```
    float salary;
};

Person pr1 = {"Frank Voltaire",12345678,3.35};

int main()
{
    Person pr2;
    pr2 = pr1;          //结构变量的赋值
    cout <<pr2.name <<"     "
         <<pr2.id <<"     "
         <<pr2.salary <<endl;
}
```

运行结果为：

```
Frank Voltaire     12345678     3.35
```

程序中定义了一个全局 Person 结构变量 pr1，它使用与初始化数组相似的方法进行初始化。在 main()函数中，定义了一个结构变量 pr2，然后使用赋值运算符将 pr1 的内容赋值给 pr2。

在结构 Person 中，成员 name 是一个字符数组，通过结构变量的赋值，该数组作为成员也被赋值了。

两个不同结构名的变量是不允许相互赋值的，即使两者包含有同样的成员。

例如，下面的代码，不能将结构 Employee 的变量赋给结构 Person 的变量：

```
struct Person
{
    char name[20];
    unsigned long id;
    float salary;
};

struct Employee     //Employee 与 Person 具有相同的成员
{
    char name[20];
    unsigned long id;
    float salary;
};

int main()
{
    Person pr1 = {"Frank Voltaire",12345678,3.35};
    Employee er1;
    er1 = pr1;      //error 类型不匹配
    //...
}
```

→ 在 C 中(而不是 C++)，结构变量定义在结构类型名前必须有 struct 关键字。

例如，定义结构变量 pr1：

```
struct Person pr1 = {"Frank Voltaire",12345678,3.35};
```

否则 C 编译器报错。

10.2 结构与指针

根据结构类型可以定义一个变量，**是变量就有地址**。结构不像数组，**结构变量不是指针**。通过取地址“&”操作，可以得到结构变量的地址，这个地址就是结构的第一个成员地址。

可以将结构变量的地址赋给结构指针，结构指针通过箭头操作符“->”（也是一种结构成员操作符）来访问结构成员。

例如，下面的代码定义了结构指针，通过结构指针来访问结构成员：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

int main()
{
   Person pr1;
   Person * prPtr;
   prPtr = &pr1;
   strcpy(prPtr->name,"David Marat");
   prPtr->id = 987654321;
   prPtr->salary = 335.0;
   cout <<prPtr->name <<"     "
        <<prPtr->id <<"     "
        <<prPtr->salary <<endl;
}
```

运行结果为：

```
Davit Marat  987654321  335
```

使用箭头操作符就是对结构成员进行操作。但必须清楚，**当用点操作符时，它的左边应是一个结构变量，当用箭头操作符时，它的左边应是一个结构指针**。

例中把 pr1 的地址赋给 Person 结构指针 prPtr，然后通过这个指向 pr1 的指针对 pr1 进行赋初值和输出。这一步是重要的，如果不把 pr1 的地址赋给 prPtr，那么，prPtr 是个随机地址，在这个地址上赋值是危险的。

指针是有类型的，引用一个整型指针得到一个整数，引用一个结构指针得到一个结构。即 * prPtr 的值就是结构 Person 的变量 pr1 的值，而不会是其他类型的值。

箭头操作符与点操作符是可以互换的，程序 ch10_3.cpp 中：

prPtr->>name　等价于　pr1.name　等价于　(*prPtr).name

也就是说，可以使用点操作符而不使用箭头操作符。

例如，下面的程序将程序 ch10_3.cpp 中结构的箭头操作符改为结构指针：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_4.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

int main()
{
   Person pr1;
   Person * prPtr;
   prPtr = &pr1;
   strcpy((*prPtr).name,"David Marat");
   (*prPtr).id = 987654321;
   (*prPtr).salary = 335.0;
   cout <<(*prPtr).name <<"     "
        <<(*prPtr).id <<"     "
        <<(*prPtr).salary <<endl;
}
```

运行结果为：

```
Davit Marat  987654321  335
```

程序的运行结构与 ch10_3.cpp 一样，但这个句法没有真正的好处，通常在这种情况时，用箭头操作符更直观一些。

10.3　结构与数组

结构是一个数据类型，所以也可以拥有结构数组。要定义结构数组，必须先声明一个结构，然后定义这个结构类型的数组。

例如，定义一个 100 个元素组成的 Person 结构类型数组：

```
struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};
```

```
Person allone[100];     //定义一个 Person 类型的数组
```

结构数组中，每个元素都是结构变量，访问结构数组元素中的成员，方法与前类似。例如，下面的程序中，对一个 Person 结构数组进行“冒泡法”排序，工资高的排在后面：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *            ch10_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

Person allone[6] = {{"jone", 12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
                    {"jasen", 42876, 623.0},
                    {"peter", 23987, 400.0},
                    {"yoke", 12335, 511.0}};

int main()
{
   Person temp;
   for(int i = 1; i<6; i ++ )        //排序
   {
    for(int j = 0; j< = 5 - i; j ++ )        //一轮比较
    {
       if(allone[j].salary > allone[j + 1].salary)     //比较工资成员
       {
         temp = allone[j];     //结构变量的交换
         allone[j] = allone[j + 1];
         allone[j + 1] = temp;
       }
    }
   }

   for(int k = 0; k<6; k ++ )     //输出
   {
    cout <<allone[k].name<<"      "
         <<allone[k].id<<"      "
         <<allone[k].salary<<endl;
   }
}
```

运行结果为：

```
marit  27519  311
jone  12345  339
peter  23987  400
david  13916  449
```

```
yoke  12335  511
jasen  42876  623
```

程序定义了一个 Person 结构的全局数组，一共 6 个元素，每个元素都是一个 Person 结构变量。在冒泡排序中，比较成员 salary 的大小，访问相应结构成员的方法是数组元素名、点操作符加上成员名：allone[i].salary。

排序中交换数组元素，就是交换结构变量，程序中以结构变量赋值的方法来进行交换。

结构变量相互交换，在结构很大(成员很多)时，并不是一种好办法。可以建立一个结构指针数组来实现同样的功能。即，一个数组中，每个元素都是一个结构指针。

例如，下面的程序建立了一个结构指针数组，实现 ch10_5.cpp 的结构排序：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch10_6.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

Person allone[6] = {{"jone",  12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
                    {"jasen", 42876, 623.0},
                    {"peter", 23987, 400.0},
                    {"yoke", 12335, 511.0}};

int main()
{
   Person * pA[6] = {&allone[0], &allone[1], &allone[2],
                     &allone[3], &allone[4], &allone[5]};
   Person * temp;
   for(int i = 1; i<6; i ++ )
   {
    for(int j = 0; j<= 5 - i; j ++ )
    {
       if(pA[j] - >salary > pA[j + 1] - >salary)     //比较工资成员
       {
         temp = pA[j];
         pA[j] = pA[j + 1];
         pA[j + 1] = temp;
       }
    }
   }

   for(int k = 0; k<6; k ++ )
   {
    cout <<pA[k] - >name<<"     "
```

```
            <<pA[k]->>id<<"    "
            <<pA[k]->>salary<<endl;
    }
}
```

运行结果为：

```
marit  27519  311
jone  12345  339
peter  23987  400
david  13916  449
yoke  12335  511
jasen  42876  623
```

从程序中看到，建立了一个结构指针数组 pA 并依次初始化为结构数组元素的地址值。对结构成员的访问，由点操作符改成了箭头操作符，因为现在是对结构指针进行操作。

发生交换时，并不是两个结构变量值交换，而是两个结构指针交换，所以交换的临时变量是一个结构指针。最后输出的是通过结构指针访问的结构变量值。运行结果与程序 ch10_5.cpp 相同。

由于不必用结构值赋值的方法来交换，运行效率较 ch10_5.cpp 程序为高。

10.4 传递结构参数

1. 传递结构值

结构可以按值传递，这种情况下整个结构值都将被复制到形参中去。

例如，下面的程序说明怎样将结构值作为参数传给函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_7.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

void Print(Person pr)
{
   cout <<pr.name <<"    "
        <<pr.id <<"    "
        <<pr.salary <<endl;
}

Person allone[4] = {{"jone",     12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
```

```
                    {"yoke", 12335, 511.0}};

int main()
{
   for(int i = 0; i<4; i ++ )
   {
    Print(allone[i]);
   }
}
```

运行结果为：

```
jone  12345  339
david  13916  449
marit  27519  311
yoke  12335  511
```

Print()函数的参数是 Person 结构变量，main()函数调用了 4 次 Print()函数，实参为 Person 结构数组的元素。

2. 传递结构的引用

结构也可以引用传递，这种情况下仅仅把结构地址传递给形参。引用传递效率较高，因为它不用传递整个结构变量的值(有时候是很大的空间)，节省了传递的时间和存储空间，同时又不影响其功能。

例如，下面的程序是修改了程序 ch10_7.cpp，以引用传递结构：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_8.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

void Print(Person& pr)
{
   cout <<pr.name <<"     "
        <<pr.id <<"     "
        <<pr.salary <<endl;
}

Person allone[4] = {{"jone",     12345, 339.0},
                    {"david", 13916, 449.0},
                    {"marit", 27519, 311.0},
                    {"yoke", 12335, 511.0}};
```

```
int main()
{
   for(int i = 0; i<4; i ++ )
   {
    Print(allone[i]);
   }
}
```

运行结果为：

```
jone   12345   339
david  13916   449
marit  27519   311
yoke   12335   511
```

程序中引用传递与值传递的差别，只在 Print()函数声明的参数中结构类型名后加上一个“&”引用声明符，而函数的实现与函数调用代码都与值传递相同，所以引用传递在程序的理解上与值传递一样容易。虽然结构值还可以通过结构指针传递，但程序的可读性比引用传递要差一些(见 9.4 节)。

由于结构的大小是随用户定义而定的，所以有时候会很大。当结构很大时，引用传递的优越性才真正开始体现。引用传递的进一步介绍在 18.3 节。在编程经验上，除非是小结构，一般很少按值传递。

10.5 返回结构

1. 返回结构

一个函数可以返回一个结构值，即若干数据类型值的一个聚集。这使得我们在 10.1 节中讨论的职工数据返回问题能得以解决。

例如，下面的程序通过函数返回一个结构值给结构变量赋值：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_9.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

Person GetPerson()
{
   Person temp;
   cout <<"Please enter a name for one person:\n";
   cin>>temp.name;
```

```
    cout <<"Please enter one's id number and his salary:\n";
    cin >>temp.id >>temp.salary;

    return temp;
}

void Print(Person& p)
{
    cout <<p.name <<"    "
         <<p.id <<"    "
         <<p.salary <<endl;
}

int main()
{
    Person employee[3];
    for(int i = 0; i<3; i++)
    {
     employee[i] = GetPerson();
     Print(employee[i]);
    }
}
```

运行结果为：

```
Please enter a name for one person:
marit
Please enter one's id number and his salary:
27519  311.0
marit  27519  311
Please enter a name for one person:
jone
Please enter one's id number and his salary:
12345  339.0
jone  12345  339
Please enter a name for one person:
peter
Please enter one's id number and his salary:
23987  400.0
peter  23987  400
```

主函数中调用了 GetPerson()函数，该函数返回 Person 结构值，该结构值被赋给了主函数中的结构数组元素。

由于结构返回时，要复制结构值给一个临时结构变量(见 9.6 节)，当结构很大时，运行效率会受影响。可以用一种效率更高的结构参数引用传递的方法来代替。结构参数引用传递时无须复制结构值，连赋值操作都不需要了。

例如，下面的程序用结构参数引用传递的方法实现程序 ch10_9.cpp 的同样功能：

```
//* * * * * * * * * * * * * * * * * * * * * *
//* *          ch10_10.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * *
#include <iostream.h>
```

```
struct Person
{
   char name[20];
   unsigned long id;
   float salary;
};

void GetPerson(Person& p)     //结构参数引用传递的函数
{
   cout <<"Please enter a name for one person:\n";
   cin>>p.name;

   cout <<"Please enter one's id number and his salary:\n";
   cin >>p.id >>p.salary;
}

void Print(Person& p)
{
   cout <<p.name <<"     "
        <<p.id <<"     "
        <<p.salary <<endl;
}

int main()
{
   Person employee[3];
   for(int i = 0; i<3; i++)
   {
    GetPerson(employee[i]);     //调用后,employee[i]被赋值
    Print(employee[i]);
   }
}
```

运行结果为：

```
Please enter a name for one person:
marit
Please enter one's id number and his salary:
27519  311.0
marit  27519  311
Please enter a name for one person:
jone
Please enter one's id number and his salary:
12345  339.0
jone  12345  339
Please enter a name for one person:
peter
Please enter one's id number and his salary:
23987  400.0
peter  23987 400
```

用结构参数引用传递，无须返回结构值，无须给另一个结构变量赋值，也无须在函数返回时复制结构值给临时结构变量，节省了系统开销。在主函数中，调用该函数的格式应作相

应的调整。

2. 返回结构的引用

一个函数可以返回一个结构的引用和结构的指针(见 9.5 节),但是,不要返回一个局部结构变量的引用或指针。

例如,下面的代码不应将局部结构变量的引用返回给上层函数:

```
Person& GetPerson()
{
   Person p;
   cout <<"Please enter a name for one person:\n";
   cin.get(p.name);

   cout <<"Please enter one's id number and his salary:\n";
   cin >>p.id >>p.salary;
   return p;      //局部结构变量的地址
}

void main()
{
   Person& sp = GetPerson();
   //...
}
```

在主函数中,引用 sp 用 GetPerson()来初始化,使得 sp 与 GetPerson()中的局部变量名同享一个空间(见 9.6 节),这是不好的程序设计。

10.6 链表结构

1. 结构的嵌套

结构可以嵌套,即结构中可以包含结构成员。例如:

```
struct Education
{
   char major[20];
   char degree[20];
   double gpa;
};

struct Student
{
   Education school;      //结构中嵌套一个 Education 结构
   char id[20];
   int graduate;
};

Student ss;     //创建一个结构变量
```

在引用嵌套结构的成员时,要使用多个点操作符,例如,ss.school.major,ss.school.degree 等。

结构成员不能是自身的结构变量，但可以用自身结构指针作为成员。

例如，下面的结构含有一个自身结构的指针，但不应有自身结构的变量：

```
struct List
{
  char name[20];
  List * pN;      //List 结构指针作为其成员
  List m;         //error 不应含有自身结构变量
};
```

2. 遍历结构变量的问题

我们可以用循环语句遍历结构数组中的所有元素来查找所要的结构变量。

在程序中，经常要用到多个相同结构类型的元素，而元素的个数要到运行时才能确定。同样我们能依赖动态内存分配来定义结构数组。这样，所有的结构变量在内存中依次排列，我们仍能很方便地遍历所有结构变量，查找所要的结构值。

然而，在很多情况下，我们自始至终不知道究竟需要多少个结构记录。例如，商场里客户购物，来一个就建立(动态分配)一个客户记录。临下班时，要汇总(遍历所有记录)营业额。当然我们用现成的数据库系统的软件可以很好地做这项工作，但数据库系统的内部实现正是在动态分配技术基础上实现的。

这时候我们在程序中所面临的结构记录内存布局是随机的，结构与结构之间没有联系，见图 10-1。

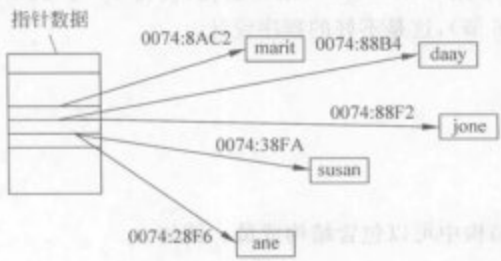


图 10-1　结构变量在内存中的布局

3. 链表结构

通过含有一个自身结构的指针，我们可以实现随机分布的结构变量的遍历。

对于下面的结构：

```
struct List
{
  char name[20];
  List * pN;
};
```

name 成员含有结构中的实际信息，pN 成员是指向另一个 List 的指针。这种结点(结构的实例)通过每个 List 的 pN 成员链接起来，能用于构造任意长的结构链，这样的结构链

称为链表。链表中的每个 List 结构变量称为结点。链表中的第一个 List 结点经常由一个指向 List 的指针引导。这个结构指针不是自身结构成员，但它指向第一个结点，而该结点的 pN 成员又指向另一个结点，而另一个结点的 pN 成员又指向一个结点，这样一直指下去，直到最后一个结点。最后一个结点的 pN 成员值为空(NULL)，见图 10-2。

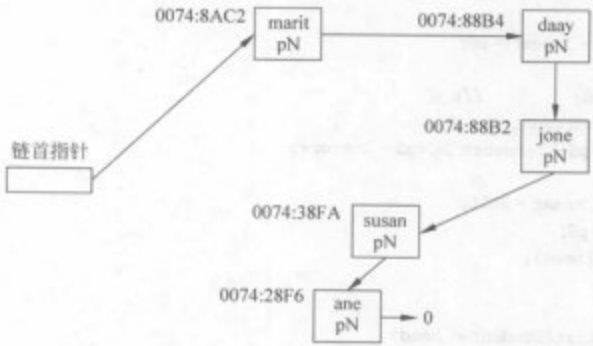


图 10-2　链表结构的描述

链表的组成是一个个链接的结点，每个结点是同类型的结构变量。可以通过程序的办法，来建立和显示链表，可以插入、删除及增加结点来维护一个链表。

一个链表总是包含一个链首指针。操作链表时，一般都先由链首指针引导。

10.7　创建与遍历链表

例如，下面的例子建立一个链表，并输出链表：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_11.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

struct Student
{
   long number;
   float score;
   Student * next;
};

Student * head;     //链首指针

Student * Create()
{
   Student * pS;        //创建的结点指针
   Student * pEnd;      //链尾指针，用于在其后面插入结点
   pS = new Student;    //新建一个结点，准备插入链表
   cin >>pS->number >>pS->score;     //给结点赋值
   head = NULL;         //一开始链表为空
```

```
    pEnd = pS;

    while(pS->number!= 0)
    {
      if(head == NULL)
        head = pS;
      else
        pEnd->next = pS;

      pEnd = pS;          //s 点
      pS = new Student;
      cin >>pS->number >>pS->score;
    }
      pEnd->next = NULL;
      delete pS;
      return(head);
    }

    void ShowList(Student * head)
    {
      cout <<"now the items of list are \n";

      while(head)
      {
        cout <<head->number <<"," <<head->score <<endl;
        head = head->next;
      }
    }
int main()
{
  ShowList(Create());
}
```

运行结果为：

```
54  3.4
23  3.2
24  3.5
15  4.1
66  4.0
0   0.0
now the items of list are
54,3.4
23,3.2
24,3.5
15,4.1
66,4.0
```

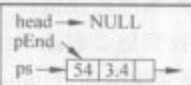
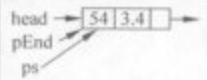
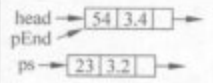
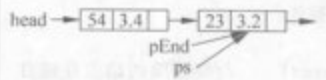
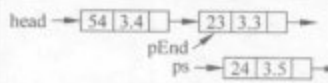
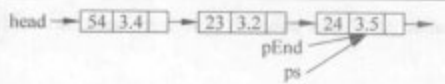
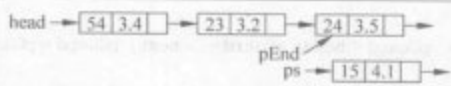
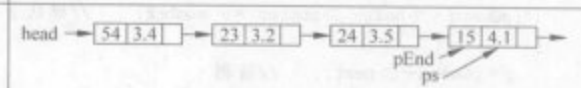
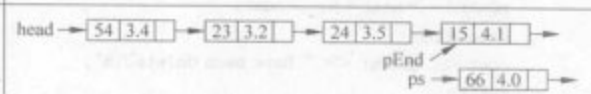
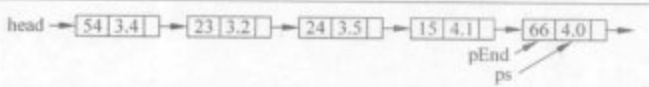
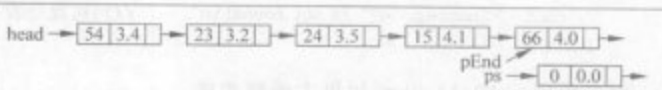
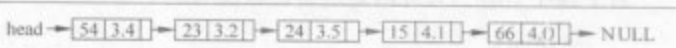
在主函数中，直接调用 ShowList()函数，其参数就是建立链表的 Create()函数的结点指针返回值。因为要保持全局变量的链首指针 head 值不变，所以这里不能用引用参数传递，而只能将链首指针 head 的值复制给 ShowList()函数。

在 Create()函数中，当指向新建结点的指针 pS 接收到第一个结点时，head 头指针指向第一个结点，随后尾结点指针 pEnd 也指向它。如果不是第一个结点，则把它放到尾结点指针 pEnd 所指向的结点的后面，使之成为最后一个结点，因而 pEnd 改为指向该结点。不断

重复直到输入结点的成员 number 值为 0。最后将尾结点指向 NULL，删除结点成员 number 是 0 的结点，返回链首指针 head(也就是说，让 head 代表整个链)。表 10-1 是 Create()函数创建链表的过程。

这里分配一个结点的 new 后面只跟一个数据类型，比之 malloc()函数要简捷。关于 new，后面 14.2 节和 14.3 节还将继续深入介绍。

表 10-1　Create()函数中链表创建的过程

进入循环之前	
第一次进入循环，到达 s 点	
第一次循环结束	
第二次循环到达 s 点	
第二次循环结束	
第三次循环到达 s 点	
第三次循环结束	
第四次循环到达 s 点	
第四次循环结束	
第五次循环到达 s 点	
第五次循环结束	
退出 Create()函数时	

ShowList()函数的目的是要遍历整个链表，输出每个结点的值。形参 head 是结点指针，它接受传递过来的链首指针。修改形参 head 不会影响全局指针变量 head 的值。遍历

是通过 head=head->next 来完成的。之所以不用 head ++ 是因为链表的结点在内存中并不是连续存放的。

10.8 删除链表结点

链表结点的删除操作要保证不破坏链表的链接关系。一个结点删除后，它的前一结点的指针成员应指向它的后一结点，这样就不会因删除而使链接中断。

删除往往还包含查找子过程，因为首先要找到想要删除的结点。它包含找不到的处理。

例如，下面的代码是删除结点的函数，它删除学号为 number 的结点：

```
void Delete(Student * head,long number)
{
  Student * p;
  if(!head)
  {
   cout <<"\nList null!\n";
   return;     //表示未作删除
  }
  if(head->number == number)     //要删除的结点在链首
  {
   p = head;
   head = head->next;
   delete p;
   cout <<number <<" the head of list have been deleted\n";
   return;
  }

  for(Student * pGuard = head; pGuard->next; pGuard = pGuard->next)
  {
     if(pGuard->next->number == number)     //确认下一个结点就是要删除的
     {
        p = pGuard->next;     //待删
        pGuard->next = p->next;
        delete p;
        cout <<number <<" have been deleted\n";
        return;
     }
  }

  cout <<number <<" is not found!\n";     //到达此处表示未找到想删的结点
}
```

对于程序 ch10_11.cpp，如果主函数改成：

```
int main()
{
  head = Create();
  Delete(head,54);
}
```

将得到下面的输出：

```
54   3.4
23   3.2
24   3.5
15   4.1
66   4.0
0    0.0
54 the head of list have been deleted
```

相应的操作见图 10-3。

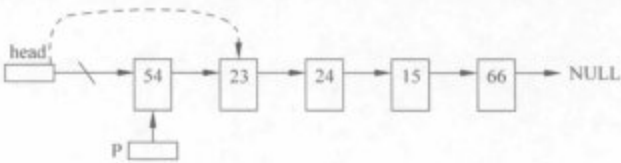


图 10-3　删除链首结点

删除链首结点的步骤为：

(1) P 指向链首结点；

(2) head 指向链首结点的下一个结点；

(3) 删除 p 指向的结点。

这里顺序是重要的。如果 p 不指向链首结点，则当 head 指向链首结点的下一个结点后，原链首结点脱链，导致结点地址丢失，以致无法释放堆空间。如果先删除链首结点，则 head 指针无法指向由链首结点导出的下一个结点地址。

Delete()函数中的循环语句处理非链首结点的删除。如果 ch10_11.cpp 程序中的主函数改成：

```
int main()
{
   head = Create();
   Delete(head,15);
}
```

将得到下面的输出：

```
54   3.4
23   3.2
24   3.5
15   4.1
66   4.0
0    0.0
15 have been deleted
```

相应的操作见图 10-4。

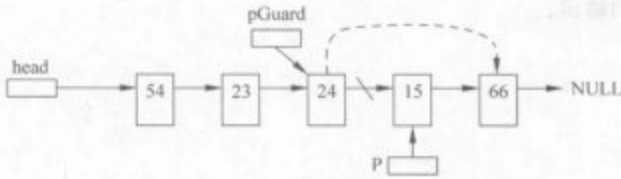


图 10-4　删除非链首结点

如果主函数中对 Delete()的调用改成 Delete(head,11),将会得到下面的输出:

```
54   3.4
23   3.2
24   3.5
15   4.1
66   4.0
0    0.0
11 not found!
```

当 Delete()函数找到要删除的结点时,删除的步骤为:

(1) p 指向待删的结点;

(2) pGuard 所指向结点的 next 成员指向待删结点的下一个结点;

(3) 删除 p 指向的结点。

在找到待删结点时,pGuard 指向待删结点的前一个结点是重要的。否则,待删结点的前一结点地址丢失,其 next 成员无法与待删结点的后一结点链接。在数据结构课程中,通常称 pGuard 指针为"哨兵"。

10.9　插入链表结点

插入操作不能破坏链接关系。应将插入结点的 next 成员指向它的后一个结点,然后将前一结点的 next 成员指向插入的结点,这样就得到了新的链表。

插入操作也包含一个插入位置查找的子过程,同样也面临链表是空表或插入到链首的特殊情况。

假设创建(调用 Create())链表时,结点是按 number 值由小到大顺序排列,则插入结点函数 Insert()设计如下:

```
void Insert(Student * head, Student * stud)
{
   if(head == NULL)     //空表时,将结点置在 head 指针下即可
   {
     head = stud;     //表示链首
     stud -> next = NULL;     //表示链尾
     return;
   }

   if(head -> number > stud -> number)        //结点插入的位置在链首
   {
     stud -> next = head;     //指向链首结点
```

```
        head = stud;     //插入结点成为链首
        return;
      }

      Student * pGuard = head;
      while(pGuard->next && pGuard->next->number<stud->number)
        pGuard = pGuard->next;

      stud->next = pGuard->next;
      pGuard->next = stud;
}
```

对于程序 ch10_11.cpp，如果主函数改成：

```
int main()
{
   Student ps;
   ps.number = 36;
   ps.score = 3.8;
   head = Create();
   Insert(head,&ps);
   ShowList(head);
}
```

将得到下面的输出：

```
15  4.1
23  3.2
24  3.5
54  3.4
66  4.0
0   0.0
now the items of list are
15,4.1
23,3.2
24,3.5
36,3.8
54,3.4
66,4.0
```

相应的操作见图 10-5。

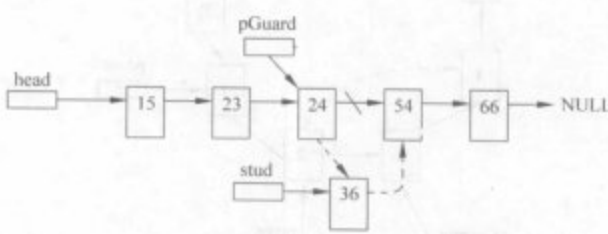


图 10-5　插入 stud 所指向的结点

函数 Insert()中，如果链表为空，则只要将结点挂接在 head 下，然后置插入的结点为链尾即可。即：

```
head = stud;
stud->next = NULL;
```

如果插入的结点值比链首结点值小，即插入位置在链首，则只要将插入结点的 next 成员指向链首，然后 head 指针指向插入结点即可。即：

```
stud->next = head;
head = stud;
```

当要插入在链表中的非链首时，首先要找到插入位置。当发现当前结点(pGuard 所指向的结点)的下一个结点值比插入结点值大时，即找到了插入位置在当前结点之后。具体的插入操作为：

(1) 将插入结点指向当前结点的下一个结点：stud->next = pGuard->next；

(2) 将当前结点指向插入结点：pGuard->next = stud；

这两步操作的顺序是重要的，如果先将当前结点的 next 成员修改，则将失去与后继结点的联系，插入结点无法找到当前结点的下一个结点。

10.10 结构应用：Josephus 问题

小孩围成圈，用环链表来表达是最自然不过的。每个结点代表一个小孩。每个结点是一种结构，内含小孩属性(编号)和结构指针(指向下一个小孩)。

一旦构成了环链，数小孩的事就是在环链中依次经历结点。小孩的离开就是代表该小孩的结点从链中删除。删除后的链表仍是环链表。

不断数小孩到一定个数时，便删除，如此循环，直到只剩最后一个。该结点自己指向自己。此时，该结点代表的小孩就是胜利者。

为了进行链表的删除，需要有“哨兵”保护结点，该结点指向当前被删结点，见图 10-6。

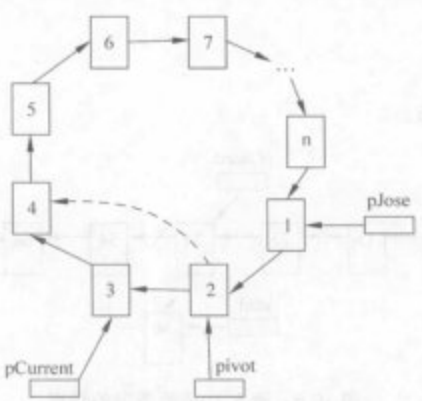


图 10-6 数完第一次小孩数，准备删除结点的情形

由于是环链表，所以没有链首指针，只有当前指针 pCurrent。但作为链表主体的数组，其第一个元素的地址必须要记住，因为从堆中分配的空间，有义务要归还（用 pJose 指向数组首地址）。

删除当前结点（pCurrent 指向的结点），意味着哨兵指向的结点要“跨过”当前结点（图中虚线），然后当前结点指针 pCurrent 立即指向哨兵结点。下一次数小孩则循着虚线开始。试看 Josephus 问题解法第二版：

```
//* * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* *              Josephus 问题解法二                  * *
//* *              jose2.cpp                             *
//* * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

struct jose       //小孩结点
{
   int code;
   jose * next;
};

int main()
{
   //赋初值
   int numOfBoys,interval;
   cout <<"please input the number of boys,\n"     //小孩数
        <<"           interval of counting:\n";   //数小孩个数
   cin >>numOfBoys >>interval;

   //建立小孩结构数组
   jose * pJose = new jose[numOfBoys];     //从堆内存分配空间
   jose * pCurrent = pJose;     //当前结点指针

   //初始化结构数组：构成环链，小孩编号，输出编号
   int itemsInLine = 0;     //输出项数

   for(int i = 1; i<= numOfBoys; i ++)
   {
    pCurrent ->next = pJose + i % numOfBoys;     //链到下一个元素
    pCurrent ->code = i;                         //小孩编号
    pCurrent = pCurrent ->next;                  //改变当前位置
    if(itemsInLine ++  % 10 == 0)
       cout <<endl;
    cout <<setw(4) <<i;
   }                  //pCurrent 此时等于 pJose
   itemsInLine = 0;

   jose * pivot;           //链表哨兵
   pCurrent = &pJose[numOfBoys - 1];  //下一个就是数组第一个元素 pJose[0]

   while(pCurrent ->next!= pCurrent)  //处理未获胜的所有小孩
   {
```

```
        for(int j = 0; j<interval; j ++ )      //数 interval 个小孩
        {
           pivot = pCurrent;
           pCurrent = pivot - >next;
        }
        if(itemsInLine ++  % 10 == 0)      //输出小孩
           cout <<endl;
        cout <<setw(4) <<pCurrent - >code;
        pivot - >next = pCurrent - >next;      //小孩脱链
        pCurrent = pivot;
      }

      cout <<"\n\nthe winner is "
           <<pCurrent - >code <<endl; //获胜者

      delete[ ]pJose;          //返还堆空间

   }    //endof main()
```

运行结果为：

```
please input the number of boys,
        interval of counting:
15  3
     1  2  3  4  5  6  7  8  9  10
     11  12  13  14  15
     3  6  9  12  15  4  8  13  2  10
     1  11  7  14

 the winner is 5
```

程序从堆中分配小孩结构数组空间，因而可以在运行时确定小孩数，给求解问题带来了灵活性。

在初始化链表的时候，将小孩数组顺序地链接起来，直至最后一个结点链到第一个结点，构成环链表。在环链表的形成过程中，给小孩编号和依次输出全体小孩。

如果起点可以是任意一个小孩，则程序应作如何修改呢？

小结

结构是一种用户定义的数据类型，声明结构时，不产生内存的分配，只有在定义结构变量时，才分配内存空间。

结构作为参数传递时，其值进行了复制，当结构很大时，宜采用结构的引用传递。这时候，结构仅仅传递地址，既省时又省空间。

结构的概念是理解各种数据结构的关键。我们在这里处理的链表称为单向链表，因为这个链表只能从一端向另一端遍历整个链表，反之则不行。此外，还有双向链表、队列、栈、树等，这些内容在数据结构课程中进一步研究。

在堆中分配结构空间时，我们看到 new 操作符比 malloc()函数的使用更简便。

在C++中，结构是类的过渡，类的功能涵盖了结构的一切。当我们在讨论结构的时候，我们往往用结构变量这个词，在讨论类的时候，我们用对象这个词。因为结构中仅仅包含各种数据变量，而类中不但包含数据变量还包含对数据变量的操作。在以后诸章，将对类进行充分的描述。

在C++中，结构在不断退化，因为类可以代表自定义数据类型的一切。与其相关的联合(union)和位段操作也在退化，它们更多地用在与硬件控制有关的低级程序设计中。

用环链表结构来解决Josephus问题是比较适宜的，因为删除结点模拟小孩脱链比较形象，程序比较容易描述和理解。但是程序设计中，将所有算法细节都放在一个主函数中描述，显得复杂。如果是一个大程序，应该怎样划分成若干小源程序呢？

当我们在过程化程序设计中，养成了良好的编程风格，掌握了C++语言要素，搞懂了C++程序结构，把握了C++函数机制，融通了指针和引用，积累了较典型的过程化程序设计经验之后，就可以轻松自在地跨入学习面向对象程序设计方法的进程。

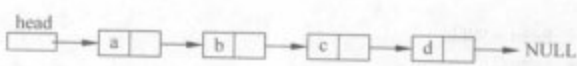
练习

10.1 利用结构类型编制程序，实现输入一个学生的数学期中和期末成绩，然后计算并输出其平均成绩。

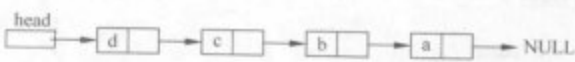
10.2 已知head指向一个带头结点的单向链表，链表中每个结点包含字符型数据和指向本结构结点的指针。编写函数实现在值为"jone"的结点前插入值为"marit"的结点，若没有值为"jone"的结点，则插在链表最后。

10.3 已知head指向一个带头结点的单向链表，链表中每个结点包含数据long和指向本结构结点的指针。编写函数实现如图所示的逆置。

原链表为：



逆置后的链表为：



10.4 读下面的链表操作程序。

(1) 将函数ShowList()和AddToEnd()改成非递归形式(可以修改函数原型)。

```
#include <iostream.h>

struct Lnode
{
   double data;
   Lnode * next;
};

void ShowList(Lnode * list)
{
   if(list)
   {
```

```
    cout <<list->data <<endl;
    if(list->next)
      ShowList(list->next);        //递归调用
  }
}

void AddToEnd(Lnode * new, Lnode * head)
{
  if(head == NULL)
  {
    head = new1;
    new1->next = NULL;
  }
  else
    AddToEnd(new1, head->next)        //递归调用
}

Lnode * GetNode()
{
  Lnode * item;
  Item = new Lnode;
  if(item)
  {
   item->next = NULL;
   item->data = 0.0;
  }
  else
   cout <<"Nothing allocated\n";
  return item;
}

int main()
{
  Lnode * head = NULL;
  Lnode * temp;
  temp = GetNode();
  while(temp)
  {
   cout <<"Data? ";
   cin >>temp->data;
   if(temp->data > 0)
     AddToEnd(temp, head);
   else
     break;
   temp = GetNode();
  }
  ShowList(head);
}
```

(2) 写出下面运行输入之后的运行结果。

```
Data?  3
Data?  5
Data?  7
Data?  6
```

```
Data? 4
Data? 8
Data? -3
```

(3) 在主函数结束之前，增加一个删除整个链表的函数调用 DeleteList()，使得从堆空间分配得到的结构变量能够返还。

10.5 定义两个同种单向链表(结点中包含一个整型数和一个指向本结点类型的指针)，该两链表中数据都已排序好，编制程序，合并这两个链表。

10.6 建立一个 10 结点的单向链表，每个结点包括：学号、姓名、性别、年龄。对其进行排序，采用插入排序法，按学号从小到大排列。

第二部分　面向对象程序设计

第11章　类

类构成了实现C++面向对象程序设计的基础。类是C++封装的基本单元，它把数据和函数封装在一起。当类的成员声明为保护时，外部不能访问；声明为公共时，则在任何地方都可以访问。学习本章后，要求掌握声明和定义类和成员函数的方法，掌握访问成员函数的方法，理解保护数据如何屏蔽外部访问的原理，使得对类的封装有更好的认识。

11.1　从结构到类

C的结构可把相关联的数据元素组成一个单独的统一体。

例如，下面的代码是一个存款结构：

```
struct Savings
{
  unsigned accountNumber; //账号
  float balance; //结算额
};
```

Savings结构的每个实例(对象)包含同样的两个数据元素。

例如，下面的代码定义两个结构对象：

```
void fn()
{  Savings a;          //一个结构的实例：一个银行存款账户
  Savings b;          //又一个结构的实例：另一个银行存款账户
  a.accountNumber = 1;      //一个银行存款的账号
  b.accountNumber = 2;      //另一个银行存款的账号
}
```

a，b是两个不同的对象，其分量accountNumber对应于不同的空间，值可以不同。

任何程序，只要说明了Savings结构对象，就可以修改其对象中属性(分量，或数据成员)的值。

→ 在C中，说明结构对象的方法为：

```
struct Savings a;
```

C++中，说明方法为：

```
Savings a; //关键字 struct 不必要
```

C 的结构不含成员函数。C++的类既能包含数据成员(data member)，又能包含函数成员或称成员函数(member function)。

例如，下面的代码中，Savings 类含有两个数据成员，一个成员函数：

```
class Savings
{
  public:
    unsigned deposit(unsigned amount)     //成员函数
    {
      balance += amount;
      return balance;
    }
  private:
    unsigned accountNumber;               //数据成员
    float balance;
};
```

关键字 class 表示类，Savings 是类名，一般首字符用大写字母表示，以示与对象名的区别。关键字 public 和 protected(或 private)表示存储控制。

在类中说明的，要么是数据成员，要么是成员函数。它们或者说明为 public 的，或者说明为 protected 的，或者说明为 private 的。

类具有封装性，它可以说明哪些成员是 public 的，哪些不是。说明了 protected 的成员，外部是不能访问的。

例如，下面的代码定义两个类对象，对于上面的 Savings 类定义，不能对其数据成员进行访问：

```
void fn()
{
  Savings a;              //定义类对象
  Savings b;
  a.balance = 100.5;      //error: 不能访问 balance,因为它是保护成员
  b.balance = 200.5;      //error
  a.deposit(100);         //ok: 使 balance 赋以值 100,deposit()是公共的
}
```

这里用类 Savings 来定义对象 a 和 b。由于在类 Savings 中，说明了 balance 数据成员是 private(私有)的，所以，在普通函数 fn()中，就不可以直接访问它。而要通过访问类的 public 成员函数间接地访问。

→ 类与结构的区别

C++中，结构是用关键字 struct 声明的类，默认情况下其成员是公共(public)的。例如，Savings 类也可以如下定义：

```
struct Savings
{
  public:       //该行可省略
    unsigned deposit(unsigned amount)     //成员函数
```

```
    {
      balance += amount;
      return balance;
    }
  private:
    unsigned accountNumber;          //数据成员
    float balance;
};
```

C++中类与结构的唯一区别是：类(class)定义中默认情况下的成员是private的，而结构(struct)定义中默认情况下的成员是public的。

在C中，结构中不允许有成员函数。而在C++中可以有成员函数。第10章介绍的结构是C中的结构，这样介绍的目的也是为了分清面向对象程序设计中的类与通常意义下的结构之本质区别。

11.2 软件方法的发展必然

较早的软件开发，用结构化程序设计方法。程序的定律是：

程序＝(算法)＋(数据结构)

即算法是一个独立的整体，数据结构也是一个独立的整体。两者分开设计，以算法(函数或过程)为主。见图11-1。

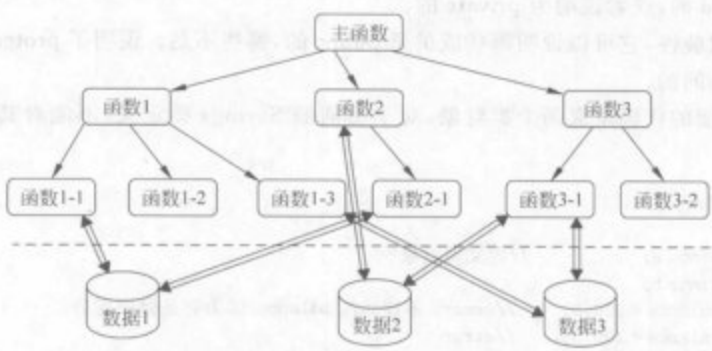


图11-1 算法与数据结构的关系

图中的虚线表示上面的算法与下面的数据分离。双箭头线表示作为黑盒的函数输入输出数据。

在第10章中，所有的程序都是“算法＋数据结构”的典型描述。

随着时间的流逝，软件工程师越来越注重于系统整体关系的表示和数据模型技术(把数据结构与算法看作一个独立功能模块)。程序定律被重新认识：

程序＝(算法＋数据结构)

即算法与数据结构是一个整体，**算法总是离不开数据结构**，算法含有对数据结构的访问，**算法只能适用于特定的数据结构**。因此设计一个算法适合于访问多个数据结构是不明智的，而且数据结构由多个算法来对其进行同种操作也是多余的。见图11-2说明。

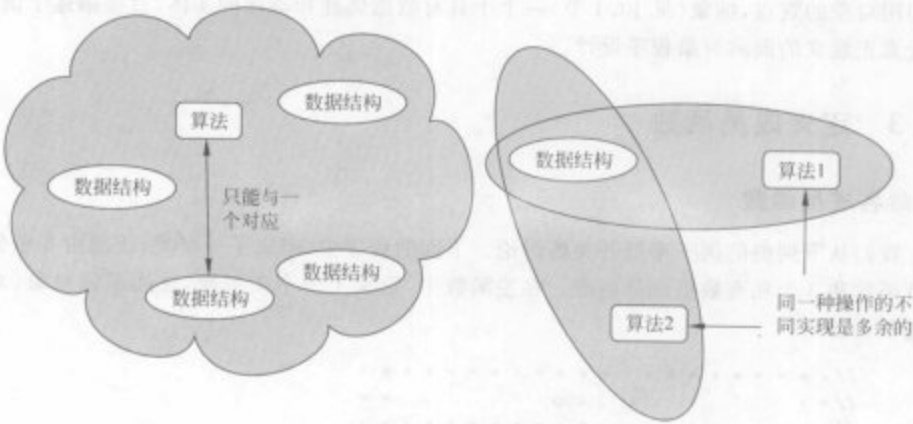


图 11-2　旧的程序定律给算法与数据结构带来不适

这是面向对象程序设计的基础，在面向对象中，算法与数据结构被捆绑成一个类，从这样的角度看问题，就不用为如何实现通盘的程序功能而费尽心机了。现实世界本身就是一个对象的世界，任何对象都具有一定的属性与操作，也就总能用数据结构与算法两者合一地来描述。这时候，程序定律被再次另眼相看：

对象＝（算法＋数据结构）

程序＝（对象＋对象＋……）

也就是说，程序就是许多对象在计算机中相继表现自己。而对象又是一个个程序实体。

人们不再静止地去看待数据结构了，而把它看成是一个程序单位，一个程序分子，或者一个对象的象征。它本身又包含有算法与数据结构。见图 11-3。

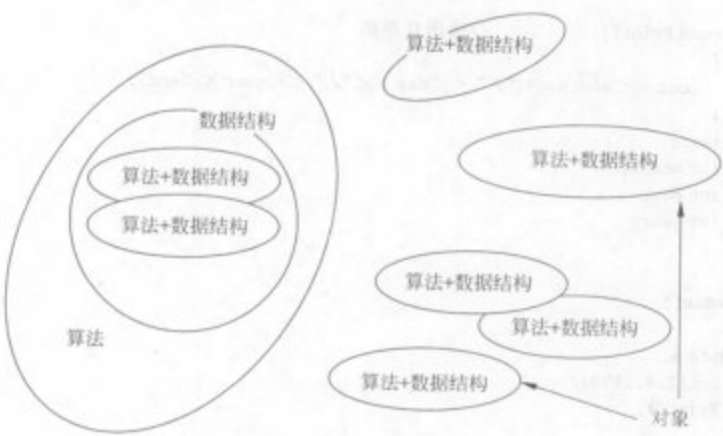


图 11-3　构成程序的对象

由于突破了软件设计思想的障碍，因此程序规模迅速扩大，软件产业得以飞速发展。类的实现机制就这样在程序语言中应运而生，C++是类机制实现得比较完善的一种高级语言。

人们用对象的观点，抽象(见 13.1 节)一个个具有数据属性和动作的实体，直接描述于语言，进行真正意义的面向对象程序设计。

11.3 定义成员函数

1. 命名成员函数

我们从下列类的例子来展开类的讨论。下面的程序中，定义了一个类，该类由 3 个公共成员函数和 3 个私有数据成员组成。在主函数中，定义了一个类对象，使用了该对象(对象表现了自己)：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch11_1.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Tdate
{
  public:
    void Set(int m,int d,int y)     //置日期值
    {
      month = m; day = d; year = y;
    }

    int IsLeapYear()                //判是否闰年
    {
      return (year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
    }

    void Print()                    //输出日期值
    {
      cout <<month <<"/ " <<day <<"/ " <<year <<endl;
    }
  private:
    int month;
    int day;
    int year;
};

int main()
{
  Tdate a;
  a.Set(2,4,1998);
  a.Print();
}
```

运行结果为：

```
2/4/1998
```

主函数开始运行时，首先创建一个 Tdate 类的对象，然后调用 Tdate 的公共成员函数

Set()来给对象的数据成员赋值，再调用成员函数 Print()打印输出结果。

→ 类定义中的 IsLeapYear()成员函数在主函数中并未用到，但其定义仍放在类中。类定义是提供给更多不同用途的程序共享的，并不受单个程序应用的影响而"优化"。这是类定义的常识。

函数 Set(int,int,int)的全名是 Tdate::Set(int,int,int)。类名 Tdate 的作用是指出 Set(int,int,int)是 Tdate 的一个成员函数，而不是其他类的成员函数(如 Teacher::Set(int))，也不是普通函数。没有类名的函数也称为非成员函数，在本章之前接触的都是非成员函数。成员函数也叫方法(mathod)，它多出现于面向对象方法论的叙述中。

数据成员 month 的全名为 Tdate::month。

::叫作用域区分符，指明一个函数属于哪个类或一个数据属于哪个类。::可以不跟类名，表示全局数据或全局函数(即非成员函数)。

例如下面的代码，在成员函数 Set()中，调用了非成员函数 Set()：

```
int month;          //全局变量
int day;
int year;

void Set(int m,int d,int y)        //非成员函数
{
  ::month = m;      //给全局变量赋值，此处可省略
  ::day = d;
  ::year = y;
}

class Tdate
{
  public:
    void Set(int m,int d,int y)        //成员函数
    {
      ::Set(m,d,y);     //调用非成员函数
    }
  private:
    int month;
    int day;
    int year;
};
```

2. 在类中定义成员函数

在 ch11_1.cpp 中，类定义的花括号所包含的 3 个成员函数是在类中定义。在类中定义的成员函数一般规模都比较小，语句只有 1～5 句，而且特别的 switch 语句不允许用。它们一般为内联函数，即使没有明确用 inline 标示。

在 C++中，类定义通常在头文件中，因此这些成员函数定义也伴随着进入头文件。我们知道函数声明一般在头文件，而函数定义不能在头文件，因为它们将被编译多次。如果是内联函数，包含在头文件中是允许的，因为内联函数在源程序中原地扩展。由于在类中定义的成员被默认为内联函数，所以就避免了不能被包含在头文件中的问题。

3. 在类之后定义成员函数

对于大的成员函数来说，直接把代码放在类定义中使用起来十分不便。为了避免这种情况，C++允许在其他地方定义成员函数。

例如，下面的程序将 ch11_1.cpp 中的类定义分成两部分，一部分为类定义的头文件，另一部分是类的成员函数定义，即：

```
//tdate.h        类定义

class Tdate
{
  public:
    void Set(int,int,int);        //成员函数声明
    int IsLeapYear();
    void Print();
  private:
    int month;
    int day;
    int year;
};
```

```
//tdate.cpp        成员函数定义

# include <iostream.h>
# indude "tdate.h"

void Tdate::Set(int m,int d,int y)
{
  month = m;
  day = d;
  year = y;
}

int Tdate::IsLeapYear()
{
  return (year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}

void Tdate::Print()
{
  cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}
```

将类定义和其成员函数定义分开，是目前开发程序的通常做法。我们把类定义（头文件）看成是类的外部接口，类的成员函数定义看成是类的内部实现。将类拿来编制应用程序时，只需类的外部接口（头文件）。这和我们使用标准库函数的道理是一样的，即只需包含某函数声明的头文件。因为类定义中，全部包含了类中成员函数的声明。

例如，上例代码等价于下面的代码，该代码在一个文件中实现类定义和其成员函数定义：

