[]this->pData;

		this->Count = argRight.Count;
		this->pData = new T[this->Count];
		for (unsigned int i = 0; i < this->Count; ++i)*(this->pData + i) = (argRight.pData + i);
		return *this;
	}
	//转移构造函数
	Array(Array &&argRight){
		cout << "Array(Array &&argRight)\n";
		this->Count = argRight.Count;
		this->pData = argRight.pData;
		argRight.pData = nullptr;
		argRight.Count = 0;
	}
	//转移赋值函数
	Array& operator=(Array &&argRight){
		cout << "Array& operator=(Array &&argRight)\n";
```

```
第45行
第46行                  if (this->Count!= 0)delete[]this->pData;
第47行
第48行                  this->Count = argRight.Count;
第49行                  this->pData = argRight.pData;
第50行                  argRight.pData = nullptr;
第51行                  argRight.Count = 0;
第52行                  return *this;
第53行          }
第54行          unsigned int getCount()const{ return this->Count; }
第55行          T& operator[](unsigned int idx){ return *(this->pData + idx); }
第56行  private:
第57行          unsigned int Count;
第58行          T * pData;
第59行  };
第60行  template<typename T>
第61行  Array<T> Test(){
第62行          Array<T> tmp(100);
第63行          return tmp;
第64行  }
第65行  int main(){
第66行          Array<int> myArr(10);
第67行          Array<int> hisArr = myArr;
第68行          Array<int> herArr = Array<int>(15);
第69行          Array<int> itsArr;
第70行          itsArr = Array<int>(20);
第71行          cout << "Test()函数开始执行！！！\n";
第72行          itsArr = Test<int>();
第73行
第74行          cout << "\nmove函数的使用。\n";
第75行          Array<int> yourArr = move(herArr);
第76行
第77行          for (unsigned int i = 0; i < yourArr.getCount(); ++i)cout << yourArr[i] << " "; cout << endl;
第78行          for (unsigned int i = 0; i < herArr.getCount(); ++i)cout << herArr[i] << " "; cout << endl;
第79行          system("pause");
第80行          return 0;
第81行  }
```

3、类的继承

两个类之间的关系可以是继承与被继承关系，假定Z继承自A，则A被称之为Z的基类，Z被称之为A的派生类，如例程20-17所示，Shape是基类，Circle，RectAngle是Shape的派生类，此处为public继承。除public继承外，还有protected和private继承关系。

例程20-17

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  const double PI = 3.1415926;
```

```
class Shape{
public:
        Shape(){ ; }
};
class Circle:public Shape{
public:
        Circle(double r) :R(r){ ; }
        double getArea(){ return PI *R*R; }
        double getPeri(){ return 2*PI *R; }
private:
        double R;
};
int main(){
        Circle circleA(2.0);

        return 0;
}
```

例程20-18是有关纯虚函数和抽象类。

例程20-18

```
#include<iostream>
using namespace std;
const double PI = 3.1415926;
class Shape{
public:
        Shape(){ ; }
        virtual double getArea()=0;
};
class Circle:public Shape{
public:
        Circle(double r) :R(r){ ; }
        double getArea(){ return PI *R*R; }
        double getPeri(){ return 2*PI *R; }
private:
        double R;
};
class RectAngle :public Shape{
public:
        RectAngle(double w,double h) :W(w),H(h){ ; }
        double getArea(){ return W*H; }
        double getPeri(){ return 2*(W+H); }
private:
        double W,H;
};
bool operator<(Shape &leftVal, Shape &rightVal){
```

```
第26行          return leftVal.getArea()<rightVal.getArea();
第27行  }
第28行  int main(){
第29行          Circle circleA(2.0);
第30行
第31行          return 0;
第32行  }
```

4、类模板

模板不仅可以用于函数，也可以用于类，称之为类模板，例程20-15就是一个类的模板，Array之中可以装入各种类型的数据，也包括各种自定义的类。例程20-19是类模板的另外一种形式。

