第 8 章　函数探幽

本章内容包括：

- 内联函数。
- 引用变量。
- 如何按引用传递函数参数。
- 默认参数。
- 函数重载。
- 函数模板。
- 函数模板具体化。

通过第 7 章，您了解到很多有关 C++函数的知识，但需要学习的知识还很多。C++还提供许多新的函数特性，使之有别于 C 语言。新特性包括内联函数、按引用传递变量、默认的参数值、函数重载（多态）以及模板函数。本章介绍的 C++在 C 语言基础上新增的特性，比前面各章都多，这是您进入加加（++）领域的重要一步。

8.1　C++内联函数

内联函数是 C++为提高程序运行速度所做的一项改进。常规函数和内联函数之间的主要区别不在于编写方式，而在于 C++编译器如何将它们组合到程序中。要了解内联函数与常规函数之间的区别，必须深入到程序内部。

编译过程的最终产品是可执行程序——由一组机器语言指令组成。运行程序时，操作系统将这些指令载入到计算机内存中，因此每条指令都有特定的内存地址。计算机随后将逐步执行这些指令。有时（如有循环或分支语句时），将跳过一些指令，向前或向后跳到特定地址。常规函数调用也使程序跳到另一个地址（函数的地址），并在函数结束时返回。下面更详细地介绍这一过程的典型实现。执行到函数调用指令时，程序将在函数调用后立即存储该指令的内存地址，并将函数参数复制到堆栈（为此保留的内存块），跳到标记函数起点的内存单元，执行函数代码（也许还需将返回值放入到寄存器中），然后跳回到地址被保存的指令处（这与阅读文章时停下来看脚注，并在阅读完脚注后返回到以前阅读的地方类似）。来回跳跃并记录跳跃位置意味着以前使用函数时，需要一定的开销。

C++内联函数提供了另一种选择。内联函数的编译代码与其他程序代码“内联”起来了。也就是说，编译器将使用相应的函数代码替换函数调用。对于内联代码，程序无需跳到另一个位置处执行代码，再跳回来。因此，内联函数的运行速度比常规函数稍快，但代价是需要占用更多内存。如果程序在 10 个不同的地方调用同一个内联函数，则该程序将包含该函数代码的 10 个副本（参见图 8.1）。

应有选择地使用内联函数。如果执行函数代码的时间比处理函数调用机制的时间长，则节省的时间将只占整个过程的很小一部分。如果代码执行时间很短，则内联调用就可以节省非内联调用使用的大部分时间。另一方面，由于这个过程相当快，因此尽管节省了该过程的大部分时间，但节省的时间绝对值并不大，

除非该函数经常被调用。

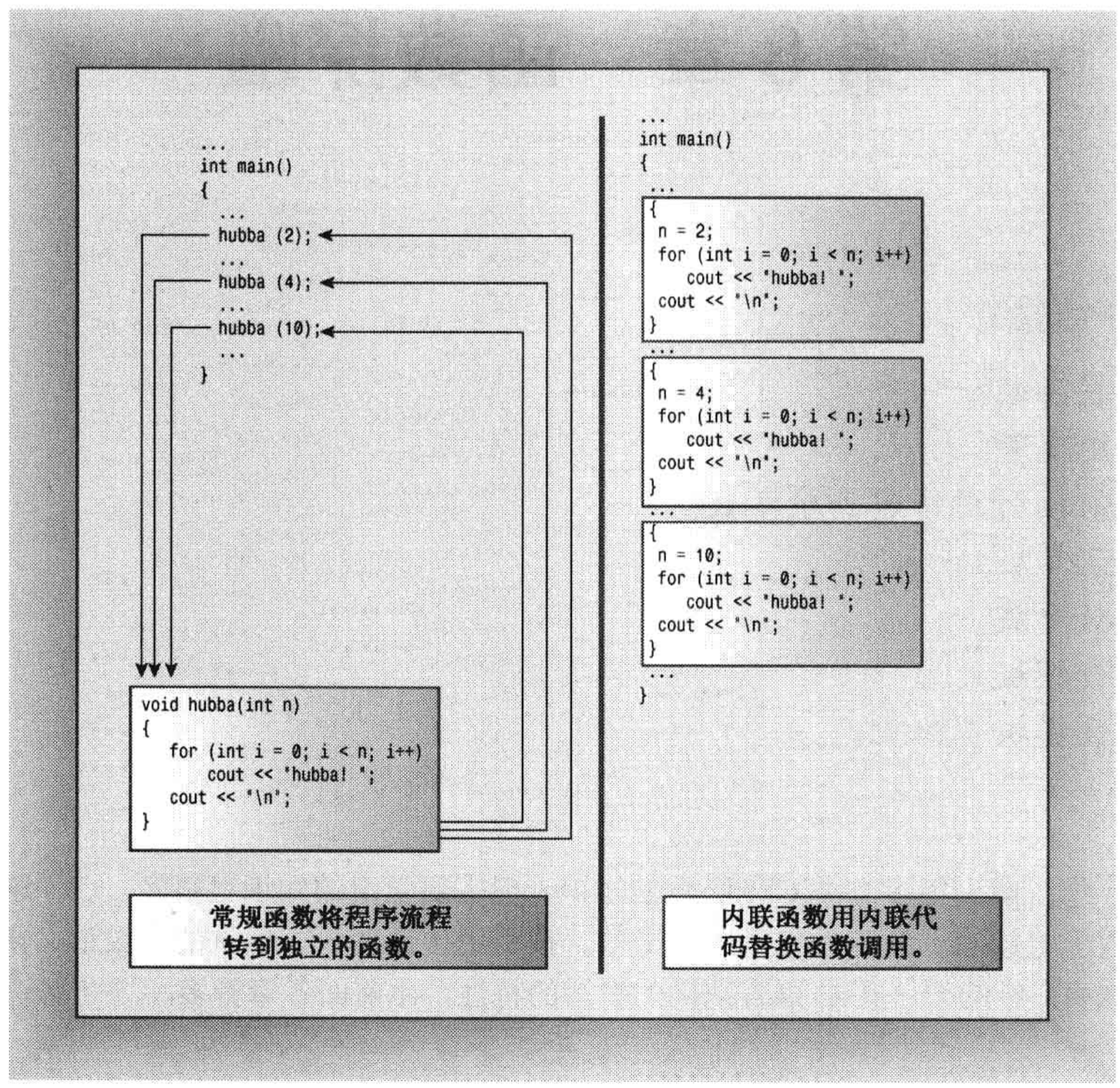


图 8.1　内联函数与常规函数

要使用这项特性，必须采取下述措施之一：

- 在函数声明前加上关键字 inline；
- 在函数定义前加上关键字 inline。

通常的做法是省略原型，将整个定义（即函数头和所有函数代码）放在本应提供原型的地方。

程序员请求将函数作为内联函数时，编译器并不一定会满足这种要求。它可能认为该函数过大或注意到函数调用了自己（内联函数不能递归），因此不将其作为内联函数；而有些编译器没有启用或实现这种特性。

程序清单 8.1 通过内联函数 square( )（计算参数的平方）演示了内联技术。注意到整个函数定义都放在一行中，但并不一定非得这样做。然而，如果函数定义占用多行（假定没有使用冗长的标识符），则将其作为内联函数就不太合适。

程序清单 8.1　inline.cpp

```
// inline.cpp -- using an inline function
#include <iostream>

// an inline function definition
inline double square(double x) { return x * x; }

int main()
{
    using namespace std;
    double a, b;
```

```
    double c = 13.0;

    a = square(5.0);
    b = square(4.5 + 7.5);   // can pass expressions
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << "\n";
    cout << "Now c = " << c << "\n";
    return 0;
}
```

下面是该程序的输出：

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

输出表明，内联函数和常规函数一样，也是按值来传递参数的。如果参数为表达式，如 4.5 + 7.5，则函数将传递表达式的值（这里为 12）。这使得 C++的内联功能远远胜过 C 语言的宏定义，请参见旁注“内联与宏”。

尽管程序没有提供独立的原型，但 C++原型特性仍在起作用。这是因为在函数首次使用前出现的整个函数定义充当了原型。这意味着可以给 square( )传递 int 或 long 值，将值传递给函数前，程序自动将这个值强制转换为 double 类型。

内联与宏

inline 工具是 C++新增的特性。C 语言使用预处理器语句#define 来提供宏——内联代码的原始实现。例如，下面是一个计算平方的宏：

```
#define SQUARE(X) X*X
```

这并不是通过传递参数实现的，而是通过文本替换来实现的——X 是“参数”的符号标记。

```
a = SQUARE(5.0); is replaced by a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); is replaced by b = 4.5 + 7.5 * 4.5 + 7.5;
d = SQUARE(c++); is replaced by d = c++*c++;
```

上述示例只有第一个能正常工作。可以通过使用括号来进行改进：

```
#define SQUARE(X) ((X)*(X))
```

但仍然存在这样的问题，即宏不能按值传递。即使使用新的定义，SQUARE（C++）仍将 c 递增两次，但是程序清单 8.1 中的内联函数 square( )计算 c 的结果，传递它，以计算其平方值，然后将 c 递增一次。

这里的目的不是演示如何编写 C 宏，而是要指出，如果使用 C 语言的宏执行了类似函数的功能，应考虑将它们转换为 C++内联函数。

8.2　引用变量

C++新增了一种复合类型——引用变量。引用是已定义的变量的别名（另一个名称）。例如，如果将 twain 作为 clement 变量的引用，则可以交替使用 twain 和 clement 来表示该变量。那么，这种别名有何作用呢？是否能帮助那些不知道如何选择变量名的人呢？有可能，但引用变量的主要用途是用作函数的形参。通过将引用变量用作参数，函数将使用原始数据，而不是其副本。这样除指针之外，引用也为函数处理大型结构提供了一种非常方便的途径，同时对于设计类来说，引用也是必不可少的。然而，介绍如何将引用用于函数之前，先介绍一下定义和使用引用的基本知识。请记住，下述讨论旨在说明引用是如何工作的，而不是其典型用法。

8.2.1 创建引用变量

前面讲过，C 和 C++使用&符号来指示变量的地址。C++给&符号赋予了另一个含义，将其用来声明引用。例如，要将 rodents 作为 rats 变量的别名，可以这样做：

```
int rats;
int & rodents = rats;    // makes rodents an alias for rats
```

其中，&不是地址运算符，而是类型标识符的一部分。就像声明中的 char*指的是指向 char 的指针一样，int &指的是指向 int 的引用。上述引用声明允许将 rats 和 rodents 互换——它们指向相同的值和内存单元，程序清单 8.2 表明了这一点。

程序清单 8.2　firstref.cpp

```
// firstref.cpp -- defining and using a reference
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats;   // rodents is a reference
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

// some implementations require type casting the following
// addresses to type unsigned
    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

请注意，下述语句中的&运算符不是地址运算符，而是将 rodents 的类型声明为 int &，即指向 int 变量的引用：

```
int & rodents = rats;
```

但下述语句中的&运算符是地址运算符，其中&rodents 表示 rodents 引用的变量的地址：

```
cout <<", rodents address = " << &rodents << endl;
```

下面是程序清单 8.2 中程序的输出：

```
rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 0x0065fd48, rodents address = 0x0065fd48
```

从中可知，rats 和 rodents 的值和地址都相同（具体的地址和显示格式随系统而异）。将 rodents 加 1 将影响这两个变量。更准确地说，rodents++操作将一个有两个名称的变量加 1。（同样，虽然该示例演示了引用是如何工作的，但并没有说明引用的典型用途，即作为函数参数，具体地说是结构和对象参数，稍后将介绍这些用法）。

对于 C 语言用户而言，首次接触到引用时可能也会有些困惑，因为这些用户很自然地会想到指针，但它们之间还是有区别的。例如，可以创建指向 rats 的引用和指针：

```
int rats = 101;
int & rodents = rats;   // rodents a reference
int * prats = &rats;    // prats a pointer
```

这样，表达式 rodents 和*prats 都可以同 rats 互换，而表达式&rodents 和 prats 都可以同&rats 互换。从

这一点来说，引用看上去很像伪装表示的指针（其中，*解除引用运算符被隐式理解）。实际上，引用还是不同于指针的。除了表示法不同外，还有其他的差别。例如，差别之一是，必须在声明引用时将其初始化，而不能像指针那样，先声明，再赋值：

```
int rat;
int & rodent;
rodent = rat;   // No, you can't do this.
```

注意：必须在声明引用变量时进行初始化。

引用更接近 const 指针，必须在创建时进行初始化，一旦与某个变量关联起来，就将一直效忠于它。也就是说：

```
int & rodents = rats;
```

实际上是下述代码的伪装表示：

```
int * const pr = &rats;
```

其中，引用 rodents 扮演的角色与表达式*pr 相同。

程序清单 8.3 演示了试图将 rats 变量的引用改为 bunnies 变量的引用时，将发生的情况。

程序清单 8.3　sceref.cpp

```
// secref.cpp -- defining and using a reference
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;

    int bunnies = 50;
    rodents = bunnies;       // can we change the reference?
    cout << "bunnies = " << bunnies;
    cout << ", rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    cout << "bunnies address = " << &bunnies;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

下面是程序清单 8.3 中程序的输出：

```
rats = 101, rodents = 101
rats address = 0x0065fd44, rodents address = 0x0065fd44
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0x0065fd48, rodents address = 0x0065fd4
```

最初，rodents 引用的是 rats，但随后程序试图将 rodents 作为 bunnies 的引用：

```
rodents = bunnies;
```

咋一看，这种意图暂时是成功的，因为 rodents 的值从 101 变为了 50。但仔细研究将发现，rats 也变成

了 50，同时 rats 和 rodents 的地址相同，而该地址与 bunnies 的地址不同。由于 rodents 是 rats 的别名，因此上述赋值语句与下面的语句等效：

```
rats = bunnies;
```

也就是说，这意味着“将 bunnies 变量的值赋给 rat 变量”。简而言之，可以通过初始化声明来设置引用，但不能通过赋值来设置。

假设程序员试图这样做：

```
int rats = 101;
int * pt = &rats;
int & rodents = *pt;
int bunnies = 50;
pt = &bunnies;
```

将 rodents 初始化为*pt 使得 rodents 指向 rats。接下来将 pt 改为指向 bunnies，并不能改变这样的事实，即 rodents 引用的是 rats。

8.2.2 将引用用作函数参数

引用经常被用作函数参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为按引用传递。按引用传递允许被调用的函数能够访问调用函数中的变量。C++新增的这项特性是对 C 语言的超越，C 语言只能按值传递。按值传递导致被调用函数使用调用程序的值的拷贝（参见图 8.2）。当然，C 语言也允许避开按值传递的限制，采用按指针传递的方式。

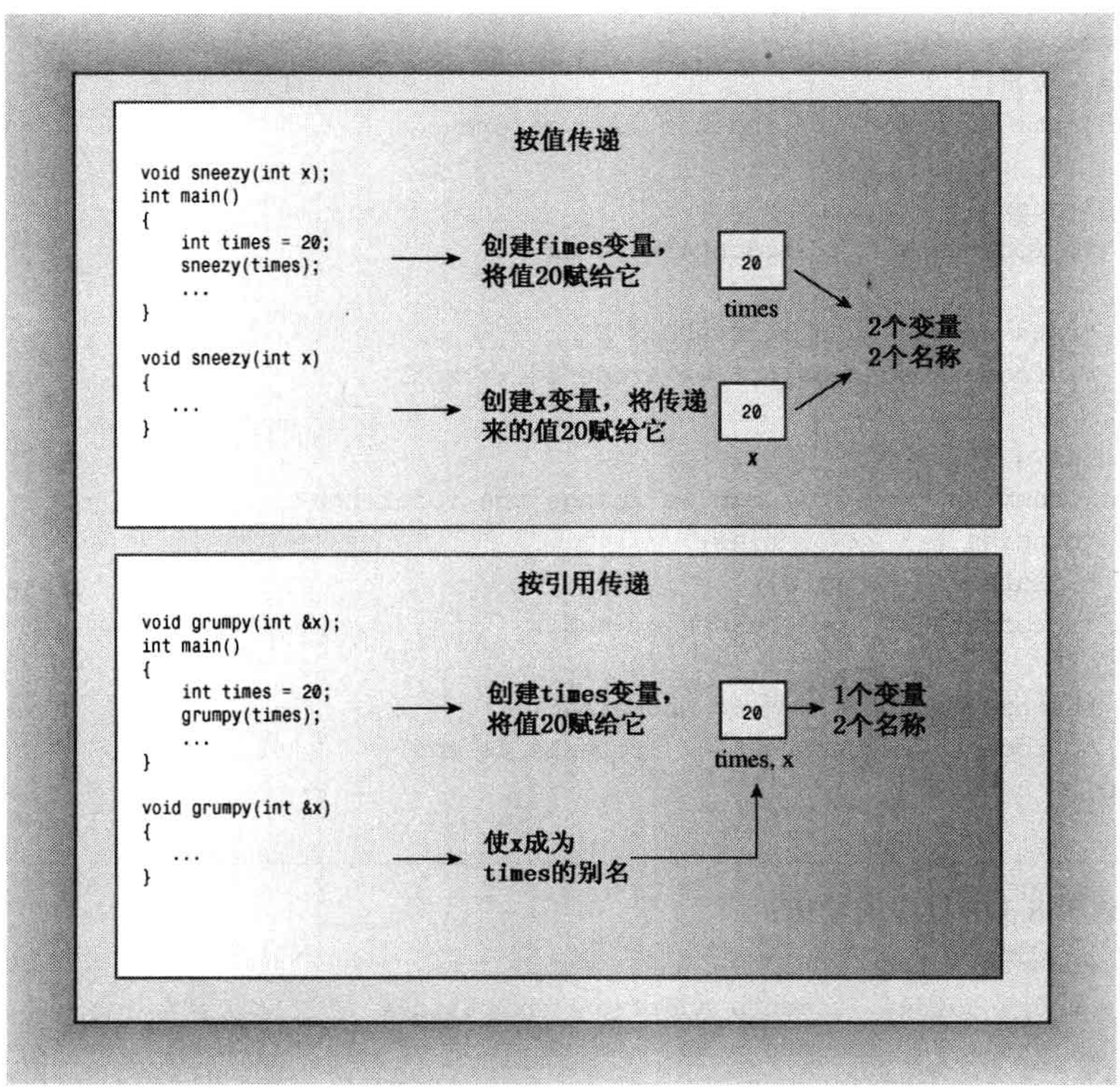


图 8.2 按值传递和按引用传递

现在我们通过一个常见的的计算机问题——交换两个变量的值，对使用引用和使用指针做一下比较。交换函数必须能够修改调用程序中的变量的值。这意味着按值传递变量将不管用，因为函数将交换原始变

量副本的内容，而不是变量本身的内容。但传递引用时，函数将可以使用原始数据。另一种方法是，传递指针来访问原始数据。程序清单 8.4 演示了这三种方法，其中包括一种不可行的方法，以便您能对这些方法进行比较。

程序清单 8.4　swaps.cpp

```
// swaps.cpp -- swapping with references and with pointers
#include <iostream>
void swapr(int & a, int & b);   // a, b are aliases for ints
void swapp(int * p, int * q);   // p, q are addresses of ints
void swapv(int a, int b);       // a, b are new variables
int main()
{
    using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using references to swap contents:\n";
    swapr(wallet1, wallet2);   // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Using pointers to swap contents again:\n";
    swapp(&wallet1, &wallet2); // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Trying to use passing by value:\n";
    swapv(wallet1, wallet2);   // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
}

void swapr(int & a, int & b)    // use references
{
    int temp;

    temp = a;       // use a, b for values of variables
    a = b;
    b = temp;
}

void swapp(int * p, int * q)    // use pointers
{
    int temp;

    temp = *p;      // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}

void swapv(int a, int b)        // try using values
```

```
{
    int temp;

    temp = a;       // use a, b for values of variables
    a = b;
    b = temp;
}
```

下面是程序清单 8.4 中程序的输出：

```
wallet1 = $300 wallet2 = $350          << original values
Using references to swap contents:
wallet1 = $350 wallet2 = $300          << values swapped
Using pointers to swap contents again:
wallet1 = $300 wallet2 = $350          << values swapped again
Trying to use passing by value:
wallet1 = $300 wallet2 = $350          << swap failed
```

正如您预想的，引用和指针方法都成功地交换了两个钱夹（wallet）中的内容，而按值传递的方法没能完成这项任务。

程序说明

首先来看程序清单 8.4 中每个函数是如何被调用的：

```
swapr(wallet1, wallet2);        // pass variables
swapp(&wallet1, &wallet2);      // pass addresses of variables
swapv(wallet1, wallet2);        // pass values of variables
```

按引用传递（swapr(wallet1, wallet2)）和按值传递（swapv(wallet1, waller2)）看起来相同。只能通过原型或函数定义才能知道 swapr( )是按引用传递的。然而，地址运算符（&）使得按地址传递（swapp(&wallet1, &wallet2)）一目了然（类型声明 int * p 表明，p 是一个 int 指针，因此与 p 对应的参数应为地址，如&wallet1）。

接下来，比较函数 swapr( )（按引用传递）和 swapv( )（按值传递）的代码，唯一的外在区别是声明函数参数的方式不同：

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

当然还有内在区别：在 swapr( )中，变量 a 和 b 是 wallet1 和 wallet2 的别名，所以交换 a 和 b 的值相当于交换 wallet1 和 wallet2 的值；但在 swapv( )中，变量 a 和 b 是复制了 wallet1 和 waller2 的值的新变量，因此交换 a 和 b 的值并不会影响 wallet1 和 wallet2 的值。

最后，比较函数 swapr( )（传递引用）和 swapp( )（传递指针）。第一个区别是声明函数参数的方式不同：

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

另一个区别是指针版本需要在函数使用 p 和 q 的整个过程中使用解除引用运算符*。

前面说过，应在定义引用变量时对其进行初始化。函数调用使用实参初始化形参，因此函数的引用参数被初始化为函数调用传递的实参。也就是说，下面的函数调用将形参 a 和 b 分别初始化为 wallet1 和 wallet2：

```
swapr(wallet1, wallet2);
```

8.2.3 引用的属性和特别之处

使用引用参数时，需要了解其一些特点。首先，请看程序清单 8.5。它使用两个函数来计算参数的立方，其中一个函数接受 double 类型的参数，另一个接受 double 引用。为了说明这一点，我们有意将计算立方的代码编写得比较奇怪。

程序清单 8.5　cubes.cpp

```
// cubes.cpp -- regular and reference arguments
#include <iostream>
double cube(double a);
double refcube(double &ra);
int main ()
{
    using namespace std;
    double x = 3.0;

    cout << cube(x);
    cout << " = cube of " << x << endl;
    cout << refcube(x);
    cout << " = cube of " << x << endl;
    return 0;
}

double cube(double a)
{
    a *= a * a;
    return a;
}

double refcube(double &ra)
{
    ra *= ra * ra;
    return ra;
}
```

下面是该程序的输出：

```
27 = cube of 3
27 = cube of 27
```

refcube( )函数修改了 main( )中的 x 值，而 cube( )没有，这提醒我们为何通常按值传递。变量 a 位于 cube( )中，它被初始化为 x 的值，但修改 a 并不会影响 x。但由于 refcube( )使用了引用参数，因此修改 ra 实际上就是修改 x。如果程序员的意图是让函数使用传递给它的信息，而不对这些信息进行修改，同时又想使用引用，则应使用常量引用。例如，在这个例子中，应在函数原型和函数头中使用 const：

```
double refcube(const double &ra);
```

如果这样做，当编译器发现代码修改了 ra 的值时，将生成错误消息。

顺便说一句，如果要编写类似于上述示例的函数（即使用基本数值类型），应采用按值传递的方式，而不要采用按引用传递的方式。当数据比较大（如结构和类）时，引用参数将很有用，您稍后便会明白这一点。

按值传递的函数，如程序清单 8.5 中的函数 cube( )，可使用多种类型的实参。例如，下面的调用都是合法的：

```
double z = cube(x + 2.0);      // evaluate x + 2.0, pass value
z = cube(8.0);                 // pass the value 8.0
int k = 10;
z = cube(k);                   // convert value of k to double, pass value
double yo[3] = { 2.2, 3.3, 4.4};
z = cube (yo[2]);              // pass the value 4.4
```

如果将与上面类似的参数传递给接受引用参数的函数，将会发现，传递引用的限制更严格。毕竟，如

果 ra 是一个变量的别名，则实参应是该变量。下面的代码不合理，因为表达式 x + 3.0 并不是变量：

```
double z = refcube(x + 3.0);  // should not compile
```

例如，不能将值赋给该表达式：

```
x + 3.0 = 5.0;  // nonsensical
```

如果试图使用像 refcube(x + 3.0)这样的函数调用，将发生什么情况呢？在现代的 C++中，这是错误的，大多数编译器都将指出这一点；而有些较老的编译器将发出这样的警告：

```
Warning: Temporary used for parameter 'ra' in call to refcube(double &)
```

之所以做出这种比较温和的反应是由于早期的 C++确实允许将表达式传递给引用变量。有些情况下，仍然是这样做的。这样做的结果如下：由于 x + 3.0 不是 double 类型的变量，因此程序将创建一个临时的无名变量，并将其初始化为表达式 x + 3.0 的值。然后，ra 将成为该临时变量的引用。下面详细讨论这种临时变量，看看什么时候创建它们，什么时候不创建。

临时变量、引用参数和 const

如果实参与引用参数不匹配，C++将生成临时变量。当前，仅当参数为 const 引用时，C++才允许这样做，但以前不是这样。下面来看看何种情况下，C++将生成临时变量，以及为何对 const 引用的限制是合理的。

首先，什么时候将创建临时变量呢？如果引用参数是 const，则编译器将在下面两种情况下生成临时变量：

- 实参的类型正确，但不是左值；
- 实参的类型不正确，但可以转换为正确的类型。

左值是什么呢？左值参数是可被引用的数据对象，例如，变量、数组元素、结构成员、引用和解除引用的指针都是左值。非左值包括字面常量（用引号括起的字符串除外，它们由其地址表示）和包含多项的表达式。在 C 语言中，左值最初指的是可出现在赋值语句左边的实体，但这是引入关键字 const 之前的情况。现在，常规变量和 const 变量都可视为左值，因为可通过地址访问它们。但常规变量属于可修改的左值，而 const 变量属于不可修改的左值。

回到前面的示例。假设重新定义了 refcube( )，使其接受一个常量引用参数：

```
double refcube(const double &ra)
{
     return ra * ra * ra;
}
```

现在考虑下面的代码：

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0};
double c1 = refcube(side);          // ra is side
double c2 = refcube(lens[2]);       // ra is lens[2]
double c3 = refcube(rd);            // ra is rd is side
double c4 = refcube(*pd);           // ra is *pd is side
double c5 = refcube(edge);          // ra is temporary variable
double c6 = refcube(7.0);           // ra is temporary variable
double c7 = refcube(side + 10.0);   // ra is temporary variable
```

参数 side、lens[2]、rd 和*pd 都是有名称的、double 类型的数据对象，因此可以为其创建引用，而不需要临时变量（还记得吗，数组元素的行为与同类型的变量类似）。然而，edge 虽然是变量，类型却不正确，double 引用不能指向 long。另一方面，参数 7.0 和 side + 10.0 的类型都正确，但没有名称，在这些情况下，编译器都将生成一个临时匿名变量，并让 ra 指向它。这些临时变量只在函数调用期间存在，此后编译器便

可以随意将其删除。

那么为什么对于常量引用，这种行为是可行的，其他情况下却不行的呢？对于程序清单 8.4 中的函数 swapr( )：

```
void swapr(int & a, int & b)  // use references
{
    int temp;

    temp = a;       // use a, b for values of variables
    a = b;
    b = temp;
}
```

如果在早期 C++较宽松的规则下，执行下面的操作将发生什么情况呢？

```
long a = 3, b = 5;
swapr(a, b);
```

这里的类型不匹配，因此编译器将创建两个临时 int 变量，将它们初始化为 3 和 5，然后交换临时变量的内容，而 a 和 b 保持不变。

简而言之，如果接受引用参数的函数的意图是修改作为参数传递的变量，则创建临时变量将阻止这种意图的实现。解决方法是，禁止创建临时变量，现在的 C++标准正是这样做的（然而，在默认情况下，有些编译器仍将发出警告，而不是错误消息，因此如果看到了有关临时变量的警告，请不要忽略）。

现在来看 refcube( )函数。该函数的目的只是使用传递的值，而不是修改它们，因此临时变量不会造成任何不利的影响，反而会使函数在可处理的参数种类方面更通用。因此，如果声明将引用指定为 const，C++将在必要时生成临时变量。实际上，对于形参为 const 引用的 C++函数，如果实参不匹配，则其行为类似于按值传递，为确保原始数据不被修改，将使用临时变量来存储值。

注意：如果函数调用的参数不是左值或与相应的 const 引用参数的类型不匹配，则 C++将创建类型正确的匿名变量，将函数调用的参数的值传递给该匿名变量，并让参数来引用该变量。

应尽可能使用 const

将引用参数声明为常量数据的引用的理由有三个：

- 使用 const 可以避免无意中修改数据的编程错误；
- 使用 const 使函数能够处理 const 和非 const 实参，否则将只能接受非 const 数据；
- 使用 const 引用使函数能够正确生成并使用临时变量。

因此，应尽可能将引用形参声明为 const。

C++11 新增了另一种引用——右值引用（rvalue reference）。这种引用可指向右值，是使用&&声明的：

```
double && rref = std::sqrt(36.00);  // not allowed for double &
double j = 15.0;
double && jref = 2.0* j + 18.5;     // not allowed for double &
std::cout << rref << '\n';          // display 6.0
std::cout << jref << '\n';          // display 48.5;
```

新增右值引用的主要目的是，让库设计人员能够提供有些操作的更有效实现。第 18 章将讨论如何使用右值引用来实现移动语义（move semantics）。以前的引用（使用&声明的引用）现在称为左值引用。

8.2.4　将引用用于结构

引用非常适合用于结构和类（C++的用户定义类型）。确实，引入引用主要是为了用于这些类型的，而不是基本的内置类型。

使用结构引用参数的方式与使用基本变量引用相同，只需在声明结构参数时使用引用运算符&即可。例如，假设有如下结构定义：

```
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};
```

则可以这样编写函数原型，在函数中将指向该结构的引用作为参数：

```
void set_pc(free_throws & ft);   // use a reference to a structure
```

如果不希望函数修改传入的结构，可使用 const：

```
void display(const free_throws & ft);  // don't allow changes to structure
```

程序清单 8.6 中的程序正是这样做的。它还通过让函数返回指向结构的引用添加了一个有趣的特点，这与返回结构有所不同。对此，有一些需要注意的地方，稍后将进行介绍。

程序清单 8.6　strtref.cpp

```
//strc_ref.cpp -- using structure references
#include <iostream>
#include <string>
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};

void display(const free_throws & ft);
void set_pc(free_throws & ft);
free_throws & accumulate(free_throws & target, const free_throws & source);

int main()
{
// partial initializations - remaining members set to 0
    free_throws one = {"Ifelsa Branch", 13, 14};
    free_throws two = {"Andor Knott", 10, 16};
    free_throws three = {"Minnie Max", 7, 9};
    free_throws four = {"Whily Looper", 5, 9};
    free_throws five = {"Long Long", 6, 14};
    free_throws team = {"Throwgoods", 0, 0};
// no initialization
    free_throws dup;

    set_pc(one);
    display(one);
    accumulate(team, one);
    display(team);
// use return value as argument
    display(accumulate(team, two));
    accumulate(accumulate(team, three), four);
    display(team);
// use return value in assignment
    dup = accumulate(team,five);
```

```
    std::cout << "Displaying team:\n";
    display(team);
    std::cout << "Displaying dup after assignment:\n";
    display(dup);
    set_pc(four);
// ill-advised assignment
    accumulate(dup,five) = four;
    std::cout << "Displaying dup after ill-advised assignment:\n";
    display(dup);
    return 0;
}

void display(const free_throws & ft)
{
    using std::cout;
    cout << "Name: " << ft.name << '\n';
    cout << "  Made: " << ft.made << '\t';
    cout << "Attempts: " << ft.attempts << '\t';
    cout << "Percent: " << ft.percent << '\n';
}
void set_pc(free_throws & ft)
{
    if (ft.attempts != 0)
        ft.percent = 100.0f *float(ft.made)/float(ft.attempts);
    else
        ft.percent = 0;
}

free_throws & accumulate(free_throws & target, const free_throws & source)
{
    target.attempts += source.attempts;
    target.made += source.made;
    set_pc(target);
    return target;
}
```

下面是该程序的输出：

```
Name: Ifelsa Branch
  Made: 13      Attempts: 14    Percent: 92.8571
Name: Throwgoods
  Made: 13      Attempts: 14    Percent: 92.8571
Name: Throwgoods
  Made: 23      Attempts: 30    Percent: 76.6667
Name: Throwgoods
  Made: 35      Attempts: 48    Percent: 72.9167
Displaying team:
Name: Throwgoods
  Made: 41      Attempts: 62    Percent: 66.129
Displaying dup after assignment:
Name: Throwgoods
  Made: 41      Attempts: 62    Percent: 66.129
Displaying dup after ill-advised assignment:
Name: Whily Looper
  Made: 5       Attempts: 9     Percent: 55.5556
```

1. 程序说明

该程序首先初始化了多个结构对象。本书前面说过，如果指定的初始值比成员少，余下的成员（这里只有 percent）将被设置为零。第一个函数调用如下：

```
set_pc(one);
```

由于函数 set_pc()的形参 ft 为引用，因此 ft 指向 one，函数 set_pc()的代码设置成员 one.percent。就这里而言，按值传递不可行，因此这将导致设置的是 one 的临时拷贝的成员 percent。根据前一章介绍的知识，另一种方法是使用指针参数并传递地址，但要复杂些：

```
set_pcp(&one);    // using pointers instead - &one instead of one
...
void set_pcp(free_throws * pt)
{
    if (pt->attempts != 0)
        pt->percent = 100.0f *float(pt->made)/float(pt->attempts);
    else
        pt->percent = 0;
}
```

下一个函数调用如下：

```
display(one);
```

由于 display()显示结构的内容，而不修改它，因此这个函数使用了一个 const 引用参数。就这个函数而言，也可按值传递结构，但与复制原始结构的拷贝相比，使用引用可节省时间和内存。

再下一个函数调用如下：

```
accumulate(team, one);
```

函数 accumulate()接收两个结构参数，并将第二个结构的成员 attempts 和 made 的数据添加到第一个结构的相应成员中。只修改了第一个结构，因此第一个参数为引用，而第二个参数为 const 引用：

```
free_throws & accumulate(free_throws & target, const free_throws & source);
```

返回值呢？当前讨论的函数调用没有使用它；就目前而言，原本可以将返回值声明为 void，但请看下述函数调用：

```
display(accumulate(team, two));
```

上述代码是什么意思呢？首先，将结构对象 team 作为第一个参数传递给了 accumulate()。这意味着在函数 accumulate()中，target 指向的是 team。函数 accumulate()修改 team，再返回指向它的引用。注意到返回语句如下：

```
return target;
```

光看这条语句并不能知道返回的是引用，但函数头和原型指出了这一点：

```
free_throws & accumulate(free_throws & target, const free_throws & source)
```

如果返回类型被声明为 free_throws 而不是 free_throws &，上述返回语句将返回 target（也就是 team）的拷贝。但返回类型为引用，这意味着返回的是最初传递给 accumulate()的 team 对象。

接下来，将 accumulate()的返回值作为参数传递给了 display()，这意味着将 team 传递给了 display()。display()的参数为引用，这意味着函数 display()中的 ft 指向的是 team，因此将显示 team 的内容。所以，下述代码：

```
display(accumulate(team, two));
```

与下面的代码等效：

```
accumulate(team, two);
display(team);
```

上述逻辑也适用于如下语句：

```
accumulate(accumulate(team, three), four);
```

因此，该语句与下面的语句等效：

```
accumulate(team, three);
accumulate(team, four);
```

接下来，程序使用了一条赋值语句：

```
dup = accumulate(team,five);
```

正如您预期的，这条语句将 team 中的值复制到 dup 中。

最后，程序以独特的方式使用了 accumulate()：

```
accumulate(dup,five) = four;
```

这条语句将值赋给函数调用，这是可行的，因为函数的返回值是一个引用。如果函数 accumulate()按值返回，这条语句将不能通过编译。由于返回的是指向 dup 的引用，因此上述代码与下面的代码等效：

```
accumulate(dup,five); // add five's data to dup
dup = four;           // overwrite the contents of dup with the contents of four
```

其中第二条语句消除了第一条语句所做的工作，因此在原始赋值语句使用 accumulate()的方式并不好。

2. 为何要返回引用

下面更深入地讨论返回引用与传统返回机制的不同之处。传统返回机制与按值传递函数参数类似：计算关键字 return 后面的表达式，并将结果返回给调用函数。从概念上说，这个值被复制到一个临时位置，而调用程序将使用这个值。请看下面的代码：

```
double m = sqrt(16.0);
cout << sqrt(25.0);
```

在第一条语句中，值 4.0 被复制到一个临时位置，然后被复制给 m。在第二条语句中，值 5.0 被复制到一个临时位置，然后被传递给 cout（这里理论上的描述，实际上，编译器可能合并某些步骤）。

现在来看下面的语句：

```
dup = accumulate(team,five);
```

如果 accumulate()返回一个结构，而不是指向结构的引用，将把整个结构复制到一个临时位置，再将这个拷贝复制给 dup。但在返回值为引用时，将直接把 team 复制到 dup，其效率更高。

注意：返回引用的函数实际上是被引用的变量的别名。

3. 返回引用时需要注意的问题

返回引用时最重要的一点是，应避免返回函数终止时不再存在的内存单元引用。您应避免编写下面这样的代码：

```
const free_throws & clone2(free_throws & ft)
{
    free_throws newguy;  // first step to big error
    newguy = ft;         // copy info
    return newguy;       // return reference to copy
}
```

该函数返回一个指向临时变量（newguy）的引用，函数运行完毕后它将不再存在。第 9 章将讨论各种变量的持续性。同样，也应避免返回指向临时变量的指针。

为避免这种问题，最简单的方法是，返回一个作为参数传递给函数的引用。作为参数的引用将指向调用函数使用的数据，因此返回的引用也将指向这些数据。程序清单 8.6 中的 accumulate()正是这样做的。

另一种方法是用 new 来分配新的存储空间。前面见过这样的函数，它使用 new 为字符串分配内存空间，并返回指向该内存空间的指针。下面是使用引用来完成类似工作的方法：

```
const free_throws & clone(free_throws & ft)
{
    free_throws * pt;
    *pt = ft;           // copy info
    return *pt;         // return reference to copy
}
```

第一条语句创建一个无名的 free_throws 结构，并让指针 pt 指向该结构，因此*pt 就是该结构。上述代码似乎会返回该结构，但函数声明表明，该函数实际上将返回这个结构的引用。这样，便可以这样使用该函数：

```
free_throws & jolly = clone(three);
```

这使得 jolly 成为新结构的引用。这种方法存在一个问题：在不再需要 new 分配的内存时，应使用 delete 来释放它们。调用 clone( )隐藏了对 new 的调用，这使得以后很容易忘记使用 delete 来释放内存。第 16 章讨论的 auto_ptr 模板以及 C++11 新增的 unique_ptr 可帮助程序员自动完成释放工作。

4. 为何将 const 用于引用返回类型

程序清单 8.6 包含如下语句：

```
accumulate(dup,five) = four;
```

其效果如下：首先将 five 的数据添加到 dup 中，再使用 four 的内容覆盖 dup 的内容。这条语句为何能够通过编译呢？在赋值语句中，左边必须是可修改的左值。也就是说，在赋值表达式中，左边的子表达式必须标识一个可修改的内存块。在这里，函数返回指向 dup 的引用，它确实标识的是一个这样的内存块，因此这条语句是合法的。

另一方面，常规（非引用）返回类型是右值——不能通过地址访问的值。这种表达式可出现在赋值语句的右边，但不能出现在左边。其他右值包括字面值（如 10.0）和表达式（如 x + y）。显然，获取字面值（如 10.0）的地址没有意义，但为何常规函数返回值是右值呢？这是因为这种返回值位于临时内存单元中，运行到下一条语句时，它们可能不再存在。

假设您要使用引用返回值，但又不允许执行像给 accumulate()赋值这样的操作，只需将返回类型声明为 const 引用：

```
const free_throws &
    accumulate(free_throws & target, const free_throws & source);
```

现在返回类型为 const，是不可修改的左值，因此下面的赋值语句不合法：

```
accumulate(dup,five) = four;  // not allowed for const reference return
```

该程序中的其他函数调用又如何呢？返回类型为 const 引用后，下面的语句仍合法：

```
display(accumulate(team, two));
```

这是因为 display()的形参也是 const free_throws &类型。但下面的语句不合法，因此 accumulate()的第一个形参不是 const：

```
accumulate(accumulate(team, three), four);
```

这影响大吗？就这里而言不大，因为您仍可以这样做：

```
accumulate(team, three);
accumulate(team, four);
```

另外，您仍可以在赋值语句右边使用 accumulate()。

通过省略 const，可以编写更简短代码，但其含义也更模糊。

通常，应避免在设计中添加模糊的特性，因为模糊特性增加了犯错的机会。将返回类型声明为 const 引用，可避免您犯糊涂。然而，有时候省略 const 确实有道理，第 11 章将讨论的重载运算符<<就是一个这样的例子。

8.2.5 将引用用于类对象

将类对象传递给函数时，C++通常的做法是使用引用。例如，可以通过使用引用，让函数将类 string、

ostream、istream、ofstream 和 ifstream 等类的对象作为参数。

下面来看一个例子，它使用了 string 类，并演示了一些不同的设计方案，其中的一些是糟糕的。这个例子的基本思想是，创建一个函数，它将指定的字符串加入到另一个字符串的前面和后面。程序清单 8.7 提供了三个这样的函数，然而其中的一个存在非常大的缺陷，可能导致程序崩溃甚至不同通过编译。

程序清单 8.7　strquote.cpp

```
// strquote.cpp -- different designs
#include <iostream>
#include <string>
using namespace std;
string version1(const string & s1, const string & s2);
const string & version2(string & s1, const string & s2);  // has side effect
const string & version3(string & s1, const string & s2);  // bad design

int main()
{
    string input;
    string copy;
    string result;

    cout << "Enter a string: ";
    getline(cin, input);
    copy = input;
    cout << "Your string as entered: " << input << endl;
    result = version1(input, "***");
    cout << "Your string enhanced: " << result << endl;
    cout << "Your original string: " << input << endl;

    result = version2(input, "###");
    cout << "Your string enhanced: " << result << endl;
    cout << "Your original string: " << input << endl;

    cout << "Resetting original string.\n";
    input = copy;
    result = version3(input, "@@@");
    cout << "Your string enhanced: " << result << endl;
    cout << "Your original string: " << input << endl;

    return 0;
}

string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;
    return temp;
}

const string & version2(string & s1, const string & s2)   // has side effect
{
    s1 = s2 + s1 + s2;
// safe to return reference passed to function
    return s1;
```

```
}

const string & version3(string & s1, const string & s2)   // bad design
{
    string temp;

    temp = s2 + s1 + s2;
// unsafe to return reference to local variable
    return temp;
}
```

下面是该程序的运行情况：

```
Enter a string: It's not my fault.
Your string as entered: It's not my fault.
Your string enhanced: ***It's not my fault.***
Your original string: It's not my fault.
Your string enhanced: ###It's not my fault.###
Your original string: ###It's not my fault.###
Resetting original string.
```

此时，该程序已经崩溃。

程序说明

在程序清单 8.7 的三个函数中，version1 最简单：

```
string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;
    return temp;
}
```

它接受两个 string 参数，并使用 string 类的相加功能来创建一个满足要求的新字符串。这两个函数参数都是 const 引用。如果使用 string 对象作为参数，最终结果将不变：

```
string version4(string s1, string  s2)  // would work the same
```

在这种情况下，s1 和 s2 将为 string 对象。使用引用的效率更高，因为函数不需要创建新的 string 对象，并将原来对象中的数据复制到新对象中。限定符 const 指出，该函数将使用原来的 string 对象，但不会修改它。

temp 是一个新的 string 对象，只在函数 version1( )中有效，该函数执行完毕后，它将不再存在。因此，返回指向 temp 的引用不可行，因此该函数的返回类型为 string，这意味着 temp 的内容将被复制到一个临时存储单元中，然后在 main( )中，该存储单元的内容被复制到一个名为 result 的 string 中：

```
result = version1(input, "***");
```

将 C-风格字符串用作 string 对象引用参数

对于函数 version1( )，您可能注意到了很有趣的一点：该函数的两个形参（s1 和 s2）的类型都是 const string &，但实参（input 和“***”）的类型分别是 string 和 const char *。由于 input 的类型为 string，因此让 s1 指向它没有任何问题。然而，程序怎么能够接受将 char 指针赋给 string 引用呢？

这里有两点需要说明。首先，string 类定义了一种 char *到 string 的转换功能，这使得可以使用 C-风格字符串来初始化 string 对象。其次是本章前面讨论过的类型为 const 引用的形参的一个属性。假设实参的类型与引用参数类型不匹配，但可被转换为引用类型，程序将创建一个正确类型的临时变量，使用转换后的实参值来初始化它，然后传递一个指向该临时变量的引用。例如，在本章前面，将 int 实参传递给 const double &形参时，就是以这种方式进行处理的。同样，也可以将实参 char *或 const char *传递给形参 const string &。

这种属性的结果是，如果形参类型为 const string &，在调用函数时，使用的实参可以是 string 对象或 C-风格字符串，如用引号括起的字符串字面量、以空字符结尾的 char 数组或指向 char 的指针变量。因此，下面的代码是可行的：

```
result = version1(input, "***");
```

函数 version2( )不创建临时 string 对象，而是直接修改原来的 string 对象：

```
const string & version2(string & s1, const string & s2)   // has side effect
{
    s1 = s2 + s1 + s2;
// safe to return reference passed to function
    return s1;
}
```

该函数可以修改 s1，因为不同于 s2，s1 没有被声明为 const。

由于 s1 是指向 main( )中一个对象（input）的引用，因此将 s1 最为引用返回是安全的。由于 s1 是指向 input 的引用，因此，下面一行代码：

```
result = version2(input, "###");
```

与下面的代码等价：

```
version2(input, "###");      // input altered directly by version2()
result = input;              // reference to s1 is reference to input
```

然而，由于 s1 是指向 input 的引用，调用该函数将带来修改 input 的副作用：

```
Your original string: It's not my fault.
Your string enhanced: ###It's not my fault.###
Your original string: ###It's not my fault.###
```

因此，如果要保留原来的字符串不变，这将是一种错误设计。

程序清单 8.7 中的第三个函数版本指出了什么不能做：

```
const string & version3(string & s1, const string & s2)   // bad design
{
    string temp;

    temp = s2 + s1 + s2;
// unsafe to return reference to local variable
    return temp;
}
```

它存在一个致命的缺陷：返回一个指向 version3( )中声明的变量的引用。这个函数能够通过编译（但编译器会发出警告），但当程序试图执行该函数时将崩溃。具体地说，问题是由下面的赋值语句引发的：

```
result = version3(input, "@@@");
```

程序试图引用已经释放的内存。

8.2.6　对象、继承和引用

ostream 和 ofstream 类凸现了引用的一个有趣属性。正如第 6 章介绍的，ofstream 对象可以使用 ostream 类的方法，这使得文件输入/输出的格式与控制台输入/输出相同。使得能够将特性从一个类传递给另一个类的语言特性被称为继承，这将在第 13 章详细讨论。简单地说，ostream 是基类（因为 ofstream 是建立在它的基础之上的），而 ofstream 是派生类（因为它是从 ostream 派生而来的）。派生类继承了基类的方法，这意味着 ofstream 对象可以使用基类的特性，如格式化方法 precision( )和 setf( )。

继承的另一个特征是，基类引用可以指向派生类对象，而无需进行强制类型转换。这种特征的一个实际结果是，可以定义一个接受基类引用作为参数的函数，调用该函数时，可以将基类对象作为参数，也可以将派生类对象作为参数。例如，参数类型为 ostream &的函数可以接受 ostream 对象（如 cout）或您声明

的 ofstream 对象作为参数。

程序清单 8.8 通过调用同一个函数（只有函数调用参数不同）将数据写入文件和显示到屏幕上来说明了这一点。该程序要求用户输入望远镜物镜和一些目镜的焦距，然后计算并显示每个目镜的放大倍数。放大倍数等于物镜的焦距除以目镜的焦距，因此计算起来很简单。该程序还使用了一些格式化方法，这些方法用于 cout 和 ofstream 对象（在这个例子中为 fout）时作用相同。

程序清单 8.8　filefunc.cpp

```
//filefunc.cpp -- function with ostream & parameter
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

void file_it(ostream & os, double fo, const double fe[],int n);
const int LIMIT = 5;
int main()
{
    ofstream fout;
    const char * fn = "ep-data.txt";
    fout.open(fn);
    if (!fout.is_open())
    {
        cout << "Can't open " << fn << ". Bye.\n";
        exit(EXIT_FAILURE);
    }
    double objective;
    cout << "Enter the focal length of your "
            "telescope objective in mm: ";
    cin >> objective;
    double eps[LIMIT];
    cout << "Enter the focal lengths, in mm, of " << LIMIT
         << " eyepieces:\n";
    for (int i = 0; i < LIMIT; i++)
    {
        cout << "Eyepiece #" << i + 1 << ": ";
        cin >> eps[i];
    }
    file_it(fout, objective, eps, LIMIT);
    file_it(cout, objective, eps, LIMIT);
    cout << "Done\n";
    return 0;
}
void file_it(ostream & os, double fo, const double fe[],int n)
{
    ios_base::fmtflags initial;
    initial = os.setf(ios_base::fixed); // save initial formatting state
    os.precision(0);
    os << "Focal length of objective: " << fo << " mm\n";
    os.setf(ios::showpoint);
    os.precision(1);
    os.width(12);
    os << "f.l. eyepiece";
    os.width(15);
```

```
    os << "magnification" << endl;
    for (int i = 0; i < n; i++)
    {
        os.width(12);
        os << fe[i];
        os.width(15);
        os << int (fo/fe[i] + 0.5) << endl;
    }
    os.setf(initial);   // restore initial formatting state
}
```

下面是该程序的运行情况：

```
Enter the focal length of your telescope objective in mm: 1800
Enter the focal lengths, in mm, of 5 eyepieces:
Eyepiece #1: 30
Eyepiece #2: 19
Eyepiece #3: 14
Eyepiece #4: 8.8
Eyepiece #5: 7.5
Focal length of objective: 1800 mm
f.l. eyepiece  magnification
        30.0             60
        19.0             95
        14.0            129
         8.8            205
         7.5            240
Done
```

下述代码行将目镜数据写入到文件 ep-data.txt 中：

```
file_it(fout, objective, eps, LIMIT);
```

而下述代码行将同样的信息以同样的格式显示到屏幕上：

```
file_it(cout, objective, eps, LIMIT);
```

程序说明

对于该程序，最重要的一点是，参数 os（其类型为 ostream &）可以指向 ostream 对象（如 cout），也可以指向 ofstream 对象（如 fout）。该程序还演示了如何使用 ostream 类中的格式化方法。下面复习（介绍）其中的一些，更详细的讨论请参阅第 17 章。

方法 setf( )让您能够设置各种格式化状态。例如，方法调用 setf(ios_base::fixed)将对象置于使用定点表示法的模式；setf(ios_base::showpoint)将对象置于显示小数点的模式，即使小数部分为零。方法 precision( )指定显示多少位小数（假定对象处于定点模式下）。所有这些设置都将一直保持不变，直到再次调用相应的方法重新设置它们。方法 width( )设置下一次输出操作使用的字段宽度，这种设置只在显示下一个值时有效，然后将恢复到默认设置。默认的字段宽度为零，这意味着刚好能容纳下要显示的内容。

函数 file_it( )使用了两个有趣的方法调用：

```
ios_base::fmtflags initial;
initial = os.setf(ios_base::fixed); // save initial formatting state
...
os.setf(initial);   // restore initial formatting state
```

方法 setf( )返回调用它之前有效的所有格式化设置。ios_base::fmtflags 是存储这种信息所需的数据类型名称。因此，将返回值赋给 initial 将存储调用 file_it( )之前的格式化设置，然后便可以使用变量 initial 作为参数来调用 setf( )，将所有的格式化设置恢复到原来的值。因此，该函数将对象回到传递给 file_it( )之前的状态。

了解更多有关类的知识将有助于更好地理解这些方法的工作原理，以及为何在代码中使用 ios_base。

然而，您不用等到第 17 章才使用这些方法。

需要说明的最后一点是，每个对象都存储了自己的格式化设置。因此，当程序将 cout 传递给 file_it( )时，cout 的设置将被修改，然后被恢复；当程序将 fout 传递给 file_it( )时，fout 的设置将被修改，然后被恢复。

8.2.7 何时使用引用参数

使用引用参数的主要原因有两个。

- 程序员能够修改调用函数中的数据对象。
- 通过传递引用而不是整个数据对象，可以提高程序的运行速度。

当数据对象较大时（如结构和类对象），第二个原因最重要。这些也是使用指针参数的原因。这是有道理的，因为引用参数实际上是基于指针的代码的另一个接口。那么，什么时候应使用引用、什么时候应使用指针呢？什么时候应按值传递呢？下面是一些指导原则：

对于使用传递的值而不作修改的函数。

- 如果数据对象很小，如内置数据类型或小型结构，则按值传递。
- 如果数据对象是数组，则使用指针，因为这是唯一的选择，并将指针声明为指向 const 的指针。
- 如果数据对象是较大的结构，则使用 const 指针或 const 引用，以提高程序的效率。这样可以节省复制结构所需的时间和空间。
- 如果数据对象是类对象，则使用 const 引用。类设计的语义常常要求使用引用，这是 C++新增这项特性的主要原因。因此，传递类对象参数的标准方式是按引用传递。

对于修改调用函数中数据的函数：

- 如果数据对象是内置数据类型，则使用指针。如果看到诸如 fixit（&x）这样的代码（其中 x 是 int），则很明显，该函数将修改 x。
- 如果数据对象是数组，则只能使用指针。
- 如果数据对象是结构，则使用引用或指针。
- 如果数据对象是类对象，则使用引用。

当然，这只是一些指导原则，很可能有充分的理由做出其他的选择。例如，对于基本类型，cin 使用引用，因此可以使用 cin>>n，而不是 cin >> &n。

8.3 默认参数

下面介绍 C++的另一项新内容——默认参数。默认参数指的是当函数调用中省略了实参时自动使用的一个值。例如，如果将 void wow（int n）设置成 n 有默认值为 1，则函数调用 wow( )相当于 wow（1）。这极大地提高了使用函数的灵活性。假设有一个名为 left( )的函数，它将字符串和 n 作为参数，并返回该字符串的前 n 个字符。更准确地说，该函数返回一个指针，该指针指向由原始字符串中被选中的部分组成的字符串。例如，函数调用 left(“theory”, 3)将创建新字符串“the”，并返回一个指向该字符串的指针。现在假设第二个参数的默认值被设置为 1，则函数调用 left(“theory”, 3)仍像前面讲述的那样工作，3 将覆盖默认值。但函数调用 left(“theory”)不会出错，它认为第二个参数的值为 1，并返回指向字符串“t”的指针。如果程序经常需要抽取一个字符组成的字符串，而偶尔需要抽取较长的字符串，则这种默认值将很有帮助。

如何设置默认值呢？必须通过函数原型。由于编译器通过查看原型来了解函数所使用的参数数目，因此函数原型也必须将可能的默认参数告知程序。方法是将值赋给原型中的参数。例如，left( )的原型如下：

```
char * left(const char * str, int n = 1);
```

您希望该函数返回一个新的字符串，因此将其类型设置为 char *（指向 char 的指针）；您希望原始字符串保持不变，因此对第一个参数使用了 const 限定符；您希望 n 的默认值为 1，因此将这个值赋给 n。默认参数值是初始化值，因此上面的原型将 n 初始化为 1。如果省略参数 n，则它的值将为 1；否则，传递的值将覆盖 1。

对于带参数列表的函数，必须从右向左添加默认值。也就是说，要为某个参数设置默认值，则必须为它右边的所有参数提供默认值：

```
int harpo(int n, int m = 4, int j = 5);          // VALID
int chico(int n, int m = 6, int j);              // INVALID
int groucho(int k = 1, int m = 2, int n = 3);    // VALID
```

例如，harpo( )原型允许调用该函数时提供 1 个、2 个或 3 个参数：

```
beeps = harpo(2);        // same as harpo(2,4,5)
beeps = harpo(1,8);      // same as harpo(1,8,5)
beeps = harpo (8,7,6);   // no default arguments used
```

实参按从左到右的顺序依次被赋给相应的形参，而不能跳过任何参数。因此，下面的调用是不允许的：

```
beeps = harpo(3, ,8);    // invalid, doesn't set m to 4
```

默认参数并非编程方面的重大突破，而只是提供了一种便捷的方式。在设计类时您将发现，通过使用默认参数，可以减少要定义的析构函数、方法以及方法重载的数量。

程序清单 8.9 使用了默认参数。请注意，只有原型指定了默认值。函数定义与没有默认参数时完全相同。

程序清单 8.9　left.cpp

```
// left.cpp -- string function with a default argument
#include <iostream>
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
    using namespace std;
    char sample[ArSize];
    cout << "Enter a string:\n";
    cin.get(sample,ArSize);
    char *ps = left(sample, 4);
    cout << ps << endl;
    delete [] ps;        // free old string
    ps = left(sample);
    cout << ps << endl;
    delete [] ps;        // free new string
    return 0;
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // copy characters
    while (i <= n)
        p[i++] = '\0';  // set rest of string to '\0'
    return p;
}
```

下面是该程序的运行情况：

```
Enter a string:
forthcoming
fort
f
```

程序说明

该程序使用 new 创建一个新的字符串，以存储被选择的字符。一种可能出现的尴尬情况是，不合作的用户要求的字符数目可能为负。在这种情况下，函数将字符计数设置为 0，并返回一个空字符串。另一种可能出现的尴尬情况是，不负责任的用户要求的字符数目可能多于字符串包含的字符数，为预防这种情况，函数使用了一个组合测试：

```
i < n && str[i]
```

i < n 测试让循环复制了 n 个字符后终止。测试的第二部分——表达式 str[i]，是要复制的字符的编码。遇到空值字符（其编码为 0）后，循环将结束。这样，while 循环将使字符串以空值字符结束，并将余下的空间（如果有的话）设置为空值字符。

另一种设置新字符串长度的方法是，将 n 设置为传递的值和字符串长度中较小的一个：

```
int len = strlen(str);
n = (n < len) ? n : len;     // the lesser of n and len
char * p = new char[n+1];
```

这将确保 new 分配的空间不会多于存储字符串所需的空间。如果用户执行像 left("Hi!", 32767)这样的调用，则这种方法很有用。第一种方法将把“Hi!”复制到由 32767 个字符组成的数组中，并将除前 3 个字符之外的所有字符都设置为空值字符；第二种方法将“Hi!”复制到由 4 个字符组成的数组中。但由于添加了另外一个函数调用（strlen( )），因此程序将更长，运行速度将降低，同时还必须包含头文件 cstring（或 string.h）。C 程序员倾向于选择运行速度更快、更简洁的代码，因此需要程序员在正确使用函数方面承担更多责任。然而，C++的传统是更强调可靠性。毕竟，速度较慢但能正常运行的程序，要比运行速度虽快但无法正常运行的程序好。如果调用 strlen( )所需的时间很长，则可以让 left( )直接确定 n 和字符串长度哪个小。例如，当 m 的值等于 n 或到达字符串结尾时，下面的循环都将终止：

```
int m = 0;
while (m <= n && str[m] != '\0')
      m++;
char * p = new char[m+1]:
// use m instead of n in rest of code
```

别忘了，在 str[m]不是空值字符时，表达式 str[m] != '\0'的结果为 true，否则为 false。由于在&&表达式中，非零值被转换为 true，而零被转换为 false，因此也可以这样编写这个 while 测试：

```
while (m<=n && str[m])
```

8.4 函数重载

函数多态是 C++在 C 语言的基础上新增的功能。默认参数让您能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）让您能够使用多个同名的函数。术语“多态”指的是有多种形式，因此函数多态允许函数可以有多种形式。类似地，术语“函数重载”指的是可以有多个同名的函数，因此对名称进行了重载。这两个术语指的是同一回事，但我们通常使用函数重载。可以通过函数重载来设计一系列函数——它们完成相同的工作，但使用不同的参数列表。

重载函数就像是有多种含义的动词。例如，Piggy 小姐可以在棒球场为家乡球队助威（root），也可以在地里种植（root）菌类作物。根据上下文可以知道在每一种情况下，root 的含义是什么。同样，C++使用上下文来确定要使用的重载函数版本。

函数重载的关键是函数的参数列表——也称为函数特征标（function signature）。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目和/或参数类型不同，则特征标也不同。例如，可以定义一组原型如下的 print( )函数：

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5
```

使用 print( )函数时，编译器将根据所采取的用法使用有相应特征标的原型：

```
print("Pancakes", 15);          // use #1
print("Syrup");                 // use #5
print(1999.0, 10);              // use #2
print(1999, 12);                // use #4
print(1999L, 15);               // use #3
```

例如，print(“Pancakes”, 15)使用一个字符串和一个整数作为参数，这与#1 原型匹配。

使用被重载的函数时，需要在函数调用中使用正确的参数类型。例如，对于下面的语句：

```
unsigned int year = 3210;
print(year, 6);            // ambiguous call
```

print( )调用与哪个原型匹配呢？它不与任何原型匹配！没有匹配的原型并不会自动停止使用其中的某个函数，因为 C++将尝试使用标准类型转换强制进行匹配。如果#2 原型是 print( )唯一的原型，则函数调用 print(year, 6)将把 year 转换为 double 类型。但在上面的代码中，有 3 个将数字作为第一个参数的原型，因此有 3 种转换 year 的方式。在这种情况下，C++将拒绝这种函数调用，并将其视为错误。

一些看起来彼此不同的特征标是不能共存的。例如，请看下面的两个原型：

```
double cube(double x);
double cube(double & x);
```

您可能认为可以在此处使用函数重载，因为它们的特征标看起来不同。然而，请从编译器的角度来考虑这个问题。假设有下面这样的代码：

```
cout << cube(x);
```

参数 x 与 double x 原型和 double &x 原型都匹配，因此编译器无法确定究竟应使用哪个原型。为避免这种混乱，编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标。

匹配函数时，并不区分 const 和非 const 变量。请看下面的原型：

```
void dribble(char * bits);          // overloaded
void dribble (const char *cbits);   // overloaded
void dabble(char * bits);           // not overloaded
void drivel(const char * bits);     // not overloaded
```

下面列出了各种函数调用对应的原型：

```
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);       // dribble(const char *);
dribble(p2);       // dribble(char *);
dabble(p1);        // no match
dabble(p2);        // dabble(char *);
drivel(p1);        // drivel(const char *);
drivel(p2);        // drivel(const char *);
```

dribble( )函数有两个原型，一个用于 const 指针，另一个用于常规指针，编译器将根据实参是否为 const 来决定使用哪个原型。dribble( )函数只与带非 const 参数的调用匹配，而 drivel( )函数可以与带 const 或非

const 参数的调用匹配。drivel( )和 dabble( )之所以在行为上有这种差别，主要是由于将非 const 值赋给 const 变量是合法的，但反之则是非法的。

请记住，是特征标，而不是函数类型使得可以对函数进行重载。例如，下面的两个声明是互斥的：

```
long gronk(int n, float m);      // same signatures,
double gronk(int n, float m);    // hence not allowed
```

因此，C++不允许以这种方式重载 gronk( )。返回类型可以不同，但特征标也必须不同：

```
long gronk(int n, float m);        // different signatures,
double gronk(float n, float m);    // hence allowed
```

在本章稍后讨论过模板后，将进一步讨论函数匹配的问题。

重载引用参数

类设计和 STL 经常使用引用参数，因此知道不同引用类型的重载很有用。请看下面三个原型：

```
void sink(double & r1);        // matches modifiable lvalue
void sank(const double & r2); // matches modifiable or const lvalue, rvalue
void sunk(double && r3);       // matches rvalue
```

左值引用参数 r1 与可修改的左值参数（如 double 变量）匹配；const 左值引用参数 r2 与可修改的左值参数、const 左值参数和右值参数（如两个 double 值的和）匹配；最后，左值引用参数 r3 与左值匹配。注意到与 r1 或 r3 匹配的参数都与 r2 匹配。这就带来了一个问题：如果重载使用这三种参数的函数，结果将如何？答案是将调用最匹配的版本：

```
void staff(double & rs);          // matches modifiable lvalue
voit staff(const double & rcs);   // matches rvalue, const lvalue
void stove(double & r1);          // matches modifiable lvalue
void stove(const double & r2);    // matches const lvalue
void stove(double && r3);         // matches rvalue
```

这让您能够根据参数是左值、const 还是右值来定制函数的行为：

```
double x = 55.5;
const double y = 32.0;
stove(x);              // calls stove(double &)
stove(y);              // calls stove(const double &)
stove(x+y);            // calls stove(double &&)
```

如果没有定义函数 stove(double &&)，stove(x+y)将调用函数 stove(const double &)。

8.4.1 重载示例

本章前面创建了一个 left( )函数，它返回一个指针，指向字符串的前 n 个字符。下面添加另一个 left( )函数，它返回整数的前 n 位。例如，可以使用该函数来查看被存储为整数的、美国邮政编码的前 3 位——如果要根据城区分拣邮件，则这种操作很有用。

该函数的整数版本编写起来比字符串版本更困难些，因为并不是整数的每一位被存储在相应的数组元素中。一种方法是，先计算数字包含多少位。将数字除以 10 便可以去掉一位，因此可以使用除法来计算数位。更准确地说，可以用下面的循环完成这种工作：

```
unsigned digits = 1;
while (n /= 10)
    digits++;
```

上述循环计算每次删除 n 中的一位时，需要多少次才能删除所有的位。前面讲过，n / = 10 是 n = n / 10 的缩写。例如，如果 n 为 8，则该测试条件将 8/10 的值（0，由于这是整数除法）赋给 n。这将结束循环，digits 的值仍然为 1。但如果 n 为 238，第一轮循环测试将 n 设置为 238/10，即 23。这个值不为零，因此循环将 digits 增加到 2。下一轮循环将 n 设置为 23/10，即 2。这个值还是不为零，因此 digits 将增加到 3。下

一轮循环将 n 设置为 2/10，即 0，从而结束循环，而 digits 被设置为正确的值——3。

现在假设知道数字共有 5 位，并要返回前 3 位，则将这个数除以 10 后再除以 10，便可以得到所需的值。每除以 10 次就删除数字的最后一位。要知道需要删除多少位，只需将总位数减去要获得的位数即可。例如，要获得 9 位数的前 4 位，需要删除后面的 5 位。可以这样编写代码：

```
ct = digits - ct;
while (ct--)
    num /= 10;
return num;
```

程序清单 8.10 将上述代码放到了一个新的 left( )函数中。该函数还包含一些用于处理特殊情况的代码，如用户要求显示 0 位或要求显示的位数多于总位数。由于新 left( )的特征标不同于旧的 left( )，因此可以在同一个程序中使用这两个函数。

程序清单 8.10　leftover.cpp

```
// leftover.cpp -- overloading the left() function
#include <iostream>
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);

int main()
{
     using namespace std;
    char * trip = "Hawaii!!";   // test value
    unsigned long n = 12345678; // test value
    int i;
    char * temp;

    for (i = 1; i < 10; i++)
    {
        cout << left(n, i) << endl;
        temp = left(trip,i);
        cout << temp << endl;
        delete [] temp; // point to temporary storage
    }
    return 0;
}

// This function returns the first ct digits of the number num.
unsigned long left(unsigned long num, unsigned ct)
{
    unsigned digits = 1;
    unsigned long n = num;

    if (ct == 0 || num == 0)
        return 0;       // return 0 if no digits
    while (n /= 10)
        digits++;
    if (digits > ct)
    {
    ct = digits - ct;
    while (ct--)
        num /= 10;
    return num;         // return left ct digits
    }
```

```
    else                    // if ct >= number of digits
        return num;         // return the whole number
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // copy characters
    while (i <= n)
        p[i++] = '\0';  // set rest of string to '\0'
    return p;
}
```

下面是该程序的输出：

```
1
H
12
Ha
123
Haw
1234
Hawa
12345
Hawai
123456
Hawaii
1234567
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

8.4.2 何时使用函数重载

虽然函数重载很吸引人，但也不要滥用。仅当函数基本上执行相同的任务，但使用不同形式的数据时，才应采用函数重载。另外，您可能还想知道，是否可以通过使用默认参数来实现同样的目的。例如，可以用两个重载函数来代替面向字符串的 left( )函数：

```
char * left(const char * str, unsigned n);   // two arguments
char * left(const char * str);               // one argument
```

使用一个带默认参数的函数要简单些。只需编写一个函数（而不是两个函数），程序也只需为一个函数（而不是两个）请求内存；需要修改函数时，只需修改一个。然而，如果需要使用不同类型的参数，则默认参数便不管用了，在这种情况下，应该使用函数重载。

什么是名称修饰

C++如何跟踪每一个重载函数呢？它给这些函数指定了秘密身份。使用 C++开发工具中的编辑器编写和编译程序时，C++编译器将执行一些神奇的操作——名称修饰（name decoration）或名称矫正（name

mangling），它根据函数原型中指定的形参类型对每个函数名进行加密。请看下述未经修饰的函数原型：

```
long MyFunctionFoo(int, float);
```

这种格式对于人类来说很适合；我们知道函数接受两个参数（一个为 int 类型，另一个为 float 类型），并返回一个 long 值。而编译器将名称转换为不太好看的内部表示，来描述该接口，如下所示：

```
?MyFunctionFoo@@YAXH
```

对原始名称进行的表面看来无意义的修饰（或矫正，因人而异）将对参数数目和类型进行编码。添加的一组符号随函数特征标而异，而修饰时使用的约定随编译器而异。

8.5　函数模板

现在的 C++编译器实现了 C++新增的一项特性——函数模板。函数模板是通用的函数描述，也就是说，它们使用泛型来定义函数，其中的泛型可用具体的类型（如 int 或 double）替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。由于模板允许以泛型（而不是具体类型）的方式编写程序，因此有时也被称为通用编程。由于类型是用参数表示的，因此模板特性有时也被称为参数化类型（parameterized types）。下面介绍为何需要这种特性以及其工作原理。

在前面的程序清单 8.4 中，定义了一个交换两个 int 值的函数。假设要交换两个 double 值，则一种方法是复制原来的代码，并用 double 替换所有的 int。如果需要交换两个 char 值，可以再次使用同样的技术。进行这种修改将浪费宝贵的时间，且容易出错。如果进行手工修改，则可能会漏掉一个 int。如果进行全局查找和替换（如用 double 替换 int）时，可能将：