```
# include <iostream.h>

class Tdate
{
```

```
  public:
    void Set(int,int,int);
    int IsLeapYear();
    void Print();
  private:
    int month;
    int day;
    int year;
};

void Tdate::Set(int m,int d,int y)
{
  month = m;
  day = d;
  year = y;
}

int Tdate::IsLeapYear()
{
  return (year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}

void Print()
{
  cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}
```

在类定义的外部定义成员函数，比在类内部定义时，成员函数名前多加上一个类名。如果该函数的前面没有用“类名::”表达形式把它与该类紧紧连在一起，编译器就会认为该函数是一个普通函数，它与类中的成员函数有相同的名字。以后在连接时，会查出缺少与成员函数相对应的定义而报错。

在类定义内部定义成员函数时，类名是省略的（如 ch11_1.cpp）。就像某人在家时，家人都叫他小宝，而出门在外时，别人都叫他范小宝。

类名加在成员函数名之前而不是加在函数的返回类型前。

例如，下面的代码不该将类名加在成员函数定义的最开头：

```
Tdate::void Set(int m,int d,int y)    //error
{
  //...
}
```

4. 重载成员函数

成员函数可以用与传统函数一样的方法重载。但由于类名是成员函数名的一部分，所以一个类的成员函数与另一个类的成员函数即使同名，也不能认为是重载。

例如，下面的代码中定义了 Student 类的两个重载函数，定义了 Slope 类的一个成员函数，定义了一个普通函数，并在主函数中分别调用了它们：

```
class Student
{
  public:
    float grade(){ //... }
```

```
    float grade(float newGPA){ //...}
    //...
  protected:
    //...
};

class Slope
{
  public:
    float grade(){ //... }
    //...
  protected:
    //...
};

char grade(float value){ //... }

void main()
{
  Student s;
  s.grade(3.2);      //Student::grade(float)
  float v = s.grade();        //Student::grade()
  char c = grade(v);          //::grade(float)
  float m = t.grade();        //Slope::grade()
}
```

在主函数运行时，一共调用了四个 grade()函数：s. grade(3. 2)匹配 Student 类中的 grade(float)；s. grade()匹配 Student 类中的 grade()；grade()函数匹配全局函数 grade(float)；t. grade()匹配 Slope 类中的 grade()。

11.4 调用成员函数

1. 调用一个成员函数

一个对象要表现其行为，就要调用它的成员函数。调用成员函数的形式类似于访问一个结构对象的分量，先指明对象，再指明分量。它必须指定对象和成员名，否则无意义。

例如，下面的代码把调用成员的形式搞错了：

```
# include <iostream.h>
# include "tdate.h"

Tdate s;        //全局对象名为 s

void func()
{
  month = 10;      //error: month 是什么，成员还是对象?
  Tdate::month = 10;      //error: month 是 Tdate 类的哪个对象?
  Tdate::Set(2,15,1998);      //error: Set 对哪个对象操作呢?
}
```

又例如，下面的代码是正确的调用成员函数形式：

```
void func()      //普通函数
{
  Tdate oneday;                    //创建对象
  oneday.Set(2,15,1998);           //调用其成员函数
  oneday.Print();
}
```

2. 用指针调用成员函数

对象可以由指针来引导。例如，下面的程序用指针引出对象的成员函数。该程序是一个多文件程序结构，工程 ch11_2.prj 包含有两个源文件：

```
//- - - - - - - - - - - - - - - - - -
//- -   ch11_2.prj   - -
//- - - - - - - - - - - - - - - - - -

ch11_2.cpp
tdate.cpp
```

在 ch11_2.cpp 中，只要包含类 Tdate 的头文件 tdate.h(见 11.3 节)，就可使用该类了：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *      ch11_2.cpp      * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "tdate.h"

void someFunc(Tdate * pS)
{
  pS->Print();     //pS 是 s 对象的指针
  if(pS->IsLeapYear())
    cout <<"oh oh\n";
  else
    cout <<"right\n";
}

int main()
{
  Tdate s;
  s.Set(2,15,1998);
  someFunc(&s);     //对象的地址传给指针
}
```

运行结果为：

```
2/15/1998
right
```

someFunc()是普通函数，所以不用在前面加对象名。s 的地址作为参数调用 someFunc()，在 someFunc()中，s 对象通过指针尽情地表现自己。

3. 用引用传递来访问成员函数

用对象的引用来调用成员函数，看上去和使用对象自身的简单情况一样。

例如，下面的程序是根据 ch11_2.cpp 改编的，它使用了引用传递，而后调用成员函数。该程序同样是一个多文件程序结构，工程 ch11_3.prj 包含有：

```
//- - - - - - - - - - - - - - - - - - -
//- -              ch11_3.prj        - -
//- - - - - - - - - - - - - - - - - - -

ch11_3.cpp
tdate.cpp
```

在 ch11_3.cpp 中，也只要包含类 Tdate 的头文件 tdate.h，就可使用该类：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch11_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "tdate.h"

void someFunc(Tdate& refs)
{
  refs.Print();     //refs 是 s 对象的别名
  if(refs.IsLeapYear())
    cout <<"oh oh\n";
  else
    cout <<"right\n";
}

int main()
{
  Tdate s;
  s.Set(2,15,1998);
  someFunc(s);      //对象的地址传给引用
}
```

运行结果为：

```
2/15/1998
right
```

4. 在成员函数中访问成员

成员函数必须用对象来调用。另一方面，在成员函数内部，访问数据成员或成员函数无须如此。

例如，下面的程序中，成员函数内部访问了数据成员：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch11_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "tdate.h"
```

```
void Tdate::Set(int m,int d,int y)
{
  month = m;              //不在 month 前加对象名
  day = d;
  year = y;
}

void Tdate::Print()
{
  cout <<month <<"/" <<day <<"/" <<year <<endl;
}

int Tdate::IsLeapYear()
{
  return(year % 4 == 0&&year % 100!= 0)||(year % 400 == 0);
}

void main()
{
  Tdate s;
  Tdate t;
  s.Set(2,15,1998);
  t.Set(3,15,1997);
  s.Print();
  t.Print();
}
```

运行结果为：

```
2/15/1998
3/15/1997
```

在主函数中，创建了 s 和 t 对象，然后 s 和 t 对象分别调用了成员函数 Set()。在 Set() 成员函数中访问了 month、day 和 year。

一个类中所有对象调用的成员函数都是同一代码段。那么，成员函数又是怎么识别 month、day 和 year 是属于哪个对象的呢？

原来，在对象调用 s.Set(2,15,1998)时，成员函数除了接受 3 个实参外，还接受到了一个对象 s 的地址。这个地址被一个隐含的形参 this 指针所获取，它等同于执行 this= &s。所有对数据成员的访问都隐含地被加上前缀 this->。所以：

```
month = m;  等价于  this->month = m;  等价于  s.month = m
```

因此，无论对应哪个对象调用的，成员函数从获得的参数（显式的和隐含的）来判断都清楚，这就是成员函数中访问成员无须对象名作前缀的原因。Set()成员函数还可表示成下列代码：

```
void Tdate::Set(int m,int d,int y)
{
  this->month = m;
  this->day = d;
  this->year = y;
}
```

但不可表示成下列代码：

```
void Tdate::Set(int m,int d,int y)
{
  s.month = m;   //error:  s 没有声明
  s.day = d;
  s.year = y;
}
```

因为成员函数是所有对象共享的代码，不是某一个对象所独占的，所以不能在成员函数内使用某个特定的对象。在编译期间，s 对象由于没有在成员函数内部或文件作用域中声明而导致失败。

→ 一个类对象所占据的内存空间由它的数据成员所占据的空间总和所决定。类的成员函数不占据对象的内存空间。

11.5 保护成员

类的成员可以声明为被保护的(protected)，这时，从类的外部(指在普通函数或其他类的成员函数中)就不能对它们访问，也可以把成员声明为公共的，公共成员在任何地方都可以被访问。

在类中设置保护屏障，不让外部访问，主要是由面向对象程序的目标决定的：

(1) 相对于外部函数(普通函数，其他类的成员函数)而言，保护类的内部数据不被肆意侵犯。电视机通过一个接口把它提供给外部世界。面板上的按钮就好像是公共成员函数，人人都可用。但是复杂的电路则装配在机壳内部，就好像是保护成员，与外部隔离，protected 关键字就是电视机外壳。

(2) 使类对它本身内部实现的维护负责。因为只有类自己才能访问该类的保护数据，所以一切对保护数据的维护只有靠类自己了。如果其他的类或函数能够自由访问该类的保护成员，那么，还让该类对它自己的内部实现负责是不公平的。如果某人打开了电视机外壳，改动了内部电路，使电视机遭损，要求厂家退换，厂家概不负责。但是，在使用外部按钮时，突然电视机发生故障，那厂家要负全面的责任，因为电视机是他们造的，内部电路的实现只有他们知道。

(3) 限制类与外部世界的接口。把一个类分作两部分，一部分是公共的，另一部分是保护的。保护成员对于使用者来说是不可见的，也是无须了解的。了解和使用一个具有有限接口(公共成员)的类是很容易的。电视机的按钮就寥寥几个，人们很容易学会使用，无须知道电视机的内部实现，照样可以将电视机用得好好的。

(4) 减少类与其他代码的关联程度。类的功能是独立的，它不依赖于应用程序的运行环境，而可以放到这个程序中使用，也可以放到那个程序中使用。这样，使得你能够非常容易地用一个类替换另一个类。电视机你会用，我也会用；放在你家可以播放，放到我家也可以播放，它并不要求我家具有一个特殊的电源或者专门的天线而工作。

类的保护机制使得人们编制的应用程序更加可靠和易维护。人们在编程时，误用保护成员的代码统统被排斥在编译的防线上。由于使用类的公共成员的错误，而导致程序运行不正常，会很快发现和纠正。当你购买了电视机后，图像不清楚，只要调一下按钮就行了。

如果再调不好，那就去店里换吧，一定是电视机内部有问题。

假定一个学生类 Student，它具有 3 个功能：

增加课程：void AddCourse(int hours,float grade)；

参数 hours 为课程学时，grade 为每学时成绩；

返回当前平均成绩：float Grade()；

返回本学期学时数：int Hours()。

Student 的其余成员可声明为保护的，以便其他函数的操作与学生事物相隔离。类代码实现如下，我们用头文件来保存它：

```
//student.h

class Student
{
  public:
    float Grade()        //取当前平均成绩
    {
      return gpa;
    }

    int Hours()          //取学时数
    {
      return semesHours;
    }

    float AddCourse(int hours,float grade)     //增加课时及成绩
    {
      gpa = semesHours * gpa + grade * hours;     //总分
      semesHours += hours;      //调整学期学时数
      gpa /= semesHours;        //调整平均成绩
    }
  protected:
    int semesHours;     //学期学时数
    float gpa;      //平均成绩
};
```

将这个类定义取名为 student.h，作为一个头文件保存。使用这个类时，只需将该头文件包含进来。

例如，下面的代码企图修改学生成绩：

```
#include <iostream.h>
#include "student.h"

int main()
{
  Student s;

  //...
  s.gpa = 3.5;     //error
  cout <<s.pga <<endl;     //error
}
```

应用程序的编制者想提高某人平均成绩，又怕太高显得虚假，3.5 正好，所以“s.gpa=3.5;”，但是程序编译通不过，因为非法访问了保护数据成员 gpa。后面一句是同一错误。他之所以无法修改他的程序来达到目的，是因为类的设计(头文件 student.h)中，没有单独修改保护数据的成员函数。Grade()和 Hours()都是取数据成员函数。

另一个学生管理员，他想维护学生成绩，编制了下面的应用程序：

```
#include <iostream.h>
#include "student.h"

int main()
{
  Student s;
  //...

  cout <<s.Grade();
  cout <<s.Hours();
  float gpa = s.Grade();
  int hours = s.Hours();
  gpa += 3;              //修改局部变量 gpa 并不能使 s 对象中数据得以修改
  hours += 4.0;
  cout <<s.gpa <<endl;     //error
  cout <<s.hours <<endl;
}
```

他想看一下该学生原来的学分与成绩，然后给他加一门课程的成绩，但是新建局部变量并不能代表 s 对象中的学分和成绩，他使用错了。所有这样那样的错误，都可以简单地查出。

若将上面的代码做如下修改，就可运行得很好，即：

```
#include <iostream.h>
#include "student.h"

int main()
{
  Student s;
  //...

  cout <<s.Grade();
  cout <<s.Hours();

  s.AddCourse(3,4.0);

  cout <<s.Grade() <<endl;
  cout <<s.Hours() <<endl;
}
```

应该养成类编制的书写习惯。类定义中总是以 public、protected 或 private 开始，让人一目了然。

可以把类的成员声明作为私有的(private)，使外部不能访问它们而起到保护作用。一个类定义，如果不写访问控制说明符(public,protected,private)，那么它就默认为 private。

例如，下面的代码没有访问控制说明符：

```
class A
{
  int x;
  int y;
}
```

等价于：

```
class A
{
  private:
    int x;
    int y;
}
```

protected 和 private 的区别，在类的继承中才表现出来，详见 17.5 节。

11.6 屏蔽类的内部实现

类能够保护它的内部状态。在上节中，把数据成员 gpa 声明为保护的，以防止在应用程序中对平均成绩的乱赋值。AddCourse()增加课程的成绩，会使 gpa 发生变化，但不能单独修改 gpa。如果要求有直接修改 gpa 的操作，则类能提供一个成员函数来达到这个目的。

例如，下面的代码在前面的学生类定义(student.h)中，添加了单独修改平均成绩的成员函数 Grade(float)：

```
class Student
{
  public:
    float Grade()
    {
      return gpa;
    }

    float Grade(float newgpa)        //修改平均成绩
    {
      float oldgpa = gpa;
      if(newgpa>= 0&&newgpa<= 5.0)
        gpa = newgpa;

      return oldgpa;
    }
    //...
  protected:
    int semesHours;
    float gpa;
};
```

尽管允许修改成绩，但不是直接访问 gpa，而是告诉成员函数，由成员函数有防备地改。成员函数先审查要赋的值是否合理(在[0,5.0]范围内)，如果不符合要求，则仅将原值返回。

返回原值的目的是防止误操作覆盖原值而使数据丢失。Grade(float)是重载函数。

这就是访问保护数据成员的途径，Student 类补充公共的成员函数去访问保护数据，在公共成员函数中，可以施加种种操作的限制条件，以此对保护数据成员取值范围加以保护。如果以后发现 gpa 数据成员的值不正常，只要查看能改变其值的两个成员函数 Grade(float)和 AddCourse()即可。

编制应用程序，想要使用某个类，所要了解的全部内容是它的公共成员，它们做什么用，参数是什么。我们使用电视机，只要学会几个按钮的用法就可以了，并不需要了解复杂的内部构造原理。

通过限制接口，很容易使用类。

由于条件的改变，或者发现了类中的错误，则只希望改变类的内部代码，而并不要求改变外部应用，因为接口没有变。

例如，下面的程序实现了一个 Point 类，并使用该类计算点的直角坐标和极坐标：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch11_5.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <math.h>

class Point
{
  public:
    void Set(double ix,double iy)   //设置坐标
    {
      x = ix;
      y = iy;
    }

    double xOffset()        //取 y 轴坐标分量
    {
      return x;
    }
    double yOffset()        //取 x 轴坐标分量
    {
      return y;
    }

    double angle()          //取点的极坐标 θ
    {
      return (180/3.14159) * atan2(y,x);
    }

    double radius()         //取点的极坐标半径
    {
      return sqrt(x * x + y * y);
    }
    protected:
      double x;             //x 轴分量
      double y;             //y 轴分量
```

```
};

int main()
{
  Point p;
  double x,y;
  for(;;)          //重复输入 x 和 y 轴分量,直到 x 分量值小于 0
  {
    cout <<"Enter x and y:\n";
    cin >>x >>y;
    if(x<0)
      break;
    p.Set(x,y);
    cout <<"angle = " <<p.angle()
         <<",radius = " <<p.radius()
         <<",x offset = " <<p.xOffset()
         <<",y offset = " <<p.yOffset() <<endl;
  }
}
```

运行结果为：

```
Enter x and y:
10 10
angle = 45,radius = 14.1421,x offset = 10,y offset = 10
Enter x and y:
50 0
angle = 0,radius = 50,x offset = 50,y offset = 0
Enter x and y:
-1 -1
```

类中，保护数据存放点对象的 x,y 坐标，一切对 x,y 的操作都由成员函数来完成。Set() 设置 x,y 坐标，xOffset(),yOffset()分别返回 x,y 的值，angle(),radius()分别返回 x,y 极坐标的值。

math.h 是 C 库函数头文件，反正切函数 atan()以弧度为参数(此处没有用到)，另一个反正切函数 atan2()则以 x,y 的坐标分量做参数，两者是不同的。此处用到的函数原型如下：

```
double sin(double x);     //x 为弧度,求其正弦值
double cos(double x);     //x 为弧度,求其余弦值
double atan2(double y,double x);  //y,x 为坐标分量,求 y/x 的反正切值
double sqrt(double x); //求平方根
```

将 Point 类从程序中分离，成为独立的头文件 point.h，则程序可以写为如下：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch11_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <math.h>
#include "point.h"

int main()
{
```

```
    Point p;
    double x,y;
    for(;;)              //重复输入 x 和 y 轴分量,直到 x 分量值小于 0
    {
      cout <<"Enter x and y:\n";
      cin >>x >>y;
      if(x<0)
        break;
      p.Set(x,y);
      cout <<"angle = " <<p.angle()
           <<",radius = " <<p.radius()
           <<",x offset = " <<p.xOffset()
           <<",y offset = " <<p.yOffset() <<"endl;
    }
}
```

该程序与 ch11_5.cpp 的功能完全一样。其中的头文件为：

```
//* * * * * * * * * * * * * * * * * * * *
//* *              point.h              * *
//* * * * * * * * * * * * * * * * * * * *

#include <math.h>

class Point
{
  public:
    void Set(double ix,double iy) //接口
    {
      x = ix;
      y = iy;
    }
    double xOffset()     //接口
    {
      return x;
    }
    double yOffset()     //接口
    {
      return y;
    }

    double angle()     //接口
    {
      return (180/3.14159) * atan2(y,x);
    }

    double radius()     //接口
    {
      return sqrt(x * x + y * y);
    }
      protected:
        double x;
        double y;
};
```

由于头文件中要用到数学函数，所以 point.h 中包含 math.h 头文件。

由于某种原因，类要修改，但接口不变，也就是说，公共成员函数名字、功能、参数形式不变，那么用户的应用程序就不须改变，即 ch11_6.cpp 无须作任何修改。

下面的 class Point 修改了保护数据结构。类的保护数据成员改为点的极坐标形式 a 和 r，相应地，成员函数的实现也要作修改。具体如下：

```
//* * * * * * * * * * * * * * * * * * * * *
//* *                point.h                *
//* * * * * * * * * * * * * * * * * * * * *

#include <math.h>

class Point
{
  public:
    void Set(double ix,double iy)     //接口
    {
      a = atan2(iy,ix);
      r = sqrt(ix * ix + iy * iy);
    }

    double xOffset()     //接口
    {
      return r * cos(a);
    }
    double yOffset()     //接口
    {
      return r * sin(a);
    }

    double angle()     //接口
    {
      return (180/3.14159) * a;
    }

    double radius()     //接口
    {
      return r;
    }
  protected:
    double a;
    double r;
};
```

类的内部实现改变了，即保护数据成员结构和成员函数内部算法都不一样了，但接口没做任何改变，所以 ch11_6.cpp 程序也不须作改动，运行结果不变。

由于类很好地屏蔽了内部数据表示，所以由类负责的内部实现上的维护不影响应用程序的开发，这大大减轻了开发应用程序的强度。

11.7 再论程序结构

1. 类的作用域

一个类的所有成员位于这个类的作用域内，一个类的任何成员都能访问同一类的任一其它成员。C++认为一个类的全部成员都是一个整体的相关部分。

例如，在 11.5 节中的 Student.h 头文件对 Student 类的成员函数定义中，AddCourse() 成员函数能够访问类中数据成员 semesHours 和 gpa。

类作用域是指类定义和相应的成员函数定义范围。在该范围内，一个类的成员函数对同一类的数据成员具有无限制的访问权。

对类作用域外的一个类的数据成员和成员函数的访问受到程序员的控制。这种思想是要把一个类的数据结构和功能封装起来，从而使得在类的成员函数之外对类的数据进行访问是有限的。

例如，在 Student 类中，保护数据 gpa 是不能让普通函数直接访问的。见 11.5 节。

又例如，m 是 X 类的数据成员，且其他成员函数定义中有同名的局部作用域变量，则：

```
class X
{
  public:
    void f1();
    void f2();
  protected:
    int m;
};

void X::f1()
{
  m = 5;
}

void X::f2()
{
  int m;
  m = 2;      //X::m 被隐藏
}
```

类 X 的数据成员 m 的作用域尽管在类 X 中，但是，成员函数中定义了同名的局部作用域变量后，就把数据成员 m 给隐藏住了。

2. 可见性

类名允许与其他变量名或函数名相同。当类名与程序中的其他变量或者函数名相同时，可以通过下面的方法来正确访问。

(1) 如果一个非类型名隐藏了类型名，则类型名通过加前缀可用。

例如，一个类名被在后面的函数中的形参所覆盖，在该函数内，要定义一个类对象，则加上 class 即可：

```
class Sample     //定义类
{
  //...
};

void func(int Sample)     //函数形参隐藏了类名
{
  class Sample a;         //定义一个对象要用到类名
  Sample ++ ;             //形参的算术操作
  //...
}
```

(2) 如果一个类型名隐藏了一个非类型名,则用一般作用域规则即可。

例如:

```
int s = 0;     //全局变量

void func()
{
  class s{  //... };     //类 s 隐藏了全局变量 s
  s a;                   //定义一个类对象
  ::s = 3;               //引用全局变量
}                        //class s 作用域结束

int g = s;               //用全局变量 s 给变量 g 初始化
```

→ 在函数中定义的类称为局部类,局部类在面向对象程序设计中并不多见。类作为类型也有作用域,局部类的作用域在定义该类的函数块中。

局部类的成员函数必须在类定义内部定义,因为若在类外部和包含该类的函数内部中定义,则导致在函数内部定义函数的矛盾。如果在包含类的函数外部定义,则该局部类无法与其取得联系。

→ 名空间

名空间是指某名字在其中必须唯一的作用域。

C++规定:一个名字不能同时指两种类型。例如:

```
class C     //定义一个类类型
{
  //...
};

typedef int C;     //error: 又定义一个类型取同名
```

非类型名(变量名、常量名、函数名、对象名或枚举成员)不能重名。例如:

```
Student a;       //定义一个对象

void a();        //error: 函数名与对象名同名
```

类型与非类型不在同一名空间。也即在一个作用域中,一个名字可以声明为一个类型,又可声明为一个非类型。当两者同时登场时,类型名要加前缀,以区别非类型名。例如:

```
class stat       //先定义类类型
```

```
{
  //...
};

stat a;                    //定义类对象
void stat(stat * ps);      //函数名与类名相同,它们不在同一名空间
class stat b;              //必须区分 stat 是类型还是函数
stat(0);                   //非类型名: 函数调用
```

又例如:

```
int f(int);          //先定义一个非类型名

class f              //定义一个类
{
  //...
};

class f g;           //必须加 class 以区分 f 是类型名
```

又例如:

```
class A
{
  //...
};

A A;        //定义一个 A 类型的对象 A,合法但不可取
```

3. 类的封装

类的封装的概念首先是,数据与算法(操作)结合,构成一个不可分割的整体(对象)。其次是,在这个整体中一些成员是保护的,它们被有效地屏蔽,以防外界的干扰和误操作。另一些成员是公共的,它们作为接口提供给外界使用。而对该对象的描述即是创建用户定义的类型,对 C++来说,就是类的实现机制。

图 11-4 是一个 Point 类的描述。这是具有一般意义的,许多有关面向对象的书都采用这种形式的描述。

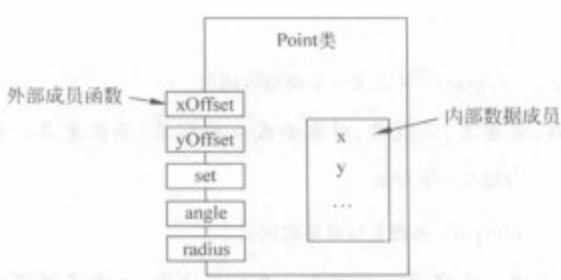


图 11-4 Point 类的描述

把图 11-4 用 C++语言来表达，就是 ch11_4.cpp 中的类定义和相关的成员函数定义。将类的描述分成类的外部接口和类的内部实现就是下面两个文件的代码：

point.h //类的外部接口

```
class Point
{
  public:          //外部接口
    void Set(double ix,double iy);
    double xOffset();
    double yOffset();
    double angle();
    double radius();
  protected:       //内部数据
    double x;
    double y;
    };
```

point.cpp //类的内部实现

```
# include "point.h"
# include <math.h>

void Point::Set(double ix,double iy)
{
  x = ix;
  y = iy;
}
double Point::xOffset()
{
  return x;
}
double Point::yOffset()
{
  return y;
}
double Point::angle()
{
  return (180/3.14159) * atan2(y,x);
}
double Point::radius()
{
  return sqrt(x * x + y * y);
}
```

在 point.h 头文件中，是一个类定义。其中保护数据成员并不是外部接口部分，但作为类定义的整体(从开括号起到闭括号止)描述，只好把它放在头文件中。对于面向对象程序设计之严格的内外界面描述来说，这是 C++类机制中无可奈何的一面。但它不妨碍类的外部接口的有效性。在上一节屏蔽类的内部实现中我们看到了这一点。

在 point.cpp 文件中，每个成员函数都加了类名，在文件开始，包含了两个头文件，一个是 point.h，这是必要的，因为所有成员函数的声明，都在类定义中。这就好像标准函数头文件的结构一样。另一个是 math.h，因为成员函数定义中调用了 math.h 中声明的函数。

4. 类库的构成

一个商业性C++类库包括一个类定义和成员函数定义。类定义以头文件的方式提供，成员函数定义则以一定的计算机硬件或操作系统为背景而编译实现的代码方式提供。

5. C++程序结构

一个C++应用程序是一个程序工程。一个C++程序工程文件中，应该组合下面这些程序文件：

```
main.cpp            //包含主函数的程序文件
class.cpp's         //用户自定义类库的内部实现程序
function.cpp's      //用户自定义函数库的实现程序
```

其中class.cpp's表示多个类成员函数定义的源文件，function.cpp's表示多个函数定义的源文件。

其中包含主函数的源文件应该是下面的形式：

main.cpp 程序文件

```
#include <标准类库头文件>'s
#include <标准函数头文件>'s
#include "自定义类库头文件"'s
#include "自定义函数头文件"'s

函数原型's
全局数据定义's

int main()
{
  //...
}

函数定义's
```

在这里，包含标准类库头文件，即称为类库重用。自定义类库头文件称为类库设计。而在main()函数开始之后的面向对象程序设计，则显得相对自然和简捷。往往是先定义若干对象，然后调用其成员函数，由成员函数来完成程序员所规定的操作(即让对象表现自己)。

小结

一个类具有数据成员，还具有成员函数，通过成员函数可以对数据成员进行操作，并实现其他的功能。

定义了一个类后，可以把该类名作为一种数据类型，定义其"变量"(对象)。

程序利用点操作符(.)访问类的公共成员。

程序可以在类的外部或内部定义它的成员函数，在类的外部定义成员函数时，必须指出所属的类名，并用全局作用域分辨符(::)把类名和函数名连接起来。

类的成员，包括数据和函数，都可以被说明为公有、保护或私有。公有成员可以在程序

中任意被访问，而保护或私有成员只能被这个类的成员函数所访问。

把成员说明为保护的，使类的使用者在使用它时，只关心接口，无须关心它的内部实现，既方便了使用，又保护了内部结构。这就是类的封装原理。

含有类的程序结构，充分体现了类的封装和重用，更容易被人所理解。

练习

11.1 下列程序有几个错误？

```
#include <iostream.h>
#include <math.h>

class Point
{
  public:
    void Set(double ix,double iy)  //设置坐标
    {
      x = ix;
      y = iy;
    }

    double xOffset()          //取 y 轴坐标分量
    {
      return x;
    }

    double yOffset()          //取 x 轴坐标分量
    {
      return y;
    }

    double angle()            //取点的极坐标 θ
    {
      return (180/3.14159) * atan2(y,x);
    }

    double radius()           //取点的极坐标半径
    {
      return sqrt(x * x + y * y);
    }
  protected:
    double x;                 //x 轴分量
    double y;                 //y 轴分量
}

int main()
{
  Point p;
  double x,y;

  cout <<"Enter x and y:\n";
```

```
    cin >>x >>y;

    p.Set(x,y);
    p.x += 5;
    p.y += 6;

    cout <<"angle = " <<p.angle()
         <<",radius = " <<p.radius()
         <<",x offset = " <<p.xOffset()
         <<",y offset = " <<p.yOffset() <<"endl;
}
```

11.2 将下列程序分离类定义和主函数,改成多文件结构。主函数使用类的方式采取包含类定义的头文件的方法。写出运行结果。

```
#include <iostream.h>

class Cat
{
  public:
    int GetAge();
    void SetAge(int age);
    void Meow();        //喵喵叫
  protected:
    int itsAge;
};

int Cat::GetAge()
{
  return itsAge;
}

void Cat::SetAge(int age)
{
  itsAge = age;
}

void Cat::Meow()
{
  cout <<"Meow.\n";
}

int main()
{
  Cat frisky;
  frisky.SetAge(5);
  frisky.Meow();
  cout <<"frisky is a cat who is "
       <<frisky.GetAge()
       <<" years old.\n";
  frisky.Meow();
}
```

11.3 定义一个满足如下要求的 Date 类。

(1) 用下面的格式输出日期：

日/月/年

(2) 可运行在日期上加一天操作；

(3) 设置日期。

11.4 定义一个时间类 Time，能提供和设置由时、分、秒组成的时间，并编出应用程序，定义时间对象，设置时间，输出该对象提供的时间。并请将类定义作为接口，用多文件结构实现之。

11.5 编写一个类，实现简单的栈(提示：用链表结构实现)。数据的操作按先进后出(FILO)的顺序。

成员函数为：

```
void queue::put(int item);     //将数据 item 插入到栈中
int queue::get();              //从栈中取一个数据
```

数据成员为：

一个指向链首的指针

链表结构为：

```
struct Node
{
  int a;
  Node * next;
};
```

使用对象的过程：

```
queue que;

que.put(10);
que.put(12);
que.put(14);

cout <<que.get() <<endl;     //输出 14,栈中剩下 10,12
cout <<que.get() <<endl;     //输出 12,栈中剩下 10
```

第12章 构造函数

C++的构造函数和析构函数，使类对象能够轻灵地被创建和撤销。构造函数创建类对象，初始化其成员，析构函数撤销类对象。构造函数和析构函数是类的特殊成员函数，它们的设计与应用，直接影响编译程序处理对象的方式。构造函数和析构函数的实现使C++的类机制得以充分的展示。所以本章内容是C++的重点之一。学习本章后，要求理解类与对象的区别，掌握定义构造函数和析构函数的方法，把握默认构造函数的意义，了解类成员初始化的问题，掌握构造类成员的方法。

12.1 类与对象

1. 类与对象的区别

人类是一个类，你是人，我是人，都是人类的实例(instance)，或称对象(object)。一个类描述一类事物，描述这些事物所应具有的属性，如人有身长、体重、文化程度、性别、年龄、民族等。

一个对象是类的一个实例，它具有确定的属性，如张三(人的实例)身高180，体重70公斤，大学本科，男，21岁，汉族。

人类只有一个，人类的实例可以有无数个。

对象可以被创建和销毁，但类是无所不在的。

例如，桌子是一个类，人们不断打造各种尺寸和风格(属性)的桌子(桌子的实例)，打造桌子，又不断毁坏桌子，年复一年，旧的去了，新的又来，但桌子的概念没变，它是一个抽象的概念。应该称它为桌子类，以区别于打造的具体桌子。

2. 定义对象

属于不同类的对象在不同的时刻、不同的地方分别被建立。全局对象在主函数开始执行前首先被建立，局部对象在程序执行遇到它们的对象定义时才被建立。与定义变量类似，定义对象时，C++为其分配空间。

例如，下面的代码定义了两个类，创建了类的全局对象、局部对象、静态对象和堆对象：

```
class Desk     //Desk 类
{
  public:
    int weight;
    int high;
    int width;
    int length;
};

class Stool     //另一个类:Stool 类
```

```
{
  public:
    int weight;
    int high;
    int width;
    int length;
};

Desk da;  //全局对象
Stool sa;

void fn()
{
  static Stool ss;     //静态局部对象
  Desk da;     //局部对象
  //...
}

int main()
{
  Stool bs;     //局部对象
  Desk * pd = new Desk;     //堆对象
  Desk nd[50];     //局部对象数组
  //...
  delete pd;     //释放堆对象
}
```

3. 对象的初始化

根据变量定义，全局变量和静态变量在定义(分配空间)时，将位模式清0，局部变量在定义时，分配的内存空间内容保持原样，故为随机数。

对象定义时，情况不一样。对象的意义表达了现实世界的实体，因此，一旦建立对象，须有一个有意义的初始值。C++建立和初始化对象的过程专门由该类的构造函数来完成。这个构造函数很特殊，只要对象建立，它马上被调用，给对象分配空间和初始化。例如一旦打造了一张桌子，桌子就应有长、宽、高和重量。因此，在桌子对象建立时，构造函数的任务是赋予一组值给该桌子对象。如果一个类没有专门定义构造函数，那么C++就仅仅创建对象而不做任何初始化。

C++另有一种析构函数，它也是类的成员函数，当对象撤销时，就会马上被调用，其作用是善后处理。例如，一张桌子要扔掉，须将桌子里面的东西拿出来，这些东西可能有用，不能随桌子一起扔。类似这些事就由析构函数来完成。

12.2 构造函数的需要性

变量初始化的方法如下所示：

```
int a = 1; int * pa = &a;
```

数组初始化的方法如下所示：

```
int b[] = {1,2,3,4};
```

结构初始化的方法如下所示：

```
struct Student
{
  int semesHours;  //总需学时数
  float gpa;     //平均成绩
};

void fn()
{
  Student s = {100, 3.5};     //创建结构变量时，初始化
  //...
}
```

但是对类对象来说，如此初始化不行，这是由类的特殊性所决定的。

例如，下面的代码企图在对象创建时，为其初始化：

```
class Student
{
  public:
//...公共成员...
  protected:
    int semesHours;     //此处的数据成员是受保护的
    float gpa;
};

void fn()
{
  Student s = {100, 3.5};   //error: 不能访问
  //...
}
```

如果上面的函数允许那样初始化的话，就意味着在函数中，允许：

```
void fn()
{
  s.semesHours = 0;
  s.gpa = 0;
}
```

类的封装性，就体现在一部分数据是不能让外界访问的。所以直接在非成员函数中访问类对象的保护或私有数据是不允许的。

类对象的初始化任务，自然就落在了类的成员函数身上，因为它们可以访问保护和私有数据成员。于是将初始化构想成下面的形式：

```
class Student
{
  public:
    void init()
    {
      semesHours = 100;
      gpa = 3.5;
```

```
    }
    //...其他公共成员
  protected:
    int semesHours;
    int gpa;
};

void fn()
{
  Student s;
  s.init();     //类的初始化
  //函数的其他部分
}
```

类的保护数据在外界是不能访问的，所以对这些数据的维护是类的分内工作。将初始化工作交由 init()成员函数无可非议，但却让系统多了一道处理初始化的解释与执行，因为它意味着在编写应用程序中每当建立对象时，都要人为增加书写代码。这样实现的类机制，并不理想。

我们要求建立对象的同时，自动调用构造函数，省去人为调用的麻烦。也就是说，类对象的定义：

```
Student ss;
```

明确表达了为对象分配空间和初始化的意向。这些工作由构造函数来完成。

类对象的定义“Student ss;”涉及到一个类名和一个对象名。类只有一个名字，但对象名可以任意多，每个对象创建时，都要调用该类的构造函数。类的唯一性和对象的多样性，使我们马上想到用类名而不是对象名来作为构造函数名是比较合适的。

12.3 构造函数的使用

C++规定与类同名的成员函数是构造函数，在该类的对象创建时，自动被调用。

例如，下面的代码初始化桌子和凳子类对象：

```
class Desk
{
  public:
      Desk()          //构造函数定义
      {
        weight = 10;
        high = 5;
        width = 5;
        length = 5;
      }
  protected:
      int weight;
      int high;
      int width;
      int length;
};
```

```
class Stool
{
  public:
       Stool()          //构造函数定义
       {
         weight = 6;
         high = 3;
         width = 3;
         length = 3;
}
  protected:
        int weight;
        int high;
        int width;
        int length;
};

void fn()
{
  Desk da;     //自动调用 Desk(),创建对象并初始化
  Stool sa;    //自动调用 Stool()
  //...
}
```

构造函数可以放在类的外部定义。

例如,下面的程序是将上例代码中的构造函数放在类的外部定义:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Desk
{
  public:
       Desk();  //构造函数声明
  protected:
       int weight;
       int high;
       int width;
       int length;
};

class Stool
{
  public:
       Stool();    //构造函数声明
  protected:
       int weight;
       int high;
       int width;
       int length;
};
```

```
Desk::Desk()     //构造函数定义
{
  weight = 10;
  high = 5;
  width = 5;
  length = 5;
  cout <<weight <<"" <<high <<""
       <<width <<"" <<length <<endl;
}

Stool::Stool()     //构造函数定义
{
  weight = 6;
  high = 3;
  width = 3;
  length = 3;
  cout <<weight <<"" <<high <<"" <<width <<"" <<length <<endl;
}

void fn()
{
  Desk da;     //自动调用 Desk()
  Stool sa;     //自动调用 Stool()
}

int main()
{
  fn();
}
```

运行结果为：

```
10 5 5 5
6 3 3 3
```

主函数 main()开始运行时，调用 fn()函数，fn()函数在创建 Desk 对象 da 和 Stool 对象 sa 时，分别调用了两者的构造函数。它们在内存中的空间分配如图 12-1 所示。da 和 sa 对象空间中，依次存放着 weight、high、width 和 length 的值。

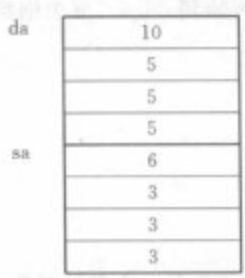


图 12-1　桌子凳子对象在内存中的分配

放在外部定义的构造函数，其函数名前要加上“类名::”，这和别的成员函数定义方法一样。因为在类定义的外部，可能有各种函数定义，为了区分成员与非成员函数，区分此类成员函数和彼类成员函数，所以加上“类名::”是必要的。

构造函数另一个特殊之处是它没有返回类型，函数体中也不允许返回值，但可以有无值返回语句“return;”。因为构造函数专门用于创建对象和为其初始化，所以它不能随意被调用。没有返回类型，正显得它与众不同。

例如，下面的代码是在类定义的外部定义一个构造函数，但是却错误地加上了返回类型：

```
class Desk
{
  public :
        Desk();      //构造函数声明
  protected:
        int weight;
        int high;
        int width;
        int length;
};

void Desk::Desk()     //error: 不能有返回类型
{
  weight = 10;
  high = 5;
  width = 5;
  length = 5;
}
```

如果在 ch12_1.cpp 的函数 fn()中，定义桌子对象的语句改成定义对象数组：

```
void fn()
{
  Desk dd[5];     //对象数组 dd
  Stool sa;
  //...
}
```

则定义对象数组的语句构造函数会被调用 5 次。因为每个元素都是一个对象，每创建一个对象都会调用构造函数。从 dd[0]开始到 dd[4]，逐个创建。所以修改后，ch12_1.cpp 的输出结果为：

```
10 5 5 5
10 5 5 5
10 5 5 5
10 5 5 5
10 5 5 5
6 3 3 3
```

一个类定义中，类的数据成员可能为另一个类的对象。

例如，下面代码的类结构表示在组合音响中包含各个套件类对象：

```
class Recorder
{
  //...
};
class Cdplayer
{
  //...
};
class Amplifier
{
  //...
};
class Tuner
{
  //...
};
class HiFi
{
  public:
//...
  protected:
        Recorder re;
        Cdplayer cd;
        Amplifier am;
        Tuner tu;
};
```

如果一个类对象是另一个类的数据成员，则在那个类的对象创建所调用的构造函数中，对该成员(对象)自动调用其构造函数。

例如，下面的程序，一个“帮教派对”类 TutorPair 中，包含有学生类对象和老师类对象：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_2.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Student
{
  public:
    Student()
    {
    cout <<"constructing student.\n";
    semesHours = 100;
    gpa = 3.5;
    }
  //其他公共成员
  protected:
    int semesHours;
    float gpa;
};

class Teacher
{
```

```
  public:
    Teacher()
    {
      cout <<"constructing teacher.\n";
    }
};

class TutorPair
{
  public:
    TutorPair()
    {
      cout <<"constructing tutorpair.\n";
      noMeetings = 0;
      }
  protected:
    Student student;
    Teacher teacher;
    int noMeetings;     //会晤次数
};

int main()
{
  TutorPair tp;
  cout <<"back in main.\n";
}
```

运行结果为：

```
constructing student.
constructing teacher.
constructing tutorpair.
back in main.
```

主函数 main()运行开始时，遇到要创建 TutorPair 类的对象，于是调用其构造函数 TutorPair()，该构造启动时，首先分配对象空间(包含一个 Student 对象、一个 Teacher 对象和一个 int 型数据)，然后根据在类中声明的对象成员的次序依次调用其构造函数。这里先调用 Student()构造函数，后调用 Teacher()构造函数，最后才执行它自己的构造函数的函数体。按照这个顺序，分别产生运行结果的第一、第二、第三行输出。当执行完 TutorPair()构造函数后，控制回到主函数中，于是，得到第四行输出。

这个例子告诉我们，类在工作过程中，分工十分明确，每个类只负责初始化它自己的对象。当 TutorPair 类要初始化成员 Student 类对象的时候，马上调用 Student 构造函数，而不是由自己去包办。正所谓"你做你的事，我做我的事"。在 12.8 节和 12.9 节中将作进一步介绍。

12.4 析构函数

一个类可能在构造函数里分配资源，这些资源需要在对象不复存在以前被释放。例如，如果构造函数打开了一个文件，文件就需要被关闭。或者，如果构造函数从堆中分配了内存，这块内存在对象消失之前必须被释放。析构函数允许类自动完成这些清理工作，不必调

用其他成员函数。堆对象析构的内容见 14.3 节进一步描述。

析构函数也是特殊的类成员函数，它没有返回类型，没有参数，不能随意调用，也没有重载。只是在类对象生命期结束的时候，由系统自动调用。在后面两节将会看到，构造函数不同于析构函数，却可以有参数，可以重载。

作为一个类，可能有许多对象，每当对象生命期结束时，都要调用析构函数，每个对象一次。这跟构造函数形成了鲜明的对立，所以析构函数名，就在构造函数名前加上一个逻辑非运算符“～”，表示“逆构造函数”。

例如，下面的代码定义了一个析构函数：

```
class XYZ
{
  public:
   XYZ()
   {
     name = new char[20];  //分配堆空间
   }
   ~XYZ()
   {
     delete name;  //释放堆空间
   }
   protected:
   char * name;
}
```

XYZ 类的构造函数中分配了一段堆内存给作为指针的 name 数据成员。一旦对象创建，该对象就在对象空间之外拥有了一段堆内存资源。对应地，当对象在撤销的时候，首先必须归还这一堆内存资源。这个工作由析构函数做。

当你进入图书馆阅览室借书阅览时，你就成了一个阅览室的阅览人(对象)，借什么书是由一进去就完成的(构造)。当你要撤离阅览室(撤销对象)时，你必须先归还图书(析构)才能顺当地离去。

析构函数以调用构造函数相反的顺序被调用。

例如，下面的程序在 ch12_2.cpp 的基础上增加了析构函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_3.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

class Student
{
  public:
    Student()
    {
      cout <<"constructing student.\n";
      semesHours = 100;
      gpa = 3.5;
      }
  ~Student()
```

```
    {
      cout <<"destructing student.\n";
    }
    //其他公共成员
    protected:
      int semesHours;
      float gpa;
};

class Teacher
{
  public:
    Teacher()
    {
      cout <<"constructing teacher.\n";
    }
    ~Teacher()
    {
      cout <<"destructing teacher.\n";
    }
};

class TutorPair
{
  public:
    TutorPair()
    {
      cout <<"constructing tutorpair.\n";
      noMeetings = 0;
    }
    ~TutorPair()
    {
      cout <<"destructing tutorpair.\n";
    }
  protected:
    Student student;
    Teacher teacher;
    int noMeetings;
};

int main()
{
  TutorPair tp;
  cout <<"back in main.\n";
}
```

运行结果为：

```
constructing student.
constructing teacher.
constructing tutorpair.
back in main.
destructing tutorpair.
destructing teacher.
destructing student.
```

当主函数运行到结束的花括号处时，析构函数依次被调用。其顺序正好与构造函数相反。

12.5 带参数的构造函数

前面介绍的构造函数不能完全满足初始化的要求。应该让构造函数可以带参数，否则，往往程序员只能先将对象构造成千篇一律的对象值，甚至一个随机值对象，然后再调用一个初始化成员函数将数据存到该对象中去。

例如，下面的代码构造一个人类的对象：

```
class Mankind
{
  public:
  //...
  protected:
    int age;
    int sex;
    Date birthday;
};

Mankind a;
```

在该人类的定义中，构造函数是默认的，所以创建的全局对象，其初始值全为 0。对于一个人，0 岁，无性别，无出生年月，是无意义的。创建对象应该是构造一个活生生的人。

带参数的构造函数使初始化一步到位。

例如，下面的程序定义了一个带参数的学生类：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_4.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
# include <string.h>

class Student
{
  public:
    Student(char * pName)
    {
      cout <<"constructing student" <<pName <<endl;
      strncpy(name,pName,sizeof(name));
      name[sizeof(name) - 1] = '\0';
    }
    ~Student()
    {
      cout <<"destructing" <<name <<endl;
    }
    //其他公共成员
  protected:
    char name[20];
```

```
};

void main()
{
  Student ss("Jenny");
  //...
}
```

运行结果为：

```
constructing student Jenny
destructing Jenny
```

学生类中定义了一个带字符串参数的构造函数。在主函数运行时，调用了 Student() 构造函数，创建对象 ss，在这当中，输出一行信息，将参数"Jenny"复制给对象 ss，并在对象的数据成员 name 数组的最后一个元素 name[19]填上'\0'。

构造函数在参数规定上和普通函数一样，可以有任意多个参数。

例如，下面的程序修改了程序 ch12_4.cpp 中的学生类，含有 3 个参数：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_5.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName, int xHours, float xgpa)
    {
      cout <<"constructing student" <<pName <<endl;
      strncpy(name,pName,sizeof(name));
      name[sizeof(name) - 1] = '\0';
      semesHours = xHours;
      gpa = xgpa;
    }
    ~Student()
    {
      cout <<"destructing" <<name <<endl;
    }
    //其他公共成员
  protected:
    char name[20];
    int semesHours;
    float gpa;
};

int main()
{
  Student ss("Jenny",22,3.5);
  //...
}
```

运行结果为：

```
constructing student Jenny
destructing Jenny
```

该程序运行结果与 ch12_4.cpp 完全一样，但是，Student 对象空间不一样了。由于增加了数据成员，为初始化它们，构造函数带了 3 个参数，在主函数中，创建对象的语句要求对象名 ss 相应的也包括 3 个参数。它所创建的学生名叫"Jenny"，学期学时数为 22，平均成绩 3.5。

12.6 重载构造函数

构造函数可以被重载，C++根据声明中的参数选择合适的构造函数。

例如，下面的程序同时声明了四个构造函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch12_6.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Tdate
{
  public:
    Tdate();
    Tdate(int d);
    Tdate(int m,int d);
    Tdate(int m,int d,int y);
    //其他公共成员
  protected:
    int month;
    int day;
    int year;
};

Tdate::Tdate()
{
  month = 4; day = 15; year = 1995;
  cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}
Tdate::Tdate(int d)
{
  month = 4; day = d; year = 1996;
  cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}
Tdate::Tdate(int m,int d)
{
  month = m; day = d; year = 1997;
  cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}
Tdate::Tdate(int m,int d,int y)
{
  month = m; day = d; year = y;
```

```
    cout <<month <<"/ " <<day <<"/ " <<year <<endl;
}

int main()
{
  Tdate aday;
  Tdate bday(10);
  Tdate cday(2,12);
  Tdate dday(1,2,1998);
}
```

运行结果为：

```
4/15/1995
4/10/1996
2/12/1997
1/2/1998
```

因为对象 aday 以无参数形式给出，所以用无参构造函数 Tdate()来构造它，无参的构造函数被称为默认构造函数。后面三个对象分别匹配三个不同的构造函数。

由于构造函数用于创建对象，所以调用它来给对象赋值是错误的。

例如，下面的代码中，构造函数企图再次调用另一个重载的构造函数来简化编程：

```
Tdate::Tdate(int d)
{
  Tdate(4,d,1995);   //构造函数创建一个无名对象
}
```

显式调用构造函数将创建一个无名对象，关于无名对象，14.8 节将专门介绍。要想共享初始化的过程，可以先定义一个共享成员函数，然后每个构造函数都调用之。例如，下面的程序定义了一个名叫 Init 的成员函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_7.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Tdate
{
  public:
    Tdate(){ Init(4,15,1995); }
    Tdate(int d){ Init(4,d,1996); }
    Tdate(int m,int d){ Init(m,d,1997); }
    Tdate(int m,int d,int y){ Init(m,d,y); }
    //其他公共成员
  protected:
    int month;
    int day;
    int year;
    void Init(int m,int d,int y)
    {
      month = m;
```

```
        day = d;
        year = y;
        cout <<month <<"/ " <<day <<"/ " <<year <<endl;
    }
};

int main()
{
  Tdate aday;
  Tdate bday(10);
  Tdate cday(2,12);
  Tdate dday(1,2,1998);
}
```

运行结果为：

```
4/15/1995
4/10/1996
2/12/1997
1/2/1998
```

Tdate 类增加了一个做具体初始化工作的 Init()函数，由于它仅仅被用于构造函数的调用，即仅被类的成员函数调用，所以可以将其放在保护成员内，以阻止其他应用程序使用。

还可以通过给最后一个构造函数以参数默认值，使这四个构造函数结合成一个。

例如，下面的程序在类中，将 4 个构造函数结合为一个单一的构造函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch12_8.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Tdate
{
  public:
    Tdate(int m = 4, int d = 15, int y = 1995)
    {
      month = m;
      day = d;
      year = y;
      cout <<month <<"/ " <<day <<"/ " <<year <<endl;
    }
    //其他公共成员
    protected:
    int month;
    int day;
    int year;
};

int main()
{
  Tdate aday;
  Tdate bday(2);
  Tdate cday(3,12);
```

```
    Tdate dday(1,2,1998);
}
```

运行结果为：

```
4/15/1995
2/15/1995
3/12/1995
1/2/1998
```

创建对象时，可以默认年，默认日与年，也可以日、月、年都默认，其他条件不允许默认。决定使用哪一个构造函数的规则和解决调用其他重载函数的规则相同。重载构造函数若与参数默认值的构造函数发生冲突，则创建对象的语句会导致编译错误。参数默认与函数重载的内容可以回顾 5.8 节和 5.9 节。

例如，下面的代码在类中重载构造函数，其定义存在二义性问题：

```
class Tdate
{
  public:
    Tdate(int d)
    {
      month = 4;
      day = d;
      year = 1998;
    }
    Tdate(int m,int d = 12)
    {
      month = m;
      day = d;
      year = 1997;
    }
    //其他公共成员
  protected:
    int month;
    int day;
    int year;
    };

int main()
{
Tdate aday(11);  //error 匹配哪个构造函数
}
```

12.7 默认构造函数

(1) C++规定，每个类必须有一个构造函数，没有构造函数，就不能创建任何对象。

(2) 若未提供一个类的构造函数(一个都未提供)，则 C++提供一个默认的构造函数，该默认构造函数是个无参构造函数，它仅负责创建对象，而不做任何初始化工作。

(3) 只要一个类定义了一个构造函数(不一定是无参构造函数)，C++就不再提供默认的构造函数。也就是说，如果为类定义了一个带参数的构造函数，还想要无参构造函数，则

必须自己定义。

(4) 与变量定义类似，在用默认构造函数创建对象时，如果创建的是全局对象或静态对象，则对象的位模式全为 0，否则，对象值是随机的。

例如，下面的代码在创建对象时，将自动调用系统提供的默认构造函数：

```
class Student
{
  //无构造函数
  protected:
    char name[20];
};

int main()
{
  Student noName; //ok: noName 的内容为随机值
}
```

上例中的类定义等价于下面的类定义：

```
class Student
{
  public:
    Student(){}  //一个空的无参构造函数
  protected:
    char name[20];
};
```

又例如，下面的代码定义了一个带参数的构造函数，面对创建无参对象，将不能正确地编译：

```
#include <string.h>

class Student
{
  public:
    Student(char * pName)
    {
      strncpy(name,pName);
    }
  protected:
    char name[20];
};

int main()
{
  Student noName;  //error: 无匹配的构造函数
}
```

如果增加一个无参的构造函数，就可解决这个问题：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_9.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *
```

```
#include <string.h>
class Student
{
  public:
    Student(char * pName)
    {
      strncpy(name,pName);
      name[sizeof(name) - 1] = '\0';
    }
    Student(){}
  protected:
    char name[20];
};

int main()
{
  Student noName;  //ok
  Student ss("Jenny"); //ok
}
```

→ 说明的含糊性

先前的例子中创建 Tdate 类对象的方法是：

```
Tdate aday;  //为什么创建无参的对象无括号?
Tdate bday(2);  //对象的参数放在括号中
Tdate cday(3,12);  //对象的参数放在括号中
Tdate dday(1,2,1998);  //对象的参数放在括号中
```

Tdate aday(); //为什么不能这样声明?

根据 C++的语法规定，这样是声明了一个名叫 aday 的普通函数，它返回 Tdate 类对象，并且没有参数。

又例如下面的代码，一个是创建对象，一个是声明函数：

```
Tdate oneday(10);       //创建对象
Tdate oneday(int);      //声明函数
```

12.8 类成员初始化的困惑

在一个类中，如果有用户自定义的类对象作为其成员，那构造函数怎么做呢?

例如，下面的程序定义了学号类和学生类，学生类中包含有学号类：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_10.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

int nextStudentID = 0;
class StudentID
{
```

```
  public:
    StudentID()
    {
      value = ++nextStudentID;
      cout <<"Assigning student id " <<value <<endl;
    }
    ~StudentID()
    {
      --nextStudentID;
      cout <<"Destructing id " <<value <<endl;
    }
  protected:
    int value;
};

class Student
{
  public:
    Student(char * pName = "noName")
    {
      cout <<"Constructing student " <<pName <<endl;
      strncpy(name,pName);
      name[sizeof(name) - 1] = '\0';
    }
  protected:
    char name[20];
    StudentID id;
};

int main()
{
  Student s("Randy");
}
```

运行结果为：

```
Assigning student id 1
Constructing student Randy
Destructing id 1
```

当学生类对象被构造时，一个学号赋予了该学生对象。

学号类 StudentID 含有保护数据成员 value，按规定不能被另一个类 Student 的类对象访问，即使 Student 类包含 StudentID 类。

Student 类包含一个成员 id，id 属于 StudentID 类，Student 构造函数不能访问 id 对象中的保护数据成员。所以，在 Student 类对象构造时，须调用 StudentID 类的构造函数来初始化 id。这就是类与类之间“互不干涉内政”的严酷关系。

“Student s("Randy");”语句执行步骤如下：

(1) 分配 s 对象空间，调用 Student 构造函数。

(2) 建立 s 对象空间中的结构，第一为 name[20]，第二为 id，因它属于 StudentID 类，所以调用 StudentID 的构造函数，这时，id 的保护数据 value 得到了赋值，全局变量 nextStudentID 得到了赋值，并且输出第一行信息。之后，返回到 Student 构造函数。

(3) 执行 Student 构造函数体。输出第二行信息，数据成员 name 得到了赋值。之后返回到主函数 main()。

如果 Student 类未定义构造函数，C++提供的默认构造函数将自动地为各个数据成员(如果是类对象的话)调用构造函数。程序结束时也与此相同。类的析构函数将自动地为各个具有析构函数的数据成员调用析构函数。

我们看到，上面的例子中，Student 构造函数调用了 StudentID 的默认构造函数。但如果想调用的构造函数不是默认构造函数，那又如何呢?