例程20-19

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   template<typename T,unsigned int i>
第4行   class Array{
第5行   public:
第6行           explicit Array() :pData(new T[i]){ ; }
第7行           ~Array(){
第8行                   delete[]pData;
第9行           }
第10行  private:
第11行          T* pData=nullptr;
第12行  };
第13行  int main(){
第14行          Array<int,10> arrData;
第15行          system("pause");
第16行          return 0;
第17行  }
```

5、函数对象

请参见STL部分。

第七节　STL

STL是Standard Template Libary的首字母简写，STL的出现使得C++更加利于工程化。STL可分为容器(containers)、迭代器(iterators)、算法(algorithms)、仿函数(functors)等组成。例程20-20是STL的一个应用示例。

例程20-20

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   using namespace std;
第4行
第5行   void printLT5(int i){
第6行           if (i < 5)cout << i << "\t";
第7行   }
第8行   class LT{
第9行   public:
```

```
第10行          LT(int k) :keyNum(k){ ; }
第11行          void operator()(int i){
第12行                  if (i < keyNum)cout << i << "\t";
第13行          }
第14行  private:
第15行          int keyNum;
第16行  };
第17行  template<typename Iter,typename Todo>
第18行  void forEach(Iter beg, Iter end, Todo DO){
第19行          while (beg!=end)DO(*beg++);
第20行  };
第21行  int main(){
第22行          vector<int> myVec;
第23行          for (int i = 0; i < 100; ++i)myVec.push_back(i);
第24行
第25行          forEach(myVec.begin(), myVec.end(), [](int i){if(i<5)cout << i <<"\t"; });
第26行          cout << endl;
第27行
第28行          forEach(myVec.begin(), myVec.end(), printLT5);
第29行          cout << endl;
第30行
第31行          forEach(myVec.begin(), myVec.end(), LT(25));
第32行          cout << endl;
第33行
第34行          LT lessThan(25);
第35行          forEach(myVec.begin(), myVec.end(), lessThan);
第36行          cout << endl;
第37行
第38行          int keyNum = 10;
第39行          forEach(myVec.begin(), myVec.end(), [keyNum](int i){if (i < keyNum)cout << i << "\t"; });
第40行          cout << endl;
第41行
第42行          return 0;
第43行  }
```

在例程中，vector属于STL容器，可以存储管理各种类型的一系列数据；forEach与STL算法中的for_each()函数功能几乎相同，其功能是遍历容器中的数据；myVec.begin()和myVec.end()是vector<int>类型的迭代器；遍历时具体做什么有for_each()第三个参数确定，其实现方法可以是lambda函数、常规函数和函数对象。从例程可以看出，lambda函数和函数对象更加具有灵活性，尤其是函数对象。

1、函数对象

函数对象，形式及其使用方法与函数相同，但本质是类的实例化(也就是对象)，在该对象中，重载括号运算符即operator()。例程中，执行LT(25)时，生成LT类的匿名对象，并将25传递给keyNum。生成的匿名对象作为实参传递给forEach()函数之DO形参。

在forEach()函数中，DO(*beg++)中的DO是形参，在例程第31行其实参为LT(25)生成的匿名对象，因此DO(*beg++)相当于LT(*beg++)【注：此处LT为LT(25)生成的匿名对象】。第34、35行也使用到函数对象，但是不是匿名对象，而是具名对象，该对象名称为lessThan，很明显没有LT(25)形象，不过也证明LT(25)确实生成了对象，不过是一个匿名对象。

函数对象比函数更加具有普适性，因为函数对象可以定义跨越调用的可持久的部分，同时又能够从对象的外面进行初始化和检查。

例程20-21是一个简单的冒泡法排序，其中bubbleSort()有两个版本，双参数和三参数，双参数默认采用降序排列，三参数版本可以

指定降序还是升序，甚至可以指定其他排序方法。由此可以看出算法、迭代器和函数对象配合，确实功能强大。

例程20-21

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<cmath>
第5行  #include<ctime>
第6行  using namespace std;
第7行  template<typename Iter>
第8行  void bubbleSort(Iter beg,Iter end){
第9行                  unsigned int count = 0;
第10行                 Iter FIRST = beg;
第11行                 for (; (beg + 1) != end;++beg){
第12行                 for (Iter first=FIRST; (first + 1 + count) != end; ++first){
第13行                         if (*first < *(first + 1)){//从大到小的排列
第14行                                 auto tmp = *first;
第15行                                 *first = *(first + 1);
第16行                                 *(first + 1) = tmp;
第17行                         }
第18行                 }
第19行                 count++;
第20行         }
第21行 }
第22行 template<typename Iter,typename CMP>
第23行 void bubbleSort(Iter beg, Iter end,CMP cmp){
第24行         unsigned int count = 0;
第25行         Iter FIRST = beg;
第26行         for (; (beg + 1) != end; ++beg){
第27行                 for (Iter first = FIRST; (first + 1 + count) != end; ++first){
第28行                         if(cmp (*first,*(first + 1))){
第29行                                 auto tmp = *first;
第30行                                 *first = *(first + 1);
第31行                                 *(first + 1) = tmp;
第32行                         }
第33行                 }
第34行                 count++;
第35行         }
第36行 }
第37行
第38行 template<typename T>
第39行 struct Less{
第40行         bool operator()(T a, T b){
第41行                 return a < b;
第42行         }
```