```
int x;
short interval;
```

转换为：

```
double x;            // intended change of type
short doubleerval;   // unintended change of variable name
```

C++的函数模板功能能自动完成这一过程，可以节省时间，而且更可靠。

函数模板允许以任意类型的方式来定义函数。例如，可以这样建立一个交换模板：

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

第一行指出，要建立一个模板，并将类型命名为 AnyType。关键字 template 和 typename 是必需的，除非可以使用关键字 class 代替 typename。另外，必须使用尖括号。类型名可以任意选择（这里为 AnyType），只要遵守 C++命名规则即可；许多程序员都使用简单的名称，如 T。余下的代码描述了交换两个 AnyType 值的算法。模板并不创建任何函数，而只是告诉编译器如何定义函数。需要交换 int 的函数时，编译器将按模板模式创建这样的函数，并用 int 代替 AnyType。同样，需要交换 double 的函数时，编译器将按模板模式创建这样的函数，并用 double 代替 AnyType。

在标准 C++98 添加关键字 typename 之前，C++使用关键字 class 来创建模板。也就是说，可以这样编写模板定义：

```
template <class AnyType>
void Swap(AnyType &a, AnyType &b)
{
```

```
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

typename 关键字使得参数 AnyType 表示类型这一点更为明显；然而，有大量代码库是使用关键字 class 开发的。在这种上下文中，这两个关键字是等价的。本书使用了这两种形式，旨在让您在其他地方遇到它们时不会感到陌生。

提示：如果需要多个将同一种算法用于不同类型的函数，请使用模板。如果不考虑向后兼容的问题，并愿意键入较长的单词，则声明类型参数时，应使用关键字 typename 而不使用 class。

要让编译器知道程序需要一个特定形式的交换函数，只需在程序中使用 Swap( )函数即可。编译器将检查所使用的参数类型，并生成相应的函数。程序清单 8.11 演示为何可以这样做。该程序的布局和使用常规函数时相同，在文件的开始位置提供模板函数的原型，并在 main( )后面提供模板函数的定义。这个示例采用了更常见的做法，即将 T 而不是 AnyType 用作类型参数。

程序清单 8.11　funtemp.cpp

```
// funtemp.cpp -- using a function template
#include <iostream>
// function template prototype
template <typename T>  // or class T
void Swap(T &a, T &b);

int main()
{
    using namespace std;
    int i = 10;
    int j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j);  // generates void Swap(int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";

    double x = 24.5;
    double y = 81.7;
    cout << "x, y = " << x << ", " << y << ".\n";
    cout << "Using compiler-generated double swapper:\n";
    Swap(x,y);  // generates void Swap(double &, double &)
    cout << "Now x, y = " << x << ", " << y << ".\n";
    // cin.get();
    return 0;
}

// function template definition
template <typename T>  // or class T
void Swap(T &a, T &b)
{
    T temp;   // temp a variable of type T
    temp = a;
    a = b;
    b = temp;
}
```

程序清单 8.11 中的第一个 Swap( )函数接受两个 int 参数，因此编译器生成该函数的 int 版本。也就是说，用 int 替换所有的 T，生成下面这样的定义：

```
void Swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

程序员看不到这些代码，但编译器确实生成并在程序中使用了它们。第二个 Swap( )函数接受两个 double 参数，因此编译器将生成 double 版本。也就是说，用 double 替换 T，生成下述代码：

```
void Swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

下面是程序清单 8.11 中程序的输出，从中可知，这种处理方式是可行的：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
Using compiler-generated double swapper:
Now x, y = 81.7, 24.5.
```

注意，函数模板不能缩短可执行程序。对于程序清单 8.11，最终仍将由两个独立的函数定义，就像以手工方式定义了这些函数一样。最终的代码不包含任何模板，而只包含了为程序生成的实际函数。使用模板的好处是，它使生成多个函数定义更简单、更可靠。

更常见的情形是，将模板放在头文件中，并在需要使用模板的文件中包含头文件。头文件将在第 9 章讨论。

8.5.1　重载的模板

需要多个对不同类型使用同一种算法的函数时，可使用模板，如程序清单 8.11 所示。然而，并非所有的类型都使用相同的算法。为满足这种需求，可以像重载常规函数定义那样重载模板定义。和常规重载一样，被重载的模板的函数特征标必须不同。例如，程序清单 8.12 新增了一个交换模板，用于交换两个数组中的元素。原来的模板的特征标为(T &, T &)，而新模板的特征标为(T [ ], T [ ], int)。注意，在后一个模板中，最后一个参数的类型为具体类型（int），而不是泛型。并非所有的模板参数都必须是模板参数类型。

编译器见到 twotemps.cpp 中第一个 Swap( )函数调用时，发现它有两个 int 参数，因此将它与原来的模板匹配。但第二次调用将两个 int 数组和一个 int 值用作参数，这与新模板匹配。

程序清单 8.12　twotemps.cpp

```
// twotemps.cpp -- using overloaded template functions
#include <iostream>
template <typename T>     // original template
void Swap(T &a, T &b);

template <typename T>     // new template
void Swap(T *a, T *b, int n);
```

```
void Show(int a[]);
const int Lim = 8;
int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j);              // matches original template
    cout << "Now i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,7,2,0,1,9,6,9};
    cout << "Original arrays:\n";
    Show(d1);
    Show(d2);
    Swap(d1,d2,Lim);        // matches new template
    cout << "Swapped arrays:\n";
    Show(d1);
    Show(d2);
    // cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}

void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << endl;
}
```

下面是程序清单 8.12 中程序的输出：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```

8.5.2　模板的局限性

假设有如下模板函数：

```
template <class T>     // or template <typename T>
void f(T a, T b)
{...}
```

通常，代码假定可执行哪些操作。例如，下面的代码假定定义了赋值，但如果 T 为数组，这种假设将不成立：

```
a = b;
```

同样，下面的语句假设定义了<，但如果 T 为结构，该假设便不成立：

```
if (a > b)
```

另外，为数组名定义了运算符>，但由于数组名为地址，因此它比较的是数组的地址，而这可能不是您希望的。下面的语句假定为类型 T 定义了乘法运算符，但如果 T 为数组、指针或结构，这种假设便不成立：

```
T c = a*b;
```

总之，编写的模板函数很可能无法处理某些类型。另一方面，有时候通用化是有意义的，但 C++语法不允许这样做。例如，将两个包含位置坐标的结构相加是有意义的，虽然没有为结构定义运算符+。一种解决方案是，C++允许您重载运算符+，以便能够将其用于特定的结构或类（运算符重载将在第 11 章讨论）。这样使用运算符+的模板便可处理重载了运算符+的结构。另一种解决方案是，为特定类型提供具体化的模板定义，下面就来介绍这种解决方案。

8.5.3　显式具体化

假设定义了如下结构：

```
struct job
{
      char name[40];
      double salary;
      int floor;
};
```

另外，假设希望能够交换两个这种结构的内容。原来的模板使用下面的代码来完成交换：

```
temp = a;
a = b;
b = temp;
```

由于 C++允许将一个结构赋给另一个结构，因此即使 T 是一个 job 结构，上述代码也适用。然而，假设只想交换 salary 和 floor 成员，而不交换 name 成员，则需要使用不同的代码，但 Swap( )的参数将保持不变（两个 job 结构的引用），因此无法使用模板重载来提供其他的代码。

然而，可以提供一个具体化函数定义——称为显式具体化（explicit specialization），其中包含所需的代码。当编译器找到与函数调用匹配的具体化定义时，将使用该定义，而不再寻找模板。

具体化机制随着 C++的演变而不断变化。下面介绍 C++标准定义的形式。

1. 第三代具体化（ISO/ANSI C++标准）

试验其他具体化方法后，C++98 标准选择了下面的方法。

- 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本。
- 显式具体化的原型和定义应以 template<>打头，并通过名称来指出类型。
- 具体化优先于常规模板，而非模板函数优先于具体化和常规模板。

下面是用于交换 job 结构的非模板函数、模板函数和具体化的原型：

```
// non template function prototype
void Swap(job &, job &);

// template prototype
template <typename T>
void Swap(T &, T &);

// explicit specialization for the job type
template <> void Swap<job>(job &, job &);
```

正如前面指出的，如果有多个原型，则编译器在选择原型时，非模板版本优先于显式具体化和模板版本，而显式具体化优先于使用模板生成的版本。例如，在下面的代码中，第一次调用 Swap( )时使用通用版本，而第二次调用使用基于 job 类型的显式具体化版本。

```
...
template <class T>          // template
void Swap(T &, T &);

// explicit specialization for the job type
template <> void Swap<job>(job &, job &);
int main()
{
    double u, v;
    ...
    Swap(u,v);  // use template
    job a, b;
    ...
    Swap(a,b);  // use void Swap<job>(job &, job &)
}
```

Swap<job>中的<job>是可选的，因为函数的参数类型表明，这是 job 的一个具体化。因此，该原型也可以这样编写：

```
template <> void Swap(job &, job &);   // simpler form
```

下面来看一看显式具体化的工作方式。

2. 显式具体化示例

程序清单 8.13 演示了显式具体化的工作方式。

程序清单 8.13　twoswap.cpp

```
// twoswap.cpp -- specialization overrides a template
#include <iostream>
template <typename T>
void Swap(T &a, T &b);

struct job
{
    char name[40];
    double salary;
```

```
    int floor;
};

// explicit specialization
template <> void Swap<job>(job &j1, job &j2);
void Show(job &j);

int main()
{
    using namespace std;
    cout.precision(2);
    cout.setf(ios::fixed, ios::floatfield);
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j);    // generates void Swap(int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";

    job sue = {"Susan Yaffee", 73000.60, 7};
    job sidney = {"Sidney Taffee", 78060.72, 9};
    cout << "Before job swapping:\n";
    Show(sue);
    Show(sidney);
    Swap(sue, sidney); // uses void Swap(job &, job &)
    cout << "After job swapping:\n";
    Show(sue);
    Show(sidney);
    // cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)    // general version
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

// swaps just the salary and floor fields of a job structure

template <> void Swap<job>(job &j1, job &j2)  // specialization
{
    double t1;
    int t2;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    t2 = j1.floor;
    j1.floor = j2.floor;
    j2.floor = t2;
}

void Show(job &j)
```

```
{
    using namespace std;
    cout << j.name << ": $" << j.salary
         << " on floor " << j.floor << endl;
}
```

下面是该程序的输出：

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Before job swapping:
Susan Yaffee: $73000.60 on floor 7
Sidney Taffee: $78060.72 on floor 9
After job swapping:
Susan Yaffee: $78060.72 on floor 9
Sidney Taffee: $73000.60 on floor 7
```

8.5.4 实例化和具体化

为进一步了解模板，必须理解术语实例化和具体化。记住，在代码中包含函数模板本身并不会生成函数定义，它只是一个用于生成函数定义的方案。编译器使用模板为特定类型生成函数定义时，得到的是模板实例（instantiation）。例如，在程序清单 8.13 中，函数调用 Swap(i, j)导致编译器生成 Swap( )的一个实例，该实例使用 int 类型。模板并非函数定义，但使用 int 的模板实例是函数定义。这种实例化方式被称为隐式实例化（implicit instantiation），因为编译器之所以知道需要进行定义，是由于程序调用 Swap( )函数时提供了 int 参数。

最初，编译器只能通过隐式实例化，来使用模板生成函数定义，但现在 C++还允许显式实例化（explicit instantiation）。这意味着可以直接命令编译器创建特定的实例，如 Swap<int>( )。其语法是，声明所需的种类——用<>符号指示类型，并在声明前加上关键字 template：

```
template void Swap<int>(int, int);  // explicit instantiation
```

实现了这种特性的编译器看到上述声明后，将使用 Swap( )模板生成一个使用 int 类型的实例。也就是说，该声明的意思是“使用 Swap( )模板生成 int 类型的函数定义。”

与显式实例化不同的是，显式具体化使用下面两个等价的声明之一：

```
template <> void Swap<int>(int &, int &);  // explicit specialization
template <> void Swap(int &, int &);       // explicit specialization
```

区别在于，这些声明的意思是“不要使用 Swap( )模板来生成函数定义，而应使用专门为 int 类型显式地定义的函数定义”。这些原型必须有自己的函数定义。显式具体化声明在关键字 template 后包含<>，而显式实例化没有。

警告：试图在同一个文件（或转换单元）中使用同一种类型的显式实例和显式具体化将出错。

还可通过在程序中使用函数来创建显式实例化。例如，请看下面的代码：

```
template <class T>
T Add(T a, T b)     // pass by value
{
    return a + b;
}
...
int m = 6;
double x = 10.2;
cout << Add<double>(x, m) << endl;  // explicit instantiation
```

这里的模板与函数调用 Add(x, m)不匹配，因为该模板要求两个函数参数的类型相同。但通过使用 Add<double>(x, m)，可强制为 double 类型实例化，并将参数 m 强制转换为 double 类型，以便与函数 Add<double>(double, double)的第二个参数匹配。

如果对 Swap()做类似的处理，结果将如何呢？

```
int m = 5;
double x  = 14.3;
Swap<double>(m, x);  // almost works
```

这将为类型 double 生成一个显式实例化。不幸的是，这些代码不管用，因为第一个形参的类型为 double &，不能指向 int 变量 m。

隐式实例化、显式实例化和显式具体化统称为具体化（specialization）。它们的相同之处在于，它们表示的都是使用具体类型的函数定义，而不是通用描述。

引入显式实例化后，必须使用新的语法——在声明中使用前缀 template 和 template <>，以区分显式实例化和显式具体化。通常，功能越多，语法规则也越多。下面的代码片段总结了这些概念：

```
...
template <class T>
void Swap (T &, T &);  // template prototype

template <> void Swap<job>(job &, job &);   // explicit specialization for job
int main(void)
{
  template void Swap<char>(char &, char &); // explicit instantiation for char
  short a, b;
  ...
  Swap(a,b);    // implicit template instantiation for short
  job n, m;
  ...
  Swap(n, m);   // use explicit specialization for job
  char g, h;
  ...
  Swap(g, h);  // use explicit template instantiation for char
  ...
}
```

编译器看到 char 的显式实例化后，将使用模板定义来生成 Swap( )的 char 版本。对于其他 Swap( )调用，编译器根据函数调用中实际使用的参数，生成相应的版本。例如，当编译器看到函数调用 Swap(a, b)后，将生成 Swap( )的 short 版本，因为两个参数的类型都是 short。当编译器看到 Swap(n, m)后，将使用为 job 类型提供的独立定义（显式具体化）。当编译器看到 Swap(g, h)后，将使用处理显式实例化时生成的模板具体化。

8.5.5　编译器选择使用哪个函数版本

对于函数重载、函数模板和函数模板重载，C++需要（且有）一个定义良好的策略，来决定为函数调用使用哪一个函数定义，尤其是有多个参数时。这个过程称为重载解析（overloading resolution）。详细解释这个策略将需要将近一章的篇幅，因此我们先大致了解一下这个过程是如何进行的。

- 第 1 步：创建候选函数列表。其中包含与被调用函数的名称相同的函数和模板函数。
- 第 2 步：使用候选函数列表创建可行函数列表。这些都是参数数目正确的函数，为此有一个隐式转换序列，其中包括实参类型与相应的形参类型完全匹配的情况。例如，使用 float 参数的函数调用可以将该参数转换为 double，从而与 double 形参匹配，而模板可以为 float 生成一个实例。
- 第 3 步：确定是否有最佳的可行函数。如果有，则使用它，否则该函数调用出错。

考虑只有一个函数参数的情况，如下面的调用：

```
may('B');    // actual argument is type char
```

首先，编译器将寻找候选者，即名称为 may( )的函数和函数模板。然后寻找那些可以用一个参数调用的函数。例如，下面的函数符合要求，因为其名称与被调用的函数相同，且可只给它们传递一个参数：

```
void may(int);                          // #1
float may(float, float = 3);            // #2
void may(char);                         // #3
```

```
char * may(const char *);                    // #4
char may(const char &);                      // #5
template<class T> void may(const T &);       // #6
template<class T> void may(T *);             // #7
```

注意，只考虑特征标，而不考虑返回类型。其中的两个候选函数（#4 和#7）不可行，因为整数类型不能被隐式地转换（即没有显式强制类型转换）为指针类型。剩余的一个模板可用来生成具体化，其中 T 被替换为 char 类型。这样剩下 5 个可行的函数，其中的每一个函数，如果它是声明的唯一一个函数，都可以被使用。

接下来，编译器必须确定哪个可行函数是最佳的。它查看为使函数调用参数与可行的候选函数的参数匹配所需要进行的转换。通常，从最佳到最差的顺序如下所述。

1．完全匹配，但常规函数优先于模板。

2．提升转换（例如，char 和 shorts 自动转换为 int，float 自动转换为 double）。

3．标准转换（例如，int 转换为 char，long 转换为 double）。

4．用户定义的转换，如类声明中定义的转换。

例如，函数#1 优于函数#2，因为 char 到 int 的转换是提升转换（参见第 3 章），而 char 到 float 的转换是标准转换（参见第 3 章）。函数#3、函数#5 和函数#6 都优于函数#1 和#2，因为它们都是完全匹配的。#3 和#5 优于#6，因为#6 函数是模板。这种分析引出了两个问题。什么是完全匹配？如果两个函数（如#3 和#5）都完全匹配，将如何办呢？通常，有两个函数完全匹配是一种错误，但这一规则有两个例外。显然，我们需要对这一点做更深入的探讨。

1. 完全匹配和最佳匹配

进行完全匹配时，C++允许某些“无关紧要的转换”。表 8.1 列出了这些转换——Type 表示任意类型。例如，int 实参与 int &形参完全匹配。注意，Type 可以是 char &这样的类型，因此这些规则包括从 char & 到 const char &的转换。Type（argument-list）意味着用作实参的函数名与用作形参的函数指针只要返回类型和参数列表相同，就是匹配的（第 7 章介绍了函数指针以及为何可以将函数名作为参数传递给接受函数指针的函数）。第 9 章将介绍关键字 volatile。

表 8.1　完全匹配允许的无关紧要转换

从　实　参	到　形　参
Type	Type &
Type &	Type
Type [ ]	* Type
Type（argument-list）	Type（*）（argument-list）
Type	const Type
Type	volatile Type
Type *	const Type
Type *	volatile Type *

假设有下面的函数代码：

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(ink);
```

在这种情况下，下面的原型都是完全匹配的：

```
void recycle(blot);          // #1  blot-to-blot
void recycle(const blot);    // #2  blot-to-(const blot)
void recycle(blot &);        // #3  blot-to-(blot &)
void recycle(const blot &);  // #4  blot-to-(const blot &)
```

正如您预期的，如果有多个匹配的原型，则编译器将无法完成重载解析过程；如果没有最佳的可行函数，则编译器将生成一条错误消息，该消息可能会使用诸如“ambiguous（二义性）”这样的词语。

然而，有时候，即使两个函数都完全匹配，仍可完成重载解析。首先，指向非 const 数据的指针和引用优先与非 const 指针和引用参数匹配。也就是说，在 recycle( )示例中，如果只定义了函数#3 和#4 是完全匹配的，则将选择#3，因为 ink 没有被声明为 const。然而，const 和非 const 之间的区别只适用于指针和引用指向的数据。也就是说，如果只定义了#1 和#2，则将出现二义性错误。

一个完全匹配优于另一个的另一种情况是，其中一个是非模板函数，而另一个不是。在这种情况下，非模板函数将优先于模板函数（包括显式具体化）。

如果两个完全匹配的函数都是模板函数，则较具体的模板函数优先。例如，这意味着显式具体化将优于使用模板隐式生成的具体化：

```
struct blot {int a; char b[10];};
template <class Type> void recycle (Type t); // template
template <> void recycle<blot> (blot & t);   // specialization for blot
...
blot ink = {25, "spots"};
...
recycle(ink);  // use specialization
```

术语“最具体（most specialized）”并不一定意味着显式具体化，而是指编译器推断使用哪种类型时执行的转换最少。例如，请看下面两个模板：

```
template <class Type> void recycle (Type t);    // #1
template <class Type> void recycle (Type * t);  // #2
```

假设包含这些模板的程序也包含如下代码：

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink);  // address of a structure
```

recycle(&ink)调用与#1 模板匹配，匹配时将 Type 解释为 blot *。recycle（&ink）函数调用也与#2 模板匹配，这次 Type 被解释为 ink。因此将两个隐式实例——recycle<blot *>(blot *)和 recycle <blot>(blot *)发送到可行函数池中。

在这两个模板函数中，recycle<blot *>(blot *)被认为是更具体的，因为在生成过程中，它需要进行的转换更少。也就是说，#2 模板已经显式指出，函数参数是指向 Type 的指针，因此可以直接用 blot 标识 Type；而#1 模板将 Type 作为函数参数，因此 Type 必须被解释为指向 blot 的指针。也就是说，在#2 模板中，Type 已经被具体化为指针，因此说它“更具体”。

用于找出最具体的模板的规则被称为函数模板的部分排序规则（partial ordering rules）。和显式实例一样，这也是 C++98 新增的特性。

2. 部分排序规则示例

我们先看一个完整的程序，它使用部分排序规则来确定要使用哪个模板定义。程序清单 8.14 有两个用来显示数组内容的模板定义。第一个定义（模板 A）假设作为参数传递的数组中包含了要显示的数据；第二个定义（模板 B）假设数组元素为指针，指向要显示的数据。

程序清单 8.14　temptempover.cpp

```
// tempover.cpp -- template overloading
#include <iostream>

template <typename T>              // template A
void ShowArray(T arr[], int n);

template <typename T>              // template B
```

```
void ShowArray(T * arr[], int n);

struct debts
{
    char name[50];
    double amount;
};

int main()
{
    using namespace std;
    int things[6] = {13, 31, 103, 301, 310, 130};
    struct debts mr_E[3] =
    {
        {"Ima Wolfe", 2400.0},
        {"Ura Foxe", 1300.0},
        {"Iby Stout", 1800.0}
    };
    double * pd[3];

// set pointers to the amount members of the structures in mr_E
    for (int i = 0; i < 3; i++)
        pd[i] = &mr_E[i].amount;

    cout << "Listing Mr. E's counts of things:\n";
// things is an array of int
    ShowArray(things, 6);  // uses template A
    cout << "Listing Mr. E's debts:\n";
// pd is an array of pointers to double
    ShowArray(pd, 3);      // uses template B (more specialized)
    return 0;
}

template <typename T>
void ShowArray(T arr[], int n)
{
    using namespace std;
    cout << "template A\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << ' ';
    cout << endl;
}

template <typename T>
void ShowArray(T * arr[], int n)
{
    using namespace std;
    cout << "template B\n";
    for (int i = 0; i < n; i++)
        cout << *arr[i] << ' ';
    cout << endl;
}
```

请看下面的函数调用：

```
ShowArray(things, 6);
```

标识符 things 是一个 int 数组的名称，因此与下面的模板匹配：

```
template <typename T>              // template A
void ShowArray(T arr[], int n);
```

其中 T 被替换为 int 类型。

接下来，请看下面的函数调用：

```
ShowArray(pd, 3);
```

其中 pd 是一个 double *数组的名称。这与模板 A 匹配：

```
template <typename T>              // template A
void ShowArray(T arr[], int n);
```

其中，T 被替换为类型 double *。在这种情况下，模板函数将显示 pd 数组的内容，即 3 个地址。该函数调用也与模板 B 匹配：

```
template <typename T>              // template B
void ShowArray(T * arr[], int n);
```

在这里，T 被替换为类型 double，而函数将显示被解除引用的元素*arr[i]，即数组内容指向的 double 值。在这两个模板中，模板 B 更具体，因为它做了特定的假设——数组内容是指针，因此被使用。

下面是程序清单 8.14 中程序的输出：

```
Listing Mr. E's counts of things:
template A
13 31 103 301 310 130
Listing Mr. E's debts:
template B
2400 1300 1800
```

如果将模板 B 从程序中删除，则编译器将使用模板 A 来显示 pd 的内容，因此显示的将是地址，而不是值。请试试看。

简而言之，重载解析将寻找最匹配的函数。如果只存在一个这样的函数，则选择它；如果存在多个这样的函数，但其中只有一个是非模板函数，则选择该函数；如果存在多个适合的函数，且它们都为模板函数，但其中有一个函数比其他函数更具体，则选择该函数。如果有多个同样合适的非模板函数或模板函数，但没有一个函数比其他函数更具体，则函数调用将是不确定的，因此是错误的；当然，如果不存在匹配的函数，则也是错误。

3. 自己选择

在有些情况下，可通过编写合适的函数调用，引导编译器做出您希望的选择。请看程序清单 8.15，该程序将模板函数定义放在文件开头，从而无需提供模板原型。与常规函数一样，通过在使用函数前提供模板函数定义，它让它也充当原型。

程序清单 8.15　choices.cpp

```
// choices.cpp -- choosing a template
#include <iostream>

template<class T>  // or template <typename T>
T lesser(T a, T b)          // #1
{
    return a < b ? a : b;
}

int lesser (int a, int b)  // #2
```

```
{
    a = a < 0 ? -a : a;
    b = b < 0 ? -b : b;
    return a < b ? a : b;
}

int main()
{
    using namespace std;
    int m = 20;
    int n = -30;
    double x = 15.5;
    double y = 25.9;

    cout << lesser(m, n) << endl;        // use #2
    cout << lesser(x, y) << endl;        // use #1 with double
    cout << lesser<>(m, n) << endl;      // use #1 with int
    cout << lesser<int>(x, y)  << endl; // use #1 with int

    return 0;
}
```

最后的函数调用将 double 转换为 int，有些编译器会针对这一点发出警告。

该程序的输出如下：

```
20
15.5
-30
15
```

程序清单 8.15 提供了一个模板和一个标准函数，其中模板返回两个值中较小的一个，而标准函数返回两个值中绝对值较小的那个。如果函数定义是在使用函数前提供的，它将充当函数原型，因此这个示例无需提供原型。请看下面的语句：

```
cout << lesser(m, n) << endl;        // use #2
```

这个函数调用与模板函数和非模板函数都匹配，因此选择非模板函数，返回 20。

接下来，下述语句中的函数调用与模板匹配（T 为 double）：

```
cout << lesser(x, y) << endl;        // use #1 with double
```

现在来看下面的语句：

```
cout << lesser<>(m, n) << endl;      // use #1 with int
```

lesser<>(m, n)中的<>指出，编译器应选择模板函数，而不是非模板函数；编译器注意到实参的类型为 int，因此使用 int 替代 T 对模板进行实例化。

最后，请看下面的语句：

```
cout << lesser<int>(x, y)  << endl; // use #1 with int
```

这条语句要求进行显式实例化（使用 int 替代 T），将使用显式实例化得到的函数。x 和 y 的值将被强制转换为 int，该函数返回一个 int 值，这就是程序显示 15 而不是 15.5 的原因所在。

4. 多个参数的函数

将有多个参数的函数调用与有多个参数的原型进行匹配时，情况将非常复杂。编译器必须考虑所有参数的匹配情况。如果找到比其他可行函数都合适的函数，则选择该函数。一个函数要比其他函数都合适，其所有参数的匹配程度都必须不比其他函数差，同时至少有一个参数的匹配程度比其他函数都高。

本书并不是要解释复杂示例的匹配过程，这些规则只是为了让任何一组函数原型和模板都存在确定的结果。

8.5.6　模板函数的发展

在 C++发展的早期，大多数人都没有想到模板函数和模板类会有这么强大而有用，它们甚至没有就这个主题发挥想象力。但聪明而专注的程序员挑战模板技术的极限，阐述了各种可能性。根据熟悉模板的程序员提供的反馈，C++98 标准做了相应的修改，并添加了标准模板库。从此以后，模板程序员在不断探索各种可能性，并消除模板的局限性。C++11 标准根据这些程序员的反馈做了相应的修改。下面介绍一些相关的问题及其解决方案。

1. 是什么类型

在 C++98 中，编写模板函数时，一个问题是并非总能知道应在声明中使用哪种类型。请看下面这个不完整的示例：

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    ?type? xpy = x + y;
    ...
}
```

xpy 应为什么类型呢？由于不知道 ft()将如何使用，因此无法预先知道这一点。正确的类型可能是 T1、T2 或其他类型。例如，T1 可能是 double，而 T2 可能是 int，在这种情况下，两个变量的和将为 double 类型。T1 可能是 short，而 T2 可能是 int，在这种情况下，两个变量的和为 int 类型。T1 还可能是 short，而 T2 可能是 char，在这种情况下，加法运算将导致自动整型提升，因此结果类型为 int。另外，结构和类可能重载运算符+，这导致问题更加复杂。因此，在 C++98 中，没有办法声明 xpy 的类型。

2. 关键字 decltype（C++11）

C++11 新增的关键字 decltype 提供了解决方案。可这样使用该关键字：

```
int x;
decltype(x) y;   // make y the same type as x
```

给 decltype 提供的参数可以是表达式，因此在前面的模板函数 ft()中，可使用下面的代码：

```
decltype(x + y) xpy;  // make xpy the same type as x + y
xpy = x + y;
```

另一种方法是，将这两条语句合而为一：

```
decltype(x + y) xpy = x + y;
```

因此，可以这样修复前面的模板函数 ft()：

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    decltype(x + y) xpy = x + y;
    ...
}
```

decltype 比这些示例演示的要复杂些。为确定类型，编译器必须遍历一个核对表。假设有如下声明：

```
decltype(expression) var;
```

则核对表的简化版如下：

第一步：如果 expression 是一个没有用括号括起的标识符，则 var 的类型与该标识符的类型相同，包括 const 等限定符：

```
double x = 5.5;
double y = 7.9;
double &rx = x;
```

```
const double * pd;
decltype(x) w;        // w is type double
decltype(rx) u = y;   // u is type double &
decltype(pd) v;       // v is type const double *
```

第二步：如果 expression 是一个函数调用，则 var 的类型与函数的返回类型相同：

```
long indeed(int);
decltype (indeed(3)) m; // m is type int
```

注意：并不会实际调用函数。编译器通过查看函数的原型来获悉返回类型，而无需实际调用函数。

第三步：如果 expression 是一个左值，则 var 为指向其类型的引用。这好像意味着前面的 w 应为引用类型，因为 x 是一个左值。但别忘了，这种情况已经在第一步处理过了。要进入第三步，expression 不能是未用括号括起的标识符。那么，expression 是什么时将进入第三步呢？一种显而易见的情况是，expression 是用括号括起的标识符：

```
double xx = 4.4;
decltype ((xx)) r2 = xx;  // r2 is double &
decltype(xx) w = xx;      // w is double (Stage 1 match)
```

顺便说一句，括号并不会改变表达式的值和左值性。例如，下面两条语句等效：

```
xx = 98.6;
(xx) = 98.6;  // () don't affect use of xx
```

第四步：如果前面的条件都不满足，则 var 的类型与 expression 的类型相同：

```
int j = 3;
int &k = j
int &n = j;
decltype(j+6) i1;   // i1 type int
decltype(100L) i2;  // i2 type long
decltype(k+n) i3;   // i3 type int;
```

请注意，虽然 k 和 n 都是引用，但表达式 k+n 不是引用；它是两个 int 的和，因此类型为 int。

如果需要多次声明，可结合使用 typedef 和 decltype：

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    typedef decltype(x + y) xytype;
    xytype xpy = x + y;
    xytype arr[10];
    xytype & rxy = arr[2];   // rxy a reference
    ...
}
```

3. 另一种函数声明语法（C++11 后置返回类型）

有一个相关的问题是 decltype 本身无法解决的。请看下面这个不完整的模板函数：

```
template<class T1, class T2>
?type? gt(T1 x, T2 y)
{
    ...
    return x + y;
}
```

同样，无法预先知道将 x 和 y 相加得到的类型。好像可以将返回类型设置为 decltype (x + y)，但不幸的是，此时还未声明参数 x 和 y，它们不在作用域内（编译器看不到它们，也无法使用它们）。必须在声明参数后使用 decltype。为此，C++新增了一种声明和定义函数的语法。下面使用内置类型来说明这种语法的工作原理。对于下面的原型：

```
double h(int x, float y);
```

使用新增的语法可编写成这样：

```
auto h(int x, float y) -> double;
```

这将返回类型移到了参数声明后面。->double 被称为后置返回类型（trailing return type）。其中 auto 是一个占位符，表示后置返回类型提供的类型，这是 C++11 给 auto 新增的一种角色。这种语法也可用于函数定义：

```
auto h(int x, float y) -> double
{/* function body */};
```

通过结合使用这种语法和 decltype，便可给 gt()指定返回类型，如下所示：

```
template<class T1, class T2>
auto gt(T1 x, T2 y) -> decltype(x + y)
{
    ...
    return x + y;
}
```

现在，decltype 在参数声明后面，因此 x 和 y 位于作用域内，可以使用它们。

8.6 总结

C++扩展了 C 语言的函数功能。通过将 inline 关键字用于函数定义，并在首次调用该函数前提供其函数定义，可以使得 C++编译器将该函数视为内联函数。也就是说，编译器不是让程序跳到独立的代码段，以执行函数，而是用相应的代码替换函数调用。只有在函数很短时才能采用内联方式。

引用变量是一种伪装指针，它允许为变量创建别名（另一个名称）。引用变量主要被用作处理结构和类对象的函数的参数。通常，被声明为特定类型引用的标识符只能指向这种类型的数据；然而，如果一个类（如 ofstream）是从另一个类（如 ostream）派生出来的，则基类引用可以指向派生类对象。

C++原型让您能够定义参数的默认值。如果函数调用省略了相应的参数，则程序将使用默认值；如果函数调用提供了参数值，则程序将使用这个值（而不是默认值）。只能在参数列表中从右到左提供默认参数。因此，如果为某个参数提供了默认值，则必须为该参数右边所有的参数提供默认值。

函数的特征标是其参数列表。程序员可以定义两个同名函数，只要其特征标不同。这被称为函数多态或函数重载。通常，通过重载函数来为不同的数据类型提供相同的服务。

函数模板自动完成重载函数的过程。只需使用泛型和具体算法来定义函数，编译器将为程序中使用的特定参数类型生成正确的函数定义。

8.7 复习题

1．哪种函数适合定义为内联函数？

2．假设 song( )函数的原型如下：

```
void song(const char * name, int times);
```

a．如何修改原型，使 times 的默认值为 1？

b．函数定义需要做哪些修改？

c．能否为 name 提供默认值“O. My Papa”？

3．编写 iquote( )的重载版本——显示其用双引号括起的参数。编写 3 个版本：一个用于 int 参数，一个用于 double 参数，另一个用于 string 参数。

4．下面是一个结构模板：

```
struct box
{
    char maker[40];
    float height;
    float width;
    float length;
    float volume;
};
```

a．请编写一个函数，它将 box 结构的引用作为形参，并显示每个成员的值。

b．请编写一个函数，它将 box 结构的引用作为形参，并将 volume 成员设置为其他 3 边的乘积。

5．为让函数 fill()和 show()使用引用参数，需要对程序清单 7.15 做哪些修改？

6．指出下面每个目标是否可以使用默认参数或函数重载完成，或者这两种方法都无法完成，并提供合适的原型。

a．mass(density, volume)返回密度为 density、体积为 volume 的物体的质量，而 mass(denstity)返回密度为 density、体积为 1.0 立方米的物体的质量。这些值的类型都为 double。

b．repeat(10, “I'm OK”)将指定的字符串显示 10 次，而 repeat(“But you're kind of stupid”)将指定的字符串显示 5 次。

c．average(3, 6)返回两个 int 参数的平均值（int 类型），而 average(3.0, 6.0)返回两个 double 值的平均值（double 类型）。

d．mangle(“I'm glad to meet you”)根据是将值赋给 char 变量还是 char*变量，分别返回字符 I 和指向字符串“I'm mad to gleet you”的指针。

7．编写返回两个参数中较大值的函数模板。

8．给定复习题 6 的模板和复习题 4 的 box 结构，提供一个模板具体化，它接受两个 box 参数，并返回体积较大的一个。

9．在下述代码（假定这些代码是一个完整程序的一部分）中，v1、v2、v3、v4 和 v5 分别是哪种类型？

```
int g(int x);
...
float m = 5.5f;
float & rm = m;
decltype(m) v1 = m;
decltype(rm) v2 = m;
decltype((m)) v3 = m;
decltype (g(100)) v4;
decltype (2.0 * m) v5;
```

8.8 编程练习

1．编写通常接受一个参数（字符串的地址），并打印该字符串的函数。然而，如果提供了第二个参数（int 类型），且该参数不为 0，则该函数打印字符串的次数将为该函数被调用的次数（注意，字符串的打印次数不等于第二个参数的值，而等于函数被调用的次数）。是的，这是一个非常可笑的函数，但它让您能够使用本章介绍的一些技术。在一个简单的程序中使用该函数，以演示该函数是如何工作的。

2．CandyBar 结构包含 3 个成员。第一个成员存储 candy bar 的品牌名称；第二个成员存储 candy bar 的重量（可能有小数）；第三个成员存储 candy bar 的热量（整数）。请编写一个程序，它使用一个这样的函数，即将 CandyBar 的引用、char 指针、double 和 int 作为参数，并用最后 3 个值设置相应的结构成员。最后 3 个参数的默认值分别为“Millennium Munch”、2.85 和 350。另外，该程序还包含一个以 CandyBar 的引用为参数，并显示结构内容的函数。请尽可能使用 const。

3．编写一个函数，它接受一个指向 string 对象的引用作为参数，并将该 string 对象的内容转换为大写，

为此可使用表 6.4 描述的函数 toupper( )。然后编写一个程序，它通过使用一个循环让您能够用不同的输入来测试这个函数，该程序的运行情况如下：

```
Enter a string (q to quit): go away
GO AWAY
Next string (q to quit): good grief!
GOOD GRIEF!
Next string (q to quit): q
Bye.
```

4．下面是一个程序框架：

```
#include <iostream>
using namespace std;
#include <cstring>      // for strlen(), strcpy()
struct stringy {
    char * str;         // points to a string
    int ct;             // length of string (not counting '\0')
    };

// prototypes for set(), show(), and show() go here
int main()
{
    stringy beany;
    char testing[] = "Reality isn't what it used to be.";

    set(beany, testing);    // first argument is a reference,
                // allocates space to hold copy of testing,
                // sets str member of beany to point to the
                // new block, copies testing to new block,
                // and sets ct member of beany
    show(beany);        // prints member string once
    show(beany, 2);     // prints member string twice
    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);      // prints testing string once
    show(testing, 3); // prints testing string thrice
    show("Done!");
    return 0;
}
```

请提供其中描述的函数和原型，从而完成该程序。注意，应有两个 show( )函数，每个都使用默认参数。请尽可能使用 cosnt 参数。set( )使用 new 分配足够的空间来存储指定的字符串。这里使用的技术与设计和实现类时使用的相似。（可能还必须修改头文件的名称，删除 using 编译指令，这取决于所用的编译器。）

5．编写模板函数 max5( )，它将一个包含 5 个 T 类型元素的数组作为参数，并返回数组中最大的元素（由于长度固定，因此可以在循环中使用硬编码，而不必通过参数来传递）。在一个程序中使用该函数，将 T 替换为一个包含 5 个 int 值的数组和一个包含 5 个 dowble 值的数组，以测试该函数。

6．编写模板函数 maxn( )，它将由一个 T 类型元素组成的数组和一个表示数组元素数目的整数作为参数，并返回数组中最大的元素。在程序对它进行测试，该程序使用一个包含 6 个 int 元素的数组和一个包含 4 个 double 元素的数组来调用该函数。程序还包含一个具体化，它将 char 指针数组和数组中的指针数量作为参数，并返回最长的字符串的地址。如果有多个这样的字符串，则返回其中第一个字符串的地址。使用由 5 个字符串指针组成的数组来测试该具体化。

7．修改程序清单 8.14，使其使用两个名为 SumArray()的模板函数来返回数组元素的总和，而不是显示数组的内容。程序应显示 thing 的总和以及所有 debt 的总和。

第 9 章　内存模型和名称空间

本章内容包括：

- 单独编译。
- 存储持续性、作用域和链接性。
- 定位（placement）new 运算符。
- 名称空间。

C++为在内存中存储数据方面提供了多种选择。可以选择数据保留在内存中的时间长度（存储持续性）以及程序的哪一部分可以访问数据（作用域和链接）等。可以使用 new 来动态地分配内存，而定位 new 运算符提供了这种技术的一种变种。C++名称空间是另一种控制访问权的方式。通常，大型程序都由多个源代码文件组成，这些文件可能共享一些数据。这样的程序涉及到程序文件的单独编译，本章将首先介绍这个主题。

9.1　单独编译

和 C 语言一样，C++也允许甚至鼓励程序员将组件函数放在独立的文件中。第 1 章介绍过，可以单独编译这些文件，然后将它们链接成可执行的程序。（通常，C++编译器既编译程序，也管理链接器。）如果只修改了一个文件，则可以只重新编译该文件，然后将它与其他文件的编译版本链接。这使得大程序的管理更便捷。另外，大多数 C++环境都提供了其他工具来帮助管理。例如，UNIX 和 Linux 系统都具有 make 程序，可以跟踪程序依赖的文件以及这些文件的最后修改时间。运行 make 时，如果它检测到上次编译后修改了源文件，make 将记住重新构建程序所需的步骤。大多数集成开发环境（包括 Embarcadero C++ Builder、Microsoft Visual C++、Apple Xcode 和 Freescale CodeWarrior）都在 Project 菜单中提供了类似的工具。

现在看一个简单的示例。我们不是要从中了解编译的细节（这取决于实现），而是要重点介绍更通用的方面，如设计。

例如，假设程序员决定分解程序清单 7.12 中的程序，将支持函数放在一个独立的文件中。清单 7.12 将直角坐标转换为极坐标，然后显示结果。不能简单地以 main( )之后的虚线为界，将原来的文件分为两个。问题在于，main( )和其他两个函数使用了同一个结构声明，因此两个文件都应包含该声明。简单地将它们输入进去无疑是自找麻烦。即使正确地复制了结构声明，如果以后要作修改，则必须记住对这两组声明都进行修改。简而言之，将一个程序放在多个文件中将引出新的问题。

谁希望出现更多的问题呢？C 和 C++的开发人员都不希望，因此他们提供了#include 来处理这种情况。与其将结构声明加入到每一个文件中，不如将其放在头文件中，然后在每一个源代码文件中包含该头文件。这样，要修改结构声明时，只需在头文件中做一次改动即可。另外，也可以将函数原型放在头文件中。因此，可以将原来的程序分成三部分。

- 头文件：包含结构声明和使用这些结构的函数的原型。

- 源代码文件：包含与结构有关的函数的代码。
- 源代码文件：包含调用与结构相关的函数的代码。

这是一种非常有用的组织程序的策略。例如，如果编写另一个程序时，也需要使用这些函数，则只需包含头文件，并将函数文件添加到项目列表或 make 列表中即可。另外，这种组织方式也与 OOP 方法一致。一个文件（头文件）包含了用户定义类型的定义；另一个文件包含操纵用户定义类型的函数的代码。这两个文件组成了一个软件包，可用于各种程序中。

请不要将函数定义或变量声明放到头文件中。这样做对于简单的情况可能是可行的，但通常会引来麻烦。例如，如果在头文件包含一个函数定义，然后在其他两个文件（属于同一个程序）中包含该头文件，则同一个程序中将包含同一个函数的两个定义，除非函数是内联的，否则这将出错。下面列出了头文件中常包含的内容。

- 函数原型。
- 使用#define 或 const 定义的符号常量。
- 结构声明。
- 类声明。
- 模板声明。
- 内联函数。

将结构声明放在头文件中是可以的，因为它们不创建变量，而只是在源代码文件中声明结构变量时，告诉编译器如何创建该结构变量。同样，模板声明不是将被编译的代码，它们指示编译器如何生成与源代码中的函数调用相匹配的函数定义。被声明为 const 的数据和内联函数有特殊的链接属性（稍后将介绍），因此可以将其放在头文件中，而不会引起问题。

程序清单 9.1、程序清单 9.2 和程序清单 9.3 是将程序清单 7.12 分成几个独立部分后得到的结果。注意，在包含头文件时，我们使用“coordin.h”，而不是<coodin.h>。如果文件名包含在尖括号中，则 C++编译器将在存储标准头文件的主机系统的文件系统中查找；但如果文件名包含在双引号中，则编译器将首先查找当前的工作目录或源代码目录（或其他目录，这取决于编译器）。如果没有在那里找到头文件，则将在标准位置查找。因此在包含自己的头文件时，应使用引号而不是尖括号。

图 9.1 简要地说明了在 UNIX 系统中将该程序组合起来的步骤。注意，只需执行编译命令 CC 即可，其他步骤将自动完成。g++和 gpp 命令行编译器以及 Borland C++命令行编译器（bcc32.exe）的行为类似。Apple Xcode、Embarcadero C++ Builderr 和 Microsoft Visual C++基本上执行同样的步骤，但正如第 1 章介绍的，启动这个过程的方式不同——使用能够创建项目并将其与源代码文件关联起来的菜单。注意，只需将源代码文件加入到项目中，而不用加入头文件。这是因为#include 指令管理头文件。另外，不要使用#include 来包含源代码文件，这样做将导致多重声明。

警告：在 IDE 中，不要将头文件加入到项目列表中，也不要在源代码文件中使用#include 来包含其他源代码文件。

程序清单 9.1　coordin.h

```
// coordin.h -- structure templates and function prototypes
// structure templates
#ifndef COORDIN_H_
#define COORDIN_H_

struct polar
{
    double distance;    // distance from origin
    double angle;        // direction from origin
};
```

```
struct rect
{
    double x;        // horizontal distance from origin
    double y;        // vertical distance from origin
};

// prototypes
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

#endif
```

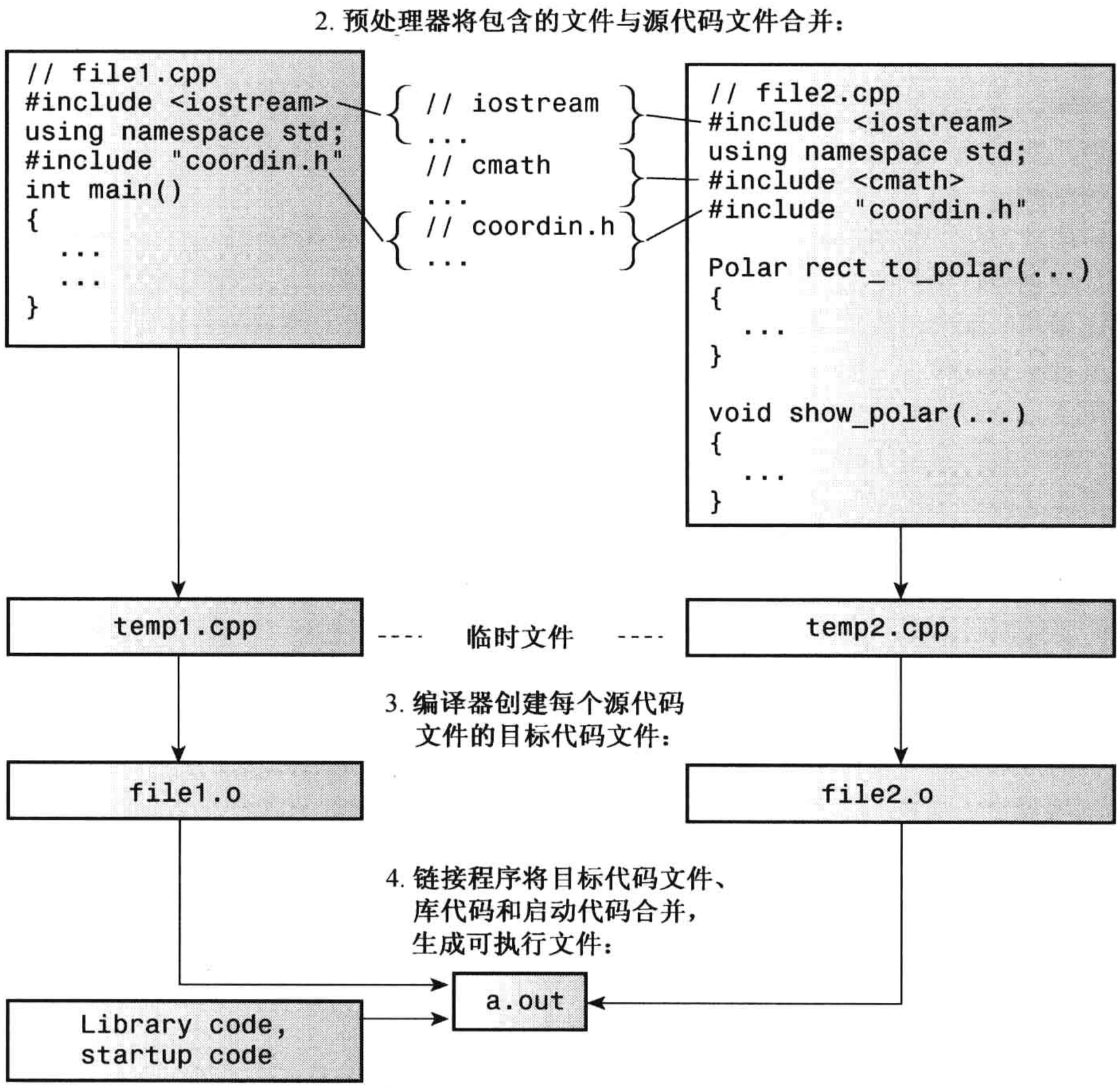


图 9.1　在 UNIX 系统中编译由多个文件组成的 C++程序

头文件管理

在同一个文件中只能将同一个头文件包含一次。记住这个规则很容易，但很可能在不知情的情况下将头文件包含多次。例如，可能使用包含了另外一个头文件的头文件。有一种标准的 C/C++技术可以避免多次包含同一个头文件。它是基于预处理器编译指令#ifndef（即 if not defined）的。下面的代码片段意味着仅当以前没有使用预处理器编译指令#define 定义名称 COORDIN_H_时，才处理#ifndef 和#endif 之间的语句：

```
#ifndef COORDIN_H_
...
#endif
```

通常，使用#define 语句来创建符号常量，如下所示：

```
#define MAXIMUM 4096
```

但只要将#define 用于名称，就足以完成该名称的定义，如下所示：

```
#define COORDIN_H_
```

程序清单 9.1 使用这种技术是为了将文件内容包含在#ifndef 中：

```
#ifndef COORDIN_H_
#define COORDIN_H_
// place include file contents here
#endif
```

编译器首次遇到该文件时，名称 COORDIN_H_没有定义（我们根据 include 文件名来选择名称，并加上一些下划线，以创建一个在其他地方不太可能被定义的名称）。在这种情况下，编译器将查看#ifndef 和#endif 之间的内容（这正是我们希望的），并读取定义 COORDIN_H_的一行。如果在同一个文件中遇到其他包含 coordin.h 的代码，编译器将知道 COORDIN_H_已经被定义了，从而跳到#endfi 后面的一行上。注意，这种方法并不能防止编译器将文件包含两次，而只是让它忽略除第一次包含之外的所有内容。大多数标准 C 和 C++ 头文件都使用这种防护（guarding）方案。否则，可能在一个文件中定义同一个结构两次，这将导致编译错误。

程序清单 9.2　file1.cpp

```
// file1.cpp -- example of a three-file program
#include <iostream>
#include "coordin.h" // structure templates, function prototypes
using namespace std;
int main()
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)  // slick use of cin
    {
        pplace = rect_to_polar(rplace);
        show_polar(pplace);
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Bye!\n";
    return 0;
}
```

程序清单 9.3　file2.cpp

```
// file2.cpp -- contains functions called in file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // structure templates, function prototypes

// convert rectangular to polar coordinates
polar rect_to_polar(rect xypos)
{
    using namespace std;
    polar answer;

    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
```

```
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;      // returns a polar structure
}

// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

将这两个源代码文件和新的头文件一起进行编译和链接，将生成一个可执行程序。下面是该程序的运行情况：

```
Enter the x and y values: 120 80
distance = 144.222, angle = 33.6901 degrees
Next two numbers (q to quit): 120 50
distance = 130, angle = 22.6199 degrees
Next two numbers (q to quit): q
```

顺便说一句，虽然我们讨论的是根据文件进行单独编译，但为保持通用性，C++标准使用了术语翻译单元（translation unit），而不是文件；文件并不是计算机组织信息时的唯一方式。出于简化的目的，本书使用术语文件，您可将其解释为翻译单元。

多个库的链接

C++标准允许每个编译器设计人员以他认为合适的方式实现名称修饰（参见第 8 章的旁注“什么是名称修饰”），因此由不同编译器创建的二进制模块（对象代码文件）很可能无法正确地链接。也就是说，两个编译器将为同一个函数生成不同的修饰名称。名称的不同将使链接器无法将一个编译器生成的函数调用与另一个编译器生成的函数定义匹配。在链接编译模块时，请确保所有对象文件或库都是由同一个编译器生成的。如果有源代码，通常可以用自己的编译器重新编译源代码来消除链接错误。

9.2 存储持续性、作用域和链接性

介绍过多文件程序后，接下来扩展第 4 章对内存方案的讨论，即存储类别如何影响信息在文件间的共享。现在读者阅读第 4 章已经有一段时间了，因此先复习一下有关内存的知识。C++使用三种（在 C++11 中是四种）不同的方案来存储数据，这些方案的区别就在于数据保留在内存中的时间。

- 自动存储持续性：在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。C++有两种存储持续性为自动的变量。
- 静态存储持续性：在函数定义外定义的变量和使用关键字 static 定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。C++有 3 种存储持续性为静态的变量。
- 线程存储持续性（C++11）：当前，多核处理器很常见，这些 CPU 可同时处理多个执行任务。这让程序能够将计算放在可并行处理的不同线程中。如果变量是使用关键字 thread_local 声明的，则其生命周期与所属的线程一样长。本书不探讨并行编程。
- 动态存储持续性：用 new 运算符分配的内存将一直存在，直到使用 delete 运算符将其释放或程序结束为止。这种内存的存储持续性为动态，有时被称为自由存储（free store）或堆（heap）。

下面介绍其他内容，包括关于各种变量何时在作用域内或可见（可被程序使用）以及链接性的细节。链接性决定了哪些信息可在文件间共享。

9.2.1　作用域和链接

作用域（scope）描述了名称在文件（翻译单元）的多大范围内可见。例如，函数中定义的变量可在该函数中使用，但不能在其他函数中使用；而在文件中的函数定义之前定义的变量则可在所有函数中使用。链接性（linkage）描述了名称如何在不同单元间共享。链接性为外部的名称可在文件间共享，链接性为内部的名称只能由一个文件中的函数共享。自动变量的名称没有链接性，因为它们不能共享。

C++变量的作用域有多种。作用域为局部的变量只在定义它的代码块中可用。代码块是由花括号括起的一系列语句。例如函数体就是代码块，但可以在函数体中嵌入其他代码块。作用域为全局（也叫文件作用域）的变量在定义位置到文件结尾之间都可用。自动变量的作用域为局部，静态变量的作用域是全局还是局部取决于它是如何被定义的。在函数原型作用域（function prototype scope）中使用的名称只在包含参数列表的括号内可用（这就是为什么这些名称是什么以及是否出现都不重要的原因）。在类中声明的成员的作用域为整个类（参见第 10 章）。在名称空间中声明的变量的作用域为整个名称空间（由于名称空间已经引入到 C++语言中，因此全局作用域是名称空间作用域的特例）。

C++函数的作用域可以是整个类或整个名称空间（包括全局的），但不能是局部的（因为不能在代码块内定义函数，如果函数的作用域为局部，则只对它自己是可见的，因此不能被其他函数调用。这样的函数将无法运行）。

不同的 C++存储方式是通过存储持续性、作用域和链接性来描述的。下面来看看各种 C++存储方式的这些特征。首先介绍引入名称空间之前的情况，然后看一看名称空间带来的影响。

9.2.2　自动存储持续性

在默认情况下，在函数中声明的函数参数和变量的存储持续性为自动，作用域为局部，没有链接性。也就是说，如果在 main( )中声明了一个名为 texas 的变量，并在函数 oil( )中也声明了一个名为 texas 变量，则创建了两个独立的变量——只有在定义它们的函数中才能使用它们。对 oil( )中的 texas 执行的任何操作都不会影响 main( )中的 texas，反之亦然。另外，当程序开始执行这些变量所属的代码块时，将为其分配内存；当函数结束时，这些变量都将消失（注意，执行到代码块时，将为变量分配内存，但其作用域的起点为其声明位置）。

如果在代码块中定义了变量，则该变量的存在时间和作用域将被限制在该代码块内。例如，假设在 main( )的开头定义了一个名为 teledeli 的变量，然后在 main( )中开始一个新的代码块，并其中定义了一个新的变量 websight，则 teledeli 在内部代码块和外部代码块中都是可见的，而 websight 就只在内部代码块中可见，它的作用域是从定义它的位置到该代码块的结尾：

```
int main()
{
    int teledeli = 5;
    {                           // websight allocated
        cout << "Hello\n";
        int websight = -2;      // websight scope begins
        cout << websight << ' ' << teledeli << endl;
    }                           // websight expires
    cout << teledeli << endl;
    ...
}  // teledeli expires
```

然而，如果将内部代码块中的变量命名为 teledeli，而不是 websight，使得有两个同名的变量（一个位于外部代码块中，另一个位于内部代码块中），情况将如何呢？在这种情况下，程序执行内部代码块中的语句时，将 teledeli 解释为局部代码块变量。我们说，新的定义隐藏了（hide）以前的定义，新定义可见，旧

定义暂时不可见。在程序离开该代码块时，原来的定义又重新可见（参见图 9.2）。

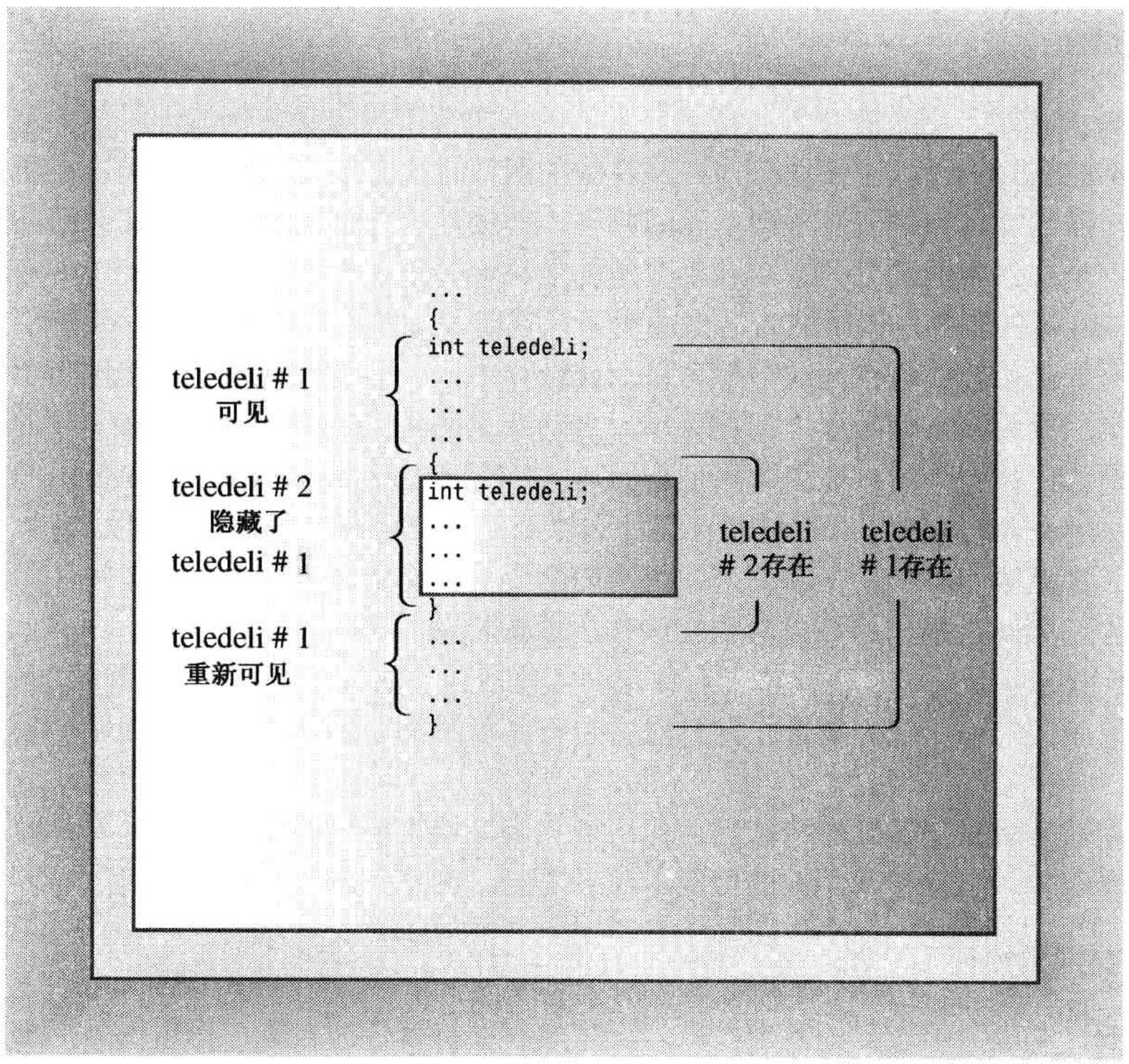


图 9.2 代码块和作用域

程序清单 9.4 表明，自动变量只在包含它们的函数或代码块中可见。

程序清单 9.4 auto.cpp

```
// autoscp.cpp -- illustrating scope of automatic variables
#include <iostream>
void oil(int x);
int main()
{
    using namespace std;

    int texas = 31;
    int year = 2011;
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    oil(texas);
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    return 0;
}
```

```
void oil(int x)
{
    using namespace std;
    int texas = 5;

    cout << "In oil(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In oil(), x = " << x << ", &x = ";
    cout << &x << endl;
    {                                   // start a block
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << endl;
                cout << "In block, x = " << x << ", &x = ";
        cout << &x << endl;
    }                                   // end a block
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << endl;
}
```

下面是该程序的输出：

```
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
In oil(), texas = 5, &texas = 0012FDE4
In oil(), x = 31, &x = 0012FDF4
In block, texas = 113, &texas = 0012FDD8
In block, x = 31, &x = 0012FDF4
Post-block texas = 5, &texas = 0012FDE4
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
```

在程序清单 9.4 中，3 个 texas 变量的地址各不相同，而程序使用当前可见的那个变量，因此将 113 赋给 oil( )中的内部代码块中的 texas，对其他同名变量没有影响。同样，实际的地址值和地址格式随系统而异。

现在总结一下整个过程。执行到 main( )时，程序为 texas 和 year 分配空间，使得这些变量可见。当程序调用 oil( )时，这些变量仍留在内存中，但不可见。为两个新变量（x 和 texas）分配内存，从而使它们可见。在程序执行到 oil( )中的内部代码块时，新的 texas 将不可见，它被一个更新的定义代替。然而，变量 x 仍然可见，这是因为该代码块没有定义 x 变量。当程序流程离开该代码块时，将释放最新的 texas 使用的内存，而第二个 texas 再次可见。当 oil( )函数结束时，texas 和 x 都将过期，而最初的 texas 和 year 再次变得可见。

使用 C++11 中的 auto

在 C++11 中，关键字 auto 用于自动类型推断，这在第 3、7 和 8 章介绍过。但在 C 语言和以前的 C++版本中，auto 的含义截然不同，它用于显式地指出变量为自动存储：

```
int froob(int n)
{
    auto float ford;  // ford has automatic storage
    ...
}
```

由于只能将关键字 auto 用于默认为自动的变量，因此程序员几乎不使用它。它的主要用途是指出当前变量为局部自动变量。

在 C++11 中，这种用法不再合法。制定标准的人不愿引入新关键字，因为这样做可能导致将该关键字用于其他目的的代码非法。考虑到 auto 的老用法很少使用，因此赋予其新含义比引入新关键字是更好的选择。

1. 自动变量的初始化

可以使用任何在声明时其值为已知的表达式来初始化自动变量，下面的示例初始化变量 x、y 和 z：

```
int w;                  // value of w is indeterminate
int x = 5;              // initialized with a numeric literal
int big = INT_MAX - 1;  // initialized with a constant expression
int y = 2 * x;          // use previously determined value of x
cin >> w;
int z = 3 * w;          // use new value of w
```

2. 自动变量和栈

了解典型的 C++编译器如何实现自动变量有助于更深入地了解自动变量。由于自动变量的数目随函数的开始和结束而增减，因此程序必须在运行时对自动变量进行管理。常用的方法是留出一段内存，并将其视为栈，以管理变量的增减。之所以被称为栈，是由于新数据被象征性地放在原有数据的上面（也就是说，在相邻的内存单元中，而不是在同一个内存单元中），当程序使用完后，将其从栈中删除。栈的默认长度取决于实现，但编译器通常提供改变栈长度的选项。程序使用两个指针来跟踪栈，一个指针指向栈底——栈的开始位置，另一个指针指向堆顶——下一个可用内存单元。当函数被调用时，其自动变量将被加入到栈中，栈顶指针指向变量后面的下一个可用的内存单元。函数结束时，栈顶指针被重置为函数被调用前的值，从而释放新变量使用的内存。

栈是 LIFO（后进先出）的，即最后加入到栈中的变量首先被弹出。这种设计简化了参数传递。函数调用将其参数的值放在栈顶，然后重新设置栈顶指针。被调用的函数根据其形参描述来确定每个参数的地址。例如，图 9.3 表明，函数 fib( )被调用时，传递一个 2 字节的 int 和一个 4 字节的 long。这些值被加入到栈中。当 fib( )开始执行时，它将名称 real 和 tell 同这两个值关联起来。当 fib( )结束时，栈顶指针重新指向以前的位置。新值没有被删除，但不再被标记，它们所占据的空间将被下一个将值加入到栈中的函数调用所使用（图 9.3 做了简化，因为函数调用可能传递其他信息，如返回地址）。

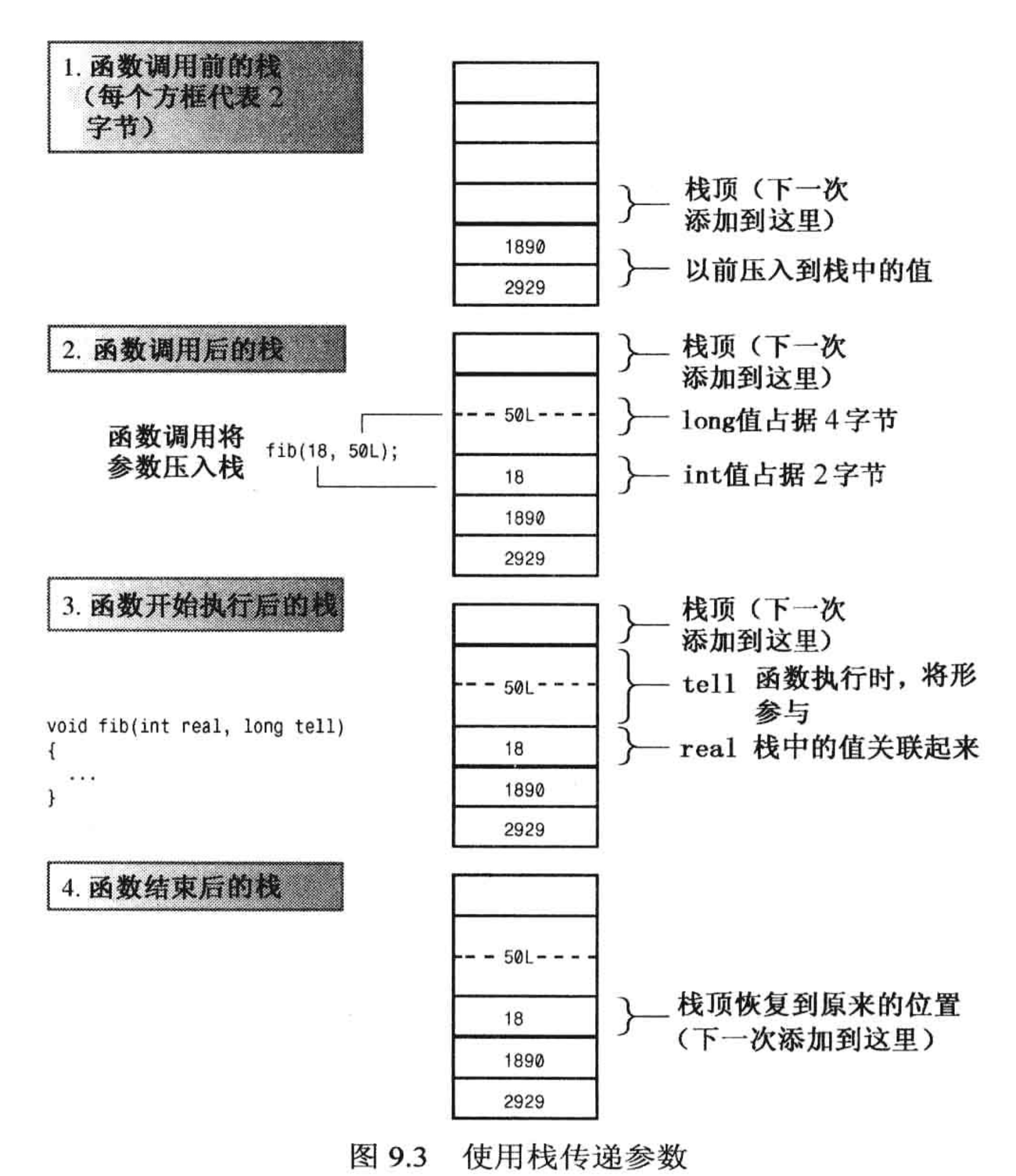


图 9.3　使用栈传递参数

3. 寄存器变量

关键字 register 最初是由 C 语言引入的，它建议编译器使用 CPU 寄存器来存储自动变量：

```
register int count_fast;  // request for a register variable
```

这旨在提高访问变量的速度。

在 C++11 之前，这个关键字在 C++中的用法始终未变，只是随着硬件和编译器变得越来越复杂，这种提示表明变量用得很多，编译器可对其做特殊处理。在 C++11 中，这种提示作用也失去了，关键字 register 只是显式地指出变量是自动的。鉴于关键字 register 只能用于原本就是自动的变量，使用它的唯一原因是，指出程序员想使用一个自动变量，这个变量的名称可能与外部变量相同。这与 auto 以前的用途完全相同。然而，保留关键字 register 的重要原因是，避免使用了该关键字的现有代码非法。

9.2.3 静态持续变量

和 C 语言一样，C++也为静态存储持续性变量提供了 3 种链接性：外部链接性（可在其他文件中访问）、内部链接性（只能在当前文件中访问）和无链接性（只能在当前函数或代码块中访问）。这 3 种链接性都在整个程序执行期间存在，与自动变量相比，它们的寿命更长。由于静态变量的数目在程序运行期间是不变的，因此程序不需要使用特殊的装置（如栈）来管理它们。编译器将分配固定的内存块来存储所有的静态变量，这些变量在整个程序执行期间一直存在。另外，如果没有显式地初始化静态变量，编译器将把它设置为 0。在默认情况下，静态数组和结构将每个元素或成员的所有位都设置为 0。