例如，下面的程序在创建学生对象时，赋予一个学号，希望将这个学号传给成员——学号类对象 id 保存：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_11.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class StudentID
{
  public:
    StudentID(int id = 0)
    {
      value = id;
      cout <<"Assigning student id " <<value <<endl;
    }
    ~StudentID()
    {
      cout <<"Destructing id " <<value <<endl;
    }
  protected:
    int value;
};

class Student
{
  public:
    Student(char * pName = "noName", int ssID = 0)
    {
      cout <<"Constructing student " <<pName <<endl;
      strncpy(name,pName);
      name[sizeof(name) - 1] = '\0';
      StudentID id(ssID);    //希望将学号传给学号类对象
    }
  protected:
    char name[20];
    StudentID id;
};

int main()
{
  Student s("Randy",9818);
}
```

运行结果为：

```
Assigning student id 0
Constructing student Randy
Assigning student id 9818
Destructing id 9818
Destructing id 0
```

StudentID 的构造函数改为从参数中接受一个学号值。在 Student 构造函数内部，增加了一条语句“StudentID id(ssID);”，企图将 ssID 参数值传给数据成员 id，但实际上，在 Student 构造函数中构造了一个名为 id 的 StudentID 局部对象。由于是局部对象，所以，当 Student 构造函数返回时，便析构了这个对象。从运行结果的第三、第四行中可以看到这一点。而真正希望被传递的 Randy 对象学号值却为 0，未被赋以 9818。

那么，能否像下面这样在 Student 类中直接给 id 对象初始化呢？

```
class Student
{
  public:
    Student(char * pName = "noName", int ssID);
  protected:
    char name[20];
    StudentID id(9818);  //error: 类定义是不分配空间和初始化的
};
```

“StudentID id(9818);”是创建对象语句，而不是类定义中允许的声明数据成员形式。“StudentID id=9818;”或者“StudentID id(ssID);”也都是不允许的。前者就是“StudentID id(9818);”后者 ssID 成了参数，类定义是不会调用构造函数的。因为**类是一个抽象的概念，并不是一个实体，并不含有属性值，而只有对象才占有一定的空间，含有明确的属性值。**我们只能按照格式“类型 标识符;”去声明类的数据成员。

12.9 构造类成员

我们需要一个机制来表示“构造已分配了空间的对象成员，而不是创建一个新对象成员”。该机制应在建立对象空间的结构时反映出来，即需要出现在函数调用刚刚转入之时，函数体执行(开括号)之前。为了做到这一点，C++定义了一个新构造方式。

例如，下面的程序修改了 ch12_11.cpp 中 Student 构造函数的错误：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch12_12.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
# include <string.h>

class StudentID
{
  public:
    StudentID(int id = 0)
    {
```

```
        value = id;
        cout <<"Assigning student id " <<value <<endl;
    }
    ~StudentID()
    {
        cout <<"Destructing id " <<value <<endl;
    }
protected:
    int value;
};

class Student
{
public:
    Student(char * pName = "no name", int ssID = 0):id(ssID)
    {
        cout <<"Constructing student " <<pName <<endl;
        strncpy(name, pName, sizeof(name));
        name[sizeof(name) - 1] = '\n';
    }
protected:
    char name[20];
    StudentID id;
};

int main()
{
    Student s("Randy", 9818);
    Student t("Jenny");
}
```

运行结果为：

```
Assigning student id 9818
Constructing student Randy
Assigning student id 0
Constructing student Jenny
Destructing id 0
Destructing id 9818
```

在 Student 构造函数头的后面，冒号表示后面要对类的数据成员的构造函数进行调用。ssID 可以是 Student 构造函数的形参，id(ssID)表示调用以 ssID 为实参的 StudentID 构造函数。

上例中，Student 构造函数头冒号后面如果是 id()的形式，表示调用 StudentID 的默认构造函数，并且可以省略。即：

```
Student(char * pName = "no name"):id()
{
    cout <<"Constructing student " <<pName <<endl;
    strncpy(name, pName);
    name[sizeof(name) - 1] = '\n';
}
```

可以写成：

```
Student(char * pName = "no name")
{
  cout <<"Constructing student " <<pName <<endl;
  strncpy(name,pName);
  name[sizeof(name) - 1] = '\n';
}
```

调用哪个构造函数完全是看参数匹配的情况，如果在 ch12_12.cpp 的 StudentID 构造函数中，参数没有默认值，即原型为：

```
StudentID(int id);
```

Student(...)；中后面无冒号形式，表明 StudentID 无默认构造函数，而 Student 构造函数的原型为：

```
Student(char * pName = "no Name", int ssID);
```

表明将调用 StudentID 的默认构造函数，于是，主函数中的两条创建 Student 对象的语句都要遇到不能匹配 StudentID 构造函数的编译错误。

冒号语法使得常量数据成员和引用数据成员的初始化成为可能。

例如，下面的程序给一个常量和一个引用初始化：

```
class SillyClass
{
  public:
    SillyClass(int& i):ten(10),refI(i){}
  protected:
    const int ten;          //常量数据成员
    int& refI;              //引用数据成员
};

int main()
{
  int i;
  SillyClass sc(i);
}
```

因为常量是不能被赋值的，一旦初始化后，其值就永不改变，另外，引用变量也是不可重新指派的，初始化之后，其值就固定不变了。所以，不允许像下面这样对常量和引用赋值或重新指派：

```
class SillyClass
{
  public:
    SillyClass()
    {
      ten = 10;     //error: 常量不能赋值
      refI = i;     //error: 引用不能重新指派
    }
  protected:
    const int ten;
    int& refI;
};
```

在 SillyClass 类的构造函数进入之后(开始执行花括号后的函数体语句时),对象结构已经建立,数据成员 ten 和 refI 已经存在,所以再在构造函数体内对常量赋值或对引用指派就不是初始化了。常量和引用的初始化必须放在构造函数正在建立数据成员结构的时候,也就是放在构造函数的冒号后面。

对于类的数据成员是一般变量的情况,则放在冒号后面与放在函数体中初始化都一样。

例如,下面两种初始化一个整数变量数据成员的方法都可以用:

```
class SillyClass1
{
  public:
    SillyClass1()
    {
      d = 10;          //方法 1
    }
    protected:
    int d;
};

class SillyClass2
{
  public:
    SillyClass2():d(10){}        //方法 2
  protected:
    int d;
};
```

→ 说明一个变量并初始化有两种形式:

```
int main()
{
  int m = 10;   //ok
  int n(20);   //TC++3.0 不允许,C++新标准允许
  Student s = "Jenny";   //ok
  Student t("Danny");  //ok: 类的形式不受限制
}
```

赋值时只有一种方法:

```
m = 10;
n(20);     //此不是赋值,而是调用名叫 n 的函数
```

构造函数的冒号后面的初始化不允许用第一种形式:

```
class SillyClass
{
  public:
     SillyClass():d = 10 //error
     {
     }
  protected:
    int d;
};
```

12.10 构造对象的顺序

在一个大程序中，各种作用域的对象很多，有些对象包含在别的对象里面，有些对象早在主函数开始运行之前就已经建立。**创建对象的唯一途径是调用构造函数**。构造函数是一段程序，所以构造对象的先后顺序不同，直接影响程序执行的先后顺序，导致不同的运行结果。C++给构造对象的顺序作了专门的规定。

1. 局部和静态对象，以声明的顺序构造

局部和静态对象是指块作用域和文件作用域的对象。它们声明的顺序与它们在程序中出现的顺序是一致的。

例如，下面的程序用了 goto 语句，企图绕过变量定义：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch12_13.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
  int m = 5;
  if(m = = 5)
goto abc;

  int n = 0;              //不能被 C++编译器所接受
abc:
  cout <<"m = " <<m <<",n = " <<n <<endl; //n 已有定义
}
```

在 C 中，其运行结果为：

```
m = 5,n = 0
```

程序中，“int n=0;”被语句“goto abc;”跳过，这将导致 C++编译器报错。即使在 C 中可以接受，但也不是根据运行顺序来决定变量定义的顺序，而是所有的变量和对象都在函数开始执行时，统一定义。统一定义的顺序正是这些变量和对象在函数中出现的顺序。

2. 静态对象只被构造一次

静态对象和静态变量一样，文件作用域的静态对象在主函数开始运行前全部构造完毕。块作用域中的静态对象，则在首次进入到定义该静态对象的函数时，进行构造。

例如，下面的程序在一个函数中定义了一个静态对象：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch12_14.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>
```

```
class SmallOne
{
  public:
    SmallOne(int sma)
    {
      cout <<"Smallone constructing with a value of"
           <<sma <<endl;
    }
};

void fn(int n)
{
  static SmallOne sm(n);
  cout <<"In function fn with n = " <<n <<endl;
}
int main()
{
  fn(10);
  fn(20);
}
```

运行结果为：

```
Smallone constructing with a value of 10
In function fn with n = 10
In function fn with n = 20
```

3. 所有全局对象都在主函数 main()之前被构造

和全局变量一样，所有全局对象在主函数开始运行之前，全部已被构造。

这会给调试带来问题。当要开始调试时，所有全局对象的构造函数都已被执行，如果它们中的一个有致命错误，那么你可能永远也得不到控制权。这种情况下，该程序在它开始执行之前就死机了。

有两种方法可以解决这个问题：一是将全局对象先作为局部对象来调试；二是在所有怀疑有错的构造函数的开头，增加输出语句，这样在程序开始调试时，你可以看到来自这些对象的输出信息。

4. 全局对象构造时无特殊顺序

全局对象不像局部对象的构造顺序那么简单，全局对象是没有这样的控制流向来指明其顺序的。对于简单应用的单文件程序来说，全局对象可以按照它们出现的顺序依次进行构造。但是，单文件程序只是出现在实验室或教室里，真正有用的程序都是由多个文件组成的，这些文件被分别编译、连接。因为编译器不能控制文件的连接顺序，所以它不能决定不同文件中全局对象之间的构造顺序。

例如，下面的代码是个多文件程序结构，创建了两个全局对象：

```
//student.h

class Student
{
```

```
  public:
    Student(int d):id(d){}
  protected:
    const int id;
};
class Tutor
{
  public:
    Tutor(Student& s)
    {
      id = s.id;
    }
  protected:
int id;
};
//file1.cpp

Student ra(1234);   //建立一个 Student 全局对象

//file2.cpp

Tutor je(ra);   //建立一个 Tutor 全局对象,以使辅导员能帮助学生
```

程序中,Tutor 全局对象使用了 Student 全局对象,它隐含假定了学生类对象先于辅导员类对象构造。但是这样的设计会引起运行中不可预料的错误。为避免这个问题,不要允许一个全局对象访问另一个全局对象。

5. 成员以其在类中声明的顺序构造

例如,下面的代码在构造函数头的冒号后初始化两个成员,但是却并不如意:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch12_15.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * * *
# include <iostream.h>

class A
{
  public:
    A(int j):age(j),num(age + 1)
    {
      cout <<"age:" <<age <<",num:" <<num <<endl;
    }
  protected:
    int num;
    int age;
};

int main()
{
  A sa(15);
}
```

运行结果为：

```
age:15,num:2
```

由于按成员在类定义中的声明顺序进行构造，而不是按构造函数说明中冒号后面的顺序，所以 num 成员被赋的是一个随机值，并不是想赋的 16，因为这时候，成员 age 还没有被赋值，age 的内存空间中是一个随机值。

小结

构造函数是一种用于创建对象的特殊成员函数，人们调用一个构造函数来为类对象分配空间，给它的数据成员赋初值，以及其他请求资源的工作。每个类对象都必须在构造函数中诞生，一个类可能拥有一个或多个构造函数，编译程序为了决定调用哪个构造函数，要把对象声明中使用的变元(实参)和构造函数的参数进行比较，该过程与普通重载函数匹配函数调用的方法相同。

在包含对象成员的类对象被创建时，需要对象成员的创建，相应地调用对象成员的构造函数。然而，构造对象成员的顺序要看类中声明的顺序，而不是看构造函数说明中冒号后面成员初始化的顺序。

构造函数尚有一些内容将在后面的章节中介绍。如何在堆中创建对象以及拷贝构造函数在第 14 章介绍，派生类的构造函数在第 16 章介绍。

练习

12.1　写出下列程序的输出。

```
#include <iostream.h>

class MyClass
{
  public:
    MyClass();
    MyClass(int);
    ~MyClass();
    void Display();
  Protected:
     int number;
};

MyClass::MyClass()
{
  cout <<"Constructing normally\n";
}

MyClass::MyClass(int m)
{
  number = m;
```

```
}

void MyClass::Display()
{
  cout <<"Display a number: " <<number <<endl;
}

MyClass::~MyClass()
{
  cout <<"Destructing\n";
}

int main()
{
  MyClass obj1;
  MyClass obj2(20);

  obj1.Display();
  obj2.Display();
}
```

12.2 创建一个 Employee 类，该类中有字符数组，表示姓名、街道地址、市、省和邮政编码。把表示构造函数、ChangeName()、Display()的函数原型放在类定义中，构造函数初始化每个成员，Display()函数把完整的对象数据打印出来。其中的数据成员是保护的，函数是公共的。

12.3 修改练习 12.2 中的类，将姓名构成类 Name，其名和姓在该类中为保护数据成员，其构造函数为接收一个指向完整姓名字符串的指针，其 Display()函数输出姓名。然后将 Employee 类中的姓名成员(字符数组)换成 Name 类对象。

将所有原型化的函数加上成员函数定义，作为类的内部实现文件。

构成完整的类库定义，要求类定义与类的成员函数定义分开。

12.4 根据练习 12.3，编制主函数如下，构成一个完整的多文件程序。

```
int main()
{
  Employee em("Mark Brooks","5 West St.","Revere","CA","12290");

  char buffer[255];
  em.Display()

  em.ChangeName("Richard Voss");
  em.Display();
}
```

第 13 章　面向对象程序设计

C++区别于 C 的特征是 C++支持面向对象程序设计。在知道了 C++中如何创建类后，必须搞清什么是面向对象程序设计，类适用于现实世界中的哪些问题，才能真正进行面向对象的思考和编程。学习本章后，应该了解结构化编程和面向对象编程的方法以及两者的区别；学会抽象和分类以及简单的面向对象程序设计。

13.1　抽象

面向对象程序设计基于两个原则：抽象和分类。

抽象与具体相对应。一个人名是抽象，它代表某人的一切属性，包括身高、体重、文化程度等。抽象是具体事物的描述的一个概括。

试想一下用微波炉炖鸡蛋。打 2 个鸡蛋在碗里，放上一点调料，把它整个放进微波炉里，烘烤 5 分钟。

使用微波炉的步骤是，先打开门，把制作的原料放进去，然后关好门并按微波炉前面控制板上的有关按钮，它就开始工作了。

使用微波炉，人们处于下面的状态：

(1) 不用重新设计布局，不用改变微波炉的内部结构即可使它工作。人们使用微波炉，只需跟微波炉的面板打交道。微波炉有一个接口，就是微波炉的面板，板上有所有的控制按钮和时间显示。微波炉的所有功能都是通过面板控制获得的。

(2) 不用重新编制软件来驱动和控制微波炉中的微处理器，即与上次使用微波炉的目的无关。

(3) 不用了解微波炉的内部结构。

(4) 一个微波炉的设计师，知道微波炉的内部一切设计细节，但在生活中使用微波炉只是用于烧菜热菜，而无须考虑其工作原理。

现实生活中，为了减少必须处理的事情，我们是在某一程度的细节中生活的。在面向对象的计算机世界中，这种细节程度就叫抽象。

在做菜时，人们仅仅把微波炉看成是一个厨房用品而使用，不会考虑微波炉的内部结构。既然只是通过它的接口来使用微波炉，按照显示的提示去做，就不会使微波炉进入不正常的工作状态而损坏微波炉或把菜烤焦。

如果正常操作下，却被炉壁烫伤了手，或微波炉冒出了火花等，那就是微波炉的质量问题。如果误操作引起菜烧焦了，或烧不熟，那就要调整操作。这在面向对象程序设计中是分工明确的两种编程：一种是面向对象应用程序设计，一种是类库设计。它们都属于面向对象程序设计范畴。

如果操作微波炉之前改动了微波炉的内部结构，或更换了一些电路，那么任何烫伤等事故概由操作人负责。面向对象程序设计中，这就像是修改了类库。那么，类库的维护也应由

该程序员负责到底。

用面向对象的方法，描述在微波炉中炖蛋的过程时，首先定义这个问题中对象的类型：蛋，微波炉，还有调料。然后，着手设计制作这些对象的模型。即考虑微波炉的制作，鸡蛋的采购等。

当做"制作微波炉"这项工作时，程序设计在具体的对象一级上，这时候，不用考虑鸡蛋如何做。

当微波炉做成之后，就可以进入到下一个抽象级。开始考虑炖鸡蛋的调制水平。这时候，不用考虑微波炉的制作，而直接可在微波炉上进行操作。

操作程序可像下面这样：打碎两个蛋，放点水和调料等，在微波炉中烧5分钟。这就是更高级抽象的描述，也是面向对象程序设计中主程序的描述。这样的描述既简单明了又完整，但这不是一个结构化程序的描述。

结构化程序是使微波炉的外壳和内部结构与鸡蛋、调料、水同处于一个程序环境中，其程序显示出层层的函数调用结构。控制从手到面板，从面板进入微波炉内部，在复杂的内部电路的逻辑中流动，最后发出"来取吧"的声音。在这个环境中，很难理解抽象及其程度。程序中没有对象，没有能隐藏事物固有复杂性的抽象。制作炖蛋，必须首先是制作微波炉的专家。等到下次制作红烧鱼时，又要重复制作一个专门烧鱼的微波炉了。

13.2 分类

层层分类，使概念逐渐细化，即具体化。相反，归类的结果，便是逐步抽象的过程。

例如，我问，什么是桑塔纳？一般的回答是：它是一种小轿车。如果我又问，什么是小轿车？一般的回答是，是一种小汽车。如果我再问，什么是小汽车？一般的回答是，是一种汽车，那么我又问，什么是汽车呢？汽车是一种交通工具，等等。

因为人们理解的桑塔纳是我们生活中称为小轿车一类东西的一个实例。也就是说，桑塔纳是一种特殊类型的小轿车，而小轿车是一种特殊的小汽车。事实上，桑塔纳还不是具体的实物，它只不过是一个名字，代表所有的桑塔纳牌小轿车。

在面向对象的计算机世界中，我们把一辆实实在在的桑塔纳小轿车称为是类(class)桑塔纳的一个实例(instance)或者说是对象(object)。类桑塔纳是类小轿车的一个子类，而类小轿车又是类小汽车的一个子类，类小汽车是类汽车的一个子类，类汽车又是类交通工具的子类，等等。

在我们生活的这个世界上，每个事物，每件东西都按规则分了类。做这项工作，是为了减少我们必须记住的东西个数。

例如，广告上说，某某牙膏具有特殊的止血功能。我们便知道，该产品是一种牙膏，具有牙膏的一切属性，外壳形状，颜色，膏状，清洁牙齿的功能，用在牙刷上等。除此之外，它还具有止血的特殊疗效。所以，牙膏是抽象的概念，它使我们理解一切具有牙膏属性的东西。任何牙膏的新产品，只要描述一下除了牙膏公共属性之外的特殊属性即可。

桑塔纳小轿车外形与其他小轿车不一样，内部的若干特性也与其他小轿车不一样，除此之外，桑塔纳小轿车便是一般意义上的小轿车了，它具有4个轮子，有方向盘，等等。

在结构化程序设计方法中没有分类的概念。因为，结构化程序设计强调的是过程的功

能划分，注重功能性。每一个功能，都靠自己解决。如果一个功能与另一个已存在的功能实现类似，也不能从中为我所用。

在面向对象的程序设计中，对象被分成类。类又是层层分解的，这些类与子类的关系可以被规格化地描述。描述了类，再描述其子类，就可以只描述其增加的部分。所有子类层次上的编程，都只需在已有的类的基础上进行。

分类是面向对象程序设计的需要。在这之前，在结构化程序设计中，存在一些问题：

(1) 在程序中，总是将制作汽车和开汽车混在一起。所以，要到达目的地，必须同时具有汽车制造技术和驾驶汽车的技术。这使得编程难度大大增加。开车是简单的，在现有某种型号汽车的基础上，制造新型的汽车也是简单的。只要分离汽车制造和开车，就能使程序编制简单化。为了分离汽车制造与开车，就要将汽车从过程中分离出来，成为一个独立的概念，并用分类技术使汽车的描述与实现简单化。只要说明与现有的某种汽车不同之处，即可完整地描述所使用的汽车。

(2) 在程序中，由于开车与汽车制造是交错实现的，所以要更换汽车很难。这使得程序不灵活。我们说，要到达目的地，不一定非得要一种型号的汽车，只要满足一定的装载量和速度等要求，其他的汽车完全可以代替。一旦有了汽车的分类，就有关于汽车的操作描述，根据该描述，完全可以从汽车的分类中，找到其他同样功能的汽车。

(3) 在程序中，如果改变了到达的目的地和运载量，将使程序推翻重来。这意味着要重新制造汽车。有了分类的概念，就可以在已存在的汽车分类(重用)中，轻易地定义出满足要求的汽车，而不必修改运载的过程。

分类是理解抽象的重要手段，也是面向对象程序设计中的重要概念。把握了分类方法，就能理解面向对象程序设计的过程。

13.3 设计和效率

有人说，面向对象程序设计，比之结构化程序设计，不能产生出高效的程序。

思考一下我们所使用的交通工具。试想一个体力很好的人(比做熟练的程序员)，骑着自行车，贯穿在大街小巷中，轻松自在地到达目的地。他走的路，是捷径。可是，如果目的地是较远的地方，例如是另一个城市，骑车去，还能那么轻松自在吗？也许他会选择坐汽车，甚至让其他驾驶员带他到目的地。这样，他就无须辨认任何路名路标，还可沿途欣赏田园风光。坐汽车比骑自行车更方便了。然而，有人会说，汽车比自行车要庞大，耗费更多的费用。

可以将高速公路比做面向对象程序设计方法，把街道马路比做结构化程序设计方法；将汽车比做面向对象程序中的对象，把开车去目的地比做面向对象程序设计；将骑自行车去目的地比做结构化的程序设计。从中看出，结构化的程序规模(自行车加骑车去目的地)比较小，但是整体过程比较复杂(分别对待道路性质不同的大街小巷)，执行的效率在大多数情况下是比较低的。即使目的地在同一城市内部，坐车也往往要比骑自行车省事。**程序规模小，并不一定效率高**。面向对象的程序从绝对的语句行数上，比结构化的程序可能要多，但它的程序结构更易理解，汽车是汽车，坐车是坐车。编译运行的效率即产生的机器代码规模和运行时间却更小和更快。坐车人人会坐，骑自行车不是人人会骑。坐车可以更换车型，只要能够载客。自行车相对更换的适应性要差一些，男同志的自行车女同志不一定能骑。

汽车可以载更多的客，自行车不行；汽车可以到达更远的目的地，自行车自叹不如。

汽车必须在大街或高速公路上开，小巷中不能开。划分出类，就成了面向对象程序设计。自行车既可在大街也可在小巷中骑，小程序也可采用面向对象程序设计方法。采用面向对象的程序设计方法才有高效运行的效果。汽车在高速公路上行驶比之自行车快很多。唯一"不足"的是，坐车用的车比自行车要耗费更大的成本。这也是很自然的，因为面向对象程序设计是以大量类库作为产品被生产出来为基础的。各种汽车就是大量供使用的产品，没有汽车，有了高速公路也白搭。汽车用来为大众所用(重用)。随着工业化程度的提高，人们感受到的是方便，人们也适应了鳞次栉比、纵横交错的交通网。人们会逐渐忘却结构化程序设计，而自然采用面向对象程序设计。

一些小程序，可以通过过程化的程序设计技巧和优化，小幅度提高运行速度，但往往以牺牲可读性为代价，给维护造成大量的困难。一旦程序规模扩大，程序的可读性和可维护性，甚至连结构化的程序设计都感到力不从心。

在现实生活中，能解决问题的小规模程序是很少的。所以，可以说，面向对象程序设计比结构化程序能够产生出更加有效的程序。而且，面向对象的程序，其可读性、可维护性都比结构化程序好。

→ 何为效率

我们说这个人办事效率高，是指他消耗的费用少，而且工作成绩大，即"投入少产出高"。

程序运行的效率是指占用尽可能少的资源而运行速度相对较快。这里的资源主要是指存储空间。

效率不能只看其运行速度的快慢，也不能只看其占据的存储空间，要综合地去比较时间和空间才能客观地评价程序运行的效率。

13.4 讨论 Josephus 问题

Josephus 问题，我们前面曾经讨论过，如果起点可以是任意一个小孩，那么在解决 Josephus 问题中，还要另给出起始点位置。

根据前面讨论的内容，可以得到一个处理过程的描述：

```
//Josephus 问题解答

建立小孩结构类型
初始化小孩数，开始位置，数小孩个数
分配小孩结构数组

for 初始化结构数组(构成环链)
  挂接下一个数组元素
  小孩编号
  输出编号
endfor

转到开始位置

while(小孩数多于一个)
  数小孩个数(一个循环)
```

```
    出列小孩
    将该小孩从环链中删除
  endwhile

  输出得胜者

  返还结构数组空间
```

该程序是一个过程化的实现。在前面章节给出的解答中，都是按这种思路实现的。相对来说，它要求更高的程序设计技巧。该问题的解决紧紧依赖于所定义的数据结构，程序员必须对 Josephus 问题的解决办法非常清楚，并且对实现其解答的数据结构操作也十分内行。编程是高度专业化的，我们在前面看到的解答就是技巧性相当高的两种实现。

但是，不能对所有的问题都用这样的直观分析法。当问题趋向复杂化时，就要将问题分割成一个个小块，从而最终解决问题。

结构化程序设计方法按功能分割问题。面向对象程序设计按对象分割问题。

13.5 结构化方法

从上节描述的程序流程来看，是比较清楚的。但是我们仍认为它太复杂，而对它进行功能分解，也即对下面各项实现一个功能子块：

(1) 初始化小孩数，开始位置，数小孩数；

(2) 初始化环链表(采用链表数据结构来解)；

(3) 数小孩；

(4) 处理未获胜的小孩。分解之后，主程序描述变得短小了：

```
//Josephus 问题解答

建立结构
初始化小孩数，开始位置，数小孩个数
初始化环链表(采用链表数据结构来解)
转到开始位置(一个循环)

处理未获胜的小孩

输出得胜者
返还结构数组空间
```

其中，初始化小孩数，开始位置，数小孩个数描述为：

```
键入小孩数、开始位置、数小孩个数
小孩数校验
开始位置校验
数小孩个数校验
```

初始化环链表描述为：

```
分配结构数组

for 初始化结构数组(构成环链)
```

```
    挂接下一个数组元素
    小孩编号赋值
    输出小孩编号
  endfor

  返回环链表
```

处理未获胜的小孩描述为：

```
while(小孩数多于一个)
  数小孩(一个循环)
  出列小孩
  将该小孩从环链中删除
endwhile
```

数小孩描述为：

```
for(从1到数小孩间隔数)
  开始位置挪到下一个小孩
endfor
```

主程序描述中，只涉及一个个功能调用，没有循环，比之前面将整个问题放在一起解决要简单和容易理解一些。每个功能调用相对一个大程序来说，也简单一些，这样便容易把握。这对程序设计的可读性来说，进了一大步，其他程序员很容易理解此程序，并且乐意对其进行维护。

主程序中省略了细节，而在真正实现时，再进一步具体化。这正是更为抽象的程序设计（面向对象程序设计）之技术与基础。

但从整个问题解决来说，它又显得复杂，用户只关心给出小孩数、开始位置和每隔多少小孩出列一个小孩的间隔以及需要得到获胜者的位置。除此之外，任何内部实现的细节，都是多余的。如果把这些内部实现的细节分离出来，以让更专业的人员来实现，岂不更符合社会化大生产的要求吗？

然而，结构化程序做不到，只能由程序员来设计解答整个过程。可以将其中的一部分作为函数调用分给别人实现，但他自己的专业化程度甚至更高。即他必须全盘把握实现中的数据结构（本例中的小孩结构和链表），并在频繁调用其他函数的过程中，为维护这些数据结构和数据操碎了心，且“吃力不讨好”，由于数据结构和数据对所有函数都可见，很难把握数据的修改出自哪个函数，这些数据的安全性得不到保障。难道软件开发真的只能由计算机专家来完成吗？

计算机发展到今天，有许多软件产品都是现成的，它们以类库的形式提供，只要拿来用就是了。

一个电视机电路设计专家，其家里摆放的电视机与别人从市场上买到的一样，能够欣赏到的节目与别人也一样。使用电视机真的无须电视机设计技术。

Josephus 问题的结构化分析方法，是将该问题按功能分解，然后按必要的顺序实现之。功能的划分并不严格，可按复杂程度分，对有些简单的功能不必划分出去，如输出小孩编号、小孩脱离链表等。

13.6 结构化方法的实现

现在，我们不得不面对这样的结构化程序，这个程序的组织，是比较紧凑和高效的，但是，程序员除了懂 Josephus 算法之外，对于数据和链表的维护，用尽了程序技巧，这在面向对象程序设计看来，根本不屑一顾：

```
//* * * * * * * * * * * * * * * * * * * *
//* *      Josephus 问题解法三          * *
//* *      jose3.cpp                    * *
//* * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

struct Jose          //小孩结构
{
  int code;          //存放小孩编号
  Jose * next;       //用于指向下一个小孩结点
};

//全局变量
int n;               //小孩数
int begin;           //开始位置
int m;               //数小孩间隔
Jose * pivot;        //链表哨兵
Jose * pCur;         //当前结点指针

//函数声明
int assign();                              //赋初值，返回 1：成功，0：失败
void initial(Jose * pBoys);                //初始化环链表
void count(int m);                         //数 m 个小孩
void process();                            //处理所有未获胜小孩

//主函数
int main()
{
  if(!assign())
  {
    cout <<"The program failed.\n";
    return;
  }

  Jose * pJose = new Jose[n];              //分配结构数组
  initial(pJose);                          //初始化结构数组
  count(begin);                            //转到开始位置
  process();                               //处理所有未获胜小孩

  cout <<"\nthe winner is " <<pCur->code <<endl;

  delete[]pJose;                           //返还结构数组给堆空间
}
```

```
//赋初值
int assign()
{
  int number,start,count;
  cout <<"please input the number,begin,count:\n";
  cin>>number >>start >>count;

  if(number<2)                              //小孩数校验
  {
    cerr <<"bad number of boys\n";
    return 0;
  }

  if(start<0)                               //开始位置校验
  {
    cerr <<"bad begin position.\n";
    return 0;
  }

  if(count<1 || count >number)              //数小孩顺序个数校验
  {
    cerr <<"bad interval number.\n";
    return 0;
  }

  n = number; begin = start - 1; m = count; //赋全局变量值
  return 1;
}

//链表初始化
void initial(Jose * pJose)
{
  int lineCount = 0;
  Jose * px = pJose;

  for(int i = 1;i<= n;i ++ )
  {
    px - >next = pJose + i % n;
    px - >code = i;
    px = px - >next;

    if((lineCount ++ % 10) == 0)            //输出行中个数控制
      cout <<endl;
    cout <<setw(4) <<i;
  }
  cout <<endl;
  pCur = pJose + n - 1;          //指向结构数组最后一个元素
}

//数 m 个小孩
void count(int m)
{
  for(int i = 0; i<m; i ++ )
  {
```

```
        pivot = pCur;
        pCur = pivot->next;
    }
}

//处理获胜者决出之前的所有小孩
void process()
{
    int l = 0;
    for(int i = 1; i<n-1; i++)
    {
        count(m);       //数 m 个小孩

        if((l++ %10) == 0)          //输出行中个数控制
            cout <<endl;
        cout <<setw(4) <<pCur->code;

        pivot->next = pCur->next; //小孩脱链
        pCur = pivot;
    }
}
```

上例程序从规模(语句行数)上看,超过了过程化的程序;从程序的模块结构看,可读性有一些提高。但是,几乎每个函数都要用小孩数,不止一个函数要用开始位置和数小孩个数,所以不得不设置全局变量。这些全局变量使得函数的独立性大为降低,函数本身具有的黑盒特性遭到破坏。函数调试过程中数据相互依赖,这使程序开发和维护的难度大大提高。程序的可读性也大打折扣。

主程序的实现体现了系统设计思想,但实现过程却使程序员无法摆脱数据结构的细节。从数据变量的命名、类型,到链表结构的指针、结构数组的堆分配、返还等,无不要求程序员具有高度的专业素质,他们必须懂得业务和计算机专业两方面的知识。

13.7 面向对象方法

在面向对象的分析和设计中,我们执行下面的步骤:

(1) 找出类;

(2) 描述类和类之间的联系;

(3) 用类来定义程序结构。

找出类主要靠经验,程序员可由一系列候选类开始,然后考虑哪一个是最基本的以及哪一个是第二位的或者是被引出的。候选类由以下各项可找出:

(1) 有形的、可视的或描述的东西,像电视机、微波炉、桌子、问题等;

(2) 角色,如操作电视机的人、桌子上摆放的东西、问题中涉及的链表结构等;

(3) 事件,如操作电视机的亮度、桌子的移动、问题中描述的操作等。

对于很复杂的程序,程序员必须做一个完全的分析,并充分了解问题的各项细节,然后分类问题:哪些跟电视机有关,哪些跟桌子有关……抽象出描述的对象。

对于简单的问题,通过问题陈述和列出名词表,可以帮助解决问题。例如一个解决

Josephus 问题的名词表：

```
Josephue 问题
小孩
链表
开始位置
小孩数
每数若干个小孩的间隔
去掉一个小孩
描述胜利者
等等
```

其中，“开始位置”，“去掉一个小孩”，“每数若干个小孩”都与链表有关。链表还包括分配一个结构数组，初始化结构数组、环链等。这些名字有些是类，有些是组成类的属性，有些是描述类的行为。

作为一个要处理的问题 Josephus，我们可以把它看作是一个类。另外链表包括其数据属性（结构数组，当前位置等）和链表操作（移动小孩位置，去掉小孩等），所以可以把它看作是个类。

这时，要把类中的要用到的数据属性（数据成员）和操作（成员函数）描述清楚。

一般来说，**有一个数据属性，就有该数据属性的访问操作**。每个数据成员和成员函数的取名也是一个环节，应该使用易懂的组合单词。没有用的名字应删掉。

例如，图 13-1 描述了 Josephus 问题类和环链表类的卡片。

Josephus 类	
Initial	Boynumber
Getwinner	BeginPos
	Interval

Ring（环链表）类	
Clear	First
Print	Pivot
Count	Current

图 13-1　类的卡片描述

图中的卡片左面是成员函数描述，表示类的外部接口，右面是数据成员描述。有了这些类的描述，就可以设计程序了：

```
//主函数
定义一个 Jose 类对象
赋初值 initial
求获胜者 getwinner
```

从中我们感受到这种编程并不需要涉及 Josephus 问题和链表结构的数据的复杂细节。由于 Jose 类包含了与环链表类的通信联络，所以在主函数中丝毫不需要涉及环链表类，而且，Jose 类中的具体实现也并未涉及，只要提供该类的实现，那么编程就可以这么轻松和简单。**面向对象程序设计使用户既不需要懂计算机太多，也不需要懂业务太多。**

下面我们来看 Jose 类的实现者是如何来做这项工作的：

```
Jose 类
```

```
{
  接口(成员函数)
    构造函数
    赋初值 initial
    求获胜者
  内部数据成员
    小孩数
    开始位置
    数小孩个数
}
```

其中,作为接口的成员函数描述为:

```
赋初值 initial(成员函数)
{
  键入小孩数,校验
  键入开始位置,校验
  键入数数间隔,校验
}

求获胜者(成员函数)
{
  初始化环链表

  转到开始位置
  while(小孩数多于一个)
    数 m 个小孩(一个循环)
    出列小孩
    将该小孩从环链中删除
  endwhile

  返回得胜者
}
```

描述类总是标准的,既有表示类的操作(成员函数),也有表示类的属性(数据成员)。一个解决 Josephus 问题的实现者,完全可以将该类商品化。因为它是标准的类描述,所以,用户拿来,按照其接口,立即可以用之来解决实际数据的 Josephus 问题。

在 Jose 类中,要用到环链表类,所以 Jose 类要包含环链表类的描述:

```
ring(环链表)类
{
  接口(成员函数)
    构造函数
    根据数数间隔数小孩
    输出当前位置的小孩
    从环链中去掉当前小孩
    析构函数
  返还小孩结构数组的内存
  内部数据成员
    小孩结构数组指针
    当前小孩指针
    小孩哨兵指针
}
```

其中，作为接口的成员函数描述为：

```
构造函数(成员函数)
{
  分配小孩结构数组

  初始化结构数组
    小孩编号
    输出编号
    构成环链表

  当前小孩指针位置
}
```

实现Jose类的程序员，也可以将环链表类拿来为自己所用。C++中，提供了标准的链表类库，只要稍加继承，就能派生为环链表类。在20.7节，我们有使用标准模板类库解决Josephus问题的解答。

在面向对象程序设计中，我们看到，程序设计主要是描述类，大大小小的类都是标准的描述结构。这使得程序结构一般化，程序的可读性大为改观。而真正的程序主体，在这里就是Josephus问题的解答，只有短短的3条语句。事实上，它只是构造了类对象，启动(调用成员函数)了一下其中的一个类的行为。完全体现了：

```
程序 = 对象 + 对象 + ……
```

因而，面向对象程序设计，归结为类的设计。只要将问题中的对象层次划分清楚，C++就能将它用语言描述出来。描述了类结构(类声明与类定义)，余下的问题就是简单地定义对象和让对象表现自己，问题的解答也就差不多有了。这就是面向对象程序设计的基本方法。

面向对象程序设计可以将程序员分成两类：

一类是面向对象应用程序设计。所谓面向对象程序设计，多是指此。他(她)们无须了解类的实现细节，就像使用微波炉那样，这要比结构化程序设计简单得多。

另一类是类库设计。他(她)们为面向对象程序设计提供"素材"。这些素材涉及各个领域，由各领域的专业人员来设计完成。他(她)们需要了解特定类的知识，如Josephus问题的解答。

由于知识以类为单位，类是规范格式的，所以每个类的描述也很轻松，并使程序设计的分工合作更易于进行。因此，只要将问题归结为对象以及对象的联系，便可得到问题的解答。

这样的程序可读性是良好的，可维护性也是良好的，因为程序的结构更加规范化，它以抽象层来划分问题的解，而且问题的描述几乎就是程序的实现。

13.8 面向对象方法的实现

下面的程序是上面描述的面向对象程序设计方法的具体实现。该程序是个多文件结构，工程文件中包括3个源文件：

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
//- -              Josephus 问题解法四                    - -
//- -   jose4.prj                                        - -
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - -

ring.cpp
jose.cpp
jose4.cpp
```

源文件和在源文件中包含的头文件分别为：

```
//* * * * * * * * * * * * * * * * * * *
//* *            ring.h                * *
//* * * * * * * * * * * * * * * * * * *

struct Boy    //小孩结构作为链表结点
{
  int code;
  Boy * next;
};

class Ring    //环链表类定义
{
  public:
    Ring(int n);
    void Count(int m);      //数 m 个小孩
    void PutBoy();          //输出当前小孩的编号
    void ClearBoy();        //将当前小孩从链表中脱钩
  ~Ring();
  protected:
    Boy * pBegin;
    Boy * pivot;
    Boy * pCurrent;
};

//* * * * * * * * * * * * * * * * * * * * *
//* *              ring.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>
#include "ring.h"

Ring::Ring(int n)
{
  pBegin = new Boy[n];      //分配小孩结构数组
  pCurrent = pBegin;

  for(int i = 1; i<= n; i ++ ,pCurrent = pCurrent - >next)
  {
    pCurrent - >next = pBegin + i % n;      //将结点链起来
    pCurrent - >code = i;      //小孩编号
    PutBoy();
  }
  cout << endl;
```

```
  pCurrent = &pBegin[n - 1];        //当前小孩位置在最后一个编号
}

void Ring::Count(int m)
{
  for(int i = 0; i<m; i ++)
  {
    pivot = pCurrent;
    pCurrent = pivot - >next;
  }
}

void Ring::PutBoy()
{
  static int numInLine;
  if(numInLine ++ % 10 == 0)
    cout <<endl;

  cout <<setw(4) <<pCurrent - >code;
}

void Ring::ClearBoy()
{
  pivot - >next = pCurrent - >next;
  pCurrent = pivot;
}

Ring::~Ring()
{
  delete[ ]pBegin;
}

//* * * * * * * * * * * * * * * * * * * *
//* *            josex.h              * *
//* * * * * * * * * * * * * * * * * * * *

class Jose
{
  public:
    Jose(int boys = 10, int begin = 1, int m = 3)
    {
      numOfBoys = boys;
      beginPos = begin;
      interval = m;
    }

    void Initial();
    void GetWinner();
  protected:
    int numOfBoys;
    int beginPos;
    int interval;
};
```

```
//* * * * * * * * * * * * * * * * * * * * *
//* *            josex.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "ringx.h"      //告诉编译,本文件中将使用 Ring 类
#include "josex.h"

void Jose::Initial()
{
  int num,begin,m;
  cout <<"please input the number of boys,"
       "beginning position,interval per count :\n";
  cin >>num >>begin >>m;

  if(num<2)
  {
    cerr<<"bad number of boys\n";
    return;
  }

  if(begin<0)
  {
    cerr<<"bad beginning position.\n";
    return;
  }

  if(m<1||m>num)
  {
    cerr<<"bad interval number.\n";
    return;
  }

  //输入数据都合法时,予以赋值
  numOfBoys = num;
  beginPos = begin;
  interval = m;
}

void Jose::GetWinner()
{
  Ring x(numOfBoys);       //小孩围成圈
  x.Count(beginPos);       //转到开始位置

  for(int i = 1; i<numOfBoys; i ++ )      //处理除了获胜者之外的所有小孩
  {
    x.Count(interval);      //数小孩
    x.PutBoy();             //输出小孩编号
    x.ClearBoy();           //当前小孩脱离环链
  }

  cout <<"\nthe winner is ";
  x.PutBoy();      //获胜者
}
```

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                jose4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *
#include "jose.h"          //告诉编译,本文件中将使用 Jose 类

int main()
{
  Jose jose;       //建立 Josephus 问题对象
  jose.Initial();
  jose.GetWinner();
}
```

图 13-2 表示了该程序中文件之间的关系。

图 13-2　程序中源文件的相互关系

图中,主函数要用到 Jose 类,Jose 类要用到 Ring 类。

类 Ring 定义了环链表的所有操作与数据。在头文件 ring.h 中,定义了类的操作界面,因为要用到小孩结构 Boy 的名字,所以在该处定义了 Boy 结构。而在 Ring 类的成员函数定义文件 ring.cpp 中,无须再次定义 Boy 结构,因为该文件包含了 ring.h 头文件。

如果哪个程序要用 ring 类,只要包含 ring.h 头文件即可,无须涉及 ring 类的实现细节(即成员函数的实现代码)。ring.h 头文件主要告诉用户其成员函数的功能和调用方法(函数原型)。在这里是:

```
void Ring::Count(int m);      //数 m 个小孩
void Ring::PutBoy();          //输出当前小孩编号
void Ring::ClearBoy();        //将当前小孩从链表中脱钩
```

另外还包含构造函数和析构函数。

至于保护数据,它们不是面向对象程序设计中类的界面。也就是说,用户使用(重用)类时,无须类中保护数据和私有数据,因为这些数据成员都由成员函数共享使用,而不能直接由用户去访问。但它们安排在类定义中,是类机制实现的需要。所以,使用类时,我们包含类的头文件,只使用其成员函数,而无须关心保护和私有数据。

所有的类,都是以类定义(类的头文件)作为类的界面,这样的一律性,使得程序结构有条不紊地组织和划分,程序可读性和可维护性大大增强。

同理,Jose 类也是这样组织的。Jose 类中使用了 Ring 类,只要包含 ring.h 头文件即可。Jose 类的头文件 jose.h 定义了类 Jose 的界面,头文件中尚无涉及 Ring 类,所以无须包含 ring.h。在 Jose 类的成员函数定义(jose.cpp)中,需要用到 Ring 类对象,所以该文件包含了 ring.h 头文件。同时也包含了 Jose 类自己的头文件(jose.h)。

成员函数 Initial()中，如果校验失败就显示错误信息，并不做其他任何事返回，以使主函数(main)求解默认的 Jose 数据成员设置的问题。

我们看到成员函数 GetWinner()，是求解 Josephus 问题的主算法。但由于使用了类，其算法十分简单。编程成了抽象描写过程的"艺术"，只要遵循面向对象的程序设计，那么大程序的开发再也不用让主设计者一个人处于必须了解所有系统的方方面面的地位，而他完全可以在划分了类之后，抽象且艺术化地编程。

在主程序中，main()函数只包括了 3 条语句，我们再次领略了面向对象程序设计的抽象性。抽象比具体容易记忆，抽象使问题能够简单表达，抽象意味着当前无须深究。

13.9 程序维护

作为 Josephus 问题的一个扩展，可以要求有若干个获胜者。这时，便涉及程序维护问题。求一个和几个获胜者，都是求获胜者，所以对面向对象的应用程序来说，最多传递一个获胜者个数参数给 Jose 类对象的求获胜者成员函数。

我们将 Josephus 问题(类)的实例看作是具备任何求解条件的对象，这样，面向对象的应用程序仍然为建立 Jose 类对象，让其具备求解条件，求获胜者。这时程序代码不必作任何改动。

在 Jose 类中，保持外部接口不变。这是面向对象应用程序不必改动的根本原因。但在具体实现上，由于要求几个获胜者，所以要适当修改类库代码。

(1) 在 Jose 类中增加一个获胜者个数的数据成员。

(2) 在具备求解条件的 Initial()成员函数中，需要输入获胜者个数，并落实对其的校验。

(3) 在求获胜者 GetWinner()成员函数中，要修改处理小孩的循环次数。

(4) 输出获胜者的过程要扩充，并不局限于打印一个获胜者，而是要将现存小孩圈的所有小孩都作为获胜小孩打印。由此，原有的 Ring 类中的 PutBoy()成员函数要作修改。

(5) 为此，Ring 类中增加 Display()成员函数，它负责将小孩圈中所有小孩都打印一遍。

(6) 由于 Ring 类的外部接口发生变化，调整 Jose 类的内部实现，即修改 GetWinner()成员函数，打印所有获胜者。

(7) Ring 类自身发生变化，调整自身，使之更趋合理。此处即修改其构造函数。

于是，对上面的程序，作适当的修改如下，成为解决 Josephus 问题的第五个解答。该工程文件为：

```
//------------------------------------
//--    Josephus 问题解法五            --
//--    jose5.prj                    --
//------------------------------------

ringx.cpp
josex.cpp
jose5.cpp        //与 jose4.cpp 相同
```

源文件和在源文件中包含的头文件分别为：

```
//* * * * * * * * * * * * * * * * * * * *
//* *            ringx.h              * *
//* * * * * * * * * * * * * * * * * * * *

struct Boy
{
  int code;
  Boy * next;
};

class Ring      //环链表类定义
{
  public:
    Ring(int n);
    void Count(int m);      //数 m 个小孩
    void PutBoy();          //输出当前小孩的编号
    void ClearBoy();        //将当前小孩从链表中脱钩
    void Display();         //打印圈中所有小孩
    ~Ring();
  protected:
    Boy * pBegin;
    Boy * pivot;
    Boy * pCurrent;
};

//* * * * * * * * * * * * * * * * * * * *
//* *            ringx.cpp            * *
//* * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>
#include "ringx.h"

Ring::Ring(int n)
{
  pBegin = new Boy[n];
  pCurrent = pBegin;

  for(int i = 1; i<=n; i++,pCurrent = pCurrent->next)
  {
    pCurrent->next = pBegin + i%n;
    pCurrent->code = i;
  }                 //循环结束时,pCurrent 指向第一个小孩

  Display();          //打印初始时所有小孩

  pCurrent = &pBegin[n-1];
}

void Ring::Count(int m)
{
  for(int i = 0; i<m; i++)
```

```
    {
      pivot = pCurrent;
      pCurrent = pivot->next;
    }
}

void Ring::PutBoy()
{
  static int numInLine;
  if(numInLine ++ % 10 == 0)
    cout <<endl;

  cout <<setw(4) <<pCurrent->code;
}

void Ring::ClearBoy()
{
  pivot->next = pCurrent->next;
  pCurrent = pivot;
}

void Ring::Display()
{
  Boy * pB = pCurrent;

  do{
    PutBoy();
    pCurrent = pCurrent->next;
  }while(pB!= pCurrent);

  cout <<endl;
}

Ring::~Ring()
{
  delete[ ]pBegin;
}

//* * * * * * * * * * * * * * * * * * * *
//* *            josex.h              * *
//* * * * * * * * * * * * * * * * * * * *

class Jose
{
  public:
    Jose(int boys = 10, int begin = 1, int m = 3)
    {
      numOfBoys = boys;
      beginPos = begin;
      interval = m;
    }

    void Initial();
    void GetWinner();
```

```
  protected:
    int numOfBoys;
    int beginPos;
    int interval;
    int wins;
};

//* * * * * * * * * * * * * * * * * * * * *
//* *              josex.cpp              * *
//* * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "ringx.h"
#include "josex.h"

void Jose::Initial()
{
  int num,begin,m,w;
  cout <<"please input the number of boys,\n"
       "beginning position,\ninterval per count,\n"
       "number of winners :\n";
  cin >>num >>begin >>m >>w;

  if(num<2)
  {
    cerr<<"bad number of boys\n";
    return;
  }

  if(begin<0)
  {
    cerr<<"bad beginning position.\n";
    return;
  }

  if(m<1||m>num)
  {
    cerr<<"bad interval number.\n";
    return;
  }

  if(w<1 || w>=num)
  {
    cerr<<"bad number of winners.\n";
    return;
  }

  //输入数据都合法时,予以赋值
  numOfBoys=num;
  beginPos=begin;
  interval=m;
  wins=w;
}
```

```
void Jose::GetWinner()
{
  Ring x(numOfBoys);
  x.Count(beginPos);

  for(int i = 1; i<numOfBoys - wins + 1; i ++ )
  {
    x.Count(interval);
    x.PutBoy();
    x.ClearBoy();
  }

  cout <<"\nthe winner(s) is ";
  x.PutBoy();
  x.Display();
}

//* * * * * * * * * * * * * * * * * * * * * * *
//* *              jose5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include "josex.h"

void main()
{
  Jose jose;
  jose.Initial();
  jose.GetWinner();
}
```

运行结果为：

```
please input the number of boys,
begin position,
interval per count,
number of winners:
12  1  3  3
1   2   3   4   5   6   7   8   9   10
11    12
4    7   10   1   5   9   2   8   3
the winner(s) is 12 6 11
```

程序中，用黑体的语句是新添部分。

小结

结构化程序设计强调程序的功能，它以函数（一个功能单位）为中心，分层逐步展开程序设计。但是功能的大小界定，没有统一的标准，不同程序员的设计，反映了不同的思维、程序组织和风格，这给理解程序带来了阴影。同时，它不能隐藏数据复杂性，强迫主程序员必须懂业务、懂计算机以及具有深层次的知识背景。

面向对象程序设计强调程序的分层分类概念，它以抽象为基础，轻灵地描述问题解决的

大体思想，以此为基础，进行对象的定义与对象的展示，亦即程序设计，即高层次的抽象程序设计。与此相对应，进行“描述问题解决的大体思想”的具体实现，即类的内部实现（成员函数定义）。

类定义是面向对象程序设计中的基础问题，对象定义是面向对象程序设计的一般操作。定义类的过程要求了解问题中的一些东西，组织思想和弄清本质。

类是很容易想象的，如果选择了一个错误的类，则描述起来很困难；如果是正确的类，则将很容易理解它，并清楚地描述出其成员函数和数据成员。

面向对象程序设计的关键是如何抽象与分类。在开发大型软件的时候，要用到面向对象的分析设计技术，它不在本书讨论的范围。

Josephus 问题我们再次提了出来。它在这里用来说明面向对象程序设计和结构化程序设计各自的设计方法，以及其本质的区别。

面向对象程序的维护使我们领略维护并不像结构化程序设计那样大面积展开，而是“各司其职”。

练习

13.1 写出从物质到人类的中间层次。如：物质分有机物与无机物，有机物分生物与非生物，生物分动物与植物，等等。

13.2 描述课程类和学生类。用重用类的多文件程序结构形式，编制面向对象应用程序。学生有名字，学生最多可学五门课程，学生实际学的门数，可以给定学生的名字，可以得到学生的名字，可以得到学生给定课程的成绩，可以得到学生所学课程的平均成绩，可以给学生增加一门课（同时在该课程中增加一个学生）。

课程最多有 30 个学生，课程有实际学生数，课程有实际学生名单，课程有学分数，课程有每个学生成绩，课程可以得到学分数，课程可以设置学分数，课程可以得到班平均成绩，课程可以得到某个学生成绩。

现有数学课，张三学数学，成绩为 3.1 分，李四学数学，成绩为 4.5 分。求其平均成绩，求张三的数学成绩。

现有物理课，学时数为 4，张三学物理，成绩为 4 分。求张三所学课程的平均成绩。

第 14 章　堆与拷贝构造函数

在 C++中，堆分配的概念得到了扩展，不仅 C++的关键字 new 和 delete 可以分配和释放堆空间，而且通过 new 建立的对象要调用构造函数，通过 delete 删除对象也要调用析构函数。另外，当对象被传递给函数或者对象从函数返回的时候，会发生对象的拷贝。但有些情况，一模一样的拷贝并不是所希望的，这就要借助于定义拷贝构造函数了。学习本章后，应该掌握 new 和 delete 这两个操作符的使用，并能把握从堆中分配和释放对象以及对象数组的时机；领会拷贝构造函数的实质，区别浅拷贝和深拷贝，在程序中适当地运用拷贝构造函数。

14.1　关于堆

C++程序的内存格局通常分为四个区：

(1) 全局数据区(data area)；

(2) 代码区(code area)；

(3) 栈区(stack area)；

(4) 堆区(即自由存储区)(heap area)。

全局变量、静态数据、常量存放在全局数据区，所有类成员函数和非成员函数代码存放在代码区，为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区，余下的空间都被作为堆区。

函数"void * malloc(size_t);"和"void free(void *);"在头文件 malloc.h 中声明，而操作符 new 和 delete 是 C++语言的一部分，无须包含头文件。它们都从堆中分配和释放内存块，但在具体操作上两者有很大的区别。

操作堆内存时，如果分配了内存，就有责任回收它，否则运行的程序将会造成内存泄漏。这与函数中在栈区分配局部变量有本质的不同。

对 C++来说，管理堆区是一件十分复杂的工作，频繁地分配和释放不同大小的堆空间，将会产生堆内碎块。

14.2　需要 new 和 delete 的原因

从 C++的立场上看，不能用 malloc()函数的一个原因是，它在分配空间的时候不能调用构造函数。类对象的建立是分配空间、构造结构以及初始化的三位一体，它们统一由构造函数来完成。

例如，下面的代码用 malloc()分配对象空间：

```
class Tdate
{
```

```
  public:
    Tdate();
    SetDate(int m = 1, int d = 1, int y = 1998);
  protected:
    int month;
    int day;
    int year;
};

Tdate::Tdate()
{
  month = 1;
  day = 1;
  year = 1;
}

void Tdate::SetDate(int m, int d, int y)
{
  if(m>0 && m<13)
    month = m;
  if(d>1 && d<32)
    day = d;
  if(y>0 && y<3000)
    year = y;
}

void fn()
{
  Tdate * pD;      //仅仅是个指针,没有产生对象
  pD = (Tdate * )malloc(sizeof Tdate); //并不调用构造函数
  //...
  free(pD);      //并不调用析构函数
}
```

指针 pD 的声明不为 Tdate 调用其构造函数,因为 pD 没有指向任何东西。假如构造函数要被调用,则必须在进行内存分配的 malloc()调用时进行。然而 malloc()仅仅只是一个函数调用,它没有足够的信息来调用一个构造函数,它所接受的参数是一个 unsigned long 类型。

pD 从 malloc()那儿获得的不过是一个含有随机数据的类对象空间而已,对应的对象空间中的值不确定。为此,须在内存分配之后再进行初始化。

例如,下面的代码描述用 malloc()来进行对象的创建过程:

```
void fn()
{
  Tdate * pD;
  pD = (Tdate * )malloc(sizeof Tdate);
  pD->SetDate();        //设置 Tdate 值
  //...
  free(pD);
}
```

这从根本上说，不是一个类对象的创建，因为它绕过了构造函数。

另外，从程序设计的需要来看，在分配内存申请的时候，总是知道分配的空间派什么用，而且分配空间大小总是某个数据类型(包括类类型)的整数倍。因而 C++用 new 代替 C 的 malloc()是必然的。

14.3 分配堆对象

C++的 new 和 delete 机制更简单易懂。例如，下面的代码可与前面的代码做一比较：

```
void fn()
{
  Tdate * pS;
  pS = new Tdate;      //分配堆空间并构造它
  //...
  delete pS;      //先析构，然后将空间返还给堆
}
```

不必显式指出从 new 返回的指针类型，因为 new 知道要分配对象的类型是 Tdate。而且 new 还必须知道对象的类型，因为它要藉此调用构造函数。

如果是分配局部对象，则在该局部对象退出作用域时(要么程序执行遇到函数结束标记"}"，要么遇到返回语句)自动调用析构函数。但是堆对象的作用域是整个程序生命期，所以除非程序运行完毕，否则堆对象作用域不会到期。堆对象析构是在释放堆对象语句 delete 执行之时。上面的堆对象在执行 delete pS;语句时，C++自动调用其析构函数。

构造函数可以有参数，所以跟在 new 后面的类类型也可以跟参数。

例如下面的代码，new 后面的类型必须跟参数：

```
class Tdate
{
  public:
    Tdate(int m,int d,int y);
  protected:
    int month;
    int day;
    int year;
};
Tdate::Tdate()
{
  if(m>0&&m<13)
    month = m;
  if(d>1&&d<32)
    day = d;
  if(y>0&&y<3000)
    year = y;
}

void fn()
{
  Tdate * pD;
```

```
    pD = new Tdate(1,1,1998);
    //...
    delete(pD);
}
```

"pD=new Tdate(1,1,1998);"这一语句,使 new 去调用了构造函数 Tdate(int, int, int),new 是根据参数匹配的原则来调用构造函数的。如果上一句写成:

```
pD = new Tdate;
```

则由于 Tdate 类没有默认构造函数(已被 Tdate(int,int,int)覆盖)而使该语句报错。从堆中还可以分配对象数组。

例如,下面的代码分配了参数给定的对象个数,并在函数结束时,予以返还:

```
class Student
{
  public:
    Student(char * pName = "no name")
    {
      strcpy(name,pName,sizeof(name));
      name[size of(name) - 1] = "\0";
    }
  protected:
    char name[40];
};

void fn(int noOfObjects)
{
  Student * pS = new Student[noOfObjects];
  //...
  delete[]pS;
}
```

分配过程将激发 noOfObjects 次构造函数的调用,从 0 至 noOfObjects－1。调用构造函数的顺序依次为 pS[0],pS[1],pS[2],… pS[noOfObjects－1]。由于分配数组时,new 的格式是类型后面跟[元素个数],不能再跟构造函数参数,所以,**从堆上分配对象数组,只能调用默认的构造函数,不能调用其他任何构造函数**。如果该类没有默认构造函数,则不能分配对象数组。

delete[]pS 中的[]是要告诉 C++,该指针指向的是一个数组。如果在[]中填上了数组的长度信息,C++编译系统将忽略,并把它作为[]对待。但如果忘了写[],则程序将会产生运行错误。

一般来说,堆空间相对其他内存空间比较空闲,随要随拿,给程序运行带来了较大的自由度。使用堆空间往往由于:

(1) 直到运行时才能知道需要多少对象空间;

(2) 不知道对象的生存期到底有多长;

(3) 直到运行时才知道一个对象需要多少内存空间。

14.4 拷贝构造函数

可用一个对象去构造另一个对象，或者说，用另一个对象值初始化一个新构造的对象，例如：

```
Student s1("Jenny");
Student s2 = s1; //用 s1 的值去初始化 s2
```

对象作为函数参数传递时，也要涉及对象的拷贝，例如：

```
void fn(Student fs)
{
  //...
}

int main()
{
  Student ms;
  fn(ms);
}
```

函数 fn()的参数传递的方式是传值，参数类型是 Student，调用时，实参 ms 传给了形参 fs，ms 在传递的过程中是不会改变的，形参 fs 是 ms 的一个拷贝。这一切是在调用的开始完成的，也就是说，形参 fs 用 ms 的值进行构造。

这时候，调用构造函数 Student(char *)就不合适，新的构造函数的参数应是 Student&，也就是：

```
Student(Student& s);
```

为什么 C++要用上面的拷贝构造函数，而它自己不会做像下面的事呢？即：

```
int a = 5;
int b = a;      //用 a 的值拷贝给新创建的 b
```

因为对象的类型多种多样，不像基本数据类型这么简单，有些对象还申请了系统资源，如图 14-1 所示，s 对象拥有了一个资源，用 s 的值创建一个 t 对象，如果仅仅只是二进制内存空间上的 s 拷贝，那意味着 t 也拥有这个资源了。由于资源归属权不清，将引起资源管理的混乱。在 14.6 节中还要对这个问题展开讨论。

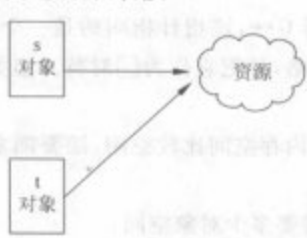


图 14-1　T 对象创建时拷贝 S 对象

下面的程序介绍了拷贝构造函数的用法：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch14_1.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName = "no name", int ssId = 0)
    {
      id = ssId;
      strcpy(name, pName);
      cout <<"Constructing new student " <<pName <<endl;
    }
    Student(Student& s)      //拷贝构造函数
    {
      cout <<"Constructing copy of " <<s.name <<endl;
      strcpy(name, "copy of ");
      strcat(name, s.name);
      id = s.id;
    }

  ~Student()
    {
      cout <<"Destructing " <<name <<endl;
    }
  protected:
    char name[40];
    int id;
};

void fn(Student s)
{
  cout <<"In function fn()\n";
}

int main()
{
  Student randy("Randy", 1234);
  cout <<"Calling fn()\n";
  fn(randy);
  cout <<"Returned from fn()\n";
}
```

运行结果为：

```
Constructing new student Randy
Calling fn()
Constructing copy of Randy
In function fn()
Destructing copy of Randy
```

```
Returned from fn()
Destructing Randy
```

randy 对象的创建调用了普通的构造函数，产生了第一行信息；随之便输出第二行信息；main()调用 fn(randy)时，发生了从实参 randy 到形参 s 的拷贝构造，于是调用拷贝构造函数而得到第三行信息；随之就进入到 fn()的函数体中，产生了第四行信息；从 fn()返回时，形参 s 被析构，所以产生了第五行信息；回到主函数后，输出第六行信息；最后主函数结束时，randy 对象被析构，所以产生了第七行信息。

拷贝构造函数中 strcat()是将"copy of"拼接在 name 之前，它的头文件是 string.h。通常拷贝构造函数将严格限制在只制作拷贝，但是，本程序为了要帮助大家了解其真相，在一些函数体中，设置了输出语句。

14.5 默认拷贝构造函数

类定义中，如果未提供自己的拷贝构造函数，则 C++提供一个默认拷贝构造函数，就像没有提供构造函数时，C++提供默认构造函数一样。

C++提供的默认拷贝构造函数工作的方法是，完成一个成员一个成员的拷贝。如果成员是类对象，则调用其拷贝构造函数或者默认拷贝构造函数。

例如，下面的程序中 Tutor 类使用了默认拷贝构造函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch14_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName = "no name")
    {
      cout <<"Constructing new student " <<pName <<endl;
      strncpy(name,pName);
      name[sizeof(name) - 1] = '\0';
    }

    Student(Student& s)
    {
      cout <<"Constructing copy of " <<s.name <<endl;
      strcpy(name,"copy of ");
      strcat(name,s.name);
    }

   ~Student()
    {
      cout <<"Destructing " <<name <<endl;
    }
  protected:
```

```
    char name[40];
};

class Tutor
{
  public:
    Tutor(Student& s):student(s)
    {
      cout <<"Constructing tutor\n";
    }
  protected:
    Student student;
};

void fn(Tutor tutor)
{
  cout <<"In function fn()\n";
}

int main()
{
  Student randy("Randy");
  Tutor tutor(randy);
  cout <<"Calling fn()\n";
  fn(tutor);
  cout <<"Returned from fn()\n";
}
```

运行结果为：

```
Constructing new student Randy
Constructing copy of Randy
Constructing tutor
Calling fn()
Constructing copy of copy of Randy
In function fn()
Destructing copy of copy of Randy
Returned from fn()
Destructing copy of Randy
Destructing Randy
```

程序一开始运行，进入主函数，首先构造对象 randy，调用 Student 构造函数，产生第一行信息；对象 tutor 是通过调用构造函数 Tutor(Student&)来创建的，该构造函数通过调用 Student 的拷贝构造函数来初始化数据成员 Tutor:: Student，产生第二行信息；在执行 Tutor 构造函数时，产生第三行信息；接着输出第四行；然后调用 fn()，需要创建 tutor 的一个拷贝，因为 Tutor 类没有定义拷贝构造函数，所以就调用 C++默认的拷贝构造函数，在拷贝成员 student 对象时，调用 Student 拷贝构造函数，结果在名字 copy of Randy 之前又接上了一个 copy of，得到第五行输出；进入 fn()函数体中，得到第六行信息；从 fn()返回时，形参 tutor 析构，调用的是默认析构函数，当析构到成员 student 时，调用 Student 析构函数，产生第七行输出；接着在主函数，输出第八行信息；退出主函数时，先析构 tutor 对象，析构中调用 Student 析构函数，产生第九行信息；最后析构 Randy 对象，得到最后一行输出。

14.6 浅拷贝与深拷贝

在默认拷贝构造函数中，拷贝的策略是逐个成员依次拷贝。但是，一个类可能会拥有资源，当其构造函数分配了一个资源（例如堆内存）的时候，会发生什么呢？如果拷贝构造函数简单地制作了一个该对象的拷贝，而不对它本身进行资源分配和复制，就得面临一个麻烦的局面：两个对象都拥有同一个资源。当对象析构时，该资源将经历两次资源返还。

例如，下面的程序描述了 Person 对象被简单拷贝后，面临析构时的困惑：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch14_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Person
{
  public:
    Person(char * pN)
    {
      cout <<"Constructing " <<pN <<endl;
      pName = new char[strlen(pN) + 1];
      if(pName!= 0)
      {
        strcpy(pName,pN);
      }
    }

    ~Person()
    {
      cout <<"Destructing " <<pName <<endl;
      pName[0] = '\0';
      delete pName;
    }
  protected:
    char * pName;
};

int main()
{
  Person p1("Randy");
  Person p2 = p1;       //即 Person p2(p1);
}
```

运行结果为：

```
Constructing Randy
Destructing Randy
Destructing
Null pointer assignment
```

程序开始运行时，创建 p1 对象，p1 对象的构造函数从堆中分配空间并赋给数据成员 pName，同时，产生第一行输出；执行“Person p2＝p1；”时，因为没有定义拷贝构造函数，于是就调用默认拷贝构造函数，使得 p2 与 p1 完全一样，并没有新分配堆空间给 p2，见图 14-2；主函数结束时，对象逐个析构，析构 p2 时，将堆中字符串清成空串，然后将堆空间返还给系统，并同时得到第二行输出；析构 p1 时，因为这时 pName 指向的是空串，所以第三行输出中显示的只是 Destructing；当执行“delete pName；”时，系统报错，显示第四行结果。

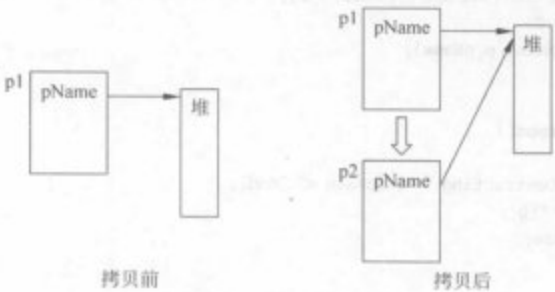


图 14-2　p1～p2 的浅拷贝

创建 p2 时，对象 p1 被复制给了 p2，但资源并未复制，因此，p1 和 p2 指向同一个资源，这称为浅拷贝。

当一个对象创建时，分配了资源，这时，就需要定义自己的拷贝构造函数，使之不但拷贝成员，也分配和拷贝资源。

例如，下面的代码是在程序 ch14_3.cpp 的基础上，增加一个 Person 类的拷贝构造函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch14_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Person
{
  public:
    Person(char * pN);
    Person(Person& p);
  ~Person();
  protected:
    char * pName;
};

Person::Person(char * pN)
{
  cout <<"Constructing " <<pN <<endl;
  pName = new char[strlen(pN) + 1];
  if(pName!= 0)
```

```
    {
      strcpy(pName,pN);
    }
}

Person::Person(Person& p)
{
  cout <<"Copying " <<p.pName <<" into its own block\n";
  pName = new char[strlen(p.pName) + 1];
  if(pName!= 0)
    strcpy(pName,p.pName);
}

Person::~Person()
{
  cout <<"Destructing " <<pName <<endl;
  pName[0] = '\0';
  delete pName;
}

int main()
{
  Person p1("Randy");
  Person p2 = p1;      //即 Person p2(p1);
}
```

运行结果为：

```
Constructing Randy
Copying Randy into its own block
Destructing Randy
Destructing Randy
```

程序开始运行时，创建 p1 对象，产生第一行输出；然后用 p1 去创建 p2 对象，调用的是自己定义的拷贝构造函数，于是得到第二行输出；拷贝构造函数中，不但复制了对象空间，也复制资源（堆内存空间），见图 14-3；当主函数退出时，先后析构 p2 和 p1，但这时候对象们有其各自的资源，所以，析构函数工作得很好，产生最后两行输出。

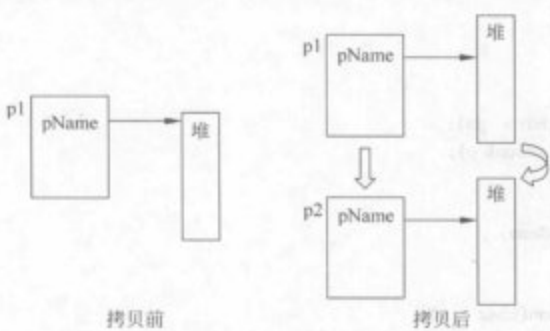


图 14-3　p1～p2 的深拷贝

创建 p2 时，对象 p1 被复制给了 p2，同时资源也作了复制，因此，p1 和 p2 指向不同的资源，这称为深拷贝。

堆内存并不是唯一需要拷贝构造函数的资源，但它是最常用的一个。打开文件，占有硬设备（例如打印机）服务等也需要深拷贝。它们也是析构函数必须返还的资源类型。因此，一个很好的经验是：**如果你的类需要析构函数来析构资源，则它也需要一个拷贝构造函数。**因为通常对象是自动被析构的。如果需要一个自定义的析构函数，那就意味着有额外资源要在对象被析构之前释放。此时，对象的拷贝就不是浅拷贝了。

14.7 临时对象

当函数返回一个对象时，要创建一个临时对象以存放返回的对象。

例如，下面的代码中，返回的 ms 对象将产生一个临时对象：

```
Student fn()
{
  //...
  Student ms("Randy");
  return ms;
}

int main()
{
  Student s;
  s = fn();
  //...
}
```

在这里，系统调用拷贝构造函数将 ms 拷贝到新创建的临时对象中，见图 14-4。

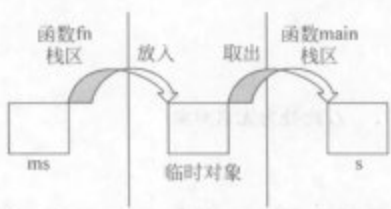


图 14-4　返回对象的函数运行结束时

一般规定，创建的临时对象，在整个创建它们的外部表达式范围内有效，否则无效。也就是说，“s=fn();”这个外部表达式，当 fn() 返回时产生的临时对象拷贝给 s 后，临时对象就析构了。

例如，下面的代码中，引用 refs 不再有效：

```
int main()
{
  Student& refs = fn();
  //...
}
```

因为外部表达式“Student& refs=fn();”到分号处结束，以后从 fn()返回的临时对象便不再有效，这就意味着引用 refs 的实体已不存在，所以接下去的任何对 refs 的引用都是错的。

又例如，下面的代码中，一切临时对象都在一个外部表达式中结束：

```
Student fn1();
int fn2(Student&);
int main()
{
  int x;
  x = 3 * fn2(fn1()) + 10;
  //...
}
```

fn1()返回时，创建临时对象作为 fn2()的实参，此时，在 fn2()中一直有效；当 fn2()返回一个 int 值参与计算表达式时，那个临时对象仍有效；一旦计算完成，赋值给 x 后，则临时对象被析构。

14.8 无名对象

可以直接调用构造函数产生无名对象。

例如，下面的代码在函数 fn()中，创建了一个无名对象：

```
class Student
{
  public:
    Student(char * );
  //...
};

void fn()
{
  Student("Randy");      //此处为无名对象
  //...
}
```

无名对象可以作为实参传递给函数，可以拿来拷贝构造一个新对象，也可以初始化一个引用的声明。

例如，下面的代码表达了无名对象典型的三种用法：

```
void fn(Student& s);

int main()
{
  Student& refs = Student("Randy");      //初始化引用
  Student s = Student("Jenny");          //初始化对象定义
  fn(Student("Danny"));                  //函数参数
}
```

主函数开始运行时，第一个执行的是拿无名对象初始化一个引用。由于是在函数内部，

所以无名对象作为局部对象产生在栈空间中，从作用域上看，该引用与无名对象是相同的，它完全等价于“Student refs="Randy";”所以这种使用是多余的。

第二个执行的是用无名对象拷贝构造一个对象 s。按理说，C++先调用构造函数“Student(char *);”创建一个无名对象，然后再调用一个拷贝构造函数“Student(Student&);”(或许是默认的)创建对象 s；但是，由于是用无名对象去拷贝构造一个对象，拷贝完后，无名对象就失去了任何作用，对于这种情况，C++特别地将其看作为“Student s="Jenny";”效果一样，而且可以省略创建无名对象这一步。

第三个执行的是无名对象作为实参传递给形参 s，C++先调用构造函数创建一个无名对象，然后将该无名对象初始化给了引用形参 s 对象，由于实参是在主函数中，所以无名对象是在主函数的栈区中创建，函数 fn()的形参 s 引用的是主函数栈空间中的一个对象。它等价于：

```
Student s("Danny");
fn(s);
```

如果对象 s 仅仅是为了充当函数 fn()实参的需要，完全可以用第三个执行来代替。

当运行到主函数结束的时候，将有一个主函数中的 s 对象和 3 个无名对象被析构。

14.9 构造函数用于类型转换

5/8 与 5.0/8 结果不同，原因是 C++执行了两种不同的操作。5.0/8 匹配了两个 double 类型数的除法，C++知道如何将 8 转换成 double 型，这是基本数据类型的转换。但是，转换用户定义的类类型，必须由用户告知。用户告知的方式就是定义含一个参数的构造函数。

例如，下面的代码中定义了学生类的构造函数：

```
class Student
{
  public:
    Student(char *);

  //...
};

void fn(Student& s);

int main()
{
  fn("Jenny");
}
```

这里 Student(char *)构造函数同时也在告知，如何将 char * 转换成一个 Student 对象。如果有重载函数 fn(char *)，则调用 fn("Jenny")马上匹配了事。但就是因为没有这样的重载函数，所以 C++对所有 fn 函数进行类型转换试探，包括构造函数。

因为有 Student(char *)的构造函数，又有 fn(Student& s)函数，于是，fn("Jenny")便被认为是 fn(Student("Jenny"))，最终予以匹配。把构造函数用来从一种类型转换为另一种类型，这是 C++从类机制中获得的附加性能。但要注意下面两点：

(1) 只会尝试含有一个参数的构造函数；

(2) 如果有二义性，则放弃尝试。例如：

```
class Student
{
  public:
    Student(char * pName = "no name");
};

class Teacher
{
  public:
    Teacher(char * pName = "no name");
};

void addCourse(Student& s);
void addCourse(Teacher& t);

int main()
{
  addCourse("Prof.Dingleberry"); //error: 二义性
}
```

改正的方法是，只要显式转换一下：

```
addCourse(Teacher("Prof.Dingleberry"));
```

小结

运算符 new 分配堆内存，如果成功，则返回指向该内存的空间，如果失败，则返回 NULL。所以每次使用运算符 new 动态分配内存时，都应测试 new 的返回指针值，以防分配失败。

堆空间的大小是有限的，视其操作系统和编译设置的不同而不同。当程序不再使用所分配的堆空间时，应及时用 delete 释放它们。

由 C++提供的默认拷贝构造函数只是对对象进行浅拷贝复制。如果对象的数据成员包括指向堆空间的指针，就不能使用这种拷贝方式，此时必须自定义拷贝构造函数，为创建的对象分配堆空间。

练习

14.1 阅读下面的程序，写出运行结果。

```
#include <iostream.h>

class Samp
{
  public:
    void Setij(int a, int b){i = a, j = b;}
```

```
    ~Samp()
      {
        cout <<"Destroying.." <<i <<endl;
      }

      int GetMulti(){return i * j;}
    protected:
      int i;
      int j;
};

int main()
{
  Samp * p;
  p = new Samp[10];
  if(!p)
  {
    cout <<"Allocation error\n";
    return;
  }

  for(int j = 0; j<10; j ++ )
    p[j].Setij(j,j);

  for(int k = 0; k<10; k ++ )
    cout <<"Multi[" <<k <<"] is:"
         <<p[k].GetMulti() <<endl;

  delete[ ]p;
}
```

14.2 写出下面程序的运行结果，请用增加拷贝构造函数的方法避免存在的问题。

```
#include <iostream.h>

class Vector
{
  public:
    Vector(int s = 100);
    int& Elem(int ndx);
    void Display();
    void Set();
  ~Vector();
  protected:
    int size;
    int * buffer;
};

Vector::Vector(int s)
{
  buffer = new int[size = s];
  for(int i = 0;icsize;i + + )
    buffer[i] = i * i;
}

int& Vector::Elem(int ndx)
{
```

```
    if(ndx<0||ndx>=size)
    {
      cout <<"error in index" <<endl;
      exit(1);
    }
    return buffer[ndx];
}

void Vector::Display()
{
    for(int j=0; j<size; j++)
      cout <<Elem(j) <<endl;
}
void Vector::Set()
{
    for(int j=0; j<size; j++)
      Elem(j)=j+1;
}

Vector::~Vector()
{
    delete[]buffer;
}

int main()
{
    Vector a(10);
    Vector b(a);
    a.Set();
    b.Display();
}
```

14.3 完善下列程序，定义每个成员函数和非成员函数，输出必要的信息，检查临时对象何时被创建，何时被析构。

```
class X
{
  public:
    X(int);
    X(X&);
    ~X();
};

X f(X);

int main()
{
    X a(1);
    X b=f(X(2));
    a=f(a);
}
```

14.4 读下面的程序与运行结果，添上一个拷贝构造函数来完善整个程序。

```
#include <iostream.h>
```

```
class CAT
{
  public:
    CAT();
    CAT(const& CAT&);
  ~CAT();
    int GetAge() const {return * itsAge;}
    void SetAge(int age){ * itsAge = age;}
  protected:
    int * itsAge;
};

CAT::CAT()
{
  itsAge = new int;
  * itsAge = 5;
}

CAT::~CAT()
{
  delete itsAge;
  itsAge = 0;
}

int main()
{
  CAT frisky;
  cout <<"frisky's age:" <<frisky.GetAge() <<endl;
  cout <<"Setting frisky to 6...\n";
  frisky.SetAge(6);
  cout <<"Creating boots from frisky\n";
  CAT boots(frisky);
  cout <<"frisky's age:" <<frisky.GetAge() <<endl;
  cout <<"boots'age:" <<boots.GetAge() <<endl;
  cout <<"setting frisky to 7...\n";
  frisky.SetAge(7);
  cout <<"frisky's age:" <<frisky.GetAge() <<endl;
  cout <<"boots'age:" <<boots.GetAge() <<endl;
}
```

运行结果为：

```
frisky's age:5
Setting frisky to 6...
Creating boots from frisky
frisky's age:6
boots'age:6
Setting frisky to 7...
frisky's age:7
boots'age:6
```

第 15 章　静态成员与友元

类是类型而不是数据对象，每个类的对象都是该类数据成员的拷贝。然而，往往需要让类的所有对象在类的范围内共享某个数据。声明为 static 的类成员便能在类范围中共享，称之为静态成员。友元函数完全是普通的 C++函数，不同的是，它可以访问类的保护或私有成员，方便编程，提高了效率，但却破坏了类的封装。学习本章后，应该懂得怎样声明一个静态数据成员，怎样使用静态成员函数以及静态成员函数为什么与特定对象无关。还应能把握友元的使用，理解友元作用的局限性。

15.1　静态成员的需要性

有一些属性是类中所有对象共有的。

例如 Student 类中，链表的第一个指针以及类中学生数：

```
class Student
{
  //...

  protected:
    Student * pFirst;      //链表首指针
    int count;      //学生数
};
```

这个类声明，意味着每个学生对象都有一个链表首指针和学生数，但是却没有集中成一个成员供所有对象共享。

要想得到现有的学生数，不能到类中去取，因为类不是一个占有内存的实体。那么，到哪个对象中去取？难道一旦学生人数变化，要每个对象逐个修改？

如果放在全局变量中，则它们在类的外面，既不安全，又影响了重用性。

例如，下面的代码用全局变量来表示学生类链表首指针和学生人数：

```
class Student
{
  //...
};

int count;          //学生人数
Student * pFirst;   //学生类链表首指针

void fn()
{
  Student ss;    //创建第一个学生对象
  count ++ ;     //学生人数增 1
  pFirst = &ss;      //没有对 pFirst 约束，随便乱用，一点也不把它当链首指针
  //...
```

```
}     //fn()退出时,ss作用域终止,ss被析构,可学生人数忘了减1
```

面对庞大的程序,没有真正指明哪个函数对 count 和 pFirst 负责。这种无规则会引起软件设计的混乱,一旦程序变大,维护量就急剧上升。

又例如,在程序 ch12_10.cpp 中,定义全局变量 nextStudentId,它既不能放在头文件中定义,也不能放在类 StudentId 的内部实现中,只能放在应用程序中主函数 main()的前面。在重用 StudentId 类的时候,总是还要额外地考虑一个全局变量的处置,这不得不使类的封装性受到伤害。

若能将学生人数和链表首指针封装在类里面,则可以受保护,而且可以作为一个类而重用。这种属于类的一部分,但既不适于用普通成员表示,也不适于全局变量表示的数据,用静态成员来表示。

15.2 静态成员的使用

成员有数据成员和成员函数之分,静态成员也有静态数据成员和静态成员函数之分。静态成员用 static 声明。

例如,下面的程序在类中定义了一个静态数据成员和一个静态成员函数,在它的构造函数和析构函数中对静态数据成员进行操作,在应用程序中,调用了静态成员函数:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch15_1.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName = "no name")
    {
      cout <<"create one student\n";
      strcpy(name, pName);
      noOfStudents++;   //静态成员:每创建一个对象,学生人数增1
      cout <<noOfStudents <<endl;
    }
   ~Student()
    {
      cout <<"destruct one student\n";
      noOfStudents--;   //每析构一个对象,学生人数减1
      cout <<noOfStudents <<endl;
    }

    static int number()    //静态成员函数
    {
      return noOfStudents;
    }
  protected:
    static int noOfStudents;       //若写成 noOfStudents = 0;则非法
```

```
    char name[40];
};
int Student::noOfStudents = 0;    //静态数据成员在类声明外分配空间和初始化

void fn()
{
  Student s1;
  Student s2;
  cout <<Student::number() <<endl; //调用静态成员函数用类名引导
}
int main()
{
  fn();
  cout <<Student::number() <<endl;   //调用静态成员函数用类名引导
}
```

运行结果为：

```
create one student
1
create one student
2
2
destruct one student
1
destruct one student
0
0
```

数据成员 noOfStudents，既不是对象 s1 也不是对象 s2 的一部分。Student 类随着对象的产生，每个对象都有一个 name 成员值，但无论对象有多少，甚至没有，静态成员 noOfStudents 也只有一个。所有 Student 对象都共享它，并且能够访问它。

在 Student 对象空间中，是没有静态数据成员 noOfStudents 的，它的空间，不会随着对象的产生而分配，或随着对象的消失而回收。所以它的空间分配并不在 Student 的构造函数里完成，并且空间回收也不在类的析构函数里完成。

静态数据成员确实是在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的 main()函数前的全局数据声明和定义处。

静态数据成员要实际地分配空间，故不能在类声明中定义(只能声明数据成员)。类声明只声明一个类的“尺寸与规格”，并不进行实际的内存分配，所以在类声明中写成定义“static int noOfStudents=0;”是错误的。它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。

静态数据成员也不能在 main()函数之前的全局数据声明处定义，因为那样会使每个重用该类的应用程序在包含了声明该类的头文件后，都不得不在应用程序中再定义一下该类的静态成员。买了电饭煲，可是店家告诉你盖子上的锣丝要自己拧上去，你说烦不烦？

例如，下面的代码重用 Student 类，但在应用程序中不得不再定义 Student 类的静态成员：

```
file1.cpp      //Student 类的内部实现部分

#include "student.h"

//类的成员函数定义(没有包括静态数据成员定义)
```

```
file2.cpp      //应用程序重用了 Student 类

#include "student.h"
#include <iostream.h>

int Student::onOfStudents = 0;        //不便于重用

void fn()
{
  Student s1;
  Student s2;
  cout <<Student::number() <<endl;
}

int main()
{
  fn();
  cout <<Student::number() <<endl;
}
```

静态数据成员是类的一部分,静态数据成员的定义是类定义的一部分,将其放在类的内部实现部分中定义是再合适不过了。定义时,要用类名引导。重用该类时,简单地包含其头文件便可。

例如,下面的程序将 ch15_1.cpp 改成了多文件程序实现结构:

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              student.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

class Student
{
  public:
    Student(char * pName = "no name");
    ~Student();
    static int number();      //静态成员函数
  protected:
    static int noOfStudents;
    char name[40];
};

//* * * * * * * * * * * * * * * * * * * * * * * * * *
//* *              student.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>
#include "student.h"
```

```
int Student::noOfStudents = 0;        //在类得内部实现中分配空间和初始化

Student::Student(char * pName)
{
  cout <<"create one student\n";
  strcpy(name, pName);
  noOfStudents ++ ;      //静态成员：每创建一个对象,学生人数增 1
  cout <<noOfStudents <<endl;
}

Student::~Student()
{
  cout <<"destruct one student\n";
  noOfStudents -- ;      //每析构一个对象,学生人数减 1
  cout <<noOfStudents <<endl;
}

int Student::number()   //静态成员函数
{
  return noOfStudents;
}

//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch15_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include "student.h"      //重用 Student 类
# include <iostream.h>

void fn()
{
  Student s1;
  Student s2;
  cout <<Student::number() <<endl;
}
int main()
{
  fn();
  cout <<Student::number() <<endl;
}
```

工程文件 ch15_2.prj 中包含：

```
student.cpp
ch15_2.cpp
```

运行结果为：

```
create one student
1
create one student
2
2
destruct one student
1
destruct one student
0
0
```

15.3 静态数据成员

公共静态数据成员可被类的外部访问，保护或私有静态数据成员只可被类的内部访问。例如，下面的代码描述一个公共的静态数据成员：

```
class Student
{
  public:
    Student()
    {
      noOfStudents ++ ;
      //...
    }
    static int noOfStudents; //公共静态数据成员
    //...
};
void fn(Student& s1,Student& s2)
{
  cout <<s1.noOfStudents; //此处也可以访问静态数据成员
}
```

在类的外部，访问静态数据成员的形式可以是 s1. noOfStudents，它等价于 s2. noOfStudents，更通常的用法是 Student::noOfStudents(不能用 Student. noOfStudents)。其意义是，静态数据成员是属于 Student 类的，而不是属于哪个特定对象的，它也不需要依赖某个特定对象的数据。

例如，下面的代码用返回对象引用的成员函数作为对象值去操作静态成员，但是静态成员只取返回对象的类型，其成员函数不一定会被执行：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch15_3.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Student
{
  public:
    static int noOfStudents;
    Student& nextStudent()
    {
      noOfStudents ++ ;
      return * this;
    }
  //...
};

int Student::noOfStudents = 0;

void fn(Student& s)
{
  cout <<s.nextStudent().noOfStudents <<endl;
}
```

```
int main()
{
  Student ss;
  fn(ss);
}
```

运行结果为：

```
1
```

s.nextStudent()返回 Student 类对象的引用，该引用作为后面的点操作符左操作数，而右操作数是静态数据成员 noOfStudents。

成员函数 nextStudent()实际上未被调用执行。引用静态成员时，C++系统只关心静态成员的类类型。

静态数据成员用得比较多的场合一般为：

(1) 用来保存流动变化的对象个数(如 noOfStudents)；

(2) 作为一个标志，指示一个特定的动作是否发生(如可能创建几个对象，每个对象要对某个磁盘文件进行写操作，但显然在同一时间里只允许一个对象写文件，在这种情况下，用户希望说明一个静态数据成员指出文件何时正在使用，何时处于空闲状态)；

(3) 一个指向一个链表第一成员或最后一个成员的指针(如 pFirst)。

例如，下面的程序描述一个学生类，该类对象是一个个的学生，它们构成一个单向链表：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch15_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName);
    ~Student();
  protected:
    static Student * pFirst;
    Student * pNext;
    char name[40];
};
Student * Student::pFirst = 0;
Student::Student(char * pName)
{
  strncpy(name,pName);
  name[sizeof(name) - 1] = '\0';
  pNext = pFirst; //每新建一个结点(对象),就将其挂在链首
  pFirst = this;
}
Student::~Student()
{
  cout <<this->name <<endl;

  if(pFirst == this) //如果要删除链首结点,则只要链首指针指向下一个
  {
```

```
        pFirst = pNext;
        return;
    }
    for(Student * pS = pFirst; pS; pS = pS->pNext)
        if(pS->pNext == this) //找到时,pS 指向当前结点的结点
        {
            pS->pNext = pNext; //pNext 即 this->pNext
            return;
        }
}

Student * fn()
{
    Student * pS = new Student("Jenny");
    Student sb("Jone");
    return pS;
}
int main()
{
    Student sa("Jamsa");
    Student * sb = fn();
    Student sc("Tracey");
    delete sb;
}
```

运行结果为：