```
};
template<typename T>
struct Greater{
        bool operator()(T a, T b){
                return a> b;
        }
};

bool MOD3(int a, int b){
        return (a % 3) > (b % 3);
}
class MOD{
public:
        MOD(int k) :keyNum(k){ ; }
        bool operator()(int a,int b){
                return (a%keyNum) > (b%keyNum);
        }
private:
        unsigned int keyNum;
};
int main(){
        srand((int)time(NULL));

        vector<int> myVec;
        for (int i = 0; i < 50; ++i)myVec.push_back(rand() % 1000);
        cout << "输出原始序列" << endl;
        for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });

        bubbleSort(myVec.begin(), myVec.end());
        cout << "输出排序后序列" << endl;
        for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });

        bubbleSort(myVec.begin(),myVec.end(),Greater<int>());
        cout << "输出排序后序列" << endl;
        for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });

        bubbleSort(myVec.begin(), myVec.end(), Greater<int>());
        cout << "输出排序后序列" << endl;
        for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });

        bubbleSort(myVec.begin(), myVec.end(), MOD3);
        cout << "输出排序后序列" << endl;
        for_each(myVec.begin(), myVec.end(), [](int i){cout << i %3<< "\t"; });

```

```
第87行          bubbleSort(myVec.begin(), myVec.end(), [](int a, int b){return (a % 2) > (b % 2); });
第88行          cout << "输出排序后序列" << endl;
第89行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i % 2 << "\t"; });
第90行
第91行          bubbleSort(myVec.begin(), myVec.end(),MOD(5));
第92行          cout << "输出排序后序列" << endl;
第93行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i % 5 << "\t"; });
第94行
第95行          return 0;
第96行  }
```

STL定义了一些列类似Less、Greater函数对象，称之为谓词，如less<T>、less_equal<T>、equal<T>、not_equal<T>、greater_equal<T>、greater<T>。有两个参数的称之为二元谓词，有一个参数的称之为一元谓词。

如果在例程20-21环境下要列出小于100的数值，该如何完成呢？在该例程中，已经定义了Less<T>类，其括号运算符需要两个参数，而例程20-20中的forEach()的cmp形参仅接受一个参数，是单独另写一个不同命的Less类吗？这不是工程的解决方法，没有实现代码复用，况且还有Greater等存在，例程20-22是更好的解决方案。

例程20-22

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<algorithm>
第4行   #include<cmath>
第5行   #include<ctime>
第6行   using namespace std;
第7行
第8行   template<typename Iter,typename Pred>
第9行   unsigned int countIF(Iter beg,Iter end,Pred pred){
第10行          unsigned int count=0;
第11行          for (; beg != end;++beg)if (pred(*beg))count++;
第12行          return count;
第13行  }
第14行  template<typename T>
第15行  struct Less{
第16行          bool operator()(T a, T b){
第17行                  return a < b;
第18行          }
第19行  };
第20行  template<typename T>
第21行  struct Greater{
第22行          bool operator()(T a, T b){
第23行                  return a > b;
第24行          }
第25行  };
第26行  template<typename T>
第27行  class UnaryLessA{
第28行  public:
```

```
第29行          UnaryLessA(T v) :Val(v){ ; }
第30行          bool operator()(T i){
第31行                  return Less<T>()(i, Val);//此处Less<T>()生成Less<T>的对象；
第32行          }
第33行  private:
第34行          T Val;
第35行  };
第36行
第37行  template<typename Compare,typename T>
第38行  class binder{
第39行  public:
第40行          binder(Compare cmp,T v) :comp(cmp),Val(v){ ; }
第41行          bool operator()(T i){
第42行                  return comp(i, Val);
第43行          }
第44行  private:
第45行          Compare comp;
第46行          T Val;
第47行  };
第48行  template<typename Compare,typename T>
第49行  binder<Compare,T> Bind(Compare cmp, T a){
第50行          return binder<Compare, T>(cmp, a);
第51行  }
第52行  int main(){
第53行          srand((int)time(NULL));
第54行
第55行          vector<int> myVec;
第56行          for (int i = 0; i < 50; ++i)myVec.push_back(rand() % 1000);
第57行          cout << "输出原始序列\n" << endl;
第58行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });
第59行
第60行          sort(myVec.begin(), myVec.end());
第61行          cout << "输出排序后序列\n" << endl;
第62行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });
第63行
第64行          cout << "输出小于100的个数：";
第65行          cout << countIF(myVec.begin(), myVec.end(), UnaryLessA<int>(100)); cout << endl;
第66行          cout << "输出小于100的个数：";
第67行          cout << countIF(myVec.begin(), myVec.end(), binder<Less<int>, int>(Less<int>(), 100)); cout << endl;
第68行          cout << "输出大于100的个数：";
第69行          cout << countIF(myVec.begin(), myVec.end(), binder<Greater<int>, int>(Greater<int>(), 100));cout<<en
dl;
第70行          cout << "输出小于100的个数：";
第71行          cout << countIF(myVec.begin(), myVec.end(), Bind(Less<int>(), 100)); cout << endl;
第72行          cout << "输出大于100的个数：";
```

```
第73行          cout << countIF(myVec.begin(), myVec.end(), Bind(Greater<int>(), 100)); cout << endl;
第74行
第75行          return 0;
第76行  }
```