注意：传统的 K&R C 不允许初始化自动数组和结构，但允许初始化静态数组和结构。ANSI C 和 C++允许对这两种数组和结构进行初始化，但有些旧的 C++翻译器使用与 ANSI C 不完全兼容的 C 编译器。如果使用的是这样的实现，则可能需要使用这 3 种静态存储类型之一，以初始化数组和结构。

下面介绍如何创建这 3 种静态持续变量，然后介绍它们的特点。要想创建链接性为外部的静态持续变量，必须在代码块的外面声明它；要创建链接性为内部的静态持续变量，必须在代码块的外面声明它，并使用 static 限定符；要创建没有链接性的静态持续变量，必须在代码块内声明它，并使用 static 限定符。下面的代码片段说明这 3 种变量：

```
...
int global = 1000;           // static duration, external linkage
static int one_file = 50;    // static duration, internal linkage
int main()
{
...
}
void funct1(int n)
{
    static int count = 0;  // static duration, no linkage
    int llama = 0;
...
}
void funct2(int q)
{
...
}
```

正如前面指出的，所有静态持续变量（上述示例中的 global、one_file 和 count）在整个程序执行期间都存在。在 funct1( )中声明的变量 count 的作用域为局部，没有链接性，这意味着只能在 funct1( )函数中使用它，就像自动变量 llama 一样。然而，与 llama 不同的是，即使在 funct1( )函数没有被执行时，count 也留在内存中。global 和 one_file 的作用域都为整个文件，即在从声明位置到文件结尾的范围内都可以被使用。具体地说，可以在 main( )、funct1( )和 funct2( )中使用它们。由于 one_file 的链接性为内部，因此只能在包含上述代码的文件中使用它；由于 global 的链接性为外部，因此可以在程序的其他文件中使用它。

所有的静态持续变量都有下述初始化特征：未被初始化的静态变量的所有位都被设置为 0。这种变量被称为零初始化的（zero-initialized）。

表 9.1 总结了引入名称空间之前使用的存储特性。下面详细介绍各种静态持续性。

表 9.1 指出了关键字 static 的两种用法，但含义有些不同：用于局部声明，以指出变量是无链接性的静态变量时，static 表示的是存储持续性；而用于代码块外的声明时，static 表示内部链接性，而变量已经是静态持续性了。有人称之为关键字重载，即关键字的含义取决于上下文。

表 9.1　　5 种变量储存方式

存储描述	持续性	作用域	链接性	如何声明
自动	自动	代码块	无	在代码块中
寄存器	自动	代码块	无	在代码块中，使用关键字 register
静态，无链接性	静态	代码块	无	在代码块中，使用关键字 static
静态，外部链接性	静态	文件	外部	不在任何函数内
静态，内部链接性	静态	文件	内部	不在任何函数内，使用关键字 static

静态变量的初始化

除默认的零初始化外，还可对静态变量进行常量表达式初始化和动态初始化。您可能猜到了，零初始化意味着将变量设置为零。对于标量类型，零将被强制转换为合适的类型。例如，在 C++代码中，空指针用 0 表示，但内部可能采用非零表示，因此指针变量将被初始化相应的内部表示。结构成员被零初始化，且填充位都被设置为零。

零初始化和常量表达式初始化被统称为静态初始化，这意味着在编译器处理文件（翻译单元）时初始化变量。动态初始化意味着变量将在编译后初始化。

那么初始化形式由什么因素决定呢？首先，所有静态变量都被零初始化，而不管程序员是否显式地初始化了它。接下来，如果使用常量表达式初始化了变量，且编译器仅根据文件内容（包括被包含的头文件）就可计算表达式，编译器将执行常量表达式初始化。必要时，编译器将执行简单计算。如果没有足够的信息，变量将被动态初始化。请看下面的代码：

```
#include <cmath>
int x;                              // zero-initialization
int y = 5;                          // constant-expression initialization
long z = 13 * 13;                   // constant-expression initialization
const double pi = 4.0 * atan(1.0);  // dynamic initialization
```

首先，x、y、z 和 pi 被零初始化。然后，编译器计算常量表达式，并将 y 和 z 分别初始化为 5 和 169。但要初始化 pi，必须调用函数 atan()，这需要等到该函数被链接且程序执行时。

常量表达式并非只能是使用字面常量的算术表达式。例如，它还可使用 sizeof 运算符：

```
int enough = 2 * sizeof (long) + 1;  // constant expression initialization
```

C++11 新增了关键字 constexpr，这增加了创建常量表达式的方式。但本书不会更详细地介绍 C++11 新增的这项新功能。

9.2.4　静态持续性、外部链接性

链接性为外部的变量通常简称为外部变量，它们的存储持续性为静态，作用域为整个文件。外部变量是在函数外部定义的，因此对所有函数而言都是外部的。例如，可以在 main( )前面或头文件中定义它们。可以在文件中位于外部变量定义后面的任何函数中使用它，因此外部变量也称全局变量（相对于局部的自动变量）。

1. 单定义规则

一方面，在每个使用外部变量的文件中，都必须声明它；另一方面，C++有“单定义规则”（One Definition

Rule，ODR），该规则指出，变量只能有一次定义。为满足这种需求，C++提供了两种变量声明。一种是定义声明（defining declaration）或简称为定义（definition），它给变量分配存储空间；另一种是引用声明（referencing declaration）或简称为声明（declaration），它不给变量分配存储空间，因为它引用已有的变量。

引用声明使用关键字 extern，且不进行初始化；否则，声明为定义，导致分配存储空间：

```
double up;              // definition, up is 0
extern int blem;        // blem defined elsewhere
extern char gr = 'z';   // definition because initialized
```

如果要在多个文件中使用外部变量，只需在一个文件中包含该变量的定义（单定义规则），但在使用该变量的其他所有文件中，都必须使用关键字 extern 声明它：

```
// file01.cpp
extern int cats = 20;  // definition because of initialization
int dogs = 22;         // also a definition
int fleas;             // also a definition
...
// file02.cpp
// use cats and dogs from file01.cpp
extern int cats;       // not definitions because they use
extern int dogs;       // extern and have no initialization
...
// file98.cpp
// use cats, dogs, and fleas from file01.cpp
extern int cats;
extern int dogs;
extern int fleas;
...
```

在这里，所有文件都使用了在 file01.cpp 中定义的变量 cats 和 dogs，但 file02.cpp 没有重新声明变量 fleas，因此无法访问它。在文件 file01.cpp 中，关键字 extern 并非必不可少的，因为即使省略它，效果也相同（参见图 9.4）

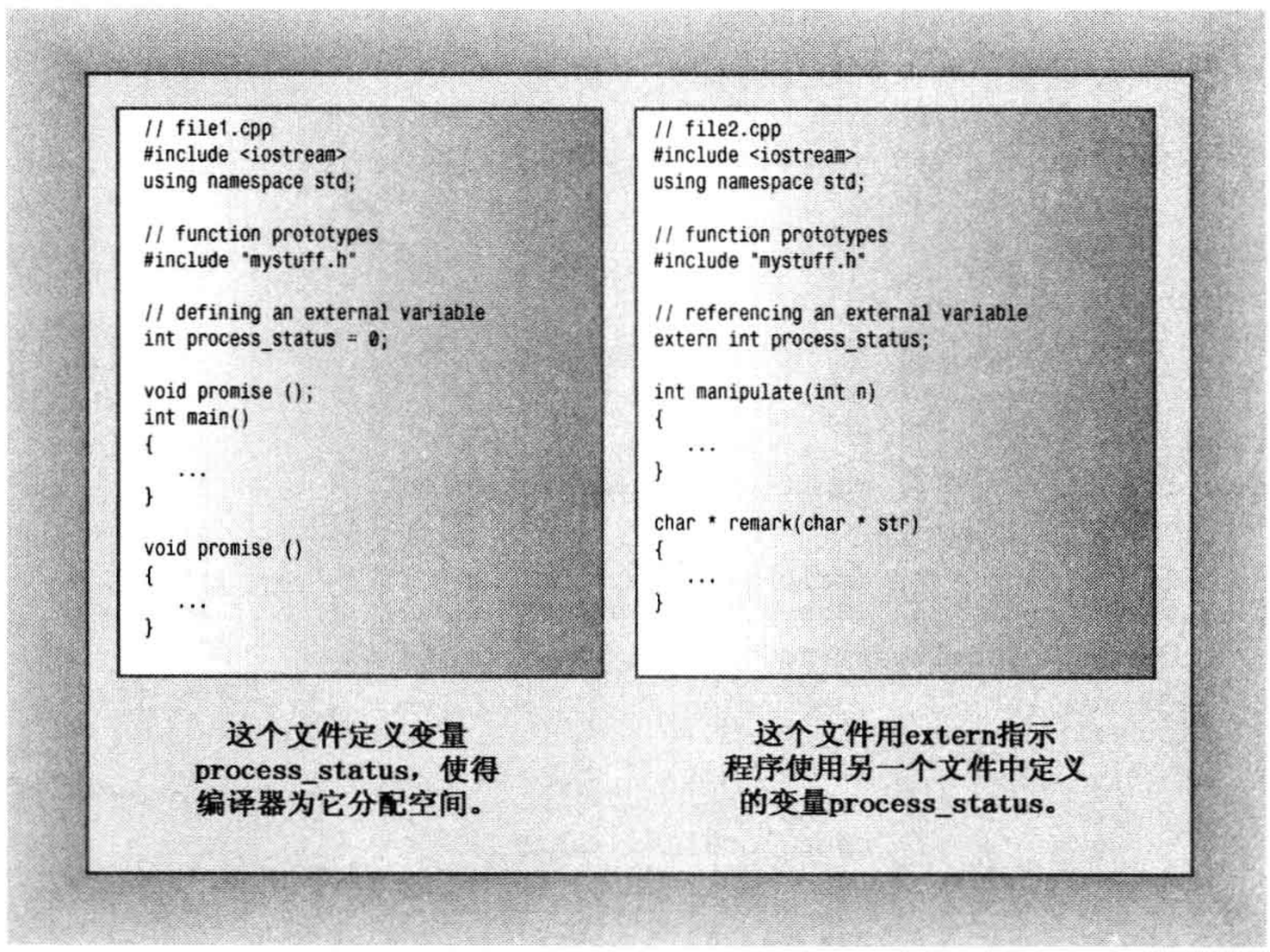


图 9.4　定义声明和引用声明

请注意，单定义规则并非意味着不能有多个变量的名称相同。例如，在不同函数中声明的同名自动变

量是彼此独立的，它们都有自己的地址。另外，正如后面的示例将表明的，局部变量可能隐藏同名的全局变量。然而，虽然程序中可包含多个同名的变量，但每个变量都只有一个定义。

如果在函数中声明了一个与外部变量同名的变量，结果将如何呢？这种声明将被视为一个自动变量的定义，当程序执行自动变量所属的函数时，该变量将位于作用域内。程序清单 9.5 和程序清单 9.6 在两个文件中使用了一个外部变量，还演示了自动变量将隐藏同名的全局变量。它还演示了如何使用关键字 extern 来重新声明以前定义过的外部变量，以及如何使用 C++的作用域解析运算符来访问被隐藏的外部变量。

程序清单 9.5　external.cpp

```
// external.cpp -- external variables
// compile with support.cpp
#include <iostream>
using namespace std;
// external variable
double warming = 0.3;        // warming defined
// function prototypes
void update(double dt);
void local();

int main()                   // uses global variable
{
    cout << "Global warming is " << warming << " degrees.\n";
    update(0.1);             // call function to change warming
    cout << "Global warming is " << warming << " degrees.\n";
    local();                 // call function with local warming
    cout << "Global warming is " << warming << " degrees.\n";
    return 0;
}
```

程序清单 9.6　support.cpp

```
// support.cpp -- use external variable
// compile with external.cpp
#include <iostream>
extern double warming;  // use warming from another file

// function prototypes
void update(double dt);
void local();

using std::cout;
void update(double dt)       // modifies global variable
{
    extern double warming;  // optional redeclaration
    warming += dt;          // uses global warming
    cout << "Updating global warming to " << warming;
    cout << " degrees.\n";
}

void local()                 // uses local variable
{
    double warming = 0.8;   // new variable hides external one

    cout << "Local warming = " << warming << " degrees.\n";
        // Access global variable with the
```

```
        // scope resolution operator
    cout << "But global warming = " << ::warming;
    cout << " degrees.\n";
}
```

下面是该程序的输出：

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

2. 程序说明

程序清单 9.5 和程序清单 9.6 所示程序的输出表明，main( )和 update( )都可以访问外部变量 warming。注意，update( )修改了 warming，这种修改在随后使用该变量时显现出来了。

在程序清单 9.5 中，warming 的定义如下：

```
double warming = 0.3;        // warming defined
```

在程序清单 9.6 中，使用关键字 extern 声明变量 warming，让该文件中的函数能够使用它：

```
extern double warming;  // use warming from another file
```

正如注释指出的，该声明的的意思是，使用外部定义的变量 warming。

另外，函数 update()使用关键字 extern 重新声明了变量 warming，这个关键字的意思是，通过这个名称使用在外部定义的变量。由于即使省略该声明，update( )的功能也相同，因此该声明是可选的。它指出该函数被设计成使用外部变量。

local( )函数表明，定义与全局变量同名的局部变量后，局部变量将隐藏全局变量。例如，local( )函数显示 warming 的值时，将使用 warming 的局部定义。

C++比 C 语言更进了一步——它提供了作用域解析运算符（::）。放在变量名前面时，该运算符表示使用变量的全局版本。因此，local( )将 warming 显示为 0.8，但将::warming 显示为 0.4。后面介绍名称空间和类时，将再次介绍该运算符。从清晰和避免错误的角度说，相对于使用 warming 并依赖于作用域规则，在函数 update()中使用::warming 是更好的选择，也更安全。

全局变量和局部变量

既然可以选择使用全局变量或局部变量，那么到底应使用哪种呢？首先，全局变量很有吸引力——因为所有的函数能访问全局变量，因此不用传递参数。但易于访问的代价很大——程序不可靠。计算经验表明，程序越能避免对数据进行不必要的访问，就越能保持数据的完整性。通常情况下，应使用局部变量，应在需要知晓时才传递数据，而不应不加区分地使用全局变量来使数据可用。读者将会看到，OOP 在数据隔离方面又向前迈进了一步。

然而，全局变量也有它们的用处。例如，可以让多个函数可以使用同一个数据块（如月份名数组或原子量数组）。外部存储尤其适于表示常量数据，因为这样可以使用关键字 const 来防止数据被修改。

```
const char * const months[12] =
{
    "January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October",
    "November", "December"
};
```

在上述示例中，第一个 const 防止字符串被修改，第二个 const 确保数组中每个指针始终指向它最初指向的字符串。

9.2.5 静态持续性、内部链接性

将 static 限定符用于作用域为整个文件的变量时，该变量的链接性将为内部的。在多文件程序中，内

部链接性和外部链接性之间的差别很有意义。链接性为内部的变量只能在其所属的文件中使用；但常规外部变量都具有外部链接性，即可以在其他文件中使用，如前面的示例所示。

如果要在其他文件中使用相同的名称来表示其他变量，该如何办呢？只需省略关键字 extern 即可吗？

```
// file1
int errors = 20;        // external declaration
...
-----------------------------------------
// file2
int errors = 5;        // ??known to file2 only??
void froobish()
{
      cout << errors;   // fails
      ...
```

这种做法将失败，因为它违反了单定义规则。file2 中的定义试图创建一个外部变量，因此程序将包含 errors 的两个定义，这是错误。

但如果文件定义了一个静态外部变量，其名称与另一个文件中声明的常规外部变量相同，则在该文件中，静态变量将隐藏常规外部变量：

```
// file1
int errors = 20;        // external declaration
...
-----------------------------------------
// file2
static int errors = 5;  // known to file2 only
void froobish()
{
      cout << errors;   // uses errors defined in file2
      ...
```

这没有违反单定义规则，因为关键字 static 指出标识符 errors 的链接性为内部，因此并非要提供外部定义。

注意：在多文件程序中，可以在一个文件（且只能在一个文件）中定义一个外部变量。使用该变量的其他文件必须使用关键字 extern 声明它。

可使用外部变量在多文件程序的不同部分之间共享数据；可使用链接性为内部的静态变量在同一个文件中的多个函数之间共享数据（名称空间提供了另外一种共享数据的方法）。另外，如果将作用域为整个文件的变量变为静态的，就不必担心其名称与其他文件中的作用域为整个文件的变量发生冲突。

程序清单 9.7 和程序清单 9.8 演示了 C++如何处理链接性为外部和内部的变量。程序清单 9.7（twofile1.cpp）定义了外部变量 tom 和 dick 以及静态外部变量 harry。这个文件中的 main( )函数显示这 3 个变量的地址，然后调用 remote_access( )函数，该函数是在另一个文件中定义的。程序清单 9.8（twofile2.cpp）列出了该文件。除定义 remote_access( )外，该文件还使用 extern 关键字来与第一个文件共享 tom。接下来，该文件定义一个名为 dick 的静态变量。static 限定符使该变量被限制在这个文件内，并覆盖相应的全局定义。然后，该文件定义了一个名为 harry 的外部变量，这不会与第一个文件中的 harry 发生冲突，因为后者的链接性为内部的。随后，remote-access( )函数显示这 3 个变量的地址，以便于将它们与第一个文件中相应变量的地址进行比较。别忘了编译这两个文件，并将它们链接起来，以得到完整的程序。

程序清单 9.7　twofile1.cpp

```
// twofile1.cpp -- variables with external and internal linkage
#include <iostream>      // to be compiled with two file2.cpp
int tom = 3;             // external variable definition
int dick = 30;           // external variable definition
static int harry = 300; // static, internal linkage
```

```
// function prototype
void remote_access();

int main()
{
    using namespace std;
    cout << "main() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    return 0;
}
```

程序清单 9.8　twofile2.cpp

```
// twofile2.cpp -- variables with internal and external linkage
#include <iostream>
extern int tom;          // tom defined elsewhere
static int dick = 10;    // overrides external dick
int harry = 200;         // external variable definition,
                         // no conflict with twofile1 harry

void remote_access()
{
    using namespace std;
    cout << "remote_access() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
}
```

下面是编译程序清单 9.7 和程序清单 9.8 生成的程序的输出：

```
main() reports the following addresses:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access() reports the following addresses:
0x0041a020 = &tom, 0x0041a450 = &dick, 0x0041a454 = &harry
```

从上述地址可知，这两个文件使用了同一个 tom 变量，但使用了不同的 dick 和 harry 变量。具体的地址和格式可能随系统而异，但两个 tom 变量的地址将相同，而两个 dick 和 harry 变量的地址不同。

9.2.6　静态存储持续性、无链接性

至此，介绍了链接性分别为内部和外部、作用域为整个文件的变量。接下来介绍静态持续家族中的第三个成员——无链接性的局部变量。这种变量是这样创建的，将 static 限定符用于在代码块中定义的变量。在代码块中使用 static 时，将导致局部变量的存储持续性为静态的。这意味着虽然该变量只在该代码块中可用，但它在该代码块不处于活动状态时仍然存在。因此在两次函数调用之间，静态局部变量的值将保持不变。（静态变量适用于再生——可以用它们将瑞士银行的秘密账号传递到下一个要去的地方）。另外，如果初始化了静态局部变量，则程序只在启动时进行一次初始化。以后再调用函数时，将不会像自动变量那样再次被初始化。程序清单 9.9 说明了这几点。

程序清单 9.9　static.cpp

```
// static.cpp -- using a static local variable
#include <iostream>
// constants
const int ArSize = 10;
// function prototype
void strcount(const char * str);
```

```
int main()
{
    using namespace std;
    char input[ArSize];
    char next;

    cout << "Enter a line:\n";
    cin.get(input, ArSize);
    while (cin)
    {
        cin.get(next);
        while (next != '\n')    // string didn't fit!
            cin.get(next);      // dispose of remainder
        strcount(input);
        cout << "Enter next line (empty line to quit):\n";
        cin.get(input, ArSize);
    }
    cout << "Bye\n";
    return 0;
}
void strcount(const char * str)
{
    using namespace std;
    static int total = 0;        // static local variable
    int count = 0;               // automatic local variable

    cout << "\"" << str <<"\" contains ";
    while (*str++)               // go to end of string
        count++;
    total += count;
    cout << count << " characters\n";
    cout << total << " characters total\n";
}
```

顺便说一句，该程序演示了一种处理行输入可能长于目标数组的方法。本书前面讲过，方法 cin.get(input, ArSize)将一直读取输入，直到到达行尾或读取了 ArSize-1 个字符为止。它把换行符留在输入队列中。该程序使用 cin.get(next)读取行输入后的字符。如果 next 是换行符，则说明 cin.get(input, ArSize)读取了整行；否则说明行中还有字符没有被读取。随后，程序使用一个循环来丢弃余下的字符，不过读者可以修改代码，让下一轮输入读取行中余下的字符。该程序还利用了这样一个事实，即试图使用 get(char *, int)读取空行将导致 cin 为 false。

下面是该程序的输出：

```
Enter a line:
nice pants
"nice pant" contains 9 characters
9 characters total
Enter next line (empty line to quit):
thanks
"thanks" contains 6 characters
15 characters total
Enter next line (empty line to quit):
parting is such sweet sorrow
"parting i" contains 9 characters
24 characters total
Enter next line (empty line to quit):
ok
"ok" contains 2 characters
```

```
26 characters total
Enter next line (empty line to quit):

Bye
```

注意，由于数组长度为 10，因此程序从每行读取的字符数都不超过 9 个。另外还需要注意的是，每次函数被调用时，自动变量 count 都被重置为 0。然而，静态变量 total 只在程序运行时被设置为 0，以后在两次函数调用之间，其值将保持不变，因此能够记录读取的字符总数。

9.2.7　说明符和限定符

有些被称为存储说明符（storage class specifier）或 cv-限定符（cv-qualifier）的 C++关键字提供了其他有关存储的信息。下面是存储说明符：

- auto（在 C++11 中不再是说明符）；
- register；
- static；
- extern；
- thread_local（C++11 新增的）；
- mutable。

其中的大部分已经介绍过了，在同一个声明中不能使用多个说明符，但 thread_local 除外，它可与 static 或 extern 结合使用。前面讲过，在 C++11 之前，可以在声明中使用关键字 auto 指出变量为自动变量；但在 C++11 中，auto 用于自动类型推断。关键字 register 用于在声明中指示寄存器存储，而在 C++11 中，它只是显式地指出变量是自动的。关键字 static 被用在作用域为整个文件的声明中时，表示内部链接性；被用于局部声明中，表示局部变量的存储持续性为静态的。关键字 extern 表明是引用声明，即声明引用在其他地方定义的变量。关键字 thread_local 指出变量的持续性与其所属线程的持续性相同。thread_local 变量之于线程，犹如常规静态变量之于整个程序。关键字 mutable 的含义将根据 const 来解释，因此先来介绍 cv-限定符，然后再解释它。

1. cv-限定符

下面就是 cv 限定符：

- const；
- volatile。

（读者可能猜到了，cv 表示 const 和 volatile）。最常用的 cv-限定符是 const，而读者已经知道其用途。它表明，内存被初始化后，程序便不能再对它进行修改。稍后再回过头来介绍它。

关键字 volatile 表明，即使程序代码没有对内存单元进行修改，其值也可能发生变化。听起来似乎很神秘，实际上并非如此。例如，可以将一个指针指向某个硬件位置，其中包含了来自串行端口的时间或信息。在这种情况下，硬件（而不是程序）可能修改其中的内容。或者两个程序可能互相影响，共享数据。该关键字的作用是为了改善编译器的优化能力。例如，假设编译器发现，程序在几条语句中两次使用了某个变量的值，则编译器可能不是让程序查找这个值两次，而是将这个值缓存到寄存器中。这种优化假设变量的值在这两次使用之间不会变化。如果不将变量声明为 volatile，则编译器将进行这种优化；将变量声明为 volatile，相当于告诉编译器，不要进行这种优化。

2. mutable

现在回到 mutable。可以用它来指出，即使结构（或类）变量为 const，其某个成员也可以被修改。例如，请看下面的代码：

```
struct data
{
    char name[30];
    mutable int accesses;
    ...
};
```

```
const data veep = {"Claybourne Clodde", 0, ... };
strcpy(veep.name, "Joye Joux");    // not allowed
veep.accesses++;                   // allowed
```

veep 的 const 限定符禁止程序修改 veep 的成员，但 access 成员的 mutable 说明符使得 access 不受这种限制。本书不使用 volatile 或 mutable，但将进一步介绍 const。

3. 再谈 const

在 C++（但不是在 C 语言）中，const 限定符对默认存储类型稍有影响。在默认情况下全局变量的链接性为外部的，但 const 全局变量的链接性为内部的。也就是说，在 C++看来，全局 const 定义（如下述代码段所示）就像使用了 static 说明符一样。

```
const int fingers = 10;    // same as static const int fingers = 10;
int main(void)
{
    ...
```

C++修改了常量类型的规则，让程序员更轻松。例如，假设将一组常量放在头文件中，并在同一个程序的多个文件中使用该头文件。那么，预处理器将头文件的内容包含到每个源文件中后，所有的源文件都将包含类似下面这样的定义：

```
const int fingers = 10;
const char * warning = "Wak!";
```

如果全局 const 声明的链接性像常规变量那样是外部的，则根据单定义规则，这将出错。也就是说，只能有一个文件可以包含前面的声明，而其他文件必须使用 extern 关键字来提供引用声明。另外，只有未使用 extern 关键字的声明才能进行初始化：

```
// extern would be required if const had external linkage
extern const int fingers;       // can't be initialized
extern const char * warning;
```

因此，需要为某个文件使用一组定义，而其他文件使用另一组声明。然而，由于外部定义的 const 数据的链接性为内部的，因此可以在所有文件中使用相同的声明。

内部链接性还意味着，每个文件都有自己的一组常量，而不是所有文件共享一组常量。每个定义都是其所属文件私有的，这就是能够将常量定义放在头文件中的原因。这样，只要在两个源代码文件中包括同一个头文件，则它们将获得同一组常量。

如果出于某种原因，程序员希望某个常量的链接性为外部的，则可以使用 extern 关键字来覆盖默认的内部链接性：

```
extern const int states = 50;   // definition with external linkage
```

在这种情况下，必须在所有使用该常量的文件中使用 extern 关键字来声明它。这与常规外部变量不同，定义常规外部变量时，不必使用 extern 关键字，但在使用该变量的其他文件中必须使用 extern。然而，请记住，鉴于单个 const 在多个文件之间共享，因此只有一个文件可对其进行初始化。

在函数或代码块中声明 const 时，其作用域为代码块，即仅当程序执行该代码块中的代码时，该常量才是可用的。这意味着在函数或代码块中创建常量时，不必担心其名称与其他地方定义的常量发生冲突。

9.2.8 函数和链接性

和变量一样，函数也有链接性，虽然可选择的范围比变量小。和 C 语言一样，C++不允许在一个函数中定义另外一个函数，因此所有函数的存储持续性都自动为静态的，即在整个程序执行期间都一直存在。在默认情况下，函数的链接性为外部的，即可以在文件间共享。实际上，可以在函数原型中使用关键字 extern 来指出函数是在另一个文件中定义的，不过这是可选的（要让程序在另一个文件中查找函数，该文件必须作为程序的组成部分被编译，或者是由链接程序搜索的库文件）。还可以使用关键字 static 将函数的链接性设置为内部的，使之只能在一个文件中使用。必须同时在原

型和函数定义中使用该关键字：

```
static int private(double x);
...
static int private(double x)
{
    ...
}
```

这意味着该函数只在这个文件中可见，还意味着可以在其他文件中定义同名的的函数。和变量一样，在定义静态函数的文件中，静态函数将覆盖外部定义，因此即使在外部定义了同名的函数，该文件仍将使用静态函数。

单定义规则也适用于非内联函数，因此对于每个非内联函数，程序只能包含一个定义。对于链接性为外部的函数来说，这意味着在多文件程序中，只能有一个文件（该文件可能是库文件，而不是您提供的）包含该函数的定义，但使用该函数的每个文件都应包含其函数原型。

内联函数不受这项规则的约束，这允许程序员能够将内联函数的定义放在头文件中。这样，包含了头文件的每个文件都有内联函数的定义。然而，C++要求同一个函数的所有内联定义都必须相同。

C++在哪里查找函数

假设在程序的某个文件中调用一个函数，C++将到哪里去寻找该函数的定义呢？如果该文件中的函数原型指出该函数是静态的，则编译器将只在该文件中查找函数定义；否则，编译器（包括链接程序）将在所有的程序文件中查找。如果找到两个定义，编译器将发出错误消息，因为每个外部函数只能有一个定义。如果在程序文件中没有找到，编译器将在库中搜索。这意味着如果定义了一个与库函数同名的函数，编译器将使用程序员定义的版本，而不是库函数（然而，C++保留了标准库函数的名称，即程序员不应使用它们）。有些编译器-链接程序要求显式地指出要搜索哪些库。

9.2.9　语言链接性

另一种形式的链接性——称为语言链接性（language linking）也对函数有影响。首先介绍一些背景知识。链接程序要求每个不同的函数都有不同的符号名。在 C 语言中，一个名称只对应一个函数，因此这很容易实现。为满足内部需要，C 语言编译器可能将 spiff 这样的函数名翻译为_spiff。这种方法被称为 C 语言链接性（C language linkage）。但在 C++中，同一个名称可能对应多个函数，必须将这些函数翻译为不同的符号名称。因此，C++编译器执行名称矫正或名称修饰（参见第 8 章），为重载函数生成不同的符号名称。例如，可能将 spiff（int）转换为_spoff_i，而将 spiff（double，double）转换为_spiff_d_d。这种方法被称为 C++语言链接（C++ language linkage）。

链接程序寻找与 C++函数调用匹配的函数时，使用的方法与 C 语言不同。但如果要在 C++程序中使用 C 库中预编译的函数，将出现什么情况呢？例如，假设有下面的代码：

```
spiff(22); // want spiff(int) from a C library
```

它在 C 库文件中的符号名称为_spiff，但对于我们假设的链接程序来说，C++查询约定是查找符号名称_spiff_i。为解决这种问题，可以用函数原型来指出要使用的约定：

```
extern "C" void spiff(int);   // use C protocol for name look-up
extern void spoff(int);       // use C++ protocol for name look-up
extern "C++" void spaff(int); // use C++ protocol for name look-up
```

第一个原型使用 C 语言链接性；而后面的两个使用 C++语言链接性。第二个原型是通过默认方式指出这一点的，而第三个显式地指出了这一点。

C 和 C++链接性是 C++标准指定的说明符，但实现可提供其他语言链接性说明符。

9.2.10　存储方案和动态分配

前面介绍 C++用来为变量（包括数组和结构）分配内存的 5 种方案（线程内存除外），它们不适用于

使用 C++运算符 new（或 C 函数 malloc( )）分配的内存，这种内存被称为动态内存。第 4 章介绍过，动态内存由运算符 new 和 delete 控制，而不是由作用域和链接性规则控制。因此，可以在一个函数中分配动态内存，而在另一个函数中将其释放。与自动内存不同，动态内存不是 LIFO，其分配和释放顺序要取决于 new 和 delete 在何时以何种方式被使用。通常，编译器使用三块独立的内存：一块用于静态变量（可能再细分），一块用于自动变量，另外一块用于动态存储。

虽然存储方案概念不适用于动态内存，但适用于用来跟踪动态内存的自动和静态指针变量。例如，假设在一个函数中包含下面的语句：

```
float * p_fees = new float [20];
```

由 new 分配的 80 个字节（假设 float 为 4 个字节）的内存将一直保留在内存中，直到使用 delete 运算符将其释放。但当包含该声明的语句块执行完毕时，p_fees 指针将消失。如果希望另一个函数能够使用这 80 个字节中的内容，则必须将其地址传递或返回给该函数。另一方面，如果将 p_fees 的链接性声明为外部的，则文件中位于该声明后面的所有函数都可以使用它。另外，通过在另一个文件中使用下述声明，便可在其中使用该指针：

```
extern float * p_fees;
```

注意：在程序结束时，由 new 分配的内存通常都将被释放，不过情况也并不总是这样。例如，在不那么健壮的操作系统中，在某些情况下，请求大型内存块将导致该代码块在程序结束不会被自动释放。最佳的做法是，使用 delete 来释放 new 分配的内存。

1. 使用 new 运算符初始化

如果要初始化动态分配的变量，该如何办呢？在 C++98 中，有时候可以这样做，C++11 增加了其他可能性。下面先来看看 C++98 提供的可能性。

如果要为内置的标量类型（如 int 或 double）分配存储空间并初始化，可在类型名后面加上初始值，并将其用括号括起：

```
int *pi = new int (6);   // *pi set to 6
double * pd = new double (99.99);  // *pd set to 99.99
```

这种括号语法也可用于有合适构造函数的类，这将在本书后面介绍。

然而，要初始化常规结构或数组，需要使用大括号的列表初始化，这要求编译器支持 C++11。C++11 允许您这样做：

```
struct where {double x; double y; double z;};
where * one = new where {2.5, 5.3, 7.2};  // C++11
int * ar = new int [4] {2,4,6,7};         // C++11
```

在 C++11 中，还可将列表初始化用于单值变量：

```
int *pin = new int {});   // *pi set to 6
double * pdo = new double {99.99};  // *pd set to 99.99
```

2. new 失败时

new 可能找不到请求的内存量。在最初的 10 年中，C++在这种情况下让 new 返回空指针，但现在将引发异常 std::bad_alloc。第 15 章通过一些简单的示例演示了这两种方法的工作原理。

3. new：运算符、函数和替换函数

运算符 new 和 new []分别调用如下函数：

```
void * operator new(std::size_t);      // used by new
void * operator new[](std::size_t);    // used by new[]
```

这些函数被称为分配函数（alloction function），它们位于全局名称空间中。同样，也有由 delete 和 delete []调用的释放函数（deallocation function）：

```
void operator delete(void  *);
void operator delete[](void *);
```

它们使用第 11 章将讨论的运算符重载语法。std::size_t 是一个 typedef，对应于合适的整型。对于下面这样的基本语句：

```
int * pi = new int;
```

将被转换为下面这样：

```
int * pi = new(sizeof(int));
```

而下面的语句：

```
int * pa = new int[40];
```

将被转换为下面这样：

```
int * pa = new(40 * sizeof(int));
```

正如您知道的，使用运算符 new 的语句也可包含初始值，因此，使用 new 运算符时，可能不仅仅是调用 new()函数。

同样，下面的语句：

```
delete pi;
```

将转换为如下函数调用：

```
delete (pi);
```

有趣的是，C++将这些函数称为可替换的（replaceable）。这意味着如果您有足够的知识和意愿，可为 new 和 delete 提供替换函数，并根据需要对其进行定制。例如，可定义作用域为类的替换函数，并对其进行定制，以满足该类的内存分配需求。在代码中，仍将使用 new 运算符，但它将调用您定义的 new()函数。

4. 定位 new 运算符

通常，new 负责在堆（heap）中找到一个足以能够满足要求的内存块。new 运算符还有另一种变体，被称为定位（placement）new 运算符，它让您能够指定要使用的位置。程序员可能使用这种特性来设置其内存管理规程、处理需要通过特定地址进行访问的硬件或在特定位置创建对象。

要使用定位 new 特性，首先需要包含头文件 new，它提供了这种版本的 new 运算符的原型；然后将 new 运算符用于提供了所需地址的参数。除需要指定参数外，句法与常规 new 运算符相同。具体地说，使用定位 new 运算符时，变量后面可以有方括号，也可以没有。下面的代码段演示了 new 运算符的 4 种用法：

```
#include <new>
struct chaff
{
    char dross[20];
    int slag;
};
char buffer1[50];
char buffer2[500];
int main()
{
    chaff *p1, *p2;
    int *p3, *p4;
// first, the regular forms of new
    p1 = new chaff;              // place structure in heap
    p3 = new int[20];            // place int array in heap
// now, the two forms of placement new
    p2 = new (buffer1) chaff;    // place structure in buffer1
    p4 = new (buffer2) int[20];  // place int array in buffer2
...
```

出于简化的目的，这个示例使用两个静态数组来为定位 new 运算符提供内存空间。因此，上述代码从 buffer1 中分配空间给结构 chaff，从 buffer2 中分配空间给一个包含 20 个元素的 int 数组。

熟悉定位 new 运算符后，来看一个示例程序。程序清单 9.10 使用常规 new 运算符和定位 new 运算符创建动态分配的数组。该程序说明了常规 new 运算符和定位 new 运算符之间的一些重要差别，在查看该程序的输出后，将对此进行讨论。

程序清单 9.10　newplace.cpp

```
// newplace.cpp -- using placement new
#include <iostream>
#include <new>          // for placement new
const int BUF = 512;
const int N = 5;
char buffer[BUF];       // chunk of memory
int main()
{
    using namespace std;
    double *pd1, *pd2;
    int i;
    cout << "Calling new and placement new:\n";
    pd1 = new double[N];            // use heap
    pd2 = new (buffer) double[N];   // use buffer array
    for (i = 0; i < N; i++)
        pd2[i] = pd1[i] = 1000 + 20.0 * i;
    cout << "Memory addresses:\n" << "  heap: " << pd1
        << "  static: " << (void *) buffer  <<endl;
    cout << "Memory contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd1[i] << " at " << &pd1[i] << "; ";
        cout << pd2[i] << " at " << &pd2[i] << endl;
    }

    cout << "\nCalling new and placement new a second time:\n";
    double *pd3, *pd4;
    pd3= new double[N];             // find new address
    pd4 = new (buffer) double[N];   // overwrite old data
    for (i = 0; i < N; i++)
        pd4[i] = pd3[i] = 1000 + 40.0 * i;
    cout << "Memory contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd3[i] << " at " << &pd3[i] << "; ";
        cout << pd4[i] << " at " << &pd4[i] << endl;
    }

    cout << "\nCalling new and placement new a third time:\n";
    delete [] pd1;
    pd1= new double[N];
    pd2 = new (buffer + N * sizeof(double)) double[N];
    for (i = 0; i < N; i++)
        pd2[i] = pd1[i] = 1000 + 60.0 * i;
    cout << "Memory contents:\n";
    for (i = 0; i < N; i++)
    {
```

```
        cout << pd1[i] << " at " << &pd1[i] << "; ";
        cout << pd2[i] << " at " << &pd2[i] << endl;
    }
    delete [] pd1;
    delete [] pd3;
    return 0;
}
```

下面是该程序在某个系统上运行时的输出：

```
Calling new and placement new:
Memory addresses:
  heap: 006E4AB0  static: 00FD9138
Memory contents:
1000 at 006E4AB0; 1000 at 00FD9138
1020 at 006E4AB8; 1020 at 00FD9140
1040 at 006E4AC0; 1040 at 00FD9148
1060 at 006E4AC8; 1060 at 00FD9150
1080 at 006E4AD0; 1080 at 00FD9158

Calling new and placement new a second time:
Memory contents:
1000 at 006E4B68; 1000 at 00FD9138
1040 at 006E4B70; 1040 at 00FD9140
1080 at 006E4B78; 1080 at 00FD9148
1120 at 006E4B80; 1120 at 00FD9150
1160 at 006E4B88; 1160 at 00FD9158

Calling new and placement new a third time:
Memory contents:
1000 at 006E4AB0; 1000 at 00FD9160
1060 at 006E4AB8; 1060 at 00FD9168
1120 at 006E4AC0; 1120 at 00FD9170
1180 at 006E4AC8; 1180 at 00FD9178
1240 at 006E4AD0; 1240 at 00FD9180
```

5. 程序说明

有关程序清单 9.10，首先要指出的一点是，定位 new 运算符确实将数组 p2 放在了数组 buffer 中，p2 和 buffer 的地址都是 00FD9138。然而，它们的类型不同，p1 是 double 指针，而 buffer 是 char 指针（顺便说一句，这也是程序使用(void *)对 buffer 进行强制转换的原因，如果不这样做，cout 将显示一个字符串）同时，常规 new 将数组 p1 放在很远的地方，其地址为 006E4AB0，位于动态管理的堆中。

需要指出的第二点是，第二个常规 new 运算符查找一个新的内存块，其起始地址为 006E4B68；但第二个定位 new 运算符分配与以前相同的内存块：起始地址为 00FD9138 的内存块。定位 new 运算符使用传递给它的地址，它不跟踪哪些内存单元已被使用，也不查找未使用的内存块。这将一些内存管理的负担交给了程序员。例如，在第三次调用定位 new 运算符时，提供了一个从数组 buffer 开头算起的偏移量，因此将分配新的内存：

```
pd2 = new (buffer + N * sizeof(double)) double[N]; // offset of 40 bytes
```

第三点差别是，是否使用 delete 来释放内存。对于常规 new 运算符，下面的语句释放起始地址为 006E4AB0 的内存块，因此接下来再次调用 new 运算符时，该内存块是可用的：

```
delete [] pd1;
```

然而，程序清单 9.10 中的程序没有使用 delete 来释放使用定位 new 运算符分配的内存。事实上，在这个例子中不能这样做。buffer 指定的内存是静态内存，而 delete 只能用于这样的指针：指向常规 new 运算符分配的堆内存。也就是说，数组 buffer 位于 delete 的管辖区域之外，下面的语句将引发运

行阶段错误：

```
delete [] pd2;   // won't work
```

另一方面，如果 buffer 是使用常规 new 运算符创建的，便可以使用常规 delete 运算符来释放整个内存块。

定位 new 运算符的另一种用法是，将其与初始化结合使用，从而将信息放在特定的硬件地址处。

您可能想知道定位 new 运算符的工作原理。基本上，它只是返回传递给它的地址，并将其强制转换为 void *，以便能够赋给任何指针类型。但这说的是默认定位 new 函数，C++允许程序员重载定位 new 函数。

将定位 new 运算符用于类对象时，情况将更复杂，这将在第 12 章介绍。

6. 定位 new 的其他形式

就像常规 new 调用一个接收一个参数的 new()函数一样，标准定位 new 调用一个接收两个参数的 new()函数：

```
int * pi = new int;              // invokes new(sizeof(int))
int * p2 = new(buffer) int;      // invokes new(sizeof(int), buffer)
int * p3 = new(buffer) int[40]; // invokes new(40*sizeof(int), buffer)
```

定位 new 函数不可替换，但可重载。它至少需要接收两个参数，其中第一个总是 std::size_t，指定了请求的字节数。这样的重载函数都被称为定义 new，即使额外的参数没有指定位置。

9.3 名称空间

在 C++中，名称可以是变量、函数、结构、枚举、类以及类和结构的成员。当随着项目的增大，名称相互冲突的可能性也将增加。使用多个厂商的类库时，可能导致名称冲突。例如，两个库可能都定义了名为 List、Tree 和 Node 的类，但定义的方式不兼容。用户可能希望使用一个库的 List 类，而使用另一个库的 Tree 类。这种冲突被称为名称空间问题。

C++标准提供了名称空间工具，以便更好地控制名称的作用域。经过了一段时间后，编译器才支持名称空间，但现在这种支持很普遍。

9.3.1 传统的 C++名称空间

介绍 C++中新增的名称空间特性之前，先复习一下 C++中已有的名称空间属性，并介绍一些术语，让读者熟悉名称空间的概念。

第一个需要知道的术语是声明区域（declaration region）。声明区域是可以在其中进行声明的区域。例如，可以在函数外面声明全局变量，对于这种变量，其声明区域为其声明所在的文件。对于在函数中声明的变量，其声明区域为其声明所在的代码块。

第二个需要知道的术语是潜在作用域（potential scope）。变量的潜在作用域从声明点开始，到其声明区域的结尾。因此潜在作用域比声明区域小，这是由于变量必须定义后才能使用。

然而，变量并非在其潜在作用域内的任何位置都是可见的。例如，它可能被另一个在嵌套声明区域中声明的同名变量隐藏。例如，在函数中声明的局部变量（对于这种变量，声明区域为整个函数）将隐藏在同一个文件中声明的全局变量（对于这种变量，声明区域为整个文件）。变量对程序而言可见的范围被称为作用域（scope），前面正是以这种方式使用该术语的。图 9.5 和图 9.6 对术语声明区域、潜在作用域和作用域进行了说明。

C++关于全局变量和局部变量的规则定义了一种名称空间层次。每个声明区域都可以声明名称，这些名称独立于在其他声明区域中声明的名称。在一个函数中声明的局部变量不会与在另一个函数中声明的局部变量发生冲突。

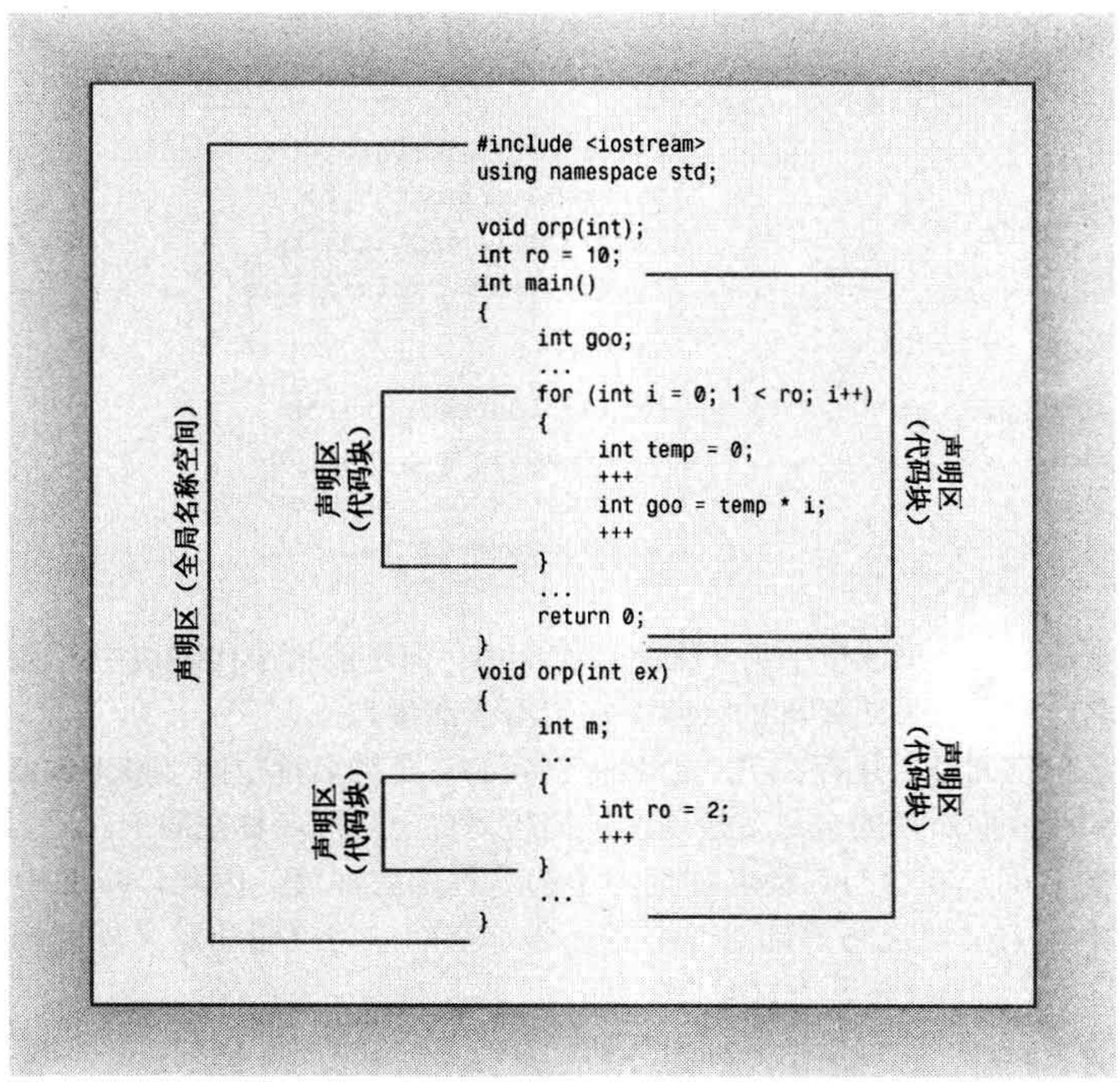


图 9.5　声明区域

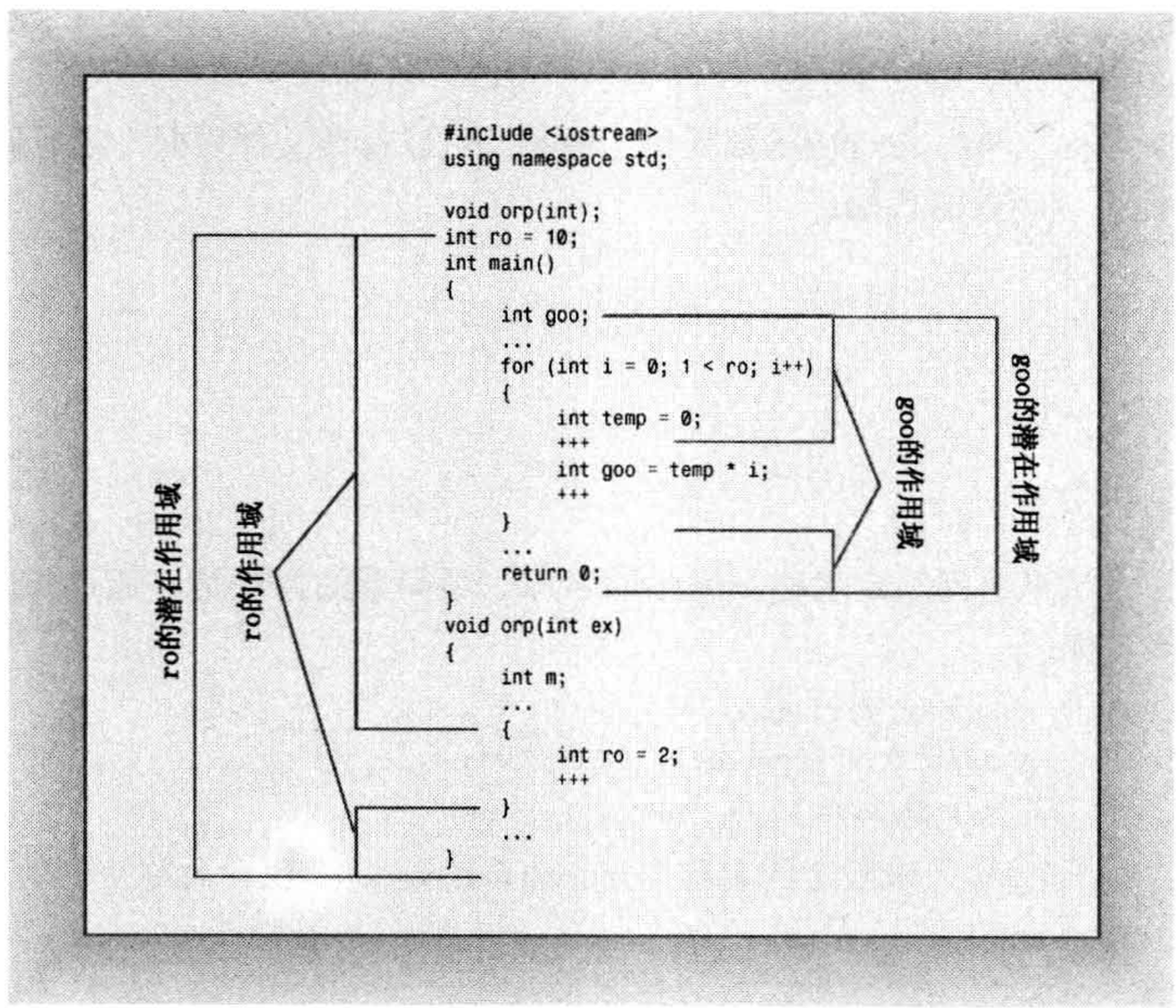


图 9.6　潜在作用域和作用域

9.3.2　新的名称空间特性

C++新增了这样一种功能，即通过定义一种新的声明区域来创建命名的名称空间，这样做的目的之一是提供一个声明名称的区域。一个名称空间中的名称不会与另外一个名称空间的相同名称发生冲突，同时

允许程序的其他部分使用该名称空间中声明的东西。例如，下面的代码使用新的关键字 namespace 创建了两个名称空间：Jack 和 Jill。

```
namespace Jack {
    double pail;                    // variable declaration
    void fetch();                   // function prototype
    int pal;                        // variable declaration
    struct Well { ... };            // structure declaration
}
namespace Jill {
    double bucket(double n) { ... } // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill { ... };            // structure declaration
}
```

名称空间可以是全局的，也可以位于另一个名称空间中，但不能位于代码块中。因此，在默认情况下，在名称空间中声明的名称的链接性为外部的（除非它引用了常量）。

除了用户定义的名称空间外，还存在另一个名称空间——全局名称空间（global namespace）。它对应于文件级声明区域，因此前面所说的全局变量现在被描述为位于全局名称空间中。

任何名称空间中的名称都不会与其他名称空间中的名称发生冲突。因此，Jack 中的 fetch 可以与 Jill 中的 fetch 共存，Jill 中的 Hill 可以与外部 Hill 共存。名称空间中的声明和定义规则同全局声明和定义规则相同。

名称空间是开放的（open），即可以把名称加入到已有的名称空间中。例如，下面这条语句将名称 goose 添加到 Jill 中已有的名称列表中：

```
namespace Jill {
    char * goose(const char *);
}
```

同样，原来的 Jack 名称空间为 fetch( )函数提供了原型。可以在该文件后面（或另外一个文件中）再次使用 Jack 名称空间来提供该函数的代码：

```
namespace Jack {
    void fetch()
    {
        ...
    }
}
```

当然，需要有一种方法来访问给定名称空间中的名称。最简单的方法是，通过作用域解析运算符::，使用名称空间来限定该名称：

```
Jack::pail = 12.34;  // use a variable
Jill::Hill mole;     // create a type Hill structure
Jack::fetch();       // use a function
```

未被装饰的名称（如 pail）称为未限定的名称（unqualified name）；包含名称空间的名称（如 Jack::pail）称为限定的名称（qualified name）。

1. using 声明和 using 编译指令

我们并不希望每次使用名称时都对它进行限定，因此 C++提供了两种机制（using 声明和 using 编译指令）来简化对名称空间中名称的使用。using 声明使特定的标识符可用，using 编译指令使整个名称空间可用。

using 声明由被限定的名称和它前面的关键字 using 组成：

```
using Jill::fetch;    // a using declaration
```

using 声明将特定的名称添加到它所属的声明区域中。例如 main( )中的 using 声明 Jill::fetch 将 fetch 添加到 main( )定义的声明区域中。完成该声明后，便可以使用名称 fetch 代替 Jill::fetch。下面的代码段说明了这几点：

```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
char fetch;
int main()
{
    using Jill::fetch;   // put fetch into local namespace
    double fetch;        // Error! Already have a local fetch
    cin >> fetch;        // read a value into Jill::fetch
    cin >> ::fetch;      // read a value into global fetch
    ...
}
```

由于 using 声明将名称添加到局部声明区域中，因此这个示例避免了将另一个局部变量也命名为 fetch。另外，和其他局部变量一样，fetch 也将覆盖同名的全局变量。

在函数的外面使用 using 声明时，将把名称添加到全局名称空间中：

```
void other();
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
using Jill::fetch;   // put fetch into global namespace
int main()
{
    cin >> fetch;    // read a value into Jill::fetch
    other()
...
}

void other()
{
    cout << fetch;   // display Jill::fetch
...
}
```

using 声明使一个名称可用，而 using 编译指令使所有的名称都可用。using 编译指令由名称空间名和它前面的关键字 using namespace 组成，它使名称空间中的所有名称都可用，而不需要使用作用域解析运算符：

```
using namespace Jack;  // make all the names in Jack available
```

在全局声明区域中使用 using 编译指令，将使该名称空间的名称全局可用。这种情况已出现过多次：

```
#include <iostream>    // places names in namespace std
using namespace std;   // make names available globally
```

在函数中使用 using 编译指令，将使其中的名称在该函数中可用，下面是一个例子：

```
int main()
{
    using namespace jack; // make names available in vorn()
...
}
```

在本书前面中，经常将这种格式用于名称空间 std。

有关 using 编译指令和 using 声明，需要记住的一点是，它们增加了名称冲突的可能性。也就是说，如果有名称空间 jack 和 jill，并在代码中使用作用域解析运算符，则不会存在二义性：

```
jack::pal = 3;
jill::pal =10;
```

变量 jack::pal 和 jill::pal 是不同的标识符，表示不同的内存单元。然而，如果使用 using 声明，情况将发生变化：

```
using jack::pal;
using jill::pal;
pal = 4;          // which one? now have a conflict
```

事实上，编译器不允许您同时使用上述两个 using 声明，因为这将导致二义性。

2. using 编译指令和 using 声明之比较

使用 using 编译指令导入一个名称空间中所有的名称与使用多个 using 声明是不一样的，而更像是大量使用作用域解析运算符。使用 using 声明时，就好像声明了相应的名称一样。如果某个名称已经在函数中声明了，则不能用 using 声明导入相同的名称。然而，使用 using 编译指令时，将进行名称解析，就像在包含 using 声明和名称空间本身的最小声明区域中声明了名称一样。在下面的示例中，名称空间为全局的。如果使用 using 编译指令导入一个已经在函数中声明的名称，则局部名称将隐藏名称空间名，就像隐藏同名的全局变量一样。不过仍可以像下面的示例中那样使用作用域解析运算符：

```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
char fetch;                    // global namespace
int main()
{
    using namespace Jill;      // import all namespace names
    Hill Thrill;               // create a type Jill::Hill structure
    double water = bucket(2);  // use Jill::bucket();
    double fetch;              // not an error; hides Jill::fetch
    cin >> fetch;              // read a value into the local fetch
    cin >> ::fetch;            // read a value into global fetch
    cin >> Jill::fetch;        // read a value into Jill::fetch
    ...
}

int foom()
{
    Hill top;                  // ERROR
    Jill::Hill crest;          // valid
}
```

在 main( )中，名称 Jill::fetch 被放在局部名称空间中，但其作用域不是局部的，因此不会覆盖全局的 fetch。然而，局部声明的 fetch 将隐藏 Jill::fetch 和全局 fetch。然而，如果使用作用域解析运算符，则后两个 fetch 变量都是可用的。读者应将这个示例与前面使用 using 声明的示例进行比较。

需要指出的另一点是，虽然函数中的 using 编译指令将名称空间的名称视为在函数之外声明的，但它不会使得该文件中的其他函数能够使用这些名称。因此，在前一个例子中，foom( )函数不能使用未限定的标识符 Hill。

注意：假设名称空间和声明区域定义了相同的名称。如果试图使用 using 声明将名称空间的名称导入

该声明区域，则这两个名称会发生冲突，从而出错。如果使用 using 编译指令将该名称空间的名称导入该声明区域，则局部版本将隐藏名称空间版本。

一般说来，使用 using 声明比使用 using 编译指令更安全，这是由于它只导入指定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using 编译指令导入所有名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

下面是本书的大部分示例采用的方法：

```
#include <iostream>
int main()
{
    using namespace std;
```

首先，#include 语句将头文件 iostream 放到名称空间 std 中。然后，using 编译指令是该名称空间在 main( ) 函数中可用。有些示例采取下述方式：

```
#include <iostream>
using namespace std;
int main()
{
```

这将名称空间 std 中的所有内容导出到全局名称空间中。使用这种方法的主要原因是方便。它易于完成，同时如果系统不支持名称空间，可以将前两行替换为：

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

然而，名称空间的支持者希望有更多的选择，既可以使用解析运算符，也可以使用 using 声明。也就是说，不要这样做：

```
using namespace std;  // avoid as too indiscriminate
```

而应这样做：

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

或者这样做：

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

可以用嵌套式名称空间（将在下一节介绍）来创建一个包含常用 using 声明的名称空间。

3. 名称空间的其他特性

可以将名称空间声明进行嵌套：

```
namespace elements
{
    namespace fire
    {
        int flame;
        ...
    }
    float water;
}
```

这里，flame 指的是 element::fire::flame。同样，可以使用下面的 using 编译指令使内部的名称可用：

```
using namespace elements::fire;
```

另外，也可以在名称空间中使用 using 编译指令和 using 声明，如下所示：

```
namespace myth
{
    using Jill::fetch;
    using namespace elements;
    using std::cout;
    using std::cin;
}
```

假设要访问 Jill::fetch。由于 Jill::fetch 现在位于名称空间 myth（在这里，它被叫做 fetch）中，因此可以这样访问它：

```
std::cin >> myth::fetch;
```

当然，由于它也位于 Jill 名称空间中，因此仍然可以称作 Jill::fetch：

```
Jill::fetch:

std::cout << Jill::fetch;  // display value read into myth::fetch
```

如果没有与之冲突的局部变量，则也可以这样做：

```
using namespace myth;
cin >> fetch;       // really std::cin and Jill::fetch
```

现在考虑将 using 编译指令用于 myth 名称空间的情况。using 编译指令是可传递的。如果 A op B 且 B op C，则 A op C，则说操作 op 是可传递的。例如，>运算符是可传递的（也就是说，如果 A>B 且 B>C，则 A>C）。在这个情况下，下面的语句将导入名称空间 myth 和 elements：

```
using namespace myth;
```

这条编译指令与下面两条编译指令等价：

```
using namespace myth;
using namespace elements;
```

可以给名称空间创建别名。例如，假设有下面的名称空间：

```
namespace my_very_favorite_things { ... };
```

则可以使用下面的语句让 mvft 成为 my_very_favorite_things 的别名：

```
namespace mvft = my_very_favorite_things;
```

可以使用这种技术来简化对嵌套名称空间的使用：

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

4. 未命名的名称空间

可以通过省略名称空间的名称来创建未命名的名称空间：

```
namespace         // unnamed namespace
{
    int ice;
    int bandycoot;
}
```

这就像后面跟着 using 编译指令一样，也就是说，在该名称空间中声明的名称的潜在作用域为：从声明点到该声明区域末尾。从这个方面看，它们与全局变量相似。然而，由于这种名称空间没有名称，因此不能显式地使用 using 编译指令或 using 声明来使它在其他位置都可用。具体地说，不能在未命名名称空间所属文件之外的其他文件中，使用该名称空间中的名称。这提供了链接性为内部的静态变量的替代品。例如，假设有这样的代码：

```
static int counts;   // static storage, internal linkage
int other();
int main()
{
...
}

int other()
{
...
}
```

采用名称空间的方法如下：

```
namespace
{
    int counts;   // static storage, internal linkage
}
int other();
int main()
{
...
}

int other()
{
...
}
```

9.3.3　名称空间示例

现在来看一个多文件示例，该示例说明了名称空间的一些特性。该程序的第一个文件（参见程序清单 9.11）是头文件，其中包含头文件中常包含的内容：常量、结构定义和函数原型。在这个例子中，这些内容被放在两个名称空间中。第一个名称空间叫做 pers，其中包含 Person 结构的定义和两个函数的原型——一个函数用人名填充结构，另一个函数显示结构的内容；第二个名称空间叫做 debts，它定义了一个结构，该结构用来存储人名和金额。该结构使用了 Person 结构，因此，debts 名称空间使用一条 using 编译指令，让 pers 中的名称在 debts 名称空间可用。debts 名称空间也包含一些原型。

程序清单 9.11　namesp.h

```
// namesp.h
#include <string>
// create the pers and debts namespaces
namespace pers
{
    struct Person
    {
        std::string fname;
        std::string lname;
     };
    void getPerson(Person &);
    void showPerson(const Person &);
}

namespace debts
```

```
{
    using namespace pers;
    struct Debt
    {
        Person name;
        double amount;
    };
    void getDebt(Debt &);
    void showDebt(const Debt &);
    double sumDebts(const Debt ar[], int n);
}
```

第二个文件（见程序清单 9.12）是源代码文件，它提供了头文件中的函数原型对应的定义。在名称空间中声明的函数名的作用域为整个名称空间，因此定义和声明必须位于同一个名称空间中。这正是名称空间的开放性发挥作用的地方。通过包含 namesp.h（参见程序清单 9.11）导入了原来的名称空间。然后该文件将函数定义添加入到两个名称空间中，如程序清单 9.12 所示。另外，文件 names.cpp 演示了如何使用 using 声明和作用域解析运算符来使名称空间 std 中的元素可用。

程序清单 9.12　namesp.cpp

```
// namesp.cpp -- namespaces
#include <iostream>
#include "namesp.h"

namespace pers
{
    using std::cout;
    using std::cin;
    void getPerson(Person & rp)
    {
        cout << "Enter first name: ";
        cin >> rp.fname;
        cout << "Enter last name: ";
        cin >> rp.lname;
    }
    void showPerson(const Person & rp)
    {
        std::cout << rp.lname << ", " << rp.fname;
    }
}

namespace debts
{
    void getDebt(Debt & rd)
    {
        getPerson(rd.name);
        std::cout << "Enter debt: ";
        std::cin >> rd.amount;
    }
    void showDebt(const Debt & rd)
    {
        showPerson(rd.name);
        std::cout <<": $" << rd.amount << std::endl;
    }
    double sumDebts(const Debt ar[], int n)
```

```
    {
        double total = 0;
        for (int i = 0; i < n; i++)
            total += ar[i].amount;
        return total;
    }
}
```

最后，该程序的第三个文件（参见程序清单 9.13）是一个源代码文件，它使用了名称空间中声明和定义的结构和函数。程序清单 9.13 演示了多种使名称空间标识符可用的方法。

程序清单 9.13　namessp.cpp

```
// usenmsp.cpp -- using namespaces
#include <iostream>
#include "namesp.h"
void other(void);
void another(void);
int main(void)
{
    using debts::Debt;

    using debts::showDebt;
    Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
    showDebt(golf);
    other();
    another();
    return 0;
}

void other(void)
{
    using std::cout;
    using std::endl;
    using namespace debts;
    Person dg = {"Doodles", "Glister"};
    showPerson(dg);
    cout << endl;
    Debt zippy[3];
    int i;
    for (i = 0; i < 3; i++)
        getDebt(zippy[i]);

    for (i = 0; i < 3; i++)
        showDebt(zippy[i]);
    cout << "Total debt: $" << sumDebts(zippy, 3) << endl;
    return;
}

void another(void)
{
    using pers::Person;
    Person collector = { "Milo", "Rightshift" };
    pers::showPerson(collector);
    std::cout << std::endl;
}
```

在程序清单 9.13 中，main( )函数首先使用了两个 using 声明：

```
using debts::Debt;        // makes the Debt structure definition available
using debts::showDebt;    // makes the showDebt function available
```

注意，using 声明只使用了名称，例如，第二个 using 声明没有描述 showDebt 的返回类型或函数特征标，而只给出了名称；因此，如果函数被重载，则一个 using 声明将导入所有的版本。另外，虽然 Debt 和 showDebt 都使用了 Person 类型，但不必导入任何 Person 名称，因为 debt 名称空间有一条包含 pers 名称空间的 using 编译指令。

接下来，other( )函数采用了一种不太好的方法，即使用一条 using 编译指令导入整个名称空间：

```
using namespace debts;   // make all debts and pers names available to other()
```

由于 debts 中的 using 编译指令导入了 pers 名称空间，因此 other( )函数可以使用 Person 类型和 showPerson( )函数。

最后，another( )函数使用 using 声明和作用域解析运算符来访问具体的名称：

```
using pers::Person;;
pers::showPerson(collector);
```

下面是程序清单 9.11～程序清单 9.13 组成的程序的运行情况：

```
Goatsniff, Benny: $120
Glister, Doodles
Enter first name: Arabella
Enter last name: Binx
Enter debt: 100
Enter first name: Cleve
Enter last name: Delaproux
Enter debt: 120
Enter first name: Eddie
Enter last name: Fiotox
Enter debt: 200
Binx, Arabella: $100
Delaproux, Cleve: $120
Fiotox, Eddie: $200
Total debt: $420
Rightshift, Milo
```

9.3.4 名称空间及其前途

随着程序员逐渐熟悉名称空间，将出现统一的编程理念。下面是当前的一些指导原则。

- 使用在已命名的名称空间中声明的变量，而不是使用外部全局变量。
- 使用在已命名的名称空间中声明的变量，而不是使用静态全局变量。
- 如果开发了一个函数库或类库，将其放在一个名称空间中。事实上，C++当前提倡将标准函数库放在名称空间 std 中，这种做法扩展到了来自 C 语言中的函数。例如，头文件 math.h 是与 C 语言兼容的，没有使用名称空间，但 C++头文件 cmath 应将各种数学库函数放在名称空间 std 中。实际上，并非所有的编译器都完成了这种过渡。
- 仅将编译指令 using 作为一种将旧代码转换为使用名称空间的权宜之计。
- 不要在头文件中使用 using 编译指令。首先，这样做掩盖了要让哪些名称可用；另外，包含头文件的顺序可能影响程序的行为。如果非要使用编译指令 using，应将其放在所有预处理器编译指令 #include 之后。
- 导入名称时，首选使用作用域解析运算符或 using 声明的方法。
- 对于 using 声明，首选将其作用域设置为局部而不是全局。

别忘了，使用名称空间的主旨是简化大型编程项目的管理工作。对于只有一个文件的简单程序，使用

using 编译指令并非什么大逆不道的事。

正如前面指出的，头文件名的变化反映了这些变化。老式头文件（如 iostream.h）没有使用名称空间，但新头文件 iostream 使用了 std 名称空间。

9.4　总结

C++鼓励程序员在开发程序时使用多个文件。一种有效的组织策略是，使用头文件来定义用户类型，为操纵用户类型的函数提供函数原型；并将函数定义放在一个独立的源代码文件中。头文件和源代码文件一起定义和实现了用户定义的类型及其使用方式。最后，将 main( )和其他使用这些函数的函数放在第三个文件中。

C++的存储方案决定了变量保留在内存中的时间（储存持续性）以及程序的哪一部分可以访问它（作用域和链接性）。自动变量是在代码块（如函数体或函数体中的代码块）中定义的变量，仅当程序执行到包含定义的代码块时，它们才存在，并且可见。自动变量可以通过使用存储类型说明符 register 或根本不使用说明符来声明，没有使用说明符时，变量将默认为自动的。register 说明符提示编译器，该变量的使用频率很高，但 C++11 摒弃了这种用法。

静态变量在整个程序执行期间都存在。对于在函数外面定义的变量，其所属文件中位于该变量的定义后面的所有函数都可以使用它（文件作用域），并可在程序的其他文件中使用（外部链接性）。另一个文件要使用这种变量，必须使用 extern 关键字来声明它。对于文件间共享的变量，应在一个文件中包含其定义声明（无需使用 extern，但如果同时进行初始化，也可使用它），并在其他文件中包含引用声明（使用 extern 且不初始化）。在函数的外面使用关键字 static 定义的变量的作用域为整个文件，但是不能用于其他文件（内部链接性）。在代码块中使用关键字 static 定义的变量被限制在该代码块内（局部作用域、无链接性），但在整个程序执行期间，它都一直存在并且保持原值。