```
Jone
Jenny
Tracey
Jamsa
```

实现链表结构的学生类，需要一个链首指针 pFirst，每个对象都需要一个指向下一个对象的指针 pNext，所以 pNext 数据成员不是静态的，而 pFirst 数据成员是静态的。

运行时，主函数先创建一个名叫 Jamsa 的学生对象，并使链表含有一个结点。随后调用函数 fn()。在函数 fn()中，从堆中创建一个名叫 Jenny 的学生对象，这时，在链中含有 2 个结点，Jenny 是首结点。fn()又在栈中创建 Jone 的学生对象，这时在链中含有 3 个结点，Jone 成为链首结点了。

函数 fn()返回时，名叫 Jone 的 sb 对象的作用域结束，析构它时，将其从链表中删除。删除的是链首结点，此时链中含有 2 个结点。函数 fn()返回一个堆对象的指针给主函数中的 sb。主函数又接着创建一个新的对象 sc，它链入链表中，成为首结点。

最后一条语句释放堆对象空间。释放时，自动析构堆对象，该对象便从链中删除，删除的该结点不在链首，删除过程中，先找到该结点，然后将前后 2 个结点连起来。此后链中剩下 2 个结点。当退出主函数时，先后析构 Tracey 和 Jamsa 的两个对象。析构后，链表清空如初。

链表操作是在构造函数和析构函数中进行的。构造中增加结点的处理比从析构中删除一个结点相对要容易一些。删除一个学生结点时，先要在链表中找到指向当前结点(this 指向的结点)的结点，然后把前一个结点和后一个结点链接起来。在实现中，找到当前结点位置时，保存指向当前结点的结点指针是至关重要的。

链表操作的原理见图 15-1、图 15-2。

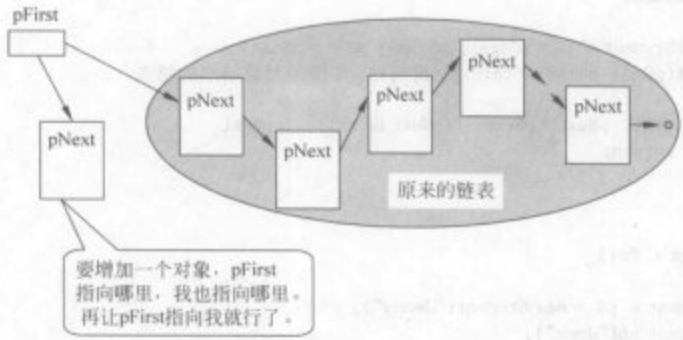


图 15-1　在链表中增加一个学生结点

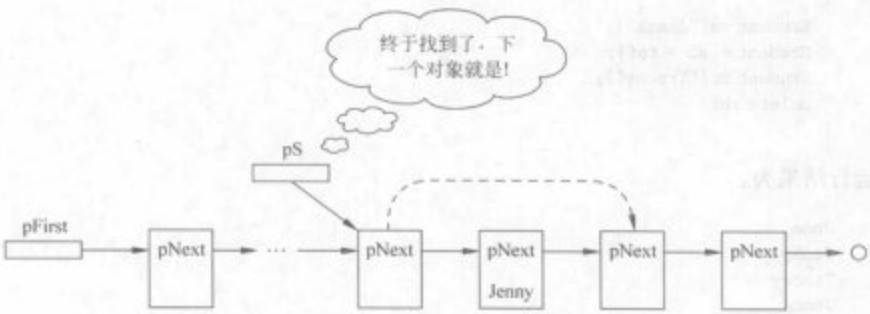


图 15-2　在链表中删除一个学生结点

15.4　静态成员函数

静态成员函数定义是类的内部实现，属于类定义的一部分。它的定义位置与一般成员函数一样。

与静态数据成员一样，静态成员函数与类相联系，不与类的对象相联系，所以访问静态成员函数时，不需要对象。如果用对象去引用静态成员函数，只是用其类型。

例如，下面的程序，两种调用静态成员函数的方法都是合法的，而且意义一样：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch15_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Student
{
  public:
    static int number()
```

```
    {
      return noOfStudents;
    }
    //...
  protected:
    char name[40];
    static int noOfStudents;
};
int Student::noOfStudents = 1;

int main()
{
  Student s;
  cout <<s.number() <<endl;      //ok 用对象引用静态成员函数
  cout <<Student::number() <<endl;      //ok 用类名引导静态成员函数
}
```

运行结果为：

```
1
1
```

一个静态成员函数不与任何对象相联系，故不能对非静态成员进行默认访问。例如，下面的代码中静态成员函数不应访问非静态数据成员 name：

```
#include <iostream.h>

class Student
{
  public:
    static char * sName()      //静态成员函数是所有对象共享的
    {
      cout <<noOfStudents <<endl;
      return name;       //error 哪个对象?
    }
  protected:
    char name[40];
    static int noOfStudents;
};

int Student::noOfStudents = 0;

void fn()
{
  //...
  Student s;
  cout <<s.sName() <<endl; //sName()从对象 s 上得到的是 Student 类型
}
```

静态成员函数 sName()只认类型，不认对象。即使用对象 s 引导的 s.sName()，也只识别对象 s 所属类型。静态成员的这种性质使得访问任何非静态成员的操作都是非法的。所以在上例的静态成员函数定义中，成员 name 对 sName()来说不知所云。

然而，并不是说静态成员函数不能对非静态成员函数访问。

例如，下面的程序说明了一个访问对象中成员的方法：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch15_6.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

class Student
{
  public:
    Student(char * pName);
  ~Student();
    static Student * findname(char * pName);
  protected:
    static Student * pFirst;
    Student * pNext;
    char name[40];
};

Student * Student::pFirst = 0; //静态成员空间分配及初始化
Student::Student(char * pName)
{
  strncpy(name,pName);
  name[sizeof(name) - 1] = '\0';
  pNext = pFirst;
  pFirst = this;
}

Student::~Studen + ()
{
  if(pFirst == this)
  {
     pFirst = pNext;
     return;
}
for(Student * pS = pFirst; pS; pS = pS->pNext)
  if (pS->pNext == this)
  {
      pS->pNext = pNext;
      return;
  }
}

Student * Student::findname(char * pName)
{
  for(Student * pS = pFirst; pS; pS = pS - >pNext)
    if(strcmp(pS - >name,pName) == 0)
    {
      return pS;
    }
  return (Student * )0;
}

int main()
{
  Student s1("Randy");
  Student s2("Jenny");
```

```
    Student s3("Kinsey");
    Student * pS = Student::findname("Jenny");
    if(pS)
      cout <<"ok." <<endl;
    else
      cout <<"no find." <<endl;
}
```

运行结果为：

```
ok.
```

findname()是静态成员函数，它查找链中所有对象，看有没有"Jenny"这个学生，因此它要访问对象中的 name。但它是从静态数据成员 pFirst 这个链首指针着手的，通过一个临时指针不断指向所要用的对象，借此来访问对象成员。一个静态成员函数不存在任何默认对象与之相联系，即使 s1. findname("jenny")，其效果也和上例的 Student::findname("Jenny")一样。

→ 静态成员函数与非静态成员函数的根本区别是什么？

它们的根本区别在于静态成员函数没有 this 指针，而非静态成员函数有一个指向当前对象的指针 this。

例如：

```
class Sc
{
  public:
    void nsfn(int a);   //像声明 Sc::nsfn(Sc * this,int a);
    static void sfn(int a); //无 this 指针
  //...
}

void f(Sc& s)
{
  s.nsfn(10);      //转换为 Sc::nsfn(&s, 10)
  s.sfn(10);       //转换为 Sc::sfn(10)
}
```

函数 nsfn()可被认为它声明为 void Sc::nsfn(Sc * this,int a)。对 nsfn()的调用，编译像注解的那样进行转换，s 的地址作为第一个传递参数。（你并不实际写该调用，由编译来完成。）

在函数内部，Sc::nsfn()对非静态成员的访问将自动地把 this 参数作为指向当前对象的指针。而当 Sc::sfn()被调用时，没有任何对象的地址被传递。因此，当访问非静态成员时，无 this 指针（出错）。这就是为什么一个静态成员函数与任何当前对象都无联系的原因。

15.5 需要友元的原因

有时候，普通函数需要直接访问一个类的保护或私有数据成员。如果没有友元机制，则只能将类的数据成员声明为公共的，从而，任何函数都可以无约束地访问它。

普通函数需要直接访问类的保护或私有数据成员的原因主要是为提高效率。

例如，下面的代码是做矩阵和向量的乘法。矩阵类和向量类分别为 Matrix 和 Vector，乘法操作的函数只能是普通函数，因为一个函数不可能既是这个类的成员又是那个类的成员：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch15_7.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <stdlib.h>

class Vector
{
  public:
    Vector(int);
    ~Vector(){delete[]v;}   //将堆中数组空间返还
    Vector(Vector & );
    int Size(){return sz;}
    void Display();
    int& Elem(int);    //返回向量元素
  protected:
    int * v;   //指向一个数组，表示向量
    int sz;   //元素个数
};

Vector::Vector(int s)
{
  if(s<=0)
  {
    cerr <<"bad Vector size.\n";
  exit(1);
  }
    sz = s;
    v = new int[s];   //从堆中分配一个数组存放向量元素
}

int& Vector::Elem(int i) //引用返回的目的是返回值可以作左值
{
  if(i<0||sz<=i)
  {
    cerr <<"Vector index out of range.\n";
  exit(1);
  }
    return v[i];
}
Vector::Veetor(Vector& vec)
{
  V=new int[sz=vec.sz];
  memcpy((void*)V,(void*)vec.V,sz*sizeof(int));
}

void Vector::Display()
{
  for(int i=0; i<sz; i++)
    cout <<v[i] <<" ";
  cout <<endl;
}
```

```
class Matrix
{
  public:
    Matrix(int,int);
    Matrix(Matrix & );
    ~Matrix(){delete[ ]m;}
    int SizeL(){return szl;}
    int SizeR(){return szr;}
    int& Elem(int,int);
  protected:
    int * m;
    int szl;
    int szr;
};

Matrix::Matrix(int i,int j)
{
  if(i<=0)||(j<=0)
  {
    cerr <<"bad Matrix size.\n";
  exit(1);
  }
    szl=i;
    szr=j;
    m =new int[i*j];
  }
}
Matrix::Matrix(Matrix& mat)
{
  szl=mot.szl;
  szr=mat.szr;
  m=new int[szl*szr];
  memcpy((void*)m,(void*)mat.m, sz(*szr*sizeof(int));
}
int& Matrix::Elem(int i,int j)   //引用返回的目的是返回值可以作左值
{
  if(i<0||szl<=i||j<0||szr<=j)
  {
    cerr <<"Matrix index out of range.\n";
  exit(1);
  }
    return m[i*szr+j];
}

Vector Multiply(Matrix& m, Vector& v) //矩阵乘向量的普通函数
{
  if(m.SizeR()!=v.Size())
  {
    cerr <<"bad multiplying Matrix with Vector.\n";
    exit(1);
  }

  Vector r(m.SizeL());   //创建一个存放结果的空向量

  for(int i=0; i<m.SizeL(); i++)
  {
    r.Elem(i)=0;
```

```
    for(int j = 0; j<m.SizeR(); j++)
      r.Elem(i) += m.Elem(i,j) * v.Elem(j);
  }

  return r;
}

int main()
{
  Matrix ma(4,3);
  ma.Elem(0,0) = 1; ma.Elem(0,1) = 2; ma.Elem(0,2) = 3;
  ma.Elem(1,0) = 0; ma.Elem(1,1) = 1; ma.Elem(1,2) = 2;
  ma.Elem(2,0) = 1; ma.Elem(2,1) = 1; ma.Elem(2,2) = 3;
  ma.Elem(3,0) = 1; ma.Elem(3,1) = 2; ma.Elem(3,2) = 1;
  Vector ve(3);
  ve.Elem(0) = 2; ve.Elem(1) = 1; ve.Elem(2) = 0;
  Vector va = Multiply(ma,ve);
  va.Display();
}
```

运行结果为：

```
4 1 3 4
```

Matrix 中的 m 和 Vector 中的 v 是保护数据。由于 Multiply()不是 Matrix 和 Vector 类的成员，不能直接操纵 m[i][j]和 v[j]，只能通过 m. Elem(i,j)和 v. Elem(j)来访问矩阵和向量的元素。Elem()函数定义中要对下标进行合法性检查，所以，Multiply()函数要频繁地调用函数和进行下标检查。做一次上例中的小乘法，矩阵(4,3)乘以向量(3)得到向量(4)，其 Elem()和 Size()成员函数要调用 3+1+4*(1+(1+2)*3)=44 次，显然效率不高。于是希望乘法不要调用 Elem()函数，能直接访问两个类的保护数据成员。

15.6 友元的使用

在类里声明一个普通函数，标上关键字 friend，就成了该类的友元，可以访问该类的一切成员。在上节中，将 Multiply()函数声明为 Matrix 和 Vector 两个类的友元，就能使 Multiply()函数既可访问 Matrix 的保护数据成员，又可访问 Vector 类的保护数据成员了。

例如，下面的代码将程序 ch15_7. cpp 中的 Multiply()改为友元：

```
#include <iostream.h>

class Vector
{
  public:
    Vector(int);
    ~Vector(){delete[]v;}
    //int Size(){return sz;} 不需要
    void Display();
    int& Elem(int);
    friend Vector Multiply(Matrix& m, Vector& v);
  protected:
    int* v;
```

```
    int sz;
};
class Matrix
{
  public:
    Matrix(int,int);
    ~Matrix(){delete[]m;}
    //int SizeL(){return szl;} 不需要
    //int sizeR(){return szr;} 不需要
    int& Elem(int,int);
    friend Vector Multiply(Matrix& m, Vector& v);
  protected:
    int * m;
    int szl;
    int szr;
};

//省略成员函数定义

Vector Multiply(Matrix& m, Vector& v) //友元定义
{
  if(m.szr!= v.sz)   //直接访问保护数据
  {
    cerr <<"bad multiplying Matrix with Vector.\n";
    exit(1);
  }
  Vector r(m.szl);   //直接访问保护数据
  for(int i = 0; i<m.szl; i++)   //直接访问保护数据
  {
    r.v[i] = 0;   //直接访问保护数据
    for(int j = 0; j<m.szr; j++)   //直接访问保护数据
      r.v[i] += m.m[i * m.szr + j] * v.v[j]; //直接访问保护数据
  }
  return r;
}
```

这样一个乘法，由于直接访问矩阵类和向量类的保护数据，避免了频繁调用成员函数，效率就高多了。

需要友元的另一个原因是为了方便重载操作符的使用。18.2 节和 18.4 节中有关于该内容的介绍。

友元函数不是成员函数，它是类的朋友，因而能够访问类的全部成员。在类的内部，只能声明它的函数原型，加上 friend 关键字。友元声明的位置可在类的任何部位，既可在 public 区，也可在 protected 区，意义完全一样。友元函数定义则在类的外部，一般与类的成员函数定义放在一起。因为类重用时，一般友元是一起提供的。

一个类的成员函数可以是另一个类的友元。

例如，下面的代码中，教师应该可以修改学生的成绩（访问学生类的保护数据），将教师类的成员函数 assignGrades()声明为学生类的友元：

```
class Student;   //前向声明 类名声明

class Teacher
{
```

```
  //...
  public:
    void assignGrades(Student& s);   //给定成绩
  protected:
    int noOfStudents;
    Student * pList[100];
};

class Student
{
  public:
    //...
    friend void Teacher::assignGrades(Student& s);
  protected:
    Teacher * pT;
    int semesterHours;
    float gpa;
};

void Teacher::assignGrades(Student& s)
{
  s.gpa = 4.0;      //修改学生的平均成绩 gpa
}
```

在教师类声明中，声明了学生类对象的指针数组；在学生类声明中，又声明了教师类对象的指针和作为友元的教师类成员函数。解决这种交叉声明问题的方法是先进行类名声明。如例中的“class Student;”让编译知道 Student 类的名字已经登记在册，后面可以引用这个名字。类名声明不能用于定义该类的对象，因为这时还没有类的完整声明，没有分配类对象空间的依据（即不能在类名声明“class Student;”后声明 Student 类之前，出现定义类对象语句：Student ss;）。

教师类中的成员函数 assignGrades()需要 Student 类的参数才可以访问参数（Student 类对象）的保护数据。

整个类可以是另一个类的友元，该友元称为友类。友类的每个成员函数都可访问另一个类中的保护或私有数据成员。

例如，下面的代码将整个教师类看成是学生类的友类，教师既可修改成绩，又可调整学时数：

```
class Student;

class Teacher
{
  public:
    void assignGrades(Student& s);   //赋成绩
    void adjustHours(Student& s);    //调整学时数
    //...
  protected:
    int noOfStudent;
    Student * pList[100];
};
class Student
{
  public:
```

```
    friend class Teacher;    //友类
    //...
  protected:
    int semesterHours;
    float gpa;
};
```

小结

使用静态数据成员，实际上可以消灭全局变量。全局变量给面向对象程序带来的问题就是违背封装原则。要使用静态数据成员必须在 main()程序运行之前分配空间和初始化。使用静态成员函数，可以在实际创建任何对象之前初始化专有的静态数据成员。静态成员不与类的任何特定对象相关联。

静态成员的 static 一词与静态存储类的 static 是两个概念，一个论及类，一个论及内存空间的位置以及作用域限定。所以要区分静态对象和静态成员。

友元的作用主要是为了提高效率和方便编程。在 18.2 节和 18.4 节中还论及了友元在操作符重载中的作用。友元越过了类的封装，直接去操作类的私有成员，带来了性能提升和编程灵活性，加上类的访问权限确实在有些场合比较呆板，还不够完美，所以就容忍了友元这一特别语法现象。

练习

15.1

(1) 编写一个类，声明一个数据成员和一个静态数据成员。让构造函数初始化数据成员，并把静态数据成员加 1，让析构函数把静态数据成员减 1。

(2) 根据(1)编写一个应用程序，创建三个对象，然后显示它们的数据成员和静态数据成员，再析构每个对象，并显示它们对静态数据成员的影响。

(3) 修改(2)，让静态成员函数访问静态数据成员，并让静态数据成员是保护的。

15.2

(1) 下述代码有何错误，改正它。

```
#include <iostream.h>

class Animal;

void SetValue(Animal&, int);
void SetValue(Animal&, int, int);

class Animal
{
  public:
    friend void setValue(Animal&, int);
  protected:
    int itsWeight;
    int itsAge;
};
```

```
void SetValue(Animal& ta, int tw)
{
  ta.itsWeight = tw;
}
void SetValue(Animal& ta, int tw, int tn)
{
  ta.itsWeught = tw;
  ta.itsAge = tn;
}

int main()
{
  Animal peppy;
  SetValue(peppy, 5);
  SetValue(peppy,7,9);
}
```

(2) 将上面程序中的友元改成普通函数，为此增加访问类中保护数据的成员函数。

15.3 重新编写下述程序，使函数 Leisure()成为类 Car 和类 Boat 的函数。作为重新编程，在类 Car 和类 Boat 中，增加函数 get()。

```
#include <iostream.h>

class Boat;

class Car
{
  public:
    Car(int j){size = j;}
    friend int Leisure(int time, Car& aobj, Boat& bobj);
  protected:
    int size;
};
class Boat
{
  public:
    Boat(int j){size = j;}
    friend int Leisure(int time, Car& aobj, Boat& bobj);
  protected:
    int size;
};

int Leisure(int time, Car& aobj, Boat& bobj)
{
  return time * aobj.size * bobj.size;
}
int main()
{
  Car c1(2);
  Boat b1(3);
  int time = 4;
  cout <<Leisure(time,c1,b1);
}
```

第16章 继 承

继承是C++语言的一种重要机制，该机制自动地为一个类提供来自另一个类的操作和数据结构，这使得程序员只需在新类中定义已有类中没有的成分来建立新类。理解继承是理解面向对象程序设计所有方面的关键，所以本章是全书的重点之一。通过学习本章，能利用继承现有的类建立新类，能理解继承如何提高软件的重用性，理解多态性对于继承的意义，掌握多态的工作原理，理解抽象类和具体类的区别，学会运用纯虚函数。

16.1 继承的概念

图16-1展示了交通工具的类层次。最顶部的类称为基类，是交通工具类。这个基类有汽车子类。这样，交通工具类就是汽车类的父类。可以从交通工具类派生出其他类，比如飞机类、火车类和轮船类。每个类都只有交通工具类作为其父类。汽车子类还有三个子类：小汽车类、旅行车类和卡车类，每个类都以汽车类作为父类，交通工具类可称为它们的祖先类。另外，小汽车类是轿车类、工具车类和面包车类这些派生类的父类。图中展示了小型四层次的类，它用继承来派生子类。每个类有且仅有一个父类，所有子类都是一种父类。例如，小汽车是一种汽车，轿车是一种汽车，汽车是一种交通工具，小汽车也是一种交通工具。这样，每个子类代表父类的特定版本。

图16-1 继承的类层次

继承使得我们得以用一种简单的方式来描述事物。比如，描述什么叫鸭子，可以回答说：它是一种嘎嘎叫的鸟。这里鸭子是鸟类的派生，所以鸭子是鸟类的一种，鸭子又同时具有自己的特征，就是会嘎嘎叫。嘎嘎叫是区别于其他鸟类的属性。由于鸟类的特点都清楚，所以用鸟类来描述鸭子，只要举出鸭子自己所具有的特点就行了。继承使我们描述事物的能力大大增强和简单化。

面向对象程序设计可以让你声明一个新类作为另一个类的派生。派生类(也称子类)继承它父类的属性和操作。子类也声明了新的属性和新的操作，剔除了那些不适合于其用途

的继承下来的操作。这样，继承可让你重用父类的代码，专注于为子类编写新代码。

那些父类已经存在，在新的应用中，你无须修改父类。所要做的就是派生子类，在子类中做一些增加与修改。这种机制，使重用成为可能。

在早些时候，标准 C 函数库中，很少能找到可重用的软件部件。如果一个程序员已经开发了一些程序，现在要开发一个新的程序，实际上不可能在先前的程序中找到完全符合要求的程序部件，通常这些部件需要调整修改。

现实世界是分类分层的客观实在，物质有无机与有机之分，有机体有生命体与非生命体之分，生命体有动物、植物、微生物等，动物有各种，人是其中的一种，人有各个种族……

继承是我们理解事物，解决问题的方法。继承帮助我们描述事物的层次关系，帮助我们精确地描述事物，帮助我们理解事物的本质。一旦看清了事物所处的层次结构，也就找到了对应的解决办法。

继承可以使已存在的类不需修改地适应新应用，理解继承是理解面向对象程序设计所有方面的关键。

16.2 继承的工作方式

现有一个学生类 Student，要增加研究生类 GraduateStudent。由于研究生除了他自己特有的性质外，具有学生的所有性质，所以我们用继承的方法来重用学生类。

```
class Student
{
  //...
};

class GraduateStudent :public Student
{
  //...
};
```

在这里，一个 GraduateStudent 类继承了 Student 类的所有成员。继承的方式是在类定义中类名后跟：public Student。一个研究生是一个学生。当然，GraduateStudent 也包含有自己特有的成员。

下例是继承 Student 的例子，给它添加一些成员：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch16_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<string.h>

class Advisor
{
  int noOfMeeting;
};
```

```
class Student
{
  public:
    Student(char * pName = "no name")
    {
      strncpy(name, pName);
      average = semesterHours = 0;
    }

    void addCourse(int hours, float grade)
    {
      average = (semesterHours * average + grade);        //总分
      semesterHours += hours;        //总修学时
      average /= semesterHours;        //平均分
    }

    int getHours(){return semesterHours;}
    float getAverage(){return average;}
    void display()
    {
      cout <<"name = \"" <<name <<"\""
           <<", hours = " <<semesterHours
           <<", average = " <<average <<endl;
    }
  protected:
    char name[40];
    int semesterHours;
    float average;
};

class GraduateStudent :public Student
{
  public:
    int getQualifier(){return qualifierGrade;}
  protected:
    Advisor advisor;
    int qualifierGrade;
};

int main()
{
  Student ds("Lo lee undergrade");
  GraduateStudent gs;
  ds.addCourse (3, 2.5);
  ds.display();
  gs.addCourse(3, 3.0);
  gs.display();
}
```

运行结果为：

```
name = "Lo lee undergrade", hours = 3, average = 0.833333
name = "no name", hours = 3, average = 1
```

ds 是 Student 类对象，gs 是 GraduateStudent 类对象。作为 Student 的子类，对象 gs 可以做 ds 能做的任何事情，它有 name，semesterHours，average 数据成员，以及 addCourse() 成员函数，此外它还比 ds 多一点东西，即它有导师 Advisor 和资格考试分 qualifierGrade。

gs 是一个学生，所以对 Student 中的 addCourse() 成员函数的调用，等于是在调用自己的成员函数。正是由于 gs 是一个学生，所以，对下面的代码：

```
void fn(Student& s)
{
  //任何学生想要做的事
}

int main()
{
  GraduateStudent gs;
  fn(gs);
}
```

函数 fn() 期望接受 Student 类对象作为它的参数，来自 main() 的调用传给它一个 GraduateStudent 对象，fn() 把它视同 Student 对象予以接受。

事实上，GraduateStudent 的内存布局，也与“gs 是研究生，当然也是学生”相吻合。见图 16-2。

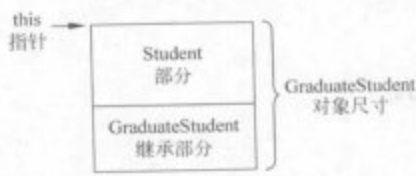


图 16-2　GraduateStudent 内存布局

在布局上 gs 对象的最初部分是 Student 数据成员，fn(gs) 等于做了一个 Student(gs) 的隐式转换，指向 gs 的 this 指针也就是指向 Student 对象的指针。由此看出，gs 对象中包含有 Student 对象空间部分，gs 用 this 指针访问 Student 成员与访问自己增加的成员没有差别。

16.3　派生类的构造

在 ch16_1.cpp 的例子中，并没有声明派生类 GraduateStudent 的构造函数，根据类的实现机制，派生类对象创建时，将执行其默认的构造函数。该默认构造函数会首先调用基类的默认构造函数，而基类没有默认构造函数，但正好匹配默认参数的构造函数。所以在运行结果中，gs 对象的 name 值为“no name”。

派生类可以直接访问基类的保护数据成员，甚至在构造时初始化它们，但是一般不这么做，而是通过基类的接口（成员函数）去访问它们，初始化也是通过基类的构造函数。这样做的好处是，一旦基类的实现有错误，只要不涉及接口，那么基类的修改不会影响派生类的操作。**类与类之间，你做你的，我做我的，以接口作沟通。即使基类与子类也不例外。**这正是

类能够发挥其生命力的原因所在。

在构造一个子类时，完成其基类部分的构造由基类的构造函数去做，C++类的继承机制满意地提供了这种支持。

例如，下面的代码定义了研究生类，其中的构造函数实现对基类数据成员的构造：

```
class GraduateStudent :public Student
{
  public:
    GraduateStudent(char * pName, Advisor& adv)
                :Student(pName), advisor(adv)
    {
      qualifierGrade = 0;
    }
    //其余见 ch16_1.cpp
};

void fn(Advisor& advisor)
{
  GraduateStudent gs("Yen Kay Doodle", advisor);
  //...
}

int main()
{
  Advisor da;
  fn(da);
}
```

main()函数中创建了 Advisor 对象，以此为参数调用了函数 fn()。fn()中创建了 GraduateStudent 对象，初始化的参数为"Yen Kay Doodle"和 advisor。在构造函数原型的后面是 Student(pName),advisor(adv)，表示对数据成员初始化的方式。

基类初始化由 Student(pName)去完成，派生类的构造总是由基类的初始化开始的。由图 16-2 我们看到，基类在派生类对象中的空间位置也是如此。如果在上面初始化的方式中，描述顺序为：advisor(adv),Student(pName)，那么，派生类构造开始时，仍然是先调用基类的构造函数。如果 Student(pName)没有，则派生类会调用基类的默认构造函数，如果找不到匹配的构造函数，则就通不过编译。此处调用的构造函数是以 pName 为参数，而 pName 就是创建派生类对象时，由应用程序传递而来，它是派生类构造函数的形参。

基类的构造函数调用 Student(pName)产生一个 Student 的无名对象。派生类构造函数启动时，首先量好了派生类对象的大小尺寸，规定了对象分配的位置(给 this 赋值)，于是后面基类构造函数所产生的无名对象就构造在这个 this 指针指向处。

advisor 是 GraduateStudent 的数据成员，它是 Advisor 类对象。advisor(adv)表示以 adv 的值来初始化 advisor。因为参数 adv 为 Advisor 对象，所以 advisor(adv)将调用 Advisor 的默认拷贝构造函数(Advisor 自己没有定义拷贝构造函数之故)。

在派生类构造函数体内，"qualifierGrade =0;" 完成对其数据成员的赋值。

派生了的析构函数以构造函数相反的顺序被调用，所以函数 fn()返回时，系统开始处理 gs 对象的析构，先调用 GraduateStudent 的析构函数，执行析构函数体的代码，然后调用

Advisor 析构函数，最后调用 Student 析构函数。

→ 创建 fn()中 gs 对象时，参数 advisor 与 gs 对象中的数据成员 advisor 处在两个不同的对象空间，所以当 fn()函数返回时，析构 gs 之后，并没有把函数 fn()的形参 advisor 析构掉，由于该形参是传引用方式，所以 fn()返回时只是解除引用关系，对其不做其他处理。advisor 形参就是 main()函数中的 Advisor 对象 da。一直到 main()函数结束时，才由 Advisor 析构函数将 da 析构。

16.4 继承与组合

类以另一个类对象作数据成员，称为组合，如程序 ch16_1.cpp 中，GraduateStudent 类组合了 Advisor 类。这种场合，称 GraduateStudent 有一个 Advisor；而在继承的场合，称 GraduateStudent 是一个 Student。继承和组合都发挥了重用，它们将以前设计好的类采用“拿来主义”，但两者在使用上不同。GraduateStudent 类是 Student 类的一种，所以 GraduateStudent 享有 Student 的一切待遇。Advisor 含在 GraduateStudent 中，不是继承关系，Advisor 并不从属于 GraduateStudent。这种关系上的差别，决定了操作上的差别。例如下面的代码中，定义了一个派生类，该类中包含有类对象成员：

```
class Vehicle
{
  //...
};

class Motor
{
  //...
};
class Car :public Vehicle
{
  public:
    Motor motor;
};

void vehicleFn(Vehicle& v);
void motorFn(Motor& m);

int main()
{
  Car c;
  vehicleFn (c);      //ok
  motorFn(c);       //error
  motorFn(c.motor);       //ok
}
```

汽车是车辆的子类，它继承了车辆的所有特征。而汽车具有马达，如果拥有了一辆汽车，因为汽车包含有马达，所以，同时也拥有了一个马达。

调用 vehicleFn(c)是允许的，因为参数要求是车辆，而汽车是车辆的一种。

调用 motorFn(c)是不允许的，因为参数要求是马达，而汽车不是马达，所以参数 c 不能

匹配该函数。

调用 motorFn(c.motor)是允许的，因为参数要求是马达，而汽车中包含的马达就是 c.motor。

16.5 多态性

在程序 ch16_1.cpp 中，GraduateStudent 类对象 gs 调用 Student 类的成员函数 display()，该函数在输出时，没办法输出 GraduateStudent 自己的数据成员 qualifierGrade。因此在使用继承时，希望重载 display()。

C++允许子类的成员函数重载基类的成员函数。例如，下面的代码中，基类和派生类中都定义了计算学费的成员函数：

```
class Student
{
  public:
    //...
    float calcTuition(){      //计算学费
      //...
    }
};

class GraduateStudent :public Student
{
  public:
    //...
    float calcTuition(){
      //...
    }
};

int main()
{
  Student s;
  GraduateStudent gs;
  s.calcTuition();      //调用 Student::calcTuition()
  gs.calcTuition();     //调用 GraduateStudent::calcTuition()
}
```

派生类中重载了 calcTuition()成员函数，学生的学费计算与研究生的学费计算方法不同。学生对象 s 调用其学费计算函数显然指的是 Student:: calcTuition()成员，而研究生对象 gs 调用其学费计算函数时，两个重载函数都在它自己可使用范围内，C++编译规定：gs.calcTuition()指的是 GraduateStudent::calcTuition()。若派生类没有定义其 calcTuition()，则 gs.calcTuition()才指的是基类 Student::calcTuition()。

有时候，会碰到对象所属的类不能确定的情况。例如将上例作一改动：

```
class Student
{
  public:
    //...
```

```
    float calcTuition()      //计算学费
    {
      //...
    }
};

class GraduateStudent :public Student
{
  public:
    //...
    float calcTuition()
    {
      //...
    }
};

void fn(Student& x)
{
  x.calcTuition();
}

int main()
{
  Student s;
  GraduateStudent gs;
  fn(s);      //计算一下学生 s 的学费
  fn(gs);     //计算一下研究生 gs 的学费
}
```

因为学生和研究生都是学生，所以要求函数 fn()的形参应该能接纳这两种对象，这是继承机制的起码要求。但是接下来在函数 fn()中有一个问题：

以 fn(s)调用时，形参 x 为 Student 对象，x.calcTuition()则指 Student::calcTuition()。

以 fn(gs) 调用时，形参 x 为 GraduateStudent 对象，x.calcTuition()应该指的是 GraguateStudent::calcTuition()。

函数调用的实际参数，随应用程序的运行进展而变更，要求函数 fn()的行为也要随着变更。这样，函数 fn()的运行执行代码就无法在编译时被确定。它一会儿调用 Student::calcTuition()，一会儿又调用 GraduateStudent::calcTuition()。

C++的继承机制中用一种称为多态性(polymorphism)的技术来解决上面的问题。这种在运行时，能依据其类型确认调用哪个函数的能力，称为多态性，或称迟后联编(late binding)，也有的译为滞后联编。

编译时就能确定哪个重载函数被调用的，称为先期联编(early binding)。本节中前一个例子是先期联编的，后一个例子是迟后联编的。

16.6 多态的思考方式

若语言不支持多态，则不能称面向对象的。只支持类而不支持多态，只能称其语言为基于对象的。

上节的函数由于增加了研究生学费计算而使学费计算操作 calcTuition()呈多态。假如处理该函数的计算机无法分辨学生与研究生学费计算的不同，那么只有人为增加一段操作代码来弥补。

在 Student 类中增加一个 type 公共数据成员(以使普通函数能够访问它)，标识学生和研究生，分别修改 Student 和 GraduateStudent 的构造函数，对 type 成员赋初值，并在应用程序中，增加学生类别的判断，以便调用相应的学费计算操作：

```
enum StudentType{STUDENT, GRADUATESTUDENT};
class Student
{
  public:
    Student(char * pName = "no name")
    {
      strncpy(name, pName);
      average = semesterHours = 0;
      type = STUDENT;
    }
    float calcTuition();
    StudentType type;        //说明为公共的,便于应用程序访问
  //...
};

class GraduateStudent :public Student
{
  public:
    GraduateStudent()
    {
      type = GRADUATESTUDENT;      //覆盖刚赋的 STUDENT 值
    }
    float calcTuition();
  //...
};

void fn(Student& s)
{
  //...
  switch(s.type)      //判断对象的 type,以确定调用哪个学费计算成员
  {
    case STUDENT:
      s.Student::calcTuition(); break;
    case GRADUATESTUDENT:
      s.GraduateStudent::calcTuition(); break;
  }
  //...
}
```

像这种多态的成员也许不止一个，都要进行类似的处理。

何况，还会继承新的类，例如继承一个博士生类，这样，应用程序的维护量很大，类的内部实现和应用程序都不能避免修改，面向对象的优越性被遏制，又回到面向过程程序设计的老路上去了。因此，避免上述程序代码，实现多态技术，是面向对象程序设计的关键之一。

计算学费的学生管理员好像一个“迟后联编”者，别人关心的是他工作的内容，即计算所有学生的学费，并不关心：如果是大学生，则按某某计算公式计算学费；如果是研究生，则按某某计算公式计算学费，等等。具体的操作（各个成员函数）决定于学生管理员自己，该用什么计算公式取决于当时实际的学生性质。

对于学生管理员，这样的工作方式体现了其工作效率，他要比一个不知道区分学生性质的新手要强。另一方面，人与人之间的思想交流，只要不是做同一件事，一般总是在更高的抽象层面上，这样有利于更直接和精确地把握思考的事物。例如，一个人对另一个人吆喝“开门”，则彼此都处于同一场景中，明白是开机舱门还是开冰箱门等等。这就是人的思考问题的方式，也就是自然的多态方式。沿着这种思路设计的程序，可使软件模型更准确地描述人们所思考的问题。

16.7 多态性如何工作

为了指明某个成员函数具有多态性，用关键字 virtual 来标志其为虚函数。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch16_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
class Base
{
  public:
    virtual void fn()
    {
      cout <<"In Base class\n";
    }
};

class SubClass :public Base
{
  public:
    virtual void fn()
    {
      cout <<"In SubClass\n";
    }
};

void test(Base& b)
{
  b.fn();
}

int main()
{
  Base bc;
  SubClass sc;
  cout <<"Calling test(bc)\n";
  test(bc);
  cout <<"Calling test(sc)\n";
```

```
    test(sc);
}
```

运行结果为：

```
Calling test(bc)
In Base Class
Calling test(sc)
In SubClass
```

fn()是 Base 类的虚函数。在 test()函数中，b 是基类 Base 类的传引用形参，Base 类对象和 SubClass 类对象都可作为参数传递给 b，所以 b.fn()的调用要等到运行时，才能确认是调用基类的 fn()还是子类的 fn()。

由于 fn()标志为虚函数，编译看见 b.fn()后，将其作为迟后联编来处理，以保证在运行时确定调用哪个 fn()虚函数。

编译通常是在先期联编状态下工作的，只有看见虚函数，才把它作为迟后联编来实现。多态性增加了一些数据存储和执行指令的代价，但与所得到的编程灵活性和方便性相比，还是值得的。

fn()在基类中声明为 virtual，该虚函数的性质自动地向下带给其子类，所以 SubClass 子类中 virtual 可以省略。

例如，下面的程序进一步说明多态的使用。圆类和长方形类分别继承于形状 Shape 类，由一个函数专门负责求某圆或某长方形对象的面积：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch16_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<math.h>

class Shape
{
  public:
    Shape(double x, double y) :xCoord(x), yCoord(y){}
    virtual double Area() const {return 0.0;}
  protected:
    double xCoord, yCoord;
};

class Circle :public Shape
{
  public:
    Circle(double x, double y, double r) :Shape(x, y), radius(r){}
    virtual double Area() const {return 3.14 * radius * radius;}
  protected:
    double radius;
};

class Rectangle :public Shape
{
  public:
    Rectangle(double x1, double y1, double x2, double y2)
```

```
            :Shape(x1,y1), x2Coord(x2), y2Coord(y2){}
    virtual double Area() const;
  protected:
    double x2Coord, y2Coord;
};

double Rectangle::Area() const
{
  return fabs((xCoord - x2Coord) * (yCoord - y2Coord));
}

void fun(const Shape& sp)
{
  cout <<sp.Area() <<endl;
}

int main()
{
  Circle c(2.0,5.0,4.0);
  fun(c);
  Rectangle t(2.0,4.0,1.0,2.0);
  fun(t);
}
```

运行结果为：

```
50.24
2
```

fun()函数负责计算所有对象的面积，它知道只要调用求面积的 Area()函数就行了，不管参数是什么类型的对象，C++的迟后联编会做好这一切的。这样一来，fun()省了很多心，程序显得简单，以后要增加一个求新的形状的面积，只要简单地派生一个类就行了，应用程序不用作修改。

多态性让类的设计者去考虑工作的细节，而且这个细节简单到在成员函数上加一个 virtual 关键字。多态性使应用程序代码极大地简化，它是开启继承能力的钥匙。

16.8 不恰当的虚函数

如果虚函数在基类与子类中出现的仅仅是名字的相同，而参数类型不同，或返回类型不同，即使写上了 virtual 关键字，则也不进行迟后联编。

例如，下面的程序中，在派生类中重载了基类的成员函数，尽管标上了 virtual，但运行中起不到多态的效果：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch16_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Base
```

```
{
  public:
    virtual void fn(int x)
    {
      cout <<"In Base class, int x = " <<x <<endl;
    }
};

class SubClass :public Base
{
  public:
    virtual void fn(float x)
    {
      cout <<"In SubClass, float x = " <<x <<endl;
    }
};

void test(Base& b)
{
   int i = 1;
  b.fn(i);
   float f = 2.0;
   b.fn(f);
}

int main()
{
   Base bc;
  SubClass sc;
  cout <<"Calling test(bc)\n";
  test(bc);
  cout <<"Calling test(sc)\n";
  test(sc);
}
```

运行结果为：

```
Calling test(bc)
In Base class, int x = 1
In Base class, int x = 2
Calling test(sc)
In Base class, int x = 1
In Base Class, int x = 2
```

基类中的 void fn(int x)和子类中 void fn(float x)是两个不同的函数，它们无非是同名函数重载而已。SubClass 类继承了 Base 类中 void fn(int x)函数，尽管标识为 virtual，但它在基类与子类中都指的是同一个成员函数。SubClass 又自己添了一个成员 void fn(float x)，也标上了 virtual，但由于没有涉及同一个函数的多种行为，即没有涉及多态，所以对 test (Base& b) 函数中 b.fn(i) 成员函数调用的理解就十分简单。

编译看见 b.fn(i)的调用，分析到 fn(i)只有在 Base 类中定义的一个版本，无论是基类对象还是子类对象，调用的都是 Base::fn(int)成员，不存在多态，也就无须迟后联编了。

接下来编译又看见 b.fn(f)的调用。分析到子类对象也是基类对象，所以在 test (Base&

b)中，b 是作为 Base 类对象被匹配的，b.fn(f)只在 Base 类中寻求匹配。除非所匹配的函数是多态的，才考虑调用 b 的对象类型中的虚函数。编译在先期联编时，成功地将 fn(f)通过类型隐式转换，匹配了 Base::fn(int)。如果不能进行隐式转换而匹配 fn(int)，将以编译报错来回答。作为先期联编，b 是 Base 类对象，编译绝不会去尝试寻求和子类的 fn(float)匹配。

上述程序的编译分析与运行结果，关键是因为不恰当的虚函数，没有构成多态，而使编译看作为先期联编。有一种例外，如果基类中的虚函数返回一个基类指针或返回一个基类的引用，子类中的虚函数返回一个子类的指针或子类的引用，则 C++将其视为同名虚函数而进行迟后联编。

例如，下面的程序中，派生类与基类的成员函数返回类型不同，但仍起到虚函数的作用：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch16_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Base
{
  public:
    virtual Base* afn()
    {
      cout <<"This is Base class.\n";
      return this;
    }
};

class SubClass :public Base
{
  public:
    SubClass* afn()                    //实为虚函数                    *
    {
      cout <<"This is SubClass.\n";
      return this;
    }
};

void test(Base& x)
{
   Base* b;
   b = x.afn();
}

int main()
{
  Base bc;
  SubClass sc;
  test(bc);
  test(sc);
}
```

运行结果为：

```
This is Base class
This is SubClass
```

两个函数参数形式相同，返回类型不同，在调用时将体现不出差异。编译认为 Base * afn()和 SubClass * afn()是多态的，所以在看见 x.afn()的调用时进行了迟后联编。运行时视 x 对象的类型而确定调用哪个 afn()虚函数。

这个例外也是合理的，如果一个函数正处理 SubClass 的对象，则它仍可继续处理 SubClass 对象，这似乎是很自然的事。

16.9 虚函数的限制

一个类中将所有的成员函数都尽可能地设置为虚函数对编程固然方便，Java 语言中正是这样做的，但是会增加一些时空上的开销。对于 C++来说，在对性能上有偏激追求的编程中，只选择设置个别成员函数为虚函数。设置虚函数，须注意下列事项：

(1) 只有类的成员函数才能说明为虚函数。这是因为虚函数仅适用于有继承关系的类对象，所以普通函数不能说明为虚函数。

(2) 静态成员函数不能是虚函数，因为静态成员函数不受限于某个对象。例如，如果下列 staticfn()是静态成员函数：

```
void fn(Base& x)
{
  x.staticfn();      //只是用了 x 的类型信息，x 并不求值
  Base::staticfn();      //调用静态成员的推荐方法
}
```

(3) 内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时，仍将其看作非内联的。

(4) 构造函数不能是虚函数，因为构造时，对象还是一片未定型的空间。只有在构造完成后，对象才能成为一个类的名副其实的实例。

(5) 析构函数可以是虚函数，而且通常声明为虚函数。例如，当基类对象和子类对象以不同方式申请了堆空间后：

```
void finishWithObject(Base * pHeapObject)
{
  //...
  delete pHeapObject;
}
```

pHeapObject 是传递过来的一个对象指针，它或者指向基类对象，或者指向子类对象。在执行 delete pHeapObject 时，要调用析构函数，但是执行基类的析构函数？还是执行子类的析构函数？将析构函数声明为虚的，就可以解决这个问题。

16.10 类的冗余

继承给程序员可在一个过程中从不同类里组合共有特征的能力，这个过程称为分解(factoring)。也就是把共同的特征分解到基类中。

用分解银行存款的例子来说明，见图 16-3。

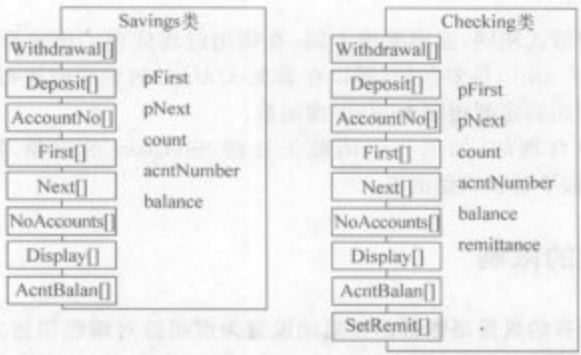


图 16-3　两个独立的银行存款类

图中有两个银行存款类，分别为 Savings(储蓄)类和 Checking(结算)类。该银行存款类是示意性的，实际在银行中的存款业务要远多于示例的内容。大方框代表类，类名在方框的顶部，小方框里面的名字是作为接口的公共成员函数，不在小方框里的名字是保护的数据成员。

所有的储蓄构成一个链表，所有的结算也构成独立的一个链表。链表中的结点个数即账户数(count)，链表首指针为 pFirst，结点中指向下一结点的指针为 pNext。银行存款一般有账号(acntNumber)和结算余额(balance)。对于结算类，还要有异地汇款的方式(remittance)，或信汇，或电汇，或其他转账结算方式。

对银行存款类的操作有 Deposit()(存款)，Withdrawal()(取款)，AccountNo()(取银行账号)，AcntBalance()(取账户中的余额)。

为了维护银行存款类中的链表，需要有操作：First()(取链表首指针)，Next()(取下一个结点)，NoAccounts()(取链表中的账户数)。

与储蓄存款不同，对于结算存款，取款若为信汇(邮寄汇款)或电汇(电报汇款)，则要加收一定的手续费。为此，设立数据成员 remittance(汇款方式)和成员函数 SetRemit()(设置汇款方式)，并且取款操作 Withdrawal()也与储蓄存款不同。

Savings 类和 Checking 类定义的代码分别为：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                savings.h                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#ifndef savings
#define savings

  class Savings
  {
    public:
      Savings(unsigned accNo, float balan = 0.0);
      unsigned AccountNo();
      float AcntBalan();
    static Savings * First();
```

```
    Savings * Next();
    Static int NoAccounts();
    void Display();
    void Deposit(float amount);
    void Withdrawal(float amount);
  protected:
    static Savings * pFirst;
    Savings * pNext;
    static int count;
    unsigned acntNumber;
    float balance;
};
#endif

//* * * * * * * * * * * * * * * * * * * *
//* *          savings.cpp              * *
//* * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "savings.h"

Savings * Savings::pFirst = 0;          //链表为空
int Savings::count = 0;                 //账户个数为0

Savings::Savings(unsigned accNo, float balan)
{
  acntNumber = accNo;
  balance = balan;

   count ++ ;
  if(pFirst == 0)
    pFirst = this;          //空链表中插入一个
  else
   {
    Savings * pS = pFirst
    for(pS -> pNext; pS = pS -> pNext);  //找到最后一个结点
    pS -> pNext = this;    //插入本结点
  }
  pNext = 0;          //本结点为最后一个结点
}

unsigned Savings::AccountNo()
{
   return acntNumber;
}

float Savings::AcntBalan()
{
   return acntBalance;
}

Savings * Savings::First()
{
   return pFirst;
```

```
}

Savings * Savings::Next()
{
   return pNext;
}

int Savings::NoAccounts()
{
   return count;
}

void Savings::Display()
{
  cout <<"Savings account number:" <<acntNumber
       <<" = " <<balance <<endl;
}

void Savings::Deposit(float amount)
{
   balance += amount;
}

void Savings::Withdrawal(float amount)
{
   if(balance<amount)
    cout <<"Insufficient funds:balance " <<balance
         <<", withdrawal"      <<amount <<endl;
   else
    balance -= amount;
}

//* * * * * * * * * * * * * * * * * * * * * * *
//* *          checking.h                    * *
//* * * * * * * * * * * * * * * * * * * * * * *

 # ifndef checking
 # define checking

 enum REMIT{remitByPost,remitByCable,other}; //信汇,电汇,其他

 class Checking
 {
    public:
     Checking(unsigned accNo, float balan = 0.0);
     int AccountNo();
     float AcntBalan();
     static Checking * First();
     Checking * Next();
     Static int NoAccounts();
     void Display();
     void Deposit(float amount);
     void Withdrawal(float amount);
     void SetRemit(REMIT re);
```

```
  protected:
    static Checking * pFirst;
    Checking * pNext;
    static int count;
    unsigned acntNumber;
    float balance;
    REMIT remittance;
};
#endif

//* * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* *                  checking.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "checking.h"

Checking * Checking::pFirst = 0;      //链表为空
int Checking::count = 0;              //账户个数为 0

Checking::Checking(unsigned accNo, float balan)
{
   acntNumber = accNo;
   balance = balan;
   remittance = other;      //设置汇款方式

   count ++ ;
   if(pFirst == 0)
    pFirst = this;          //空链表中插入一个
   else
   {
    checking * pS = pFirst;
    for(pS->pNext;pS = pS->pNext); //找到最后一个结点
    pS->pNext = this; //插入本结点
   }
   pNext = 0;          //本结点为最后一个结点
}

unsigned Checking::AccountNo()
{
   return acntNumber;
}

float Checking::AcntBalan()
{
   return acntBalance;
}

Checking * Checking::First()
{
   return pFirst;
```

```
}

Checking * Checking::Next()
{
  return pNext;
}

int Checking::NoAccounts()
{
   return count;
}

void Checking::Display()
{
   cout <<"Checking account number:" <<acntNumber
        <<" = " <<balance <<endl;
}

void Checking::Deposit(float amount)
{
   balance += amount;
}

void Checking::Withdrawal(float amount)
{
   float temp = amount;      //非信汇、电汇结算方式,则不收手续费
   if(remittance = remitByPost) //若信汇,则加收 30 元手续费
    temp = amount + 30;
   else if(remittance = remitByCable) //若电汇,则加收 60 元手续费
    temp = amount + 60;

   if(balance<temp)
    cout <<"Insufficient funds:balance " <<balance
         <<", withdrawal" <<temp <<endl;
   else
    balance -= temp;
}

void SetRemit::SetRemit(REMIT re)
{
   remittance = re;
}
```