第65行代码实现了一个参数，其原因在于第26-35行，尤其是第31行，UnaryLessA的括号运算符重载仅有一个参数，但通过Less<T>()形成Less的对象，并执行其括号运算符，实现了一个参数向两个参数的转换，一个参数来自UnaryLessA的构造函数，另外一个参数来自向其括号运算符传递的参数。

UnaryLessA仅能针对Less，但经过简单改造后能适应更多，如Greater等。Less<T>()是一个对象，则可以通过参数进行设置，如第37-47行的binder类，可以通过构造函数设置binder类的comp值，第42行则调用其对应的括号运算符。采用binder后，其调用形式如第67、69行所示，虽然能完成指定功能，但看起来太复杂，不友好。利用函数模板的类型推断，可以大大简化调用形式，如第71、73所示，相关实现代码如第48-51行所示。

对于形如binder<Less<int>, int>(Less<int>(), 100))的代码还可以优化，在Less<int>中已有类型int，其后的int应想办法省略，减少代码量形式更友好，同时也能确保类型一直，防止类型不同。如何才能获取到Less<int>的类型呢？可以考虑如例程20-23所示的代码。由于binder中的myDefType依赖于Compare类型，因此必须在类型前增加typename。这样改造，将使得代码更加简洁，并且能保持类型一致，但仅仅Less不够，对Greater也得使用。为了保持一致性，可以考虑让Less和Greater继承自同一基类，如例程20-24所示。经过如此改造后，代码更加简洁。

例程20-23

```
第1行   template<typename T>
第2行   struct Less{
第3行           typedef T myDefType;//类型别名
第4行           bool operator()(T a, T b){
第5行                   return a < b;
第6行           }
第7行   };
第8行
第9行   template<typename Compare>
第10行  class binder{
第11行  public:
第12行          binder(Compare cmp, typename Compare::myDefType v) :comp(cmp), Val(v){ ; }
第13行          bool operator()(typename Compare::myDefType i){
第14行                  return comp(i, Val);
第15行          }
第16行  private:
第17行          Compare comp;
第18行          typename Compare::myDefType Val;
第19行  };
```

例程20-24

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<algorithm>
第4行   #include<cmath>
第5行   #include<ctime>
第6行   using namespace std;
第7行
第8行   template<typename Iter,typename Pred>
```