在默认情况下，C++函数的链接性为外部，因此可在文件间共享；但使用关键字 static 限定的函数的链接性为内部的，被限制在定义它的文件中。

动态内存分配和释放是使用 new 和 delete 进行的，它使用自由存储区或堆来存储数据。调用 new 占用内存，而调用 delete 释放内存。程序使用指针来跟踪这些内存单元。

名称空间允许定义一个可在其中声明标识符的命名区域。这样做的目的是减少名称冲突，尤其当程序非常大，并使用多个厂商的代码时。可以通过使用作用域解析运算符、using 声明或 using 编译指令，来使名称空间中的标识符可用。

9.5　复习题

1．对于下面的情况，应使用哪种存储方案？

a．homer 是函数的形参。

b．secret 变量由两个文件共享。

c．topsecret 变量由一个文件中的所有函数共享，但对于其他文件来说是隐藏的。

D．beencalled 记录包含它的函数被调用的次数。

2．using 声明和 using 编译指令之间有何区别？

3．重新编写下面的代码，使其不使用 using 声明和 using 编译指令。

```
#include <iostream>
using namespace std;
int main()
{
```

```
double x;
cout << "Enter value: ";
while (! (cin >> x) )
{
    cout << "Bad input. Please enter a number: ";
    cin.clear();
   while (cin.get() != '\n')
       continue;
}
cout << "Value = " << x << endl;
    return 0;
}
```

4．重新编写下面的代码，使之使用 using 声明，而不是 using 编译指令。

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Enter value: ";
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get() != '\n')
           continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5．在一个文件中调用 average(3, 6)函数时，它返回两个 int 参数的 int 平均值，在同一个程序的另一个文件中调用时，它返回两个 int 参数的 double 平均值。应如何实现？

6．下面的程序由两个文件组成，该程序显示什么内容？

```
// file1.cpp
#include <iostream>
using namespace std;
void other();
void another();
int x = 10;
int y;

int main()
{
    cout << x << endl;
    {
        int x = 4;
        cout << x << endl;
        cout << y << endl;
       }
       other();
       another();
       return 0;
    }

    void other()
```

```
{
    int y = 1;
    cout << "Other: " << x << ", " << y << endl;
}

// file 2.cpp
#include <iostream>
using namespace std;
extern int x;
namespace
{
    int y = -4;
}

void another()
{
    cout << "another(): " << x << ", " << y << endl;
}
```

7. 下面的代码将显示什么内容？

```
#include <iostream>
using namespace std;
void other();
namespace n1
{
    int x = 1;
}

namespace n2
{
    int x = 2;
}


int main()
{
   using namespace n1;
   cout << x << endl;
    {
        int x = 4;
        cout << x << ", " << n1::x << ", " << n2::x << endl;
    }
    using n2::x;
    cout << x << endl;
    other();
    return 0;
}

void other()
{
    using namespace n2;
    cout << x << endl;
    {
        int x = 4;
        cout << x << ", " << n1::x << ", " << n2::x << endl;
```

```
    }
    using n2::x;
    cout << x << endl;
}
```

9.6 编程练习

1. 下面是一个头文件：

```
// golf.h -- for pe9-1.cpp

const int Len = 40;
struct golf
{
    char fullname[Len];
    int handicap;
};

// non-interactive version:
//  function sets golf structure to provided name, handicap
//  using values passed as arguments to the function
void setgolf(golf & g, const char * name, int hc);

// interactive version:
//  function solicits name and handicap from user
//  and sets the members of g to the values entered
//  returns 1 if name is entered, 0 if name is empty string
int setgolf(golf & g);
// function resets handicap to new value
void handicap(golf & g, int hc);

// function displays contents of golf structure
void showgolf(const golf & g);
```

注意到 setgolf( )被重载，可以这样使用其第一个版本：

```
golf ann;
setgolf(ann, "Ann Birdfree", 24);
```

上述函数调用提供了存储在 ann 结构中的信息。可以这样使用其第二个版本：

```
golf andy;
setgolf(andy);
```

上述函数将提示用户输入姓名和等级，并将它们存储在 andy 结构中。这个函数可以（但是不一定必须）在内部使用第一个版本。

根据这个头文件，创建一个多文件程序。其中的一个文件名为 golf.cpp，它提供了与头文件中的原型匹配的函数定义；另一个文件应包含 main( )，并演示原型化函数的所有特性。例如，包含一个让用户输入的循环，并使用输入的数据来填充一个由 golf 结构组成的数组，数组被填满或用户将高尔夫选手的姓名设置为空字符串时，循环将结束。main( )函数只使用头文件中原型化的函数来访问 golf 结构。

2. 修改程序清单 9.9：用 string 对象代替字符数组。这样，该程序将不再需要检查输入的字符串是否过长，同时可以将输入字符串同字符串“”进行比较，以判断是否为空行。

3. 下面是一个结构声明：

```
struct chaff
{
    char dross[20];
    int slag;
};
```

编写一个程序，使用定位 new 运算符将一个包含两个这种结构的数组放在一个缓冲区中。然后，给结构的成员赋值（对于 char 数组，使用函数 strcpy( )），并使用一个循环来显示内容。一种方法是像程序清单 9.10 那样将一个静态数组用作缓冲区；另一种方法是使用常规 new 运算符来分配缓冲区。

4. 请基于下面这个名称空间编写一个由 3 个文件组成的程序：

```
namespace SALES
{
    const int QUARTERS = 4;
    struct Sales
    {
        double sales[QUARTERS];
        double average;
        double max;
        double min;
    };
    // copies the lesser of 4 or n items from the array ar
    // to the sales member of s and computes and stores the
    // average, maximum, and minimum values of the entered items;
    // remaining elements of sales, if any, set to 0
    void setSales(Sales & s, const double ar[], int n);
    // gathers sales for 4 quarters interactively, stores them
    // in the sales member of s and computes and stores the
    // average, maximum, and minimum values
    void setSales(Sales & s);
    // display all information in structure s
    void showSales(const Sales & s);
}
```

第一个文件是一个头文件，其中包含名称空间；第二个文件是一个源代码文件，它对这个名称空间进行扩展，以提供这三个函数的定义；第三个文件声明两个 Sales 对象，并使用 setSales( )的交互式版本为一个结构提供值，然后使用 setSales( )的非交互式版本为另一个结构提供值。另外它还使用 showSales( )来显示这两个结构的内容。

第 10 章　对象和类

本章内容包括：

- 过程性编程和面向对象编程。
- 类概念。
- 如何定义和实现类。
- 公有类访问和私有类访问。
- 类的数据成员。
- 类方法（类函数成员）。
- 创建和使用类对象。
- 类的构造函数和析构函数。
- const 成员函数。
- this 指针。
- 创建对象数组。
- 类作用域。
- 抽象数据类型。

面向对象编程（OOP）是一种特殊的、设计程序的概念性方法，C++通过一些特性改进了 C 语言，使得应用这种方法更容易。下面是最重要的 OOP 特性：

- 抽象；
- 封装和数据隐藏；
- 多态；
- 继承；
- 代码的可重用性。

为了实现这些特性并将它们组合在一起，C++所做的最重要的改进是提供了类。本章首先介绍类，将解释抽象、封装、数据隐藏，并演示类是如何实现这些特性的。本章还将讨论如何定义类、如何为类提供公有部分和私有部分以及如何创建使用类数据的成员函数。另外，还将介绍构造函数和析构函数，它们是特殊的成员函数，用于创建和删除属于当前类的对象。最后介绍 this 指针，对于有些类编程而言，它是至关重要的。后面的章节还将把讨论扩展到运算符重载（另一种多态）和继承，它们是代码重用的基础。

10.1　过程性编程和面向对象编程

虽然本书前面偶尔探讨过 OOP 在编程方面的前景，但讨论的更多的还是诸如 C、Pascal 和 BASIC 等语言的标准过程性方法。下面来看一个例子，它揭示了 OOP 的观点与过程性编程的差别。

Genre Giants 垒球队的一名新成员被要求记录球队的统计数据。很自然，会求助于计算机来完成这项工作。如果是一位过程性程序员，可能会这样考虑：

我要输入每名选手的姓名、击球次数、击中次数、命中率（命中率指的是选手正式的击球次数除以击中次数；当选手在垒上或被罚出局时，击球停止，但某些情况不计作正式击球次数，如选手走步时）以及其他重要的基本统计数据。之所以使用计算机，是为了简化工作，因此让它来计算某些数据，如命中率。另外，我还希望程序能够显示这些结果。应如何组织呢？我想我能正确地完成这项工作，并使用了函数。是的，我让 main( )调用一个函数来获取输入，调用另一个函数来进行计算，然后再调用第三个函数来显示结果。那么，获得下一场比赛的数据后，又该做什么呢？我不想再从头开始，可以添加一个函数来更新统计数据。可能需要在 main( )中提供一个菜单，选择是输入、计算、更新还是显示数据。则如何表示这些数据呢？可以用一个字符串数组来存储选手的姓名，用另一个数组存储每一位选手的击球数，再用一个数组存储击中数目等等。这种方法太不灵活了，可以设计一个结构来存储每位选手的所有信息，然后用这种结构组成的数组来表示整个球队。

总之，采用过程性编程方法时，首先考虑要遵循的步骤，然后考虑如何表示这些数据（并不需要程序一直运行，用户可能希望能够将数据存储在一个文件中，然后从这个文件中读取数据）。

如果换成一位 OOP 程序员，又将如何呢？首先考虑数据——不仅要考虑如何表示数据，还要考虑如何使用数据：

我要跟踪的是什么？当然是选手。因此要有一个对象表示整个选手的各个方面（而不仅仅是命中率或击球次数）。是的，这将是基本数据单元——一个表示选手的姓名和统计数据的对象。我需要一些处理该对象的方法。首先需要一种将基本信息加入到该单元中的方法；其次，计算机应计算一些东西，如命中率，因此需要添加一些执行计算的方法。程序应自动完成这些计算，而无需用户干涉。另外，还需要一些更新和显示信息的方法。所以，用户与数据交互的方式有三种：初始化、更新和报告——这就是用户接口。

总之，采用 OOP 方法时，首先从用户的角度考虑对象——描述对象所需的数据以及描述用户与数据交互所需的操作。完成对接口的描述后，需要确定如何实现接口和数据存储。最后，使用新的设计方案创建出程序。

10.2　抽象和类

生活中充满复杂性，处理复杂性的方法之一是简化和抽象。人的身体是由无数个原子组成的，而一些学者认为人的思想是由半自主的主体组成的。但将人自己看作一个实体将简单得多。在计算中，为了根据信息与用户之间的接口来表示它，抽象是至关重要的。也就是说，将问题的本质特征抽象出来，并根据特征来描述解决方案。在垒球统计数据示例中，接口描述了用户如何初始化、更新和显示数据。抽象是通往用户定义类型的捷径，在 C++中，用户定义类型指的是实现抽象接口的类设计。

10.2.1　类型是什么

我们来看看是什么构成了类型。例如，讨厌鬼是什么？受流行的固定模式影响，可能会指出讨厌鬼的一些外表特点：胖、戴黑宽边眼镜、兜里插满钢笔等。稍加思索后，又可能觉得从行为上定义讨厌鬼可能更合适，如他（或她）是如何应对尴尬的社交场面的。如果将这种类比扩展到过程性语言（如 C 语言），我们得到类似的情形。首先，倾向于根据数据的外观（在内存中如何存储）来考虑数据类型。例如，char 占用 1 个字节的内存，而 double 通常占用 8 个字节的内存。但是稍加思索就会发现，也可以根据要对它执行的操作来定义数据类型。例如，int 类型可以使用所有的算术运算，可对整数执行加、减、乘、除运算，还可以对它们使用求模运算符（%）。

而指针需要的内存数量很可能与 int 相同，甚至可能在内部被表示为整数。但不能对指针执行与整数相同的运算。例如，不能将两个指针相乘，这种运算没有意义的，因此 C++没有实现这种运算。因此，将

变量声明为 int 或 float 指针时，不仅仅是分配内存，还规定了可对变量执行的操作。总之，指定基本类型完成了三项工作：

- 决定数据对象需要的内存数量；
- 决定如何解释内存中的位（long 和 float 在内存中占用的位数相同，但将它们转换为数值的方法不同）；
- 决定可使用数据对象执行的操作或方法。

对于内置类型来说，有关操作的信息被内置到编译器中。但在 C++中定义用户自定义的类型时，必须自己提供这些信息。付出这些劳动换来了根据实际需要定制新数据类型的强大功能和灵活性。

10.2.2 C++中的类

类是一种将抽象转换为用户定义类型的 C++工具，它将数据表示和操纵数据的方法组合成一个整洁的包。下面来看一个表示股票的类。

首先，必须考虑如何表示股票。可以将一股作为基本单元，定义一个表示一股股票的类。然而，这意味着需要 100 个对象才能表示 100 股，这不现实。相反，可以将某人当前持有的某种股票作为一个基本单元，数据表示中包含他持有的股票数量。一种比较现实的方法是，必须记录最初购买价格和购买日期（用于计算纳税）等内容。另外，还必须管理诸如如拆股等事件。首次定义类就考虑这么多因素有些困难，因此我们对其进行简化。具体地说，应该将可执行的操作限制为：

- 获得股票；
- 增持；
- 卖出股票；
- 更新股票价格；
- 显示关于所持股票的信息。

可以根据上述清单定义 stock 类的公有接口（如果您有兴趣，还可以添加其他特性）。为支持该接口，需要存储一些信息。我们再次进行简化。例如，不考虑标准的美式股票计价方式（八分之一美元的倍数。显然，纽约证券交易所一定看到过本书以前的版本中关于简化的论述，因为它已经决定将系统转换为书中采用的方式）。我们将存储下面的信息：

- 公司名称；
- 所持股票的数量；
- 每股的价格；
- 股票总值。

接下来定义类。一般来说，类规范由两个部分组成。

- 类声明：以数据成员的方式描述数据部分，以成员函数（被称为方法）的方式描述公有接口。
- 类方法定义：描述如何实现类成员函数。

简单地说，类声明提供了类的蓝图，而方法定义则提供了细节。

什么是接口

接口是一个共享框架，供两个系统（如在计算机和打印机之间或者用户或计算机程序之间）交互时使用；例如，用户可能是您，而程序可能是字处理器。使用字处理器时，您不能直接将脑子中想到的词传输到计算机内存中，而必须同程序提供的接口交互。您敲打键盘时，计算机将字符显示到屏幕上；您移动鼠标时，计算机移动屏幕上的光标；您无意间单击鼠标时，计算机对您输入的段落进行奇怪的处理。程序接口将您的意图转换为存储在计算机中的具体信息。

对于类，我们说公共接口。在这里，公众（public）是使用类的程序，交互系统由类对象组成，而接口由编写类的人提供的方法组成。接口让程序员能够编写与类对象交互的代码，从而让程序能够使用类对象。例如，要计算 string 对象中包含多少个字符，您无需打开对象，而只需使用 string 类提供的 size( )方法。类设计禁止公共用户直接访问类，但公众可以使用方法 size( )。方法 size( )是用户和 string 类对象之间的公共

接口的组成部分。通常，方法 getline( )是 istream 类的公共接口的组成部分，使用 cin 的程序不是直接与 cin 对象内部交互来读取一行输入，而是使用 getline( )。

如果希望更人性化，不要将使用类的程序视为公共用户，而将编写程序的人视为公共用户。然而，要使用某个类，必须了解其公共接口；要编写类，必须创建其公共接口。

为开发一个类并编写一个使用它的程序，需要完成多个步骤。这里将开发过程分成多个阶段，而不是一次性完成。通常，C++程序员将接口（类定义）放在头文件中，并将实现（类方法的代码）放在源代码文件中。这里采用这种典型做法。程序清单 10.1 是第一个阶段的代码，它是 Stock 类的类声明。这个文件按第 9 章介绍的那样，使用了#ifndef 等来访问多次包含同一个文件。

为帮助识别类，本书遵循一种常见但不通用的约定——将类名首字母大写。您将发现，程序清单 10.1 看起来就像一个结构声明，只是还包括成员函数、公有部分和私有部分等内容。稍后将对该声明进行改进（所以不要将它用作模型），但先来看一看该定义的工作方式。

程序清单 10.1　stock00.h

```
// stock00.h -- Stock class interface
// version 00
#ifndef STOCK00_H_
#define STOCK00_H_

#include <string>

class Stock  // class declaration
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const std::string & co, long n, double pr);
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};    // note semicolon at the end

#endif
```

稍后将详细介绍类的细节，但先看一下更通用的特性。首先，C++关键字 class 指出这些代码定义了一个类设计（不同于在模板参数中，在这里，关键字 class 和 typename 不是同义词，不能使用 typename 代替 class）。这种语法指出，Stock 是这个新类的类型名。该声明让我们能够声明 Stock 类型的变量——称为对象或实例。每个对象都表示一支股票。例如，下面的声明创建两个 Stock 对象，它们分别名为 sally 和 solly：

```
Stock sally;
Stock solly;
```

例如，sally 对象可以表示 Sally 持有的某公司股票。

接下来，要存储的数据以类数据成员（如 company 和 shares）的形式出现。例如，sally 的 company 成员存储了公司名称，share 成员存储了 Sally 持有的股票数量，share_val 成员存储了每股的价格，total_val 成员存储了股票总价格。同样，要执行的操作以类函数成员（方法，如 sell( )和 update( )）的形式出现。成员函数可以就地定义（如 set_tot( )），也可以用原型表示（如其他成员函数）。其他成员

函数的完整定义稍后将介绍，它们包含在实现文件中；但对于描述函数接口而言，原型足够了。将数据和方法组合成一个单元是类最吸引人的特性。有了这种设计，创建 Stock 对象时，将自动制定使用对象的规则。

istream 和 ostream 类有成员函数，如 get( )和 getline( )，而 Stock 类声明中的函数原型说明了成员函数是如何建立的。例如，头文件 iostream 将 getline( )的原型放在 istream 类的声明中。

1. 访问控制

关键字 private 和 public 也是新的，它们描述了对类成员的访问控制。使用类对象的程序都可以直接访问公有部分，但只能通过公有成员函数（或友元函数，参见第 11 章）来访问对象的私有成员。例如，要修改 Stock 类的 shares 成员，只能通过 Stock 的成员函数。因此，公有成员函数是程序和对象的私有成员之间的桥梁，提供了对象和程序之间的接口。防止程序直接访问数据被称为数据隐藏（参见图 10.1）。C++还提供了第三个访问控制关键字 protected，第 13 章介绍类继承时将讨论该关键字。

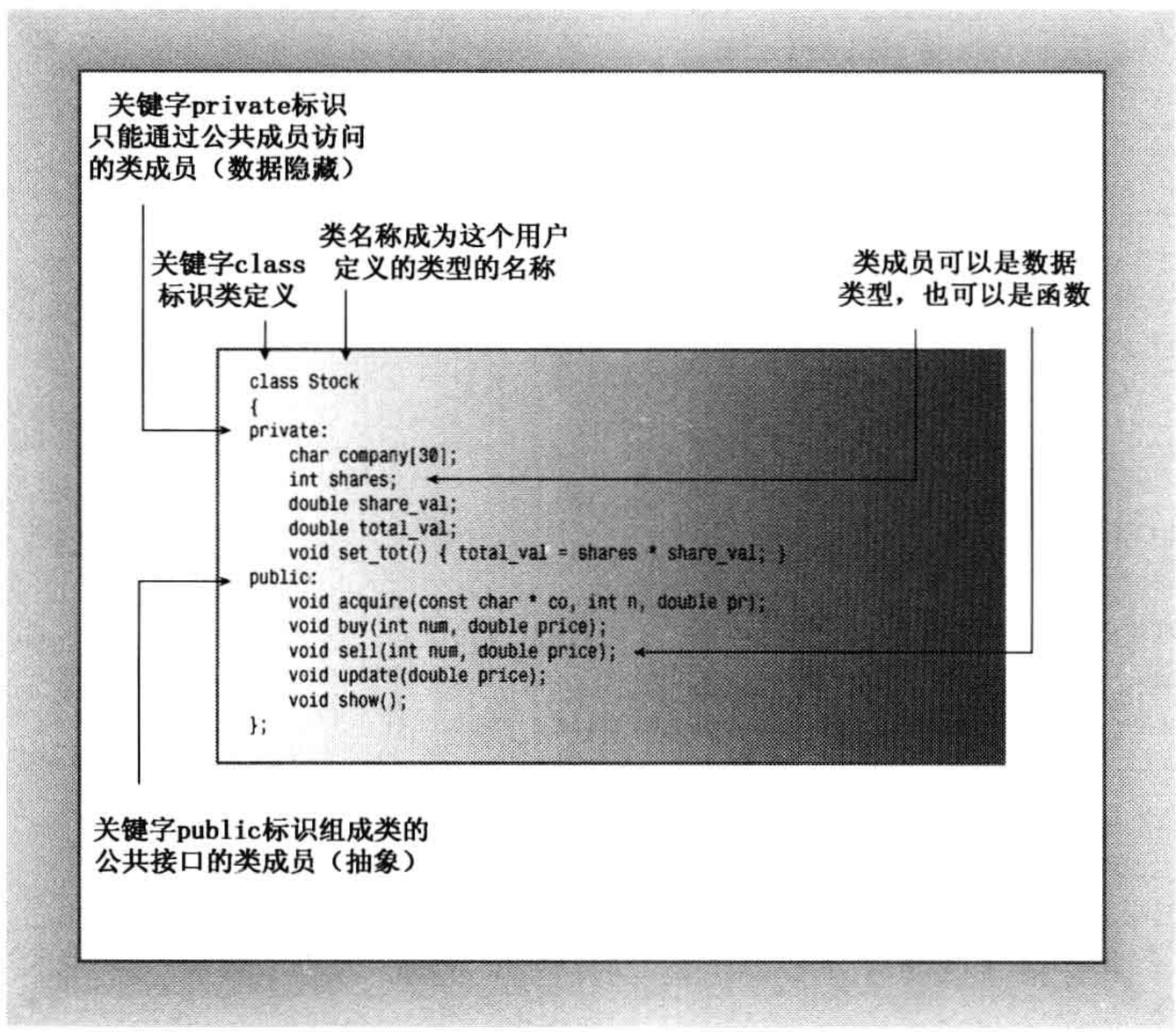


图 10.1 Stock 类

类设计尽可能将公有接口与实现细节分开。公有接口表示设计的抽象组件。将实现细节放在一起并将它们与抽象分开被称为封装。数据隐藏（将数据放在类的私有部分中）是一种封装，将实现的细节隐藏在私有部分中，就像 Stock 类对 set_tot( )所做的那样，也是一种封装。封装的另一个例子是，将类函数定义和类声明放在不同的文件中。

OOP 和 C++

OOP 是一种编程风格，从某种程度说，它用于任何一种语言中。当然，可以将 OOP 思想融合到常规的 C 语言程序中。例如，在第 9 章的一个示例（程序清单 9.1、程序清单 9.2、程序清单 9.3）中，头文件中包含结构原型和操纵该结构的函数的原型，便是这样的例子。因此，main( )函数只需定义这个结构类型的变量，并使用相关函数处理这些变量即可；main( )不直接访问结构成员。实际上，该示例定义了一种抽象类型，它将存储格式和函数原型置于头文件中，对 main( )隐藏了实际的数据表示。然而，C++中包括了许多专门用来实现 OOP 方法的特性，因此它使程序员更进一步。首先，将数据表示和函数原型放在一个类声明中（而不是放在一个文件中），通过将所有内容放在一个类声明中，来使描述成为一个整体。其次，让

数据表示成为私有，使得数据只能被授权的函数访问。在 C 语言的例子中，如果 main( )直接访问了结构成员，则违反了 OOP 的精神，但没有违反 C 语言的规则。然而，试图直接访问 Stock 对象的 shares 成员便违反了 C++语言的规则，编译器将捕获这种错误。

数据隐藏不仅可以防止直接访问数据，还让开发者（类的用户）无需了解数据是如何被表示的。例如，show( )成员将显示某支股票的总价格（还有其他内容），这个值可以存储在对象中（上述代码正是这样做的），也可以在需要时通过计算得到。从使用类的角度看，使用哪种方法没有什么区别。所需要知道的只是各种成员函数的功能；也就是说，需要知道成员函数接受什么样的参数以及返回什么类型的值。原则是将实现细节从接口设计中分离出来。如果以后找到了更好的、实现数据表示或成员函数细节的方法，可以对这些细节进行修改，而无需修改程序接口，这使程序维护起来更容易。

2. 控制对成员的访问：公有还是私有

无论类成员是数据成员还是成员函数，都可以在类的公有部分或私有部分中声明它。但由于隐藏数据是 OOP 主要的目标之一，因此数据项通常放在私有部分，组成类接口的成员函数放在公有部分；否则，就无法从程序中调用这些函数。正如 Stock 声明所表明的，也可以把成员函数放在私有部分中。不能直接从程序中调用这种函数，但公有方法却可以使用它们。通常，程序员使用私有成员函数来处理不属于公有接口的实现细节。

不必在类声明中使用关键字 private，因为这是类对象的默认访问控制：

```
class World
{
    float mass;          // private by default
    char name[20];       // private by default
public:
    void tellall(void);
    ...
};
```

然而，为强调数据隐藏的概念，本书显式地使用了 private。

类和结构

类描述看上去很像是包含成员函数以及 public 和 private 可见性标签的结构声明。实际上，C++对结构进行了扩展，使之具有与类相同的特性。它们之间唯一的区别是，结构的默认访问类型是 public，而类为 private。C++程序员通常使用类来实现类描述，而把结构限制为只表示纯粹的数据对象（常被称为普通老式数据（POD，Plain Old Data）结构）。

10.2.3　实现类成员函数

还需要创建类描述的第二部分：为那些由类声明中的原型表示的成员函数提供代码。成员函数定义与常规函数定义非常相似，它们有函数头和函数体，也可以有返回类型和参数。但是它们还有两个特殊的特征：

- 定义成员函数时，使用作用域解析运算符（::）来标识函数所属的类；
- 类方法可以访问类的 private 组件。

首先，成员函数的函数头使用作用域运算符解析（::）来指出函数所属的类。例如，update( )成员函数的函数头如下：

```
void Stock::update(double price)
```

这种表示法意味着我们定义的 update( )函数是 Stock 类的成员。这不仅将 update( )标识为成员函数，还意味着我们可以将另一个类的成员函数也命名为 update( )。例如，Buffoon( )类的 update( )函数的函数头如下：

```
void Buffoon::update()
```

因此，作用域解析运算符确定了方法定义对应的类的身份。我们说，标识符 update( )具有类作用域（class scope）。Stock 类的其他成员函数不必使用作用域解析运算符，就可以使用 update( )方法，这是因为它们属于同一个类，因此 update( )是可见的。然而，在类声明和方法定义之外使用 update( )时，需要采取特殊的措施，稍后将作介绍。

类方法的完整名称中包括类名。我们说，Stock::update( )是函数的限定名（qualified name）；而简单的 update( )是全名的缩写（非限定名，unqualified name），它只能在类作用域中使用。

方法的第二个特点是，方法可以访问类的私有成员。例如，show( )方法可以使用这样的代码：

```
std::cout << "Company: " << company
          << "  Shares: " << shares << endl
          << "  Share Price: $" << share_val
          << "  Total Worth: $" << total_val << endl;
```

其中，company、shares 等都是 Stock 类的私有数据成员。如果试图使用非成员函数访问这些数据成员，编译器禁止这样做（但第 11 章中将介绍的友元函数例外）。

了解这两点后，就可以实现类方法了，如程序清单 10.2 所示。这里将它们放在了一个独立的实现文件中，因此需要包含头文件 stock00.h，让编译器能够访问类定义。为让您获得更多有关名称空间的经验，在有些方法中使用了限定符 std::，在其他方法中则使用了 using 声明。

程序清单 10.2　stock00.cpp

```
// stock00.cpp -- implementing the Stock class
// version 00
#include <iostream>
#include "stock00.h"

void Stock::acquire(const std::string & co, long n, double pr)
{
    company = co;
    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(long num, double price)
{
    if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
```

```
void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    std::cout << "Company: " << company
              << "  Shares: " << shares << '\n'
              << "  Share Price: $" << share_val
              << "  Total Worth: $" << total_val << '\n';
}
```

1. 成员函数说明

acquire( )函数管理对某个公司股票的首次购买，而 buy( )和 sell( )管理增加或减少持有的股票。方法 buy( )和 sell( )确保买入或卖出的股数不为负。另外，如果用户试图卖出超过他持有的股票数量，则 sell( )函数将结束这次交易。这种使数据私有并限于对公有函数访问的技术允许我们能够控制数据如何被使用；在这个例子中，它允许我们加入这些安全防护措施，避免不适当的交易。

4 个成员函数设置或重新设置了 total_val 成员值。这个类并非将计算代码编写 4 次，而是让每个函数都调用 set_tot( )函数。由于 set_tot( )只是实现代码的一种方式，而不是公有接口的组成部分，因此这个类将其声明为私有成员函数（即编写这个类的人可以使用它，但编写代码来使用这个类的人不能使用）。如果计算代码很长，则这种方法还可以省去许多输入代码的工作，并可节省空间。然而，这种方法的主要价值在于，通过使用函数调用，而不是每次重新输入计算代码，可以确保执行的计算完全相同。另外，如果必须修订计算代码（在这个例子中，这种可能性不大），则只需在一个地方进行修改即可。

2. 内联方法

其定义位于类声明中的函数都将自动成为内联函数，因此 Stock::set_tot( )是一个内联函数。类声明常将短小的成员函数作为内联函数，set_tot( )符合这样的要求。

如果愿意，也可以在类声明之外定义成员函数，并使其成为内联函数。为此，只需在类实现部分中定义函数时使用 inline 限定符即可：

```
class Stock
{
private:
    ...
    void set_tot();  // definition kept separate
public:
    ...
};

inline void Stock::set_tot()  // use inline in definition
{
    total_val = shares * share_val;
}
```

内联函数的特殊规则要求在每个使用它们的文件中都对其进行定义。确保内联定义对多文件程序中的所有文件都可用的、最简便的方法是：将内联定义放在定义类的头文件中（有些开发系统包含智能链接程序，允许将内联定义放在一个独立的实现文件）。

顺便说一句，根据改写规则（rewrite rule），在类声明中定义方法等同于用原型替换方法定义，然后在类声明的后面将定义改写为内联函数。也就是说，程序清单 10.1 中 set_tot( )的内联定义与上述代码（定义紧跟在类声明之后）是等价的。

3. 方法使用哪个对象

下面介绍使用对象时最重要的一个方面：如何将类方法应用于对象。下面的代码使用了一个对象的 shares 成员：

```
shares += num;
```

是哪个对象呢？问得好！要回答这个问题，首先来看看如何创建对象。最简单的方式是声明类变量：

```
Stock kate, joe;
```

这将创建两个 Stock 类对象，一个为 kate，另一个为 joe。

接下来，看看如何使用对象的成员函数。和使用结构成员一样，通过成员运算符：

```
kate.show();     // the kate object calls the member function
joe.show();      // the joe object calls the member function
```

第 1 条语句调用 kate 对象的 show( )成员。这意味着 show( )方法将把 shares 解释为 kate.shares，将 share_vla 解释为 kate.share_val。同样，函数调用 joe.show( )使 show( )方法将 shares 和 share_val 分别解释为 joe.share 和 joe.share_val。

注意：调用成员函数时，它将使用被用来调用它的对象的数据成员。

同样，函数调用 kate.sell( )在调用 set_tot( )函数时，相当于调用 kate.set_tot( )，这样该函数将使用 kate 对象的数据。

所创建的每个新对象都有自己的存储空间，用于存储其内部变量和类成员；但同一个类的所有对象共享同一组类方法，即每种方法只有一个副本。例如，假设 kate 和 joe 都是 Stock 对象，则 kate.shares 将占据一个内存块，而 joe.shares 占用另一个内存块，但 kate.show( )和 joe.show( )都调用同一个方法，也就是说，它们将执行同一个代码块，只是将这些代码用于不同的数据。在 OOP 中，调用成员函数被称为发送消息，因此将同样的消息发送给两个不同的对象将调用同一个方法，但该方法被用于两个不同的对象（参见图 10.2）。

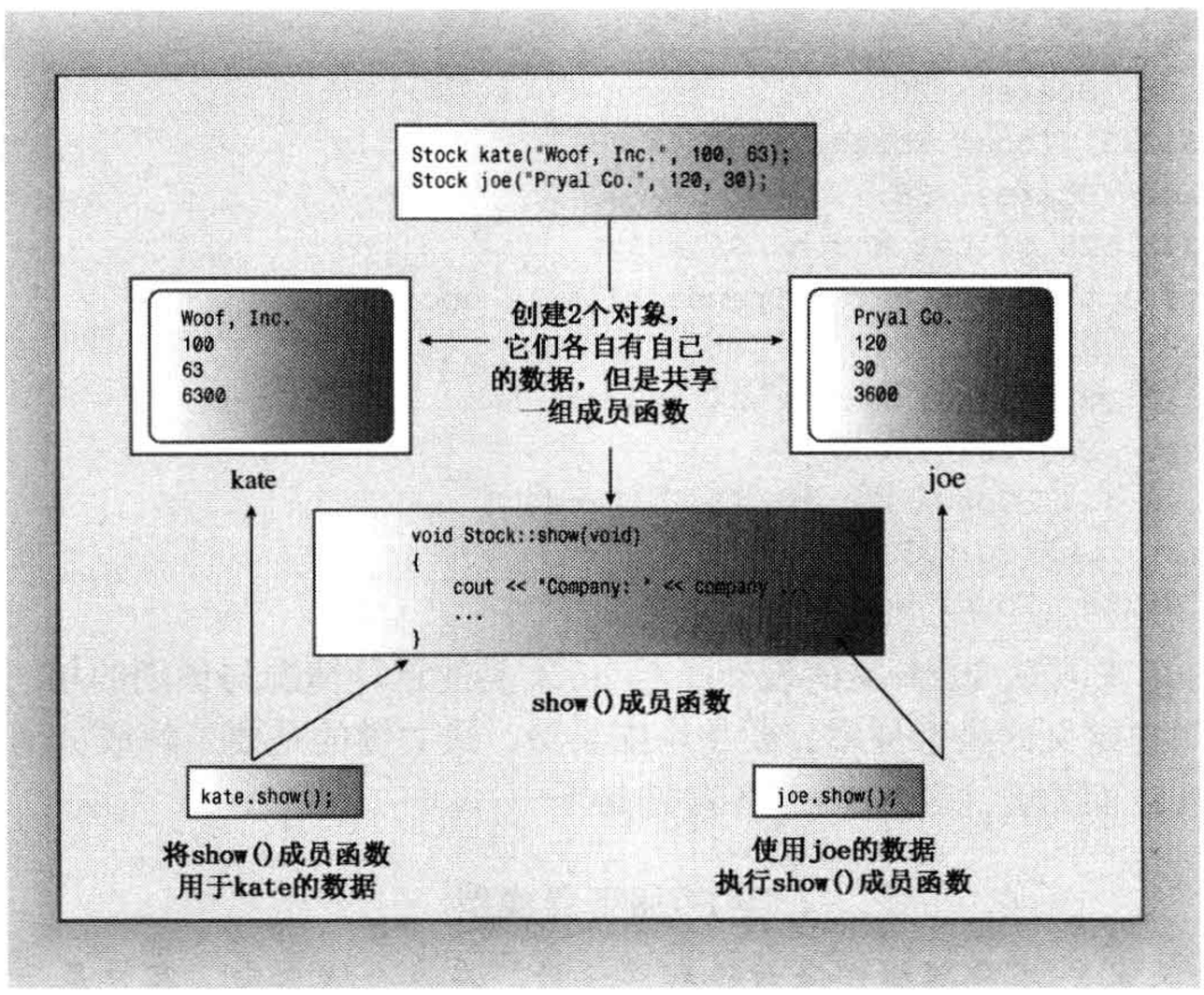


图 10.2　对象、数据和成员函数

10.2.4　使用类

知道如何定义类及其方法后，来创建一个程序，它创建并使用类对象。C++的目标是使得使用类与使用基本的内置类型（如 int 和 char）尽可能相同。要创建类对象，可以声明类变量，也可以使用 new 为类对象分配存储空间。可以将对象作为函数的参数和返回值，也可以将一个对象赋给另一个。C++提供了一些工具，可用于初始化对象、让 cin 和 cout 识别对象，甚至在相似的类对象之间进行自动类型转换。虽然要做到这些工作还需要一段时间，但可以先从比较简单的属性着手。实际上，您已经知道如何声明类对象和调用成员函数。程序清单 10.3 提供了一个使用上述接口和实现文件的程序，它创建了一个名为 fluffy_the_cat 的 Stock 对象。该程序非常简单，但确实测试了这个类的特性。要编译该程序，可使用用于多文件程序的方法，这在第 1 章和第 9 章介绍过。具体地说，将其与 stock00.cpp 一起编译，并确保 stock00.h 位于当前文件夹中。

程序清单 10.3　usestock0.cpp

```
// usestck0.cpp -- the client program
// compile with stock00.cpp
#include <iostream>
#include "stock00.h"
int main()
{
    Stock fluffy_the_cat;
    fluffy_the_cat.acquire("NanoSmart", 20, 12.50);
    fluffy_the_cat.show();
    fluffy_the_cat.buy(15, 18.125);
    fluffy_the_cat.show();
    fluffy_the_cat.sell(400, 20.00);
    fluffy_the_cat.show();
    fluffy_the_cat.buy(300000,40.125);
    fluffy_the_cat.show();
    fluffy_the_cat.sell(300000,0.125);
    fluffy_the_cat.show();
    return 0;
}
```

下面是该程序的输出：

```
Company: NanoSmart  Shares: 20
  Share Price: $12.5  Total Worth: $250
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.375
You can't sell more than you have! Transaction is aborted.
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.375
Company: NanoSmart  Shares: 300035
  Share Price: $40.125  Total Worth: $1.20389e+007
Company: NanoSmart  Shares: 35
  Share Price: $0.125  Total Worth: $4.375
```

注意，main( )只是用来测试 Stock 类的设计。当 Stock 类的运行情况与预期的相同后，便可以在其他程序中将 Stock 类作为用户定义的类型使用。要使用新类型，最关键的是要了解成员函数的功能，而不必考虑其实现细节。请参阅后面的旁注“客户/服务器模型”。

客户/服务器模型

OOP 程序员常依照客户/服务器模型来讨论程序设计。在这个概念中，客户是使用类的程序。类声明（包括类方法）构成了服务器，它是程序可以使用的资源。客户只能通过以公有方式定义的接口使用服务器，这意味着客户（客户程序员）唯一的责任是了解该接口。服务器（服务器设计人员）的责任是确保服务器根据该接口可靠并准确地执行。服务器设计人员只能修改类设计的实现细节，而不能修改接口。这样程序员独立地对客户和服务器进行改进，对服务器的修改不会客户的行为造成意外的影响。

10.2.5 修改实现

在前面的程序输出中，可能有一个方面让您恼火——数字的格式不一致。现在可以改进实现，但保持接口不变。ostream 类包含一些可用于控制格式的成员函数。这里不做太详细的探索，只需像在程序清单 8.8 那样使用方法 setf()，便可避免科学计数法：

```
std::cout.setf(std::ios_base::fixed, std::ios_base::floatfield);
```

这设置了 cout 对象的一个标记，命令 cout 使用定点表示法。同样，下面的语句导致 cout 在使用定点表示法时，显示三位小数：

```
std::cout.precision(3);
```

第 17 章将介绍这方面的更多细节。

可在方法 show()中使用这些工具来控制格式，但还有一点需要考虑。修改方法的实现时，不应影响客户程序的其他部分。上述格式修改将一直有效，直到您再次修改，因此它们可能影响客户程序中的后续输出。因此，show()应重置格式信息，使其恢复到自己被调用前的状态。为此，可以像程序清单 8.8 那样，使用返回的值：

```
std::streamsize prec =
    std::cout.precision(3);  // save preceding value for precision
...
std::cout.precision(prec);           // reset to old value

// store original flags
std::ios_base::fmtflags orig = std::cout.setf(std::ios_base::fixed);
...
// reset to stored values
std::cout.setf(orig, std::ios_base::floatfield);
```

您可能还记得，fmtflags 是在 ios_base 类中定义的一种类型，而 ios_base 类又是在名称空间 std 中定义的，因此 orig 的类型名非常长。其次，orig 存储了所有的标记，而重置语句使用这些信息来重置 floatfield，

而 floatfield 包含定点表示法标记和科学表示法标记。第三，请不要过多考虑这里的细节。这里的要旨是，将修改限定在实现文件中，以免影响程序的其他方面。

根据上面的介绍，可在实现文件中将方法 show()的定义修改成如下所示：

```
void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // set format to #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);

    cout << "Company: " << company
        << "  Shares: " << shares << '\n';
    cout << "  Share Price: $" << share_val;
    // set format to #.##
    cout.precision(2);
    cout << "  Total Worth: $" << total_val << '\n';

    // restore original format
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}
```

完成上述修改后（保留头文件和客户文件不变），可重新编译该程序。该程序的输出将类似于下面这样：

```
Company: NanoSmart  Shares: 20
  Share Price: $12.500  Total Worth: $250.00
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.38
You can't sell more than you have! Transaction is aborted.
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.38
Company: NanoSmart  Shares: 300035
  Share Price: $40.125  Total Worth: $12038904.38
Company: NanoSmart  Shares: 35
  Share Price: $0.125  Total Worth: $4.38
```

10.2.6　小结

指定类设计的第一步是提供类声明。类声明类似结构声明，可以包括数据成员和函数成员。声明有私有部分，在其中声明的成员只能通过成员函数进行访问；声明还具有公有部分，在其中声明的成员可被使用类对象的程序直接访问。通常，数据成员被放在私有部分中，成员函数被放在公有部分中，因此典型的类声明的格式如下：

```
class  className
{
private:
    data member declarations
public:
    member function prototypes
};
```

公有部分的内容构成了设计的抽象部分——公有接口。将数据封装到私有部分中可以保护数据的完整性，这被称为数据隐藏。因此，C++通过类使得实现抽象、数据隐藏和封装等 OOP 特性很容易。

指定类设计的第二步是实现类成员函数。可以在类声明中提供完整的函数定义，而不是函数原型，但

是通常的做法是单独提供函数定义（除非函数很小）。在这种情况下，需要使用作用域解析运算符来指出成员函数属于哪个类。例如，假设 Bozo 有一个名为 Retort( )的成员函数，该函数返回 char 指针，则其函数头如下所示：

```
char * Bozo::Retort()
```

换句话来说，Retort( )不仅是一个 char *类型的函数，而是一个属于 Bozo 类的 char *函数。该函数的全名（或限定名）为 Bozo::Retort( )。而名称 Retort( )是限定名的缩写，只能在某些特定的环境中使用，如类方法的代码中。

另一种描述这种情况的方式是，名称 Retort 的作用域为整个类，因此在类声明和类方法之外使用该名称时，需要使用作用域解析运算符进行限定。

要创建对象（类的实例），只需将类名视为类型名即可：

```
Bozo bozetta;
```

这样做是可行的，因为类是用户定义的类型。

类成员函数（方法）可通过类对象来调用。为此，需要使用成员运算符句点：

```
cout << Bozetta.Retort();
```

这将调用 Retort( )成员函数，每当其中的代码引用某个数据成员时，该函数都将使用 bozetta 对象中相应成员的值。

10.3 类的构造函数和析构函数

对于 Stock 类，还有其他一些工作要做。应为类提供被称为构造函数和析构函数的标准函数。下面来看一看为什么需要这些函数以及如何使用这些函数。

C++的目标之一是让使用类对象就像使用标准类型一样，然而，到现在为止，本章提供的代码还不能让您像初始化 int 或结构那样来初始化 Stock 对象。也就是说，常规的初始化语法不适用于类型 Stock：

```
int year = 2001;                                    // valid initialization
struct thing
{
    char * pn;
    int m;
};
thing amabob = {"wodget", -23};                     // valid initialization
Stock hot = {"Sukie's Autos, Inc.", 200, 50.25};    // NO! compile error
```

不能像上面这样初始化 Stock 对象的原因在于，数据部分的访问状态是私有的，这意味着程序不能直接访问数据成员。您已经看到，程序只能通过成员函数来访问数据成员，因此需要设计合适的成员函数，才能成功地将对象初始化（如果使数据成员成为公有，而不是私有，就可以按刚才介绍的方法初始化类对象，但使数据成为公有的违背了类的一个主要初衷：数据隐藏）。

一般来说，最好是在创建对象时对它进行初始化。例如，请看下面的代码：

```
Stock gift;
gift.buy(10, 24.75);
```

就 Stock 类当前的实现而言，gift 对象的 company 成员是没有值的。类设计假设用户在调用任何其他成员函数之前调用 acquire( )，但无法强加这种假设。避开这种问题的方法之一是在创建对象时，自动对它进行初始化。为此，C++提供了一个特殊的成员函数——类构造函数，专门用于构造新对象、将值赋给它们的数据成员。更准确地说，C++为这些成员函数提供了名称和使用语法，而程序员需要提供方法定义。名称与类名相同。例如，Stock 类一个可能的构造函数是名为 Stock( )的成员函数。构造函数的

原型和函数头有一个有趣的特征——虽然没有返回值，但没有被声明为 void 类型。实际上，构造函数没有声明类型。

10.3.1　声明和定义构造函数

现在需要创建 Stock 的构造函数。由于需要为 Stock 对象提供 3 个值，因此应为构造函数提供 3 个参数。（第 4 个值，total_val 成员，是根据 shares 和 share_val 计算得到的，因此不必为构造函数提供这个值。）程序员可能只想设置 company 成员，而将其他值设置为 0；这可以使用默认参数来完成（参见第 8 章）。因此，原型如下所示：

```
// constructor prototype with some default arguments
Stock(const string & co, long n = 0, double pr = 0.0);
```

第一个参数是指向字符串的指针，该字符串用于初始化成员 company。n 和 pr 参数为 shares 和 share_val 成员提供值。注意，没有返回类型。原型位于类声明的公有部分。

下面是构造函数的一种可能定义：

```
// constructor definition
Stock::Stock(const string & co, long n, double pr)
{
company = co;

    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
```

上述代码和本章前面的函数 acquire( )相同。区别在于，程序声明对象时，将自动调用构造函数。

成员名和参数名

不熟悉构造函数的您会试图将类成员名称用作构造函数的参数名，如下所示：

```
// NO!
Stock::Stock(const string & company, long shares, double share_val)
{
...
}
```

这是错误的。构造函数的参数表示的不是类成员，而是赋给类成员的值。因此，参数名不能与类成员相同，否则最终的代码将是这样的：

```
shares = shares;
```

为避免这种混乱，一种常见的做法是在数据成员名中使用 m_前缀：

```
class Stock
{
private:
    string m_company;
    long m_shares;
    ...
```

另一种常见的做法是，在成员名中使用后缀_：

```
class Stock
{
private:
    string company_;
    long shares_;
    ...
```

无论采用哪种做法，都可在公有接口中在参数名中包含 company 和 shares。

10.3.2 使用构造函数

C++提供了两种使用构造函数来初始化对象的方式。第一种方式是显式地调用构造函数：

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

这将 food 对象的 company 成员设置为字符串“World Cabbage”，将 shares 成员设置为 250，依此类推。另一种方式是隐式地调用构造函数：

```
Stock garment("Furry Mason", 50, 2.5);
```

这种格式更紧凑，它与下面的显式调用等价：

```
Stock garment = Stock("Furry Mason", 50, 2.5));
```

每次创建类对象（甚至使用 new 动态分配内存）时，C++都使用类构造函数。下面是将构造函数与 new 一起使用的方法：

```
Stock *pstock = new Stock("Electroshock Games", 18, 19.0);
```

这条语句创建一个 Stock 对象，将其初始化为参数提供的值，并将该对象的地址赋给 pstock 指针。在这种情况下，对象没有名称，但可以使用指针来管理该对象。我们将在第 11 章进一步讨论对象指针。

构造函数的使用方式不同于其他类方法。一般来说，使用对象来调用方法：

```
stock1.show();  // stock1 object invokes show() method
```

但无法使用对象来调用构造函数，因为在构造函数构造出对象之前，对象是不存在的。因此构造函数被用来创建对象，而不能通过对象来调用。

10.3.3 默认构造函数

默认构造函数是在未提供显式初始值时，用来创建对象的构造函数。也就是说，它是用于下面这种声明的构造函数：

```
Stock fluffy_the_cat;  // uses the default constructor
```

程序清单 10.3 就是这样做的！这条语句管用的原因在于，如果没有提供任何构造函数，则 C++将自动提供默认构造函数。它是默认构造函数的隐式版本，不做任何工作。对于 Stock 类来说，默认构造函数可能如下：

```
Stock::Stock() { }
```

因此将创建 fluffy_the_cat 对象，但不初始化其成员，这和下面的语句创建 x，但没有提供值给它一样：

```
int x;
```

默认构造函数没有参数，因为声明中不包含值。

奇怪的是，当且仅当没有定义任何构造函数时，编译器才会提供默认构造函数。为类定义了构造函数后，程序员就必须为它提供默认构造函数。如果提供了非默认构造函数（如 Stock(const char * co, int n, double pr)），但没有提供默认构造函数，则下面的声明将出错：

```
Stock stock1;  // not possible with current constructor
```

这样做的原因可能是想禁止创建未初始化的对象。然而，如果要创建对象，而不显式地初始化，则必须定义一个不接受任何参数的默认构造函数。定义默认构造函数的方式有两种。一种是给已有构造函数的所有参数提供默认值：

```
Stock(const string & co = "Error", int n = 0, double pr = 0.0);
```

另一种方式是通过函数重载来定义另一个构造函数——一个没有参数的构造函数：

```
Stock();
```

由于只能有一个默认构造函数，因此不要同时采用这两种方式。实际上，通常应初始化所有的对象，以确保所有成员一开始就有已知的合理值。因此，用户定义的默认构造函数通常给所有成员提供隐式初始值。例如，下面是为 Stock 类定义的一个默认构造函数：

```
Stock::Stock()          // default constructor
{
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
```

提示：在设计类时，通常应提供对所有类成员做隐式初始化的默认构造函数。

使用上述任何一种方式（没有参数或所有参数都有默认值）创建了默认构造函数后，便可以声明对象变量，而不对它们进行显式初始化：

```
Stock first;                // calls default constructor implicitly
Stock first = Stock();      // calls it explicitly
Stock *prelief = new Stock; // calls it implicitly
```

然而，不要被非默认构造函数的隐式形式所误导：

```
Stock first("Concrete Conglomerate");      // calls constructor
Stock second();                            // declares a function
Stock third;                               // calls default constructor
```

第一个声明调用非默认构造函数，即接受参数的构造函数；第二个声明指出，second( )是一个返回 Stock 对象的函数。隐式地调用默认构造函数时，不要使用圆括号。

10.3.4　析构函数

用构造函数创建对象后，程序负责跟踪该对象，直到其过期为止。对象过期时，程序将自动调用一个特殊的成员函数，该函数的名称令人生畏——析构函数。析构函数完成清理工作，因此实际上很有用。例如，如果构造函数使用 new 来分配内存，则析构函数将使用 delete 来释放这些内存。Stock 的构造函数没有使用 new，因此析构函数实际上没有需要完成的任务。在这种情况下，只需让编译器生成一个什么要不做的隐式析构函数即可，Stock 类第一版正是这样做的。然而，了解如何声明和定义析构函数是绝对必要的，下面为 Stock 类提供一个析构函数。

和构造函数一样，析构函数的名称也很特殊：在类名前加上~。因此，Stock 类的析构函数为~Stock( )。另外，和构造函数一样，析构函数也可以没有返回值和声明类型。与构造函数不同的是，析构函数没有参数，因此 Stock 析构函数的原型必须是这样的：

```
~Stock();
```

由于 Stock 的析构函数不承担任何重要的工作，因此可以将它编写为不执行任何操作的函数：

```
Stock::~Stock()
{
}
```

然而，为让您能看出析构函数何时被调用，这样编写其代码：

```
Stock::~Stock()     // class destructor
{
    cout << "Bye, " << company << "!\n";
}
```

什么时候应调用析构函数呢？这由编译器决定，通常不应在代码中显式地调用析构函数（有关例外情形，请参阅第 12 章的“再谈定位 new 运算符”）。如果创建的是静态存储类对象，则其析构函数将在程序结束时自动被调用。如果创建的是自动存储类对象（就像前面的示例中那样），则其析构函数将在程序执行完代码块时（该对象是在其中定义的）自动被调用。如果对象是通过 new 创建的，则它将驻留在栈内存或自由存储区中，当使用 delete 来释放内存时，其析构函数将自动被调用。最后，程序可以创建临时对象来完成特定的操作，在这种情况下，程序将在结束对该对象的使用时自动调用其析构函数。

由于在类对象过期时析构函数将自动被调用，因此必须有一个析构函数。如果程序员没有提供析构函数，编译器将隐式地声明一个默认析构函数，并在发现导致对象被删除的代码后，提供默认析构函数的定义。

10.3.5 改进 Stock 类

下面将构造函数和析构函数加入到类和方法的定义中。鉴于添加构造函数的重大意义，这里将名称从 stock00.h 改为 stock10.h。类方法放在文件 stock10.cpp 中。最后，将使用这些资源的程序放在第三个文件中，这个文件名为 usestock2.cpp。

1. 头文件

程序清单 10.4 列出了头文件。它将构造函数和析构函数的原型加入到原来的类声明中。另外，它还删除了 acquire( )函数——现在已经不再需要它了，因为有构造函数。该文件还使用第 9 章介绍的#ifndef 技术来防止多重包含。

程序清单 10.4　stock10.h

```
// stock10.h -- Stock class declaration with constructors, destructor added
#ifndef STOCK10_H_
#define STOCK01_H_
#include <string>

class Stock
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
// two constructors
    Stock();        // default constructor
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock();       // noisy destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};

#endif
```

2. 实现文件

程序清单 10.5 提供了方法的定义。它包含了文件 stock10.h，以提供类声明（将文件名放在双引号而不是方括号中意味着编译器将源文件所在的目录中搜索它）。另外，程序清单 10.5 还包含了头文件 iostream，

以提供 I/O 支持。该程序清单还使用 using 声明和限定名称（如 std::string）来访问头文件中的各种声明。该文件将构造函数和析构函数的方法定义添加到以前的方法定义中。为让您知道这些方法何时被调用，它们都显示一条消息。这并不是构造函数和析构函数的常规功能，但有助于您更好地了解类是如何使用它们的。

程序清单 10.5　stock10.cpp

```
// stock10.cpp -- Stock class with constructors, destructor added
#include <iostream>
#include "stock10.h"

// constructors (verbose versions)
Stock::Stock()        // default constructor
{
    std::cout << "Default constructor called\n";
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const std::string & co, long n, double pr)
{
    std::cout << "Constructor using " << co << " called\n";
    company = co;

    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
// class destructor
Stock::~Stock()        // verbose class destructor
{
    std::cout << "Bye, " << company << "!\n";
}

// other methods
void Stock::buy(long num, double price)
{
     if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
```

```
        set_tot();
    }
}

void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // set format to #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);

    cout << "Company: " << company
        << "  Shares: " << shares << '\n';
    cout << "  Share Price: $" << share_val;
    // set format to #.##
    cout.precision(2);
    cout << "  Total Worth: $" << total_val << '\n';

    // restore original format
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}
```

3. 客户文件

程序清单 10.6 提供了一个测试这些新方法的小程序；由于它只是使用 Stock 类，因此是 Stock 类的客

户。和 stock10.cpp 一样，它也包含了文件 stock10.h 以提供类声明。该程序显示了构造函数和析构函数，它还使用了程序清单 10.3 调用的格式化命令。要编译整个程序，必须使用第 1 章和第 9 章介绍的多文件程序技术。

程序清单 10.6　usestock2.cpp

```
// usestok1.cpp -- using the Stock class
// compile with stock10.cpp
#include <iostream>
#include "stock10.h"

int main()
{
  {
    using std::cout;
    cout << "Using constructors to create new objects\n";
    Stock stock1("NanoSmart", 12, 20.0);            // syntax 1
    stock1.show();
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // syntax 2
    stock2.show();

    cout << "Assigning stock1 to stock2:\n";
    stock2 = stock1;
    cout << "Listing stock1 and stock2:\n";
    stock1.show();
    stock2.show();

    cout << "Using a constructor to reset an object\n";
    stock1 = Stock("Nifty Foods", 10, 50.0);    // temp object
    cout << "Revised stock1:\n";
    stock1.show();
    cout << "Done\n";
  }
    return 0;
}
```

编译程序清单 10.4、程序清单 10.5 和程序清单 10.6 所示的程序，得到一个可执行程序。下面是使用某个编译器得到的可执行程序的输出：

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
  Share Price: $2.00  Total Worth: $4.00
Assigning stock1 to stock2:
Listing stock1 and stock2:
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Using a constructor to reset an object
Constructor using Nifty Foods called
Bye, Nifty Foods!
Revised stock1:
```

```
Company: Nifty Foods  Shares: 10
  Share Price: $50.00  Total Worth: $500.00
Done
Bye, NanoSmart!
Bye, Nifty Foods!
```

使用某些编译器编译该程序时，该程序输出的前半部分可能如下（比前面多了一行）：

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Constructor using Boffo Objects called
Bye, Boffo Objects!                          << additional line
Company: Boffo Objects  Shares: 2
  Share Price: $2.00  Total Worth: $4.00
...
```

下一小节将解释输出行“Bye, Boffo Objects!”。

提示：您可能注意到了，在程序清单 10.6 中，main()的开头和末尾多了一个大括号。诸如 stock1 和 stock2 等自动变量将在程序退出其定义所属代码块时消失。如果没有这些大括号，代码块将为整个 main()，因此仅当 main()执行完毕后，才会调用析构函数。在窗口环境中，这意味着将在两个析构函数调用前关闭，导致您无法看到最后两条消息。但添加这些大括号后，最后两个析构函数调用将在到达返回语句前执行，从而显示相应的消息。

4. 程序说明

程序清单 10.6 中的下述语句：

```
Stock stock1("NanoSmart", 12, 20.0);
```

创建一个名为 stock1 的 Stock 对象，并将其数据成员初始化为指定的值：

```
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
```

下面的语句使用另一种语法创建并初始化一个名为 stock2 的对象：

```
stock2:
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
```

C++标准允许编译器使用两种方式来执行第二种语法。一种是使其行为和第一种语法完全相同：

```
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
```

另一种方式是允许调用构造函数来创建一个临时对象，然后将该临时对象复制到 stock2 中，并丢弃它。如果编译器使用的是这种方式，则将为临时对象调用析构函数，因此生成下面的输出：

```
Constructor using Boffo Objects called
Bye, Boffo Objects!
Company: Boffo Objects  Shares: 2
```

生成上述输出的编译器可能立刻删除临时对象，但也可能会等一段时间，在这种情况下，析构函数的消息将会过一段时间才显示。

下面的语句表明可以将一个对象赋给同类型的另一个对象：

```
stock2 = stock1;     // object assignment
```

与给结构赋值一样，在默认情况下，给类对象赋值时，将把一个对象的成员复制给另一个。在这个例子中，stock2 原来的内容将被覆盖。

注意：在默认情况下，将一个对象赋给同类型的另一个对象时，C++将源对象的每个数据成员的内容

复制到目标对象中相应的数据成员中。

构造函数不仅仅可用于初始化新对象。例如，该程序的 main( )中包含下面的语句：

```
stock1 = Stock("Nifty Foods", 10, 50.0);
```

stock1 对象已经存在，因此这条语句不是对 stock1 进行初始化，而是将新值赋给它。这是通过让构造程序创建一个新的、临时的对象，然后将其内容复制给 stock1 来实现的。随后程序调用析构函数，以删除该临时对象，如下面经过注释后的输出所示：

```
Using a constructor to reset an object
Constructor using Nifty Foods called >> temporary object created
Bye, Nifty Foods!                    >> temporary object destroyed
Revised stock1:
Company: Nifty Foods  Shares: 10     >> data now copied to stock1
  Share Price: $50.00  Total Worth: $500.00
```

有些编译器可能要过一段时间才删除临时对象，因此析构函数的调用将延迟。

最后，程序显示了下面的内容：

```
Done
Bye, NanoSmart!
Bye, Nifty Foods!
```

函数 main( )结束时，其局部变量（stock1 和 stock2）将消失。由于这种自动变量被放在栈中，因此最后创建的对象将最先被删除，最先创建的对象将最后被删除（“NanoSmart”最初位于 stock1 中，但随后被传输到 stock2 中，然后 stock1 被重置为“Nifty Food”）。

输出表明，下面两条语句有根本性的差别：

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
stock1 = Stock("Nifty Foods", 10, 50.0); // temporary object
```

第一条语句是初始化，它创建有指定值的对象，可能会创建临时对象（也可能不会）；第二条语句是赋值。像这样在赋值语句中使用构造函数总会导致在赋值前创建一个临时对象。

提示：如果既可以通过初始化，也可以通过赋值来设置对象的值，则应采用初始化方式。通常这种方式的效率更高。

5. C++11 列表初始化

在 C++11 中，可将列表初始化语法用于类吗？可以，只要提供与某个构造函数的参数列表匹配的内容，并用大括号将它们括起：

```
Stock hot_tip = {"Derivatives Plus Plus", 100, 45.0};
Stock jock {"Sport Age Storage, Inc"};
Stock temp {};
```

在前两个声明中，用大括号括起的列表与下面的构造函数匹配：

```
Stock::Stock(const std::string & co, long n = 0, double pr = 0.0);
```

因此，将使用该构造函数来创建这两个对象。创建对象 jock 时，第二和第三个参数将为默认值 0 和 0.0。第三个声明与默认构造函数匹配，因此将使用该构造函数创建对象 temp。

另外，C++11 还提供了名为 std::initialize_list 的类，可将其用作函数参数或方法参数的类型。这个类可表示任意长度的列表，只要所有列表项的类型都相同或可转换为相同的类型，这将在第 16 章介绍。

6. const 成员函数

请看下面的代码片段：

```
const Stock land = Stock("Kludgehorn Properties");
land.show();
```

对于当前的 C++来说，编译器将拒绝第二行。这是什么原因呢？因为 show( )的代码无法确保调用对象不被修改——调用对象和 const 一样，不应被修改。我们以前通过将函数参数声明为 const 引用或指向 const 的指针来解决这种问题。但这里存在语法问题：show( )方法没有任何参数。相反，它所使用的对象是由方法调用隐式地提供的。需要一种新的语法——保证函数不会修改调用对象。C++的解决方法是将 const 关键字放在函数的括号后面。也就是说，show( )声明应像这样：

```
void show() const;          // promises not to change invoking object
```

同样，函数定义的开头应像这样：

```
void stock::show() const    // promises not to change invoking object
```

以这种方式声明和定义的类函数被称为 const 成员函数。就像应尽可能将 const 引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为 const。从现在开始，我们将遵守这一规则。

10.3.6 构造函数和析构函数小结

介绍一些构造函数和析构函数的例子后，您可能想停下来，整理一下学到的知识。为此，下面对这些方法进行总结。

构造函数是一种特殊的类成员函数，在创建类对象时被调用。构造函数的名称和类名相同，但通过函数重载，可以创建多个同名的构造函数，条件是每个函数的特征标（参数列表）都不同。另外，构造函数没有声明类型。通常，构造函数用于初始化类对象的成员，初始化应与构造函数的参数列表匹配。例如，假设 Bozo 类的构造函数的原型如下：

```
Bozo(const char * fname, const char * lname);   // constructor prototype
```

则可以使用它来初始化新对象：

```
Bozo bozetta = bozo("Bozetta", "Biggens");   // primary form
Bozo fufu("Fufu", "O'Dweeb");                // short form
Bozo *pc = new Bozo("Popo", "Le Peu");       // dynamic object
```

如果编译器支持 C++11，则可使用列表初始化：

```
Bozo bozetta = {"Bozetta", "Biggens"};       // C++11
Bozo fufu{"Fufu", "O'Dweeb"}                 // C++11;
Bozo *pc = new Bozo{"Popo", "Le Peu"};       // C++11
```

如果构造函数只有一个参数，则将对象初始化为一个与参数的类型相同的值时，该构造函数将被调用。例如，假设有这样一个构造函数原型：

```
cout << trip;
```

则可以使用下面的任何一种形式来初始化对象：

```
Bozo dribble = bozo(44);   // primary form
Bozo roon(66);             // secondary form
Bozo tubby = 32;           // special form for one-argument constructors
```

实际上，第三个示例是新内容，不属于复习内容，但现在正是介绍它的好时机。第 11 章将介绍一种关闭这项特性的方式，因为它可能带来令人不愉快的意外。

警告：接受一个参数的构造函数允许使用赋值语法将对象初始化为一个值：

```
Classname object = value;
```

这种特性可能导致问题，但正如第 11 章将介绍的，可关闭这项特性。

默认构造函数没有参数，因此如果创建对象时没有进行显式地初始化，则将调用默认构造函数。如果程序中没有提供任何构造函数，则编译器会为程序定义一个默认构造函数；否则，必须自己提供默认构造函数。默认构造函数可以没有任何参数；如果有，则必须给所有参数都提供默认值：

```
Bozo();                                  // default constructor prototype
Bistro(const char * s = "Chez Zero");    // default for Bistro class
```

对于未被初始化的对象，程序将使用默认构造函数来创建：

```
Bozo bubi;              // use default
Bozo *pb = new Bozo;    // use default
```

就像对象被创建时程序将调用构造函数一样，当对象被删除时，程序将调用析构函数。每个类都只能有一个析构函数。析构函数没有返回类型（连 void 都没有），也没有参数，其名称为类名称前加上~。例如，Bozo 类的析构函数的原型如下：

```
~Bozo();     // class destructor
```

如果构造函数使用了 new，则必须提供使用 delete 的析构函数。

10.4　this 指针

对于 Stock 类，还有很多工作要做。到目前为止，每个类成员函数都只涉及一个对象，即调用它的对象。但有时候方法可能涉及到两个对象，在这种情况下需要使用 C++的 this 指针。

虽然 Stock 类声明可以显示数据，但它缺乏分析能力。例如，从 show( )的输出我们可以知道持有的哪一支股票价格最高，但由于程序无法直接访问 total_val，因此无法作出判断。要让程序知道存储的数据，最直接的方式是让方法返回一个值。为此，通常使用内联代码，如下例所示：

```
class Stock
{
private:
    ...
    double total_val;
    ...
public:
    double total() const { return total_val; }
    ...
};
```

就直接程序访问而言，上述定义实际上是使 total_val 为只读的。也就是说，可以使用方法 total_val( ) 来获得 total_val 的值，但这个类没有提供专门用于重新设置 total_val 的值的方法（作为一种副产品，其他方法，如 buy( )、sell( )和 update( )确实在重新设置成员 shares 和 share_val 的值的同时修改了 total_val 的值）。

通过将该函数添加到类声明中，可以让程序查看一系列股票，找到价格最高的那一支。然而，可以采用另一种方法——一种帮助您了解 this 指针的方法。这种方法是，定义一个成员函数，它查看两个 Stock 对象，并返回股价较高的那个对象的引用。实现这种方法时，将出现一些有趣的问题，下面就来讨论这些问题。

首先，如何将两个要比较的对象提供给成员函数呢？例如，假设将该方法命名为 topval( )，则函数调用 stock1.topval( )将访问 stock1 对象的数据，而 stock2.topval( )将访问 stock2 对象的数据。如果希望该方法对两个对象进行比较，则必须将第二个对象作为参数传递给它。出于效率方面的考虑，可以按引用来传递参数，也就是说，topval( )方法使用一个类型为 const Stock &的参数。

其次，如何将方法的答案传回给调用程序呢？最直接的方法是让方法返回一个引用，该引用指向股价总值较高的对象。因此，用于比较的方法的原型如下：

```
const Stock & topval(const Stock & s) const;
```

该函数隐式地访问一个对象，而显式地访问另一个对象，并返回其中一个对象的引用。括号中的 const 表明，该函数不会修改被显式地访问的对象；而括号后的 const 表明，该函数不会修改被隐式地访问的对象。由于该函数返回了两个 const 对象之一的引用，因此返回类型也应为 const 引用。

假设要对 Stock 对象 stock1 和 stock2 进行比较，并将其中股价总值较高的那一个赋给 top 对象，则可以使用下面两条语句之一：

```
top = stock1.topval(stock2);
top = stock2.topval(stock1);
```

第一种格式隐式地访问 stock1，而显式地访问 stock2；第二种格式显式地访问 stock1，而隐式地访问 stock2（参见图 10.3）。无论使用哪一种方式，都将对这两个对象进行比较，并返回股价总值较高的那一个对象。

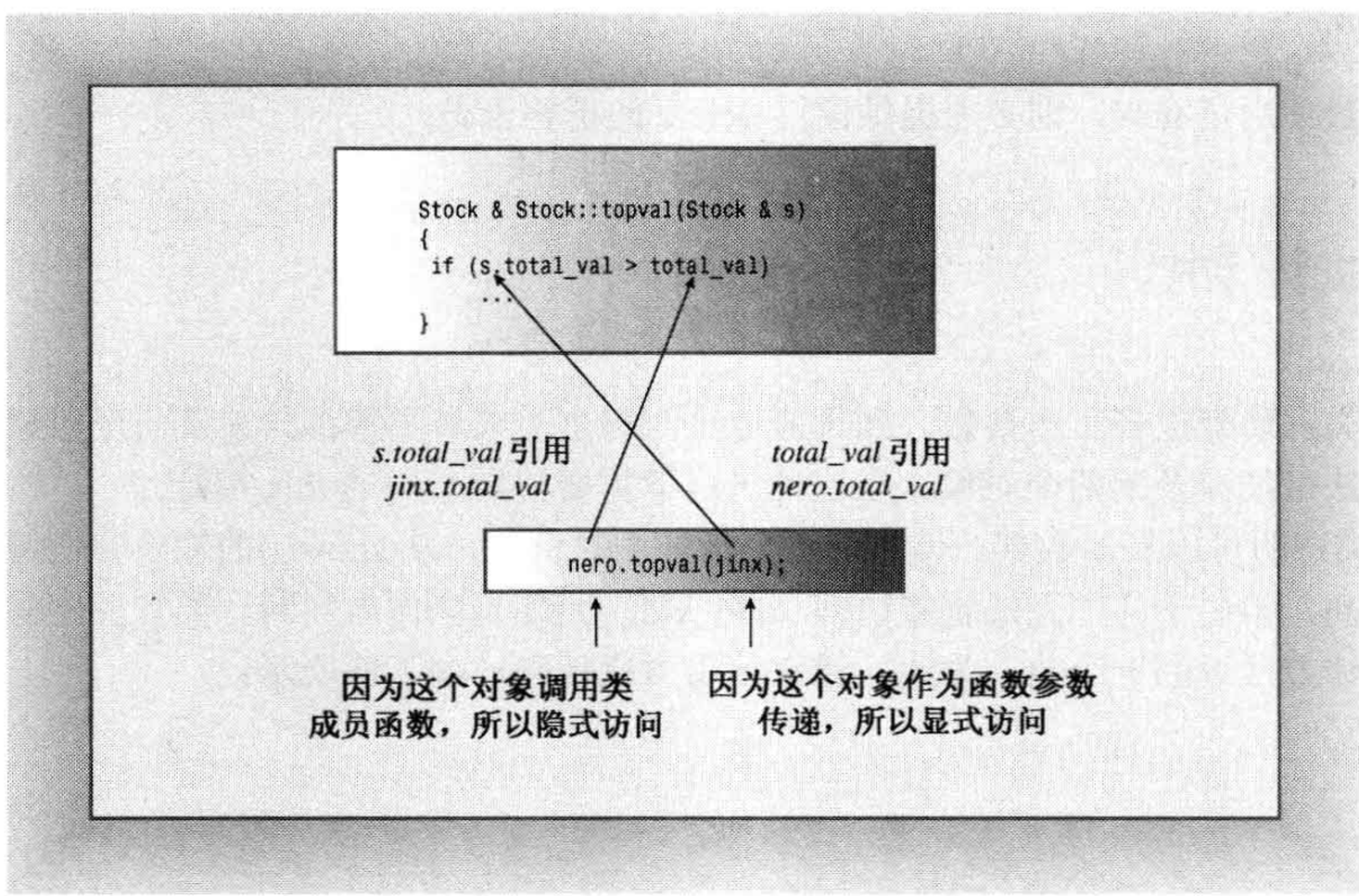


图 10.3　使用成员函数访问两个对象

实际上，这种表示法有些混乱。如果可以使用关系运算符>来比较这两个对象，将更为清晰。可以使用运算符重载（参见第 11 章）完成这项工作。

同时，还要注意的是 topval( )的实现，它将引发一个小问题。下面的部分实现强调了这个问题：

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;                // argument object
    else
        return ?????;            // invoking object
}
```