我们注意到 Checking 和 Savings 类有许多相同的东西。但因为它们不完全相同,所以以分离的类形式出现。然而,应该有一种方法能避免这种重复。

16.11 克服冗余带来的问题

针对上节中的问题,我们让其中的一个类继承另一个类。Checking 有额外的数据成员,因此让它继承 Savings 类更合理些。如图 16-4 所示,该类通过增加数据成员 remittance 和成员函数 SetRemit()以及将成员函数 Withdrawal()设为虚函数的办法来完成。

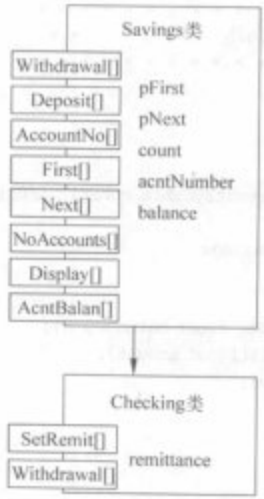


图 16-4　Checking 类作为 Savings 类的一个派生

如此一来，节省了打字的时间，Savings 和 Checking 类的界面实现可以如下设计：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              savings.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#ifndef savings
#define savings

  class Savings
  {
  public:
    Savings(unsigned accNo, float balan = 0.0);
    unsigned AccountNo();
    float AcntBalan();
    static Savings * First();
    Savings * Next();
    Static int NoAccounts();
    void Display();
    void Deposit(float amount);
    virtual void Withdrawal(float amount);   //虚函数
  protected:
    static Savings * pFirst;
    Savings * pNext;
    static int count;
    unsigned acntNumber;
    float balance;
  };
#endif
```

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              checking.h                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#ifndef checking
#define checking

  enum REMIT{remitByPost,remitByCable,other}; //信汇,电汇,其他

  class Checking :public Savings
  {
  public:
    Checking(unsigned accNo, float balan = 0.0);
    virtual void Withdrawal(float amount);
    void SetRemit(REMIT re);
  protected:
    REMIT remittance;
  };
#endif
```

Savings 类的内部实现不必修改,而 Checking 类的内部实现则简化为:

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              checking.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include "checking.h"

Checking::Checking(unsigned accNo, float balan)
        :Savings(accNo, balan)
{
  remittance = other;        //设置汇款方式
}

virtual void Checking::Withdrawal(float amount)
{
  float temp = amount;      //非信汇、电汇结算方式,则不收手续费
  if(remittance = remitByPost)   //若信汇,则加收 30 元手续费
    temp = amount + 30;
  else if(remittance = remitByCable)   //若电汇,则加收 60 元手续费
    temp = amount + 60;

  Savings::Withdrawal(temp);
}

void SetRemit(REMIT re)
{
  remittance = re;
}
```

从中看出,Checking 类只包含了与 Savings 类之间不同的部分。

尽管这样的解决可以省力,但不能令人完全满意。Checking 账户继承 Savings 账户的关系,暗示一个结算账户是储蓄账户的一个特殊类型,但事实上不是。

例如,银行改变了在储蓄账户上的政策。假设银行决定储蓄账户的余额可以出现允许

范围的负数(一定程度的透支)。根据这样的假定,在 Savings 账户中增加一个新数据成员 minbalance(透支范围),然后修改一下虚成员函数 Withdrawal()即可完成。

但是,问题来了。因为 Checking 类继承 Savings 类,Checking 类也得到了 minbalance 这个新数据成员。minbalance 对 Checking 类账户没用,一个额外的数据成员虽不算大,但增加了混淆。

这样的变化是日积月累的。一旦确定了类的层次关系,也就确定了系统的实现。今天增加一个额外的数据成员,明天又可能要改变成员函数。

这样的分类和继承关系能够正常工作,并且省力,这是事实。但是,类的层次必须体现其意义,否则会使程序员混淆。因为终究会有一天,一个不太熟悉该程序的程序员接手这一系统,将不得不阅读理解这些代码到底干什么。错误导向的技巧将更难理解和一致化。

16.12 类的分解

为了避免上节中的问题,可为 Savings 类和 Checking 类专门建立一个新类,并让这两个类都基于新类之上。不妨将该新类称之为 Account 类,该类包含了储蓄类和结算类所共有的特征,见图 16-5。

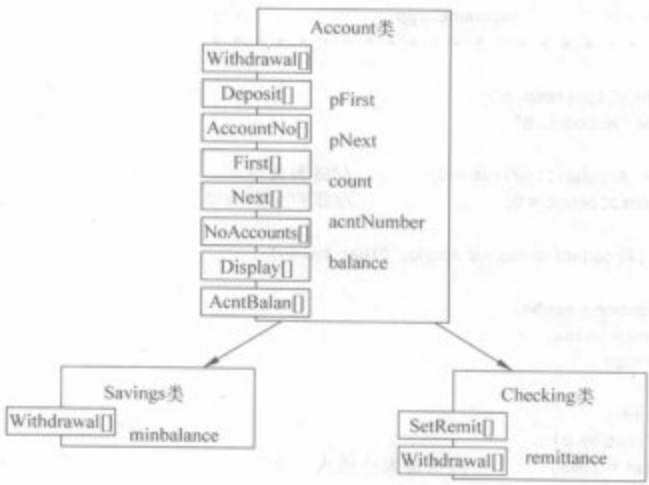


图 16-5 Account 类派生 Savings 和 Checking 类

在图中,Savings 类和 Checking 类的成员函数 Withdrawal()为虚函数。Savings 类继承了 Account 类,自身补充了 minbalance 数据成员。Checking 类继承了 Account 类,自身补充了 remittance 数据成员和 SetRemit()成员函数。相应的类代码描述为:

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              account.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

# ifndef account
# define account
```

```
class Account
{
   public:
     Account(unsigned accNo, float balan = 0.0);
     int AccountNo();
     float AcntBalan();
     static Account * First();
     Account * Next();
     Static int NoAccounts();
     void Display();
     void Deposit(float amount);
     virtual void Withdrawal(float amount);
   protected:
     static Account * pFirst;
     Account * pNext;
     static int count;
     unsigned acntNumber;
     float balance;
};
#endif

//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *                 account.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "account.h"

Account * Account::pFirst = 0;          //链表为空
int Account::count = 0;                 //账户个数为0

Account::Account(unsigned accNo, float balan)
{
   acntNumber = accNo;
   balance = balan;
   pNext = 0;

   count ++ ;
   if(pFirst == 0)
     pFirst = this;          //空链表中插入一个
   else
   {
     Account * pS = pFirst
     for(pS->pNext; pS = pS->pNext); //找到最后一个结点
     pS->pNext = this;    //插入本结点
   }
   pNext = 0;          //本结点为最后一个结点
}

int Account::AccountNo()
{
   return acntNumber;
}
```

```
float Account::AcntBalan()
{
   return balance;
}

Account * Account::First()
{
   return pFirst;
}

Account * Account::Next()
{
   return pNext;
}

int Account::NoAccounts()
{
   return count;
}

void Account::Display()
{
   cout <<"Account number:" <<acntNumber
        <<" = " <<balance <<endl;
}

void Account::Deposit(float amount)
{
   balance += amount;
}

void Account::Withdrawal(float amount)
{
   return;      //不做任何事
}

//* * * * * * * * * * * * * * * * * * * * * * *
//* *              savings.h                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#ifndef savings
#define savings

#include "account.h"

   class Savings :public Account
   {
     public:
       Savings(unsigned accNo, float balan = 0.0);
       virtual void Withdrawal(float amount);
      protected:
       static float minbalance;
};
#endif
```

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              savings.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "account.h"
#include "savings.h"

static Savings::minbalance = 500.0;      //设置透支范围

Savings::Savings(unsigned accNo, float balan)
          :Account(accNo, balan){}

void Savings::Withdrawal(float amount)
{
   if(balance + minbalance<amount)
    cout <<"Insufficient funds:balance " <<balance
         <<", withdrawal"      <<amount <<endl;
   else
    balance -= amount;
}

//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              checking.h                     * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#ifndef checking
   #define checking

   #include "account.h"

   enum REMIT{remitByPost,remitByCable,other}; //信汇,电汇,其他

   class Checking :public Account
   {
   public:
       Checking(unsigned accNo, float balan = 0.0);
       virtual void Withdrawal(float amount);
       void SetRemit(REMIT re);
   protected:
       REMIT remittance;
};
#endif

//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              checking.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "account.h"
#include "checking.h"

Checking::Checking(unsigned accNo, float balan)
          :Account(accNo, balan)
```

```
{
    remittance = other;      //设置汇款方式
}

virtual void Checking::Withdrawal(float amount)
{
    float temp = amount;      //非信汇、电汇结算方式,则不收手续费
    if(remittance = remitByPost)    //若信汇,则加收 30 元手续费
      temp = amount + 30;
    else if(remittance = remitByCable)    //若电汇,则加收 60 元手续费
      temp = amount + 60;

    if(balance<temp)
      cout <<"Insufficient funds:balance " <<balance
           <<", withdrawal" <<temp <<endl;
    else
      balance -= temp;
}

void checking::SetRemit(REMIT re)
{
    remittance = re;
}
```

这样的类层次更准确地描述了现实世界。现实中确实有账户这一概念,也确实有储蓄类账户和结算类账户之分。

而且,Savings 类和 Checking 类互不干涉。Checking 类不再受 Savings 类的影响,反之亦然。如果银行对账户类有所改变,则可以修改 Account 类,如果只改变结算类政策,则只需修改 Checking 类即可,而储蓄类保持不变。

从相似的类中,将共有特征提取出来的过程称为分解(factoring)。分解使类的层次合理化和减少冗余。只有当继承关系与实际相符合时,分解才是合理的。

一个程序员努力工作,写出比较巧妙的代码,以达到减少一些程序行的目的,是不值得的。这种巧妙常常会弄巧成拙。但是通过继承而分解出多余部分,可以合理地减少编程的工作量。

16.13 抽象类

分解带来了很多好处,但同时也产生了一个问题。在定义 Account 类的成员函数时,大多数成员函数不存在问题,因为两种账户都以相同的方式完成。

但 Withdrawal()就不同了。由于从一个储蓄账户中取款的操作与从一个结算账户中取款不同,即 Savings::Withdrawal()和 Checking::Withdrawal()的实现不同,那么 Account::Withdrawal()的定义该如何呢?

要新建一个账户,总是先要确定是储蓄账户还是结算账户,否则银行不知如何办理和不予办理。所有的账户要么是储蓄要么是结算账户,而一个 Account 账户仅仅是对这两个具体账户的共性进行分解而得到的一个抽象。Account 类是不完整的,因为它缺少具体账户的操作(专指 Withdrawal())。

对于我们人类，从中可以分出中国人、美国人、德国人、埃及人等，中国人中还可以分出汉族和各种少数民族。一个人，他(她)必定属于世界上的某个国家和某个民族。脱离国家和民族的"纯粹"的人是没有的。人类是我们创造的一个高度抽象的概念，并不存在人类本身的实例。

我们不希望程序员建立 Account 类或人类对象。因为我们并不知道用它做什么。为了解决这个问题，C++允许程序员声明一个不能有实例对象的类，这样的类唯一的用途是被继承。这种类称为抽象类(abstract class)。

一个抽象类至少具有一个纯虚函数。所谓纯虚函数(pure virtual function)是指被标明为不具体实现的虚成员函数。例如，我们并不知道怎样实现 Account::Withdrawal()。然而又不得不给它一个定义，像上节中 account.cpp 中描述的"不做任何事"那样，否则，C++认为是缺定义成员函数的连接错误。

声明一个函数是纯虚函数的语法，即让 C++知道该函数无定义，在 Account 类中示例如下：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              account.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#ifndef account
#define account

  class Account
  {
  public:
    Account(unsigned accNo, float balan = 0.0);
    int AccountNo();
    float AcntBalan();
    static Account * First();
    Account * Next();
    Static int NoAccounts();
    void Display();
    void Deposit(float amount);

    virtual void Withdrawal(float amount) = 0;    //纯虚函数

  protected:
    static Account * pFirst;
    Account * pNext;
    static int count;
    unsigned acntNumber;
    float balance;
};
#endif
```

在 Withdrawal()的声明之后的"=0"表明程序员将不定义该函数。该声明是为派生类而保留的位置。Account 的派生类被期待用一个具体的函数来重载该函数。

一个抽象类不能有实例对象，即不能由该抽象类来制造一个对象。所以，下面的声明是非法的：

```
void func()
{
   Account a(3145, 300.0);        //企图建立对象：账号为 3145,金额为 300 元
   a.Withdrawal(100);             //企图取 100 元
}
```

假如定义 Account 类对象允许，其结果是不完备的，缺少取款能力，因为并不存在 Account::Withdrawal()的具体行为。

抽象类是作为基类为其他类服务的。一个 Account 类包含一个银行账户的所有特征。可以通过继承 Account，来创建其他类型的银行账户类，但是 Account 自身不能有实例对象。

16.14 由抽象类派生具体类

Savings 类不是抽象类，因为它用一个完全实在的定义对纯虚函数 Withdrawal()进行了重载。Savings 类的一个对象知道在 Withdrawal()被调用时怎样执行。同样，Checking 类也如此，该类不是虚的，因为成员函数 Withdrawal()重载了基类中的纯虚函数。

知道所有纯虚函数被重载之前，抽象类的子类也一直保持抽象状态。例如：

```
class Display      //显示器类
{
  public:
    virtual void init() = 0;       //纯虚函数
    virtual void write(char * pString) = 0;       //输出字符串
};

class Monochrome :public Display      //单色显示器类
{
   virtual void init();
   virtual void write(char * pString);
};

class ColorAdapter :public Display      //彩色显示器类
{
   virtual void write(char * pString);
};

class SVGA :public ColorAdapter      //VGA 系列彩色显示器类
{
   virtual void init();
};

void Monochrome::init()
{
   //省略
}

void Monochrome::write(char * pString)
{
   //省略
}
```

```
void ColorAdapter::write(char * pString)
{
   //省略
}

void SVGA::init()
{
   //省略
}
void func()
{
   Monochrome mc;
   VGA vga;
}
```

Display 类用来表述 PC 的显示器。它分别有两个纯虚函数：init()和 write()。对不同显示器，init()(初始化)和 write()(写屏幕)的操作是不同的。所以 Display 类无法实现这两个成员函数，将其设计为抽象类。

Display 类的子类 Monochrome 不是抽象类。这是一个特定的显示器类型，程序员知道该如何编程。因此，定义了 init()和 write()这两个虚函数。

SVGA 类也不是抽象类，程序员也知道怎样对该显示器编程。然而，在介于 Display 类和 SVGA 类之间，有一个 ColorAdapter 类，SVGA 是所有彩色显示器的一个特例。该类能够确定如何写屏(实现了成员函数 write())，这里假定了所有的彩色显示器都以相同的方法进行写屏。但对不同的彩色显示器，还是不能决定如何初始化。因此其 init()成员函数从 Display 类中继承过来，还是纯虚函数。所以，它也仍然是抽象类。

SVGA 类中实现了 init()成员函数，使之不再具有纯虚函数，所以，SVGA 是可以创建具体对象的实在类。

16.15 纯虚函数的需要性

不能创建一个抽象类的对象，但是可以声明一个抽象类的指针或引用。

例如，下面的代码，设计了一个专门处理账户的函数：

```
void func(Account * pA) //专门处理账户中的取款
{
   pA->Withdrawal(100.0);
}

void otherFunc()
{
   Savings s;
   Checking c;
   func(&s);      //合法：因为一个 Savings 是一个 Account
   func(&c);      //合法：一个 Checking 也是一个 Account
}
```

在这里，函数 func()的参数 pA 是一个 Account 指针。从 otherFunc()函数调用 func()

时，传递的实参都是具有实际地址的子类对象。它们要么是 Savings 类对象，要么是 Checking 类对象，但决不可能是 Account 类对象，因为在 otherFunc()函数中，决不可能创建 Account 对象。

在分解银行存款的例子中，**如果在 Account 类里去掉 Withdrawal()纯虚函数，使得在其子类中分别添加 Withdrawal()成员函数的定义，程序仍能通过编译和连接**，在某些情况下也能运行。

但若考虑上例多态的情形，则下例的代码将不能通过编译：

```
class Account
{
  //除了不声明 Withdrawal()外，其他都一样
};
class Savings :public Account
{
  public:
   virtual void Withdrawal(float amount);
};

//...

void func(Account * pA)
{
  pA->Withdrawal(100.0);   //error:Withdrawal()不是 Account 的成员
}

int main()
{
  Savings s;
  func(&s);
}
```

C++是强类型语言。当访问一个成员函数时，C++坚持要证明该成员函数在类中存在。否则，拒绝接受。在上例的 func()函数中，pA 指针是 Account 类的，在编译时，就将验证所指向的函数 Withdrawal()是否为其成员。如果 Withdrawal()是 Account 的成员函数，即使是纯虚函数，也能顺利通过编译。这正是纯虚函数为什么非要不可的原因所在。

纯虚函数是在基类中为子类保留的一个位置，以便子类用自己的实在函数定义来覆盖之。如果在基类中没有保留位置，则就没有覆盖。

小结

C++支持类的继承机制。继承是面向对象设计的关键概念之一。有了继承，才使面向对象程序设计真正进入了实用。

派生类可以继承基类的所有公有和保护的数据成员和成员函数。

保护的访问权限对于派生类来说是公有的，而对于其他的对象来说是私有的。即使是派生类也不能访问基类中私有的数据成员和成员函数。

构造函数可以在它的函数体之前进行初始化。此时调用基类的构造函数并把参数传递

给它。

在派生类中允许重载基类的成员函数。如果基类中的函数是虚函数，当使用指针或引用访问对象时，将基于实际运行时指针所指向的对象类型来调用派生类的函数。

通过加上类名和双冒号作为前缀，可以显式的调用基类中的成员。

纯虚函数是一个没有定义函数语句的基类虚函数，纯虚函数的值是 0。派生类必须为每一个基类纯虚函数提供一个相应的函数定义。

练习

16.1 根据 16.12 节中所定义的 Account 类、Savings 类和 Checking 类，编写一个应用程序，它读入一系列账号和存款，创建若干储蓄和结算账户，直到碰到一个标志结束的符号。并输出所有账号的存款数据。

16.2 根据 16.12 节中所定义的 Account 类、Savings 类和 Checking 类，编写一个应用程序，它取出一系列账号的存款，直到碰到一个标志结束的符号。要求程序用多态的方法实现，并输出取出的账号和金额数。

16.3 将习题 16.1 和 16.2 中的应用程序设计成函数，设计一个菜单，选择处理储蓄和结算账户；还需在子菜单中选择处理存款和取款业务，并分别调用上面设计的两个函数。

16.4 用多文件程序结构实现习题 16.3，画出其类层次图。

16.5 信用卡是储蓄类的一种，假设它可以在 5000 元范围内透支，它有一个用户密码，取款时，必须验证密码。从 Account 账户类体系中继承一个信用卡类，然后编制应用程序，实现取款和存款业务，并将其纳入习题 16.3 程序的菜单之中。

16.6 定期储蓄是储蓄的一种，假设定期分一年期、三年期和五年期，利率分别为 5%，8% 和 10%。用户在办理定期存款账户时，必须确定其定期时段，中途不再在同一账号上办理存款业务。取款是一次性完成，若提前取款，则全部金额的利息按活期利率 1%计算。将其银行业务纳入习题 16.3。

16.7 专用存款是结算类账户的一种，银行为了监督专用存款的使用，特地将该存款另设账户进行管理。专用存款的存取业务与结算类账户相同。从银行账户体系中派生专用存款账户，并将其应用纳入习题 16.3 中。

第17章　多重继承

我们可以为一个派生类指定多个基类，这样的继承结构被称为多重继承或多继承。通过学习本章，应能理解多继承的工作原理，了解多继承要解决的问题，认识虚拟继承的实质，把握多继承的方法，并能够简单地从多个基类中派生出新类。

17.1　多继承如何工作

到目前为止，所讨论的类层次中，每个类只继承一个父辈，在现实世界中事情通常是这样的。但是一些类却代表两个类的合成。例如，两用沙发，它是一个沙发，也是一张床，两用沙发应允许同时继承沙发和床的特征，即 SleeperSofa 继承 Bed 和 Sofa 两个类，如图 17-1 所示。其程序代码如下：

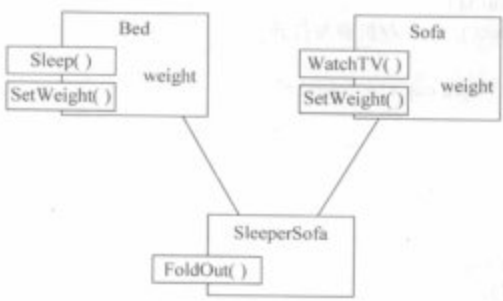


图 17-1　两用沙发的类层次

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch17_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class Bed
{
  public:
    Bed() :weight(0){}

    void Sleep()
    {
      cout <<"Sleeping...\n";
    }

  void SetWeight(int i){weight = i;}
  protected:
    int weight;
```

```
};

class Sofa
{
   public:
     Sofa() :weight(0){}

     void WatchTV()
     {
       cout <<"Watching TV.\n";
     }

     void SetWeight(int i){weight = i;}
   protected:
     int weight;
};

class SleeperSofa :public Bed, public Sofa
{
   public:
     SleeperSofa(){}
     void FoldOut()      //折叠与打开
     {
       cout <<"Fold out the sofa.\n";
     }
};

int main()
{
   SleeperSofa ss;
   ss.WatchTV();
   ss.FoldOut();
   ss.Sleep();
}
```

运行结果为：

```
Watching TV.
Fold out the sofa.
Sleeping...
```

两用沙发继承两个基类的所有成员，这样 ss.Sleep()和 ss.WatchTV()的调用是合法的。也就可以把 SleeperSofa 当作一个 Bed 或一个 Sofa 用。另外，SleeperSofa 类还有它自己的成员 FoldOut()。

17.2 继承的模糊性

在上节中，Sofa 和 Bed 都有一个成员 weight，这是合理的，因为两者都是实体，都有一个重量。问题是 SleeperSofa 继承哪个重量？既然两者都继承，由于两者有相同的名字 weight，使得对 weight 的引用变得模糊不清。

例如，按照下面引用：

```
int main()
{
   SleeperSofa ss;
   ss.SetWeight(20);      //Bed 的 SetWeight 还是 Sofa 的 SetWeight?
}
```

这样导致了名称冲突(name collision)，在编译时将予以拒绝。

程序必须在重量前面说明基类：

```
int main()
{
   SleeperSofa ss;
   ss.Sofa.SetWeight(20);      //说明是沙发重量 20 斤
}
```

在编写应用程序时，程序员还得额外知道类的层次信息，加大了复杂度。这些在单继承中是不会出现的。

17.3 虚拟继承

从意义上来看，一个 SleeperSofa 没有沙发和床两种重量，如此的继承不是真实的现实世界描述。进一步分析可得，床和沙发都是家具的一种，凡家具都有重量，所以通过分解来考察其关系，图 17-2 所示。

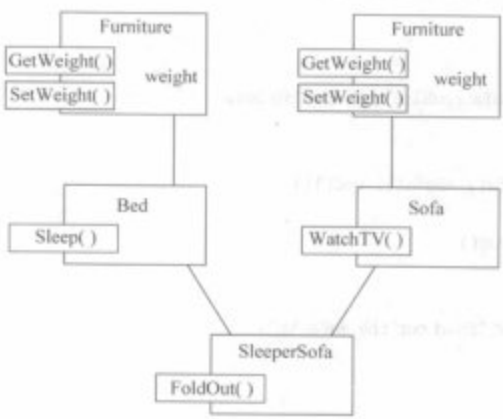


图 17-2　床和沙发的分解

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch17_2.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class Furniture
```

```
{
   public:
     Furniture(){}
     void SetWeight(int i){weight = i;}
     int GetWeight(){return weight;}
   protected:
     int weight;
};

class Bed :public Furniture
{
   public:
     Bed(){}

     void Sleep()
     {
       cout <<"Sleeping...\n";
     }
};
class Sofa :public Furniture
{
   public:
     Sofa(){}

     void WatchTV()
     {
       cout <<"Watching TV.\n";
     }
};

class SleeperSofa :public Bed, public Sofa
{
   public:
     SleeperSofa() :Sofa(), Bed(){}

     void FoldOut()

     {
       cout <<"Fold out the sofa.\n";
     }
};

int main()
{
   SleeperSofa ss;
   ss.SetWeight(20);          //编译出错!模糊的 SetWeight 成员
   Furniture * pF;
   pF = (Furniture * ) &ss;          //编译出错!模糊的 Furniture *
   cout <<pF->GetWeight() <<endl;
}
```

因为 SleeperSofa 不是直接继承 Furniture，而是 Bed 和 Sofa 各自继承 Furniture，所以

完整的 SleeperSofa 对象内存布局如图 17-3 所示。

图 17-3　完整 SleeperSofa 对象内存布局

这里一个 SleeperSofa 包括一个完整的 Bed，随后还有一个完整的 Sofa，后面还有一个 SleeperSofa 特有的东西。SleeperSofa 中的每一个子对象都有它自己的 Furniture 部分。因为每个子对象继承 Furniture，所以一个 SleeperSofa 包含两个 Furniture 对象，实际上的继承层次如图 17-4 所示。

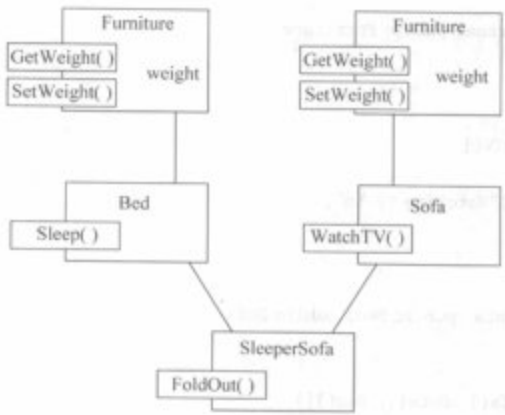


图 17-4　SleeperSofa 的实际继承关系

编译 ch17_2.cpp 时，不知道 SetWeight()属于哪一个 Furniture 成员，指向 Furniture 的指针也不知道究竟指哪一个 Furniture。这就是为什么 ch17_2.cpp 编译通不过的原因。

SleeperSofa 只需一个 Furniture，所以我们希望它只含一个 Furniture 拷贝，同时又要共享 Bed 和 Sofa 的成员函数与数据成员，C++实现这种继承结构的方法称为虚拟继承(virtual inheritance)。

下面是虚拟继承的代码：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch17_3.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
```

```
class Furniture
{
   public:
     Furniture(){}
     void SetWeight(int i){weight = i;}
     int GetWeight(){return weight;}
   protected:
     int weight;
};

class Bed :virtual public Furniture
{
   public:
     Bed(){}
     void Sleep()
     {
       cout <<"Sleeping...\n";
     }
};

class Sofa :virtual public Furniture
{
   public:
     Sofa(){}
     void WatchTV()
     {
       cout <<"Watching TV.\n";
     }
};

class SleeperSofa :public Bed, public Sofa
{
   public:
     SleeperSofa() :Sofa(), Bed(){}
     void FoldOut()
     {
       cout <<"Fold out the sofa.\n";
     }
};
int main()
{
   SleeperSofa ss;
   ss.SetWeight(20);
   cout <<ss.GetWeight() <<endl;
}
```

运行结果为：

```
20
```

在 Bed 和 Sofa 继承 Furniture 中加上 virtual 关键字，这相当于说，“如果还没有 Furniture 类，则加入一个 Furniture 拷贝，否则就用有的那一个。”此时一个 SleeperSofa 在内存中的布局见图 17-5。

图 17-5　虚拟继承的 SleeperSofa 内存布局

在虚拟继承的情况下，应用程序 main()中引用 GetWeight()不再模糊，我们得到了真正的图 17-2 所示的继承关系。

→ 虚拟继承的虚拟和虚拟函数的虚拟没有任何关系。

17.4　多继承的构造顺序

构造对象的规则需要扩展以控制多重继承。构造函数按下列顺序被调用：

(1) 任何虚拟基类的构造函数按照它们被继承的顺序构造；

(2) 任何非虚拟基类的构造函数按照它们被继承的顺序构造；

(3) 任何成员对象的构造函数按照它们声明的顺序调用；

(4) 类自己的构造函数。

例如：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch17_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class OBJ1
{
  public:
    OBJ1(){ cout <<"OBJ1\n";}
};

class OBJ2
{
  public:
    OBJ2(){ cout <<"OBJ2\n";}
};

class Base1
{
```

```
    public:
      Base1(){ cout <<"Base1\n";}
};

class Base2
{
   public:
     Base2(){ cout <<"Base2\n";}
};

class Base3
{
   public:
     Base3(){ cout <<"Base3\n";}
};

class Base4
{
   public:
     Base4(){ cout <<"Base4\n";}
};

class Derived :public Base1, virtual public Base2,
               public Base3, virtual public Base4
{
   public:
     Derived() :Base4(), Base3(), Base2(),
                Base1(), obj2(), obj1()
     {
       cout <<"Derived ok.\n";
     }
   protected:
     OBJ1 obj1;
     OBJ2 obj2;
};

void main()
{
   Derived aa;
   cout <<"This is ok.\n";
}
```

运行结果为：

```
Base2
Base4
Base1
Base3
OBJ1
OBJ2
Derived ok.
This is ok.
```

Derived 的虚基类 Base2 和 Base4 最先构造，尽管它在 Derived 类中出现的顺序不在最

前面；Derived 的非虚基类其次构造，不管它在 Derived 构造函数中出现的顺序如何；Derived 的组合对象 obj1 和 obj2 随后构造，它以类定义时，数据成员排列顺序为准，不管在 Derived 构造函数中出现顺序怎样；最后是 Derived 类构造函数本身。

→ 在语言中实现多继承并不容易，这主要是编译程序问题，还有模糊性问题。建议你如果可能，在进一步阅读有关参考书之前，尽量避免用多重继承。单个继承提供了足够强大的功能，不一定非用多重继承不可。我们应先学会阅读一些商品化的类库源程序中有关多重继承的部分，因为那些都是经过测试的安全代码。

17.5 继承的访问控制

继承可以公共继承，也可保护继承和私有继承。对于不同的继承方式，其访问控制是不同的。表 17-1 列出了其中的区别。

表 17-1 存取方式与继承的关系

存取方式 / 继承类型	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

我们在前面讨论的类的继承关系都是公共继承的。在公共继承的类中，基类的每个成员在子类中保持同样的访问方式。即在基类中为 public 的成员，子类中可以访问之，并据为 public 的；基类中为 protected 的成员，子类中也可访问之，并据为 protected 的；基类中为 private 的成员，在子类中不能访问之，这就像在应用程序中不能访问类中私有成员一样。

下面的例子中的代码，是对上表的一个解释：

```
class Base
{
  public:
    int m1;
  protected:
    int m2;
  private:
    int m3;
};

class PrivateClass :private Base      //私有继承
{
  public:
    void test()
    {
      m1 = 1;      //ok: 将 m1 据为 private
      m2 = 2;      //ok: 将 m2 据为 private
      m3 = 3;      //不可访问
    }
```

```
};
class DerivedFromPri :public PrivateClass
{
  public:
    void test()
    {
      m1 = 1;      //不可访问基类的私有成员
      m2 = 2;      //不可访问
      m3 = 3;      //不可访问
    }
};

class ProtectedClass :protected Base      //保护继承
{
  public:
    void test()
    {
      m1 = 1;      //m1 据为 protected
      m2 = 2;      //m2 据为 protected
      m3 = 3;      //不可访问
    }
};

class DerivedFromPro :public ProtectedClass
{
  public:
    void test()
    {
      m1 = 1;      //m1 仍为 protected
      m2 = 2;      //m2 仍为 protected
      m3 = 3;      //不可访问
    }
};

class PublicClass :public Base      //公共继承

{
  public:
    void test()
    {
      m1 = 1;      //m1 为 public
      m2 = 2;      //m2 为 protected
      m3 = 3;      //不可访问
    }
};

class DerivedFromPub :public PublicClass
{
  public:
    void test()
    {
      m1 = 1;      //m1 仍保持为 public
      m2 = 2;      //m2 仍保持为 protected
```

```
        m3 = 3;      //不可访问
    }
};

int main()
{
    PrivateClass priObj;
    priObj.m1 = 1;      //error
    priObj.m2 = 2;      //error
    priObj.m3 = 3;      //error

    ProtectedClass proObj;
    proObj.m1 = 1;      //error
    proObj.m2 = 2;      //error
    proObj.m3 = 3;.     //error

    PublicClass pubObj;
    pubObj.m1 = 1;
    pubObj.m2 = 2;      //error
    pubObj.m3 = 3;      //error
}
```

类 Base 包括三个成员 m1，m2，m3，分别定义为 public，protected，private。Base 作为 PrivateClass，ProtectedClass，PublicClass 三个类的基类，这三个类分别用私有、保护和公共继承派生。

于是在这三个类中，数据成员的性质发生了变化：私有继承时，基类中不管是公有的，还是保护的或者为私有的，一律在子类中变成私有成员。保护继承时，基类中公共和保护的成员在子类中变成保护的，而基类中私有的成员在子类中变成私有的。公共继承时，基类中为公共、保护和私有的成员在子类中仍保持为公共、保护和私有的。

m3 是私有的，它只能被 Base 访问，不能被派生类和非成员函数访问。在所有的 test() 函数中引用 m3 时将报错。m2 和 m3 对直接从 Base 派生的类都可以访问。所以由 PrivateClass::test()，ProtectedClass::test() 和 PublicClass::test() 引用这两个成员是允许的。

PrivateClass 类私有继承 Base，这意味着 m1 和 m2 对 PrivateClass 中的成员现在可以私有访问。因而不能被像 DerivedFromPri 一样由 PrivateClass 直接派生的类访问。

ProtectedClass 类被保护地继承 Base，这使得 m1 和 m2 都被保护。对 ProtectedClass::test() 和 DerivedFromPro::test() 这两个成员都是可访问的。

PublicClass 类公共继承 Base。在 PublicClass 中三个成员都保持它们在 Base 中的访问特性。

如果不标明继承为公共还是保护或者私有，则默认的继承是私有。但最好不要依赖默认，如果把继承类型表示清楚，则程序的可读性会更好。

在单个类中，protected 和 private 没有什么区别。**但在继承关系中，基类的 private 成员不但对应用程序隐藏，甚至对派生类也隐藏。而基类的保护成员则只对应用程序隐藏，而对派生类则毫不隐瞒。**

17.6 保护继承与私有继承

一个私有的或保护的派生类不是子类，因为非公共的派生类不能做基类能做的所有的事。例如，下面的代码定义了一个私有继承基类的类：

```
#include <iostream.h>

class Animal
{
   public:
     Animal(){}
     void eat(){ cout <<"eat\n"};
};

class Giraffe :private Animal
{
   Giraffe(){}
   void StrechNeck(double){ cout <<"strech neck\n"};
};

class Cat :public Animal
{
   Cat(){}
   Void Meaw(){ cout <<"meaw\n"};
};

void Func(Animal& an)
{
   an.eat();
}

int main()
{
   Cat dao;
   Giraffe gir;
   Func(dao);
   Func(gir);      //error
}
```

函数 Func()要用一个 Animal 类型的对象，但调用 Func(dao)实际上传递的是 Cat 类的对象。因为 Cat 是公共继承 Animal 类，所以 Cat 类对象拥有 Animal 的所有成员的使用。Animal 对象可以做的事，Cat 对象也可以做。

但是，对于 gir 对象就不一样。Giraffe 类私有继承了 Animal 类，意味着对象 gir 不能直接访问 Animal 类的成员。其实，在 gir 对象空间中，包含有 Animal 类的对象，只是无法让其公开访问。

公有继承就像是三口之家的小孩，饱受父母的温暖，享有父母的一切(public 和 protected 的成员)。其中保护的成员不能被外界所享有，但可以为小孩所拥有。只是父母还是有其一点点隐私(private 成员)不能为小孩所知道。

私有继承就像是离家出走的小孩，一个人在外面飘泊。他(她)不能拥有父母的住房和

财产(如 gir. eat()是非法的),在外面自然也就不能代表其父母,甚至他(她)不算是其父母的小孩(如 Func(gir)调用被禁)。但是在他(她)的身体中,流淌着父母的血液(在 gir 对象空间中,含有 Animal 对象),所以,在小孩自己的行为中又有其与父母相似的成分(可以通过自身成员函数访问父类私有成员)。

例如,下面的代码中,Giraffe 继承了 Animal 类,Giraffe 的成员函数可以访问像 Animal 对象那样访问其 Animal 成员:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                 ch17_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Animal
{
   public:
     Animal(){}
     void eat(){ cout <<"eat.\n";}
};

class Giraffe :private Animal
{
  public:
     Giraffe(){}
     void StretchNeck(){ cout <<"stretch neck.\n";}
     void take()
   {
     eat();          //ok
   }
};

void Func(Giraffe & an)
{
   an.take();
}

int main()
{
   Giraffe gir;
   gir.StretchNeck();
   Func(gir);       //ok
}
```

运行结果为:

```
stretch neck.
eat.
```

上例中,gir 对象就好比是小孩。eat()成员函数是其父母的行为,take()成员函数是小孩的行为,在该行为中,渗透着父母的行为。但是小孩无法直接使用 eat()成员函数,因为,离家出走的他(她)无法拥有其父母的权力。

保护继承与私有继承类似,继承之后的类相对于基类来说是独立的;保护继承的类对象,在公开场合同样不能使用基类的成员;

```
#include <iostream.h>

class Animal
{
   public:
     Animal(){}
     void eat(){ cout <<"eat\n"};
};

class Giraffe :private Animal
{
   Giraffe(){}
   void StrechNeck(double){ cout <<"strech neck\n"};
   void take()
   {
     eat();      //ok
   }
};

int main()
{
   Giraffe gir;

   gir.eat();      //error
   gir.take();      //ok
   gir.StretchNeck();
}
```

小结

C++支持多重继承，从而大大增强了面向对象程序设计的能力。

多重继承是一个类从多个基类派生而来的能力。派生类实际上获取了所有基类的特性。

当一个类是两个或多个基类的派生类时，必须在派生类名和冒号之后，列出所有基类的类名，基类间用逗号隔开。

派生类的构造函数必须激活所有基类的构造函数，并把相应的参数传递给它们。

派生类可以是另一个类的基类，这样，相当于形成了一个继承链，当派生类的构造函数被激活时，它的所有基类的构造函数也都会被激活。

在面向对象的程序设计中，继承和多重继承一般指公共继承。在无继承的类中，protected和private控制符是没有差别的。在继承中，基类的private对所有的外界都屏蔽(包括自己的派生类)，基类的protected控制符对应用程序是屏蔽的，但对其派生类是可访问的。

保护继承和私有继承在类接口安全技术讨论上有其一席之地。

练习

17.1 给定如图17-6所示的继承图，写出程序代码，在应用程序中，建立C类对象，访问A类中的成员函数以设置和读取其数据成员。

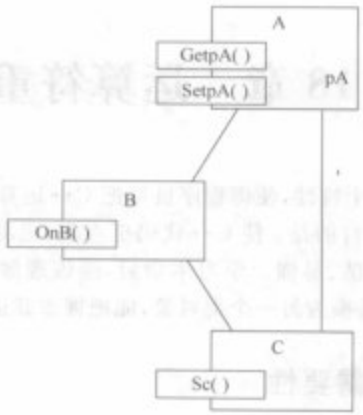


图 17-6　继承图

17.2　一个三口之家，大家都知道其父亲会开车，母亲会唱歌。但是其父亲还会修电视机，只有家里人知道。小孩既会开车又会唱歌甚至也会修电视机。母亲瞒着任何人在外面做小工以补贴家用。此外小孩还会打乒乓球。

编制程序，让这三口之家从事一天的活动：先是父亲出去开车，然后母亲出去工作(唱歌)，母亲下班后去做两小时小工。小孩在俱乐部打球，在父亲回家后，开车玩，后又高兴地唱歌。晚上，小孩和父亲一起修电视机。

后来父亲的修电视机技术让大家知道了，人们经常上门要他修电视机。这时，程序要作什么样的变动。

第 18 章　运算符重载

重载运算符是 C++的一个特性，使得程序员可把 C++运算符的定义扩展到运算分量是对象的情况。运算符重载的目的是：使 C++代码更直观，更易读。由简单的运算符构成的表达式常常比函数调用更简洁、易懂。学习本章后，应该理解怎样重定义与类有关的运算符，学会怎样把一个类对象转换为另一个类对象，能把握重载运算符的时机。

18.1　运算符重载的需要性

C++认为用户定义的数据类型就像基本数据类型 int 和 char 一样有效。运算符(如＋，－，＊，/)是为基本数据类型定义的，为什么不允许使之适用于用户定义类型呢？例如：

```
class A
{
   int a;
   public:
    A(int x)
    {
      a = x;
    }
   //
};

A a ( 5 ), b ( 10 ), c;
c = a + b;      //类对象也应能运算
```

运算符重载可以改进可读性，但不是非有不可。

下列例子计算应付给的人民币，分别用了成员函数和运算符成员函数两种方法：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_1.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class RMB      //人民币类
{
   public:
     RMB(doubled) {yuan = d; if = (d - yuan)/100;}
     RMB interest(double rate);      //计算利息
     RMB add(RMB d);       //人民币加
     void display()
     {
        cout <<(yuan + jf / 100.0) << endl;
     }
```

```
    RMB operator + (RMB d) {return RMB(yuan + d.yuan + (jf + d.if)/100); }
    //人民币加的运算符重载
    RMB operator * (double rate) {return RMB((yuan + jf/100) * rate);}
  private:
    unsigned int yuan;      //元
    unsigned int jf;      //角分
};

RMB RMB::interest(double rate)
{
   return RMB((yuan + jf / 100.0) * rate);
}
RMB RMB::add(RMB d)
{
   return RMB(yuan + d.yuan + jf / 100.0 + d.jf / 100.0);
}

//以下是计算应付人民币的两个版本
RMB expense1(RMB principle, double rate)
{
   RMB interest = principle.interest(rate);
   return principle.add(interest);
}

RMB expense2(RMB principle, double rate)
{
   RMB interest = principle * rate;      //本金乘利息
   return principle + interest;      //连本带利
}

int main()
{
   RMB x = 10000.0;
   double yrate = 0.035;
   expense1(x,yrate).display();
   expense2(x,yrate).display();
}
```

运行结果为：

```
10350
10350
```

expense() 的两个版本都可以计算应付人民币，运行结果相同。expense2()可读性更好一点，它符合我们计算用+，* 运算符的习惯。

如果不定义运算符重载，则 expense2() 中 principle * rate 和 principle + interest 是非法的。因为参加运算的是类对象而不是浮点值。

18.2 如何重载运算符

运算符是在 C++系统内部定义的，具有特定语法规则，如参数说明、运算顺序、优先级等。重载运算符时，要注意该重载运算符的运算顺序和优先级不变，如下例：

```
class A
{
  public:
    A(int n)
    {
      //...
    }
    friend A operator + (A&, A&)
    {
      //...
    }
    friend A operator * (A&, A&)
    {
      //...
    }

    //...
};

A a = 5, b = 6, c = 7, d = 8, e;
e = a + b * c + d;      // 即 ( a + ( b * c ) ) + d
```

有了运算符，编程就显得方便。例如，对于直角三角形斜边长度公式 $c=\sqrt{a^2+b^2}$，用函数化的格式表示：

```
c = sqrt ( add ( mult ( a, a ), mult ( b, b ) ) );
```

用运算符的格式表示：

```
c = sqrt ( a * a + b * b );
```

运算符是函数，除了运算顺序和优先级不能更改外，参数和返回类型是可以重新说明的，即可以重载。重载的形式是：

```
返回类型 operator 运算符号(参数说明);
```

例如：A 类对象加法：

```
class A{ };

int operator + (A&, A&);      //两个 A 类对象参加运算，返回 int 型值
```

C++规定，运算符中，参数说明都是内部类型时，不能重载。例如不允许声明：

```
int * operator + (int, int * );
```

即不允许进行下述运算：

```
int a = 5;
int * pa = &a;
pa = a * pa;      //error
```

C++基本数据类型之间的关系是确定的，如果允许定义其上的新操作，那么，基本数据类型的内在关系将混乱。

C++还规定了“.、::、.*、->、?:”这五个运算符不能重载，也不能创造新运算符。

例如，不允许声明：

```
int operator @(int, int);
```

或者：

```
int operator ::(int, int);
```

例如，下面的程序将运算符＋和＋＋声明为人民币类的友元：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_2.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class RMB
{
   public:
     RMB(unsigned int d, unsigned int c);
     friend RMB operator +(RMB&, RMB&);
     friend RMB& operator ++(RMB&);
     void display()
     {
       cout <<(yuan + jf / 100.0) << endl;
     }
   protected:
     unsigned int yuan;
     unsigned int jf;
};

RMB::RMB(unsigned int d, unsigned int c)
{
   yuan = d;
   jf = c;
   while ( jf >=100 )
   {                              //以使构造时，确保角分值小于100
     yuan ++;
     jf -= 100;
   }
}

RMB operator +(RMB& s1, RMB& s2)
{
   unsigned int jf = s1.jf + s2.jf;
   unsigned int yuan = s1.yuan + s2.yuan;
   RMB result ( yuan, jf );
   return result;
}

RMB& operator ++(RMB& s)
{
   s.jf ++;
   if(s.jf >= 100)
   {
```

```
        s.jf -= 100;
        s.yuan ++ ;
      }
      return s;
}

int main()
{
   RMB d1(1, 60);
   RMB d2(2, 50);
   RMB d3(0, 0);
   d3 = d1 + d2;
   ++d3;
   d3.display();
}
```

运行结果为：

```
4.11
```

operator +() 和 operator ++() 定义为友元是为了能访问人民币类的保护成员。

operator +()是一个双目运算符，它有两个参数 s1 和 s2，并且相加的结果仍为人民币类，返回人民币类对象。

→ 不是必须要让 operator +()执行加法，可以让它做任何事。但是不让它做加法，而作其他操作是一个很糟的想法。如果重载+运算符，向一个磁盘文件写 10 次"I like C++"，语法上可以，但与语义相差悬殊，不利于可读性，背离了允许运算符重载的初衷。当别人读这个程序时，发现 s1 + s2 的操作，想象是某种加法操作，怎么也想不到会是这样的写操作。所以在使重载运算符脱离原义之前，必须保证有充分的理由。

→ 为什么 operator +()中的参数用引用传递而不用指针传递？

因为指针传递存在程序上的可读性问题。如果操作符重载声明为：

```
RMB operator + (RMB * a, RMB * b);
```

则调用时，

```
RMB s1(5.1);
RMB s2(6.7);
RMB c = &s1 + &s2;      //是 s1 的地址与 s2 的地址相加吗?
```

operator ++()是单目运算符，它含有一个参数。operator ++()对人民币类对象的角分做加 1 运算，如果它超过 100，则对该对象的元做加 1 运算并使角分为 0(减 100)。

如果只给出一个 operator ++()定义，那么它一般可用于前缀后缀两种形式。即 d3 ++ 与 ++ d3 不作区别。

18.3 值返回与引用返回

上节中，为什么 operator +()由值返回，而 operator ++()由引用返回呢？

重载定义 + 和 ++ 操作的意义是人为的，所以返回类型并非一定如此规定。但如上节

定义的 + 和 ++ 操作的意义，应该规定 + 由值返回，++ 由引用返回。

对于 operator +()，两个对象相加，不改变其中任一个对象。而且它必须生成一个结果对象来存放加法的结果，并将该结果对象以值的方式返回给调用者。

如果+以引用返回如下例：

```
RMB& operator +(RMB& s1, RMB& s2)
{
  unsigned int jf = s1.jf + s2.jf;
  unsigned int yuan = s1.yuan + s2.yuan;
  RMB result(yuan, jf);
  return result;
}
```

则尽管编译正确，能够运行，但会产生奇怪的结果。例中的 result 对象由+运算符函数的栈空间分配内存，受限于块作用域，引用返回导致了调用者使用这块会被随时分配的空间(见 9.6 节)。

能否将结果对象从堆中分配来避免上例的问题呢？例如：

```
RMB& operator +(RMB& s1, RMB& s2)
{
  unsigned int jf = s1.jf + s2.jf;
  unsigned int yuan = s1.yuan + s2.yuan;
  return *new RMB(yuan, jf);
}
```

虽然它无编译问题，可以运行，但是该堆空间无法回收，因为没有指向该堆空间的指针，会导致内存泄漏，程序不断做加法时，堆空间也在不断流失。

如果坚持结果对象从堆中分配，而返回一个指针，那样在应用程序中就要付出代价：

```
void fn(RMB& a, RMB& b)
{
  RMB* pc = a + b;      // c=a+b; 必须由此三条语句代替
  RMB c = *pc;
  delete pc;
}
```

通过值返回，将有一个临时对象在调用者的栈空间产生，它复制被调函数的 result 对象，以便参加调用者中的表达式运算，对于“c=a+b;”，则 a+b 的临时对象赋给 c，然后临时对象的作用域也结束了。

与 operator +()不一样，operator ++()确实修改了它的参数，而且其返回值要求是左值，这个条件决定了它不能以值返回。如果以值返回：

```
RMB operator ++(RMB& s)
{
  s.jf++;
  if(s.jf >= 100)
  {
   s.jf -= 100;
   s.yuan++;
  }
```