```
unsigned int countIF(Iter beg,Iter end,Pred pred){
        unsigned int count=0;
        for (; beg != end;++beg)if (pred(*beg))count++;
        return count;
}
template<typename firstType,typename secondType,typename rtnType>
class Binary{
public:
        typedef firstType argOneType_Bin;
        typedef secondType argTwoType_Bin;
        typedef rtnType argRtnType_Bin;
};
template<typename T>
struct Less:public Binary<T,T,bool>{
        bool operator()(T a, T b){
                return a < b;
        }
};
template<typename T>
struct Greater :public Binary<T, T, bool>{
        bool operator()(T a, T b){
                return a > b;
        }
};

template<typename Compare>
class binder{
public:
        binder(Compare cmp, typename Compare::argTwoType_Bin v) :comp(cmp), Val(v){ ; }
        bool operator()(typename Compare::argOneType_Bin i){
                return comp(i, Val);
        }
private:
        Compare comp;
        typename Compare::argTwoType_Bin Val;
};

template<typename Compare>
binder<Compare> bind2nd(Compare cmp, typename Compare::argTwoType_Bin a){
        return binder<Compare>(cmp, a);
}
int main(){
        srand((int)time(NULL));

        vector<int> myVec;
```

```
第53行
第54行          for (int i = 0; i < 50; ++i)myVec.push_back(rand() % 1000);
第55行          cout << "输出原始序列\n" << endl;
第56行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });
第57行
第58行          sort(myVec.begin(), myVec.end());
第59行          cout << "输出排序后序列\n" << endl;
第60行          for_each(myVec.begin(), myVec.end(), [](int i){cout << i << "\t"; });
第61行
第62行          cout << "输出小于100的个数：";
第63行          cout << countIF(myVec.begin(), myVec.end(), binder<Less<int>>(Less<int>(), 100)); cout << endl;
第64行          cout << "输出大于100的个数：";
第65行          cout << countIF(myVec.begin(), myVec.end(), binder<Greater<int>>(Greater<int>(), 100)); cout << end
    l;
第66行          cout << "输出小于100的个数：";
第67行          cout << countIF(myVec.begin(), myVec.end(), bind2nd(Less<int>(), 100)); cout << endl;
第68行          cout << "输出大于100的个数：";
第69行          cout << countIF(myVec.begin(), myVec.end(), bind2nd(Greater<int>(), 100)); cout << endl;
第70行
第71行          return 0;
第72行  }
```

C++的STL提供了bind1st、bind2nd、not2等函数，其工作原理与之类似。bind1st绑定第一个参数，bind2nd绑定第二个参数，not2是对二元函数对象求反。这些函数的返回值均为一元函数对象。对于一元函数对象而言，同样有参数类型标准化问题，可以参照二元函数对象的方式解决。not1是对一元函数对象求反。

常规函数也能转换为函数对象，使之能与STL提供的函数联用，其ptr_fun及提供将常规函数转换为函数对象，其原理与之类似，如例程20-25所示。

例程20-25

```
第1行   //包装一元函数指针，使之成为具有仿函数一样能力
第2行   //定义一个指向一元函数的类
第3行   //Result:返回值类型，Arg是参数
第4行   template<typename Arg, typename Result>
第5行   class Pointer_to_unary_function:
第6行           public unary_function<Arg, Resultunary_function<Arg, Result>{
第7行   protected:
第8行           Result(*ptr)(Arg);
第9行   public:
第10行          Pointer_to_unary_function(){}//空构造函数
第11行          explicit Pointer_to_unary_function(Result(*x)(Arg)) :ptr(x){}//构造函数
第12行          Result operator()(Arg x)const{ return ptr(x); }
第13行  };
第14行  //为规避干扰，将ptr_fun改为ptrFun
第15行  template<typename Arg, class Result>
第16行  inline Pointer_to_unary_function<Arg, Result> ptrFun(Result(*x)(Arg)){
第17行          return Pointer_to_unary_function<Arg, Result>(x);
```