其中，s.total_val 是作为参数传递的对象的总值，total_val 是用来调用该方法的对象的总值。如果 s.total_val 大于 toatl_val，则函数将返回指向 s 的引用；否则，将返回用来调用该方法的对象（在 OOP 中，是 topval 消息要发送给的对象）。问题在于，如何称呼这个对象？如果调用 stock1.topval(stock2)，则 s 是 stock2 的引用（即 stock2 的别名），但 stock1 没有别名。

C++解决这种问题的方法是：使用被称为 this 的特殊指针。this 指针指向用来调用成员函数的对象（this 被作为隐藏参数传递给方法）。这样，函数调用 stock1.topval（stock2）将 this 设置为 stock1 对象的地址，使得这个指针可用于 topval( )方法。同样，函数调用 stock2.topval（stock1）将 this 设置为 stock2 对象的地址。一般来说，所有的类方法都将 this 指针设置为调用它的对象的地址。确实，topval( )中的 total_val 只不过是 this->total_val 的简写（第 4 章使用->运算符，通过指针来访问结构成员。这也适用于类成员）（参见图 10.4）。

注意：

*每个成员函数（包括构造函数和析构函数）都有一个 this 指针。this 指针指向调用对象。如果方法需要引用整个调用对象，则可以使用表达式*this。在函数的括号后面使用 const 限定符将 this 限定为 const，*

这样将不能使用 this 来修改对象的值。

然而，要返回的并不是 this，因为 this 是对象的地址，而是对象本身，即*this（将解除引用运算符*用于指针，将得到指针指向的值）。现在，可以将*this 作为调用对象的别名来完成前面的方法定义。

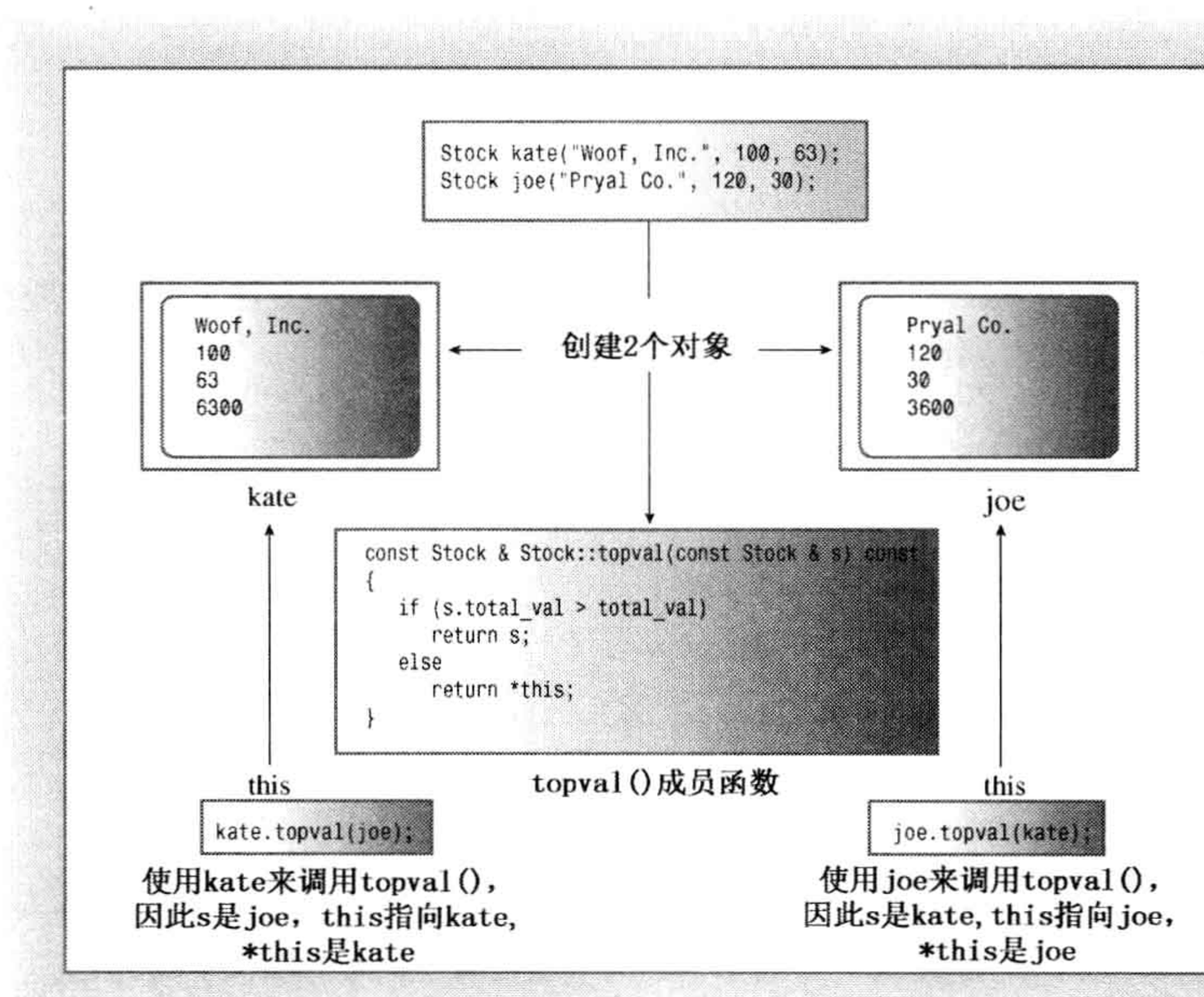


图 10.4　this 指向调用对象

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
       return s;            // argument object
    else
       return *this;        // invoking object
}
```

返回类型为引用意味着返回的是调用对象本身，而不是其副本。程序清单 10.7 列出了新的头文件。

程序清单 10.7　stock20.h

```
// stock20.h -- augmented version
#ifndef STOCK20_H_
#define STOCK20_H_
#include <string>

class Stock
{
private:
    std::string company;
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();        // default constructor
```

```
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock();        // do-nothing destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show()const;
    const Stock & topval(const Stock & s) const;
};

#endif
```

程序清单 10.8 列出了修订后的类方法文件，其中包括新的 topval( )方法。另外，现在您已经了解了构造函数和析构函数的工作原理，因此这里没有显示消息。

程序清单 10.8　stock20.cpp

```
// stock20.cpp -- augmented version
#include <iostream>
#include "stock20.h"

// constructors
Stock::Stock()         // default constructor
{
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const std::string & co, long n, double pr)
{
    company = co;

    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

// class destructor
Stock::~Stock()        // quiet class destructor
{
}

// other methods
void Stock::buy(long num, double price)
{
     if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
```

```
                << "Transaction is aborted.\n";
        }
        else
        {
            shares += num;
            share_val = price;
            set_tot();
        }
    }

    void Stock::sell(long num, double price)
    {
        using std::cout;
        if (num < 0)
        {
            cout << "Number of shares sold can't be negative. "
                 << "Transaction is aborted.\n";
        }
        else if (num > shares)
        {
            cout << "You can't sell more than you have! "
                 << "Transaction is aborted.\n";
        }
        else
        {
            shares -= num;
            share_val = price;
            set_tot();
        }
    }

    void Stock::update(double price)
    {
        share_val = price;
        set_tot();
    }

    void Stock::show() const
    {
        using std::cout;
        using std::ios_base;
        // set format to #.###
        ios_base::fmtflags orig =
            cout.setf(ios_base::fixed, ios_base::floatfield);
        std::streamsize prec = cout.precision(3);

        cout << "Company: " << company
            << "  Shares: " << shares << '\n';
        cout << "  Share Price: $" << share_val;
        // set format to #.##
        cout.precision(2);
        cout << "  Total Worth: $" << total_val << '\n';

        // restore original format
```

```
    // restore original format
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}

const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;
    else
        return *this;
}
```

当然，我们想知道 this 指针是否有用。显然，应在一个包含对象数组的程序中使用这种新方法。因此接下来介绍对象数组这一主题。

10.5 对象数组

和 Stock 示例一样，用户通常要创建同一个类的多个对象。可以创建独立对象变量，就像本章前面的示例所做的，但创建对象数组将更合适。这似乎是在介绍一个未知领域，但实际上，声明对象数组的方法与声明标准类型数组相同：

```
Stock mystuff[4]; // creates an array of 4 Stock objects
```

前面讲过，当程序创建未被显式初始化的类对象时，总是调用默认构造函数。上述声明要求，这个类要么没有显式地定义任何构造函数（在这种情况下，将使用不执行任何操作的隐式默认构造函数），要么定义了一个显式默认构造函数（就像这个例子那样）。每个元素（mystuff[0]、mystuff[1]等）都是 Stock 对象，可以使用 Stock 方法：

```
mystuff[0].update();       // apply update() to 1st element
mystuff[3].show();         // apply show() to 4th element
const Stock * tops = mystuff[2].topval(mystuff[1]);
      // compare 3rd and 2nd elements and set tops
      // to point at the one with a higher total value
```

可以用构造函数来初始化数组元素。在这种情况下，必须为每个元素调用构造函数：

```
const int STKS = 4;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 20),
    Stock("Boffo Objects", 200, 2.0),
    Stock("Monolithic Obelisks", 130, 3.25),
    Stock("Fleep Enterprises", 60, 6.5)
    };
```

这里的代码使用标准格式对数组进行初始化：用括号括起的、以逗号分隔的值列表。其中，每次构造函数调用表示一个值。如果类包含多个构造函数，则可以对不同的元素使用不同的构造函数：

```
const int STKS = 10;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 20),
    Stock(),
    Stock("Monolithic Obelisks", 130, 3.25),
};
```

上述代码使用 Stock(const string & co, long n, double pr)初始化 stock[0]和 stock[2]，使用构造函数 Stock( )

初始化 stock[1]。由于该声明只初始化了数组的部分元素，因此余下的 7 个元素将使用默认构造函数进行初始化。

初始化对象数组的方案是，首先使用默认构造函数创建数组元素，然后花括号中的构造函数将创建临时对象，然后将临时对象的内容复制到相应的元素中。因此，要创建类对象数组，则这个类必须有默认构造函数。

程序清单 10.9 在一个小程序中使用了这些原理，该程序对 4 个数组元素进行初始化，显示它们的内容，并找出这些元素中总值最高的一个。由于 topval( )每次只检查两个对象，因此程序使用 for 循环来检查整个数组。另外，它使用 stock 指针来跟踪值最高的元素。该程序使用程序清单 10.7 中的头文件和程序清单 10.8 中的方法文件。

程序清单 10.9　usestock2.cpp

```
// usestok2.cpp -- using the Stock class
// compile with stock20.cpp
#include <iostream>
#include "stock20.h"

const int STKS = 4;
int main()
{
// create an array of initialized objects
    Stock stocks[STKS] = {
        Stock("NanoSmart", 12, 20.0),
        Stock("Boffo Objects", 200, 2.0),
        Stock("Monolithic Obelisks", 130, 3.25),
        Stock("Fleep Enterprises", 60, 6.5)
        };

    std::cout << "Stock holdings:\n";
    int st;
    for (st = 0; st < STKS; st++)
        stocks[st].show();
// set pointer to first element
    const Stock * top = &stocks[0];
    for (st = 1; st < STKS; st++)
        top = &top->topval(stocks[st]);
// now top points to the most valuable holding
    std::cout << "\nMost valuable holding:\n";
    top->show();
     return 0;
}
```

下面是该程序的输出：

```
Stock holdings:
Company: NanoSmart  Shares: 12
  Share Price: $20.000  Total Worth: $240.00
Company: Boffo Objects  Shares: 200
  Share Price: $2.000  Total Worth: $400.00
Company: Monolithic Obelisks  Shares: 130
  Share Price: $3.250  Total Worth: $422.50
Company: Fleep Enterprises  Shares: 60
  Share Price: $6.500  Total Worth: $390.00

Most valuable holding:
Company: Monolithic Obelisks  Shares: 130
  Share Price: $3.250  Total Worth: $422.50
```

有关程序清单 10.9，需要注意的一点是，大部分工作是在类设计中完成的。完成类设计后，编写程序的工作本身便相当简单。

顺便说一句，知道 this 指针就可以更深入了解 C++的工作方式。例如，最初的 UNIX 实现使用 C++前端 cfront 将 C++程序转换为 C 程序。处理方法的定义时，只需将下面这样的 C++方法定义：

```
void Stock::show() const
{
    cout << "Company: " << company
         << "  Shares: " << shares << '\n'
         << "  Share Price: $" << share_val
         << "  Total Worth: $" << total_val << '\n';
```

转换为下面这样的 C-风格定义：

```
void show(const Stock * this)
{
    cout << "Company: " << this->company
         << "  Shares: " << this->shares << '\n'
         << "  Share Price: $" << this->share_val
         << "  Total Worth: $" << this->total_val << '\n';
}
```

即将 Stock::限定符转换为函数参数（指向 Stock 的指针），然后用这个指针来访问类成员。

同样，该前端将下面的函数调用：

```
top.show();
```

转换为：

```
show(&top);
```

这样，将调用对象的地址赋给了 this 指针（实际情况可能更复杂些）。

10.6 类作用域

第 9 章介绍了全局（文件）作用域和局部（代码块）作用域。可以在全局变量所属文件的任何地方使用它，而局部变量只能在其所属的代码块中使用。函数名称的作用域也可以是全局的，但不能是局部的。C++类引入了一种新的作用域：类作用域。

在类中定义的名称（如类数据成员名和类成员函数名）的作用域都为整个类，作用域为整个类的名称只在该类中是已知的，在类外是不可知的。因此，可以在不同类中使用相同的类成员名而不会引起冲突。例如，Stock 类的 shares 成员不同于 JobRide 类的 shares 成员。另外，类作用域意味着不能从外部直接访问类的成员，公有成员函数也是如此。也就是说，要调用公有成员函数，必须通过对象：

```
Stock sleeper("Exclusive Ore", 100, 0.25);  // create object
sleeper.show();       // use object to invoke a member function
show();               // invalid -- can't call method directly
```

同样，在定义成员函数时，必须使用作用域解析运算符：

```
void Stock::update(double price)
{
    ...
}
```

总之，在类声明或成员函数定义中，可以使用未修饰的成员名称（未限定的名称），就像 sell( )调用 set_tot( )成员函数时那样。构造函数名称在被调用时，才能被识别，因为它的名称与类名相同。在其他情

况下，使用类成员名时，必须根据上下文使用直接成员运算符（.）、间接成员运算符（->）或作用域解析运算符（::）。下面的代码片段演示了如何访问具有类作用域的标识符：

```
class Ik
{
private:
   int fuss;        // fuss has class scope
public:
   Ik(int f  = 9) {fuss = f; }  // fuss is in scope
   void ViewIk() const;         // ViewIk has class scope
};

void Ik::ViewIk() const    //Ik:: places ViewIk into Ik scope
{
   cout << fuss << endl;   // fuss in scope within class methods
}
...
int main()
{
   Ik * pik = new Ik;
   Ik ee = Ik(8); // constructor in scope because has class name
   ee.ViewIk();   // class object brings ViewIk into scope
   pik->ViewIk(); // pointer-to-Ik brings ViewIk into scope
...
```

10.6.1　作用域为类的常量

有时候，使符号常量的作用域为类很有用。例如，类声明可能使用字面值 30 来指定数组的长度，由于该常量对于所有对象来说都是相同的，因此创建一个由所有对象共享的常量是个不错的主意。您可能以为这样做可行：

```
class Bakery
{
private:
    const int Months = 12;    // declare a constant? FAILS
    double costs[Months];
    ...
```

但这是行不通的，因为声明类只是描述了对象的形式，并没有创建对象。因此，在创建对象前，将没有用于存储值的空间（实际上，C++11 提供了成员初始化，但不适用于前述数组声明，第 12 章将介绍该主题）。然而，有两种方式可以实现这个目标，并且效果相同。

第一种方式是在类中声明一个枚举。在类声明中声明的枚举的作用域为整个类，因此可以用枚举为整型常量提供作用域为整个类的符号名称。也就是说，可以这样开始 Bakery 声明：

```
class Bakery
{
private:
    enum {Months = 12};
    double costs[Months];
    ...
```

注意，用这种方式声明枚举并不会创建类数据成员。也就是说，所有对象中都不包含枚举。另外，Months 只是一个符号名称，在作用域为整个类的代码中遇到它时，编译器将用 30 来替换它。

由于这里使用枚举只是为了创建符号常量，并不打算创建枚举类型的变量，因此不需要提供枚举名。顺便说一句，在很多实现中，ios_base 类在其公有部分中完成了类似的工作，诸如 ios_base::fixed 等标识符就来自这里。其中，fixed 是 ios_base 类中定义的典型的枚举量。

C++提供了另一种在类中定义常量的方式——使用关键字 static：

```
class Bakery
{
private:
    static const int Months = 12;
    double costs[Months];
    ...
```

这将创建一个名为 Months 的常量，该常量将与其他静态变量存储在一起，而不是存储在对象中。因此，只有一个 Months 常量，被所有 Bakery 对象共享。第 12 章将深入介绍静态类成员。在 C++98 中，只能使用这种技术声明值为整数或枚举的静态常量，而不能存储 double 常量。C++11 消除了这种限制。

10.6.2 作用域内枚举（C++11）

传统的枚举存在一些问题，其中之一是两个枚举定义中的枚举量可能发生冲突。假设有一个处理鸡蛋和 T 恤的项目，其中可能包含类似下面这样的代码：

```
enum egg {Small, Medium, Large, Jumbo};
enum t_shirt {Small, Medium, Large, Xlarge};
```

这将无法通过编译，因为 egg Small 和 t_shirt Small 位于相同的作用域内，它们将发生冲突。为避免这种问题，C++11 提供了一种新枚举，其枚举量的作用域为类。这种枚举的声明类似于下面这样：

```
enum class egg {Small, Medium, Large, Jumbo};
enum class t_shirt {Small, Medium, Large, Xlarge};
```

也可使用关键字 struct 代替 class。无论使用哪种方式，都需要使用枚举名来限定枚举量：

```
egg choice = egg::Large;          // the Large enumerator of the egg enum
t_shirt Floyd = t_shirt::Large; // the Large enumerator of the t_shirt enum
```

枚举量的作用域为类后，不同枚举定义中的枚举量就不会发生名称冲突了，而您可继续编写处理鸡蛋和 T 恤的项目。

C++11 还提高了作用域内枚举的类型安全。在有些情况下，常规枚举将自动转换为整型，如将其赋给 int 变量或用于比较表达式时，但作用域内枚举不能隐式地转换为整型：

```
enum egg_old {Small, Medium, Large, Jumbo};          // unscoped
enum class t_shirt {Small, Medium, Large, Xlarge}; // scoped
egg_old one = Medium;                                // unscoped
t_shirt rolf = t_shirt::Large;                       // scoped
int king = one;       // implicit type conversion for unscoped
int ring = rolf;      // not allowed, no implicit type conversion
if (king < Jumbo)     // allowed
    std::cout << "Jumbo converted to int before comparison.\n";
if (king < t_shirt::Medium)   // not allowed
    std::cout << "Not allowed: < not defined for scoped enum.\n";
```

但在必要时，可进行显式类型转换：

```
int Frodo = int(t_shirt::Small); // Frodo set to 0
```

枚举用某种底层整型类型表示，在 C++98 中，如何选择取决于实现，因此包含枚举的结构的长度可能随系统而异。对于作用域内枚举，C++11 消除了这种依赖性。默认情况下，C++11 作用域内枚举的底层类型为 int。另外，还提供了一种语法，可用于做出不同的选择：

```
// underlying type for pizza is short
enum class : short pizza {Small, Medium, Large, XLarge};
```

:short 将底层类型指定为 short。底层类型必须为整型。在 C++11 中，也可使用这种语法来指定常规枚举的底层类型，但如果没有指定，编译器选择的底层类型将随实现而异。

10.7　抽象数据类型

Stock 类非常具体。然而，程序员常常通过定义类来表示更通用的概念。例如，就实现计算机专家们所说的抽象数据类型（abstract data type，ADT）而言，使用类是一种非常好的方式。顾名思义，ADT 以通用的方式描述数据类型，而没有引入语言或实现细节。例如，通过使用栈，可以以这样的方式存储数据，即总是从堆顶添加或删除数据。例如，C++程序使用栈来管理自动变量。当新的自动变量被生成后，它们被添加到堆顶；消亡时，从栈中删除它们。

下面简要地介绍一下栈的特征。首先，栈存储了多个数据项（该特征使得栈成为一个容器——一种更为通用的抽象）；其次，栈由可对它执行的操作来描述。

- 可创建空栈。
- 可将数据项添加到堆顶（压入）。
- 可从栈顶删除数据项（弹出）。
- 可查看栈否填满。
- 可查看栈是否为空。

可以将上述描述转换为一个类声明，其中公有成员函数提供了表示栈操作的接口，而私有数据成员负责存储栈数据。类概念非常适合于 ADT 方法。

私有部分必须表明数据存储的方式。例如，可以使用常规数组、动态分配数组或更高级的数据结构（如链表）。然而，公有接口应隐藏数据表示，而以通用的术语来表达，如创建栈、压入等。程序清单 10.10 演示了一种方法，它假设系统实现了 bool 类型。如果您使用的系统没有实现，可以使用 int、0 和 1 代替 bool、false 和 true。

程序清单 10.10　stack.h

```
// stack.h -- class definition for the stack ADT
#ifndef STACK_H_
#define STACK_H_

typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};     // constant specific to class
    Item items[MAX];     // holds stack items
    int top;             // index for top stack item
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);   // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);          // pop top into item
};
#endif
```

在程序清单 10.10 所示的示例中，私有部分表明，栈是使用数组实现的；而公有部分隐藏了这一点。因此，可以使用动态数组来代替数组，而不会改变类的接口。这意味着修改栈的实现后，不需要重新编写使用栈的程序，而只需重新编译栈代码，并将其与已有的程序代码链接起来即可。

接口是冗余的，因为 pop( )和 push( )返回有关栈状态的信息（满或空），而不是 void 类型。在如何处理超出栈限制或者清空栈方面，这为程序员提供了两种选择。他可以在修改栈前使用 isempty( )和 isfull( )来查看，也可以使用 push( )和 pop( )的返回值来确定操作是否成功。

这个类不是根据特定的类型来定义栈，而是根据通用的 Item 类型来描述。在这个例子中，头文件使用 typedef 用 Item 代替 unsigned long。如果需要 double 栈或结构类型的栈，则只需修改 typedef 语句，而类声明和方法定义保持不变。类模板（参见第 14 章）提供了功能更强大的方法，来将存储的数据类型与类设计隔离开来。

接下来需要实现类方法，程序清单 10.11 提供了一种可行的实现。

程序清单 10.11　stack.cpp

```
// stack.cpp -- Stack member functions
#include "stack.h"
Stack::Stack()    // create an empty stack
{
    top = 0;
}

bool Stack::isempty() const
{
    return top == 0;
}

bool Stack::isfull() const
{
    return top == MAX;
}

bool Stack::push(const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

bool Stack::pop(Item & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

默认构造函数确保所有栈被创建时都为空。pop( )和 push( )的代码确保栈顶被正确地处理。这种保证措施是 OOP 更可靠的原因之一。假设要创建一个独立数组来表示栈，创建一个独立变量来表示栈顶索引。则每次创建新栈时，都必须确保代码是正确的。没有私有数据提供的保护，则很可能由于无意修改了数据而导致程序出现非常严重的故障。

下面来测试该栈。程序清单 10.12 模拟了售货员的行为——使用栈的后进先出方式，从购物筐的最上面开始处理购物订单。

程序清单 10.12　stacker.cpp

```
// stacker.cpp -- testing the Stack class
#include <iostream>
#include <cctype>  // or ctype.h
#include "stack.h"
int main()
{
    using namespace std;
    Stack st; // create an empty stack
    char ch;
    unsigned long po;
    cout << "Please enter A to add a purchase order,\n"
        << "P to process a PO, or Q to quit.\n";
    while (cin >> ch && toupper(ch) != 'Q')
    {
        while (cin.get() != '\n')
            continue;
        if (!isalpha(ch))
        {
            cout << '\a';
            continue;
        }
        switch(ch)
        {
            case 'A':
            case 'a': cout << "Enter a PO number to add: ";
                      cin >> po;
                      if (st.isfull())
                          cout << "stack already full\n";
                      else
                          st.push(po);
                      break;
            case 'P':
            case 'p': if (st.isempty())
                          cout << "stack already empty\n";
                      else {
                          st.pop(po);
                          cout << "PO #" << po << " popped\n";
                      }
                      break;
        }
        cout << "Please enter A to add a purchase order,\n"
             << "P to process a PO, or Q to quit.\n";
    }
    cout << "Bye\n";
    return 0;
}
```

程序清单 10.12 中的 while 循环删除输入行中剩余部分，就现在而言这并非是必不可少的，但它使程序的修改更方便（第 14 章将对这个程序进行修改）。下面是该程序的运行情况：

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17885
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17885 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17965
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 18002
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #18002 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17965 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

10.8 总结

面向对象编程强调的是程序如何表示数据。使用 OOP 方法解决编程问题的第一步是根据它与程序之间的接口来描述数据，从而指定如何使用数据。然后，设计一个类来实现该接口。一般来说，私有数据成员存储信息，公有成员函数（又称为方法）提供访问数据的唯一途径。类将数据和方法组合成一个单元，其私有性实现数据隐藏。

通常，将类声明分成两部分组成，这两部分通常保存在不同的文件中。类声明（包括由函数原型表示的方法）应放到头文件中。定义成员函数的源代码放在方法文件中。这样便将接口描述与实现细节分开了。从理论上说，只需知道公有接口就可以使用类。当然，可以查看实现方法（除非只提供了编译形式），但程序不应依赖于其实现细节，如知道某个值被存储为 int。只要程序和类只通过定义接口的方法进行通信，程序员就可以随意地对任何部分做独立的改进，而不必担心这样做会导致意外的不良影响。

类是用户定义的类型，对象是类的实例。这意味着对象是这种类型的变量，例如由 new 按类描述分配的内存。C++试图让用户定义的类型尽可能与标准类型类似，因此可以声明对象、指向对象的指针和对象数组。可以按值传递对象、将对象作为函数返回值、将一个对象赋给同类型的另一个对象。如果提供了构造函数，则在创建对象时，可以初始化对象。如果提供了析构函数方法，则在对象消亡后，程序将执行该函数。

每个对象都存储自己的数据，而共享类方法。如果 mr_object 是对象名，try_me( )是成员函数，则可以

使用成员运算符句点调用成员函数：mr_object.try_me( )。在 OOP 中，这种函数调用被称为将 try_me 消息发送给 mr_object 对象。在 try_me( )方法中引用类数据成员时，将使用 mr_object 对象相应的数据成员。同样，函数调用 i_object.try_me( )将访问 i_object 对象的数据成员。

如果希望成员函数对多个对象进行操作，可以将额外的对象作为参数传递给它。如果方法需要显式地引用调用它的对象，则可以使用 this 指针。由于 this 指针被设置为调用对象的地址，因此*this 是该对象的别名。

类很适合用于描述 ADT。公有成员函数接口提供了 ADT 描述的服务，类的私有部分和类方法的代码提供了实现，这些实现对类的客户隐藏。

10.9　复习题

1．什么是类？

2．类如何实现抽象、封装和数据隐藏？

3．对象和类之间的关系是什么？

4．除了是函数之外，类函数成员与类数据成员之间的区别是什么？

5．定义一个类来表示银行帐户。数据成员包括储户姓名、账号（使用字符串）和存款。成员函数执行如下操作：

- 创建一个对象并将其初始化；
- 显示储户姓名、账号和存款；
- 存入参数指定的存款；
- 取出参数指定的款项。

请提供类声明，而不用给出方法实现。（编程练习 1 将要求编写实现）

6．类构造函数在何时被调用？类析构函数呢？

7．给出复习题 5 中的银行账户类的构造函数的代码。

8．什么是默认构造函数，拥有默认构造函数有何好处？

9．修改 Stock 类的定义（stock20.h 中的版本），使之包含返回各个数据成员值的成员函数。注意：返回公司名的成员函数不应为修改数组提供便利，也就是说，不能简单地返回 string 引用。

10．this 和*this 是什么？

10.10　编程练习

1．为复习题 5 描述的类提供方法定义，并编写一个小程序来演示所有的特性。

2．下面是一个非常简单的类定义：

```
class Person {
private:
    static const LIMIT = 25;
    string lname;       // Person's last name
    char fname[LIMIT];  // Person's first name
public:
    Person() {lname = ""; fname[0] = '\0';  } // #1
    Person(const string & ln, const char * fn = "Heyyou");   // #2
// the following methods display lname and fname
    void Show() const;        // firstname lastname format
    void FormalShow() const;  // lastname, firstname format
};
```

它使用了一个 string 对象和一个字符数组，让您能够比较它们的用法。请提供未定义的方法的代码，以完成这个类的实现。再编写一个使用这个类的程序，它使用了三种可能的构造函数调用（没有参数、一个参数和两个参数）以及两种显示方法。下面是一个使用这些构造函数和方法的例子：

```
Person one;                        // use default constructor
Person two("Smythecraft");         // use #2 with one default argument
Person three("Dimwiddy", "Sam");   // use #2, no defaults
one.Show();
cout << endl;
one.FormalShow();
// etc. for two and three
```

3．完成第 9 章的编程练习 1，但要用正确的 golf 类声明替换那里的代码。用带合适参数的构造函数替换 setgolf（golf &, const char *, int），以提供初始值。保留 setgolf( )的交互版本，但要用构造函数来实现它（例如，setgolf( )的代码应该获得数据，将数据传递给构造函数来创建一个临时对象，并将其赋给调用对象，即*this）。

4．完成第 9 章的编程练习 4，但将 Sales 结构及相关的函数转换为一个类及其方法。用构造函数替换 setSales（sales &，double [ ]，int）函数。用构造函数实现 setSales（Sales &）方法的交互版本。将类保留在名称空间 SALES 中。

5．考虑下面的结构声明：

```
struct customer {
    char fullname[35];
    double payment;
};
```

编写一个程序，它从栈中添加和删除 customer 结构（栈用 Stack 类声明表示）。每次 customer 结构被删除时，其 payment 的值都被加入到总数中，并报告总数。注意：应该可以直接使用 Stack 类而不作修改；只需修改 typedef 声明，使 Item 的类型为 customer，而不是 unsigned long 即可。

6．下面是一个类声明：

```
class Move
{
private:
    double x;
    double y;
public:
    Move(double a = 0, double b = 0);   // sets x, y to a, b
    showmove() const;                   // shows current x, y values
    Move add(const Move & m) const;
// this function adds x of m to x of invoking object to get new x,
// adds y of m to y of invoking object to get new y, creates a new
// move object initialized to new x, y values and returns it
    reset(double a = 0, double b = 0);  // resets x,y to a, b
};
```

请提供成员函数的定义和测试这个类的程序。

7．Betelgeusean plorg 有这些特征。

数据：

- plorg 的名称不超过 19 个字符；
- plorg 有满意指数（CI），这是一个整数。

操作：

- 新的 plorg 将有名称，其 CI 值为 50；
- plorg 的 CI 可以修改；

- plorg 可以报告其名称和 CI；
- plorg 的默认名称为“Plorga”。

请编写一个 Plorg 类声明（包括数据成员和成员函数原型）来表示 plorg，并编写成员函数的函数定义。然后编写一个小程序，以演示 Plorg 类的所有特性。

8．可以将简单列表描述成下面这样：

- 可存储 0 或多个某种类型的列表；
- 可创建空列表；
- 可在列表中添加数据项；
- 可确定列表是否为空；
- 可确定列表是否为满；
- 可访问列表中的每一个数据项，并对它执行某种操作。

可以看到，这个列表确实很简单，例如，它不允许插入或删除数据项。

请设计一个 List 类来表示这种抽象类型。您应提供头文件 list.h 和实现文件 list.cpp，前者包含类定义，后者包含类方法的实现。您还应创建一个简短的程序来使用这个类。

该列表的规范很简单，这主要旨在简化这个编程练习。可以选择使用数组或链表来实现该列表，但公有接口不应依赖于所做的选择。也就是说，公有接口不应有数组索引、节点指针等。应使用通用概念来表达创建列表、在列表中添加数据项等操作。对于访问数据项以及执行操作，通常应使用将函数指针作为参数的函数来处理：

```
void visit(void (*pf)(Item &));
```

其中，pf 指向一个将 Item 引用作为参数的函数（不是成员函数），Item 是列表中数据项的类型。visit( ) 函数将该函数用于列表中的每个数据项。

第 11 章　使用类

本章内容包括：

- 运算符重载。
- 友元函数。
- 重载<<运算符，以便用于输出。
- 状态成员。
- 使用 rand( )生成随机值。
- 类的自动转换和强制类型转换。
- 类转换函数。

C++类特性丰富、复杂、功能强大。在第 9 章，您通过学习定义和使用简单的类，已踏上了面向对象编程之旅。通过定义用于表示对象的数据的类型以及（通过成员函数）定义可对数据执行的操作，您知道了类是如何定义数据类型的。我们还学习了两个特殊的成员函数——构造函数和析构函数，其作用是管理类对象的创建和删除。本章将进一步探讨类的特征，重点是类设计技术，而不是通用原理。您可能发现，本章介绍的一些特性很容易，而另一些很微妙。要更好地理解这些新特性，应使用这些示例进行练习。如果函数使用常规参数而不是引用参数，将发生什么情况呢？如果忽略了析构函数，又将发生什么情况呢？不要害怕犯错误，因为在解决问题的过程中学到的知识，比生搬硬套而不犯错误时要多得多（然而，不要认为所有的错误就都会让人增长见识）。这样，您将更全面地了解 C++是如何工作的以及它可以为我们做哪些工作。

本章首先介绍运算符重载，它允许将标准 C++运算符（如=和+）用于类对象。然后介绍友元，这种 C++机制使得非成员函数可以访问私有数据。最后介绍如何命令 C++对类执行自动类型转换。学习本章和第 12 章后，您将对类构造函数和类析构函数所起的作用有更深入的了解。另外，您还将知道开发和改进类设计时，需要执行的步骤。

学习 C++的难点之一是需要记住大量的东西，但在拥有丰富的实践经验之前，根本不可能全部记住这些东西。从这种意义上说，学习 C++就像学习功能复杂的字处理程序或电子制表程序一样。任何特性都不可怕，但多数人只掌握了那些经常使用的特性，如查找文本或设置为斜体等。您可能在那里曾经学过如何生成替换字符或者创建目录，除非经常使用它们，否则这些技能可能根本与日常工作无关。也许，学习本章知识的最好方法是，在我们自己开发的 C++程序中使用其中的新特性。对这些新特性有了充分的认识后，就可以添加其他 C++特性了。正如 C++创始人 Bjarne Stroustrup 在一次 C++专业程序员大会上所建议的："轻松地使用这种语言。不要觉得必须使用所有的特性，不要在第一次学习时就试图使用所有的特性。"

11.1　运算符重载

下面介绍一种使对象操作更美观的技术。运算符重载是一种形式的 C++多态。第 8 章介绍了 C++是如何使用户能够定义多个名称相同但特征标（参数列表）不同的函数的。这被称为函数重载或函数多态，旨

在让您能够用同名的函数来完成相同的基本操作，即使这种操作被用于不同的数据类型（想象一下，如果必须对不同的物体使用不同的动词，如抬起左脚（lift_lft），拿起汤匙（lift_sp），英语将会多么笨拙）。运算符重载将重载的概念扩展到运算符上，允许赋予 C++运算符多种含义。实际上，很多 C++（也包括 C 语言）运算符已经被重载。例如，将*运算符用于地址，将得到存储在这个地址中的值；但将它用于两个数字时，得到的将是它们的乘积。C++根据操作数的数目和类型来决定采用哪种操作。

C++允许将运算符重载扩展到用户定义的类型，例如，允许使用+将两个对象相加。编译器将根据操作数的数目和类型决定使用哪种加法定义。重载运算符可使代码看起来更自然。例如，将两个数组相加是一种常见的运算。通常，需要使用下面这样的 for 循环来实现：

```
for (int i = 0; i < 20; i++)
      evening[i] = sam[i] + janet[i];  // add element by element
```

但在 C++中，可以定义一个表示数组的类，并重载+运算符。于是便可以有这样的语句：

```
evening = sam + janet;                 // add two array objects
```

这种简单的加法表示法隐藏了内部机理，并强调了实质，这是 OOP 的另一个目标。

要重载运算符，需使用被称为运算符函数的特殊函数形式。运算符函数的格式如下：

```
operatorop(argument-list)
```

例如，operator +( )重载+运算符，operator *( )重载*运算符。op 必须是有效的 C++运算符，不能虚构一个新的符号。例如，不能有 operator@( )这样的函数，因为 C++中没有@运算符。然而，operator [ ]( )函数将重载[ ]运算符，因为[ ]是数组索引运算符。例如，假设有一个 Salesperson 类，并为它定义了一个 operator +( )成员函数，以重载+运算符，以便能够将两个 Saleperson 对象的销售额相加，则如果 district2、sid 和 sara 都是 Salesperson 类对象，便可以编写这样的等式：

```
district2 = sid + sara;
```

编译器发现，操作数是 Salesperson 类对象，因此使用相应的运算符函数替换上述运算符：

```
district2 = sid.operator+(sara);
```

然后该函数将隐式地使用 sid（因为它调用了方法），而显式地使用 sara 对象（因为它被作为参数传递），来计算总和，并返回这个值。当然最重要的是，可以使用简便的+运算符表示法，而不必使用笨拙的函数表示法。

虽然 C++对运算符重载做了一些限制，但了解重载的工作方式后，这些限制就很容易理解了。因此，下面首先通过一些示例对运算符重载进行阐述，然后再讨论这些限制。

11.2　计算时间：一个运算符重载示例

如果今天早上在 Priggs 的账户上花费了 2 小时 35 分钟，下午又花费了 2 小时 40 分钟，则总共花了多少时间呢？这个示例与加法概念很吻合，但要相加的单位（小时与分钟的混合）与内置类型不匹配。第 7 章通过定义一个 travel_time 结构和将这种结构相加的 sum( )函数来处理类似的情况。现在将其推广，采用一个使用方法来处理加法的 Time 类。首先使用一个名为 Sum( )的常规方法，然后介绍如何将其转换为重载运算符。程序清单 11.1 列出了这个类的声明。

程序清单 11.1　mytime0.h

```
// mytime0.h -- Time class before operator overloading
#ifndef MYTIME0_H_
#define MYTIME0_H_

class Time
{
```

```
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time Sum(const Time & t) const;
    void Show() const;
};
#endif
```

Time 类提供了用于调整和重新设置时间、显示时间、将两个时间相加的方法。程序清单 11.2 列出了方法定义。请注意，当总的分钟数超过 59 时，AddMin( )和 Sum( )方法是如何使用整数除法和求模运算符来调整 minutes 和 hours 值的。另外，由于这里只使用了 iostream 的 cout，且只使用了一次，因此使用 std::cout 比导入整个名称空间更经济。

程序清单 11.2　mytime0.cpp

```
// mytime0.cpp  -- implementing Time methods
#include <iostream>
#include "mytime0.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
}

void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}

void Time::AddHr(int h)
{
    hours += h;
}

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::Sum(const Time & t) const
{
```

```
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show() const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

来看一下 Sum( )函数的代码。注意参数是引用，但返回类型却不是引用。将参数声明为引用的目的是为了提高效率。如果按值传递 Time 对象，代码的功能将相同，但传递引用，速度将更快，使用的内存将更少。

然而，返回值不能是引用。因为函数将创建一个新的 Time 对象（sum），来表示另外两个 Time 对象的和。返回对象（如代码所做的那样）将创建对象的副本，而调用函数可以使用它。然而，如果返回类型为 Time &，则引用的将是 sum 对象。但由于 sum 对象是局部变量，在函数结束时将被删除，因此引用将指向一个不存在的对象。使用返回类型 Time 意味着程序将在删除 sum 之前构造它的拷贝，调用函数将得到该拷贝。

警告：不要返回指向局部变量或临时对象的引用。函数执行完毕后，局部变量和临时对象将消失，引用将指向不存在的数据。

最后，程序清单 11.3 对 Time 类中计算时间总和的部分进行了测试。

程序清单 11.3　usetime0.cpp

```
// usetime0.cpp -- using the first draft of the Time class
// compile usetime0.cpp and mytime0.cpp together
#include <iostream>
#include "mytime0.h"

int main()
{
    using std::cout;
    using std::endl;
    Time planning;
    Time coding(2, 40);
    Time fixing(5, 55);
    Time total;

    cout << "planning time = ";
    planning.Show();
    cout << endl;

    cout << "coding time = ";
    coding.Show();
    cout << endl;

    cout << "fixing time = ";
    fixing.Show();
    cout << endl;

    total = coding.Sum(fixing);
```

```
    cout << "coding.Sum(fixing) = ";
    total.Show();
    cout << endl;

    return 0;
}
```

下面是程序清单 11.1、程序清单 11.2 和程序清单 11.3 组成的程序的输出：

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding.Sum(fixing) = 8 hours, 35 minutes
```

11.2.1 添加加法运算符

将 Time 类转换为重载的加法运算符很容易，只要将 Sum( )的名称改为 operator +( )即可。这样做是对的，只要把运算符（这里为+）放到 operator 的后面，并将结果用作方法名即可。在这里，可以在标识符中使用字母、数字或下划线之外的其他字符。程序清单 11.4 和程序清单 11.5 反映了这些细微的修改。

程序清单 11.4 mytime1.h

```
// mytime1.h -- Time class before operator overloading
#ifndef MYTIME1_H_
#define MYTIME1_H_

class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    void Show() const;
};
#endif
```

程序清单 11.5 mytime1.cpp

```
// mytime1.cpp  -- implementing Time methods
#include <iostream>
#include "mytime1.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
```

```
}

void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}

void Time::AddHr(int h)
{
    hours += h;
}

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show() const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

和 Sum( )一样，operator +( )也是由 Time 对象调用的，它将第二个 Time 对象作为参数，并返回一个 Time 对象。因此，可以像调用 Sum( )那样来调用 operator +( )方法：

```
total = coding.operator+(fixing);    // function notation
```

但将该方法命令为 operator +( )后，也可以使用运算符表示法：

```
total = coding + fixing;              // operator notation
```

这两种表示法都将调用 operator +( )方法。注意，在运算符表示法中，运算符左侧的对象（这里为 coding）是调用对象，运算符右边的对象（这里为 fixing）是作为参数被传递的对象。程序清单 11.6 说明了这一点。

程序清单 11.6　usetime1.cpp

```
// usetime1.cpp -- using the second draft of the Time class
// compile usetime1.cpp and mytime1.cpp together
#include <iostream>
#include "mytime1.h"

int main()
{
    using std::cout;
    using std::endl;
    Time planning;
```

```
    Time coding(2, 40);
    Time fixing(5, 55);
    Time total;

    cout << "planning time = ";
    planning.Show();
    cout << endl;

    cout << "coding time = ";
    coding.Show();
    cout << endl;

    cout << "fixing time = ";
    fixing.Show();
    cout << endl;

    total = coding + fixing;
    // operator notation
    cout << "coding + fixing = ";
    total.Show();
    cout << endl;

    Time morefixing(3, 28);
    cout << "more fixing time = ";
    morefixing.Show();
    cout << endl;
    total = morefixing.operator+(total);
    // function notation
    cout << "morefixing.operator+(total) = ";
    total.Show();
    cout << endl;

    return 0;
}
```

下面是程序清单 11.4～程序清单 11.6 组成的程序的输出：

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding + fixing = 8 hours, 35 minutes
more fixing time = 3 hours, 28 minutes
morefixing.operator+(total) = 12 hours, 3 minutes
```

总之，operator +( )函数的名称使得可以使用函数表示法或运算符表示法来调用它。编译器将根据操作数的类型来确定如何做：

```
int a, b, c;
Time A, B, C;
c = a + b;      // use int addition
C = A + B;      // use addition as defined for Time objects
```

可以将两个以上的对象相加吗？例如，如果 t1、t2、t3 和 t4 都是 Time 对象，可以这样做吗：

```
t4 = t1 + t2 + t3;                    // valid?
```

为回答这个问题，来看一些上述语句将被如何转换为函数调用。由于+是从左向右结合的运算符，因此上述语句首先被转换成下面这样：

```
t4 = t1.operator+(t2 + t3);           // valid?
```

然后，函数参数本身被转换成一个函数调用，结果如下：

```
t4 = t1.operator+(t2.operator+(t3));   // valid? YES
```

上述语句合法吗？是的。函数调用 t2.operator+(t3)返回一个 Time 对象，后者是 t2 和 t3 的和。然而，该对象成为函数调用 t1.operator+( )的参数，该调用返回 t1 与表示 t2 和 t3 之和的 Time 对象的和。总之，最后的返回值为 t1、t2 和 t3 之和，这正是我们期望的。

11.2.2　重载限制

多数 C++运算符（参见表 11.1）都可以用这样的方式重载。重载的运算符（有些例外情况）不必是成员函数，但必须至少有一个操作数是用户定义的类型。下面详细介绍 C++对用户定义的运算符重载的限制。

1．重载后的运算符必须至少有一个操作数是用户定义的类型，这将防止用户为标准类型重载运算符。因此，不能将减法运算符（-）重载为计算两个 double 值的和，而不是它们的差。虽然这种限制将对创造性有所影响，但可以确保程序正常运行。

2．使用运算符时不能违反运算符原来的句法规则。例如，不能将求模运算符（%）重载成使用一个操作数：

```
int x;
Time shiva;
% x;        // invalid for modulus operator
% shiva;    // invalid for overloaded operator
```

同样，不能修改运算符的优先级。因此，如果将加号运算符重载成将两个类相加，则新的运算符与原来的加号具有相同的优先级。

3．不能创建新运算符。例如，不能定义 operator **( )函数来表示求幂。

4．不能重载下面的运算符。

- sizeof：sizeof 运算符。
- .：成员运算符。
- .*：成员指针运算符。
- ::：作用域解析运算符。
- ?:：条件运算符。
- typeid：一个 RTTI 运算符。
- const_cast：强制类型转换运算符。
- dynamic_cast：强制类型转换运算符。
- reinterpret_cast：强制类型转换运算符。
- static_cast：强制类型转换运算符。

然而，表 11.1 中所有的运算符都可以被重载。

5．表 11.1 中的大多数运算符都可以通过成员或非成员函数进行重载，但下面的运算符只能通过成员函数进行重载。

- =：赋值运算符。
- ( )：函数调用运算符。
- [ ]：下标运算符。
- ->：通过指针访问类成员的运算符。

注意：本章不介绍这里列出的所有运算符，但附录 E 对本书正文中没有介绍的运算符进行了总结。

表 11.1　可重载的运算符

+	-	*	/	%	^
&	|	~=	!	=	<
>	+=	-=	*=	/=	%=

续表

^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	—	,	->*	->
()	[]	new	delete	new []	delete []

除了这些正式限制之外，还应在重载运算符时遵循一些明智的限制。例如，不要将*运算符重载成交换两个 Time 对象的数据成员。表示法中没有任何内容可以表明运算符完成的工作，因此最好定义一个其名称具有说明性的类方法，如 Swap( )。

11.2.3 其他重载运算符

还有一些其他的操作对 Time 类来说是有意义的。例如，可能要将两个时间相减或将时间乘以一个因子，这需要重载减法和乘法运算符。这和重载加法运算符采用的技术相同，即创建 operator –( )和 operator *( )方法。也就是说，将下面的原型添加到类声明中：

```
Time operator-(const Time & t) const;
Time operator*(double n) const;
```

程序清单 11.7 是新的头文件。

程序清单 11.7 mytime2.h

```
// mytime2.h -- Time class after operator overloading
#ifndef MYTIME2_H_
#define MYTIME2_H_

class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    void Show() const;
};
#endif
```

然后将新增方法的定义添加到实现文件中，如程序清单 11.8 所示。

程序清单 11.8 mytime2.cpp

```
// mytime2.cpp  -- implementing Time methods
#include <iostream>
#include "mytime2.h"

Time::Time()
{
    hours = minutes = 0;
}
```

```
Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
}

void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

Time Time::operator-(const Time & t) const
{
    Time diff;
    int tot1, tot2;
    tot1 = t.minutes + 60 * t.hours;
    tot2 = minutes + 60 * hours;
    diff.minutes = (tot2 - tot1) % 60;
    diff.hours = (tot2 - tot1) / 60;
    return diff;
}

Time Time::operator*(double mult) const
{
    Time result;
    long totalminutes = hours * mult * 60 + minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}

void Time::Show() const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

完成上述修改后，就可以使用程序清单 11.9 中的代码来测试新定义了。

程序清单 11.9　usetime2.cpp

```
// usetime2.cpp -- using the third draft of the Time class
// compile usetime2.cpp and mytime2.cpp together
#include <iostream>
#include "mytime2.h"

int main()
{
    using std::cout;
    using std::endl;
    Time weeding(4, 35);
    Time waxing(2, 47);
    Time total;
    Time diff;
    Time adjusted;

    cout << "weeding time = ";
    weeding.Show();
    cout << endl;

    cout << "waxing time = ";
    waxing.Show();
    cout << endl;

    cout << "total work time = ";
    total = weeding + waxing;   // use operator+()
    total.Show();
    cout << endl;

    diff = weeding - waxing;    // use operator-()
    cout << "weeding time - waxing time = ";
    diff.Show();
    cout << endl;

    adjusted = total * 1.5;      // use operator+()
    cout << "adjusted work time = ";
    adjusted.Show();
    cout << endl;
    return 0;
}
```

下面是程序清单 11.7～程序清单 11.9 组成的程序得到的输出：

```
weeding time = 4 hours, 35 minutes
waxing time = 2 hours, 47 minutes
total work time = 7 hours, 22 minutes
weeding time - waxing time = 1 hours, 48 minutes
adjusted work time = 11 hours, 3 minutes
```

11.3　友元

您知道，C++控制对类对象私有部分的访问。通常，公有类方法提供唯一的访问途径，但是有时候这

种限制太严格，以致于不适合特定的编程问题。在这种情况下，C++提供了另外一种形式的访问权限：友元。友元有 3 种：

- 友元函数；
- 友元类；
- 友元成员函数。

通过让函数成为类的友元，可以赋予该函数与类的成员函数相同的访问权限。下面介绍友元函数，其他两种友元将在第 15 章介绍。

介绍如何成为友元前，先介绍为何需要友元。在为类重载二元运算符时（带两个参数的运算符）常常需要友元。将 Time 对象乘以实数就属于这种情况，下面来看看。

在前面的 Time 类示例中，重载的乘法运算符与其他两种重载运算符的差别在于，它使用了两种不同的类型。也就是说，加法和减法运算符都结合两个 Time 值，而乘法运算符将一个 Time 值与一个 double 值结合在一起。这限制了该运算符的使用方式。记住，左侧的操作数是调用对象。也就是说，下面的语句：

```
A = B * 2.75;
```

将被转换为下面的成员函数调用：

```
A = B.operator*(2.75);
```

但下面的语句又如何呢？

```
A = 2.75 * B;      // cannot correspond to a member function
```

从概念上说，2.75 * B 应与 B *2.75 相同，但第一个表达式不对应于成员函数，因为 2.75 不是 Time 类型的对象。记住，左侧的操作数应是调用对象，但 2.75 不是对象。因此，编译器不能使用成员函数调用来替换该表达式。

解决这个难题的一种方式是，告知每个人（包括程序员自己），只能按 B * 2.75 这种格式编写，不能写成 2.75 * B。这是一种对服务器友好-客户警惕的（server-friendly, client-beware）解决方案，与 OOP 无关。

然而，还有另一种解决方式——非成员函数（记住，大多数运算符都可以通过成员或非成员函数来重载）。非成员函数不是由对象调用的，它使用的所有值（包括对象）都是显式参数。这样，编译器能够将下面的表达式：

```
A = 2.75 * B;      // cannot correspond to a member function
```

与下面的非成员函数调用匹配：

```
A = operator*(2.75, B);
```

该函数的原型如下：

```
Time operator*(double m, const Time & t);
```

对于非成员重载运算符函数来说，运算符表达式左边的操作数对应于运算符函数的第一个参数，运算符表达式右边的操作数对应于运算符函数的第二个参数。而原来的成员函数则按相反的顺序处理操作数，也就是说，double 值乘以 Time 值。

使用非成员函数可以按所需的顺序获得操作数（先是 double，然后是 Time），但引发了一个新问题：非成员函数不能直接访问类的私有数据，至少常规非成员函数不能访问。然而，有一类特殊的非成员函数可以访问类的私有成员，它们被称为友元函数。

11.3.1　创建友元

创建友元函数的第一步是将其原型放在类声明中，并在原型声明前加上关键字 friend：

```
friend Time operator*(double m, const Time & t);  // goes in class declaration
```

该原型意味着下面两点：

- 虽然 operator *( )函数是在类声明中声明的，但它不是成员函数，因此不能使用成员运算符来调用；
- 虽然 operator *( )函数不是成员函数，但它与成员函数的访问权限相同。

第二步是编写函数定义。因为它不是成员函数，所以不要使用 Time::限定符。另外，不要在定义中使

用关键字 friend，定义应该如下：

```
Time operator*(double m, const Time & t)  // friend not used in definition
{
    Time result;
    long totalminutes = t.hours * mult * 60 +t. minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}
```

有了上述声明和定义后，下面的语句：

```
A = 2.75 * B;
```

将转换为如下语句，从而调用刚才定义的非成员友元函数：

```
A = operator*(2.75, B);
```

总之，类的友元函数是非成员函数，其访问权限与成员函数相同。

友元是否有悖于 OOP

乍一看，您可能会认为友元违反了 OOP 数据隐藏的原则，因为友元机制允许非成员函数访问私有数据。然而，这个观点太片面了。相反，应将友元函数看作类的扩展接口的组成部分。例如，从概念上看，double 乘以 Time 和 Time 乘以 double 是完全相同的。也就是说，前一个要求有友元函数，后一个使用成员函数，这是 C++句法的结果，而不是概念上的差别。通过使用友元函数和类方法，可以用同一个用户接口表达这两种操作。另外请记住，只有类声明可以决定哪一个函数是友元，因此类声明仍然控制了哪些函数可以访问私有数据。总之，类方法和友元只是表达类接口的两种不同机制。

实际上，按下面的方式对定义进行修改（交换乘法操作数的顺序），可以将这个友元函数编写为非友元函数：

```
Time operator*(double m, const Time & t)
{
    return t * m;      // use t.operator*(m)
}
```

原来的版本显式地访问 t.minutes 和 t.hours，所以它必须是友元。这个版本将 Time 对象 t 作为一个整体使用，让成员函数来处理私有值，因此不必是友元。然而，将该版本作为友元也是一个好主意。最重要的是，它将该作为正式类接口的组成部分。其次，如果以后发现需要函数直接访问私有数据，则只要修改函数定义即可，而不必修改类原型。

提示：如果要为类重载运算符，并将非类的项作为其第一个操作数，则可以用友元函数来反转操作数的顺序。

11.3.2 常用的友元：重载<<运算符

一个很有用的类特性是，可以对<<运算符进行重载，使之能与 cout 一起来显示对象的内容。与前面介绍的示例相比，这种重载要复杂些，因此我们分两步（而不是一步）来完成。

假设 trip 是一个 Time 对象。为显示 Time 的值，前面使用的是 Show( )。然而，如果可以像下面这样操作将更好：

```
cout << trip;  // make cout recognize Time class?
```

之所以可以这样做，是因为<<是可被重载的 C++运算符之一。实际上，它已经被重载很多次了。最初，<<运算符是 C 和 C++的位运算符，将值中的位左移（参见附录 E）。ostream 类对该运算符进行了重载，将其转换为一个输出工具。前面讲过，cout 是一个 ostream 对象，它是智能的，能够识别所有的 C++基本类型。这是因为对于每种基本类型，ostream 类声明中都包含了相应的重载的 operator<<( )定义。也就是说，

一个定义使用 int 参数，一个定义使用 double 参数，等等。因此，要使 cout 能够识别 Time 对象，一种方法是将一个新的函数运算符定义添加到 ostream 类声明中。但修改 iostream 文件是个危险的主意，这样做会在标准接口上浪费时间。相反，通过 Time 类声明来让 Time 类知道如何使用 cout。

1. <<的第一种重载版本

要使 Time 类知道使用 cout，必须使用友元函数。这是什么原因呢？因为下面这样的语句使用两个对象，其中第一个是 ostream 类对象（cout）：

```
cout << trip;
```

如果使用一个 Time 成员函数来重载<<，Time 对象将是第一个操作数，就像使用成员函数重载*运算符那样。这意味着必须这样使用<<：

```
trip << cout;   // if operator<<() were a Time member function
```

这样会令人迷惑。但通过使用友元函数，可以像下面这样重载运算符：

```
void operator<<(ostream & os, const Time & t)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
}
```

这样可以使用下面的语句：

```
cout << trip;
```

按下面这样的格式打印数据：

```
4 hours, 23 minutes
```

友元还是非友元？

新的 Time 类声明使 operatro<<( )函数成为 Time 类的一个友元函数。但该函数不是 ostream 类的友元(尽管对 ostream 类并无害处)。operator<<( )函数接受一个 ostream 参数和一个 Time 参数，因此表面看来它必须同时是这两个类的友元。然而，看看函数代码就会发现，尽管该函数访问了 Time 对象的各个成员，但从始至终都将 ostream 对象作为一个整体使用。因为 operator<<( )直接访问 Time 对象的私有成员，所以它必须是 Time 类的友元。但由于它并不直接访问 ostream 对象的私有成员，所以并不一定必须是 ostream 类的友元。这很好，因为这就意味着不必修订 ostream 的定义。

注意，新的 operator<<( )定义使用 ostream 引用 os 作为它的第一个参数。通常情况下，os 引用 cout 对象，如表达式 cout << trip 所示。但也可以将这个运算符用于其他 ostream 对象，在这种情况下，os 将引用相应的对象。

不知道其他 ostream 对象？

另一个 ostream 对象是 cerr，它将输出发送到标准输出流——默认为显示器，但在 UNIX、Linux 和 Windows 命令行环境中，可将标准错误流重定向到文件。另外，第 6 章介绍的 ofstream 对象可用于将输出写入到文件中。通过继承(参见第 13 章),ofstream 对象可以使用 ostream 的方法。这样,便可以用 operator<<( )定义来将 Time 的数据写入到文件和屏幕上，为此只需传递一个经过适当初始化的 ofstream 对象（而不是 cout 对象）。

调用 cout << trip 应使用 cout 对象本身，而不是它的拷贝，因此该函数按引用（而不是按值）来传递该对象。这样，表达式 cout << trip 将导致 os 成为 cout 的一个别名；而表达式 cerr << trip 将导致 os 成为 cerr 的一个别名。Time 对象可以按值或按引用来传递，因为这两种形式都使函数能够使用对象的值。按引用传递使用的内存和时间都比按值传递少。

2. <<的第二种重载版本

前面介绍的实现存在一个问题。像下面这样的语句可以正常工作：

```
cout << trip;
```

但这种实现不允许像通常那样将重新定义的<<运算符与 cout 一起使用：

```
cout << "Trip time: " << trip << " (Tuesday)\n"; // can't do
```

要理解这样做不可行的原因以及必须如何做才能使其可行，首先需要了解关于 cout 操作的一点知识。请看下面的语句：

```
int x = 5;
int y = 8;
cout << x << y;
```

C++从左至右读取输出语句，意味着它等同于：

```
(cout << x) << y;
```

正如 iosream 中定义的那样，<<运算符要求左边是一个 ostream 对象。显然，因为 cout 是 ostream 对象，所以表达式 cout << x 满足这种要求。然而，因为表达式 cout << x 位于<< y 的左侧，所以输出语句也要求该表达式是一个 ostream 类型的对象。因此，ostream 类将 operator<<( )函数实现为返回一个指向 ostream 对象的引用。具体地说，它返回一个指向调用对象（这里是 cout）的引用。因此，表达式(cout << x)本身就是 ostream 对象 cout，从而可以位于<<运算符的左侧。

可以对友元函数采用相同的方法。只要修改 operator<<( )函数，让它返回 ostream 对象的引用即可：

```
ostream & operator<<(ostream & os, const Time & t)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
    return os;
}
```

注意，返回类型是 ostream &。这意味着该函数返回 ostream 对象的引用。因为函数开始执行时，程序传递了一个对象引用给它，这样做的最终结果是，函数的返回值就是传递给它的对象。也就是说，下面的语句：

```
cout << trip;
```

将被转换为下面的调用：

```
operator<<(cout, trip);
```

而该调用返回 cout 对象。因此，下面的语句可以正常工作：

```
cout << "Trip time: " << trip << " (Tuesday)\n"; // can do
```

我们将这条语句分成多步，来看看它是如何工作的。首先，下面的代码调用 ostream 中的<<定义，它显示字符串并返回 cout 对象：

```
cout << "Trip time: "
```

因此表达式 cout << “Trip time:”将显示字符串，然后被它的返回值 cout 所替代。原来的语句被简化为下面的形式：

```
cout << trip << " (Tuesday)\n";
```

接下来，程序使用<<的 Time 声明显示 trip 值，并再次返回 cout 对象。这将语句简化为：

```
cout << " (Tuesday)\n";
```

现在，程序使用 ostream 中用于字符串的<<定义，来显示最后一个字符串，并结束运行。

有趣的是，这个 operator<<( )版本还可用于将输出写入到文件中：

```
#include <fstream>
...
ofstream fout;
fout.open("savetime.txt");
Time trip(12, 40);
fout << trip;
```

其中最后一条语句将被转换为这样：

```
operator<<(fout, trip);
```

另外，正如第 8 章指出的，类继承属性让 ostream 引用能够指向 ostream 对象和 ofstream 对象。

提示：一般来说，要重载<<运算符来显示 c_name 的对象，可使用一个友元函数，其定义如下：

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

程序清单 11.10 列出了修改后的类定义，其中包括 operator*( )和 operator<<( )这两个友元函数。它将第一个友元函数作为内联函数，因为其代码很短。（当定义同时也是原型时，就像这个例子中那样，要使用 friend 前缀。）

警告：只有在类声明中的原型中才能使用 friend 关键字。除非函数定义也是原型，否则不能在函数定义中使用该关键字。

程序清单 11.10　mytime3.h

```
// mytime3.h -- Time class with friends
#ifndef MYTIME3_H_
#define MYTIME3_H_
#include <iostream>

class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    friend Time operator*(double m, const Time & t)
        { return t * m; }   // inline definition
    friend std::ostream & operator<<(std::ostream & os, const Time & t);
};

#endif
```

程序清单 11.11 列出了修改后的定义。方法使用了 Time::限定符，而友元函数不使用该限定符。另外，由于在 mytime3.h 中包含了 iostream 并提供了 using 声明 std::ostream，因此在 mytime3.cpp 中包含 mytime3.h 后，便提供了在实现文件中使用 ostream 的支持。

程序清单 11.11　mytime3.cpp

```
// mytime3.cpp  -- implementing Time methods
#include "mytime3.h"

Time::Time()
{
    hours = minutes = 0;
}

Time::Time(int h, int m )
{
```

```
    hours = h;
    minutes = m;
}

void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}

void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}

Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

Time Time::operator-(const Time & t) const
{
    Time diff;
    int tot1, tot2;
    tot1 = t.minutes + 60 * t.hours;
    tot2 = minutes + 60 * hours;
    diff.minutes = (tot2 - tot1) % 60;
    diff.hours = (tot2 - tot1) / 60;
    return diff;
}

Time Time::operator*(double mult) const
{
    Time result;
    long totalminutes = hours * mult * 60 + minutes * mult;
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}

std::ostream & operator<<(std::ostream & os, const Time & t)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
    return os;
}
```

程序清单 11.12 是一个示例程序。从技术上说，在 usetime3.cpp 中不必包含头文件 iostream，因为在 mytime3.h 中已经包含了该文件。然而，作为 Time 类的用户，您并不知道在类代码文件中已经包含了哪些文件，因此您应负责将您编写的代码所需的头文件包含进来。

程序清单 11.12　usetime3.cpp

```
//usetime3.cpp -- using the fourth draft of the Time class
// compile usetime3.cpp and mytime3.cpp together
#include <iostream>
#include "mytime3.h"

int main()
{
   using std::cout;
   using std::endl;
   Time aida(3, 35);
   Time tosca(2, 48);
   Time temp;

   cout << "Aida and Tosca:\n";
   cout << aida<<"; " << tosca << endl;
   temp = aida + tosca;     // operator+()
   cout << "Aida + Tosca: " << temp << endl;
   temp = aida* 1.17;  // member operator*()
   cout << "Aida * 1.17: " << temp << endl;
   cout << "10.0 * Tosca: " << 10.0 * tosca << endl;

   return 0;
}
```

下面是程序清单 11.10～程序清单 11.12 组成的程序的输出：

```
Aida and Tosca:
3 hours, 35 minutes; 2 hours, 48 minutes
Aida + Tosca: 6 hours, 23 minutes
Aida * 1.17: 4 hours, 11 minutes
10.0 * Tosca: 28 hours, 0 minutes
```

11.4　重载运算符：作为成员函数还是非成员函数

对于很多运算符来说，可以选择使用成员函数或非成员函数来实现运算符重载。一般来说，非成员函数应是友元函数，这样它才能直接访问类的私有数据。例如，Time 类的加法运算符在 Time 类声明中的原型如下：

```
Time operator+(const Time & t) const; // member version
```

这个类也可以使用下面的原型：

```
// nonmember version
friend Time operator+(const Time & t1, const Time & t2);
```

加法运算符需要两个操作数。对于成员函数版本来说，一个操作数通过 this 指针隐式地传递，另一个操作数作为函数参数显式地传递；对于友元版本来说，两个操作数都作为参数来传递。

注意：非成员版本的重载运算符函数所需的形参数目与运算符使用的操作数数目相同；而成员版本所需的参数数目少一个，因为其中的一个操作数是被隐式地传递的调用对象。

这两个原型都与表达式 T2 + T3 匹配，其中 T2 和 T3 都是 Time 类型对象。也就是说，编译器将下面的语句：