```
    return s;
}
//...
RMB a(2, 50);
c = a++;      //ok
c = ++a;      //ok,a 为 2.52
c = ++ ( ++a );      //error,a 为 2.53,理应 2.54
```

因为 ++a 返回一个对象值，这个对象值并非 a 本身，是临时对象的值，它从形参 s 中拷贝而来，随后又进行了括号外的 ++ 操作，再次产生临时对象，将值赋给 c。所以 a 本身只进行了一次 ++ 操作。

18.4 运算符作成员函数

一个运算符除了可以作为一个非成员函数实现外，还可以作为一个成员函数实现。例如，下面的程序将 ch18_2.cpp 中的 + 和 ++ 运算符改成作为成员予以实现：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_3.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class RMB
{
  public:
    RMB(unsigned int d, unsigned int c);
    RMB operator +(RMB&);
    RMB& operator ++();
    void display()
    {
      cout << (yuan + jf / 100.0) << endl;
    }
  protected:
    unsigned int yuan;
    unsigned int jf;
};

RMB::RMB(unsigned int d, unsigned int c)
{
  yuan = d;
  jf = c;
  while(jf >= 100)
  {
    yuan ++;
    jf -= 100;
  }
}

RMB RMB::operator +(RMB& s)
{
```

```
    unsigned int c = jf + s.jf;
    unsigned int d = yuan + s.yuan;
    RMB result(d,c);
    return result;
}
RMB& RMB::operator ++()
{
    jf ++;
    if(jf >= 100)
    {
      jf -= 100;
      yuan ++;
    }

    return *this;
}

int main()
{
    RMB d1(1, 60);
    RMB d2(2, 50);
    RMB d3(0, 0);
    d3 = d1 + d2;
    ++d3;
    d3.display();
}
```

运行结果为：

```
4.11
```

从中看出，作为成员的运算符比之作为非成员的运算符，在声明和定义时，形式上少一个参数。这是由于C++对所有的成员函数隐藏了第一个参数this(见11.4节)。

下面列出非成员和成员形式的运算符来进行比较：

```
RMB operator +(RMB& s1, RMB& s2)      //非成员形式
{
    unsigned int jf = s1.jf + s2.jf;
    unsigned int yuan = s1.yuan + s2.yuan;
    RMB result(yuan, jf);
    return result;
}

RMB RMB::operator +(RMB& s)      //成员形式
{
    unsigned int c = jf + s.jf;
    unsigned int d = yuan + s.yuan;
    RMB result(c, d);
    return result;
}
```

可见函数体中内容几乎相同，只是非成员形式加s1和s2，成员形式s加当前对象，当前对象的成员隐含着由this指向。即yuan意味着this->yuan。

一个运算符成员形式，将比非成员形式少一个参数，左边参数是隐含的。

作为人民币类的一种常规操作，我们应该允许其中有一个操作数是 double 型的情况：

```
c = c + 2.5; c = 2.7 + c;
```

但是由于参数类型不同，上例的运算符不论是成员形式还是非成员形式，都不能被这两个调用所匹配，还必须重载下列两个成员运算符：

```
RMB operator + (RMB& s, double d)
{
   unsigned int y = s.yuan + d;
   unsigned int j = s.jf + ( d - s.yuan ) * 100 + 0.5;
   RMB result(y, j);
}

inline RMB operator + (double d, RMB& s)
{
   return s + d;
}
```

这里第二个重载运算符调用了第一个重载运算符，两者之间只是参数顺序相反，定义后者为内联函数是一个技巧，省去了必要的开销。

从中得出，为了适应其中一个操作数是 double 的情况，不得不额外引入两个重载运算符。如果有构造函数：

```
RMB(double value)
{
   yuan = value;
   jf = ( value - yuan ) * 100.0 + 0.5;
}
```

就能够将 double 通过构造，变换成 RMB 类，于是：

```
class RMB
{
   public:
    RMB(unsigned int d, unsigned int c);
    RMB(double value);
    friend RMB operator + (RMB& s1, RMB& s2);

   //其余同前
};

int main ( )
{
   RMB s(5.8);
   s = RMB(1.5) + s;      //显式转换(创建一个无名对象)
   s = 1.5 + s;       //隐式转换
   s = s + 1.5;      //隐式转换
   s = s + 1;      //将 int 变换成 double,然后像上面那样变换
}
```

现在不必定义 operator +(double, RMB&) 和 operator +(RMB&, double)了，因为可将 double 转换成 RMB 类，然后匹配 operator +(RMB&, RMB&)。

该变换可以是显式的，如 s=RMB(1.5) + s 那样，也可以是隐含的。此时，由于其中的一个操作数是 RMB 对象，而且参数个数相同，所以它首先假定 operator +(RMB&, RMB&)可以匹配，然后寻找能够使用的转换。发现构造函数 RMB(double)可作为转换的依据。在完成转换后，真正匹配 operator +(RMB&, RMB&)运算符。所以程序员可以通过定义转换函数，来减少定义的运算符个数。

但是如果是下面的情况：

```
s=1.5+6.4;
```

那么由于左右操作数都是 double 型，所以匹配基本数据类型的加法，进行浮点运算。然后因为赋值表达式左面是 RMB 对象，所以该赋值运算将右面表达式的结果用构造函数 RMB(double)进行 RMB 转换，再赋值给 s。

C++规定：=,( ),[ ],-> 这四种运算符必须为成员形式。

→ 我们在实现加法运算符时，用的是非成员形式。如果将运算符改成成员形式，那么，对于 s=1.5 + s; 的形式，仍然必须要有重载运算符 RMB operator +(double, RMB&)来支持，因为在表达式 1.5+s 中，左面的 1.5 不能匹配 RMB::operator +(RMB&)中的隐含类对象，由于没有双目运算符的非成员形式，所以也无法利用类的转换来创造匹配的条件。这就是为什么有些运算符重载(如复数类的+、-运算)用非成员形式的原因。

18.5 重载增量运算符

在 ch18_2.cpp 中，描述的重载增量运算符是不区分前增量与后增量的。那么编译器是如何区别前增量和后增量的呢？

1. 前增量与后增量的区别

使用前增量时，对对象(操作数)进行增量修改，然后再返回该对象。所以前增量运算符操作时，参数与返回的是同一个对象。这与基本数据类型的前增量操作类似，返回的也是左值。

使用后增量时，必须在增量之前返回原有的对象值。为此，需要创建一个临时对象，存放原有的对象，以便对操作数(对象)进行增量修改时，保存最初的值。后增量操作返回的是原有对象值，不是原有对象，原有对象已经被增量修改，所以，返回的应该是存放原有对象值的临时对象。

2. 成员形式的重载

C++约定，在增量运算符定义中，放上一个整数形参，就是后增量运算符。

例如，下面的程序分别定义了前增量与后增量成员运算符：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
```

```
class Increase
{
  public:
    Increase(int x):value(x){}
    Increase & operator ++ ();       //前增量
    Increase operator ++ (int);      //后增量
    void display()
    {
      cout <<"the value is " <<value <<endl;
    }
  private:
    int value;
};

Increase & Increase::operator ++ ()
{
  value ++ ;      //先增量
  return * this;      //再返回原对象
}

Increase Increase::operator ++ (int)
{
  Increase temp( * this);      //临时对象存放原有对象值
  value ++ ;      //原有对象增量修改
  return temp;      //返回原有对象值
}

int main()
{
  Increase n(20);
  n.display();
  (n ++ ).display();      //显示临时对象值
  n.display();      //显示原有对象
  ++ n;
  n.display();
  ++ ( ++ n);
  n.display();

  (n ++ ) ++ ;      //第二次增量操作对临时对象进行
  n.display();
}
```

运行结果为：

```
the value is 20
the value is 20
the value is 21
the value is 22
the value is 24
the value is 25
```

前后增量操作的意义，决定了其不同的返回方式。前增量运算符返回引用，后增量运算符返回值。

后增量运算符中的参数 int 只是为了区别前增量与后增量，除此之外没有任何作用。

因为定义中，无须使用该参数，所以形参名在声明与定义中均省略。

对于(n++)++中的第二个++是对返回的临时对象所作的，从最后一行输出可以看出对n的修改只发生一次。

3. 非成员形式重载

前增量和后增量的非成员运算符，也有类似的编译区分方法。例如，下面的程序将ch18_4.cpp中的前增量和后增量运算符修改为非成员形式：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

class Increase
{
  public:
    Increase(int x):value(x){}
    friend Increase & operator ++ (Increase &);          //前增量
    friend Increase operator ++ (Increase &, int);      //后增量
    void display()
    {
      cout <<"the value is " <<value <<endl;
    }
  private:
    int value;
};

Increase & operator ++ (Increase & a)
{
   a.value++;       //前增量
   return a;        //再返回原对象
}

Increase operator ++ (Increase & a, int)
{
   Increase temp(a);      //通过拷贝构造函数保存原有对象值
   a.value++;           //原有对象增量修改
   return temp;       //返回原有对象值
}

int main()
{
   Increase n(20);
   n.display();
   (n++).display();      //显示临时对象值
   n.display();       //显示原有对象
   ++n;
   n.display();
   ++(++n);
   n.display();
```

```
    (n++)++;        //第二次增量操作对临时对象进行
    n.display();
}
```

运行结果为：

```
the value is 20
the value is 20
the value is 21
the value is 22
the value is 24
the value is 25
```

可见，前增量和后增量运算符的定义以及成员形式与非成员形式稍有不同，但前增量和后增量运算符的使用完全相同。

18.6 转换运算符

转换运算符的声明形式为：

```
operator 类型名();
```

它没有返回类型，因为类型名就代表了它的返回类型，故返回类型显得多余。

转换运算符将对象转换成类型名规定的类型。转换时的形式就像强制转换一样。如果没有转换运算符定义，直接用强制转换是不行的，因为强制转换只能对基本数据类型进行操作，对类类型的操作是没有定义的。

例如，下面的程序在类中定义了转换运算符，在主函数中将 double 数分别显式和隐式转换成 RMB 对象：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

class RMB
{
  public:
    RMB(double value = 0.0);
    operator double()      //转换运算符
    {
      return yuan + jf / 100.0;
    }

    void display()
    {
      cout << (yuan + jf / 100.0) << endl;
    }
  protected:
    unsigned int yuan;
    unsigned int jf;
};
```

```
RMB::RMB(double value)
{
   yuan = value;
   jf = ( value - yuan ) * 100 + 0.5;
}

int main()
{
   RMB d1(2.0), d2(1.5), d3;
   d3 = RMB((double)d1 + (double)d2);     //显式转换
   d3 = d1 + d2;                          //隐式转换
   display(d3);
}
```

运行结果为：

```
3.5
```

对于 d3 = d1 + d2，C++系统依次：

(1) 寻找成员函数的+运算符(此处未找到)；

(2) 寻找非成员+运算符(此处未找到)；

(3) 由于存在内部运算符 operator +(double, double)；所以假定匹配其程序中的加法；

(4) 寻找能将实参(RMB 对象)转换成 double 型的转换运算符 operator double()(找到)。

于是，d1，d2 转换成 double 型，匹配内部的 double 加法，得到一个 double 的结果值，然后，对左面是 RMB 对象的赋值运算符，将右面的表达式转换成 RMB 临时对象，赋值给 RMB 对象 d3。

转换运算符的优点：

有了转换运算符，不必提供对象参数的重载运算符。可以从转换路径，到达 double 型，进行基本运算，得到 double 结果，再构造回来。

转换运算符的缺点：

无法定义其类对象运算符操作的真正含义，因为转换之后，只能进行其他类型的运算符操作(如 double 加法运算)；

通过提供一个 double 转换，所有甚至无意义的 RMB 运算也将获得 double 转换而得以可操作。

转换运算符与转换构造函数(简称转换函数)互逆。例如，RMB(double)转换构造函数将 double 转换为 RMB，而 RMB::operator double()将 RMB 转换成 double。

除此之外，还要防止同一类型提供多个转换路径(转换的二义性)，它会导致编译出错。

例如，下面的代码将使编译出错：

```
class A
{
  public:
    A(B & b);       //用 B 类对象构造 A 对象(B 类对象转换成 A 类对象)
  //...
};

class B
{
  public:
```

```
    operator A();      //将 B 类对象转换成 A 类对象
  //...
};

int main()
{
  B sb;
  A a = A(sb);      //A(sb)是构造还是转换?
}
```

遇到 A(sb)时,编译找到 A 的转换函数,准备将其转换成 A 对象,可是又从 B 类找到转换运算符,也可转换成 A 对象,由于多义性,编译报错。

18.7 赋值运算符

1. 为什么要赋值运算符

只要是用户定义的类或结构,都应能进行赋值运算,这也是继承了 C 语言:

```
struct S { int a, b; };

S a, b;
a = b;      //C 语言允许如此赋值
```

→ 数组名不能赋值,一个数组名代表一个数据类型的实体集合,实质上是一个常量指针,所以它不能:

```
int a[5];
int b[] = {3, 4, 5, 6, 7};
a = b; //error
```

对任何类,像拷贝构造函数一样,C++也默认提供赋值运算符,但要区别拷贝构造函数和赋值运算符:

```
void fn(MyClass& mc)
{
  MyClass newMC = mc;      //这是拷贝构造函数
  newMC = mc;              //这是赋值运算符
}
```

当拷贝构造函数执行时,newMC 对象还不存在,拷贝构造函数起初始化的作用。当赋值运算符在 newMC 上执行时,它已经是一个 MyClass 对象。

在拷贝构造函数中,我们碰到浅拷贝和深拷贝的问题(见 14.6 节),赋值运算符也同样,什么时候浅拷贝不合适,就应提供成员赋值运算符。

2. 如何重载赋值运算符

重载赋值运算符与重载其他运算符类似。

例如,下面的程序提供了赋值运算符作为 Name 类的公共成员,以使主函数(普通函数)中两个对象之间允许互相赋值:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch18_7.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <string.h>
#include <iostream.h>

class Name
{
   public:
     Name()              //默认构造函数
     {
       pName = 0;
     }

     Name(char * pn)                //构造函数
     {
       copyName(pn);
     }

     Name(Name & s)              //拷贝构造函数
     {
       copyName(s.pName);
     }

     ~Name()
     {
       deleteName();
     }

     Name & operator = (Name & s)      //赋值运算符
     {
       deleteName();
       copyName(s.pName);
       return * this;
     }

     void display()
     {
       cout << pName << endl;
     }
   protected:
     void copyName(char * pN);
     void deleteName();
     char * pName;
};

void Name::copyName(char * pN)
{
   pName = new char[strlen(pN) + 1];
   if(pName)
   {
     strcpy(pName, pN);
   }
```

```
}

void Name::deleteName()
{
   if(pName)
   {
    delete pName;
    pName = 0;
   }
}

int main()
{
   Name s("claudette");
   Name t("temporary");
   t.display();
   t = s;      //赋值
   t.display();
}
```

运行结果为：

```
temporary
claudette
```

Name 类在存储区中保留了一个人的名字，在构造函数中该存储区是从堆中分配来的，存在浅拷贝问题，必须自定义赋值运算符与拷贝构造函数。

赋值运算符以 operator =()的名称出现，看起来像一个析构函数后面跟着拷贝构造函数。通常赋值运算符有两部分，第一部分与析构函数类似，在其中取消对象已经占用的资源。第二部分与拷贝构造函数类似，在其中分配新的资源。

对象 t 创建时，具有名字“temporary”，它在堆中存放。在 t = s 赋值过程中，通过调用 deleteName()，原先名字占用的空间还给堆，再另外调用 copyName()从堆中分配新存储区去存储新名字“claudette”。

拷贝构造函数不需要调用 deleteName()，因为刚创建时，还没有分配存放名字的堆空间。

赋值运算符 operator =()的返回类型是 Name&。这与赋值的语义相匹配。C++要求赋值表达式左边的表达式是左值，它能进行诸如下列的运算：

```
int a, b=5;
(a=b)++;      //结果 a 为 6
```

如果一个类定义了 ++ 运算符，则它也能执行类似上面的表达式，得到正确的 a 值。例如，在 ch18_3.cpp 中如果增加一个人民币类的赋值运算符，且不返回引用：

```
RMB operator =(RMB& s)
{
   yuan = s.yuan;
   jf = s.jf;
}
```

这时执行下列表达式：

```
RMB a(5.2), b(2.6);
(b=a)++;      //结果 b 为 5.2,并不是期望的 5.3
```

因为 b=a 返回的是对象值,而 RMB 类定义了 ++ 操作,所以,"(b=a)++;"是合法的。但由于返回的不是引用,该值是 b 对象的一个复制,并不是 b 本身,所以 ++ 的操作数是 b 的复制对象而已。

➡ 如果赋值运算符说明为保护或私有的,则可以将赋值操作限定在类的作用域范围,防止应用程序中使用赋值操作:

```
class Name
{
  //
  protected:
    Name& operator = (Name& s)
    {
      //...
      return *this;
    }
};

//...
void fn(Name& n)
{
  Name newN;
  newN = n;      //error
}
```

因为 Name 类对象 newN 使得=匹配为 Name 类的赋值运算符,但是 protected 限定符使之不能在普通函数中被调用,从而防止了对象非法赋值操作。

小结

使用运算符重载可以使程序易于理解并易于对对象进行操作。几乎所有的 C++运算符都可以被重载,但应注意不要重载违反常规的运算符。不能改变运算符操作数的数量,也不能发明新运算符。

如果在类中没有说明本身的拷贝构造函数和赋值运算符,编译程序将会提供,但它们都只是对对象进行成员浅拷贝。在那些以指向堆空间指针作为数据成员的类中,必须避免使用浅拷贝,而要为类定义自己的赋值运算符,以给对象分配堆内存。

this 指针指向当前的对象,它是所有成员函数的不可见的参数,在重载运算符时,经常返回 this 指针的间接引用。

通过转换运算符可以在表达式中使用不同类型的对象。转换运算符不遵从函数应有返回值类型的规定,与构造函数和析构函数相同,它没有返回值。

在前增量和后增量运算符定义中,使用 int 形参只是为了标志前后有别,没有其他作用。

拷贝构造函数用已存在的对象创建一个相同的新对象。而赋值运算符用已存在的对象赋予一个已存在的同类对象。

练习

18.1 定义复数类的加法与减法，使之能够执行下列运算：

```
Complex a(2, 5), b(7, 8), c(0, 0);
c = a + b;
c = 4.1 + a;
c = b + 5.6;
```

18.2 编写一个时间类，实现时间的加、减、读和输出。

18.3 根据 ch18_3.cpp，增加操作符，以允许人民币与 double 型数相乘。

```
friend money operator * (const money&, double);
friend money operator * (double, const money&);
```

注意：两个 money 对象不允许相乘。

18.4 根据 ch18_3.cpp，增加操作符，以允许作相应赋值。

```
money& operator += (const money&);
money& operator += (double);
money& operator -= (const money&);
money& operator -= (double);
```

第19章 I/O 流

C++的I/O流类，是最常用的I/O系统，到目前为止，我们一直在用这个类。学习了本章后，应该理解怎样使用C++面向对象的I/O流，能够格式化输入和输出，理解I/O流类的层次结构，理解怎样输入和输出用户自定义类型的对象，能够建立用户自定义的流操作符，能够确定流操作的成败，能够把输出流系到输入流上。

19.1 printf 和 scanf 的缺陷

1. 非类型安全

函数原型使编译系统对它进行必要的类型检查，免除了许多错误。但对于printf()和scanf()，它却毫无帮助。printf()和scanf()所期望的参数个数与类型取决于包含在第一个参数中的信息，而这一信息对编译器是没有用的。编译器无法检查对printf()和scanf()的调用的正确性。

例如，下面的函数企图输入和输出异于格式符的数据：

```
#include<stdio.h>

int j=10;
float f = 2.3;

void fn()
{
   printf("%d",f);
   scanf("%d",&f);
   scanf("%d",j);
   printf("%d","abcde");
}
```

在int型占两个字节的情况下，语句printf("%d",f)；只输出f变量中前2个字节的内容，并按int型数据格式进行解释；

语句scanf("%d",&f)；只输入到f变量中前2字节中，按int型格式进行存放，而后面两个字节内容却没有改变；

语句scanf("%d",j)；将键入值存放到地址为0x000A的内存空间中；

语句printf("%d","abcde")；输出"abcde"的地址值，而不是想要的字串。

上面这些语句，用错了数据类型，而编译都能通过。为此，程序员将化更多的代价在程序运行中出现的错误诊断上。特别对于scanf()中的错误，往往是致命的。

2. 不可扩充性

printf()和scanf()知道如何输入输出已知的基本数据类型值，但是，C++程序中大量的

类对象，其输入输出格式是未预先定义的，这就希望输入输出语句能够更加灵活与可扩充。printf()和 scanf()却无能为力，它们既不能识别，也不能学会如何识别用户定义的对象。

例如，下面的函数企图输入和输出一个类的对象：

```
class A{ /*...*/ };
A a;

//...
void fn()
{
   printf("%?",a);        //? 表示不知以什么格式符来识别 A 的对象
   scanf("%?",&a);
}
```

19.2 I/O 标准流类

1. 标准流的设备名

iostream.h 是 I/O 流的标准头文件。其对应的标准设备见表 19-1。

表 19-1 标准 I/O 流设备

C++名字	设 备	C 中的名字	默认的含义
cin	键盘	stdin	标准输入
cout	屏幕	stdout	标准输出
cerr	屏幕	stderr	标准错误
clog	打印机	stdprn	打印机

这表明 cout 对象的默认输出是屏幕，cin 对象的默认输入是键盘。

2. 原理

cout 是 ostream 流类的对象，它在 iostream.h 头文件中作为全局对象定义：

```
ostream cout(stdout);      //标准设备名作为其构造时的参数
```

ostream 流类对应每个基本数据类型都有友元，它们在 iostream.h 中声明：

```
ostream& operator<<(ostream& dest,char * pSource);
ostream& operator<<(ostream& dest,int source);
ostream& operator<<(ostream& dest,char source);
//等等
```

分析语句：

```
cout <<"My name is Jone";
```

cout 是 ostream 对象，<< 是操作符，右面是 char * 类型，故匹配上面的“ostream& operator<<(ostream& dest,char * pSource);”操作符。它将整个字串输出，并返回

ostream 流对象的引用。如果是：

```
cout <<"this is " <<7;
```

则根据<<的运算优先级，可以看作为：

```
(cout <<"this is ") <<7;
```

由于"cout <<"this is ""返回 ostream 流对象的引用，与后面的<<7 匹配了另一个"ostream& operator<<(ostream& dest,int source);"操作符，结果构成了连续的输出。

同理，cin 是 istream 的全局对象，istream 流类也有若干个友元：

```
istream& operator>>(istream& dest,char * pSource);
istream& operator>>(istream& dest,int source);
istream& operator>>(istream& dest,char source);
//等等
```

除了标准输入输出设备，还有标准错误设备 cerr。

当程序测试并处理关键错误时，不希望程序的错误信息从屏幕显示重定向到其他地方，这时使用 cerr 流显示信息。

例如，下面程序在除法操作不能进行时显示一条错误信息：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_1.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

void fn(int a, int b)
{
   if(b==0)
    cerr <<"zero encountered. "
         <<"The message cannot be redirected";
   else
    cout <<a/b <<endl;
}

int main()
{
   fn(20,2);
   fn(20,0);
}
```

运行结果为：

```
c>ch19_1 >abc.dat
zero encountered. The message cannot be redirected.
```

文件 abc.dat 的内容为：

```
10
```

主函数第一次调用 fn()函数时，没有碰到除 0 运算，得到文件的写内容 10，第二次调用 fn()函数时，碰到除 0 运算，于是在屏幕上输出错误信息。**写到 cerr 上的信息是不能被重定**

向的，所以它只能在屏幕上显示。

19.3 文件流类

ofstream，ifstream，fstream 是文件流类，在 fstream. h 中定义。其中，fstream 是 ofstream 和 ifstream 多重继承的子类。文件流类不是标准设备，所以没有 cout 那样预先定义的全局对象。文件流类定义的操作应用于外部设备，最典型的设备是磁盘中的文件。要定义一个文件流类对象，须规定文件名和打开方式。

类 ofstream 用于执行文件输出，该类有几个构造函数，其中最有用的是：

```
ofstream::ofstream(char * pFileName,
                   int mode = ios::out,
                   int prot = filebuf::openprot);
```

第一个参数是指向要打开的文件名字串，第二和第三个参数说明文件如何被打开。mode 是打开方式，它的选择项见表 19-2。

表 19-2 文件打开选择项

标 志	含 义
ios::ate	如果文件存在，输出内容加在末尾
ios::in	具有输入能力(ifstream 默认)
ios::out	具有输出能力(ofstream 默认)
ios::trunc	如文件存在，清除文件内容(默认)
ios::nocreate	如文件不存在，返回错误
ios::noreplace	如文件存在，返回错误
ios::binary	以二进制方式打开文件

prot 是文件保护方式，它的选择项见表 19-3。

表 19-3 文件保护方式选择项

标 志	含 义
filebuf::openprot	兼容共享方式
filebuf::sh_none	独占，不共享
filebuf::sh_read	允许读共享
filebuf::sh_write	允许写共享

例如，下面的程序在文件 myname 中，写入一些信息：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *          ch19_2.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * *
```

```
#include<fstream.h>

void fn()
{
   ofstream myf("c:\\bctemp\\myname"); //ios::out|ios::trunc 方式

   myf <<"In each of the following questions, a related pair\n"
       <<"of words or phrases is followed by five lettered pairs\n"
       <<"of words or phrases.\n";
}
int main()
{
   fn();
}
```

此处的文件名要说明其路径，斜杠要双写，因为编译器理解下的斜杠是字符转换符。这与包含头文件时的路径不一样，因为包含头文件是由编译预处理器处理的。

文件打开时，匹配了构造函数 ofstream::ofstream(char *)，只需一个文件名，其他为默认，打开方式默认为 ios::out|ios::trunc，即该文件用于接受程序的输出，如果该文件原先已存在，则其内容必须先清除，否则就新建。

例如，若要打开二进制文件，写方式，输出到文件尾，则：

```
ofstream bfile("binfile", ios::binary|ios::ate);
```

又例如，要检查文件打开否，则判断 fail()成员函数：

```
#include <fstream.h>

void fn()
{
   ofstream myf("myname");
   if(myf.fail())      //fail() = =1 表示失败
   {
    cerr <<"error opening file myname\n";
    return;
   }
   myf <<"...";
}
```

例如，打开一个输入文件(要从文件中读数据)：

```
#include <fstream.h>

void fn()
{
   ifstream myinf("abc.dat", ios::nocreate);      //若文件丢失，就报错
   //...}
```

可以通过检查 myinf.fail()来确定打开文件是否有错。

例如，打开同时用于输入和输出的文件：

```
fstream myinout("abc.dat", ios::in|ios::out);
```

用 ifstream 打开的文件，默认打开方式为 ios::in，用 fstream 打开的文件，打开方式不能默认。

19.4 串流类

ostrstream，istrstream，strstream 是串流类，在 strstrea.h 中定义。其中，strstream 是 ostrstream 和 istrstream 多重继承的子类。同样串流类不是标准设备，所以没有 cout 那样预先定义的全局对象。串流类允许将 fstream 类定义的文件操作应用于存储区中的字符串，即将字符串看作为设备，这很像 C 中的库函数 sprintf() 和 sscanf ()。要定义一个串流类对象，须提供字符数组和数组大小。

类 istrstream 用于执行串流输入，该类有几个构造函数，其中最有用的是：

```
istrstream::istrstream(const char * str);
istrstream::istrstream(const char * str, int size);
```

第一个参数指出字符串数组，第二个参数说明数组大小。当 size 为 0 时，表示把 istrstream 类对象连接到由 str 指向的以空字符结束的字符串。

例如，下面的代码定义一个串流类对象，并对其进行输入操作：

```
char str[100] = "I am a student.\n";
char a;
istrstream ai(str);        //将 str 看作输入设备
ai >>a;                    //从输入设备中输入一个字符
cout <<a <<endl;           //输出一个字符
```

输出结果为：

```
I
```

类 ostrstream 用于执行串流输出，该类也有几个构造函数，其中最有用的是：

```
ostrstream::ostrstream(char * , int size, int = ios::out);
```

第一个参数指出字符串数组，第二个参数说明数组大小，第三个参数指出打开方式。

例如，下面代码使用串流输入对字符串中的数据进行解读：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<strstrea.h>

char * parseString(char * pString)
{
   istrstream inp(pString,0);      //ios::in 方式，读到 Null 结束
   int aNumber;
   float balance;
   inp >>aNumber >>balance;        //从串流中读入一个整数和浮点数
   char * pBuffer = new char[128];
   ostrstream outp(pBuffer,128);     //ios::out 方式，字串长度 128
```

```
    outp <<"a Number = "<<aNumber      //写入 pBuffer 中
         <<", balance = "<<balance;
    return pBuffer;
}

int main()
{
    char * str = "1234 100.35";
    char * pBuf = parseString(str);
    cout <<pBuf <<endl;
    delete []pBuf;
}
```

运行结果为：

```
a Number = 1234, balance = 100.35
```

在函数 parseString()中，以 pString 为输入设备，先定义一个输入串流对象 inp，从中输入一个整数和一个浮点数。

然后，开辟一个字符串空间(pBuffer 指向的 128 个字符)作为输出设备而定义输出串流对象 outp，将从输入设备中输入的该两个变量值输出。

19.5 控制符

C++有两种方法控制格式输出。

1. 用流对象的成员函数

例如，下面的程序改变有效位数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_4.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

void fn(float interest, float amount)
{
    cout <<"RMB amount = ";
    cout.precision(2);      //置有效位数为 2 位
    cout <<amount;
    cout <<"\nthe interest = ";
    cout.precision(4);      //置有效位数为 4 位
    cout <<interest <<endl;
}

int main()
{
    float f1 = 29.41560067;
    float f2 = 12.567188;
    fn(f1,f2);
}
```

运行结果为：

```
RMB amount = 13
the interest = 29.42
```

precision()为 cout 对象的成员函数，在要求输出一定精度的数据之前，先调用这个精度设置成员函数。

2. 用控制符

manipulators(控制符)是在头文件 iomanip.h 中定义的对象，与成员函数调用效果一样。控制符的优点是程序可以直接将它们插入流中，不必单独调用。

例如，用控制符设置有效位数，重写 ch19_4.cpp：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

void fn(float interest, float amount)
{
   cout <<"RMB amount = "
        <<setprecision(2) <<amount;
   cout <<"\nthe interest = "
        <<setprecision(4) <<interest
        <<endl;
}

int main()
{
  float f1 = 29.41560067;
  float f2 = 12.567188;
  fn(f1,f2);
}
```

常用控制符和流格式控制成员函数如表 19-4 所示。

表 19-4　常用控制符与流控制成员函数

控制符	成员函数	描述
dec	flags(10)	置基数为 10
hex	flags(16)	置基数为 16
oct	flags(8)	置基数为 8
setfill(c)	flags(c)	设填充字符为 c
setprecision(n)	precision(n)	设显示小数精度为 n 位
setw(n)	width(n)	设域宽为 n 个字符

控制符和流成员函数相对应，它们用法不同，但作用相同。

其中 setw(n)或 width(n)很特别，它们在下一个域输出后，又回到原先的默认值。例

如，输出下面的两个数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch19_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include<iomanip.h>

int main()
{
   cout.width(8);
   cout <<10
        <<20 <<endl;
}
```

运行结果为：

```
------1020
```

运行结果中的下横线表示空格。整数 20 并没有按宽度 8 输出。setw()的默认值为宽度 0，即 setw(0)，意思是，按输出对象的表示宽度输出。所以 20 就紧挨 10 了。若要每个数值都有域宽度 8，则每个值都要设置：

```
cout <<setw(8) <<10
     <<setw(8) <<20 <<endl;
```

从中得出，用控制符的方法更加直接。

又例如，下面的程序打印一个倒三角形：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch19_7.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<iomanip.h>

int main()
{
  for(int n = 1; n<8; n ++ )
    cout <<setfill(' ') <<setw(n) <<" "
         <<setfill('m') <<setw(15 - 2 * n) <<"m " <<endl;
}
```

运行结果为：

```
 mmmmmmmmmmmmm
  mmmmmmmmmmm
   mmmmmmmmm
    mmmmmmm
     mmmmm
      mmm
       m
```

cout<<setfill(' ')<<setw(n)<<" "中所要显示的" "长度为 n，但它本身长度只有

1，所以其余的内容就由 setfill(' ')来填充了，效果就使得'm'前的空格逐行增加。同样，cout<<setfill('m') <<setw(15－2＊n) <<"m"中所要显示的"m"长度为 15－2＊n，但它本身长度只有 1，setfill('m')的作用就是将 15－2＊n 个空格用其'm'来填充，由于 15－2＊n 逐行递减，结果就显出一个用'm'构筑的倒三角形。

比较下列同样完成倒三角形显示的程序：

```
//＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊
//＊＊              ch19_8.cpp                ＊＊
//＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊

#include<iostream.h>

int main()
{
   for(int k =1; k <=7; k++)
   {
    for(int k =1; k <=n; k++) cout <<"  ";

    for(int k =1; k <=15－2＊n; k++) cout <<"m";

    cout <<endl;
   }
}
```

但流成员函数也有其优点，它种类多，而且可以返回以前的设置，便于恢复设置。

例如，下面的函数设置某有效位数输出，然后恢复成原来的有效位数设置：

```
//＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊
//＊＊              ch19_9.cpp                ＊＊
//＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊

#include<iostream.h>

int main()
{
   float value =2.345678;
   int prePrecision;
   prePrecision =cout.precision(4);
   cout <<value;
   cout.precision(prePrecision);
   //...
}
```

运行结果为：

```
2.346
```

假定程序中原来的有效位数设置不知道，"cout.precision(4)"可以返回原来设置的有效位数，保存该值在 prePrecision 变量中，使得最后用该值恢复原来的设置。

19.6 使用 I/O 成员函数

大多数简单的 C++程序使用 cin 来进行输入操作。有时候需要对输入操作进行更加精

细的控制，这时可以用 I/O 流成员函数。

1. 用 getline()读取输入行

当程序使用 cin 输入时，cin 用空白符和行结束符将各个值分开。根据所需输入的值，可能需要读取一整行文本并且分开不同的域。为了读取一行文本，可以使用 getline 成员函数。其函数原型为：

```
getline(char * line, int size, char = '\n');
```

第一个参数是字符数组，用于放置读取的文本，第二个参数是本次读取的最大字符个数，第三个参数是分隔字符，作为读取一行结束的标志。

例如，下面的函数从键盘读取一行文本：

```
#include<iostream.h>

void fn()
{
  char str[128];

  cout <<"Type in a line of text and press Enter" <<endl;
  cin.getline(str, sizeof(str));
  cout <<"You typed: " <<str <<endl;
}
```

在默认状态，getline 成员函数读字符直到回车，或者读到指定的字符数。为了要在遇到某个字符(比如字母'X')时，结束输入操作，可以按下面方式使用：

```
cin.getline(line, sizeof(line), 'X');
```

例如，下面程序在遇到'X'字符处结束第一个输入操作，然后执行第二个输入：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_10.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

int main()
{
  char str[128];

  cout <<"Type in a line of text and press Enter" <<endl;
  cin.getline(str, sizeof(str), 'X');
  cout <<"First line: " <<str <<endl;
  cin.getline(str, sizeof(str));
  cout <<"Second line: " <<line;
}
```

运行结果为：

```
Type in a line of text press Enter
You should look "X" up in the subject catalogue.
First line: You should look "
```

```
Second line: " up in the subject catalogue.
```

运行时，当出现输入文本的提示时，输入一组以 X 分隔开的单词。当程序显示其输出时，将把 X 以前的文本显示为一行，X 以后的文本显示为一行，不包括 X 字符本身。

程序中的 X 为大小写敏感的。一个小写 X 不会结束第一个 cin. getline()的输入，而且，在输入 X 之前，可以按一到多次回车键，而并不结束第一个 cin. getline ()的输入。第一个 cin. getline()的输入操作将以键入 X 后的第一个回车结束。

→ "cin. getline();"与"cin >>str;"的一个不同是，前者输入一行，行中可以包含空格，后者却以空格或回车作为字串结束，不包含空格。

2. 用 get()读取一个字符

根据程序的输入要求，有时需要执行每次一个字符的输入操作。这时，可以使用 get()成员函数。其格式为：

```
char istream::get();
```

例如，下面程序循环读入字符，直到用户输入一个 Y 字符，或遇到 ctrl－C(文件尾)：

```
//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_11.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<ctype.h>

int main()
{
   char letter;

   while(!cin.eof())
   {
    letter = cin.get();
    letter = toupper(letter);
    if(letter == 'Y')
    {
      cout <<"'Y' be met.";
      break;
    }
    cout <<letter;
   }
}
```

如上所示，该程序在遇到一个 Y 字符之前做简单循环，为了简化测试，该程序将每个字符都转换成大写。

"char toupper(char);" 函数原型在 ctype. h 中声明，如果参数为小写字母，将其转换为大写字母，否则，原样返回。上例中函数 toupper()将 letter 转换为大写字母后，赋值给 letter 变量，所以 letter 变量值被改变。

使用流成员函数的输入操作不只限于键盘，上例程序可从重定向输入中每次读入一个字符。下面的命令把 ch19_11. cpp 文件作为重定向输入，并输出运行结果：

```
c>ch19_11 < ch19_11.cpp
#include<iostream.h>
#include<ct'Y' be met.
```

→ "letter=cin.get();"与"cin>>letter;"都是从输入流中取一个字符，但却有区别。默认情况下，cin>>letter 将跳过任何在文件中发现的任何空白字符（空白字符指空格、tab 符、backspace 符和回车符）。而 cin.get()则不跳过空白字符。

3. 用 get()输入一系列字符

用 get()成员函数的第二种形式可以输入一系列字符，直到输入流中出现结束符或所读字符个数已达到要求读的字符个数。其原型为：

```
istream& istream::get(char * ,int n,char delim = '\n');
```

由于可以规定输入字符个数，所以下面不安全的代码：

```
istream fin("abc.dat");
char buffer[80];
fin >>buffer;          //不能保证输入字符个数在 80 以内
```

可以改写为：

```
istream fin("abc.dat");
char buffer[80];
fin.get(buffer,80);       //保证输入字符个数在 80 以内
```

→ getline()和 get()第二种形式相同。唯一的例外是 getline()从输入流中输入一系列字符时包括分隔符，而 get()不包括分隔符。

4. 输出一个字符

下例程序使用 put()成员函数，在屏幕上依次显示字母表中的字母：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_12.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

int main()
{
   char letter;

   for(letter = 'A'; letter<= 'Z'; letter ++ )
    cout.put(letter);
}
```

运行结果为：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

→ cout <<letter; 与 cout.put(letter); 有一个区别，前者显示以该数据类型表示的形式，后者将参数值以字符方式显示。所以，若 letter 是 char 型，那么这两种方法都可以用来显

示字母，若 letter 为 int 型，那么前者将在屏幕上显示 65 到 90 的数字，而不是字母 A 到 Z。

上述流成员函数同样适用于文件流和串流。例如，下面的程序打开以命令行变元形式指定的文件，然后逐个读取字符并显示：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch19_13.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<fstream.h>
#include<iostream.h>

int main(int argc, char * * argv )
{
   ifstream in(argv[1]);

   if(in.fail())
   {
    cerr <<"Error opening the file: " <<argv[1] <<endl;
    return;
   }
   while(!in.eof())
      cout.put(in.get());

   in.close();
}
```

put()成员函数的参数，是文件流对象 in 的成员函数 get()的返回值。

19.7 重载插入运算符

C++重载左移运算符<<执行输出，对程序员来说，很满足。因为你可以随意重载同一个运算符去执行自己定义的类输出。左移运算符也称插入运算符，它比较形象，执行“cout <<x;”输出时，好像 x 被插入到输出设备上。重载插入运算符的特性使得流 I/O 可扩展，这与 printf()是重要的区别。

例如，下面程序进行插入运算符重载，打印人民币类对象：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch19_14.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<iomanip.h>

class RMB
{
   public:
    RMB(double v = 0.0)
    {
      yuan = v;       //yuan 得到 v 的整数部分
      jf = (v - yuan) * 100.0 + 0.5;
    }
```

```
    operator double()
    {
      return yuan + jf/100.0;
    }

    void display(ostream& out)
    {
      out <<yuan <<'.'
          <<setfill('0') <<setw(2) <<jf   //如：8 分显示 08
          <<setfill(' ');
    }
  protected:
    unsigned int yuan;
    unsigned int jf;
};

ostream& operator <<(ostream& oo, RMB& d)
{
  d.display(oo);
  return oo;
}

int main()
{
   RMB rmb(1.5);
   cout <<"Initially rmb = " <<rmb <<"\n";
   rmb = 2.0 * rmb;
   cout <<"then rmb = " <<rmb <<"\n";
}
```

运行结果为：

```
Initially rmb = 1.50
then rmb = 3.00
```

display(ostream& out)中的参数是向它传递的任何 ostream 对象输出，并不一定是 cout 输出，这是出于灵活的考虑。它允许 fstream 和 strstream 对象，fstream 操作的对象是设备中的文件，典型的例子就是磁盘文件，strstream 操作的对象是内存中的字符串。因为 fstream 和 strstream 都是 ostream 的子类，这样就便于输出到不同的地方。如，定义一个文件流，并将 rmb 对象输出：

```
fstream fout("abc.dat");
fout <<rmb;
```

这时，“fout<<rmb;”仍能够匹配重载的插入运算符。

重载插入运算符 ostream& operator<<() 调用了 RMB 类的成员 display()，以此来完成输出任务，所以它自己不必是友元。这样，RMB 类的界面显得更干净。

由于输出都是通过重载插入运算符来完成，所以也可以更简单地把成员函数 display() 里面所做的一切都挪到重载插入运算符里来：

```
ostream& operator <<(ostream& oo, RMB& d){
```

```
    oo <<d.yuan <<'.'
       <<setfill('0') <<setw(2) <<d.jf
       <<setfill(' ');
    return oo;
}
```

这时候，重载插入运算符应为 RMB 类的友元，因为它要直接访问 RMB 类的保护数据。但是它不能是成员，因为首先，插入运算符跟在 ostream 对象的后面，显然它不能是 RMB 类的成员，其次，ostream 类在 iostream.h 头文件中定义，是标准类库，用户只能继承，不能修改标准类库，所以它更不能是 ostream 类的成员。

→ 重载插入运算符中最后一条语句是“return oo;”，为什么要返回传递给它的 ostream 对象？

这样做允许该运算符在单个表达式中与其他插入运算符联结在一起。<<的运算顺序是从左到右，下面的表达式

```
void fn(RMB r, float interest)
{
  cout <<"Sun of these = "<<r <<" + " <<interest <<endl;
}
```

被翻译成：

```
void fn(RMB r, float interest)
{
  ((((cout<<"Sum of these = ")<<r)<<" + ")<<interest)<<endl;
}
```

第一次插入向 cout 输出字符串“Sum of these =”。该表达式的结果是对象 cout，它然后被传递给重载插入运算符

```
ostream& operator<<(ostream&, RMB&)
```

这个运算符返回它的 ostream 对象，这一点很重要，这样做，对象才能被传递给下一个插入运算符。

假设重载插入运算符返回类型是 void，像前面那样相当有效的用法将出现编译错。因为你不能向 void 内插入一个字符串。下列错误更糟糕，因为更难发现：

```
ostream& operator<<(ostream& os, RMB& rmb)
{
  rmb.display(os);
  return cout;
}
```

这个运算符没有返回它给出的 ostream 对象，而返回 ostream 对象 cout。cout 最常用，编译也正确，但是在下面程序中：

```
void fn(int acc, RMB& balance, char * pName)
{
  ofstream outf("accounts.dat", ios::ate);
  outf <<acc <<balance <<pName <<endl;
}
```

int acc 通过 outf 的插入运算符输出到 outf 中，返回 outf。然后 RMB 通过重载插入运算符输出到 outf 中，可是错误地返回 cout，而不是 outf，现在 pName 输出到 cout 中，而不是输出到想要输出的文件中。

19.8 插入运算符与虚函数

插入运算符不能是成员函数，也就不能成为虚函数，因此对于上节中的重载插入运算符定义，在下例的派生类中，显得无能为力：

```
class DerivedRMB :public RMB
{
   public:
    DerivedRMB(double v, int n) :RMB(v),c(n){}
   protected:
    int c;
};

DerivedRMB a(5.2,3);

void fn()
{
   cout <<a;      //a.c 将不能显示
}
```

cout<<a；能匹配重载的插入运算符，但是执行的结果是输出 a 的人民币值而不能输出 a 作为派生的附加信息 a.c。

解决方法：在重载插入运算符中，不直接实现输出，而是调用 display()成员，再将 display()定义为虚函数。这样，重载插入运算符的行为便可随 display()的不同而不同。这就是上一节为什么要间接实现重载插入运算符的用意。

1. 先定义一个抽象类

```
class Currency
{
   public:
    Currency(double v = 0.0)
    {
      yuan = v;
      jf = (v - yuan) * 100.0 + 0.5;
    }

    virtual void display(ostream& out) = 0;
   protected:
    unsigned int yuan;
    unsigned int jf;
};
```

2. 派生人民币类等子类

```
class RMB :public Currency
```

```
{
  public:
    RMB(double v = 0.0) :Currency(v){}
    virtual void display(ostream& out)
    {
      out <<yuan <<'.'
          <<setfill('0') <<setw(2) <<jf <<setfill(' ')
          <<" RMB";
    }
};

class Dmark :public Currency      //德国马克
{
  Dmark(double v = 0.0):Currency(v){}
  virtual void display(ostream& out)
  {
    out <<yuan <<'.'
        <<setfill('0') <<setw(2) <<jf <<setfill(' ');
    out <<" DM";
  }
};

class WHQ :public RMB      //人民币外汇券
{
  public:
    WHQ(double v = 0.0):RMB(v){}
    virtual void display(ostream& out)
    {
      RMB::display(out);
      out <<" WHQ";
    }
};
```

3. 定义插入运算符

```
ostream& operator <<(ostream& oo, Currency& c)
{
  c.display(oo);
  oo <<endl;
  return oo;
}
```

4. 应用

```
void fn(Currency& c)
{
  cout <<"Deposit is " <<c <<endl;
}

int main()
{
  RMB rmb(5.5);
```

```
    fn(rmb);
    WHQ whq(5.6);
    fn(whq);
    Dmark mark(10.6);
    fn(mark);
}
```

运行结果为：

```
Deposit is 5.5 RMB
Deposit is 5.6 RMB WHQ
deposit is 10.6 DM
```

类 Currency 有两个子类 RMB 和 Dmark，RMB 有一个派生类 WHQ。Currency 中 display()成员定义为纯虚函数。在每一个子类中，display()成员被重载，从而可以适当的格式输出相应对象。重载插入运算符函数对 display()的调用是一个虚调用。因此当它被传递以 RMB 类对象时，则像人民币那样输出；当它被传递以 Dmark 对象时，则像德国马克那样输出。因而，尽管重载的插入运算符不是虚拟的，因为它调用了一个虚函数，结果令人满意。

重载插入运算符的参数 Currency& 必须为引用，如果以值传递，那么对于：

```
ostream& operator<<(ostream& oo, Currency& c)
```

编译将报错。因为 Currency 是抽象类，不能构造该类的对象。

19.9 文件操作

1. 文件输出

在文件输出时，往往涉及到大量相关记录的集合。把文件看作是相关记录的集合。如要输出若干个学生对象，每个学生对象含有学生姓名、学号、成绩等，将其看作是一个数据记录。

例如，下面的程序输出三个学生对象，其中一个大学生，两个硕士生。程序由一个工程文件 ch19_15.prj 组成：

```
//- - - - - - - - - - - - - - - - - - - - - - - - -
//- -              ch19_15.prj                  - -
//- - - - - - - - - - - - - - - - - - - - - - - - -

ch19_15.cpp
student.cpp
master.cpp
```

其头文件和程序源文件分别为：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              student.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
```

```
#include <string.h>
#ifndef STUDENT
#define STUDENT

//大学生类
class Student
{
   public:
     Student(char * pS, unsigned num, float g)
     {
       strcpy(pName,pS);
       uID = num;
       grade = g;
     }
     virtual void display(ostream& out);
   protected:
     char pName[20];
     unsigned int uID;
     float grade;
};
ostream& operator<<(ostream& out, Student& st);
#endif

//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              student.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *


#include "student.h"
#include <iomanip.h>
#include <iostream.h>

void Student::display(ostream& out)
{
  out <<setiosflags(ios::left) <<setw(20)<<pName
      <<uID <<","
      <<setiosflags(ios::right) <<setw(4) <<grade;
}

//插入操作符
ostream& operator<<(ostream& out, Student& st)
{
  st.display(out);
  out <<endl;
  return out;
}

//* * * * * * * * * * * * * * * * * * * * * * *
//* *              master.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include "student.h"
#include <iostream.h>

//硕士生类
class MasterStudent: public Student
```

```
{
  public:
    MasterStudent(char * pS, unsigned num, float g, char t)
                  :Student(pS,num,g),type(t){}
    void display(ostream& out);
  protected:
    char type;
};

//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              master.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "master.h"

void MasterStudent::display(ostream& out)
{
  Student::display(out);
  out <<", " <<type;
}

//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_15.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

#include <fstream.h>
#include "studeut.h"
#include "master.h"

int main()
{
  ofstream out("e:\\bctemp\\abc.txt");

  Student s1("Dill Arnson", 12567, 3.5);
  MasterStudent s2("Welch Shammas", 12667, 4.1, 'A');
  MasterStudent s3("Portel Braumbel", 12579, 3.8, 'B');

  out <<s1;
  out <<s2;
  out <<s3;
}
```

运行结果可以在打开的文件中看到：

```
e:\>type abc.txt
Dill Arnson             12567, 3.5
Welch Shammas           12667, 4.1, A
Portel Braumbel         12579, 3.8, B
```

2. 文件输入

如果要打开一个文件用于输入，可以用 ifstream 类。用上例中的类定义，假定文件 abc.txt 中的内容为：

```
Dill Arnson          12567 3.5 A
Welch Shammas        12667 4.1 A
Portel Braumbel      12579 3.8 B
```