```
第18行  };
第19行  //包装二元函数指针，使之成为具有仿函数一样能力
第20行  //定义一个指二元函数的类
第21行  //Result:返回值类型，Arg1、Arg2是参数
第22行  template<typename Arg1, typename Arg2, typename Result>
第23行  class Pointer_to_binary_function : public binary_function<Arg1, Arg2, Result>{
第24行  protected:
第25行          Result(*ptr)(Arg1, Arg2);
第26行  public:
第27行          Pointer_to_binary_function(){}//空构造函数
第28行          explicit Pointer_to_binary_function(Result(*x)(Arg1, Arg2)) :ptr(x){}//构造函数
第29行          Result operator()(Arg1 x, Arg2 y)const{ return ptr(x, y); }
第30行  };
第31行  //为规避干扰，将ptr_fun改为ptrFun
第32行  template<typename Arg1, typename Arg2, class Result>
第33行  inline Pointer_to_binary_function<Arg1, Arg2, Result> ptrFun(Result(*x)(Arg1, Arg2)){
第34行          return Pointer_to_binary_function<Arg1, Arg2, Result>(x);
第35行  };
```

第八节　智能指针

例程20-26

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   template<typename T>
第4行   class sharePtr{
第5行   public:
第6行           struct Box{
第7行                   unsigned int count = 0;
第8行                   T* pData = nullptr;
第9行                   void LocateMem(unsigned int N){
第10行                          cout << "Here!" << endl;
第11行                          this->pData = new T[N];
第12行                          this->count++;
第13行                  }
第14行          };
第15行  public:
第16行          sharePtr(){};
第17行          explicit sharePtr(unsigned int N) {
第18行                  this->pAddr = new Box;
第19行                  this->pAddr->LocateMem(N);
第20行          }
第21行          sharePtr(const sharePtr<T> & me){
第22行                  this->pAddr = me.pAddr;
第23行                  this->pAddr->count++;
第24行          }
```

```
第25行          ~sharePtr(){
第26行                  unsigned int usedCount = this->use_count();
第27行                  if (usedCount == 1){
第28行                          delete[]this->pAddr->pData;
第29行                          delete this->pAddr;
第30行                          cout << "~sharePtr() delete" << endl;
第31行                  }
第32行                  else{
第33行                          this->pAddr->count--;
第34行                          cout << "~sharePtr() --" << endl;
第35行                  }
第36行          }
第37行          sharePtr<T>& operator=(const sharePtr<T> &me){
第38行                  unsigned int usedCount = this->use_count();
第39行                  switch (usedCount){
第40行                  case 0://没有分配
第41行                          this->pAddr = me.pAddr;
第42行                          this->pAddr->count++;
第43行                          break;
第44行                  case 1://仅分配一次
第45行                          delete[]this->pAddr->pData;
第46行                          delete this->pAddr;
第47行                          this->pAddr = me.pAddr;
第48行                          this->pAddr->count++;
第49行                          break;
第50行                  default://超过1次
第51行                          this->pAddr->count--;
第52行                          this->pAddr = me.pAddr;
第53行                          this->pAddr->count++;
第54行                          break;
第55行                  }
第56行                  return *this;
第57行          }
第58行          T* get()const{
第59行                  return this->pAddr->pData;
第60行          }
第61行          unsigned int use_count(){
第62行                  return this->pAddr->count;
第63行          }
第64行  private:
第65行          Box * pAddr=nullptr;
第66行  };
第67行  template<typename T>
第68行  void Test(const sharePtr<T> & shareP){
```

```
第69行          sharePtr<T> tmpSharePtr = shareP;
第70行          cout << tmpSharePtr.use_count() << endl;
第71行  }
第72行  int main(){
第73行          sharePtr<int> mySharePtr(10), tmpSharePtr(20);
第74行          for (int i = 0; i < 10; ++i)*(mySharePtr.get() + i) = 10 + i;
第75行          for (int i = 0; i < 20; ++i)*(tmpSharePtr.get() + i) = 20 + i;
第76行          sharePtr<int> hisSharePtr = mySharePtr;
第77行          cout << mySharePtr.use_count() << endl;
第78行          cout << hisSharePtr.use_count() << endl;
第79行          Test(hisSharePtr);
第80行          cout << hisSharePtr.use_count() << endl;
第81行          for (int i = 0; i < 10; ++i)cout << *(mySharePtr.get() + i) << " "; cout << endl;
第82行          for (int i = 0; i < 10; ++i)cout << *(hisSharePtr.get() + i) << " "; cout << endl;
第83行          for (int i = 0; i < 20; ++i)cout << *(tmpSharePtr.get() + i) << " "; cout << endl;
第84行
第85行          hisSharePtr = tmpSharePtr;
第86行          for (int i = 0; i < 10; ++i)cout << *(mySharePtr.get() + i) << " "; cout << endl;
第87行          for (int i = 0; i < 20; ++i)cout << *(hisSharePtr.get() + i) << " "; cout << endl;
第88行          for (int i = 0; i < 20; ++i)cout << *(tmpSharePtr.get() + i) << " "; cout << endl;
第89行
第90行          return 0;
第91行  }
```

第九节　异常

第十节　文件操作