```
T1 = T2 + T3;
```

转换为下面两个的任何一个：

```
T1 = T2.operator+(T3);     // member function
T1 = operator+(T2, T3);    // nonmember function
```

记住，在定义运算符时，必须选择其中的一种格式，而不能同时选择这两种格式。因为这两种格式都与同一个表达式匹配，同时定义这两种格式将被视为二义性错误，导致编译错误。

那么哪种格式最好呢？对于某些运算符来说（如前所述），成员函数是唯一合法的选择。在其他情况下，这两种格式没有太大的区别。有时，根据类设计，使用非成员函数版本可能更好（尤其是为类定义类型转换时）。本章后面的“转换和友元”一节将更深入地讨论这种情形。

11.5 再谈重载：一个矢量类

下面介绍另一种使用了运算符重载和友元的类设计——一个表示矢量的类。这个类还说明了类设计的其他方面，例如，在同一个对象中包含两种描述同一样东西的不同方式等。即使并不关心矢量，也可以在其他情况下使用这里介绍的很多新技术。矢量（vector），是工程和物理中使用的一个术语，它是一个有大小和方向的量。例如，推东西时，推的效果将取决于推力的大小和推的方向。从某个方向推可能会省力，而从相反的方向推则要费很大的劲。为完整地描述汽车的运动情况，应指出其运动速度（大小）和运动方向；如果逆行，则向高速公路的巡警辩解没有超速、超载是徒劳的（免疫学家和计算机专家使用术语矢量的方式不同，请不要考虑这一点，至少在第 16 章介绍计算机科学版本——vector 模板类之前应如此）。下面的旁注介绍了更多有关矢量的知识，但对于下面的 C++示例来说，并不必完全理解这些知识。

矢量

假设工蜂发现了一个非凡的花蜜储藏处，它匆忙返回蜂巢，告知其他蜜蜂，该花蜜储藏处离蜂巢 120 码。“这种信息是不完整的”，其他蜜蜂感到很茫然——“还必须告知方向！”，该工蜂答道：“太阳方向偏北 30 度”。知道了距离（大小）和方向，其他的蜜蜂能很快找到蜜源。蜜蜂懂得矢量。

许多数量都有大小和方向。例如，推的效果取决于力气的大小和方向。在计算机屏幕上移动对象时也涉及到距离和方向。可以使用矢量来描述这类问题。例如，可以用矢量来描述如何在屏幕上移动（放置）对象，即用箭头从起始位置画到终止位置，来对它作形象化处理。矢量的长度是其大小——描述了移动的距离；箭头的指向描述了方向（参见图 11.1）。表示这种位置变化的矢量称为位移矢量（displacement vector）。

现在，假设您是 Lhanappa——伟大的毛象猎手。猎狗报告毛象群位于西北 14.1 公里处。但由于当时刮的是东南风，您不想从东南方向接近毛象群，因此先向西走了 10 公里，再向北走了 10 公里，最终从南面接近毛象群。您知道这两个位移矢量与指向西北的 14.1 公里的矢量的方向相同。伟大的毛象猎手 Lhanappa 也知道如何将两个矢量相加。

将两个矢量相加有一种简单的几何解释。首先，画一个矢量，然后从第一个矢量的尾部开始画第二个矢量。最后从第一个矢量的开始处向第二个矢量的结尾处画一个矢量。第三个矢量表示前两个矢量的和（参见图 11.2）。注意，两个矢量之和的长度可能小于它们的长度之和。

显然，应为矢量重载运算符。首先，无法用一个数来表示矢量，因此应创建一个类来表示矢量。其次，矢量与普通数学运算（如加法、减法）有相似之处。这种相似表明，应重载运算符，使之能用于矢量。

出于简化的目的，本节将实现一个二维矢量（如屏幕位移），而不是三维矢量（如表示直升机或体操运动员的运动情况）。描述二维矢量只需两个数，但可以选择到底使用哪两个数：

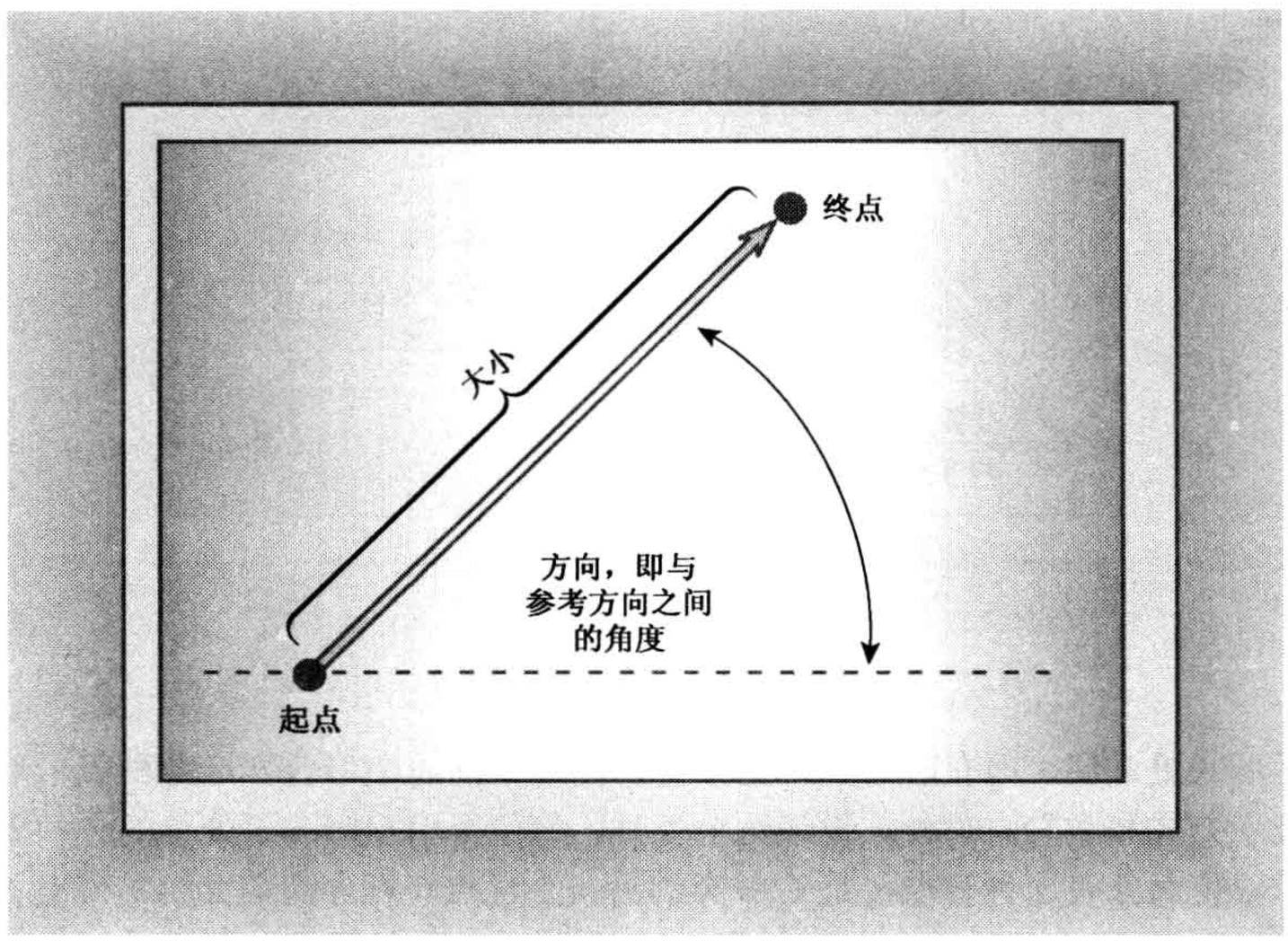


图 11.1　使用矢量描述位移

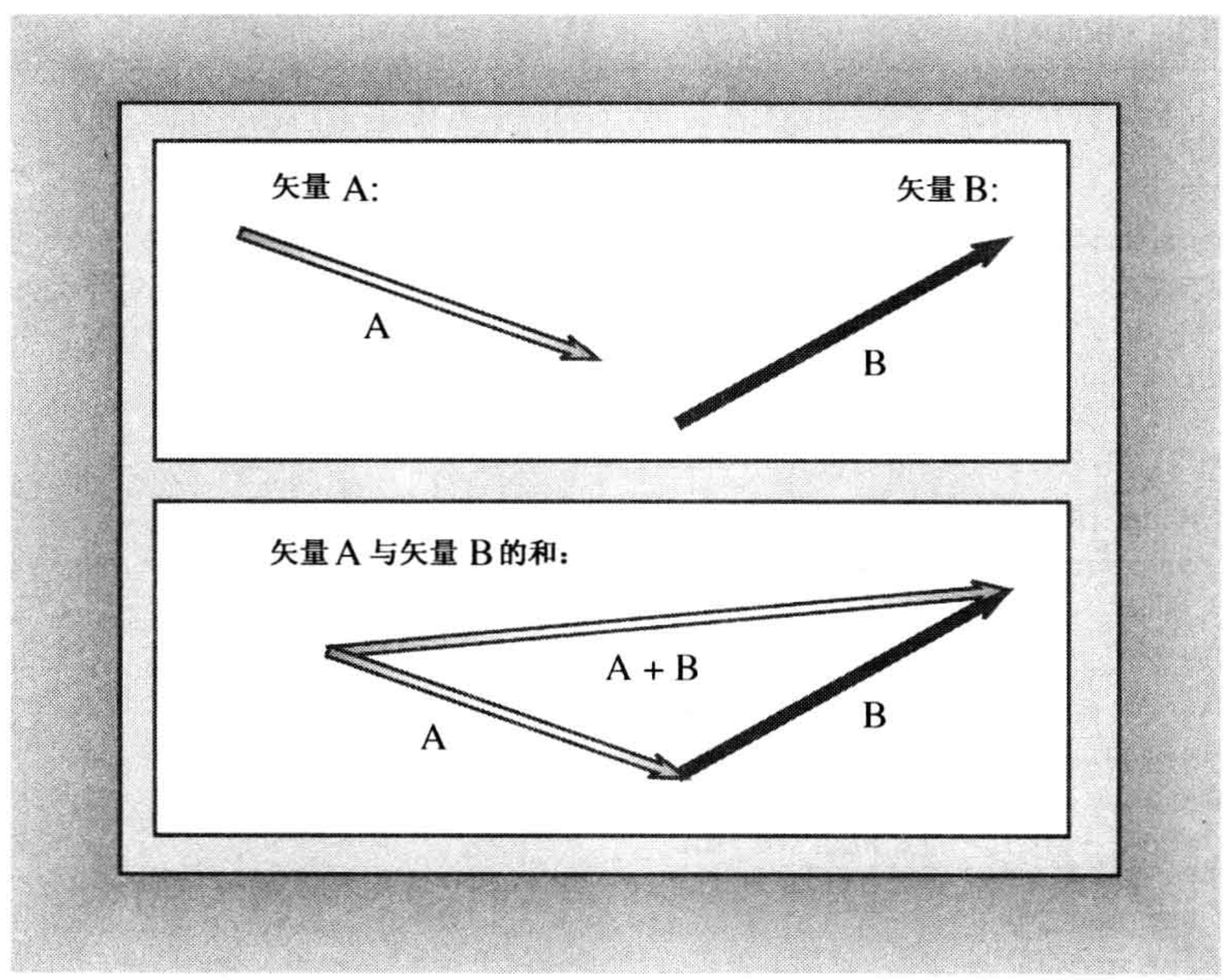


图 11.2　将两个矢量相加

- 可以用大小（长度）和方向（角度）描述矢量；
- 可以用分量 x 和 y 表示矢量。

两个分量分别是水平矢量（x 分量）和垂直矢量（y 分量），将其相加可以得到最终的矢量。例如，可以这样描述点的运动：向右移动 30 个单位，再向上移动 40 个单位（参见图 11.3）。这将把该点沿与水平方向呈 53.1 度的方向移动 50 个单位，因此，水平分量为 30 个单位、垂直分量为 40 个单位的矢量，与长度为 50 个单位、方向为 53.1 度的矢量相同。位移矢量指的是从何处开始、到何处结束，而不是经过的路线。这种表示基本上和第 7 章在直角坐标与极坐标之间转换的程序中介绍的相同。

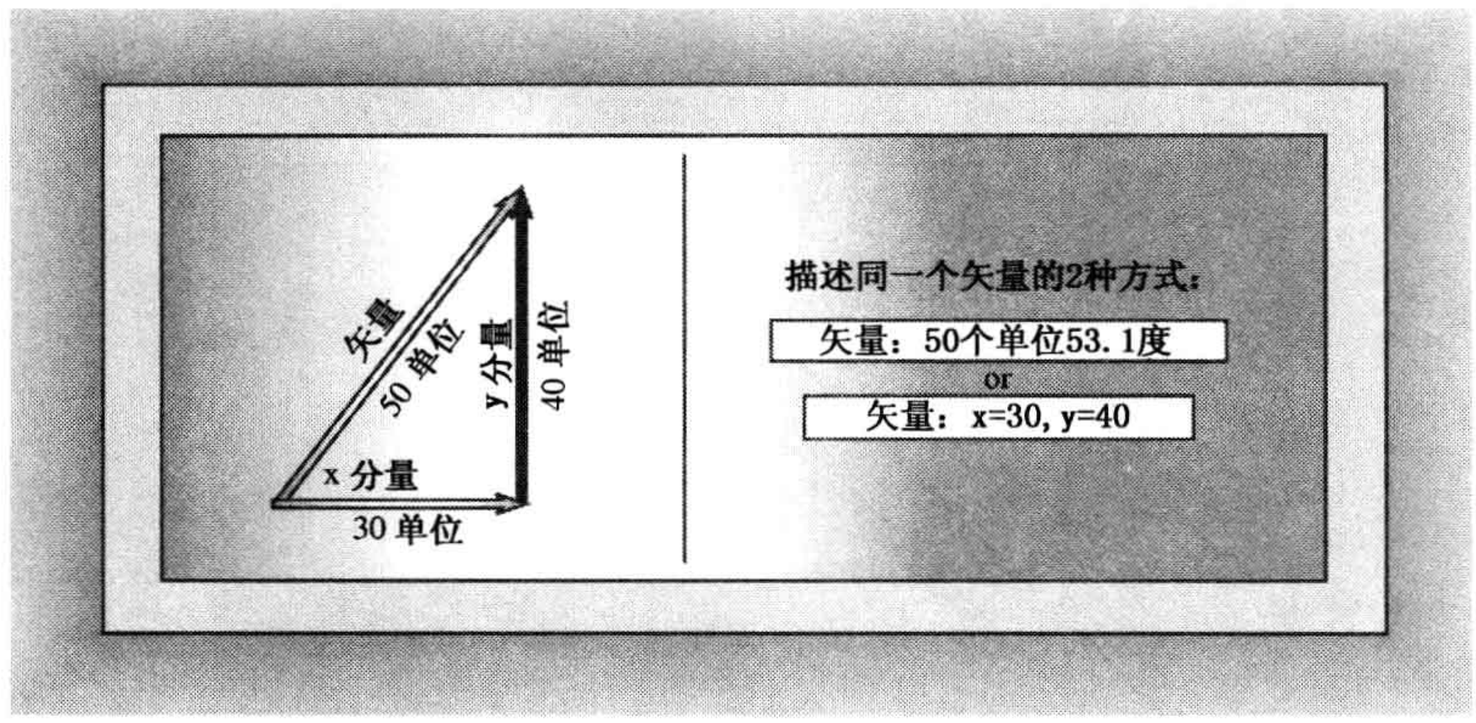


图 11.3　矢量的 x 和 y 分量

有时一种表示形式更方便，而有时另一种更方便，因此类描述中将包含这两种表示形式（参见本章后面的旁注“多种表示方式和类”）。另外，设计这个类时，将使得用户修改了矢量的一种表示后，对象将自动更新另一种表示。使对象有这种智能，是 C++类的另一个优点。程序清单 11.13 列出了这个类的声明。为复习名称空间，该清单将类声明放在 VECTOR 名称空间中。另外，该程序使用枚举创建了两个常量（RECT 和 POL），用于标识两种表示法（枚举在第 10 章介绍过，因此这里直接使用它）。

程序清单 11.13　vector.h

```
// vect.h -- Vector class with <<, mode state
#ifndef VECTOR_H_
#define VECTOR_H_
#include <iostream>
namespace VECTOR
{
    class Vector
    {
    public:
        enum Mode {RECT, POL};
    // RECT for rectangular, POL for Polar modes
    private:
        double x;          // horizontal value
        double y;          // vertical value
        double mag;        // length of vector
        double ang;        // direction of vector in degrees
        Mode mode;         // RECT or POL
    // private methods for setting values
        void set_mag();
        void set_ang();
        void set_x();
        void set_y();
    public:
       Vector();
        Vector(double n1, double n2, Mode form = RECT);
        void reset(double n1, double n2, Mode form = RECT);
        ~Vector();
        double xval() const {return x;}       // report x value
        double yval() const {return y;}       // report y value
        double magval() const {return mag;}   // report magnitude
```

```
    double angval() const {return ang;}   // report angle
    void polar_mode();                    // set mode to POL
    void rect_mode();                     // set mode to RECT
// operator overloading
    Vector operator+(const Vector & b) const;
    Vector operator-(const Vector & b) const;
    Vector operator-() const;
    Vector operator*(double n) const;
// friends
    friend Vector operator*(double n, const Vector & a);
    friend std::ostream &
           operator<<(std::ostream & os, const Vector & v);
  };

}   // end namespace VECTOR
#endif
```

注意，程序清单 11.13 中 4 个报告分量值的函数是在类声明中定义的，因此将自动成为内联函数。这些函数非常短，因此适于声明为内联函数。因为它们都不会修改对象数据，所以声明时使用了 const 限定符。第 10 章介绍过，这种句法用于声明那些不会对其显式访问的对象进行修改的函数。

程序清单 11.14 列出了程序清单 11.13 中声明的方法和友元函数的定义，该清单利用了名称空间的开放性，将方法定义添加到 VECTOR 名称空间中。请注意，构造函数和 reset( )函数都设置了矢量的直角坐标和极坐标表示，因此需要这些值时，可直接使用而无需进行计算。另外，正如第 4 章和第 7 章指出的，C++的内置数学函数在使用角度时以弧度为单位，所以函数在度和弧度之间进行转换。该 Vector 类实现对用户隐藏了极坐标和直角坐标之间的转换以及弧度和度之间的转换等内容。用户只需知道：类在使用角度时以度为单位，可以使用两种等价的形式来表示矢量。

程序清单 1.14　vector.cpp

```
// vect.cpp -- methods for the Vector class
#include <cmath>
#include "vect.h"   // includes <iostream>
using std::sqrt;
using std::sin;
using std::cos;
using std::atan;
using std::atan2;
using std::cout;

namespace VECTOR
{
    // compute degrees in one radian
    const double Rad_to_deg = 45.0 / atan(1.0);
    // should be about 57.2957795130823

    // private methods
    // calculates magnitude from x and y
    void Vector::set_mag()
    {
        mag = sqrt(x * x + y * y);
    }

    void Vector::set_ang()
    {
```

```
    if (x == 0.0 && y == 0.0)
        ang = 0.0;
    else
        ang = atan2(y, x);
}

// set x from polar coordinate
void Vector::set_x()
{
    x = mag * cos(ang);
}

// set y from polar coordinate
void Vector::set_y()
{
    y = mag * sin(ang);
}
// public methods
Vector::Vector()              // default constructor
{
    x = y = mag = ang = 0.0;
    mode = RECT;
}

// construct vector from rectangular coordinates if form is r
// (the default) or else from polar coordinates if form is p
Vector::Vector(double n1, double n2, Mode form)
{
    mode = form;
    if (form == RECT)
     {
         x = n1;
         y = n2;
         set_mag();
         set_ang();
    }
    else if (form == POL)
    {
         mag = n1;
         ang = n2 / Rad_to_deg;
         set_x();
         set_y();
    }
    else
    {
         cout << "Incorrect 3rd argument to Vector() -- ";
         cout << "vector set to 0\n";
         x = y = mag = ang = 0.0;
         mode = RECT;
    }
}

// reset vector from rectangular coordinates if form is
// RECT (the default) or else from polar coordinates if
// form is POL
```

```
void Vector:: reset(double n1, double n2, Mode form)
{
    mode = form;
    if (form == RECT)
     {
         x = n1;
         y = n2;
         set_mag();
         set_ang();
     }
     else if (form == POL)
     {
         mag = n1;
         ang = n2 / Rad_to_deg;
         set_x();
         set_y();
     }
     else
     {
         cout << "Incorrect 3rd argument to Vector() -- ";
         cout << "vector set to 0\n";
         x = y = mag = ang = 0.0;
         mode = RECT;
     }
}

Vector::~Vector()    // destructor
{
}

void Vector::polar_mode()    // set to polar mode
{
    mode = POL;
}

void Vector::rect_mode()     // set to rectangular mode
{
    mode = RECT;
}

// operator overloading
// add two Vectors
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}

// subtract Vector b from a
Vector Vector::operator-(const Vector & b) const
{
    return Vector(x - b.x, y - b.y);
}

// reverse sign of Vector
```

```
Vector Vector::operator-() const
{
    return Vector(-x, -y);
}

// multiply vector by n
Vector Vector::operator*(double n) const
{
    return Vector(n * x, n * y);
}

// friend methods
// multiply n by Vector a
Vector operator*(double n, const Vector & a)
{
    return a * n;
}

// display rectangular coordinates if mode is RECT,
// else display polar coordinates if mode is POL
std::ostream & operator<<(std::ostream & os, const Vector & v)
{
    if (v.mode == Vector::RECT)
         os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == Vector::POL)
    {
         os << "(m,a) = (" << v.mag << ", "
                << v.ang * Rad_to_deg << ")";
    }
    else
         os << "Vector object mode is invalid";
    return os;
}

} // end namespace VECTOR
```

也可以以另一种方式来设计这个类。例如，在对象中存储直角坐标而不是极坐标，并使用方法 magval( ) 和 angval( )来计算极坐标。对于很好进行坐标转换的应用来说，这将是一种效率更高的设计。另外，方法 reset( )并非必不可少的。假设 shove 是一个 Vector 对象，而您编写了如下代码：

```
shove.reset(100,300);
```

可以使用构造函数来得到相同的结果：

```
shove = Vector(100,300);    // create and assign a temporary object
```

然而，方法 set( )直接修改 shove 的内容，而使用构造函数将增加额外的步骤：创建一个临时对象，然后将其赋给 shove。

这些设计决策遵守了 OOP 传统，即将类接口的重点放在其本质上（抽象模型），而隐藏细节。这样，当用户使用 Vector 类时，只需考虑矢量的通用特性，例如，矢量可以表示位移，可以将两个矢量相加等。使用分量还是大小和方向来表示矢量已无关紧要，因为程序员可以设置矢量的值，并选择最方便的格式来显示它们。

下面更详细地介绍 Vector 类的一些特性。

11.5.1 使用状态成员

Vector 类储存了矢量的直角坐标和极坐标。它使用名为 mode 的成员来控制使用构造函数、reset( )方法

和重载的 operator<<( )函数使用哪种形式，其中枚举 RECT 表示直角坐标模式（默认值）、POL 表示极坐标模式。这样的成员被称为状态成员（state member），因为这种成员描述的是对象所处的状态。要知道具体含义，请看构造函数的代码：

```
Vector::Vector(double n1, double n2, Mode form)
{
    mode = form;
    if (form == RECT)
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == POL)
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Incorrect 3rd argument to Vector() -- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = RECT;
    }
}
```

如果第三个参数是 RECT 或省略了（原型将默认值设置为 RECT），则将输入解释为直角坐标；如果为 POL，则将输入解释为极坐标：

```
Vector folly(3.0, 4.0);              // set x = 3, y = 4
Vector foolery(20.0, 30.0, VECTOR::Vector::POL);   // set mag = 20, ang = 30
```

标识符 POL 的作用域为类，因此类定义可使用未限定的名称。但全限定名为 VECTOR::Vector::POL，因为 POL 是在 Vector 类中定义的，而 Vector 是在名称空间 VECTOR 中定义的。注意，如果用户提供的是 x 值和 y 值，则构造函数将使用私有方法 set_mag( )和 set_ang( )来设置距离和角度值；如果提供的是距离和角度值，则构造函数将使用 set_x( )和 set_y( )方法来设置 x 值和 y 值。另外，如果用户指定的不是 RECT 或 POL，则构造函数将显示一条警告消息，并将状态设置为 RECT。

看起来好像难以将 RECT 和 POL 外的其他值传递给构造函数，因为第三个参数的类型为 VECTOR::Vector::Mode。像下面这样的调用无法通过编译，因为诸如 2 等整数不能隐式地转换为枚举类型：

```
Vector rector(20.0, 30.0, 2);  // type mismatch  - 2 not an enum type
```

然而，机智而好奇的用户可尝试下面这样的代码，看看结果如何：

```
Vector rector(20.0, 30.0, VECTOR::Vector::Mode (2));  // type cast
```

就这里而言，编译器将发出警告。

接下来，operator<<( )函数也使用模式来确定如何显示值：

```
// display rectangular coordinates if mode is RECT,
// else display polar coordinates if mode is POL
std::ostream & operator<<(std::ostream & os, const Vector & v)
{
    if (v.mode == Vector::RECT)
```

```
        os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == Vector::POL)
    {
        os << "(m,a) = (" << v.mag << ", "
            << v.ang * Rad_to_deg << ")";
    }
    else
        os << "Vector object mode is invalid";
    return os;
}
```

由于 operator<<()是一个友元函数，而不在类作用域内，因此必须使用 Vector::RECT，而不能使用 RECT。但这个友元函数在名称空间 VECTOR 中，因此无需使用全限定名 VECTOR:: Vector::RECT。

设置模式的各种方法只接受 RECT 和 POL 为合法值，因此该函数中的 else 永远不会执行。但进行检查还是一个不错的主意，它有助于捕获难以发现的编程错误。

多种表示方式和类

可以用不同但等价的方式表示的量很常见。例如，可以按每加仑汽油消耗汽车能行驶的英里数来计算油耗（美国），也可以按每 100 公里消耗多少公升汽油来计算（欧洲）。可以用字符串表示数字，也可以用数值方式表示，可以使用 IQ 或 kiloturkey 的方法表示智商。类非常适于在一个对象中表示实体的不同方面。首先在一个对象中存储多种表示方式；然后，编写这样的类函数，以便给一种表示方式赋值时，将自动给其他表示方式赋值。例如，Vector 类的 set_by_polar( )方法将 mag 和 ang 成员设置为函数参数的值，并同时设置成员 x 和 y。也可存储一种表示方式，并使用方法来提供其他表示方式。通过在内部处理转换，类允许从本质（而不是表示方式）上来看待一个量。

11.5.2 为 Vector 类重载算术运算符

在使用 x、y 坐标时，将两个矢量相加将非常简单，只要将两个 x 分量相加，得到最终的 x 分量，将两个 y 分量相加，得到最终的 y 分量即可。根据这种描述，可能使用下面的代码：

```
Vector Vector::operator+(const Vector & b) const
{
    Vector sum;
    sum.x = x + b.x;
    sum.y = y + b.y;
    return sum;          // incomplete version
}
```

如果对象只存储 x 和 y 分量，则这很好。遗憾的是，上述代码无法设置极坐标值。可以通过添加另外一些代码来解决这种问题：

```
Vector Vector::operator+(const Vector & b) const
{
    Vector sum;
    sum.x = x + b.x;
    sum.y = y + b.y;
    sum.set_ang(sum.x, sum.y);
    sum.set_mag(sum.x, sum.y);
    return sum;          // version duplicates needlessly
}
```

然而，使用构造函数来完成这种工作，将更简单、更可靠：

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);      // return the constructed Vector
}
```

上述代码将新的 x 分量和 y 分量传递给 Vector 构造函数，而后者将使用这些值来创建无名的新对象，并返回该对象的副本。这确保了新的 Vector 对象是根据构造函数制定的标准规则创建的。

提示：如果方法通过计算得到一个新的类对象，则应考虑是否可以使用类构造函数来完成这种工作。这样做不仅可以避免麻烦，而且可以确保新的对象是按照正确的方式创建的。

1. 乘法

将矢量与一个数相乘，将使该矢量加长或缩短（取决于这个数）。因此，将矢量乘以 3 得到的矢量的长度为原来的三倍，而方向不变。要在 Vector 类中实现矢量的这种行为很容易。对于极坐标，只要将长度进行伸缩，并保持角度不变即可；对于直角坐标，只需将 x 和 y 分量进行伸缩即可。也就是说，如果矢量的分量为 5 和 12，则将其乘以 3 后，分量将分别是 15 和 36。这正是重载的乘法运算符要完成的工作：

```
Vector Vector::operator*(double n) const
{
    return Vector(n * x, n * y);
}
```

和重载加法一样，上述代码允许构造函数使用新的 x 和 y 分量来创建正确的 Vector 对象。上述函数用于处理 Vector 值和 double 值相乘。可以像 Time 示例那样，使用一个内联友元函数来处理 double 与 Vector 相乘：

```
Vector operator*(double n, const Vector & a)  // friend function
{
    return a * n;   // convert double times Vector to Vector times double
}
```

2. 对已重载的运算符进行重载

在 C++中，–运算符已经有两种含义。首先，使用两个操作数，它是减法运算符。减法运算符是一个二元运算符，因为它有两个操作数。其次，使用一个操作数时（如–x），它是负号运算符。这种形式被称为一元运算符，即只有一个操作数。对于矢量来说，这两种操作（减法和符号反转）都是有意义的，因此 Vector 类有这两种操作。

要从矢量 A 中减去矢量 B，只要将分量相减即可，因此重载减法与重载加法相似：

```
Vector operator-(const Vector & b) const;            // prototype
Vector Vector::operator-(const Vector & b) const     // definition
{
    return Vector(x - b.x, y - b.y);  // return the constructed Vector
}
```

操作数的顺序非常重要。下面的语句：

```
diff = v1 - v2;
```

将被转换为下面的成员函数调用：

```
diff = v1.operator-(v2);
```

这意味着将从隐式矢量参数减去以显式参数传递的矢量，所以应使用 x – b.x，而不是 b.x – x。

接下来，来看一元负号运算符，它只使用一个操作数。将这个运算符用于数字（如–x）时，将改变它的符号。因此，将这个运算符用于矢量时，将反转矢量的每个分量的符号。更准确地说，函数应返回一个与原来的矢量相反的矢量（对于极坐标，长度不变，但方向相反）。下面是重载负号的原型和定义：

```
Vector operator-() const;
Vector Vector::operator-() const
{
    return Vector (-x, -y);
}
```

现在，operator-( )有两种不同的定义。这是可行的，因为它们的特征标不同。可以定义-运算符的一元和二元版本，因为 C++提供了该运算符的一元和二元版本。对于只有二元形式的运算符（如除法运算符），只能将其重载为二元运算符。

注意：因为运算符重载是通过函数来实现的，所以只要运算符函数的特征标不同，使用的运算符数量与相应的内置 C++运算符相同，就可以多次重载同一个运算符。

11.5.3 对实现的说明

前几节介绍的实现在 Vector 对象中存储了矢量的直角坐标和极坐标，但公有接口并不依赖于这一事实。所有接口都只要求能够显示这两种表示，并可以返回各个值。内部实现方式可以完全不同。正如前面指出的，对象可以只存储 x 和 y 分量，而返回矢量长度的 magval( )方法可以根据 x 和 y 的值来计算出长度，而不是查找对象中存储的这个值。这种方法改变了实现，但用户接口不变。将接口与实现分离是 OOP 的目标之一，这样允许对实现进行调整，而无需修改使用这个类的程序中的代码。

这两种实现各有利弊。存储数据意味着对象将占据更多的内存，每次 Vector 对象被修改时，都需要更新直角坐标和极坐标表示；但查找数据的速度比较快。如果应用程序经常需要访问矢量的这两种表示，则这个例子采用的实现比较合适；如果只是偶尔需要使用极坐标，则另一种实现更好。可以在一个程序中使用一种实现，而在另一个程序中使用另一种实现，但它们的用户接口相同。

11.5.4 使用 Vector 类来模拟随机漫步

程序清单 11.15 是一个小程序，它使用了修订后的 Vector 类。该程序模拟了著名的醉鬼走路问题（Drunkard Walk problem）。实际上，醉鬼被认为是一个有许多健康问题的人，而不是大家娱乐消遣的谈资，因此这个问题通常被称为随机漫步问题。其意思是，将一个人领到街灯柱下。这个人开始走动，但每一步的方向都是随机的（与前一步不同）。这个问题的一种表述是，这个人走到离灯柱 50 英尺处需要多少步。从矢量的角度看，这相当于不断将方向随机的矢量相加，直到长度超过 50 英尺。

程序清单 11.15 允许用户选择行走距离和步长。该程序用一个变量来表示位置（一个矢量），并报告到达指定距离处（用两种格式表示）所需的步数。可以看到，行走者前进得相当慢。虽然走了 1000 步，每步的距离为 2 英尺，但离起点可能只有 50 英尺。这个程序将行走者所走的净距离（这里为 50 英尺）除以步数，来指出这种行走方式的低效性。随机改变方向使得该平均值远远小于步长。为了随机选择方向，该程序使用了标准库函数 rand( )、srand( )和 time( )（参见程序说明）。请务必将程序清单 11.14 和程序清单 11.15 一起进行编译。

程序清单 11.15 randwalk.cpp

```
// randwalk.cpp -- using the Vector class
// compile with the vect.cpp file
#include <iostream>
#include <cstdlib>      // rand(), srand() prototypes
#include <ctime>        // time() prototype
#include "vect.h"
int main()
{
    using namespace std;
    using VECTOR::Vector;
    srand(time(0));     // seed random-number generator
    double direction;
    Vector step;
    Vector result(0.0, 0.0);
    unsigned long steps = 0;
    double target;
```

```
    double dstep;
    cout << "Enter target distance (q to quit): ";
    while (cin >> target)
    {
        cout << "Enter step length: ";
        if (!(cin >> dstep))
            break;

        while (result.magval() < target)
        {
            direction = rand() % 360;
            step.reset(dstep, direction, Vector::POL);
            result = result + step;
            steps++;
        }
        cout << "After " << steps << " steps, the subject "
            "has the following location:\n";
        cout << result << endl;
        result.polar_mode();
        cout << " or\n" << result << endl;
        cout << "Average outward distance per step = "
            << result.magval()/steps << endl;
        steps = 0;
        result.reset(0.0, 0.0);
        cout << "Enter target distance (q to quit): ";
    }
    cout << "Bye!\n";
    cin.clear();
    while (cin.get() != '\n')
        continue;
    return 0;
}
```

该程序使用 using 声明导入了 Vector，因此该程序可使用 Vector::POL，而不必使用 VECTOR::Vector::POL。

下面是程序清单 11.13～程序清单 11.15 组成的程序的运行情况：

```
Enter target distance (q to quit): 50
Enter step length: 2
After 253 steps, the subject has the following location:
(x,y) = (46.1512, 20.4902)
 or
(m,a) = (50.495, 23.9402)
Average outward distance per step = 0.199587
Enter target distance (q to quit): 50
Enter step length: 2
After 951 steps, the subject has the following location:
(x,y) = (-21.9577, 45.3019)
 or
(m,a) = (50.3429, 115.8593)
Average outward distance per step = 0.0529362
Enter target distance (q to quit): 50
Enter step length: 1
After 1716 steps, the subject has the following location:
(x,y) = (40.0164, 31.1244)
 or
```

```
(m,a) = (50.6956, 37.8755)
Average outward distance per step = 0.0295429
Enter target distance (q to quit): q
Bye!
```

这种处理的随机性使得每次运行结果都不同，即使初始条件相同。然而，平均而言，步长减半，步数将为原来的 4 倍。概率理论表明，平均而言，步数（N）、步长（s），净距离 D 之间的关系如下：

$N = (D/s)^2$

这只是平均情况，但每次试验结果可能相差很大。例如，进行 1000 次试验（走 50 英尺，步长为 2 英尺）时，平均步数为 636（与理论值 625 非常接近），但实际步数位于 91～3951。同样，进行 1000 次试验（走 50 英尺，步长为 1 英尺）时，平均步数为 2557（与理论值 2500 非常接近），但实际步数位于 345～10882。因此，如果发现自己在随机漫步时，请保持自信，迈大步走。虽然在蜿蜒前进的过程中仍旧无法控制前进的方向，但至少会走得远一点。

程序说明

首先需要指出的是，在程序清单 11.15 中使用 VECTOR 名称空间非常方便。下面的 using 声明使 Vector 类的名称可用：

```
using VECTOR::Vector;
```

因为所有的 Vector 类方法的作用域都为整个类，所以导入类名后，无需提供其他 using 声明，就可以使用 Vector 的方法。

接下来谈谈随机数。标准 ANSI C 库（C++也有）中有一个 rand( )函数，它返回一个从 0 到某个值（取决于实现）之间的随机整数。该程序使用求模操作数来获得一个 0～359 的角度值。rand( )函数将一种算法用于一个初始种子值来获得随机数，该随机值将用作下一次函数调用的种子）依此类推。这些数实际上是伪随机数，因为 10 次连续的调用通常将生成 10 个同样的随机数（具体值取决于实现）。然而，srand( )函数允许覆盖默认的种子值，重新启动另一个随机数序列。该程序使用 time（0）的返回值来设置种子。time（0）函数返回当前时间，通常为从某一个日期开始的秒数（更广义地，time( )接受 time_t 变量的地址，将时间放到该变量中，并返回它。将 0 用作地址参数，可以省略 time_t 变量声明）。因此，下面的语句在每次运行程序时，都将设置不同的种子，使随机输出看上去更为随机：

```
srand(time(0));
```

头文件 cstdlib（以前为 stdlib.h）包含了 srand( )和 rand( )的原型，而 ctime（以前是 time.h）包含了 time( )的原型。C++11 使用头文件 radom 中的函数提供了更强大的随机数支持。

该程序使用 result 矢量记录行走者的前进情况。内循环每轮将 step 矢量设置为新的方向，并将它与当前的 result 矢量相加。当 result 的长度超过指定的距离后，该循环结束。

程序通过设置矢量的模式，用直角坐标和极坐标显示最终的位置。

下面这条语句将 result 设置为 RECT 模式，而不管 result 和 step 的初始模式是什么：

```
result = result + step;
```

这样做的原因如下。首先，加法运算符函数创建并返回一个新矢量，该矢量存储了这两个参数的和。该函数使用默认构造函数以 RECT 模式创建矢量。因此，被赋给 result 的矢量的模式为 RECT。默认情况下，赋值时将分别给每个成员变量赋值，因此将 RECT 赋给了 result.mode。如果偏爱其他方式，例如，result 保留原来的模式，可以通过为类定义赋值运算符来覆盖默认的赋值方式。第 12 章将介绍这样的示例。

顺便说一句，在将一系列位置存储到文件中很容易。首先包含头文件 fstream，声明一个 ofstream 对象，将其同一个文件关联起来：

```
#include <fstream>
...
ofstream fout;
fout.open("thewalk.txt");
```

然后，在计算结果的循环中加入类似于下面的代码：

```
fout << result << endl;
```

这将调用友元函数 operator<<(fout, result)，导致引用参数 os 指向 fout，从而将输出写入到文件中。您还可以使用 fout 将其他信息写入到文件中，如当前由 cout 显示的总结信息。

11.6　类的自动转换和强制类型转换

下面介绍类的另一个主题——类型转换。本节讨论 C++如何处理用户定义类型的转换。在讨论这个问题之前，我们先来复习一下 C++是如何处理内置类型转换的。将一个标准类型变量的值赋给另一种标准类型的变量时，如果这两种类型兼容，则 C++自动将这个值转换为接收变量的类型。例如，下面的语句都将导致数值类型转换：

```
long count = 8;       // int value 8 converted to type long
double time = 11;     // int value 11 converted to type double
int side = 3.33;      // double value 3.33 converted to type int 3
```

上述赋值语句都是可行的，因为在 C++看来，各种数值类型都表示相同的东西——一个数字，同时 C++包含用于进行转换的内置规则。然而，第 3 章介绍过，这些转换将降低精度。例如，将 3.33 赋给 int 变量时，转换后的值为 3，丢失了 0.33。

C++语言不自动转换不兼容的类型。例如，下面的语句是非法的，因为左边是指针类型，而右边是数字：

```
int * p = 10;  // type clash
```

虽然计算机内部可能使用整数来表示地址，但从概念上说，整数和指针完全不同。例如，不能计算指针的平方。然而，在无法自动转换时，可以使用强制类型转换：

```
int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```

上述语句将 10 强制转换为 int 指针类型（即 int *类型），将指针设置为地址 10。这种赋值是否有意义是另一回事。

可以将类定义成与基本类型或另一个类相关，使得从一种类型转换为另一种类型是有意义的。在这种情况下，程序员可以指示 C++如何自动进行转换，或通过强制类型转换来完成。为了说明这是如何进行的，我们将第 3 章中的磅转换为英石的程序改写成类的形式。首先，设计一种合适的类型。我们基本上是以两种方式（磅和英石）来表示重量的。对于在一个实体中包含一个概念的两种表示来说，类提供了一种非常好的方式。因此可以将重量的两种表示放在同一个类中，然后提供以这两种方式表达重量的类方法。程序清单 11.16 提供了这个类的头文件。

程序清单 11.16　stonewt.h

```
// stonewt.h -- definition for the Stonewt class
#ifndef STONEWT_H_
#define STONEWT_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};      // pounds per stone
    int stone;                    // whole stones
    double pds_left;              // fractional pounds
    double pounds;                // entire weight in pounds
public:
    Stonewt(double lbs);          // constructor for double pounds
    Stonewt(int stn, double lbs); // constructor for stone, lbs
    Stonewt();                    // default constructor
    ~Stonewt();
```

```
    void show_lbs() const;          // show weight in pounds format
    void show_stn() const;          // show weight in stone format
};
#endif
```

正如第 10 章指出的，对于定义类特定的常量来说，如果它们是整数，enum 提供了一种方便的途径。也可以采用下面这种方法：

```
static const int Lbs_per_stn = 14;
```

Stonewt 类有 3 个构造函数，让您能够将 Stonewt 对象初始化为一个浮点数（单位为磅）或两个浮点数（分别代表英石和磅）。也可以创建 Stonewt 对象，而不进行初始化：

```
Stonewt blossem(132.5);    // weight = 132.5 pounds
Stonewt buttercup(10, 2); // weight = 10 stone, 2 pounds
Stonewt bubbles;           // weight = default value
```

这个类并非真的需要声明构造函数，因为自动生成的默认构造函数就很好。另一方面，提供显式的声明可为以后做好准备，以防必须定义构造函数

另外，Stonewt 类还提供了两个显示函数。一个以磅为单位来显示重量，另一个以英石和磅为单位来显示重量。程序清单 11.17 列出了类方法的实现。每个构造函数都给这三个私有成员全部赋了值。因此创建 Stonewt 对象时，将自动设置这两种重量表示。

程序清单 11.17　stonewt.cpp

```
// stonewt.cpp -- Stonewt methods
#include <iostream>
using std::cout;
#include "stonewt.h"

// construct Stonewt object from double value
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}

// construct Stonewt object from stone, double values
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}

Stonewt::Stonewt()          // default constructor, wt = 0
{
    stone = pounds = pds_left = 0;
}

Stonewt::~Stonewt()         // destructor
{
}
// show weight in stones
void Stonewt::show_stn() const
{
```

```
    cout << stone << " stone, " << pds_left << " pounds\n";
}

// show weight in pounds
void Stonewt::show_lbs() const
{
    cout << pounds << " pounds\n";
}
```

因为 Stonewt 对象表示一个重量，所以可以提供一些将整数或浮点值转换为 Stonewt 对象的方法。我们已经这样做了！在 C++中，接受一个参数的构造函数为将类型与该参数相同的值转换为类提供了蓝图。因此，下面的构造函数用于将 double 类型的值转换为 Stonewt 类型：

```
Stonewt(double lbs);  // template for double-to-Stonewt conversion
```

也就是说，可以编写这样的代码：

```
Stonewt myCat;        // create a Stonewt object
myCat = 19.6;         // use Stonewt(double) to convert 19.6 to Stonewt
```

程序将使用构造函数 Stonewt(double)来创建一个临时的 Stonewt 对象，并将 19.6 作为初始化值。随后，采用逐成员赋值方式将该临时对象的内容复制到 myCat 中。这一过程称为隐式转换，因为它是自动进行的，而不需要显式强制类型转换。

只有接受一个参数的构造函数才能作为转换函数。下面的构造函数有两个参数，因此不能用来转换类型：

```
Stonewt(int stn, double lbs);  // not a conversion function
```

然而，如果给第二个参数提供默认值，它便可用于转换 int：

```
Stonewt(int stn, double lbs = 0);  // int-to-Stonewt conversion
```

将构造函数用作自动类型转换函数似乎是一项不错的特性。然而，当程序员拥有更丰富的 C++经验时，将发现这种自动特性并非总是合乎需要的，因为这会导致意外的类型转换。因此，C++新增了关键字 explicit，用于关闭这种自动特性。也就是说，可以这样声明构造函数：

```
explicit Stonewt(double lbs);   // no implicit conversions allowed
```

这将关闭上述示例中介绍的隐式转换，但仍然允许显式转换，即显式强制类型转换：

```
Stonewt myCat;          // create a Stonewt object
myCat = 19.6;           // not valid if Stonewt(double) is declared as explicit
mycat = Stonewt(19.6);  // ok, an explicit conversion
mycat = (Stonewt) 19.6; // ok, old form for explicit typecast
```

注意：只接受一个参数的构造函数定义了从参数类型到类类型的转换。如果使用关键字 explicit 限定了这种构造函数，则它只能用于显示转换，否则也可以用于隐式转换。

编译器在什么时候将使用 Stonewt(double)函数呢？如果在声明中使用了关键字 explicit，则 Stonewt(double)将只用于显式强制类型转换，否则还可以用于下面的隐式转换。

- 将 Stonewt 对象初始化为 double 值时。
- 将 double 值赋给 Stonewt 对象时。
- 将 double 值传递给接受 Stonewt 参数的函数时。
- 返回值被声明为 Stonewt 的函数试图返回 double 值时。
- 在上述任意一种情况下，使用可转换为 double 类型的内置类型时。

下面详细介绍最后一点。函数原型化提供的参数匹配过程，允许使用 Stonewt（double）构造函数来转换其他数值类型。也就是说，下面两条语句都首先将 int 转换为 double，然后使用 Stonewt（double）构造函数。

```
Stonewt Jumbo(7000);    // uses Stonewt(double), converting int to double
Jumbo = 7300;           // uses Stonewt(double), converting int to double
```

然而，当且仅当转换不存在二义性时，才会进行这种二步转换。也就是说，如果这个类还定义了构造函数 Stonewt（long），则编译器将拒绝这些语句，可能指出：int 可被转换为 long 或 double，因此调用存在二义性。

程序清单 11.18 使用类的构造函数初始化了一些 Stonewt 对象，并处理类型转换。请务必将程序清单 11.18 和程序清单 11.17 一起编译。

程序清单 11.18　stone.cpp

```
// stone.cpp -- user-defined conversions
// compile with stonewt.cpp
#include <iostream>
using std::cout;
#include "stonewt.h"
void display(const Stonewt & st, int n);
int main()
{
    Stonewt incognito = 275; // uses constructor to initialize
    Stonewt wolfe(285.7);    // same as Stonewt wolfe = 285.7;
    Stonewt taft(21, 8);

    cout << "The celebrity weighed ";
    incognito.show_stn();
    cout << "The detective weighed ";
    wolfe.show_stn();
    cout << "The President weighed ";
    taft.show_lbs();
    incognito = 276.8;      // uses constructor for conversion
    taft = 325;             // same as taft = Stonewt(325);
    cout << "After dinner, the celebrity weighed ";
    incognito.show_stn();
    cout << "After dinner, the President weighed ";
    taft.show_lbs();
    display(taft, 2);
    cout << "The wrestler weighed even more.\n";
    display(422, 2);
    cout << "No stone left unearned\n";
    return 0;
}

void display(const Stonewt & st, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Wow! ";
        st.show_stn();
    }
}
```

下面是程序清单 11.18 所示程序的输出：

```
The celebrity weighed 19 stone, 9 pounds
The detective weighed 20 stone, 5.7 pounds
The President weighed 302 pounds
After dinner, the celebrity weighed 19 stone, 10.8 pounds
After dinner, the President weighed 325 pounds
Wow! 23 stone, 3 pounds
```

```
Wow! 23 stone, 3 pounds
The wrestler weighed even more.
Wow! 30 stone, 2 pounds
Wow! 30 stone, 2 pounds
No stone left unearned
```

程序说明

当构造函数只接受一个参数时，可以使用下面的格式来初始化类对象：

```
// a syntax for initializing a class object when
// using a constructor with one argument
Stonewt incognito = 275;
```

这等价于前面介绍过过的另外两种格式：

```
// standard syntax forms for initializing class objects
Stonewt incognito(275);
Stonewt incognito = Stonewt(275);
```

然而，后两种格式可用于接受多个参数的构造函数。

接下来，请注意程序清单 11.18 的下面两条赋值语句：

```
incognito = 276.8;
taft = 325;
```

第一条赋值语句使用接受 double 参数的构造函数，将 276.8 转换为一个 Stonewt 值，这将把 incognito 的 pound 成员设置为 276.8。因为该语句使用了构造函数，所以还将设置 stone 和 pds_left 成员。同样，第二条赋值语句将一个 int 值转换为 double 类型，然后使用 Stonewt(double)来设置全部 3 个成员。

最后，请注意下面的函数调用：

```
display(422, 2);    // convert 422 to double, then to Stonewt
```

display( )的原型表明，第一个参数应是 Stonewt 对象（Stonewt 和 Stonewt &形参都与 Stonewt 实参匹配）。遇到 int 参数时，编译器查找构造函数 Stonewt(int)，以便将该 int 转换为 Stonewt 类型。由于没有找到这样的构造函数，因此编译器寻找接受其他内置类型（int 可以转换为这种类型）的构造函数。Stone(double)构造函数满足这种要求，因此编译器将 int 转换为 double，然后使用 Stonewt(double)将其转换为一个 Stonewt 对象。

11.6.1　转换函数

程序清单 11.18 将数字转换为 Stonewt 对象。可以做相反的转换吗？也就是说，是否可以将 Stonewt 对象转换为 double 值，就像如下所示的那样？

```
Stonewt wolfe(285.7);
double host = wolfe;  // ?? possible ??
```

可以这样做，但不是使用构造函数。构造函数只用于从某种类型到类类型的转换。要进行相反的转换，必须使用特殊的 C++运算符函数——转换函数。

转换函数是用户定义的强制类型转换，可以像使用强制类型转换那样使用它们。例如，如果定义了从 Stonewt 到 double 的转换函数，就可以使用下面的转换：

```
Stonewt wolfe(285.7);
double host = double (wolfe);      // syntax #1
double thinker = (double) wolfe;   // syntax #2
```

也可以让编译器来决定如何做：

```
Stonewt wells(20, 3);
double star = wells;   // implicit use of conversion function
```

编译器发现，右侧是 Stonewt 类型，而左侧是 double 类型，因此它将查看程序员是否定义了与此匹配的转换函数。（如果没有找到这样的定义，编译器将生成错误消息，指出无法将 Stonewt 赋给 double。）

那么，如何创建转换函数呢？要转换为 typeName 类型，需要使用这种形式的转换函数：

```
operator typeName();
```

请注意以下几点：

- 转换函数必须是类方法；
- 转换函数不能指定返回类型；
- 转换函数不能有参数。

例如，转换为 double 类型的函数的原型如下：

```
operator double();
```

typeName（这里为 double）指出了要转换成的类型，因此不需要指定返回类型。转换函数是类方法意味着：它需要通过类对象来调用，从而告知函数要转换的值。因此，函数不需要参数。

要添加将 stone_wt 对象转换为 int 类型和 double 类型的函数，需要将下面的原型添加到类声明中：

```
operator int();
operator double();
```

程序清单 11.19 列出了修改后的类声明。

程序清单 11.19　stonewt1.h

```
// stonewt1.h -- revised definition for the Stonewt class
#ifndef STONEWT1_H_
#define STONEWT1_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};       // pounds per stone
    int stone;                     // whole stones
    double pds_left;               // fractional pounds
    double pounds;                 // entire weight in pounds
public:
    Stonewt(double lbs);           // construct from double pounds
    Stonewt(int stn, double lbs); // construct from stone, lbs
    Stonewt();                     // default constructor
    ~Stonewt();
    void show_lbs() const;         // show weight in pounds format
    void show_stn() const;         // show weight in stone format
// conversion functions
    operator int() const;
    operator double() const;
};
#endif
```

程序清单 11.20 是在程序清单 11.18 的基础上修改而成的，包括了这两个转换函数的定义。注意，虽然没有声明返回类型，这两个函数也将返回所需的值。另外，int 转换将待转换的值四舍五入为最接近的整数，而不是去掉小数部分。例如，如果 pounds 为 114.4，则 pounds +0.5 等于 114.9，int（114.9）等于 114。但是如果 pounds 为 114.6，则 pounds + 0.5 是 115.1，而 int（115.1）为 115。

程序清单 11.20　stonewt1.cpp

```
// stonewt1.cpp -- Stonewt class methods + conversion functions
#include <iostream>
using std::cout;
#include "stonewt1.h"

// construct Stonewt object from double value
```

```
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}

// construct Stonewt object from stone, double values
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}
Stonewt::Stonewt()          // default constructor, wt = 0
{
    stone = pounds = pds_left = 0;
}

Stonewt::~Stonewt()         // destructor
{
}

// show weight in stones
void Stonewt::show_stn() const
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}

// show weight in pounds
void Stonewt::show_lbs() const
{
    cout << pounds << " pounds\n";
}

// conversion functions
Stonewt::operator int() const
{

    return int (pounds + 0.5);

}

Stonewt::operator double()const
{
    return pounds;
}
```

程序清单 11.21 对新的转换函数进行测试。该程序中的赋值语句使用隐式转换，而最后的 cout 语句使用显式强制类型转换。请务必将程序清单 11.20 与程序清单 11.21 一起编译。

程序清单 11.21　stone1.cpp

```
// stone1.cpp -- user-defined conversion functions
// compile with stonewt1.cpp
#include <iostream>
```

```
#include "stonewt1.h"

int main()
{
    using std::cout;
    Stonewt poppins(9,2.8);      // 9 stone, 2.8 pounds
    double p_wt = poppins;       // implicit conversion
    cout << "Convert to double => ";
    cout << "Poppins: " << p_wt << " pounds.\n";
    cout << "Convert to int => ";
    cout << "Poppins: " << int (poppins) << " pounds.\n";
    return 0;
}
```

下面是程序清单 11.19～程序清单 11.21 组成的程序的输出；它显示了将 Stonewt 对象转换为 double 类型和 int 类型的结果：

```
Convert to double => Poppins: 128.8 pounds.
Convert to int => Poppins: 129 pounds.
```

自动应用类型转换

程序清单 11.21 将 int(poppins)和 cout 一起使用。假设省略了显式强制类型转换：

```
cout << "Poppins: " << poppins << " pounds.\n";
```

程序会像在下面的语句中那样使用隐式转换吗？

```
double p_wt = poppins;
```

答案是否定的。在 p_wt 示例中，上下文表明，poppins 应被转换为 double 类型。但在 cout 示例中，并没有指出应转换为 int 类型还是 double 类型。在缺少信息时，编译器将指出，程序中使用了二义性转换。该语句没有指出要使用什么类型。

有趣的是，如果类只定义了 double 转换函数，则编译器将接受该语句。这是因为只有一种转换可能，因此不存在二义性。

赋值的情况与此类似。对于当前的类声明来说，编译器将认为下面的语句有二义性而拒绝它。

```
long gone = poppins;   // ambiguous
```

在 C++中，int 和 double 值都可以被赋给 long 变量，所以编译器使用任意一个转换函数都是合法的。编译器不想承担选择转换函数的责任。然而，如果删除了这两个转换函数之一，编译器将接受这条语句。例如，假设省略了 double 定义，则编译器将使用 int 转换，将 poppins 转换为一个 int 类型的值。然后在将它赋给 gone 时，将 int 类型值转换为 long 类型。

当类定义了两种或更多的转换时，仍可以用显式强制类型转换来指出要使用哪个转换函数。可以使用下面任何一种强制类型转换表示法：

```
long gone = (double) poppins;  // use double conversion
long gone = int (poppins);     // use int conversion
```

第一条语句将 poppins 转换为一个 double 值，然后赋值操作将该 double 值转换为 long 类型。同样，第二条语句将 poppins 首先转换为 int 类型，随后转换为 long。

和转换构造函数一样，转换函数也有其优缺点。提供执行自动、隐式转换的函数所存在的问题是：在用户不希望进行转换时，转换函数也可能进行转换。例如，假设您在睡眠不足时编写了下面的代码：

```
int ar[20];
...
Stonewt temp(14, 4);
...
int Temp = 1;
...
cout << ar[temp] << "!\n";  // used temp instead of Temp
```

通常，您以为编译器能够捕获诸如使用了对象而不是整数作为数组索引等错误，但 Stonewt 类定义了一个 operator int( )，因此 Stonewt 对象 temp 将被转换为 int 200，并用作数组索引。原则上说，最好使用显式转换，而避免隐式转换。在 C++98 中，关键字 explicit 不能用于转换函数，但 C++11 消除了这种限制。因此，在 C++11 中，可将转换运算符声明为显式的：

```
class Stonewt
{
...
// conversion functions
    explicit operator int() const;
    explicit operator double() const;
};
```

有了这些声明后，需要强制转换时将调用这些运算符。

另一种方法是，用一个功能相同的非转换函数替换该转换函数即可，但仅在被显式地调用时，该函数才会执行。也就是说，可以将：

```
Stonewt::operator int() { return int (pounds + 0.5); }
```

替换为：

```
int Stonewt::Stone_to_Int() { return int (pounds + 0.5); }
```

这样，下面的语句将是非法的：

```
int plb = poppins;
```

但如果确实需要这种转换，可以这样做：

```
int plb = poppins.Stone_to_Int();
```

警告：应谨慎地使用隐式转换函数。通常，最好选择仅在被显式地调用时才会执行的函数。

总之，C++为类提供了下面的类型转换。

- 只有一个参数的类构造函数用于将类型与该参数相同的值转换为类类型。例如，将 int 值赋给 Stonewt 对象时，接受 int 参数的 Stonewt 类构造函数将自动被调用。然而，在构造函数声明中使用 explicit 可防止隐式转换，而只允许显式转换。
- 被称为转换函数的特殊类成员运算符函数，用于将类对象转换为其他类型。转换函数是类成员，没有返回类型、没有参数、名为 operator typeName( )，其中，typeName 是对象将被转换成的类型。将类对象赋给 typeName 变量或将其强制转换为 typeName 类型时，该转换函数将自动被调用。

11.6.2　转换函数和友元函数

下面为 Stonewt 类重载加法运算符。在讨论 Time 类时指出过，可以使用成员函数或友元函数来重载加法。（出于简化的目的，我们假设没有定义 operator double( )转换函数。）可以使用下面的成员函数实现加法：

```
Stonewt Stonewt::operator+(const Stonewt & st) const
{
    double pds = pounds + st.pounds;
    Stonewt sum(pds);
    return sum;
}
```

也可以将加法作为友元函数来实现，如下所示：

```
Stonewt operator+(const Stonewt & st1, const Stonewt & st2)
{
    double pds = st1.pounds + st2.pounds;
    Stonewt sum(pds);
    return sum;
}
```

别忘了，可以提供方法定义或友元函数定义，但不能都提供。上面任何一种格式都允许这样做：

```
Stonewt jennySt(9, 12);
Stonewt bennySt(12, 8);
Stonewt total;
total = jennySt + bennySt;
```

另外，如果提供了 Stonewt（double）构造函数，则也可以这样做：

```
Stonewt jennySt(9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

但只有友元函数才允许这样做：

```
Stonewt jennySt(9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

为了解其中的原因，将每一种加法都转换为相应的函数调用。首先：

```
total = jennySt + bennySt;
```

被转换为：

```
total = jennySt.operator+(bennySt);   // member function
```

或：

```
total = operator+(jennySt, bennySt);  // friend function
```

上述两种转换中，实参的类型都和形参匹配。另外，成员函数是通过 Stonewt 对象调用的。

其次：

```
total = jennySt + kennyD;
```

被转换为：

```
total = jennySt.operator+(kennyD);   // member function
```

或：

```
total = operator+(jennySt, kennyD);  // friend function
```

同样，成员函数也是通过 Stonewt 对象调用的。这一次，每个调用中都有一个参数（kennyD）是 double 类型的，因此将调用 Stonewt（double）构造函数，将该参数转换为 Stonewt 对象。

另外，在这种情况下，如果定义了 operator double( )成员函数，将造成混乱，因为该函数将提供另一种解释方式。编译器不是将 kennyD 转换为 double 并执行 Stonewt 加法，而是将 jennySt 转换为 double 并执行 double 加法。过多的转换函数将导致二义性。

最后：

```
total = pennyD + jennySt;
```

被转换为：

```
total = operator+(pennyD, jennySt);  // friend function
```

其中，两个参数都是 double 类型，因此将调用构造函数 Stonewt(double)，将它们转换为 Stonewt 对象。

然而，不能调用成员函数将 jennySt 和 peenyD 相加。将加法语法转换为函数调用将类似于下面这样：

```
total = pennyD.operator+(jennySt);   // not meaningful
```

这没有意义，因为只有类对象才可以调用成员函数。C++不会试图将 pennyD 转换为 Stonewt 对象。将对成员函数参数进行转换，而不是调用成员函数的对象。

这里的经验是，将加法定义为友元可以让程序更容易适应自动类型转换。原因在于，两个操作数都成为函数参数，因此与函数原型匹配。

实现加法时的选择

要将 double 量和 Stonewt 量相加，有两种选择。第一种方法是（刚介绍过）将下面的函数定义为友元

函数，让 Stonewt(double)构造函数将 double 类型的参数转换为 Stonewt 类型的参数：

```
operator+(const Stonewt &, const Stonewt &)
```

第二种方法是，将加法运算符重载为一个显式使用 double 类型参数的函数：

```
Stonewt operator+(double x);  // member function
friend Stonewt operator+(double x, Stonewt & s);
```

这样，下面的语句将与成员函数 operator + (double x)完全匹配：

```
total = jennySt + kennyD; // Stonewt + double
```

而下面的语句将与友元函数 operator + (double x, Stonewt &s)完全匹配：

```
total = pennyD + jennySt; // double + Stonewt
```

前面对 Vector 乘法做了类似的处理。

每一种方法都有其优点。第一种方法（依赖于隐式转换）使程序更简短，因为定义的函数较少。这也意味程序员需要完成的工作较少，出错的机会较小。这种方法的缺点是，每次需要转换时，都将调用转换构造函数，这增加时间和内存开销。第二种方法（增加一个显式地匹配类型的函数）则正好相反。它使程序较长，程序员需要完成的工作更多，但运行速度较快。

如果程序经常需要将 double 值与 Stonewt 对象相加，则重载加法更合适；如果程序只是偶尔使用这种加法，则依赖于自动转换更简单，但为了更保险，可以使用显式转换。

11.7　总结

本章介绍了定义和使用类的许多重要方面，其中的一些内容可能较难理解，但随着实践经验的不断增加，您将逐渐掌握它们。

一般来说，访问私有类成员的唯一方法是使用类方法。C++使用友元函数来避开这种限制。要让函数成为友元，需要在类声明中声明该函数，并在声明前加上关键字 friend。

C++扩展了对运算符的重载，允许自定义特殊的运算符函数，这种函数描述了特定的运算符与类之间的关系。运算符函数可以是类成员函数，也可以是友元函数（有一些运算符函数只能是类成员函数）。要调用运算符函数，可以直接调用该函数，也可以以通常的句法使用被重载的运算符。对于运算符 op，其运算符函数的格式如下：

```
operatorop(argument-list)
```

argument-list 表示该运算符的操作数。如果运算符函数是类成员函数，则第一个操作数是调用对象，它不在 argument-list 中。例如，本章通过为 Vector 类定义 operator +( )成员函数重载了加法。如果 up、right 和 result 都是 Vector 对象，则可以使用下面的任何一条语句来调用矢量加法：

```
result = up.operator+(right);
result = up + right;
```

在第二条语句中，由于操作数 up 和 right 的类型都是 Vector，因此 C++将使用 Vector 的加法定义。

当运算符函数是成员函数时，则第一个操作数将是调用该函数的对象。例如，在前面的语句中，up 对象是调用函数的对象。定义运算符函数时，如果要使其第一个操作数不是类对象，则必须使用友元函数。这样就可以将操作数按所需的顺序传递给函数了。

最常见的运算符重载任务之一是定义<<运算符，使之可与 cout 一起使用，来显示对象的内容。要让 ostream 对象成为第一个操作数，需要将运算符函数定义为友元；要使重新定义的运算符能与其自身拼接，需要将返回类型声明为 ostream &。下面的通用格式能够满足这种要求：

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

然而，如果类包含这样的方法，它返回需要显示的数据成员的值，则可以使用这些方法，无需在operator<<( )中直接访问这些成员。在这种情况下，函数不必（也不应当）是友元。

C++允许指定在类和基本类型之间进行转换的方式。首先，任何接受唯一一个参数的构造函数都可被用作转换函数，将类型与该参数相同的值转换为类。如果将类型与该参数相同的值赋给对象，则 C++将自动调用该构造函数。例如，假设有一个 String 类，它包含一个将 char *值作为其唯一参数的构造函数，那么如果 bean 是 String 对象，则可以使用下面的语句：

```
bean = "pinto";   // converts type char * to type String
```

然而，如果在该构造函数的声明前加上了关键字 explicit，则该构造函数将只能用于显式转换：

```
bean = String("pinto");   // converts type char * to type String explicitly
```

要将类对象转换为其他类型，必须定义转换函数，指出如何进行这种转换。转换函数必须是成员函数。将类对象转换为 typeName 类型的转换函数的原型如下：

```
operator typeName();
```

注意，转换函数没有返回类型、没有参数，但必须返回转换后的值（虽然没有声明返回类型）。例如，下面是将 Vector 转换为 double 类型的函数：

```
Vector::operator double()
{
    ...
    return a_double_value;
}
```

经验表明，最好不要依赖于这种隐式转换函数。

您可能已经注意到了，与简单的 C-风格结构相比，使用类时，必须更谨慎、更小心，但作为补偿，它们为我们完成的工作也更多。

11.8 复习题

1．使用成员函数为 Stonewt 类重载乘法运算符，该运算符将数据成员与 double 类型的值相乘。注意，用英石和磅表示时，需要进位。也就是说，将 10 英石 8 磅乘以 2 等于 21 英石 2 磅。

2．友元函数与成员函数之间的区别是什么？

3．非成员函数必须是友元才能访问类成员吗？

4．使用友元函数为 Stonewt 类重载乘法运算符，该运算符将 double 值与 Stone 值相乘。

5．哪些运算符不能重载？

6．在重载运算符=、( )、[ ]和->时，有什么限制？

7．为 Vector 类定义一个转换函数，将 Vector 类转换为一个 double 类型的值，后者表示矢量的长度。

11.9 编程练习

1．修改程序清单 11.5，使之将一系列连续的随机漫步者位置写入到文件中。对于每个位置，用步号进行标示。另外，让该程序将初始条件（目标距离和步长）以及结果小结写入到该文件中。该文件的内容与下面类似：

```
Target Distance: 100, Step Size: 20
0: (x,y) = (0, 0)
1: (x,y) = (-11.4715, 16.383)
2: (x,y) = (-8.68807, -3.42232)
...
```

```
26: (x,y) = (42.2919, -78.2594)
27: (x,y) = (58.6749, -89.7309)
After 27 steps, the subject has the following location:
(x,y) = (58.6749, -89.7309)
 or
(m,a) = (107.212, -56.8194)
Average outward distance per step = 3.97081
```

2．对 Vector 类的头文件（程序清单 11.13）和实现文件（程序清单 11.14）进行修改，使其不再存储矢量的长度和角度，而是在 magval( )和 angval( )被调用时计算它们。

应保留公有接口不变（公有方法及其参数不变），但对私有部分（包括一些私有方法）和方法实现进行修改。然后，使用程序清单 11.15 对修改后的版本进行测试，结果应该与以前相同，因为 Vector 类的公有接口与原来相同。

3．修改程序清单 11.15，使之报告 N 次测试中的最高、最低和平均步数（其中 N 是用户输入的整数），而不是报告每次测试的结果。

4．重新编写最后的 Time 类示例（程序清单 11.10、程序清单 11.11 和程序清单 11.12），使用友元函数来实现所有的重载运算符。

5．重新编写 Stonewt 类（程序清单 11.16 和程序清单 11.17），使它有一个状态成员，由该成员控制对象应转换为英石格式、整数磅格式还是浮点磅格式。重载<<运算符，使用它来替换 show_stn( )和 show_lbs( )方法。重载加法、减法和乘法运算符，以便可以对 Stonewt 值进行加、减、乘运算。编写一个使用所有类方法和友元的小程序，来测试这个类。

6. 重新编写 Stonewt 类(程序清单 11.16 和程序清单 11.17)，重载全部 6 个关系运算符。运算符对 pounds 成员进行比较，并返回一个 bool 值。编写一个程序，它声明一个包含 6 个 Stonewt 对象的数组，并在数组声明中初始化前 3 个对象。然后使用循环来读取用于设置剩余 3 个数组元素的值。接着报告最小的元素、最大的元素以及大于或等于 11 英石的元素的数量（最简单的方法是创建一个 Stonewt 对象，并将其初始化为 11 英石，然后将其同其他对象进行比较）。

7．复数有两个部分组成：实数部分和虚数部分。复数的一种书写方式是：(3.0，4.0)，其中，3.0 是实数部分，4.0 是虚数部分。假设 a = (A, Bi)，c = (C, Di)，则下面是一些复数运算。

- 加法：a + c = (A+C, (B+D)i)。
- 减法：a – c = (A–C, (B–D)i)。
- 乘法：a * c = (A*C–B*D, (A*D + B*C)i)。
- 乘法::x*c = (x * C, x *Di)，其中 x 为实数。
- 共轭：~a = (A, –Bi)。

请定义一个复数类，以便下面的程序可以使用它来获得正确的结果。

```
#include <iostream>
using namespace std;
#include "complex0.h"  // to avoid confusion with complex.h
int main()
{
    complex a(3.0, 4.0);   // initialize to (3,4i)
    complex c;
    cout << "Enter a complex number (q to quit):\n";
    while (cin >> c)
    {
        cout << "c is " << c << '\n';
        cout << "complex conjugate is " << ~c << '\n';
        cout << "a is " << a << '\n';
        cout << "a + c is " << a + c << '\n';
        cout << "a - c is " << a - c << '\n';
        cout << "a * c is " << a * c << '\n';
```

```
        cout << "2 * c is " << 2 * c << '\n';
        cout << "Enter a complex number (q to quit):\n";
    }
    cout << "Done!\n";
    return 0;
}
```

注意，必须重载运算符<<和>>。标准 C++使用头文件 complex 提供了比这个示例更广泛的复数支持，因此应将自定义的头文件命名为 complex0.h，以免发生冲突。应尽可能使用 const。

下面是该程序的运行情况。

```
Enter a complex number (q to quit):
real: 10
imaginary: 12
c is (10,12i)
complex conjugate is (10,-12i)
a is (3,4i)
a + c is (13,16i)
a - c is (-7,-8i)
a * c is (-18,76i)
2 * c is (20,24i)
Enter a complex number (q to quit):
real: q
Done!
```

请注意，经过重载后，cin >>c 将提示用户输入实数和虚数部分。

第 12 章　类和动态内存分配

本章内容包括：

- 对类成员使用动态内存分配。
- 隐式和显式复制构造函数。
- 隐式和显式重载赋值运算符。
- 在构造函数中使用 new 所必须完成的工作。
- 使用静态类成员。
- 将定位 new 运算符用于对象。
- 使用指向对象的指针。
- 实现队列抽象数据类型（ADT）。

本章将介绍如何对类使用 new 和 delete 以及如何处理由于使用动态内存而引起的一些微妙的问题。这里涉及的主题好像不多，但它们将影响构造函数和析构函数的设计以及运算符的重载。

来看一个具体的例子——C++如何增加内存负载。假设要创建一个类，其一个成员表示某人的姓。最简单的方法是使用字符数组成员来保存姓，但这种方法有一些缺陷。开始也许会使用一个 14 个字符的数组，然后发现数组太小，更保险的方法是，使用一个 40 个字符的数组。然而，如果创建包含 2000 个这种对象的数组，就会由于字符数组只有部分被使用而浪费大量的内存（在这种情况下，增加了计算机的内存负载）。但可以采取另一种方法。

通常，最好是在程序运行时（而不是编译时）确定诸如使用多少内存等问题。对于在对象中保存姓名来说，通常的 C++方法是，在类构造函数中使用 new 运算符在程序运行时分配所需的内存。为此，通常的方法是使用 string 类，它将为您处理内存管理细节。但这样您就没有机会更深入地学习内存管理了，因此这里将直接对问题发起攻击。除非同时执行一系列额外步骤，如扩展类析构函数、使所有的构造函数与 new 析构函数协调一致、编写额外的类方法来帮助正确完成初始化和赋值（当然，本章将介绍这些步骤），否则，在类构造函数中使用 new 将导致新问题。

12.1　动态内存和类

您希望下个月的早餐、午餐和晚餐吃些什么？在第三天的晚餐喝多少盎司的牛奶？在第 15 天的早餐中需要在谷类食品添加多少葡萄干？如果您与大多数人一样，就会等到进餐时再做决定。C++在分配内存时采取的部分策略与此相同，让程序在运行时决定内存分配，而不是在编译时决定。这样，可根据程序的需要，而不是根据一系列严格的存储类型规则来使用内存。C++使用 new 和 delete 运算符来动态控制内存。遗憾的是，在类中使用这些运算符将导致许多新的编程问题。在这种情况下，析构函数将是必不可少的，而不再是可有可无的。有时候，还必须重载赋值运算符，以保证程序正常运行。下面来看一看这些问题。

12.1.1　复习示例和静态类成员

我们已经有一段时间没有使用 new 和 delete 了，所以这里使用一个小程序来复习它们。这个程序使用

了一个新的存储类型：静态类成员。首先设计一个 StringBad 类，然后设计一个功能稍强的 String 类（本书前面介绍过 C++标准 string 类，第 16 章将更深入地讨论它；而本章的 StringBad 和 String 类将介绍这个类的底层结构，提供这种友好的接口涉及大量的编程技术）。

StringBad 和 String 类对象将包含一个字符串指针和一个表示字符串长度的值。这里使用 StringBad 和 String 类，主要是为了深入了解 new、delete 和静态类成员的工作原理。因此，构造函数和析构函数调用时将显示一些消息，以便您能够按照提示来完成操作。另外，将省略一些有用的成员和友元函数，如重载的++和>>运算符以及转换函数，以简化类接口（但本章的复习题将要求您添加这些函数）。程序清单 12.1 列出了这个类的声明。

为什么将它命名为 StringBad 呢？这是为了表示提醒，StringBad 是一个还没有开发好的示例。这是使用动态内存分配来开发类的第一步，它正确地完成了一些显而易见的工作，例如，它在构造函数和析构函数中正确地使用了 new 和 delete。它其实不会执行有害的操作，但省略了一些有益的功能，这些功能是必需的，但却不是显而易见的。通过说明这个类存在的问题，有助于在稍后将它转换为一个功能更强的 String 类时，理解和牢记所做的一些并不明显的修改。

程序清单 12.1　strngbad.h