将该文件的内容逐条输入，创建 MasterStudent 对象，屏幕输出该记录内容。其程序为一个工程：

```
//- - - - - - - - - - - - - - - - - - - - - - - -
//- -              ch19_16.prj                - -
//- - - - - - - - - - - - - - - - - - - - - - - -
```

```
ch19_16.cpp
student.cpp
master.cpp
```

其 ch19_16.cpp 文件内容为：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch19_16.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *
```

```
#include <iostream.h>
#include <fstream.h>
#include "student.h"
#include "master.h"
#include <string.h>

int main()
{
   ifstream fin("e:\\bctemp\\abc.txt");

   char sFirst[10];
   char sLast[10];
   unsigned int uid;
   float nGrade;
   char type;
   char name[20];
   Student * pS;
   int i = 0;

   fin>>sLast >>sFirst >>uid >>nGrade >>type;

   while(!fin.eof())
   {
     strcpy(name,strcat(sLast," "));
     strcpy(name,strcat(name,sFirst));

     pS = new MasterStudent(name,uid,nGrade,type);
     cout <<"student #" << ++ i <<":" << * pS;
     delete pS;

     name[0] = 0;      //将 name 中内容置空串
     fin>>sLast >>sFirst >>uid >>nGrade >>type;
   }
}
```

运行结果为：

```
student #1:Dill Arnson          12567, 3.5, A
student #2:Welch Shammas        12667, 4.1, A
student #3:Portel Braumbel      12579, 3.8, B
```

程序中先将文件的一行分5个数据项读入(一条记录)，然后将前两项组合成一个名字。用名字name，学号uid，成绩nGrade，类别type这四项初始化一个MasterStudent堆对象，在屏幕上输出。随后，继续这一过程，直到文件尾。文件尾的判断标志为fin.eof()成员函数。1表示文件尾，0表示未到文件尾。

小结

C的I/O是丰富、灵活和强大的，但是，C的I/O系统一点也不了解对象，不具有类型的安全性。C++的I/O流扬弃了C的I/O系统，它操作更简捷，更易理解，它使标准I/O流、文件流和串流的操作在概念上统一了起来。有了控制符，C++更灵活。由它所重载的插入运算符完全融入了C++的类及其继承的体系。

本章介绍了文件操作的文本顺序读写。文件的其他操作一样可以完成，进一步的学习请看其他参考书。

练习

19.1 编写一个程序，它读入一个文件，并统计文件中的行数。

19.2 改写第19.7节中RMB类的重载插入运算符，使得人民币输出格式为：长度一律10位，小数两位，币值前有一¥符号。

19.3 设字符串string ="1 2 3 4 5 6 7 8 9 "，用串流I/O的方法编程逐个读取这个串的每个数，直到读完为止，并在屏幕上输出。

19.4 设置格式，显示ABCDE…Z这26个字母的ASCII码大写字母16进制数。

19.5 实现一个通讯录打印程序。通讯录的记录格式为：

姓名，单位，电话，住址，宅电

要求先建立Person类，然后按对象读入和打印。

第20章 模　板

模板是C++语言相对较新的一个重要特性。模板使程序员能够快速建立具有类型安全的类库集合和函数集合，它的实现，方便了更大规模的软件开发。本章介绍了模板的概念、定义和使用模板的方法，通过这些介绍，使读者有效地把握模板，以便能正确使用C++系统中日渐庞大的标准模板类库。

20.1 模板的概念

若一个程序的功能是对某种特定的数据类型进行处理，则将所处理的数据类型说明为参数，就可把这个程序改写为模板。模板可以让程序对任何其他数据类型进行同样方式的处理。

C++程序由类和函数组成，模板也分为类模板(class template)和函数模板(function template)。因此，可以使用一个带多种不同数据类型的函数和类，而不必显式记忆针对不同的数据类型的各种具体版本。

函数模板的一般定义形式是：

```
template<类型形式参数表> 返回类型 FunctionName ( 形式参数表 )
{
  //函数定义体
}
```

其中的类型形式参数表可以包含基本数据类型，也可以包含类类型。如果是类类型，则须加前缀class。

这样的函数模板定义，不是一个实实在在的函数，编译系统不为其产生任何执行代码。该定义只是对函数的描述，表示它每次能单独处理在类型形式参数表中说明的数据类型。

当编译系统发现有一个函数调用：

```
FunctionName ( 实在参数表 );
```

将根据实在参数表中的类型，确认是否匹配函数模板中对应的形式参数表，然后生成一个重载函数。该重载函数的定义体与函数模板的函数定义体相同，而形式参数表的类型则以实在参数表的实际类型为依据。该重载函数称模板函数(template function)。

→ 函数模板与模板函数的区别

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，它由编译系统在碰见具体的函数调用时所生成，具有程序代码。

类模板的一般说明形式是：

```
template<类型形式参数表> class className
```

```
{
  //类声明体
};

template<类型形式参数表>
返回类型 className<类型名表>::MemberFuncName1 ( 形式参数表 )
{
  //成员函数定义体
}

template<类型形式参数表>
返回类型 className<类型名表>::MemberFuncName2 ( 形式参数表 )
{
  //成员函数定义体
}

...

template<类型形式参数表>
返回类型 className<类型名表>::MemberFuncNameN ( 形式参数表 )
{
  //成员函数定义体
}
```

其中的类型形式参数表与函数模板中的意义一样。后面的成员函数定义中，class-Name<类型名表>中的类型名表，是类型形式参数的使用。

这样的一个说明(包括成员函数定义)，不是一个实实在在的类，只是对类的描述，称为类模板(class template)。

建立类模板之后，可用下列方式创建类模板的实例：

```
className<类型实在参数表> object;
```

其中类型实在参数表应与该类模板中的类型形式参数表匹配。class_name<类型实在参数表>是模板类(template class)，object 是该模板类的一个对象。

→ 类模板与模板类的区别。

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。

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。

20.2 为什么要用模板

1. 关于函数

考察两个 swap()函数，一个交换两个整型数，另一个交换两个浮点数。两个 swap ()的主体行为是一样的，无非一个是处理 int 型，一个是处理 float 型。两个函数分别如下：

```
void swap(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
```

```
}

swap(float& a, float& b)
{
   float temp = a;
   a = b;
   b = temp;
}
```

交换任何一对类类型对象，可以定义如下：

```
void swap(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

能不能对于任一类型 T 的两个对象 x1，x2，函数调用 swap(x1，x2)总能使编译系统理解其交换意义而予以实现呢？若不然，每交换一对新类型的对象，都要定义一个执行同样操作的重载函数。

有了函数模板之后，重载就不必要了。

2. 关于类

再来考察一个链表类。对于 Cat 类的链表，我们有：

```
class Cat
{
   //...
};

class CatList
{
   public:
     List();
     void Add(Cat&);
     void Remove(Cat&);
     Cat * Find(Cat&);
     ~List();
   private:
     //...
};
```

该链表类将 Cat 类对象作为链表结点，进行插入、删除和查找处理。所以称之为 CatList 类。

同样如果想处理其他的任何一种类类型的对象作为结点的链表，我们必须重新对这链表进行定义。这种工作很繁冗，因为所定义的类的行为没有任何变化，只是处理的结点之类型有所不同。

如果让类模板来工作，就能最大限度地解决这些问题。

20.3 函数模板

对于具有各种参数类型，相同个数、相同顺序的同一函数（重载函数），如果用宏定义来写：

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

则它不能检查其数据类型，损害了类型安全性。这也是为什么要使用函数模板的一个原因。

另外，如果一个个地定义其函数重载，例如，求同一数据类型数值中的最大值：

```
int max(int a, int b)
{
   return a>b?a:b;
}
float max(float a, float b)
{
   return a>b?a:b;
}
```

对于与整数相容的 char 类型数据值的调用，也不能得到满意的结果。如调用：

```
max('3', '5');
```

则编译系统会为其找到一个 int 型的匹配，调用“int max(int a,int b);”函数，但是它将返回 53 而不是'5'（其 ASCII 码是 53）。

用函数模板可将许多重载函数简单地归为一个，如下例所示：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch20_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>

template<class T>
T max(T a,T b)
{
   return a>b?a:b;      //T类的>操作须有定义
}

int main()
{
   cout <<"Max(3,5) is "
        <<max(3,5) <<endl;
   cout <<"Max('3','5') is "
        << max('3','5') <<endl;
}
```

运行结果为：

```
Max(3,5) is 5
Max('3','5') is 5
```

当编译发现用指定数据类型调用函数模板时，就创建一个模板函数。

上例中，当编译程序发现 max(3,5)调用时，它就产生了一个如下的函数定义，生成其程序代码：

```
int max(int a,int b)
{
   return a>b?a:b;
}
```

当发现 max('3','5')调用时，它又产生另一个如下的函数定义，也生成其程序代码：

```
char max(char a, char b)
{
   return a>b?a:b;
}
```

这样，实参是什么数据类型，返回值也是什么数据类型，不会出现前面的问题。而且模板又避免了相同操作的重载函数定义。

20.4 重载模板函数

可以像重载普通函数那样重载模板函数。

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch20_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<string.h>

template<class T> T max(T a,T b)
{
   return a>b?a:b;
}

char * max(char * a,char * b)
{
   return (strcmp(a,b)?a:b);
}

int main()
{
   cout <<"Max(\"Hello\",\"Gold\") is "
        << max("Hello","Gold") <<endl;
}
```

运行结果为：

```
Max("Hello","Gold") is Hello
```

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。

编译程序看到 max("Hello","Gold")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。

20.5 类模板的定义

链表操作并不依赖于要处理的链表的数据类型(如在上面所说的 Cat 类)。定义类模板时，就是利用了这种独立性。链表操作时，只是把要处理的数据类型当作参数。一个类模板用于构筑一个通用链表。如，整数链表，结构链表，以致任何其他定义过的数据类型的链表。上节描述的 CatList 类也可以通过通用链表来构筑。

用类模板来定义一个通用链表，此时该通用链表还不是一个类定义，只是类定义的一个框架，即类模板。

下例定义了一个单向链表的模板类，它分别实现增加、删除、寻找和打印操作。见图 20-1。

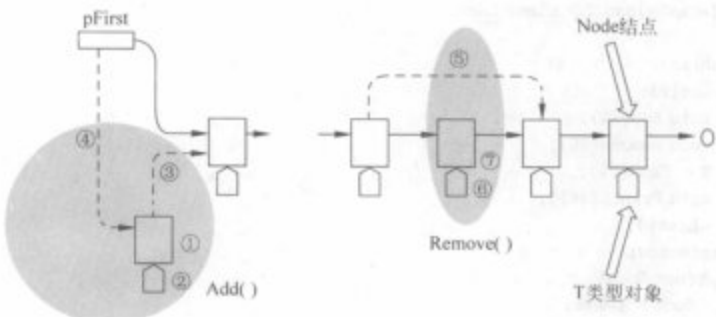


图 20-1 单向通用链表操作示意图

增加时，Add()成员函数在链首挂接上一个携有 T 类型对象的结点，使之成为链首结点，由下列步骤①，②，③，④完成：

① 从堆空间申请一个结点；

② 将 T 类对象挂接在这个结点上；

③ 将该结点指向链首的结点；

④ 将该结点成为链首(链首指针 pFirst 指向它)。

删除时，Remove()成员函数寻找到挂接该对象的结点，先脱链，再删除结点。由下列步骤⑤，⑥，⑦完成：

⑤ 链中待删结点前后的两个结点链接起来，以使待删结点脱链；

⑥ 删除待删结点上的 T 类对象(将空间还给堆)；

⑦ 删除待删结点(将空间还给堆)。

如果找不到对应结点，则无功而返；如果找到的是链首结点，则步骤⑤的脱链，由链首指针 pFirst 指向下一个结点来完成；删除结点须先将挂接的对象删除。

寻找时，Find()成员函数从链首开始遍历整个链表，查找是否有结点含有对应对象，若无，则返回空指针。

打印时，PrintList()成员函数从链首开始遍历整个链表，打印每个结点下的对象值，因而要求 T 类型的输出运算符须有定义。

链表类对象包含唯一的一个数据成员 pFirst，刚创建时，pFirst 指向空，链表也为空。当进行过增加操作时，链表非空，而 pFirst 指向链首。所有成员函数的操作，都从 pFirst 着手。

链表类析构的任务是把链表中所有的结点空间收回，包括挂接在结点下的 T 类对象。它从链首开始遍历整个链表，处理结点时，先删除挂接的对象，再删除结点，然后去处理下一个结点。代码如下：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              listtmp.h                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include<iostream>

template<class T> class List
{
   public:
     List();
     void Add(T&);
     void Remove(T&);
     T* Find(T&);
     void PrintList();
     ~List();
   protected:
    struct Node{
      Node* pNext;
      T* pT;
    };
    Node *pFirst;      //链首结点指针
};

template<class T> List<T>::List()
{
   pFirst = 0;
}

template<class T>
void List<T>::Add(T& t)
{
  Node* temp = new Node;      //①
  temp->pT = &t;      //②
  temp->pNext = pFirst;      //③
  pFirst = temp;      //④
}

template<class T>
void List<T>::Remove(T& t)
{
```

```
    Node * q = 0;       //用来定位待删的结点
    if ( * (pFirst->pT) == t)      //T 类中 == 须有定义
    {
     q = pFirst;
     pFirst = pFirst->pNext;      //待删结点在链首时的脱链
    }
    else
    {
     for(Node * p = pFirst; p->pNext; p = p->pNext)      //顺链查找
       if ( * (p->pNext->pT) == t)
       {
         q = p->pNext;
         p->pNext = q->pNext;      //⑤
         break;
       }
    }
    if(q){
     delete q->pT;      //⑥
     delete q;      //⑦
    }
}

template<class T>
T * List<T>::Find(T& t)
{
   for(Node * p = pFirst; p; p = p->pNext)
   {
    if( * (p->pT) == t)
      return p->pT;
   }
   return 0;
}

template<class T>
void List<T>::PrintList()
{
   for(Node * p = pFirst; p; p = p->pNext)
   {
    cout << * (p->pT)<<" ";    //须有 T 的友元处理 T 对象输出
   }
   cout <<endl;
}

template<class T>
List<T>::~List()
{
   Node * p;
   while(p = pFirst)
   {
    pFirst = pFirst->pNext;
    delete p->pT;
    delete p;
   }
}
```

上例模板类中，嵌套了一个 Node 结构类型。Node 结构变量是通用链表类中的链表结点，只在通用链表类的范围内操作，所以，该定义嵌套在模板类中。

20.6 使用类模板

使用类模板的方法为：

(1) 在程序开始的头文件中说明类模板的定义。

(2) 在适当的地方创建一个模板类的实例，编译发现正在创建一个类模板的对象时，便会创建该模板类的定义，同时，创建相应的对象实体。

(3) 有了对象名，以后的使用就和通常一样。但要记住，你规定了什么类型的模板类，在使用成员函数时，所赋的实参也要对应该类型。

例如：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch20_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include "listtmp.h"

int main()
{
   List<float> floatList;
   for(int i = 1; i<7; i++)
   {
    floatList.Add( * new float(i + 0.6));
   }
   floatList.PrintList();
   float b = 3.6;
   float * pa = floatList.Find(b);
   if (pa)
    floatList.Remove( * pa);
   floatList.PrintList();
}
```

运行结果为：

```
6.6 5.6 4.6 3.6 2.6 1.6
6.6 5.6 4.6 2.6 1.6
```

上例类模板定义(包含其成员函数定义)都在头文件中，这与一般类定义的方法不同。一般类定义时，类定义部分作为界面放在头文件中，成员函数定义部分作为实现放在 cpp 文件中。但作为模板，在应用程序中，编译发现 List<float> floatList; 这样的声明时，要为其生成模板类的实实在在的定义，所以模板中必须包含整个模板(包含其成员函数定义)的完整定义，在生成模板类之后，系统又为其创建该类的对象。

创建了模板类对象，就是创建了一个链表，这时链表为空。

程序执行了 Add() 操作后，链表便一个个长起来。执行 Add()时，程序先从堆中申请分配 float 变量空间，初始化，然后作为参数传递给 Add()，Add() 马上从堆中申请分配结点

空间，挂接该 float 变量，最后作为链首链入链表中。

程序创建了 float 变量并赋值，然后以此作为实际参数寻找链表中等于该值的 float 变量。如果找到，将该找到的变量作为引用参数调用 Remove ()成员函数。

→ 这里不能直接以赋过值的 float 变量 b 为参数调用 Remove()，因为 Remove() 参数是待删除的指定 float 变量的引用。而 float 变量 b 则是区别于待删除的另一个变量，该变量不存在于链表结点挂接的变量中，b 与真正要删除的变量之间只是变量值相等的关系。

20.7 使用标准模板类库:Josephus 问题

标准模板类库 STL(Standard Template Library)是一个基于模板的容器类库，它包括向量、链表、队列和栈，还包括了一些通用的算法，如排序和查找等。它已经成为 C++标准。使用标准模板类库的好处是：可以避免自己在开发模板类库时，不同模板类之间的功能重复；最大限度的类库重用；作为 C++标准，可移植性强是不言而喻的。面向对象程序设计和面向对象数据库设计时，大量用到容器类库，所以，标准模板类库对 C++的发展来说，影响将是巨大的。

我们使用 BC++ 5.0 中提供的标准模板类库，用其中的单向链表类模板来求解 Josephus 问题。

该模板类库的单向链表模板类名为 TIListImp<T>，其中 T 表示由链表处理的对象类型，内有：

- 链首指针数据成员(本例中未直接用到)；
- Add 成员函数　　void Add(Object&)；
  将携有 Object 对象的结点挂接在链首；
- Detach 成员函数　　void Detach(Object&)；
  使携有 Object 对象的结点脱链；
- 单向链表迭代算子 TIListIteratorImp<T>作为友类。

单向链表模板类的操作很像上例的通用链表，只不过模板类名是标准模板类名，使用时，只要声明该标准模板类名，创建其对象，了解并操作其成员函数。成员函数的操作示意见图 20-2。Add()和上例一样，也是将结点挂接在链首。Detach()与上例不一样之处在于脱链之后，结点不予删除，所有结点和 T 类对象都由上层函数来统一析构。

对应的迭代算子模板类名为 TIListIteratorImp<T>，内有：

- Current 成员函数　　Object * Current()；
  返回当前 Object 对象的指针；
- Restart 成员函数　　void Restart()；
  将链首结点中的对象成为当前对象；
- operator ++ 成员函数　　Object& operator ++ ()；

返回下一个对象的指针，若当前指针指向链尾，则返回 0。

迭代算子作为友类相伴单向链表，它操作的对象是链表结点所携带的 T 类型对象，算子即运算符。运用迭代算子的好处是不用关心链表操作的细节而直接对所关心的 T 类型进行来回操作。迭代算子操作示意图见图 20-2。

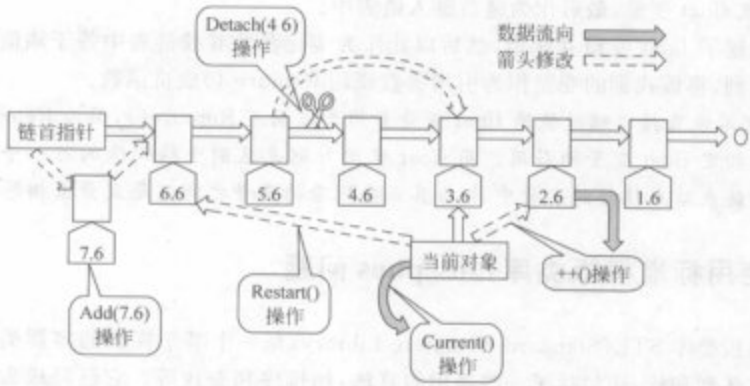


图 20-2　单向链表模板类操作示意

标准模板类和迭代算子有许多有用的成员函数，这里没有用到的都省略。

下例是使用标准模板类库的一个 Josephus 问题解法：

```
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* *              Josephus 问题解法六                          * *
//* *              jose6.cpp                                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream>
#include<iomanip>
#include<classlib\listimp>

void Display(int);
void Step(int m);
int * Init(int);

TIListImp<int> josephus;      //创建单向链表模板类的全局对象
TIListIteratorImp<int> iter(josephus);   //创建 josephus 迭代算子
int * curpos;      //当前小孩位置


int main()
{
   int n = 10, s = 4, m = 4;   //随取三个合理的整数
   int * ap = Init(n);
   Step(s);
   for(int i = 1; i<n; i ++ )
   {
    Step(m);
    Display( * curpos);
    josephus.Detach(curpos);
   }
   cout <<"\nThe winner is "<< * iter.Current() << endl;
   delete[ ]ap;
}

//以下是构造链表，初始化小孩编号的函数
```

```
int * Init(int n)
{
   int * a = new int[n];
   for(int i = 0; i<n; i ++ )
   {
    a[i] = n - i;
    Display(i + 1);
    josephus.Add(&a[i]) ;
   }
   Display ( -1 );
   iter.Restart();        //初始化迭代算子
   return a;
}

//以下是显示一个小孩编号的函数
void Display(int n)
{
   static int k;
   if(n<0)
   {
    k = 0;
    cout <<endl;
    return ;
   }
   cout <<setw(4) <<n;
   if (!( ++k % 10))
    cout <<endl;
}

//以下是要将指针从当前位置挪到往下数第 m 个小孩的位置
void Step(int m)
{
  for(int i = 0; i<m; i ++ )
  {
    curpos = iter.Current();
    iter ++ ;
    if(!iter.Current())      //模拟环链表操作
      iter.Restart();
  }
}
```

运行结果为：

```
1   2   3   4   5   6   7   8   9   10
8   2   6   1   7   4   3   5   10
The winner is 9
```

小结

模板是一种安全的、高效的重用代码的方式。它被用于参数化类型，在创建对象或函数时所传递的类型参数可以改变其行为。

每个模板类的实例是一个实际的对象，可以像其他对象一样使用，甚至可以作为函数的参数，或作为返回值。

与类和函数的定义不同，类模板和函数模板的定义一般放在头文件中。

在C++中，一个发展趋势是使用标准模板类库(STL)，VC和BC都把它作为编译器的一部分。STL是一个基于模板的包容类库，包括向量、链表和队列，还包括一些通用的排序和查找算法等。

STL的目的是为替代那些需要重复编写的通用程序。当理解了如何使用一个STL类之后，在所有的程序中不用重新编写就可以使用它。

练习

20.1 编写一个函数模板，它返回两个值中的最小者。但同时要确保能正确处理字符串。

20.2 以下是一个整数栈类的定义：

```
const int SIZE = 100;
class Stack
{
   public:
     Stack();
     ~Stack();
     void Push(int n);
     int Pop();
   private:
     int stack[SIZE];
     int tos;
};
```

编写一个栈的类模板(包括其成员函数定义)，以便为任何类型的对象提供栈结构数据操作。

在应用程序中创建整数栈、字符栈和浮点数栈，并提供一些数据供进栈、退栈和打印操作。

第21章 异常处理

在编写程序时，应该试着确定程序可能出现的错误，然后加入处理错误的代码。例如，当程序执行文件I/O操作时，应测试文件打开以及读写操作是否成功，并且在出现错误时作出正确的反应。随着程序复杂性的增加，为处理错误而必须包括在程序中代码的复杂性也增加了。为使程序更易于测试和处理错误，C++实现了异常处理机制。本章介绍了C++异常处理。程序使用try、throw和catch语句来支持异常处理。

21.1 异常的概念

在大型软件开发中，最大的问题就是错误连篇的、不稳定的代码。而在设计与实现中，最大的开销是花在测试、查找和修改错误上。

程序的错误，一种是编译错误，即语法错误。如果使用了错误的语法、函数、结构和类，程序就无法被生成运行代码。另一种是在运行时发生的错误，它分为不可预料的逻辑错误和可以预料的运行异常。

逻辑错误是由于不当的设计造成的，如，某个排序算法不合适，导致在边界条件下，不能正常完成排序任务。一般只有当用户做了某些出乎意料的事才会出现逻辑错误，这些错误，安静的潜伏着，连许多大型的优秀软件都不能避免。就像大战之后残留的地雷，在“一切正常”中，突然某人进入了误区，程序发生了“爆炸”。一旦发现了逻辑错误，专门为其写一段处理错误的代码，就可避免错误的发生，比如数组下标溢出检查，这样错误就防范在先了。

运行异常，可以预料，但不能避免，它是由系统运行环境造成的。如，内存空间不足，而程序运行中提出内存分配申请时，得不到满足，就会发生异常；在硬盘上的文件被挪离，或者光盘没有放好，导致程序运行中文件打不开而发生异常；程序中发生除0的代码，导致系统除0中断；打印机未打开，调制解调器掉线等，导致程序运行中挂接这些设备失败，等等。这些错误会使程序变得脆弱。然而这些错误是能够预料的，通常加入一些预防代码便可防止这些异常。如，对文件打不开时的保护：

```
#include<fstream.h>

//...
void f(char * str)
{
  ifstream source(str);          //打开 str 串中的文件
  if(source.fail())              //打不开
  {
    cerr <<"Error opening the file: "<<str <<endl;
    exit(1);                  //退出程序
  }
  //...
}
```

异常是一种程序定义的错误，它对程序的逻辑错误进行设防，对运行异常加以控制。C++中，异常是对所能预料的运行错误进行处理的一套实现机制。

21.2 异常的基本思想

在小型程序中，一旦发生异常，一般是将程序立即中断运行，从而无条件释放所有资源。对于大型程序来说，运行中一旦发生异常，应该允许恢复和继续运行。恢复的过程就是把产生异常所造成的恶劣影响去掉，中间可能要涉及一系列的函数调用链的退栈，对象的析构，资源的释放等。继续运行就是异常处理之后，在紧接着异常处理的代码区域中继续运行。在C++中，异常是指从发生问题的代码区域传递到处理问题的代码区域的一个对象。见图 21-1。

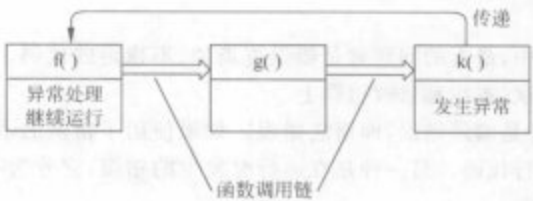


图 21-1 异常的发生、传递与处理

发生异常的地方在函数 k()中，处理异常的地方在其上层函数 f()中，处理异常后，函数 k()和 g()都退栈，然后程序在函数 f()中继续运行。如果不用异常处理机制，在程序中单纯地嵌入错误处理语句，要实现这一目的是艰难的。

异常的基本思想是：

(1) 实际的资源分配(如内存申请或文件打开)通常在程序的低层进行，如图 21-1 中的 k()。

(2) 当操作失败、无法分配内存或无法打开一个文件时，在逻辑上如何进行处理通常是在程序的高层，如图 21-1 中的 f()，中间还可能有与用户的对话。

(3) 异常为从分配资源的代码转向处理错误状态的代码提供了一种表达方式。如果还存在中间层次的函数，如图 21-1 中的 g()，则为它们释放所分配的内存提供了机会，但这并不包括用于传递错误状态信息的代码。

从中可以看出，C++异常处理的目的，是在异常发生时，尽可能地减小破坏，周密地善后，而不去影响其他部分程序的运行。这在大型程序中是非常必要的。对如图 17-1 所示的程序调用关系，如果像上一节处理文件打开失败异常的方法，那么，异常只能在发生的函数 k()中进行处理，无法直接传递到函数 f()中，而且调用链中的函数 g()的善后处理也十分困难。

21.3 异常的实现

使用异常的步骤是：

(1) 定义异常(try 语句块)。将那些有可能产生错误的语句框定在 try 块中；

(2) 定义异常处理(catch 语句块)。将异常处理的语句放在 catch 块中，以便异常被传递过来时就处理它；

(3) 抛掷异常(throw 语句)。检测是否产生异常，若是，则抛掷异常。

例如，下面的程序，设置了防备文件打不开的异常：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch21_1.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<fstream.h>
#include<iostream.h>
#include<stdlib.h>

int main(int argc, char * * argv)
{
   ifstream source(argv[1]);      //打开文件
   char line[128];
   try
   {
    if(source.fail())
      throw argv[1];
   }

   catch(char * s)
   {
    cout <<"error opening the file "<<s <<endl;
    exit(1);
   }

   while(!source.eof())
   {
    source.getline(line, sizeof(line));
    cout <<line <<endl;
   }
   source.close();
}
```

假定 C 盘中无 abc.txt 文件，有 xyz.txt 文件，内容为两行问候句子，则运行结果为：

```
c:\>ch21_1 abc.txt
error opening the file abc.txt
c:\> ch21_1 xyz.txt
hello!
How are you?
```

这里抛掷异常(throw argv[1])与处理异常(catch 块)在同一个函数中。当打开文件失败时，就执行"throw argv[1];" 语句，throw 后面的表达式 argv[1]的类型被称为所引发的

异常之类型。

try 块结构表示块中的语句可能会发生异常，放在其中加以监控。**C++只理会受监控的过程之异常。**

在 try 块之后必须紧跟一个或多个 catch()语句，目的是对发生的异常进行处理。catch()括号中的声明只能容纳一个形参，当类型与抛掷异常的类型匹配时，该 catch()块便称捕获了一个异常而转到其块中进行异常处理。

程序中如果没有发生异常，即 source.fail()为逻辑假，则将继续执行：

```
while(!source.eof())
{
  source.getline(line, sizeof(line));
  cout <<line <<endl;
}

source.close ( );
```

如果发生了异常，即 source.fail()为逻辑真，则抛掷的异常 throw argv[1] 将被

```
catch(char * s)
{
   cout <<"error opening the file " <<s <<endl;
   exit(1);
}
```

捕获，最后以执行 exit(1)；而告终。

可以将抛掷异常与处理异常放在不同的函数中。

例如，下面的程序定义一个除零异常：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch21_2.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include<iostream.h>

double Div ( double , double );

int main ( )
{
   try
   {
    cout <<"7.3/2.0 = " <<Div(7.3, 2.0) <<endl;
    cout <<"7.3/0.0 = " <<Div(7.3, 0.0) <<endl;
    cout <<"7.3/1.0 = " <<Div(7.3, 1.0) <<endl;
   }

   catch(double)
   {
    cout <<"except of deviding zero.\n";
   }

   cout << "That is ok. \n";
}
```

```
double Div(double a, double b)
{
   if(b = = 0.0)
    throw b;

   return a/b;
}
```

运行结果为：

```
7.3/2.0 = 3.65
exception of dividing zero.
That is ok.
```

语句“cout <<"7.3/1.0=" <<Div(7.3, 1.0) <<endl;”没有被执行。

当调用函数表达式：

```
Div(7.3, 0.0)
```

时，控制转移到函数 Div()内执行，这时，b==0.0 为逻辑真，被除 0 在数学上是无意义的，所以，程序中定义为异常。

由于发生了异常，函数 Div()被退栈处理，紧跟调用函数 Div()后面的语句：

```
cout <<"7.3/1.0 = " <<Div(7.3, 1.0) <<endl;
```

不再被执行。异常被

```
catch(double)
{
  cout << "except of deviding zero.\n";
}
```

所捕获，执行完异常处理，程序紧接着执行异常处理后面的语句：

```
cout <<"That is ok. \n";
```

如果程序中不发生异常，try 语句块中的第二条语句改为“Div(7.3,1.5);”，则运行结果为：

```
7.3/2.0 = 3.65
7.3/1.5 = 4.86667
7.3/1.0 = 7.3
That is ok.
```

程序在执行完 try 语句块之后，紧接着就执行 catch()语句块后面的语句。

21.4 异常的规则

以 catch 开始的程序块是异常处理程序，编写异常处理程序的规则是：

(1) 任意数量的 catch 分程序立即出现在 try 分程序之后。在 try 分程序出现之前，不能出现这些 catch 程序块。例如：

```
int j;
double d;
char str[20];

class Coords
{
   public:
    Coords(double a, double b);
    //...
};

class String
{
  public:
    String(char * );
    //...
};

void f()
{
   try
   {
    //...
    throw 10;
    //...

    throw j;          //j 为 int 型
    //...
    throw d;          //d 为 double 型
    throw "abc";
    //...

    throw str;        //字符串
    //...

    throw Coords(1.0, 3.0);
    //...

    throw String("def");
   }

   catch(int k)
   {
//...
   }

   catch(double x)
   {
   //...
   }

   catch(char * ptr)
   {
//...
```

```
    }

    catch(Coords c)
    {
//...
    }

    catch(String s)
    {
//...
    }

    cout <<"That is ok.\n";
}
```

在上例中 catch 程序块必须出现在 try 块之后，并且在 try 块之后可以出现多个 catch 程序块。

(2) 在 catch 行的圆括号中可包含数据类型声明，它与函数定义中参数声明起的作用相同。应把异常处理 catch 块看作是函数分程序。**跟在 catch 之后的圆括号中必须含有数据类型，捕获是利用数据类型匹配实现的。**在数据类型之后放参数名是可选的。参数名使得被捕获的对象在处理程序分程序中被引用。

在上例中，抛掷异常：

"throw 10;" 和"throw j;" 被 catch(int j)的处理程序捕获；

"throw d;"被 catch(double x)捕获；

"throw "abc";"被 catch(char * ptr)捕获；

"throw Coords(1.0,3.0);"被 catch(Coords c)捕获；

"throw String("def");"被 catch(String s)捕获。

捕获的原因是抛掷的数据类型与异常处理程序的数据类型相匹配。

我们在程序 ch21_2.cpp 中已经看到 catch(double)的声明中，参数名是省略的，在该异常处理中，没有用到参数名。

(3) 如果一个函数抛掷一个异常，但在通往异常处理函数的调用链中找不到与之匹配的 catch，则该程序通常以 abort()函数调用终止。

在上例中，在 try 块中，如果我们增加一个抛掷异常：

```
throw 'w';
```

则由于数据类型不匹配，而未能被任何 catch 块捕获，这时，系统用它自己的默认异常处理程序 abort()来做这项工作。

→ 在函数调用中，实参与形参可以通过相容类型的类型提升或自动转换来匹配：

```
void g(int b)
{
  //...
}

void func()
{
```

```
    g('w');
    //...
}
```

g('w')将能顺利地匹配函数 g(int b),但是抛掷异常与异常处理程序之间,是按数据类型的严格匹配来捕获的。不允许类型转换,甚至下列代码:

```
try
{
  throw 20;
}

catch(unsigned int)
{
  //...
}
```

由于常量 20 的数据类型是 int,而不是 unsigned int,所以该抛掷异常无法被 catch (unsigned int)所捕获。

(4) 如果 catch 分程序执行完毕,则跟随最后 catch 分程序的代码(如果有的话)就被执行。

例如,上面例子中函数 f()的最后一条语句:

```
cout <<"That is ok.\n";
```

就是跟随最后 catch 分程序的代码。

21.5 多路捕获

多数程序可能有若干不同种类的运行错误,它们可以用异常处理机制。每种错误可与一个类、一个数据类型或一个值有关。这样,在程序中就会出现多路捕获。

例如,操作 String 类对象时,预设两个异常:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch21_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include<iostream.h>
#include<string.h>

class String
{
  public:
    String(char*, int);

    class Range      //异常类 1
    {
      public:
        Range(int j):index(j){}
        int index;
    };
```

```
    class Size{};      //异常类 2

    char& operator[](int k)
    {
      if(0<=k && k<len)
        return p[k];

      throw Range(k);
    }

  private:
    char * p;
    int len;
    static int max;
};

int String::max = 20;

String::String(char * str, int si)
{
  if(si<0 || max<si)
    throw Size();

  p = new char[si];
  strncpy(p, str, si);
  len = si;
}

void g(String& str)
{
  int num = 10;
  for(int n = 0; n<num; n++)
    cout <<str[n];
  cout <<endl;
}

void f()
{
  //代码区 1

  try
  {
    //代码区 2

    String s("abcdefghijklmnop", 10);
    g(s);
  }

  catch(String::Range r)
  {
    cerr <<" ->out of range: " <<r.index <<endl;

    //代码区 3
```

```
    }

    catch(String::Size)
    {
      cerr <<"size illegal!\n";
    }

    cout <<"The program will be continued here.\n\n";

    //代码区 4
  }

  int main()
  {
    //代码区 5

    f();
    cout <<"These code is not effected by probably exception in f().";
  }
```

运行结果为：

```
abcdefghij
The program will be continued here

These code is not effected by probably exception in f().
```

如果在代码区 2 中，构造 String 对象的定义为：

```
String s("abcdefghijklmnopqrstuvwxyz", 26);
```

则运行结果为：

```
size illegal!
The program will be continued here

These code is not effected by probably exception in f().
```

类 String 包含了类 Range 和一个空类 Size 的定义，它们是嵌套类。设置 Size 的唯一目的就是为了作为异常类来抛掷。如果下标值超出范围，[]运算符重载就抛掷一个 Range 异常。

→ 在一个类定义的内部定义一个类，称为嵌套类。

嵌套类的成员函数和静态成员可以在包含该类的外部定义，但嵌套类的作用域在包含该类定义的内部。

如果 String 构造函数的参数 si 不在给定的数值范围内，它就抛掷一个 Size 异常。当函数 f()开始执行时，就首先执行标志为代码区 1 的代码。

如果代码区 1 抛掷一个异常，它只能被系统默认的异常处理程序 abort()所捕获，导致终止运行。因为它不在用户定义的异常区域。

在代码区 1 执行之后(没有发生异常)，在 try 块中代码区 2 就开始执行。如果这段代码直接或间接地抛掷一个异常，若该异常的数据类型与 catch 参数的数据类型匹配，跟在 try 块之后的 catch 块就可捕获这个异常。如果它未被捕获，该异常又只能被系统默认的异常处理程序 abort()捕获。

如果其中有一个 catch 块实际地捕获了所抛掷的异常，那么该 catch 块的代码就执行。如果这段代码没有执行终止运行的语句，没有返回语句，也没有再抛掷语句，那么就执行代码区 4 的代码。如果 catch 抛掷一个异常 throw，则由系统默认的异常处理程序来捕获。

异常处理完后，就执行代码区 4 的代码，以后，函数 f()正常结束。如果代码区 4 中有抛掷异常，则只能被系统默认的异常处理程序捕获。因为它也不在用户定义的异常区域内。

在主程序中，如果在执行代码区 5 的代码时，抛掷一个异常，则由系统默认的异常处理程序捕获。因为它不是用户定义的异常区域。

在调用 f()中，可能发生抛掷异常，被程序定义的异常处理程序所捕获。但是，从 f()返回后，其他部分的代码继续运行，不受影响。

→ catch(Size)并没有对抛掷的 Size 对象指派名字。因为 Size 对象不含数据成员，也没有必要给捕获对象一个名字。

21.6 异常处理机制

在处理程序和语句之间的相互作用使异常在大型应用程序中变得复杂。通常人们希望抛掷被及时捕获，以避免程序突然终止。此外，跟踪抛掷很重要，因为捕获确定该程序的后继进展。例如，抛掷和捕获可以用来重新开始程序内的一个过程，或者从应用程序的一部分跳到另一部分，或者回到菜单。

例如，下面的代码说明了异常处理机制：

```
void f ( )
{
  try
  {
    g();
  }

  catch(Range)
  {
    //...
  }

  catch(Size)
  {
    //...
  }

  catch( ... )
  {
    //...
  }
}

void g()
{
  h();
}
```

```
void h()
{
  try
  {
    h1();
  }
  catch(Size)
  {
    //...
    throw 10;
  }

  catch(Matherr)
  {
    //...
  }
}

void h1()
{
  //...
  throw(Size);
  try
  {
    //...
    throw Range;

    h2();
    h3();
  }

  catch(Size)
  {
    //...
    throw;
  }
}

void h2()
{
    //...
    throw Matherr;
}

void h3()
{
    //...
    throw Size;
}
```

处理程序的模式可由函数调用链中的异常处理来描述，见图 21-2。它显示包含 try 和异常处理程序的各个函数。它们使程序员能确定在一个应用程序中抛掷和捕获的模式。

在图 21-2 中每个函数以方框形式出现。每个方框分为两部分。左边部分表示该函数

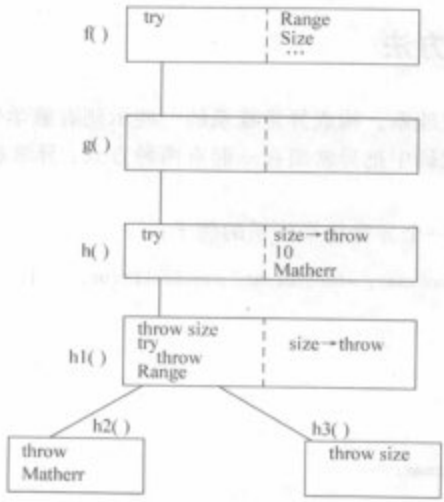


图 21-2　在函数调用链中的异常处理

是否包含一个 try 分程序，它也指出在 try 之前或在最后的异常处理程序之后的所有显式抛掷。在 try 块中的所有显式抛掷语句被表示成在 try 之下把 throw 缩进的形式。

方框的右边部分通过 catch 的数据类型列出各异常处理。图中表明一个异常处理中是否执行重新抛掷。重新抛掷时，处理程序后面是一个箭头和被抛掷对象的数据类型。

函数 f()中的 catch(...) 块，参数为省略号，定义一个"默认"的异常处理程序。通常这个处理程序应在所有异常处理块的最后，因为它与任何 throw 都匹配，目的是为避免定义的异常处理程序没能捕获抛掷的异常而使程序运行终止。

函数 h()中的 catch(Size) 块，包含有一个抛掷异常语句 throw 10，当实际执行这条语句时，将沿着调用链向上传递，被函数 f()中的 catch(...) 所捕获。如果没有 f()中的 catch(...)，那么，异常将被系统的 terminate() 函数调用，后者按常规再调用 abort()。

函数 h1()中的抛掷 throw Size，由于不在本函数的 try 块中，所以只能沿函数调用链向上传递，结果被 h()中的 catch(Size)捕获。

函数 h1()中的抛掷 throw Range，在 try 块中，所以首先匹配 try 块后的异常处理程序，可是没有被捕获，因而它又沿函数调用链向上，在函数 f()中，catch(Range)块终于捕获了该抛掷。

函数 h1()中的 catch(Size)块，包含一个抛掷 throw，没有带参数类型，它表示将捕获到的异常对象重新抛掷出去，于是，它将沿函数调用链向上传递，在 h()中的 catch(Size) 块，捕获了该抛掷。

函数 h2()中的抛掷 throw Matherr，首先传递给 h1()中的 catch 块组，但未能被捕获，然后继续沿调用链向上，在 h()中的 catch(Matherr)块，捕获了该抛掷。

函数 h3()中的抛掷 throw Size，向上传递给 h1()中的 catch 块组，被 catch(Size)块捕获。

21.7 使用异常的方法

可以把多个异常组成族系。构成异常族系的一些示例有数学错误异常族系和文件处理错误异常族系。在C++代码中把异常组在一起有两种方式：异常枚举族系和异常派生层次结构。

例如，下面的代码是一个异常枚举族系的例子：

```
enum FileErrors{nonExist, wrongFormat, diskSeekError, ...};

int f()
{
  try
  {
   //...
   throw wrongFormat;
  }

  catch(FileErrors fe)
  {
   switch(fe)
   {
     case nonExist:
      //...
     case wrongFormat:
      //...
     case diskSeekError:
      //...
   }
  }
  //...
}
```

在try块中有一个throw，它抛掷一个FileErrors枚举中的常量。这个抛掷可被catch(FileErrors)块捕获到，接着后者执行一个switch，对照情况列表，匹配捕获到的枚举常量值。

上面的异常族系也可按异常派生层次结构来实现，如下例所示：

```
class FileErrors{};

class NonExist :public FileErrors{};
class WrongFormat :public FileErrors{};
class DiskSeekError :public FileErrors{};

int f()
{
  try
  {
   //...
   throw WrongFormat;
  }
```

```
    catch(NonExist)
    {
//...
    }

    catch(DiskSeekError)
    {
//...
    }

    catch(FileErrors)
    {
//...
    }
    //...
}
```

上面的各异常处理程序块定义了针对类 NonExist 和 DiskSeekError 的派生异常类对象，针对 FileErrors 的异常处理，既捕获 FileErrors 类对象，也捕获 WrongFormat 对象。

异常捕获的规则除了前面所说的，必须严格匹配数据类型外，对于类的派生，下列情况可以捕获异常：

(1) 异常处理的数据类型是公有基类，抛掷异常的数据类型是派生类；

(2) 异常处理的数据类型是指向公有基类的指针，抛掷异常的数据类型是指向派生类的指针。

→ 对于派生层次结构的异常处理，catch 块组中的顺序是重要的。因为“catch(基类)”总能够捕获“throw 派生类对象”，所以“catch(基类)”块总是放在“catch(派生类)”块的后面，以避免“catch(派生类)”永远不能捕获异常。

小结

为了检测异常，程序中使用 try、catch 和 throw 语句。异常处理使程序中错误的检测简单化，并提高程序处理错误的能力。

所谓异常是指程序中有运行错误。程序应能检测错误；

try 语句使 C++能够进行异常检测；

catch 紧跟在 try 语句后面，以捕获异常；

通过程序中的 throw 语句报告异常；

异常通过 throw 一个类型和 catch 的参数相匹配而捕获；

捕获异常后，程序将执行 catch 中的语句；

如果程序抛出一个不能捕获的异常，C++将执行默认异常处理函数。

练习

21.1 给 21.5 节中 ch21_3.cpp 添加一个“Pastm”异常类型的处理程序，如果[]运算符重载

在 String 对象中检测到一个字符——按字典顺序在'm'之后的小写字母，该异常处理程序就在屏幕上显示一个错误。

21.2 设有下列类声明：

```
class A{
  public:
    A()
    {
      n = new int;
      init();
    }
  private:
    int n;
};
```

写出 init()引发异常的处理程序。

参考文献

Bjarne Stroustrup. The C++ Programming Language. 3rd edition. Addison Wesley Longman, 1997.

Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, 1988.

章烨,郑茜译. C++技术详解. 北京:北京希望电子出版社,1991.

孟文辉等译. C++技术和应用. 北京:北京希望电子出版社, 1991.

林亨利等编译. TURBO C++应用教程(上)(中)(下). 北京:清华大学出版社,1992.

宛延闿. C++语言和面向对象程序设计. 第2版. 北京:清华大学出版社,1998.

Stephen R. Davis 著,卢凌云等译. C++编程指南. 北京:电子工业出版社,1996.

Stephen R. Davis 著,卢凌云等译. C++编程指南(续). 北京:电子工业出版社,1997.

Bruce Eckel 著,侯雪萍等译. C++深入浅出. 北京:学苑出版社,1994.

Borland C++ 3.0 技术丛书. 北京:北京希望电子出版社,1993.

William H. Murray III, Chris H. Pappas. Borland C/C++从入门到精通. 毛选等译. 北京:电子工业出版社,1998

周宁等译. Borland C++ 4 编程技巧与实例. 北京:清华大学出版社,1995.

Scott Meyers 著,陈迅雷等译. C++编程技巧. 上海:上海科学普及出版社,1994.

Stephen Blaha 著,孟庆昌等译. 最新C++应用编程技巧. 北京:国防工业出版社,1997.

江燕. C++成分和算法. 北京:北京希望电子出版社.

Kris Jamsa 著,凌涛等译. 新编C++自学教程. 北京:电子工业出版社,1996.

Namir C. Shammas 著,宋炎等译. 面向对象的编程指南. 北京:电子工业出版社,1997.

Jesse Liberty 著,路明等译. C++自学通. 北京:机械工业出版社,1997.

Herbert Schildt,马力文等译. C++从入门到精通. 北京:学苑出版社,1994.

William Ford, William Topp. 数据结构[C++语言描述]. 北京:清华大学出版社,1997.

Jesse Liberty 等著,张文旭等译. C++程序设计轻松入门. 北京:机械工业出版社,1996.

Microsoft 公司著,袁勤勇译. 程序设计范例与教学参考. 北京:学苑出版社,1994.

H. M. Deitel, P. J. Deitel 著,薛万鹏等译. C/C++程序设计大全,北京:机械工业出版社,1997.

曹康等译. Microsoft Visual C++自学教程. 北京:清华大学出版社,1997.

吴洁明等译. Visual C++面向对象程序设计. 北京:清华大学出版社,1994.

沈纪新. Visual C++使用速成. 北京:清华大学出版社,1997.

Michael Hyman Bob Arnson 著,马岚等译. 学用 Visual C++5. 北京:电子工业出版社,1998.

掌握 Visual C++的奥秘(影印). 北京:清华大学出版社,1994.

谭浩强著. C语言程序设计教程. 北京:高等教育出版社,1992.

Kris Jamsa 著,魏津等译. C/C++使用技巧1001例. 北京:电子工业出版社,1996.

Herbert Schildt 著,杨长虹等译. 最新C++语言精华(第2版). 北京:电子工业出版社,1997.

Steve Oualline 著,辛运帏等译. 使用C++编程大全. 北京:电子工业出版社,1997.

Peter van der Linden 著,张自力译. C程序设计奥秘. 昆明:云南科技出版社,1998.

Herbert Schildt 著,石桥林等译. C++自学教程. 北京:学苑出版社,1994.

张国峰等. C++程序设计实用教程. 北京:清华大学出版社,1996.

Cameron Hughes 等著,尤晓东等译. C++ iostream 面向对象 I/O 程序设计. 北京:电子工业出版社,1997.

Kaare Christian. Microsoft C++程序设计指南. 北京:清华大学出版社,1994.

Greg Perry 著. C++程序设计教程(配盘). 北京:清华大学出版社,1994.

王燕编. 面向对象的理论与C++实践. 北京:清华大学出版社,1997.

Tom Swan 著,宋建云译. C++编程秘诀. 北京:电子工业出版社,1994.