```
// strngbad.h -- flawed string class definition
#include <iostream>
#ifndef STRNGBAD_H_
#define STRNGBAD_H_
class StringBad
{
private:
    char * str;                 // pointer to string
    int len;                    // length of string
    static int num_strings;     // number of objects
public:
    StringBad(const char * s); // constructor
    StringBad();               // default constructor
    ~StringBad();              // destructor
// friend function
    friend std::ostream & operator<<(std::ostream & os,
                        const StringBad & st);
};
#endif
```

为何将这个类命名为 StringBad 呢？这旨在告诉您，这是一个不太完整的类。它是使用动态内存分配来开发类的第一个阶段，正确地完成了一些显而易见的工作，例如，在构造函数和析构函数中正确地使用了 new 和 delete。这个类并没有什么错误，但忽略了一些不明显却必不可少的东西。通过了解这个类存在的问题，将有助于您理解并记住后面将其转换为功能更强大的 String 类时，所做的不明显的修改。

对这个声明，需要注意的有两点。首先，它使用 char 指针（而不是 char 数组）来表示姓名。这意味着类声明没有为字符串本身分配存储空间，而是在构造函数中使用 new 来为字符串分配空间。这避免了在类声明中预先定义字符串的长度。

其次，将 num_strings 成员声明为静态存储类。静态类成员有一个特点：无论创建了多少对象，程序都只创建一个静态类变量副本。也就是说，类的所有对象共享同一个静态成员，就像家中的电话可供全体家庭成员共享一样。假设创建了 10 个 StringBad 对象，将有 10 个 str 成员和 10 个 len 成员，但只有一个共享的 num_strings 成员（参见图 12.1）。这对于所有类对象都具有相同值的类私有数据是非常方便的。例如，num_strings 成员可以记录所创建的对象数目。

随便说一句，程序清单 21.1 使用 num_strings 成员，只是为了方便说明静态数据成员，并指出潜在的

编程问题，字符串类通常并不需要这样的成员。

来看一看程序清单 12.2 中的类方法实现，它演示了如何使用指针和静态成员。

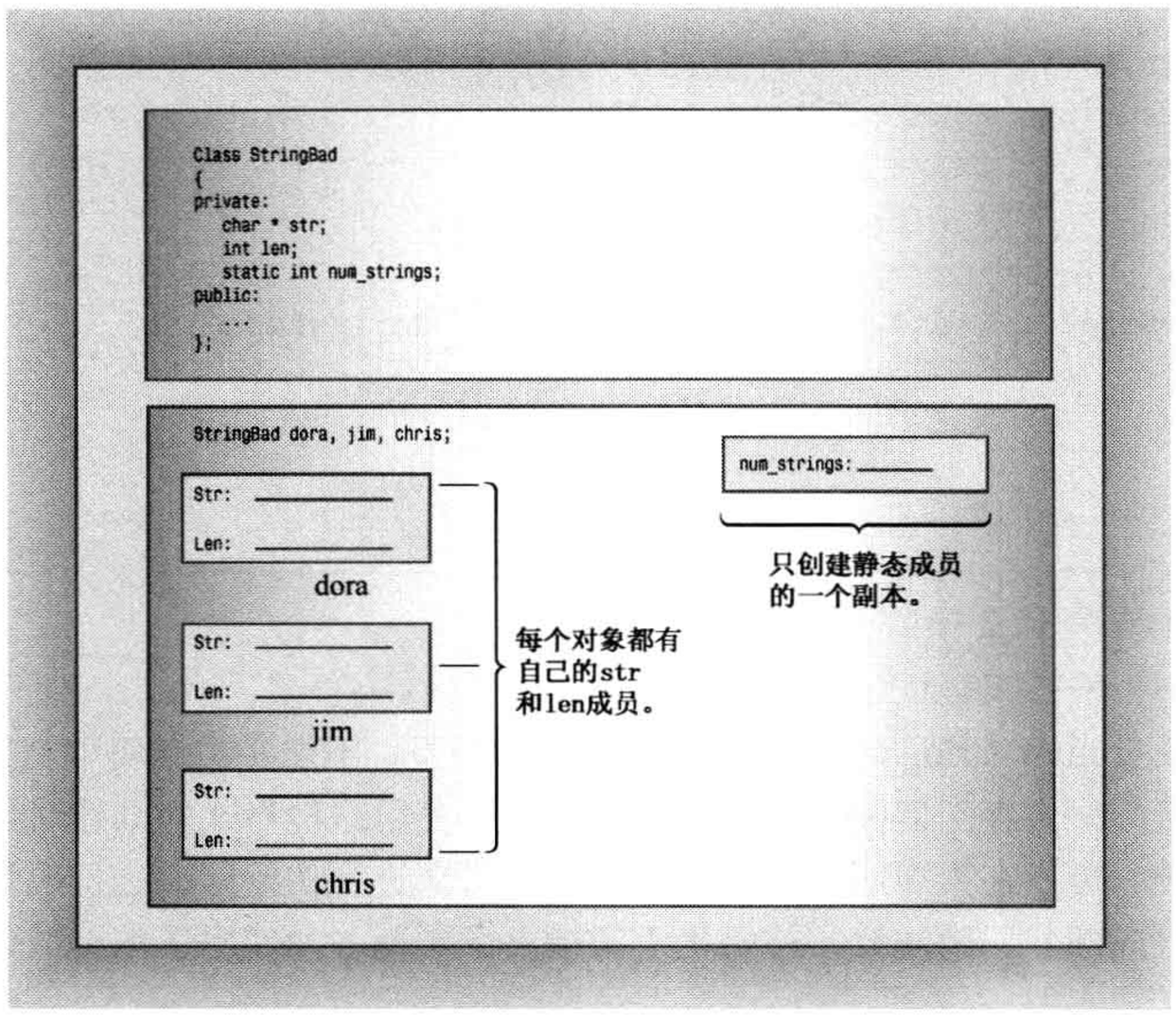


图 12.1　静态数据成员

程序清单 12.2　strngbad.cpp

```
// strngbad.cpp -- StringBad class methods
#include <cstring>                    // string.h for some
#include "strngbad.h"
using std::cout;

// initializing static class member
int StringBad::num_strings = 0;

// class methods
// construct StringBad from C string
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);             // set size
    str = new char[len + 1];          // allot storage
    std::strcpy(str, s);              // initialize pointer
    num_strings++;                    // set object count
    cout << num_strings << ": \"" << str
         << "\" object created\n";    // For Your Information
}

StringBad::StringBad()                // default constructor
{
    len = 4;
    str = new char[4];
    std::strcpy(str, "C++");          // default string
    num_strings++;
    cout << num_strings << ": \"" << str
         << "\" default object created\n";  // FYI
}
```

```
StringBad::~StringBad()                // necessary destructor
{
    cout << "\"" << str << "\" object deleted, ";    // FYI
    --num_strings;                     // required
    cout << num_strings << " left\n"; // FYI
    delete [] str;                     // required
}

std::ostream & operator<<(std::ostream & os, const StringBad & st)
{
    os << st.str;
    return os;
}
```

首先，请注意程序清单 12.2 中的下面一条语句：

```
int StringBad::num_strings = 0;
```

这条语句将静态成员 num_strings 的值初始化为零。请注意，不能在类声明中初始化静态成员变量，这是因为声明描述了如何分配内存，但并不分配内存。您可以使用这种格式来创建对象，从而分配和初始化内存。对于静态类成员，可以在类声明之外使用单独的语句来进行初始化，这是因为静态类成员是单独存储的，而不是对象的组成部分。请注意，初始化语句指出了类型，并使用了作用域运算符，但没有使用关键字 static。

初始化是在方法文件中，而不是在类声明文件中进行的，这是因为类声明位于头文件中，程序可能将头文件包括在其他几个文件中。如果在头文件中进行初始化，将出现多个初始化语句副本，从而引发错误。

对于不能在类声明中初始化静态数据成员的一种例外情况（见第 10 章）是，静态数据成员为整型或枚举型 const。

注意： *静态数据成员在类声明中声明，在包含类方法的文件中初始化。初始化时使用作用域运算符来指出静态成员所属的类。但如果静态成员是整型或枚举型 const，则可以在类声明中初始化。*

接下来，注意到每个构造函数都包含表达式 num_strings++，这确保程序每创建一个新对象，共享变量 num_strings 的值都将增加 1，从而记录 String 对象的总数。另外，析构函数包含表达式--num_strings，因此 String 类也将跟踪对象被删除的情况，从而使 num_string 成员的值是最新的。

现在来看程序清单 12.2 中的第一个构造函数，它使用一个常规 C 字符串来初始化 String 对象：

```
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);          // set size
    str = new char[len + 1];       // allot storage
    std::strcpy(str, s);           // initialize pointer
    num_strings++;                 // set object count
    cout << num_strings << ": \"" << str
         << "\" object created\n"; // For Your Information
}
```

类成员 str 是一个指针，因此构造函数必须提供内存来存储字符串。初始化对象时，可以给构造函数传递一个字符串指针：

```
String boston("Boston");
```

构造函数必须分配足够的内存来存储字符串，然后将字符串复制到内存中。下面介绍其中的每一个步骤。

首先，使用 strlen()函数计算字符串的长度，并对 len 成员进行初始化。接着，使用 new 分配足够的空间来保存字符串，然后将新内存的地址赋给 str 成员。（strlen()返回字符串长度，但不包括末尾的空字符，

因此构造函数将 len 加 1，使分配的内存能够存储包含空字符的字符串。）

接着，构造函数使用 strcpy()将传递的字符串复制到新的内存中，并更新对象计数。最后，构造函数显示当前的对象数目和当前对象中存储的字符串，以助于掌握程序运行情况。稍后故意使 Stringbad 出错时，该特性将派上用场。

要理解这种方法，必须知道字符串并不保存在对象中。字符串单独保存在堆内存中，对象仅保存了指出到哪里去查找字符串的信息。

不能这样做：

```
str = s;  // not the way to go
```

这只保存了地址，而没有创建字符串副本。

默认构造函数与此相似，但它提供了一个默认字符串："C++"。

析构函数中包含了示例中对处理类来说最重要的东西：

```
StringBad::~StringBad()                // necessary destructor
{
    cout << "\"" << str << "\" object deleted, ";    // FYI
    --num_strings;                     // required
    cout << num_strings << " left\n"; // FYI
    delete [] str;                     // required
}
```

该析构函数首先指出自己何时被调用。这部分包含了丰富的信息，但并不是必不可少的。然而，delete 语句却是至关重要的。str 成员指向 new 分配的内存。当 StringBad 对象过期时，str 指针也将过期。但 str 指向的内存仍被分配，除非使用 delete 将其释放。删除对象可以释放对象本身占用的内存，但并不能自动释放属于对象成员的指针指向的内存。因此，必须使用析构函数。在析构函数中使用 delete 语句可确保对象过期时，由构造函数使用 new 分配的内存被释放。

警告：在构造函数中使用 new 来分配内存时，必须在相应的析构函数中使用 delete 来释放内存。如果使用 new[]（包括中括号）来分配内存，则应使用 delete[]（包括中括号）来释放内存。

程序清单 12.3 是从处于开发阶段的 Daily Vegetable 程序中摘录出来的，演示了 StringBad 的构造函数和析构函数何时运行及如何运行。该程序将对象声明放在一个内部代码块中，因为析构函数将在定义对象的代码块执行完毕时调用。如果不这样做，析构函数将在 main()函数执行完毕时调用，导致您无法在执行窗口关闭前看到析构函数显示的消息。请务必将程序清单 12.2 和程序清单 12.3 一起编译。

程序清单 12.3　vegnews.cpp

```
// vegnews.cpp -- using new and delete with classes
// compile with strngbad.cpp
#include <iostream>
using std::cout;
#include "strngbad.h"

void callme1(StringBad &);  // pass by reference
void callme2(StringBad);    // pass by value

int main()
{
    using std::endl;
    {
        cout << "Starting an inner block.\n";
        StringBad headline1("Celery Stalks at Midnight");
        StringBad headline2("Lettuce Prey");
        StringBad sports("Spinach Leaves Bowl for Dollars");
```

```
    cout << "headline1: " << headline1 << endl;
    cout << "headline2: " << headline2 << endl;
    cout << "sports: " << sports << endl;
    callme1(headline1);
    cout << "headline1: " << headline1 << endl;
    callme2(headline2);
    cout << "headline2: " << headline2 << endl;
    cout << "Initialize one object to another:\n";
    StringBad sailor = sports;
    cout << "sailor: " << sailor << endl;
    cout << "Assign one object to another:\n";
    StringBad knot;
    knot = headline1;
    cout << "knot: " << knot << endl;
    cout << "Exiting the block.\n";
  }
  cout << "End of main()\n";

  return 0;
}

void callme1(StringBad & rsb)
{
  cout << "String passed by reference:\n";
  cout << "    \"" << rsb << "\"\n";
}

void callme2(StringBad sb)
{
  cout << "String passed by value:\n";
  cout << "    \"" << sb << "\"\n";
}
```

注意：StringBad 的第一个版本有许多故意留下的缺陷，这些缺陷使得输出是不确定的。例如，有些编译器无法编译它。虽然输出的具体内容有所差别，但基本问题和解决方法（稍后将介绍）是相同的。

下面是使用 Borland C++5.5 命令行编译器进行编译时，该程序的输出：

```
Starting an inner block.
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
String passed by reference:
    "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
String passed by value:
    "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: Dû°
Initialize one object to another:
sailor: Spinach Leaves Bowl for Dollars
Assign one object to another:
3: "C++" default object created
```

```
knot: Celery Stalks at Midnight
Exiting the block.
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-|" object deleted, -2 left
End of main()
```

输出中出现的各种非标准字符随系统而异，这些字符表明，StringBad 类名副其实（是一个糟糕的类）。另一种迹象是对象计数为负。在使用较新的编译器和操作系统的机器上运行时，该程序通常会在显示有关还有-1 个对象的信息之前中断，而有些这样的机器将报告通用保护错误（GPF）。GPF 表明程序试图访问禁止它访问的内存单元，这是另一种糟糕的信号。

程序说明

程序清单 12.3 中的程序开始时还是正常的，但逐渐变得异常，最终导致了灾难性结果。首先来看正常的部分。构造函数指出自己创建了 3 个 StringBad 对象，并为这些对象进行了编号，然后程序使用重载运算符>>列出了这些对象：

```
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
```

然后，程序将 headline1 传递给 callme1( )函数，并在调用后重新显示 headline1。代码如下：

```
callme1(headline1);
cout << "headline1: " << headline1 << endl;
```

下面是运行结果：

```
String passed by reference:
    "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
```

这部分代码看起来也正常。

但随后程序执行了如下代码：

```
callme2(headline2);
cout << "headline2: " << headline2 << endl;
```

这里，callme2()按值（而不是按引用）传递 headline2，结果表明这是一个严重的问题！

```
String passed by value:
    "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: Dû°
```

首先，将 headline2 作为函数参数来传递从而导致析构函数被调用。其次，虽然按值传递可以防止原始参数被修改，但实际上函数已使原始字符串无法识别，导致显示一些非标准字符（显示的文本取决于内存中包含的内容）。

请看输出结果，在为每一个创建的对象自动调用析构函数时，情况更糟糕：

```
Exiting the block.
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-|" object deleted, -2 left
End of main()
```

因为自动存储对象被删除的顺序与创建顺序相反，所以最先删除的 3 个对象是 knots、sailor 和 sport。

删除 knots 和 sailor 时是正常的，但在删除 sport 时，Dollars 变成了 Doll8。对于 sport，程序只使用它来初始化 sailor，但这种操作修改了 sport。最后被删除的两个对象（headline2 和 headline1）已经无法识别。这些字符串在被删除之前，有些操作将它们搞乱了。另外，计数也很奇怪，如何会余下-2 个对象呢？

实际上，计数异常是一条线索。因为每个对象被构造和析构一次，因此调用构造函数的次数应当与析构函数的调用次数相同。对象计数（num_strings）递减的次数比递增次数多 2，这表明使用了不将 num_string 递增的构造函数创建了两个对象。类定义声明并定义了两个构造函数（这两个构造函数都使 num_string 递增），但结果表明程序使用了 3 个构造函数。例如，请看下面的代码：

```
StringBad sailor = sports;
```

这使用的是哪个构造函数呢？不是默认构造函数，也不是参数为 const char *的构造函数。记住，这种形式的初始化等效于下面的语句：

```
StringBad sailor = StringBad(sports); //constructor using sports
```

因为 sports 的类型为 StringBad，因此相应的构造函数原型应该如下：

```
StringBad(const StringBad &);
```

当您使用一个对象来初始化另一个对象时，编译器将自动生成上述构造函数（称为复制构造函数，因为它创建对象的一个副本）。自动生成的构造函数不知道需要更新静态变量 num_string，因此会将计数方案搞乱。实际上，这个例子说明的所有问题都是由编译器自动生成的成员函数引起的，下面介绍这个主题。

12.1.2 特殊成员函数

StringBad 类的问题是由特殊成员函数引起的。这些成员函数是自动定义的，就 StringBad 而言，这些函数的行为与类设计不符。具体地说，C++自动提供了下面这些成员函数：

- 默认构造函数，如果没有定义构造函数；
- 默认析构函数，如果没有定义；
- 复制构造函数，如果没有定义；
- 赋值运算符，如果没有定义；
- 地址运算符，如果没有定义。

更准确地说，编译器将生成上述最后三个函数的定义——如果程序使用对象的方式要求这样做。例如，如果您将一个对象赋给另一个对象，编译器将提供赋值运算符的定义。

结果表明，StringBad 类中的问题是由隐式复制构造函数和隐式赋值运算符引起的。

隐式地址运算符返回调用对象的地址（即 this 指针的值）。这与我们的初衷是一致的，在此不详细讨论该成员函数。默认析构函数不执行任何操作，因此这里也不讨论，但需要指出的是，这个类已经提供默认构造函数。至于其他成员函数还需要进一步讨论。

C++11 提供了另外两个特殊成员函数：移动构造函数（move constructor）和移动赋值运算符（move assignment operator），这将在第 18 章讨论。

1. 默认构造函数

如果没有提供任何构造函数，C++将创建默认构造函数。例如，假如定义了一个 Klunk 类，但没有提供任何构造函数，则编译器将提供下述默认构造函数：

```
Klunk::Klunk() { }  // implicit default constructor
```

也就是说，编译器将提供一个不接受任何参数，也不执行任何操作的构造函数（默认的默认构造函数），这是因为创建对象时总是会调用构造函数：

```
Klunk lunk;  // invokes default constructor
```

默认构造函数使 Lunk 类似于一个常规的自动变量，也就是说，它的值在初始化时是未知的。

如果定义了构造函数，C++将不会定义默认构造函数。如果希望在创建对象时不显式地对它进行初始化，则必须显式地定义默认构造函数。这种构造函数没有任何参数，但可以使用它来设置特定的值：

```
Klunk::Klunk()  // explicit default constructor
{
```

```
    klunk_ct = 0;
    ...
}
```

带参数的构造函数也可以是默认构造函数，只要所有参数都有默认值。例如，Klunk 类可以包含下述内联构造函数：

```
Klunk(int n = 0) { klunk_ct = n; }
```

但只能有一个默认构造函数。也就是说，不能这样做：

```
Klunk() { klunk_ct = 0 }              // constructor #1
Klunk(int n = 0) { klunk_ct = n; }    // ambiguous constructor #2
```

这为何有二义性呢？请看下面两个声明：

```
Klunk kar(10);     // clearly matches Klunt(int n)
Klunk bus;         // could match either constructor
```

第二个声明既与构造函数#1（没有参数）匹配，也与构造函数#2（使用默认参数 0）匹配。这将导致编译器发出一条错误消息。

2. 复制构造函数

复制构造函数用于将一个对象复制到新创建的对象中。也就是说，它用于初始化过程中（包括按值传递参数），而不是常规的赋值过程中。类的复制构造函数原型通常如下：

```
Class_name(const Class_name &);
```

它接受一个指向类对象的常量引用作为参数。例如，String 类的复制构造函数的原型如下：

```
StringBad(const StringBad &);
```

对于复制构造函数，需要知道两点：何时调用和有何功能。

3. 何时调用复制构造函数

新建一个对象并将其初始化为同类现有对象时，复制构造函数都将被调用。这在很多情况下都可能发生，最常见的情况是将新对象显式地初始化为现有的对象。例如，假设 motto 是一个 StringBad 对象，则下面 4 种声明都将调用复制构造函数：

```
StringBad ditto(motto);  // calls StringBad(const StringBad &)
StringBad metoo = motto; // calls StringBad(const StringBad &)
StringBad also = StringBad(motto);
                         // calls StringBad(const StringBad &)
StringBad * pStringBad = new StringBad(motto);
                         // calls StringBad(const StringBad &)
```

其中中间的 2 种声明可能会使用复制构造函数直接创建 metoo 和 also，也可能使用复制构造函数生成一个临时对象，然后将临时对象的内容赋给 metoo 和 also，这取决于具体的实现。最后一种声明使用 motto 初始化一个匿名对象，并将新对象的地址赋给 pstring 指针。

每当程序生成了对象副本时，编译器都将使用复制构造函数。具体地说，当函数按值传递对象（如程序清单 12.3 中的 callme2()）或函数返回对象时，都将使用复制构造函数。记住，按值传递意味着创建原始变量的一个副本。编译器生成临时对象时，也将使用复制构造函数。例如，将 3 个 Vector 对象相加时，编译器可能生成临时的 Vector 对象来保存中间结果。何时生成临时对象随编译器而异，但无论是哪种编译器，当按值传递和返回对象时，都将调用复制构造函数。具体地说，程序清单 12.3 中的函数调用将调用下面的复制构造函数：

```
callme2(headline2);
```

程序使用复制构造函数初始化 sb——callme2()函数的 StringBad 型形参。

由于按值传递对象将调用复制构造函数，因此应该按引用传递对象。这样可以节省调用构造函数的时间以及存储新对象的空间。

4. 默认的复制构造函数的功能

默认的复制构造函数逐个复制非静态成员（成员复制也称为浅复制），复制的是成员的值。在程序清单

12.3 中，下述语句：

```
StringBad sailor = sports;
```

与下面的代码等效（只是由于私有成员是无法访问的，因此这些代码不能通过编译）：

```
StringBad sailor;
sailor.str = sports.str;
sailor.len = sports.len;
```

如果成员本身就是类对象，则将使用这个类的复制构造函数来复制成员对象。静态函数（如 num_strings）不受影响，因为它们属于整个类，而不是各个对象。图 12.2 说明了隐式复制构造函数执行的操作。

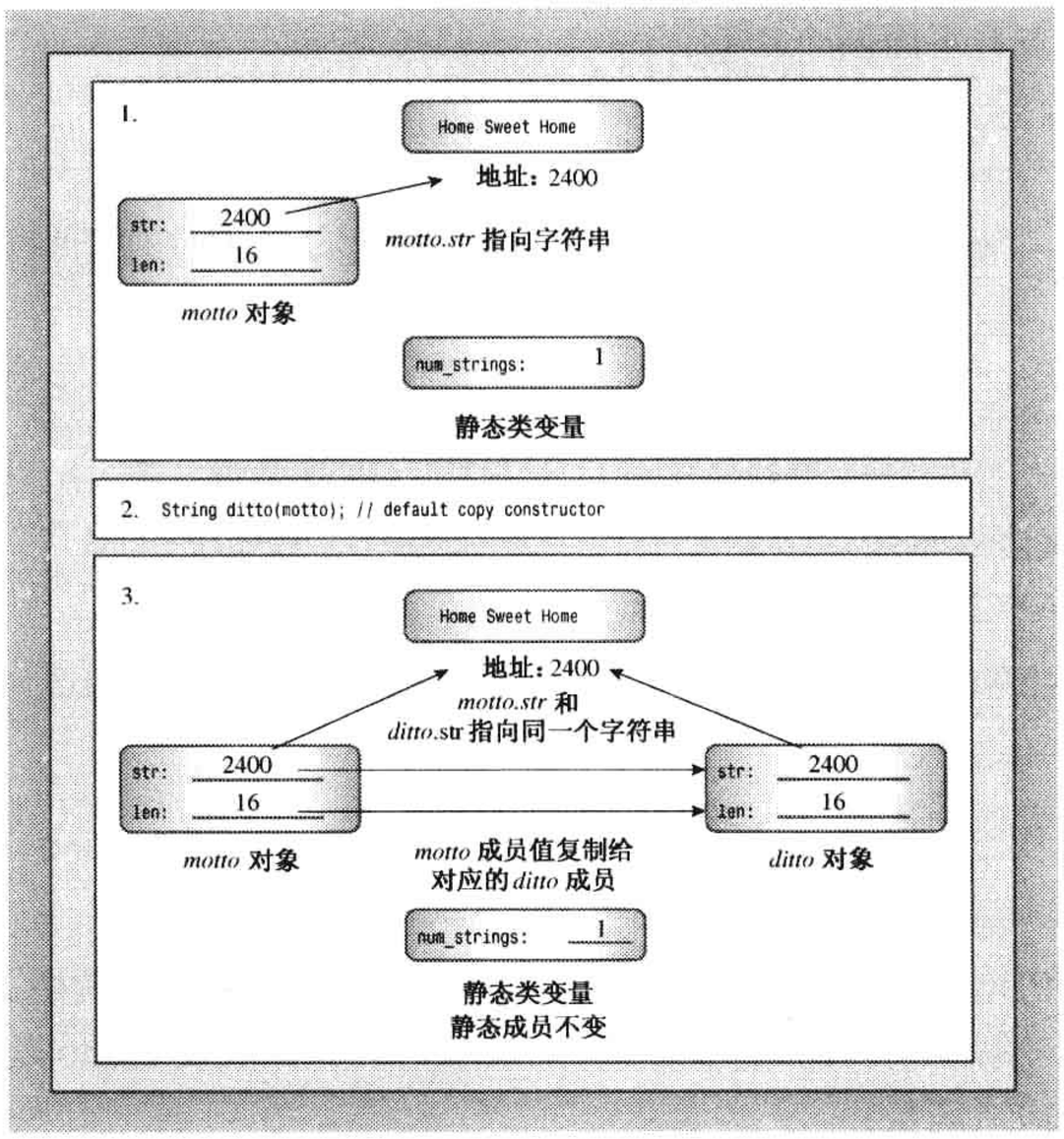


图 12.2　逐个复制成员

12.1.3　回到 Stringbad：复制构造函数的哪里出了问题

现在介绍程序清单 12.3 的两个异常之处（假设输出为该程序清单后面列出的）。首先，程序的输出表明，析构函数的调用次数比构造函数的调用次数多 2，原因可能是程序确实使用默认的复制构造函数另外创建了两个对象。当 callme2()被调用时，复制构造函数被用来初始化 callme2()的形参，还被用来将对象 sailor 初始化为对象 sports。默认的复制构造函数不说明其行为，因此它不指出创建过程，也不增加计数器 num_strings 的值。但析构函数更新了计数，并且在任何对象过期时都将被调用，而不管对象是如何被创建的。这是一个问题，因为这意味着程序无法准确地记录对象计数。解决办法是提供一个对计数进行更新的显式复制构造函数：

```
StringBad::StringBad(const String & s)
{
    num_strings++;
    ...// important stuff to go here
}
```

提示：如果类中包含这样的静态数据成员，即其值将在新对象被创建时发生变化，则应该提供一个显式复制构造函数来处理计数问题。

第二个异常之处更微妙，也更危险，其症状之一是字符串内容出现乱码：

```
headline2: Dû°
```

原因在于隐式复制构造函数是按值进行复制的。例如，对于程序清单 12.3，隐式复制构造函数的功能相当于：

```
sailor.str = sport.str;
```

这里复制的并不是字符串，而是一个指向字符串的指针。也就是说，将 sailor 初始化为 sports 后，得到的

是两个指向同一个字符串的指针。当 operator <<()函数使用指针来显示字符串时，这并不会出现问题。但当析构函数被调用时，这将引发问题。析构函数 StringBad 释放 str 指针指向的内存，因此释放 sailor 的效果如下：

```
delete [] sailor.str;    // delete the string that ditto.str points to
```

sailor.str 指针指向“Spinach Leaves Bowl for Dollars”，因为它被赋值为 sports.str，而 sports.str 指向的正是上述字符串。所以 delete 语句将释放字符串“Spinach Leaves Bowl for Dollars”占用的内存。

然后，释放 sports 的效果如下：

```
delete [] sports.str;   // effect is undefined
```

sports.str 指向的内存已经被 sailor 的析构函数释放，这将导致不确定的、可能有害的后果。程序清单 12.3 中的程序生成受损的字符串，这通常是内存管理不善的表现。

另一个症状是，试图释放内存两次可能导致程序异常终止。例如，Microsoft Visual C++ 2010（调试模式）显示一个错误消息窗口，指出“Debug Assertion Failed!”；而在 Linux 中，g++ 4.4.1 显示消息“double free or corruption”并终止程序运行。其他系统可能提供不同的消息，甚至不提供任何消息，但程序中的错误是相同的。

1. 定义一个显式复制构造函数以解决问题

解决类设计中这种问题的方法是进行深度复制（deep copy）。也就是说，复制构造函数应当复制字符串并将副本的地址赋给 str 成员，而不仅仅是复制字符串地址。这样每个对象都有自己的字符串，而不是引用另一个对象的字符串。调用析构函数时都将释放不同的字符串，而不会试图去释放已经被释放的字符串。可以这样编写 String 的复制构造函数：

```
StringBad::StringBad(const StringBad & st)
{
    num_strings++;              // handle static member update
    len = st.len;               // same length
    str = new char [len + 1];   // allot space
    std::strcpy(str, st.str);   // copy string to new location
    cout << num_strings << ": \"" << str
         << "\" object created\n"; // For Your Information
}
```

必须定义复制构造函数的原因在于，一些类成员是使用 new 初始化的、指向数据的指针，而不是数据本身。图 12.3 说明了深度复制。

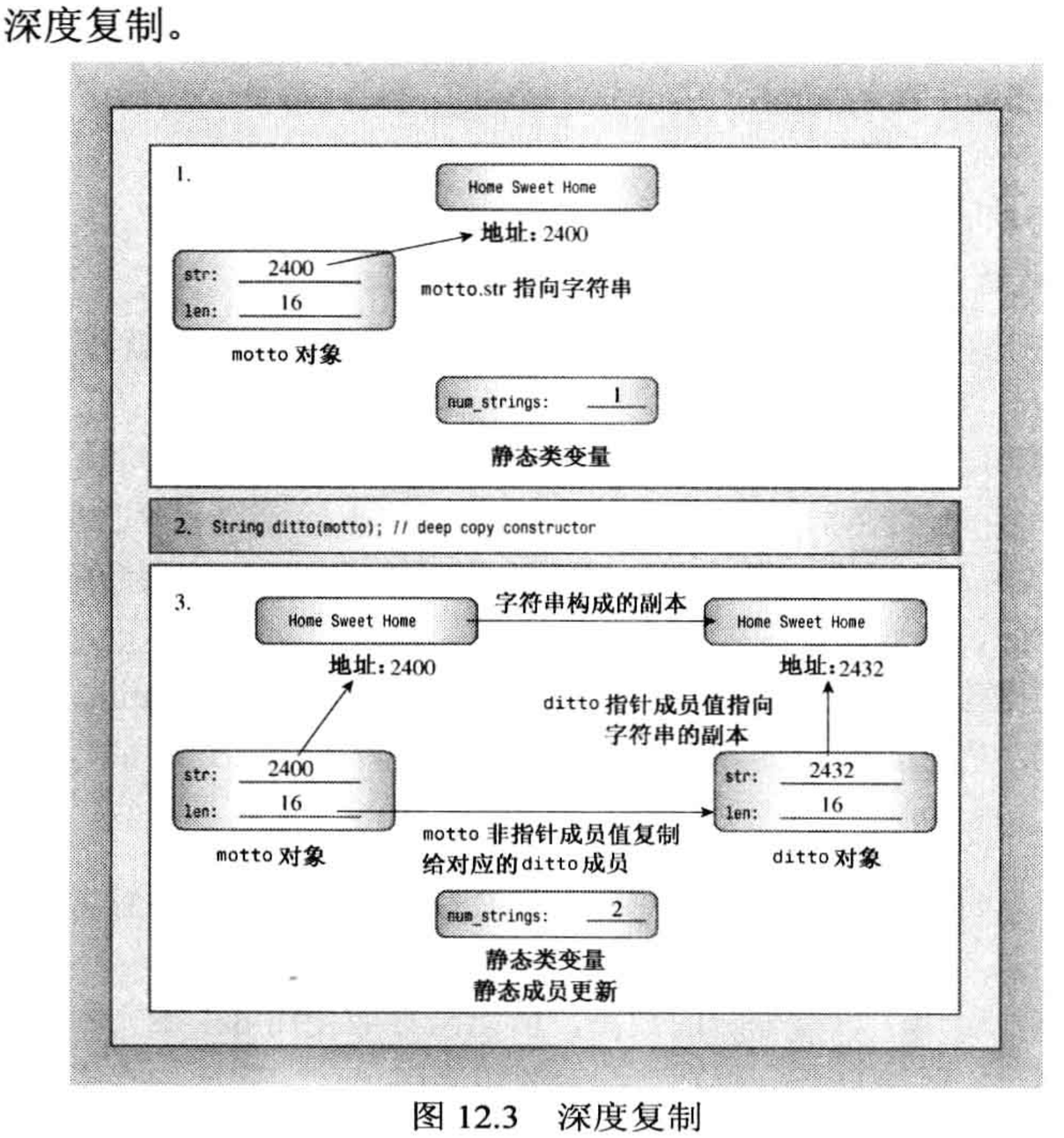


图 12.3　深度复制

警告：如果类中包含了使用 new 初始化的指针成员，应当定义一个复制构造函数，以复制指向的数据，而不是指针，这被称为深度复制。复制的另一种形式（成员复制或浅复制）只是复制指针值。浅复制仅浅浅地复制指针信息，而不会深入“挖掘”以复制指针引用的结构。

12.1.4 Stringbad 的其他问题：赋值运算符

并不是程序清单 12.3 的所有问题都可以归咎于默认的复制构造函数，还需要看一看默认的赋值运算符。ANSI C 允许结构赋值，而 C++允许类对象赋值，这是通过自动为类重载赋值运算符实现的。这种运算符的原型如下：

```
Class_name & Class_name::operator=(const Class_name &);
```

它接受并返回一个指向类对象的引用。例如，StringBad 类的赋值运算符的原型如下：

```
StringBad & StringBad::operator=(const StringBad &);
```

1. 赋值运算符的功能以及何时使用它

将已有的对象赋给另一个对象时，将使用重载的赋值运算符：

```
StringBad headline1("Celery Stalks at Midnight");
...
StringBad knot;
knot = headline1;   // assignment operator invoked
```

初始化对象时，并不一定会使用赋值运算符：

```
StringBad metoo = knot; // use copy constructor, possibly assignment, too
```

这里，metoo 是一个新创建的对象，被初始化为 knot 的值，因此使用复制构造函数。然而，正如前面指出的，实现时也可能分两步来处理这条语句：使用复制构造函数创建一个临时对象，然后通过赋值将临时对象的值复制到新对象中。这就是说，初始化总是会调用复制构造函数，而使用=运算符时也可能调用赋值运算符。

与复制构造函数相似，赋值运算符的隐式实现也对成员进行逐个复制。如果成员本身就是类对象，则程序将使用为这个类定义的赋值运算符来复制该成员，但静态数据成员不受影响。

2. 赋值的问题出在哪里

程序清单 12.3 将 headline1 赋给 knot：

```
knot = headline1;
```

为 knot 调用析构函数时，将显示下面的消息：

```
"Celery Stalks at Midnight" object deleted, 2 left
```

为 Headline1 调用析构函数时，显示如下消息（有些实现方式在此之前就异常终止了）：

```
"-|" object deleted, -2 left
```

出现的问题与隐式复制构造函数相同：数据受损。这也是成员复制的问题，即导致 headline1.str 和 knot.str 指向相同的地址。因此，当对 knot 调用析构函数时，将删除字符串“Celery Stalks at Midnight”；当对 headline1 调用析构函数时，将试图删除前面已经删除的字符串。正如前面指出的，试图删除已经删除的数据导致的结果是不确定的，因此可能改变内存中的内容，导致程序异常终止。要指出的是，如果操作结果是不确定的，则执行的操作将随编译器而异，包括显示独立声明（Declaration of Independence）或释放隐藏文件占用的硬盘空间。当然，编译器开发人员通常不会花时间添加这样的行为。

3. 解决赋值的问题

对于由于默认赋值运算符不合适而导致的问题，解决办法是提供赋值运算符（进行深度复制）定义。其实现与复制构造函数相似，但也有一些差别。

- 由于目标对象可能引用了以前分配的数据，所以函数应使用 delete[ ]来释放这些数据。
- 函数应当避免将对象赋给自身；否则，给对象重新赋值前，释放内存操作可能删除对象的内容。

- 函数返回一个指向调用对象的引用。

通过返回一个对象，函数可以像常规赋值操作那样，连续进行赋值，即如果 S0、S1 和 S2 都是 StringBad 对象，则可以编写这样的代码：

```
S0 = S1 = S2;
```

使用函数表示法时，上述代码为：

```
S0.operator=(S1.operator=(S2));
```

因此，S1.operator=（S2）的返回值是函数 S0.operator=()的参数。

因为返回值是一个指向 StringBad 对象的引用，因此参数类型是正确的。

下面的代码说明了如何为 StringBad 类编写赋值运算符：

```
StringBad & StringBad::operator=(const StringBad & st)
{
    if (this == &st)            // object assigned to itself
        return *this;           // all done
    delete [] str;              // free old string
    len = st.len;
    str = new char [len + 1];   // get space for new string
    std::strcpy(str, st.str);   // copy the string
    return *this;               // return reference to invoking object
}
```

代码首先检查自我复制，这是通过查看赋值运算符右边的地址（&s）是否与接收对象（this）的地址相同来完成的。如果相同，程序将返回*this，然后结束。第 10 章介绍过，赋值运算符是只能由类成员函数重载的运算符之一。

如果地址不同，函数将释放 str 指向的内存，这是因为稍后将把一个新字符串的地址赋给 str。如果不首先使用 delete 运算符，则上述字符串将保留在内存中。由于程序中不再包含指向该字符串的指针，因此这些内存被浪费掉。

接下来的操作与复制构造函数相似，即为新字符串分配足够的内存空间，然后将赋值运算符右边的对象中的字符串复制到新的内存单元中。

上述操作完成后，程序返回*this 并结束。

赋值操作并不创建新的对象，因此不需要调整静态数据成员 num_strings 的值。

将前面介绍的复制构造函数和赋值运算符添加到 StringBad 类中后，所有的问题都解决了。例如，下面是在完成上述修改后，程序输出的最后几行：

```
End of main()
"Celery Stalks at Midnight" object deleted, 4 left
"Spinach Leaves Bowl for Dollars" object deleted, 3 left
"Spinach Leaves Bowl for Dollars" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Celery Stalks at Midnight" object deleted, 0 left
```

现在，对象计数是正确的，字符串也没有被损坏。

12.2　改进后的新 String 类

有了更丰富的知识后，可以对 StringBad 类进行修订，将它重命名为 String 了。首先，添加前面介绍过的复制构造函数和赋值运算符，使类能够正确管理类对象使用的内存。其次，由于您已经知道对象何时被创建和释放，因此可以让类构造函数和析构函数保持沉默，不再在每次被调用时都显示消息。另外，也不用再监视构造函数的工作情况，因此可以简化默认构造函数，使之创建一个空字符串，而不是“C++”。

接下来，可以在类中添加一些新功能。String 类应该包含标准字符串函数库 cstring 的所有功能，才会比较有用，但这里只添加足以说明其工作原理的功能（注意，String 类只是一个用作说明的示例，而 C++

标准 string 类的内容丰富得多）。具体地说，将添加以下方法：

```
int length () const { return len; }
friend bool operator<(const String &st, const String &st2);
friend bool operator>(const String &st1, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend operator>>(istream & is, String & st);
char & operator[](int i);
const char & operator[](int i) const;
static int HowMany();
```

第一个新方法返回被存储的字符串的长度。接下来的 3 个友元函数能够对字符串进行比较。Operator>>()函数提供了简单的输入功能；两个 operator[]()函数提供了以数组表示法访问字符串中各个字符的功能。静态类方法 Howmany()将补充静态类数据成员 num_string。下面来看一看具体情况。

12.2.1 修订后的默认构造函数

请注意新的默认构造函数，它与下面类似：

```
String::String()
{
    len = 0;
    str = new char[1];
    str[0] = '\0';                // default string
}
```

您可能会问，为什么代码为：

```
str = new char[1];
```

而不是：

```
str = new char;
```

上面两种方式分配的内存量相同，区别在于前者与类析构函数兼容，而后者不兼容。析构函数中包含如下代码：

```
delete [] str;
```

delete[]与使用 new[]初始化的指针和空指针都兼容。因此对于下述代码：

```
str = new char[1];
str[0] = '\0';                // default string
```

可修改为：

```
str = 0; // sets str to the null pointer
```

对于以其他方式初始化的指针，使用 delete [ ]时，结果将是不确定的：

```
char words[15] = "bad idea";
char * p1= words;
char * p2 = new char;
char * p3;
delete [] p1; // undefined, so don't do it
delete [] p2; // undefined, so don't do it
delete [] p3; // undefined, so don't do it
```

C++11 空指针

在 C++98 中，字面值 0 有两个含义：可以表示数字值零，也可以表示空指针，这使得阅读程序的人和编译器难以区分。有些程序员使用（void *） 0 来标识空指针（空指针本身的内部表示可能不是零），还有些程序员使用 NULL，这是一个表示空指针的 C 语言宏。C++11 提供了更好的解决方案：引入新关键字 nullptr，用于表示空指针。您仍可像以前一样使用 0——否则大量现有的代码将非法，但建议您使用 nullptr：

```
str = nullptr;  // C++11 null pointer notation
```

12.2.2　比较成员函数

在 String 类中，执行比较操作的方法有 3 个。如果按字母顺序（更准确地说，按照机器排序序列），第一个字符串在第二个字符串之前，则 Operator<( )函数返回 true。要实现字符串比较函数，最简单的方法是使用标准的 trcmp()函数，如果依照字母顺序，第一个参数位于第二个参数之前，则该函数返回一个负值；如果两个字符串相同，则返回 0；如果第一个参数位于第二个参数之后，则返回一个正值。因此，可以这样使用 strcmp()：

```
bool operator<(const String &st1, const String &st2)
{
    if (std::strcmp(st1.str, st2.str) < 0)
        return true;
    else
        return false;
}
```

因为内置的>运算符返回的是一个布尔值，所以可以将代码进一步简化为：

```
bool operator<(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) < 0);
}
```

同样，可以按照下面的方式来编写另外两个比较函数：

```
bool operator>(const String &st1, const String &st2)
{
    return st2 < st1;
}
bool operator==(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) == 0);
}
```

第一个定义利用了<运算符来表示>运算符，对于内联函数，这是一种很好的选择。

将比较函数作为友元，有助于将 String 对象与常规的 C 字符串进行比较。例如，假设 answer 是 String 对象，则下面的代码：

```
if ("love" == answer)
```

将被转换为：

```
if (operator==("love", answer))
```

然后，编译器将使用某个构造函数将代码转换为：

```
if (operator==(String("love"), answer))
```

这与原型是相匹配的。

12.2.3　使用中括号表示法访问字符

对于标准 C-风格字符串来说，可以使用中括号来访问其中的字符：

```
char city[40] = "Amsterdam";
cout << city[0] << endl; // display the letter A
```

在 C++中，两个中括号组成一个运算符——中括号运算符，可以使用方法 operator[ ]( )来重载该运算符。通常，二元 C++运算符（带两个操作数）位于两个操作数之间，例如 2 +5。但对于中括号运算符，一个操作数位于第一个中括号的前面，另一个操作数位于两个中括号之间。因此，在表达式 city[0]中，city 是第一个操作数，[]是运算符，0 是第二个操作数。

假设 opera 是一个 String 对象：

```
String opera("The Magic Flute");
```

则对于表达式 opera[4]，C++将查找名称和特征标与此相同的方法：

```
String::operator[](int i)
```

如果找到匹配的原型，编译器将使用下面的函数调用来替代表达式 opera[4]：

```
opera.operator[](4)
```

opera 对象调用该方法，数组下标 4 成为该函数的参数。

下面是该方法的简单实现：

```
char & String::operator[](int i)
{
    return str[i];
}
```

有了上述定义后，语句：

```
cout << opera[4];
```

将被转换为：

```
cout << opera.operator[4];
```

返回值是 opera.str[4]（字符 M）。由此，公有方法可以访问私有数据。

将返回类型声明为 char &，便可以给特定元素赋值。例如，可以编写这样的代码：

```
String means("might");
means[0] = 'r';
```

第二条语句将被转换为一个重载运算符函数调用：

```
means.operator[][0] = 'r';
```

这里将 r 赋给方法的返回值，而函数返回的是指向 means.str[0]的引用，因此上述代码等同于下面的代码：

```
means.str[0] = 'r';
```

代码的最后一行访问的是私有数据，但由于 operator [ ]( )是类的一个方法，因此能够修改数组的内容。最终的结果是“might”被改为“right”。

假设有下面的常量对象：

```
const String answer("futile");
```

如果只有上述 operator[ ]( )定义，则下面的代码将出错：

```
cout << answer[1];  // compile-time error
```

原因是 answer 是常量，而上述方法无法确保不修改数据（实际上，有时该方法的工作就是修改数据，因此无法确保不修改数据）。

但在重载时，C++将区分常量和非常量函数的特征标，因此可以提供另一个仅供 const String 对象使用的 operator[]()版本：

```
// for use with const String objects
const char & String::operator[](int i) const
{
    return str[i];
}
```

有了上述定义后，就可以读/写常规 String 对象了；而对于 const String 对象，则只能读取其数据：

```
String text("Once upon a time");
const String answer("futile");
cout << text[1];    // ok, uses non-const version of operator[]()
cout << answer[1];  // ok, uses const version of operator[]()
cin >> text[1];     // ok, uses non-const version of operator[]()
cin >> answer[1];   // compile-time error
```

12.2.4　静态类成员函数

可以将成员函数声明为静态的（函数声明必须包含关键字 static，但如果函数定义是独立的，则其中不能包含关键字 static），这样做有两个重要的后果。

首先，不能通过对象调用静态成员函数；实际上，静态成员函数甚至不能使用 this 指针。如果静态成员函数是在公有部分声明的，则可以使用类名和作用域解析运算符来调用它。例如，可以给 String 类添加一个名为 HowMany( )的静态成员函数，方法是在类声明中添加如下原型/定义：

```
static int HowMany() { return num_strings; }
```

调用它的方式如下：

```
int count = String::HowMany();  // invoking a static member function
```

其次，由于静态成员函数不与特定的对象相关联，因此只能使用静态数据成员。例如，静态方法 HowMany()可以访问静态成员 num_string，但不能访问 str 和 len。

同样，也可以使用静态成员函数设置类级（classwide）标记，以控制某些类接口的行为。例如，类级标记可以控制显示类内容的方法所使用的格式。

12.2.5　进一步重载赋值运算符

介绍针对 String 类的程序清单之前，先来考虑另一个问题。假设要将常规字符串复制到 String 对象中。例如，假设使用 getline()读取了一个字符串，并要将这个字符串放置到 String 对象中，前面定义的类方法让您能够这样编写代码：

```
String name;
char temp[40];
cin.getline(temp, 40);
name = temp;  // use constructor to convert type
```

但如果经常需要这样做，这将不是一种理想的解决方案。为解释其原因，先来回顾一下最后一条语句是怎样工作的。

1．程序使用构造函数 String（const char *）来创建一个临时 String 对象，其中包含 temp 中的字符串副本。第 11 章介绍过，只有一个参数的构造函数被用作转换函数。

2．本章后面的程序清单 12.6 中的程序使用 String & String::operator=（const String &）函数将临时对象中的信息复制到 name 对象中。

3．程序调用析构函数~String()删除临时对象。

为提高处理效率，最简单的方法是重载赋值运算符，使之能够直接使用常规字符串，这样就不用创建和删除临时对象了。下面是一种可能的实现：

```
String & String::operator=(const char * s)
{
    delete [] str;
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
    return *this;
}
```

一般说来，必须释放 str 指向的内存，并为新字符串分配足够的内存。

程序清单 12.4 列出了修订后的类声明。除了前面提到过的修改之外，这里还定义了一个 CINLIM 常量，用于实现 operator>>()。

程序清单 12.4　string1.h

```
// string1.h -- fixed and augmented string class definition
```

```
#ifndef STRING1_H_
#define STRING1_H_
#include <iostream>
using std::ostream;
using std::istream;

class String
{
private:
    char * str;             // pointer to string
    int len;                // length of string
    static int num_strings; // number of objects
    static const int CINLIM = 80;  // cin input limit
public:
// constructors and other methods
    String(const char * s); // constructor
    String();               // default constructor
    String(const String &); // copy constructor
    ~String();              // destructor
    int length () const { return len; }
// overloaded operator methods
    String & operator=(const String &);
    String & operator=(const char *);
    char & operator[](int i);
    const char & operator[](int i) const;
// overloaded operator friends
    friend bool operator<(const String &st, const String &st2);
    friend bool operator>(const String &st1, const String &st2);
    friend bool operator==(const String &st, const String &st2);
    friend ostream & operator<<(ostream & os, const String & st);
    friend istream & operator>>(istream & is, String & st);
// static function
    static int HowMany();
};
#endif
```

程序清单 12.5 给出了修订后的方法定义。

程序清单 12.5　string1.cpp

```
// string1.cpp -- String class methods
#include <cstring>                // string.h for some
#include "string1.h"              // includes <iostream>
using std::cin;
using std::cout;

// initializing static class member

int String::num_strings = 0;

// static method
int String::HowMany()
{
    return num_strings;
}
```

```
// class methods
String::String(const char * s)     // construct String from C string
{
    len = std::strlen(s);          // set size
    str = new char[len + 1];       // allot storage
    std::strcpy(str, s);           // initialize pointer
    num_strings++;                 // set object count
}

String::String()                   // default constructor
{
    len = 4;
    str = new char[1];
    str[0] = '\0';                 // default string
    num_strings++;
}

String::String(const String & st)
{
    num_strings++;             // handle static member update
    len = st.len;              // same length
    str = new char [len + 1];  // allot space
    std::strcpy(str, st.str);  // copy string to new location
}

String::~String()                      // necessary destructor
{
    --num_strings;                     // required
    delete [] str;                     // required
}

// overloaded operator methods

    // assign a String to a String
String & String::operator=(const String & st)
{
    if (this == &st)
        return *this;
    delete [] str;
    len = st.len;
    str = new char[len + 1];
    std::strcpy(str, st.str);
    return *this;
}

    // assign a C string to a String
String & String::operator=(const char * s)
{
    delete [] str;
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
    return *this;
}
```

```
    // read-write char access for non-const String
char & String::operator[](int i)
{
    return str[i];
}

    // read-only char access for const String
const char & String::operator[](int i) const
{
    return str[i];
}

// overloaded operator friends

bool operator<(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) < 0);
}

bool operator>(const String &st1, const String &st2)
{
    return st2 < st1;
}

bool operator==(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) == 0);
}

    // simple String output
ostream & operator<<(ostream & os, const String & st)
{
    os << st.str;
    return os;
}

    // quick and dirty String input
istream & operator>>(istream & is, String & st)
{
    char temp[String::CINLIM];
    is.get(temp, String::CINLIM);
    if (is)
        st = temp;
    while (is && is.get() != '\n')
        continue;
    return is;
}
```

重载>>运算符提供了一种将键盘输入行读入到 String 对象中的简单方法。它假定输入的字符数不多于 String::CINLIM 的字符数，并丢弃多余的字符。在 if 条件下，如果由于某种原因（如到达文件尾或 get（char *, int）读取的是一个空行）导致输入失败，istream 对象的值将置为 false。

程序清单 12.6 通过一个小程序来使用这个类，该程序允许输入几个字符串。程序首先提示用户输入，然后将用户输入的字符串存储到 String 对象中，并显示它们，最后指出哪个字符串最短、哪个字符串按字

母顺序排在最前面。

程序清单 12.6　sayings1.cpp

```
// sayings1.cpp -- using expanded String class
// compile with string1.cpp
#include <iostream>
#include "string1.h"
const int ArSize = 10;
const int MaxLen =81;
int main()
{
    using std::cout;
    using std::cin;
    using std::endl;
    String name;
    cout <<"Hi, what's your name?\n>> ";
    cin >> name;

    cout << name << ", please enter up to " << ArSize
         << " short sayings <empty line to quit>:\n";
    String sayings[ArSize];     // array of objects
    char temp[MaxLen];          // temporary string storage
    int i;
    for (i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while (cin && cin.get() != '\n')
            continue;
        if (!cin || temp[0] == '\0')    // empty line?
            break;              // i not incremented
        else
            sayings[i] = temp;  // overloaded assignment
    }
    int total = i;              // total # of lines read

    if ( total > 0)
    {
        cout << "Here are your sayings:\n";
        for (i = 0; i < total; i++)
            cout << sayings[i][0] << ": " << sayings[i] << endl;

        int shortest = 0;
        int first = 0;
        for (i = 1; i < total; i++)
        {
            if (sayings[i].length() < sayings[shortest].length())
                shortest = i;
            if (sayings[i] < sayings[first])
                first = i;
        }
        cout << "Shortest saying:\n" << sayings[shortest] << endl;;
        cout << "First alphabetically:\n" << sayings[first] << endl;
        cout << "This program used "<< String::HowMany()
```

```
            << " String objects. Bye.\n";
    }
    else
        cout << "No input! Bye.\n";
     return 0;
}
```

注意：较早的 get（char *, int）版本在读取空行后，返回的值不为 false。然而，对于这些版本来说，如果读取了一个空行，则字符串中第一个字符将是一个空字符。这个示例使用了下述代码：

```
if (!cin || temp[0] == '\0')     // empty line?
    break;                       // i not incremented
```

如果实现遵循了最新的 C++标准，则 if 语句中的第一个条件将检测到空行，第二个条件用于旧版本实现中检测空行。

程序清单 12.6 中程序要求用户输入至多 10 条谚语。每条谚语都被读到一个临时字符数组，然后被复制到 String 对象中。如果用户输入空行，break 语句将终止输入循环。显示用户的输入后，程序使用成员函数 length()和 operator <()来确定最短的字符串以及按字母顺序排列在最前面的字符串。程序还使用下标运算符（[]）提取每条谚语的第一个字符，并将其放在该谚语的最前面。下面是运行情况：

```
Hi, what's your name?
>> Misty Gutz
Misty Gutz, please enter up to 10 short sayings <empty line to quit>:
1: a fool and his money are soon parted
2: penny wise, pound foolish
3: the love of money is the root of much evil
4: out of sight, out of mind
5: absence makes the heart grow fonder
6: absinthe makes the hart grow fonder
7:
Here are your sayings:
a: a fool and his money are soon parted
p: penny wise, pound foolish
t: the love of money is the root of much evil
o: out of sight, out of mind
a: absence makes the heart grow fonder
a: absinthe makes the hart grow fonder
Shortest saying:
penny wise, pound foolish
First alphabetically:
a fool and his money are soon parted
This program used 11 String objects. Bye.
```

12.3 在构造函数中使用 new 时应注意的事项

至此，您知道使用 new 初始化对象的指针成员时必须特别小心。具体地说，应当这样做。

- 如果在构造函数中使用 new 来初始化指针成员，则应在析构函数中使用 delete。
- new 和 delete 必须相互兼容。new 对应于 delete，new[ ]对应于 delete[ ]。

- 如果有多个构造函数，则必须以相同的方式使用 new，要么都带中括号，要么都不带。因为只有一个析构函数，所有的构造函数都必须与它兼容。然而，可以在一个构造函数中使用 new 初始化指针，而在另一个构造函数中将指针初始化为空（0 或 C++11 中的 nullptr），这是因为 delete（无论是带中括号还是不带中括号）可以用于空指针。

NULL、0 还是 nullptr：以前，空指针可以用 0 或 NULL（在很多头文件中，NULL 是一个被定义为 0 的符号常量）来表示。C 程序员通常使用 NULL 而不是 0，以指出这是一个指针，就像使用'\0'而不是 0 来表示空字符，以指出这是一个字符一样。然而，C++传统上更喜欢用简单的 0，而不是等价的 NULL。但正如前面指出的，C++11 提供了关键字 nullptr，这是一种更好的选择。

- 应定义一个复制构造函数，通过深度复制将一个对象初始化为另一个对象。通常，这种构造函数与下面类似。

```
String::String(const String & st)
{
     num_strings++;              // handle static member update if necessary
     len = st.len;               // same length as copied string
     str = new char [len + 1]; // allot space
     std::strcpy(str, st.str); // copy string to new location
}
```

具体地说，复制构造函数应分配足够的空间来存储复制的数据，并复制数据，而不仅仅是数据的地址。另外，还应该更新所有受影响的静态类成员。

- 应当定义一个赋值运算符，通过深度复制将一个对象复制给另一个对象。通常，该类方法与下面类似：

```
String & String::operator=(const String & st)
{
     if (this == &st)              // object assigned to itself
          return *this;            // all done
     delete [] str;                // free old string
     len = st.len;
     str = new char [len + 1];     // get space for new string
     std::strcpy(str, st.str);     // copy the string
     return *this;                 // return reference to invoking object
}
```

具体地说，该方法应完成这些操作：检查自我赋值的情况，释放成员指针以前指向的内存，复制数据而不仅仅是数据的地址，并返回一个指向调用对象的引用。

12.3.1　应该和不应该

下面的摘要包含了两个不正确的示例（指出什么是不应当做的）以及一个良好的构造函数示例：

```
String::String()
{
   str = "default string";      // oops, no new []
   len = std::strlen(str);
}

String::String(const char * s)
{
   len = std::strlen(s);
   str = new char;              // oops, no []
   std::strcpy(str, s);         // oops, no room
}
```

```
String::String(const String & st)
{
        len = st.len;
        str = new char[len + 1];        // good, allocate space
        std::strcpy(str, st.str);       // good, copy value
}
```

第一个构造函数没有使用 new 来初始化 str。对默认对象调用析构函数时，析构函数使用 delete 来释放 str。对不是使用 new 初始化的指针使用 delete 时，结果将是不确定的，并可能是有害的。可将该构造函数修改为下面的任何一种形式：

```
String::String()
{
    len = 0;
    str = new char[1];  // uses new with []
    str[0] = '\0';
}
String::String()
{
    len = 0;
    str = 0;  // or, with C++11, str = nullptr;
}

String::String()
{
    static const char * s = "C++";      // initialized just once
    len = std::strlen(s);
    str = new char[len + 1];            // uses new with []
    std::strcpy(str, s);
}
```

摘录中的第二个构造函数使用了 new，但分配的内存量不正确。因此，new 返回的内存块只能保存一个字符。试图将过长的字符串复制到该内存单元中，将导致内存问题。另外，这里使用的 new 不带中括号，这与另一个构造函数的正确格式不一致。

第三个构造函数是正确的。

最后，下面的析构函数无法与前面的构造函数正常地协同工作：

```
String::~String()
{
    delete str;       // oops, should be delete [] str;
}
```

该析构函数未能正确地使用 delete。由于构造函数创建的是一个字符数组，因此析构函数应该删除一个数组。

12.3.2 包含类成员的类的逐成员复制

假设类成员的类型为 String 类或标准 string 类：

```
class Magazine
{
private:
    String title;
    string publisher;
...
};
```

String 和 string 都使用动态内存分配，这是否意味着需要为 Magazine 类编写复制构造函数和赋值运算

符？不，至少对这个类本身来说不需要。默认的逐成员复制和赋值行为有一定的智能。如果您将一个 Magazine 对象复制或赋值给另一个 Magazine 对象，逐成员复制将使用成员类型定义的复制构造函数和赋值运算符。也就是说，复制成员 title 时，将使用 String 的复制构造函数，而将成员 title 赋给另一个 Magazine 对象时，将使用 String 的赋值运算符，依此类推。然而，如果 Magazine 类因其他成员需要定义复制构造函数和赋值运算符，情况将更复杂；在这种情况下，这些函数必须显式地调用 String 和 string 的复制构造函数和赋值运算符，这将在第 13 章介绍。

12.4　有关返回对象的说明

当成员函数或独立的函数返回对象时，有几种返回方式可供选择。可以返回指向对象的引用、指向对象的 const 引用或 const 对象。到目前为止，介绍了前两种方式，但没有介绍最后一种方式，现在是复习这些方式的好时机。

12.4.1　返回指向 const 对象的引用

使用 const 引用的常见原因是旨在提高效率，但对于何时可以采用这种方式存在一些限制。如果函数返回（通过调用对象的方法或将对象作为参数）传递给它的对象，可以通过返回引用来提高其效率。例如，假设要编写函数 Max()，它返回两个 Vector 对象中较大的一个，其中 Vector 是第 11 章开发的一个类。该函数将以下面的方式被使用：

```
Vector force1(50,60);
Vector force2(10,70);
Vector max;
max = Max(force1, force2);
```

下面两种实现都是可行的：

```
// version 1
Vector Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}

// version 2
const Vector & Max(const Vector & v1, const Vector & v2)
{
    if (v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

这里有三点需要说明。首先，返回对象将调用复制构造函数，而返回引用不会。因此，第二个版本所做的工作更少，效率更高。其次，引用指向的对象应该在调用函数执行时存在。在这个例子中，引用指向 force1 或 force2，它们都是在调用函数中定义的，因此满足这种条件。第三，v1 和 v2 都被声明为 const 引用，因此返回类型必须为 const，这样才匹配。

12.4.2　返回指向非 const 对象的引用

两种常见的返回非 const 对象情形是，重载赋值运算符以及重载与 cout 一起使用的<<运算符。前者这

样做旨在提高效率，而后者必须这样做。

operator=()的返回值用于连续赋值：

```
String s1("Good stuff");
String s2, s3;
s3 = s2 = s1;
```

在上述代码中，s2.operator=()的返回值被赋给 s3。为此，返回 String 对象或 String 对象的引用都是可行的，但与 Vector 示例中一样，通过使用引用，可避免该函数调用 String 的复制构造函数来创建一个新的 String 对象。在这个例子中，返回类型不是 const，因为方法 operator=()返回一个指向 s2 的引用，可以对其进行修改。

Operator<<()的返回值用于串接输出：

```
String s1("Good stuff");
cout << s1 << "is coming!";
```

在上述代码中，operator<<（cout, s1）的返回值成为一个用于显示字符串“is coming!”的对象。返回类型必须是 ostream &，而不能仅仅是 ostream。如果使用返回类型 ostream，将要求调用 ostream 类的复制构造函数，而 ostream 没有公有的复制构造函数。幸运的是，返回一个指向 cout 的引用不会带来任何问题，因为 cout 已经在调用函数的作用域内。

12.4.3 返回对象

如果被返回的对象是被调用函数中的局部变量，则不应按引用方式返回它，因为在被调用函数执行完毕时，局部对象将调用其析构函数。因此，当控制权回到调用函数时，引用指向的对象将不再存在。在这种情况下，应返回对象而不是引用。通常，被重载的算术运算符属于这一类。请看下述示例，它再次使用了 Vector 类：

```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

返回的不是 force1，也不是 force2，force1 和 force2 在这个过程中应该保持不变。因此，返回值不能是指向在调用函数中已经存在的对象的引用。相反，在 Vector::operator+( )中计算得到的两个矢量的和被存储在一个新的临时对象中，该函数也不应返回指向该临时对象的引用，而应该返回实际的 Vector 对象，而不是引用：

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

在这种情况下，存在调用复制构造函数来创建被返回的对象的开销，然而这是无法避免的。

在上述示例中，构造函数调用 Vector（x + b.x，y + b.y）创建一个方法 operator+()能够访问的对象；而返回语句引发的对复制构造函数的隐式调用创建一个调用程序能够访问的对象。

12.4.4 返回 const 对象

前面的 Vector::operator+( )定义有一个奇异的属性，它旨在让您能够以下面这样的方式使用它：

```
net = force1 + force2;                          // 1: three Vector objects
```

然而，这种定义也允许您这样使用它：

```
force1 + force2 = net;                         // 2: dyslectic programming
cout << (force1 + force2 = net).magval() << endl; // 3: demented programming
```

这提出了三个问题。为何编写这样的语句？这些语句为何可行？这些语句有何功能？

首先，没有要编写这种语句的合理理由，但并非所有代码都是合理的。即使是程序员也会犯错。例如，

为 Vector 类定义 operator==()时，您可能错误地输入这样的代码：

```
if (force1 + force2 = net)
```

而不是：

```
if (force1 + force2 == net)
```

另外，程序员通常很有创意，这可能导致错误。

其次，这种代码之所以可行，是因为复制构造函数将创建一个临时对象来表示返回值。因此，在前面的代码中，表达式 force1 + force2 的结果为一个临时对象。在语句 1 中，该临时对象被赋给 net；在语句 2 和 3 中，net 被赋给该临时对象。

第三，使用完临时对象后，将把它丢弃。例如，对于语句 2，程序计算 force1 和 force2 之和，将结果复制到临时返回对象中，再用 net 的内容覆盖临时对象的内容，然后将该临时对象丢弃。原来的矢量全都保持不变。语句 3 显示临时对象的长度，然后将其删除。

如果您担心这种行为可能引发的误用和滥用，有一种简单的解决方案：将返回类型声明为 const Vector。例如，如果 Vector::operator+()的返回类型被声明为 const Vector，则语句 1 仍然合法，但语句 2 和语句 3 将是非法的。

总之，如果方法或函数要返回局部对象，则应返回对象，而不是指向对象的引用。在这种情况下，将使用复制构造函数来生成返回的对象。如果方法或函数要返回一个没有公有复制构造函数的类（如 ostream 类）的对象，它必须返回一个指向这种对象的引用。最后，有些方法和函数（如重载的赋值运算符）可以返回对象，也可以返回指向对象的引用，在这种情况下，应首选引用，因为其效率更高。

12.5　使用指向对象的指针

C++程序经常使用指向对象的指针，因此，这里来练习一下。程序清单 12.6 使用数组索引值来跟踪最短的字符串和按字母顺序排在最前面的字符串。另一种方法是使用指针指向这些类别的开始位置，程序清单 12.7 使用两个指向 String 的指针实现了这种方法。最初，shortest 指针指向数组中的第一个对象。每当程序找到比指向的字符串更短的对象时，就把 shortest 重新设置为指向该对象。同样，first 指针跟踪按字母顺序排在最前面的字符串。这两个指针并不创建新的对象，而只是指向已有的对象。因此，这些指针并不要求使用 new 来分配内存。

除此之外，程序清单 12.7 中的程序还使用一个指针来跟踪新对象：

```
String * favorite = new String(sayings[choice]);
```

这里指针 favorite 指向 new 创建的未被命名对象。这种特殊的语法意味着使用对象 saying [choice]来初始化新的 String 对象，这将调用复制构造函数，因为复制构造函数（const String &）的参数类型与初始化值（saying [choice]）匹配。程序使用 srand( )、rand( )和 time( )随机选择一个值。

程序清单 12.7　sayings2.cpp

```
// sayings2.cpp -- using pointers to objects
// compile with string1.cpp
#include <iostream>
#include <cstdlib>      // (or stdlib.h) for rand(), srand()
#include <ctime>        // (or time.h) for time()
#include "string1.h"
const int ArSize = 10;
const int MaxLen = 81;
int main()
{
    using namespace std;
```

```
    String name;
    cout <<"Hi, what's your name?\n>> ";
    cin >> name;

    cout << name << ", please enter up to " << ArSize
         << " short sayings <empty line to quit>:\n";
    String sayings[ArSize];
    char temp[MaxLen];                // temporary string storage
    int i;
    for (i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while (cin && cin.get() != '\n')
            continue;
        if (!cin || temp[0] == '\0') // empty line?
            break;                   // i not incremented
        else
            sayings[i] = temp;       // overloaded assignment
    }
    int total = i;                   // total # of lines read

    if (total > 0)
    {
        cout << "Here are your sayings:\n";
        for (i = 0; i < total; i++)
            cout << sayings[i] << "\n";

    // use pointers to keep track of shortest, first strings
        String * shortest = &sayings[0]; // initialize to first object
        String * first = &sayings[0];
        for (i = 1; i < total; i++)
        {
            if (sayings[i].length() < shortest->length())
                shortest = &sayings[i];
            if (sayings[i] < *first)     // compare values
                first = &sayings[i];     // assign address
        }
        cout << "Shortest saying:\n" << * shortest << endl;
        cout << "First alphabetically:\n" << * first << endl;
        srand(time(0));
        int choice = rand() % total; // pick index at random
    // use new to create, initialize new String object
        String * favorite = new String(sayings[choice]);
        cout << "My favorite saying:\n" << *favorite << endl;
        delete favorite;
    }
    else
        cout << "Not much to say, eh?\n";
    cout << "Bye.\n";
    return 0;
}
```

使用 new 初始化对象

通常，如果 Class_name 是类，value 的类型为 Type_name，则下面的语句：

```
Class_name * pclass = new Class_name(value);
```

将调用如下构造函数：

```
Class_name(Type_name);
```

这里可能还有一些琐碎的转换，例如：

```
Class_name(const Type_name &);
```

另外，如果不存在二义性，则将发生由原型匹配导致的转换（如从 int 到 double）。下面的初始化方式将调用默认构造函数：

```
Class_name * ptr = new Class_name;
```

下面是程序清单 12.7 中程序的运行情况：

```
Hi, what's your name?
>> Kirt Rood
Kirt Rood, please enter up to 10 short sayings <empty line to quit>:
1: a friend in need is a friend indeed
2: neither a borrower nor a lender be
3: a stitch in time saves nine
4: a niche in time saves stine
5: it takes a crook to catch a crook
6: cold hands, warm heart
7:
Here are your sayings:
a friend in need is a friend indeed
neither a borrower nor a lender be
a stitch in time saves nine
a niche in time saves stine
it takes a crook to catch a crook
cold hands, warm heart
Shortest saying:
cold hands, warm heart
First alphabetically:
a friend in need is a friend indeed
My favorite saying:
a stitch in time saves nine
Bye
```

由于该程序随机选择用户输入的格言，因此即使输入相同，显示的结果也可能不同。

12.5.1　再谈 new 和 delete

程序清单 12.4、程序清单 12.5 和程序清单 12.7 组成的程序在两个层次上使用了 new 和 delete。首先，它使用 new 为创建的每一个对象的名称字符串分配存储空间，这是在构造函数中进行的，因此析构函数使用 delete 来释放这些内存。因为字符串是一个字符数组，所以析构函数使用的是带中括号的 delete。这样，当对象被释放时，用于存储字符串内容的内存将被自动释放。其次，程序清单 12.7 中的代码使用 new 来为整个对象分配内存：

```
String * favorite = new String(sayings[choice]);
```

这不是为要存储的字符串分配内存，而是为对象分配内存；也就是说，为保存字符串地址的 str 指针和 len 成员分配内存（程序并没有给 num_string 成员分配内存，这是因为 num_string 成员是静态成员，它独

立于对象被保存）。创建对象将调用构造函数，后者分配用于保存字符串的内存，并将字符串的地址赋给 str。然后，当程序不再需要该对象时，使用 delete 删除它。对象是单个的，因此，程序使用不带中括号的 delete。与前面介绍的相同，这将只释放用于保存 str 指针和 len 成员的空间，并不释放 str 指向的内存，而该任务将由析构函数来完成（参见图 12.4）。

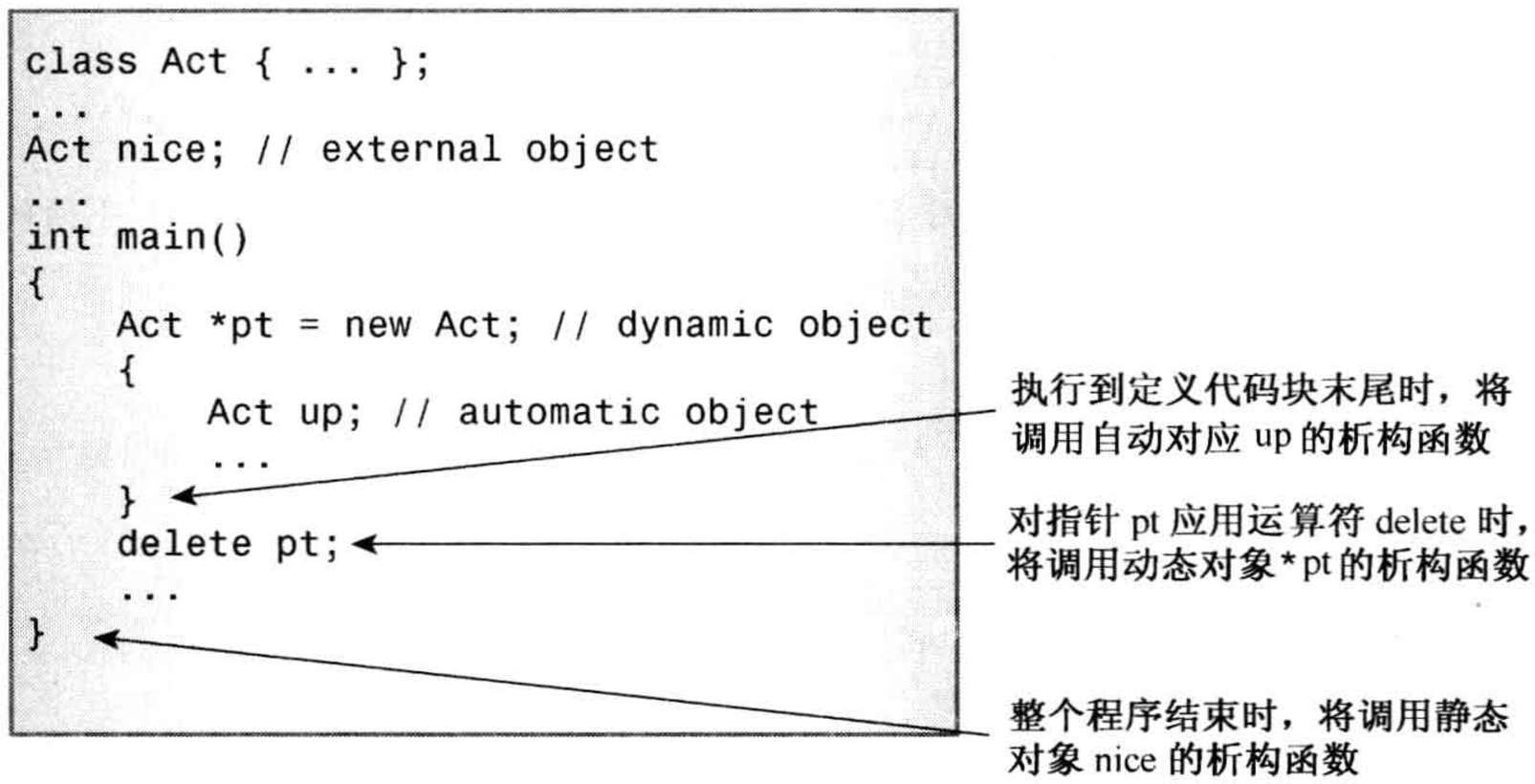


图 12.4 调用析构函数

在下述情况下析构函数将被调用（参见图 12.4）。

- 如果对象是动态变量，则当执行完定义该对象的程序块时，将调用该对象的析构函数。因此，在程序清单 12.3 中，执行完 main()时，将调用 headline[0]和 headline[1]的析构函数；执行完 callme1( )时，将调用 grub 的析构函数。
- 如果对象是静态变量（外部、静态、静态外部或来自名称空间），则在程序结束时将调用对象的析构函数。这就是程序清单 12.3 中 sports 对象所发生的情况。
- 如果对象是用 new 创建的，则仅当您显式使用 delete 删除对象时，其析构函数才会被调用。

12.5.2 指针和对象小结

使用对象指针时，需要注意几点（参见图 12.5）：

- 使用常规表示法来声明指向对象的指针：

```
String * glamour;
```

- 可以将指针初始化为指向已有的对象：

```
String * first = &sayings[0];
```

- 可以使用 new 来初始化指针，这将创建一个新的对象（有关使用 new 初始化指针的细节，请参见图 12.6）：

```
String * favorite = new String(sayings[choice]);
```

- 对类使用 new 将调用相应的类构造函数来初始化新创建的对象：

```
// invokes default constructor
String * gleep = new String;

// invokes the String(const char *) constructor
String * glop = new String("my my my");
```

```
// invokes the String(const String &) constructor
String * favorite = new String(sayings[choice]);
```

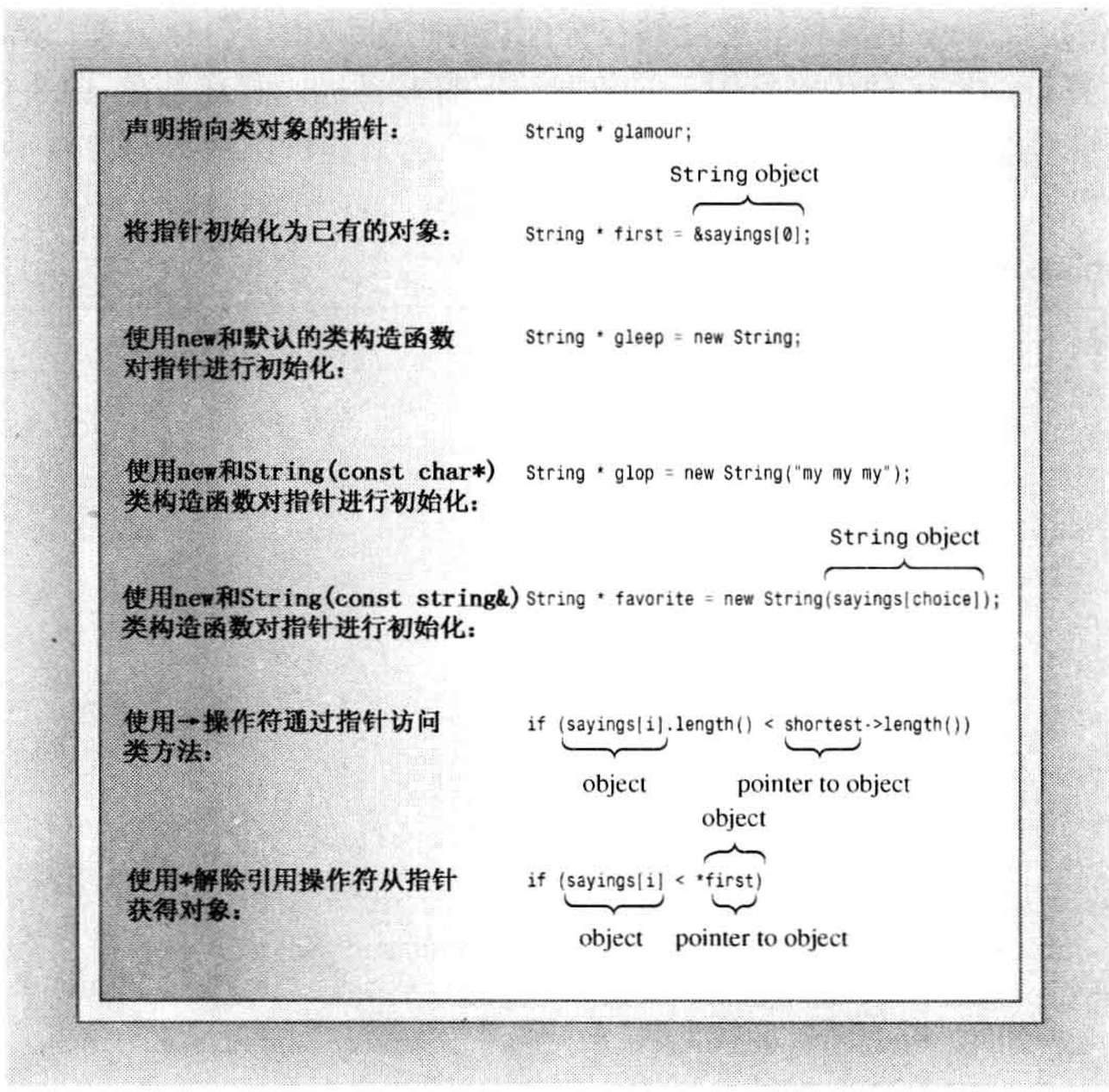


图 12.5　指针和对象

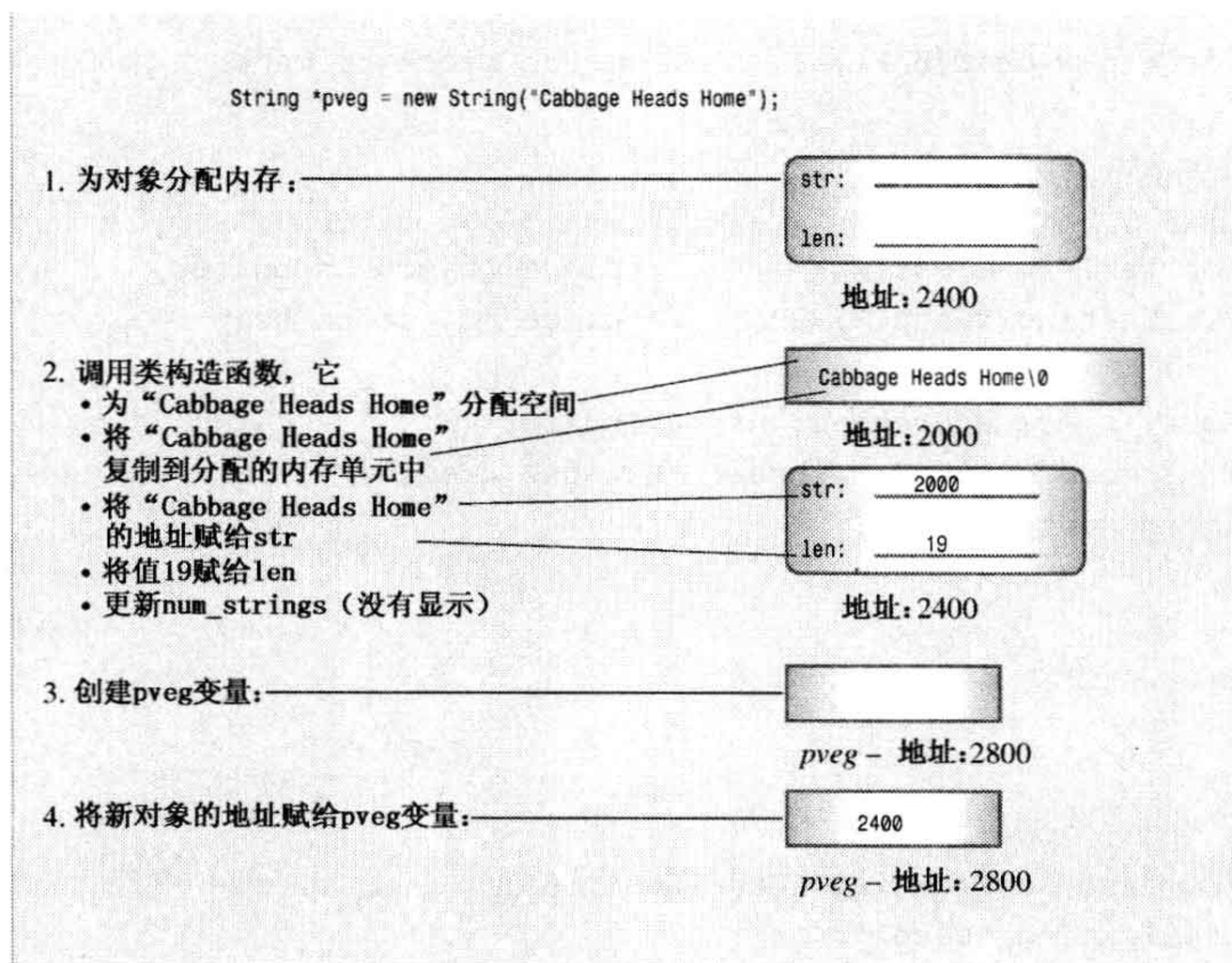


图 12.6　使用 new 创建对象

- 可以使用->运算符通过指针访问类方法：

```
if (sayings[i].length() < shortest->length())
```

- 可以对对象指针应用解除引用运算符（*）来获得对象：

```
if (sayings[i] < *first)     // compare object values
     first = &sayings[i];    // assign object address
```

12.5.3 再谈定位 new 运算符

本书前面介绍过，定位 new 运算符让您能够在分配内存时能够指定内存位置。第 9 章从内置类型的角度讨论了定位 new 运算符，将这种运算符用于对象时情况有些不同，程序清单 12.8 使用了定位 new 运算符和常规 new 运算符给对象分配内存，其中定义的类的构造函数和析构函数都会显示一些信息，让用户能够了解对象的历史。

程序清单 12.8 placenew1.cpp

```
// placenew1.cpp  -- new, placement new, no delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;
class JustTesting
{
private:
    string words;
    int number;
public:
    JustTesting(const string & s = "Just Testing", int n = 0)
    {words = s; number = n; cout << words << " constructed\n"; }
    ~JustTesting() { cout << words << " destroyed\n";}
    void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
    char * buffer = new char[BUF];      // get a block of memory

    JustTesting *pc1, *pc2;

    pc1 = new (buffer) JustTesting;      // place object in buffer
    pc2 = new JustTesting("Heap1", 20);  // place object on heap

    cout << "Memory block addresses:\n" << "buffer: "
        << (void *) buffer << "    heap: " << pc2 <<endl;
    cout << "Memory contents:\n";
    cout << pc1 << ": ";
    pc1->Show();
    cout << pc2 << ": ";
    pc2->Show();

    JustTesting *pc3, *pc4;
    pc3 = new (buffer) JustTesting("Bad Idea", 6);
    pc4 = new JustTesting("Heap2", 10);

    cout << "Memory contents:\n";
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();

    delete pc2;                          // free Heap1
    delete pc4;                          // free Heap2
```

```
    delete [] buffer;                    // free buffer
    cout << "Done\n";
    return 0;
}
```

该程序使用 new 运算符创建了一个 512 字节的内存缓冲区，然后使用 new 运算符在堆中创建两个 JustTesting 对象，并试图使用定位 new 运算符在内存缓冲区中创建两个 JustTesting 对象。最后，它使用 delete 来释放使用 new 分配的内存。下面是该程序的输出：

```
Just Testing constructed
Heap1 constructed
Memory block addresses:
buffer: 00320AB0    heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Bad Idea constructed
Heap2 constructed
Memory contents:
00320AB0: Bad Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Done
```

和往常一样，内存地址的格式和值将随系统而异。

程序清单 12.8 在使用定位 new 运算符时存在两个问题。首先，在创建第二个对象时，定位 new 运算符使用一个新对象来覆盖用于第一个对象的内存单元。显然，如果类动态地为其成员分配内存，这将引发问题。

其次，将 delete 用于 pc2 和 pc4 时，将自动调用为 pc2 和 pc4 指向的对象调用析构函数；然而，将 delete[] 用于 buffer 时，不会为使用定位 new 运算符创建的对象调用析构函数。

这里的经验教训与第 9 章介绍的相同：程序员必须负责管用定位 new 运算符用从中使用的缓冲区内存单元。要使用不同的内存单元，程序员需要提供两个位于缓冲区的不同地址，并确保这两个内存单元不重叠。例如，可以这样做：

```
pc1 = new (buffer) JustTesting;
pc3 = new (buffer + sizeof (JustTesting)) JustTesting("Better Idea", 6);
```

其中指针 pc3 相对于 pc1 的偏移量为 JustTesting 对象的大小。

第二个教训是，如果使用定位 new 运算符来为对象分配内存，必须确保其析构函数被调用。但如何确保呢？对于在堆中创建的对象，可以这样做：

```
delete pc2;   // delete object pointed to by pc2
```

但不能像下面这样做：

```
delete pc1;   // delete object pointed to by pc1? NO!
delete pc3;   // delete object pointed to by pc3? NO!
```

原因在于 delete 可与常规 new 运算符配合使用，但不能与定位 new 运算符配合使用。例如，指针 pc3 没有收到 new 运算符返回的地址，因此 delete pc3 将导致运行阶段错误。在另一方面，指针 pc1 指向的地址与 buffer 相同，但 buffer 是使用 new []初始化的，因此必须使用 delete [ ]而不是 delete 来释放。即使 buffer 是使用 new 而不是 new []初始化的，delete pc1 也将释放 buffer，而不是 pc1。这是因为 new/delete 系统知道已分配的 512 字节块 buffer，但对定位 new 运算符对该内存块做了何种处理一无所知。

该程序确实释放了 buffer：

```
delete [] buffer;                    // free buffer
```

正如上述注释指出的，delete [] buffer;释放使用常规 new 运算符分配的整个内存块，但它没有为定位 new 运算符在该内存块中创建的对象调用析构函数。您之所以知道这一点，是因为该程序使用了一个显示信息的析构函数，该析构函数宣布了“Heap1”和“Heap2”的死亡，但却没有宣布“Just Testing”和“Bad Idea”的死亡。

这种问题的解决方案是，显式地为使用定位 new 运算符创建的对象调用析构函数。正常情况下将自动调用析构函数，这是需要显式调用析构函数的少数几种情形之一。显式地调用析构函数时，必须指定要销毁的对象。由于有指向对象的指针，因此可以使用这些指针：

```
pc3->~JustTesting();  // destroy object pointed to by pc3
pc1->~JustTesting();  // destroy object pointed to by pc1
```

程序清单 12.9 对定位 new 运算符使用的内存单元进行管理，加入到合适的 delete 和显式析构函数调用，从而修复了程序清单 12.8 中的问题。需要注意的一点是正确的删除顺序。对于使用定位 new 运算符创建的对象，应以与创建顺序相反的顺序进行删除。原因在于，晚创建的对象可能依赖于早创建的对象。另外，仅当所有对象都被销毁后，才能释放用于存储这些对象的缓冲区。

程序清单 12.9　placenew2.cpp

```
// placenew2.cpp  -- new, placement new, no delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
    string words;
    int number;
public:
    JustTesting(const string & s = "Just Testing", int n = 0)
    {words = s; number = n; cout << words << " constructed\n"; }
    ~JustTesting() { cout << words << " destroyed\n";}
    void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
    char * buffer = new char[BUF];       // get a block of memory

    JustTesting *pc1, *pc2;

    pc1 = new (buffer) JustTesting;      // place object in buffer
    pc2 = new JustTesting("Heap1", 20);  // place object on heap

    cout << "Memory block addresses:\n" << "buffer: "
        << (void *) buffer << "    heap: " << pc2 <<endl;
    cout << "Memory contents:\n";
    cout << pc1 << ": ";
    pc1->Show();
    cout << pc2 << ": ";
    pc2->Show();

    JustTesting *pc3, *pc4;
// fix placement new location
    pc3 = new (buffer + sizeof (JustTesting))
```

```
            JustTesting("Better Idea", 6);
    pc4 = new JustTesting("Heap2", 10);

    cout << "Memory contents:\n";
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();

    delete pc2;             // free Heap1
    delete pc4;             // free Heap2
// explicitly destroy placement new objects
    pc3->~JustTesting();  // destroy object pointed to by pc3
    pc1->~JustTesting();  // destroy object pointed to by pc1
    delete [] buffer;     // free buffer
    cout << "Done\n";
    return 0;
}
```

该程序的输出如下：

```
Just Testing constructed
Heap1 constructed
Memory block addresses:
buffer: 00320AB0    heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Better Idea constructed
Heap2 constructed
Memory contents:
00320AD0: Better Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Better Idea destroyed
Just Testing destroyed
Done
```

该程序使用定位 new 运算符在相邻的内存单元中创建两个对象，并调用了合适的析构函数。

12.6　复习各种技术

至此，介绍了多种用于处理各种与类相关的问题的编程技术。可能难以掌握这些技术，下面对它们进行总结，并介绍何时使用它们。

12.6.1　重载<<运算符

要重新定义 << 运算符，以便将它和 cout 一起用来显示对象的内容，请定义下面的友元运算符函数：

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

其中 c_name 是类名。如果该类提供了能够返回所需内容的公有方法，则可在运算符函数中使用这些方法，这样便不用将它们设置为友元函数了。

12.6.2 转换函数

要将单个值转换为类类型，需要创建原型如下所示的类构造函数：

```
c_name(type_name value);
```

其中 c_name 为类名，type_name 是要转换的类型的名称。

要将类转换为其他类型，需要创建原型如下所示的类成员函数：

```
operator type_name();
```

虽然该函数没有声明返回类型，但应返回所需类型的值。

使用转换函数时要小心。可以在声明构造函数时使用关键字 explicit，以防止它被用于隐式转换。

12.6.3 其构造函数使用 new 的类

如果类使用 new 运算符来分配类成员指向的内存，在设计时应采取一些预防措施（前面总结了这些预防措施，应牢记这些规则，这是因为编译器并不知道这些规则，因此无法发现错误）。

- 对于指向的内存是由 new 分配的所有类成员，都应在类的析构函数中对其使用 delete，该运算符将释放分配的内存。
- 如果析构函数通过对指针类成员使用 delete 来释放内存，则每个构造函数都应当使用 new 来初始化指针，或将它设置为空指针。
- 构造函数中要么使用 new []，要么使用 new，而不能混用。如果构造函数使用的是 new[]，则析构函数应使用 delete []；如果构造函数使用的是 new，则析构函数应使用 delete。
- 应定义一个分配内存（而不是将指针指向已有内存）的复制构造函数。这样程序将能够将类对象初始化为另一个类对象。这种构造函数的原型通常如下：

```
className(const className &)
```

- 应定义一个重载赋值运算符的类成员函数，其函数定义如下（其中 c_pointer 是 c_name 的类成员，类型为指向 type_name 的指针）。下面的示例假设使用 new []来初始化变量 c_pointer：

```
c_name & c_name::operator=(const c_name & cn)
{
    if (this == & cn)
        return *this;     // done if self-assignment
    delete [] c pointer;
    // set size number of type_name units to be copied
    c_pointer = new type_name[size];
    // then copy data pointed to by cn.c_pointer to
    // location pointed to by c_pointer
    ...
    return *this;
}
```

12.7 队列模拟

进一步了解类后，可将这方面的知识用于解决编程问题。Heather 银行打算在 Food Heap 超市开设一个自动柜员机（ATM）。Food Heap 超市的管理者担心排队等待使用 ATM 的人流会干扰超市的交通，希望限制排队等待的人数。Heather 银行希望对顾客排队等待的时间进行估测。要编写一个程序来模拟这种情况，让超市的管理者可以了解 ATM 可能造成的影响。

对于这种问题，最自然的方法是使用顾客队列。队列是一种抽象的数据类型（Abstract Data Type，ADT），可以存储有序的项目序列。新项目被添加在队尾，并可以删除队首的项目。队列有点像栈，但栈在同一端进行

添加和删除。这使得栈是一种后进先出（LIFO，last-in，first-out）的结构，而队列是先进先出（FIFO，first-in，first-out）的。从概念上说，队列就好比是收款台或 ATM 前面排的队，所以对于上述问题，队列非常合适。因此，工程的任务之一是定义一个 Queue 类（第 16 章将介绍标准模板库类 queue，也将介绍如何开发自己的类）。

队列中的项目是顾客。Heather 银行的代表介绍：通常，三分之一的顾客只需要一分钟便可获得服务，三分之一的顾客需要两分钟，另外三分之一的顾客需要三分钟。另外，顾客到达的时间是随机的，但每个小时使用自动柜员机的顾客数量相当稳定。工程的另外两项任务是：设计一个表示顾客的类；编写一个程序来模拟顾客和队列之间的交互（参见图 12.7）。

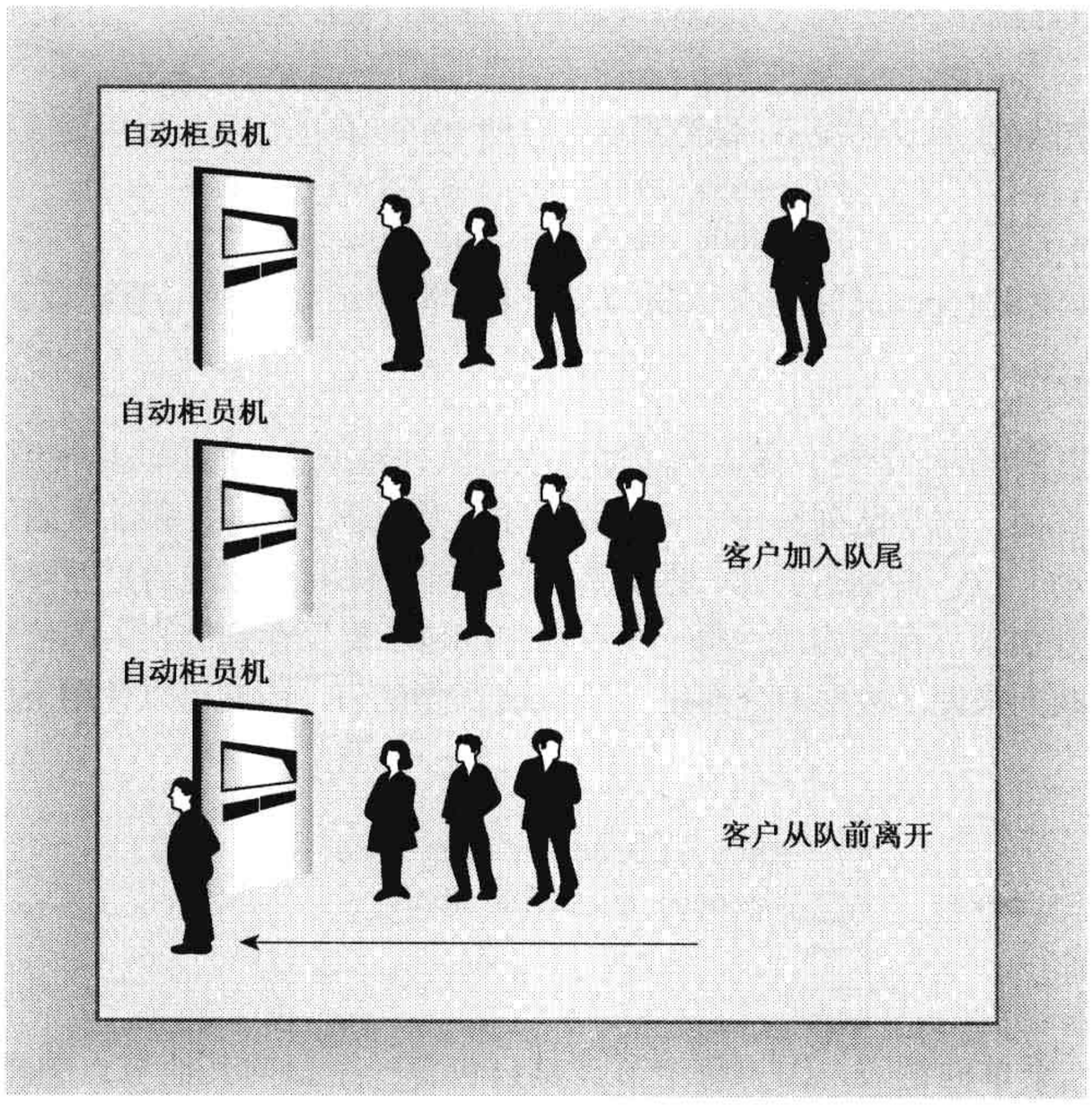


图 12.7　队列

12.7.1　队列类

首先需要设计一个 Queue 类。这里先列出队列的特征：

- 队列存储有序的项目序列；
- 队列所能容纳的项目数有一定的限制；
- 应当能够创建空队列；
- 应当能够检查队列是否为空；
- 应当能够检查队列是否是满的；
- 应当能够在队尾添加项目；
- 应当能够从队首删除项目；
- 应当能够确定队列中项目数。

设计类时，需要开发公有接口和私有实现。

1. Queue 类的接口

从队列的特征可知，Queue 类的公有接口应该如下：

```
class Queue
{
    enum {Q_SIZE = 10};
```

```
private:
// private representation to be developed later
public:
    Queue(int qs = Q_SIZE); // create queue with a qs limit
    ~Queue();
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item); // add item to end
    bool dequeue(Item &item);       // remove item from front
};
```

构造函数创建一个空队列。默认情况下，队列最多可存储 10 个项目，但是可以用显式初始化参数覆盖该默认值：

```
Queue line1;                // queue with 10-item limit
Queue line2(20);            // queue with 20-item limit
```

使用队列时，可以使用 typedef 来定义 Item（第 14 章将介绍如何使用类模板）。

2. Queue 类的实现

确定接口后，便可以实现它。首先，需要确定如何表示队列数据。一种方法是使用 new 动态分配一个数组，它包含所需的元素数。然而，对于队列操作而言，数组并不太合适。例如，删除数组的第一个元素后，需要将余下的所有元素向前移动一位；否则需要作一些更费力的工作，如将数组视为是循环的。然而，链表能够很好地满足队列的要求。链表由节点序列构成。每一个节点中都包含要保存到链表中的信息以及一个指向下一个节点的指针。对于这里的队列来说，数据部分都是一个 Item 类型的值，因此可以使用下面的结构来表示节点：

```
struct Node
{
    Item item;              // data stored in the node
    struct Node * next;     // pointer to next node
};
```

图 12.8 说明了链表。

如图 12.8 所示是一个单向链表，因为每个节点都只包含一个指向其他节点的指针。知道第一个节点的地址后，就可以沿指针找到后面的每一个节点。通常，链表最后一个节点中的指针被设置为 NULL（或 0），以指出后面没有节点了。在 C++11 中，应使用新增的关键字 nullptr。要跟踪链表，必须知道第一个节点的地址。可以让 Queue 类的一个数据成员指向链表的起始位置。具体地说，这是所需要的全部信息，有了这种信息后，就可以沿节点链找到任何节点。然而，由于队列总是将新项目添加到队尾，因此包含一个指向最后一个节点的数据成员将非常方便（参见图 12.9）。此外，还可以使用数据成员来跟踪队列可存储的最大项目数以及当前的项目数。所以，类声明的私有部分与下面类似：

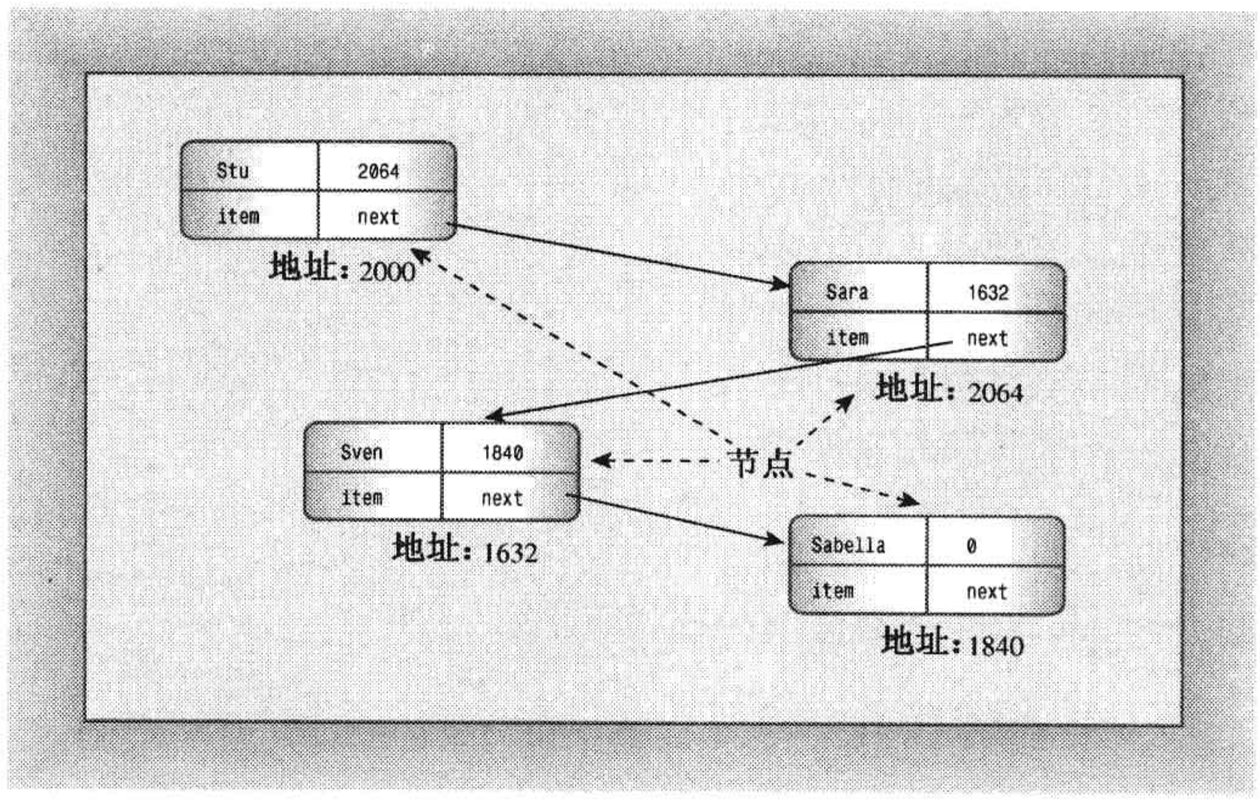


图 12.8　链表

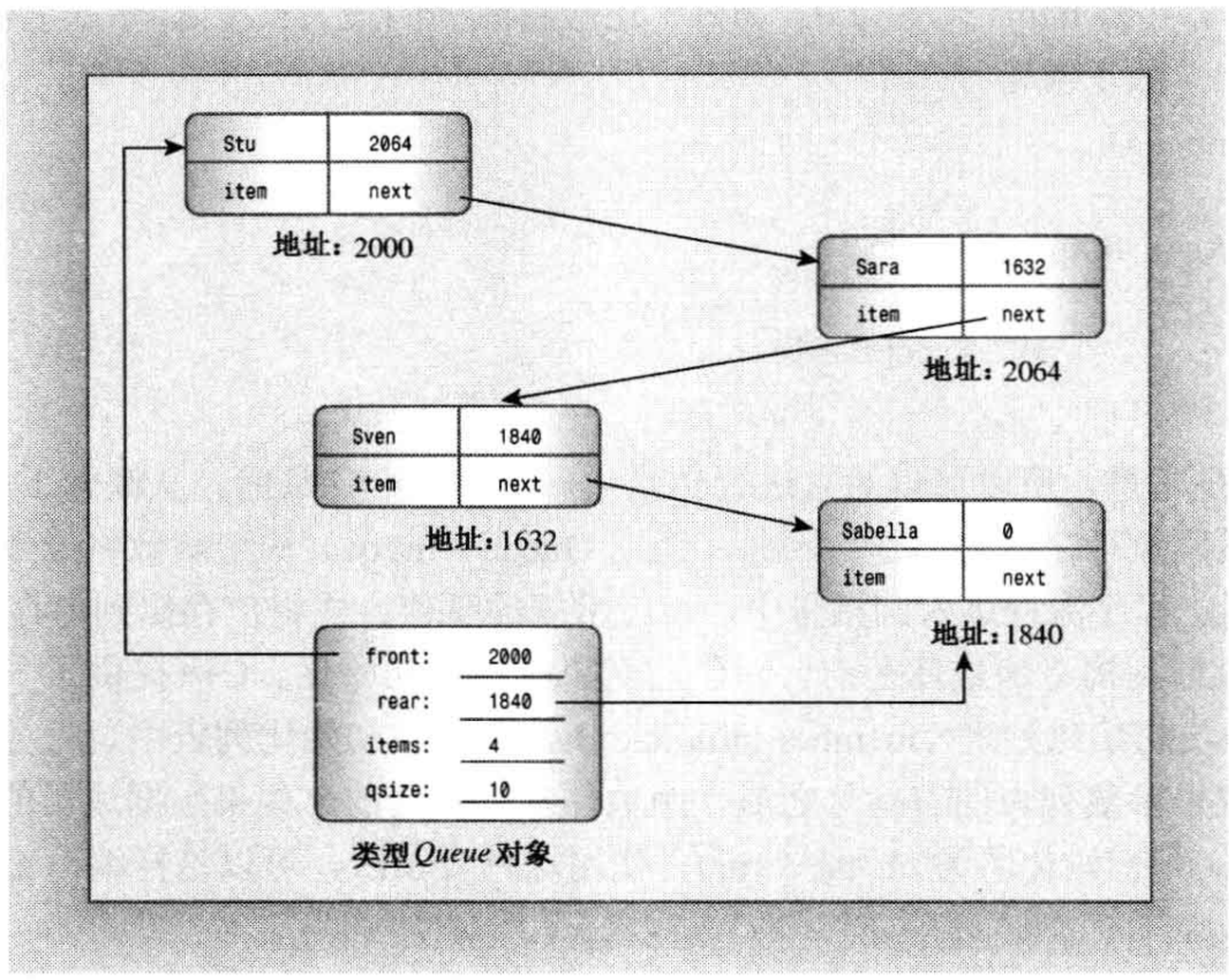


图 12.9　Queue 对象

```
class Queue
{
private:
// class scope definitions
    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};
// private class members
    Node * front;        // pointer to front of Queue
    Node * rear;         // pointer to rear of Queue
    int items;           // current number of items in Queue
    const int qsize;     // maximum number of items in Queue
    ...
public:
//...
};
```

上述声明使用了 C++的一项特性：在类中嵌套结构或类声明。通过将 Node 声明放在 Queue 类中，可以使其作用域为整个类。也就是说，Node 是这样一种类型：可以使用它来声明类成员，也可以将它作为类方法中的类型名称，但只能在类中使用。这样，就不必担心该 Node 声明与某些全局声明或其他类中声明的 Node 发生冲突。有些较老的编译器不支持嵌套的结构和类，如果您的编译器是这样的，则必须将 Node 结构定义为全局的，将其作用域设置为整个文件。

嵌套结构和类

在类声明中声明的结构、类或枚举被称为是被嵌套在类中，其作用域为整个类。这种声明不会创建数据对象，而只是指定了可以在类中使用的类型。如果声明是在类的私有部分进行的，则只能在这个类使用被声明的类型；如果声明是在公有部分进行的，则可以从类的外部通过作用域解析运算符使用被声明的类型。例如，如果 Node 是在 Queue 类的公有部分声明的，则可以在类的外面声明 Queue::Node 类型的变量。

设计好数据的表示方式后，接下来需要编写类方法。

3. 类方法

类构造函数应提供类成员的值。由于在这个例子中，队列最初是空的，因此队首和队尾指针都设置为

NULL（0 或 nullptr），并将 items 设置为 0。另外，还应将队列的最大长度 qsize 设置为构造函数参数 qs 的值。下面的实现方法无法正常运行：

```
Queue::Queue(int qs)
{
    front = rear = NULL;
    items = 0;
    qsize = qs;     // not acceptable!
}
```

问题在于 qsize 是常量，所以可以对它进行初始化，但不能给它赋值。从概念上说，调用构造函数时，对象将在括号中的代码执行之前被创建。因此，调用 Queue（int qs）构造函数将导致程序首先给 4 个成员变量分配内存。然后，程序流程进入到括号中，使用常规的赋值方式将值存储到内存中。因此，对于 const 数据成员，必须在执行到构造函数体之前，即创建对象时进行初始化。C++提供了一种特殊的语法来完成上述工作，它叫做成员初始化列表（member initializer list）。成员初始化列表由逗号分隔的初始化列表组成（前面带冒号）。它位于参数列表的右括号之后、函数体左括号之前。如果数据成员的名称为 mdata，并需要将它初始化为 val，则初始化器为 mdata（val）。使用这种表示法，可以这样编写 Queue 的构造函数：

```
Queue::Queue(int qs) : qsize(qs)   // initialize qsize to qs
{
    front = rear = NULL;
    items = 0;
}
```

通常，初值可以是常量或构造函数的参数列表中的参数。这种方法并不限于初始化常量，可以将 Queue 构造函数写成如下所示：

```
Queue::Queue(int qs) : qsize(qs), front(NULL), rear(NULL), items(0)
{
}
```

只有构造函数可以使用这种初始化列表语法。如上所示，对于 const 类成员，必须使用这种语法。另外，对于被声明为引用的类成员，也必须使用这种语法：

```
class Agency {...};
class Agent
{
private:
    Agency & belong;    // must use initializer list to initialize
    ...
};
Agent::Agent(Agency & a) : belong(a) {...}
```

这是因为引用与 const 数据类似，只能在被创建时进行初始化。对于简单数据成员（例如 front 和 items），使用成员初始化列表和在函数体中使用赋值没有什么区别。然而，正如第 14 章将介绍的，对于本身就是类对象的成员来说，使用成员初始化列表的效率更高。

成员初始化列表的语法

如果 Classy 是一个类，而 mem1、mem2 和 mem3 都是这个类的数据成员，则类构造函数可以使用如下的语法来初始化数据成员：

```
Classy::Classy(int n, int m) :mem1(n), mem2(0), mem3(n*m + 2)
{
//...
}
```

上述代码将 mem1 初始化为 n，将 mem2 初始化为 0，将 mem3 初始化为 n*m + 2。从概念上说，这些

初始化工作是在对象创建时完成的，此时还未执行括号中的任何代码。请注意以下几点：

- 这种格式只能用于构造函数；
- 必须用这种格式来初始化非静态 const 数据成员（至少在 C++11 之前是这样的）；
- 必须用这种格式来初始化引用数据成员。

数据成员被初始化的顺序与它们出现在类声明中的顺序相同，与初始化器中的排列顺序无关。

警告：不能将成员初始化列表语法用于构造函数之外的其他类方法。

成员初始化列表使用的括号方式也可用于常规初始化。也就是说，如果愿意，可以将下述代码：

```
int games = 162;
double talk = 2.71828;
```

替换为：

```
int games(162);
double talk(2.71828);
```

这使得初始化内置类型就像初始化类对象一样。

C++11 的类内初始化

C++11 允许您以更直观的方式进行初始化：

```
class Classy
{
    int mem1 = 10;        // in-class initialization
    const int mem2 = 20; // in-class initialization
//...
};
```

这与在构造函数中使用成员初始化列表等价：

```
Classy::Classy() : mem1(10), mem2(20) {...}
```

成员 mem1 和 mem2 将分别被初始化为 10 和 20，除非调用了使用成员初始化列表的构造函数，在这种情况下，实际列表将覆盖这些默认初始值：

```
Classy::Classy(int n) : mem1(n) {...}
```

在这里，构造函数将使用 n 来初始化 mem1，但 mem2 仍被设置为 20。

isempty()、isfull()和 queuecount()的代码都非常简单。如果 items 为 0，则队列是空的；如果 items 等于 qsize，则队列是满的。要知道队列中的项目数，只需返回 items 的值。后面的程序清单 12.11 列出了这些代码。

将项目添加到队尾（入队）比较麻烦。下面是一种方法：

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;  // create node
// on failure, new throws std::bad_alloc exception
    add->item = item;       // set node pointers
    add->next = NULL;       // or nullptr;
    items++;
    if (front == NULL)      // if queue is empty,
        front = add;        // place item at front
    else
        rear->next = add;   // else place at rear
    rear = add;             // have rear point to new node
    return true;
}
```

总之，方法需要经过下面几个阶段（见图 12.10）。

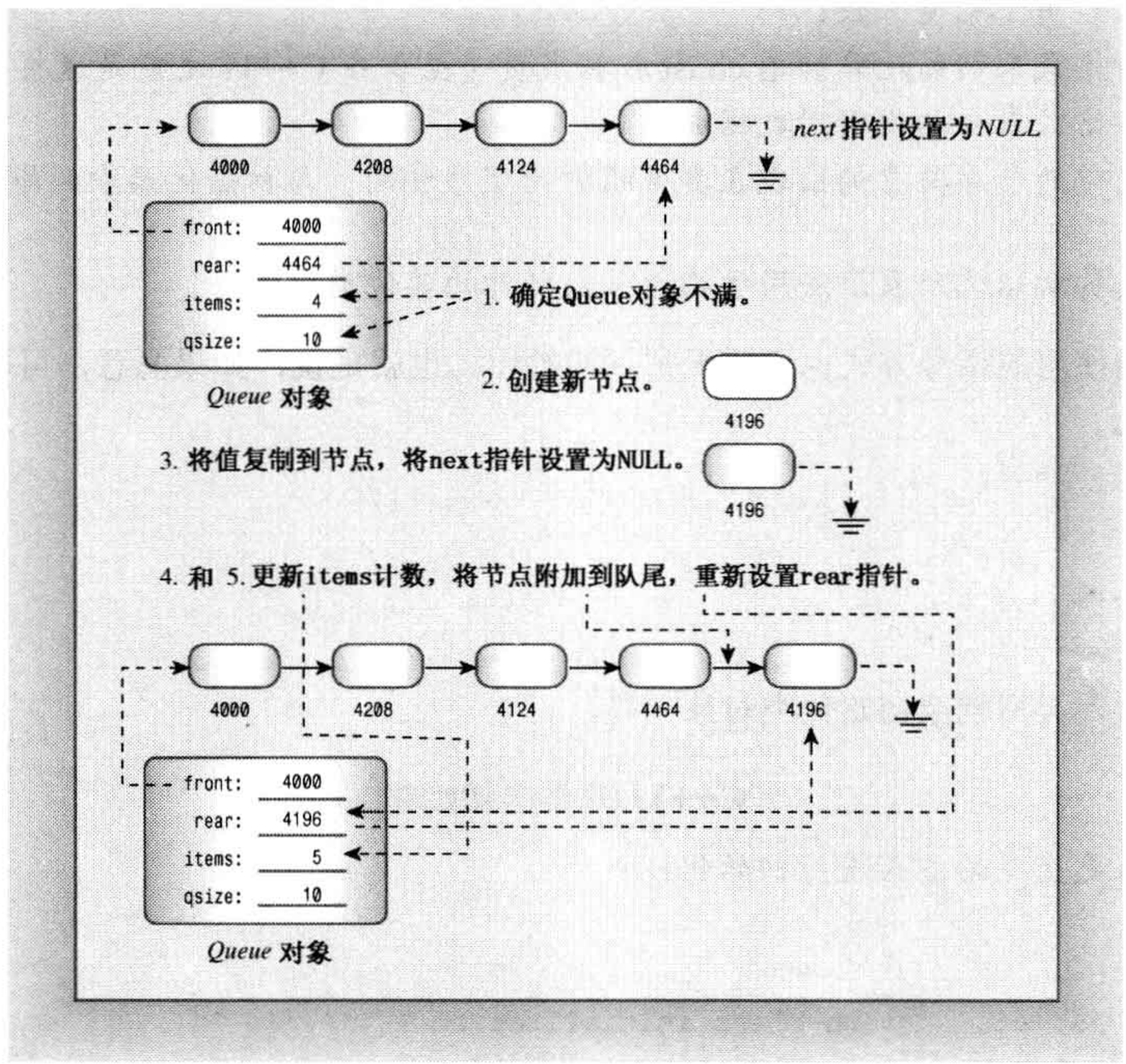


图 12.10　将项目入队

1．如果队列已满，则结束（在这里的实现中，队列的最大长度由用户通过构造函数指定）。

2．创建一个新节点。如果 new 无法创建新节点，它将引发异常，这个主题将在第 15 章介绍。最终的结果是，除非提供了处理异常的代码，否则程序将终止。

3．在节点中放入正确的值。在这个例子中，代码将 Item 值复制到节点的数据部分，并将节点的 next 指针设置为 NULL（0 或 C++11 新增的 nullptr）。这样就为将节点作为队列中的最后一个项目做好了准备。

4．将项目计数（items）加 1。

5．将节点附加到队尾。这包括两个部分。首先，将节点与列表中的另一个节点连接起来。这是通过将当前队尾节点的 next 指针指向新的队尾节点来完成的。第二部分是将 Queue 的成员指针 rear 设置为指向新节点，使队列可以直接访问最后一个节点。如果队列为空，则还必须将 front 指针设置成指向新节点（如果只有一个节点，则它既是队首节点，也是队尾节点）。

删除队首项目（出队）也需要多个步骤才能完成。下面是一种方式：

```
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;     // set item to first item in queue
    items--;
    Node * temp = front;    // save location of first item
    front = front->next;    // reset front to next item
    delete temp;            // delete former first item
    if (items == 0)
        rear = NULL;
    return true;
}
```

总之，需要经过下面几个阶段（参见图 12.11）：

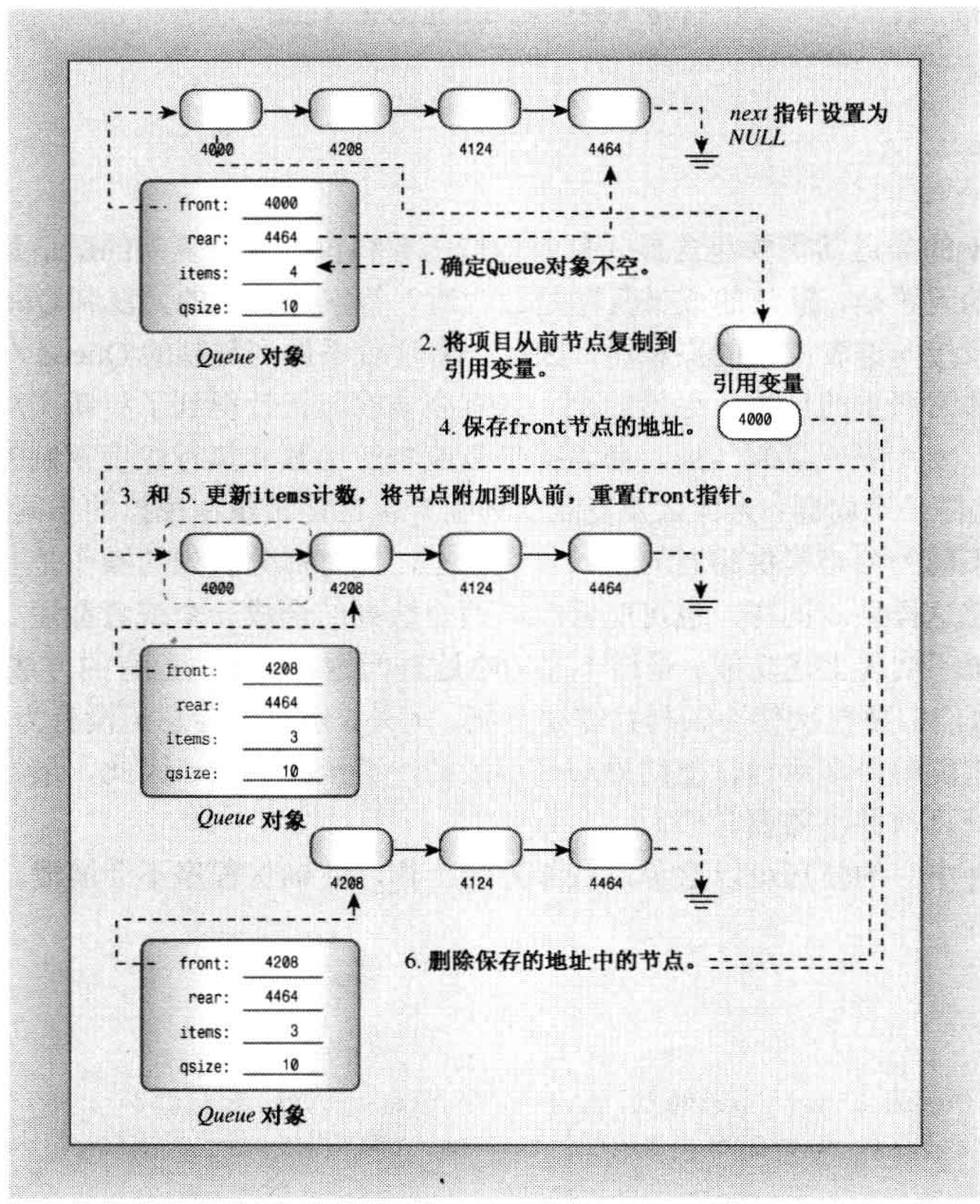


图 12.11　将项目出队

1．如果队列为空，则结束。

2．将队列的第一个项目提供给调用函数，这是通过将当前 front 节点中的数据部分复制到传递给方法的引用变量中来实现。

3．将项目计数（items）减 1。

4．保存 front 节点的位置，供以后删除。

5．让节点出队。这是通过将 Queue 成员指针 front 设置成指向下一个节点来完成的，该节点的位置由 front->next 提供。

6．为节省内存，删除以前的第一个节点。

7．如果链表为空，则将 rear 设置为 NULL（在这个例子中，将 front 指针设置成 front->next 后，它已经是 NULL 了）。同样，可使用 0 而不是 NULL，也可使用 C++11 新增的 nullptr。

第 4 步是必不可少的，这是因为第 5 步将删除关于先前第一个节点位置的信息。

4．是否需要其他类方法

是否需要其他方法呢？类构造函数没有使用 new，所以乍一看，好像不用理会由于在构造函数中使用 new 给类带来的特殊要求。当然，这种印象是错误的，因为向队列中添加对象将调用 new 来创建新的节点。通过删除节点的方式，dequeue( )方法确实可以清除节点，但这并不能保证队列在到期时为空。因此，类需要一个显式析构函数——该函数删除剩余的所有节点。下面是一种实现，它从链表头开始，依次删除其中的每个节点：

```
Queue::~Queue()
{
    Node * temp;
    while (front != NULL)   // while queue is not yet empty
    {
```

```
        temp = front;        // save address of front item
        front = front->next;// reset pointer to next item
        delete temp;         // delete former front
    }
}
```

您知道，使用 new 的类通常需要包含显式复制构造函数和执行深度复制的赋值运算符，这个例子也是如此吗？首先要回答的问题是，默认的成员复制是否合适？答案是否定的。复制 Queue 对象的成员将生成一个新的对象，该对象指向链表原来的头和尾。因此，将项目添加到复制的 Queue 对象中，将修改共享的链表。这样做将造成非常严重的后果。更糟的是，只有副本的尾指针得到了更新，从原始对象的角度看，这将损坏链表。显然，要克隆或复制队列，必须提供复制构造函数和执行深度复制的赋值构造函数。

当然，这提出了这样一个问题：为什么要复制队列呢？也许是希望在模拟的不同阶段保存队列的瞬像，也可能是希望为两个不同的策略提供相同的输入。实际上，拥有拆分队列的操作是非常有用的，超市在开设额外的收款台时经常这样做。同样，也可能希望将两个队列结合成一个或者截短一个队列。

但假设这里的模拟不实现上述功能。难道不能忽略这些问题，而使用已有的方法吗？当然可以。然而，在将来的某个时候，可能需要再次使用队列且需要复制。另外，您可能会忘记没有为复制提供适当的代码。在这种情况下，程序将能编译和运行，但结果却是混乱的，甚至会崩溃。因此，最好还是提供复制构造函数和赋值运算符，尽管目前并不需要它们。

幸运的是，有一种小小的技巧可以避免这些额外的工作，并确保程序不会崩溃。这就是将所需的方法定义为伪私有方法：

```
class Queue
{
private:
    Queue(const Queue & q) : qsize(0) { }   // preemptive definition
    Queue & operator=(const Queue & q) { return *this;}
//...
};
```

这样做有两个作用：第一，它避免了本来将自动生成的默认方法定义。第二，因为这些方法是私有的，所以不能被广泛使用。也就是说，如果 nip 和 tuck 是 Queue 对象，则编译器就不允许这样做：

```
Queue snick(nip);      // not allowed
tuck = nip;            // not allowed
```

所以，与其将来面对无法预料的运行故障，不如得到一个易于跟踪的编译错误，指出这些方法是不可访问的。另外，在定义其对象不允许被复制的类时，这种方法也很有用。

C++11 提供了另一种禁用方法的方式——使用关键字 delete，这将在第 18 章介绍。

还有没有其他影响需要注意呢？当然有。当对象被按值传递（或返回）时，复制构造函数将被调用。然而，如果遵循优先采用按引用传递对象的惯例，将不会有任何问题。另外，复制构造函数还被用于创建其他的临时对象，但 Queue 定义中并没有导致创建临时对象的操作，例如重载加法运算符。

12.7.2 Customer 类

接下来需要设计客户类。通常，ATM 客户有很多属性，例如姓名、账户和账户结余。然而，这里的模拟需要使用的唯一一个属性是客户何时进入队列以及客户交易所需的时间。当模拟生成新客户时，程序将创建一个新的客户对象，并在其中存储客户的到达时间以及一个随机生成的交易时间。当客户到达队首时，程序将记录此时的时间，并将其与进入队列的时间相减，得到客户的等候时间。下面的代码演示了如何定义和实现 Customer 类：

```
class Customer
{
private:
    long arrive;         // arrival time for customer
    int processtime;     // processing time for customer
```

```
public:
    Customer() { arrive = processtime = 0; }
    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};
void Customer::set(long when)
{
    processtime = std::rand() % 3 + 1;
    arrive = when;
}
```

默认构造函数创建一个空客户。set()成员函数将到达时间设置为参数，并将处理时间设置为 1～3 中的一个随机值。

程序清单 12.10 将 Queue 和 Customer 类声明放到一起，而程序清单 12.11 列出了方法。

程序清单 12.10　queue.h

```
// queue.h -- interface for a queue
#ifndef QUEUE_H_
#define QUEUE_H_
// This queue will contain Customer items
class Customer
{
private:
    long arrive;        // arrival time for customer
    int processtime;    // processing time for customer
public:
    Customer() { arrive = processtime = 0; }

    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};

typedef Customer Item;

class Queue
{
private:
// class scope definitions
    // Node is a nested structure definition local to this c
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};
// private class members
    Node * front;       // pointer to front of Queue
    Node * rear;        // pointer to rear of Queue
    int items;          // current number of items in Queue
    const int qsize;    // maximum number of items in Queue
    // preemptive definitions to prevent public copying
    Queue(const Queue & q) : qsize(0) { }
    Queue & operator=(const Queue & q) { return *this;}
public:
    Queue(int qs = Q_SIZE); // create queue with a qs limit
    ~Queue();
    bool isempty() const;
```

```
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item); // add item to end
    bool dequeue(Item &item);       // remove item from front
};
#endif
```

程序清单 12.11　queue.cpp

```
// queue.cpp -- Queue and Customer methods
#include "queue.h"
#include <cstdlib>          // (or stdlib.h) for rand()

// Queue methods
Queue::Queue(int qs) : qsize(qs)
{
    front = rear = NULL;    // or nullptr
    items = 0;
}

Queue::~Queue()
{
    Node * temp;
    while (front != NULL)   // while queue is not yet empty
    {
        temp = front;       // save address of front item
        front = front->next;// reset pointer to next item
        delete temp;        // delete former front
    }
}

bool Queue::isempty() const
{
    return items == 0;
}
bool Queue::isfull() const
{
    return items == qsize;
}

int Queue::queuecount() const
{
    return items;
}

// Add item to queue
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;  // create node
// on failure, new throws std::bad_alloc exception
    add->item = item;       // set node pointers
    add->next = NULL;       // or nullptr;
```

```
    items++;
    if (front == NULL)        // if queue is empty,
        front = add;          // place item at front
    else
        rear->next = add;     // else place at rear
    rear = add;               // have rear point to new node
    return true;
}

// Place front item into item variable and remove from queue
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;       // set item to first item in queue
    items--;
    Node * temp = front;      // save location of first item
    front = front->next;      // reset front to next item
    delete temp;              // delete former first item
    if (items == 0)
        rear = NULL;
    return true;
}
// time set to a random value in the range 1 - 3
void Customer::set(long when)
{
    processtime = std::rand() % 3 + 1;
    arrive = when;
}
```

12.7.3　ATM 模拟

现在已经拥有模拟 ATM 所需的工具。程序允许用户输入 3 个数：队列的最大长度、程序模拟的持续时间（单位为小时）以及平均每小时的客户数。程序将使用循环——每次循环代表一分钟。在每分钟的循环中，程序将完成下面的工作。

1．判断是否来了新的客户。如果来了，并且此时队列未满，则将它添加到队列中，否则拒绝客户入队。

2．如果没有客户在进行交易，则选取队列的第一个客户。确定该客户的已等候时间，并将 wait_time 计数器设置为新客户所需的处理时间。

3．如果客户正在处理中，则将 wait_time 计数器减 1。

4．记录各种数据，如获得服务的客户数目、被拒绝的客户数目、排队等候的累积时间以及累积的队列长度等。

当模拟循环结束时，程序将报告各种统计结果。

一个有趣的问题是，程序如何确定是否有新的客户到来。假设平均每小时有 10 名客户到达，则相当于每 6 分钟有一名客户。程序将计算这个值，并将它保存在 min_per_cust 变量中。然而，刚好每 6 分钟来一名客户不太现实，我们真正（至少在大部分时间内）希望的是一个更随机的过程——但平均每 6 分钟来一名客户。程序将使用下面的函数来确定是否在循环期间有客户到来：

```
bool newcustomer(double x)
{
    return (std::rand() * x / RAND_MAX < 1);
}
```

其工作原理如下：值 RAND_MAX 是在 cstdlib 文件（以前是 stdlib.h）中定义的，是 rand( )函数可能返

回的最大值（0 是最小值）。假设客户到达的平均间隔时间 x 为 6，则 rand( )* x /RAND_MAX 的值将位于 0 到 6 之间。具体地说，平均每隔 6 次，这个值会有 1 次小于 1。然而，这个函数可能会导致客户到达的时间间隔有时为 1 分钟，有时为 20 分钟。这种方法虽然很笨拙，但可使实际情况不同于有规则地每 6 分钟到来一个客户。如果客户到达的平均时间间隔少于 1 分钟，则上述方法将无效，但模拟并不是针对这种情况设计的。如果确实需要处理这种情况，最好提高时间分辨率，比如每次循环代表 10 秒钟。

程序清单 12.12 给出了模拟的细节。长时间运行该模拟程序，可以知道长期的平均值；短时间运行该模拟程序，将只能知道短期的变化。

程序清单 12.12　bank.cpp

```
// bank.cpp -- using the Queue interface
// compile with queue.cpp
#include <iostream>
#include <cstdlib> // for rand() and srand()
#include <ctime>   // for time()
#include "queue.h"
const int MIN_PER_HR = 60;

bool newcustomer(double x); // is there a new customer?

int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
    using std::ios_base;
// setting things up
    std::srand(std::time(0));    //  random initializing of rand()

    cout << "Case Study: Bank of Heather Automatic Teller\n";
    cout << "Enter maximum size of queue: ";
    int qs;
    cin >> qs;
    Queue line(qs);          // line queue holds up to qs people

    cout << "Enter the number of simulation hours: ";
    int hours;               //  hours of simulation
    cin >> hours;
    // simulation will run 1 cycle per minute
    long cyclelimit = MIN_PER_HR * hours; // # of cycles

    cout << "Enter the average number of customers per hour: ";
    double perhour;          //  average # of arrival per hour
    cin >> perhour;
    double min_per_cust;     //  average time between arrivals
    min_per_cust = MIN_PER_HR / perhour;

    Item temp;               //  new customer data
    long turnaways = 0;      //  turned away by full queue
    long customers = 0;      //  joined the queue
    long served = 0;         //  served during the simulation
    long sum_line = 0;       //  cumulative line length
    int wait_time = 0;       //  time until autoteller is free
    long line_wait = 0;      //  cumulative time in line
```

```
// running the simulation
    for (int cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer(min_per_cust))  // have newcomer
        {
            if (line.isfull())
                turnaways++;
            else
            {
                customers++;
                temp.set(cycle);    // cycle = time of arrival
                line.enqueue(temp); // add newcomer to line
            }
        }
        if (wait_time <= 0 && !line.isempty())
        {
            line.dequeue (temp);      // attend next customer
            wait_time = temp.ptime(); // for wait_time minutes
            line_wait += cycle - temp.when();
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += line.queuecount();
    }

// reporting results
    if (customers > 0)
    {
        cout << "customers accepted: " << customers << endl;
        cout << "  customers served: " << served << endl;
        cout << "         turnaways: " << turnaways << endl;
        cout << "average queue size: ";
        cout.precision(2);
        cout.setf(ios_base::fixed, ios_base::floatfield);
        cout << (double) sum_line / cyclelimit << endl;
        cout << " average wait time: "
             << (double) line_wait / served << " minutes\n";
    }
    else
        cout << "No customers!\n";
    cout << "Done!\n";

    return 0;
}

//  x = average time, in minutes, between customers
//  return value is true if customer shows up this minute
bool newcustomer(double x)
{
    return (std::rand() * x / RAND_MAX < 1);
}
```

注意：编译器如果没有实现 bool，可以用 int 代替 bool，用 0 代替 false，用 1 代替 true；还可能必须使用 stdlib.h 和 time.h 代替较新的 cstdlib 和 ctime；另外可能必须自己来定义 RAND_MAX。

下面是程序清单 12.10-12.12 组成的程序长时间运行的几个例子：

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 15
customers accepted: 1485
  customers served: 1485
         turnaways: 0
average queue size: 0.15
 average wait time: 0.63 minutes
Done!

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2896
  customers served: 2888
         turnaways: 101
average queue size: 4.64
 average wait time: 9.63 minutes
Done!
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 20
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2943
  customers served: 2943
         turnaways: 93
average queue size: 13.06
 average wait time: 26.63 minutes
Done!
```

注意，每小时到达的客户从 15 名增加到 30 名时，等候时间并不是加倍，而是增加了 15 倍。如果允许队列更长，情况将更糟。然而，模拟没有考虑到这个事实——许多客户由于不愿意排很长的队而离开了。

下面是该程序的另外几个运行示例。从中可知，即使平均每小时到达的客户数不变，也会出现短期变化。

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 114
  customers served: 110
         turnaways: 0
average queue size: 2.15
 average wait time: 4.52 minutes
Done!

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 121
  customers served: 116
         turnaways: 5
average queue size: 5.28
```

```
 average wait time: 10.72 minutes
Done!

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 112
  customers served: 109
         turnaways: 0
average queue size: 2.41
 average wait time: 5.16 minutes
Done!
```

12.8　总结

本章介绍了定义和使用类的许多重要方面。其中的一些方面是非常微妙甚至很难理解的概念。如果其中的某些概念对于您来说过于复杂，也不用害怕——这些问题对于大多数 C++的初学者来说都是很难的。通常，对于诸如复制构造函数等概念，都是在由于忽略它们而遇到了麻烦后逐步理解的。本章介绍的一些内容乍看起来非常难以理解，但是随着经验越来越丰富，对其理解也将越透彻。

在类构造函数中，可以使用 new 为数据分配内存，然后将内存地址赋给类成员。这样，类便可以处理长度不同的字符串，而不用在类设计时提前固定数组的长度。在类构造函数中使用 new，也可能在对象过期时引发问题。如果对象包含成员指针，同时它指向的内存是由 new 分配的，则释放用于保存对象的内存并不会自动释放对象成员指针指向的内存。因此在类构造函数中使用 new 类来分配内存时，应在类析构函数中使用 delete 来释放分配的内存。这样，当对象过期时，将自动释放其指针成员指向的内存。

如果对象包含指向 new 分配的内存的指针成员，则将一个对象初始化为另一个对象，或将一个对象赋给另一个对象时，也会出现问题。在默认情况下，C++逐个对成员进行初始化和赋值，这意味着被初始化或被赋值的对象的成员将与原始对象完全相同。如果原始对象的成员指向一个数据块，则副本成员将指向同一个数据块。当程序最终删除这两个对象时，类的析构函数将试图删除同一个内存数据块两次，这将出错。解决方法是：定义一个特殊的复制构造函数来重新定义初始化，并重载赋值运算符。在上述任何一种情况下，新的定义都将创建指向数据的副本，并使新对象指向这些副本。这样，旧对象和新对象都将引用独立的、相同的数据，而不会重叠。由于同样的原因，必须定义赋值运算符。对于每一种情况，最终目的都是执行深度复制，也就是说，复制实际的数据，而不仅仅是复制指向数据的指针。

对象的存储持续性为自动或外部时，在它不再存在时将自动调用其析构函数。如果使用 new 运算符为对象分配内存，并将其地址赋给一个指针，则当您将 delete 用于该指针时将自动为对象调用析构函数。然而，如果使用定位 new 运算符（而不是常规 new 运算符）为类对象分配内存，则必须负责显式地为该对象调用析构函数，方法是使用指向该对象的指针调用析构函数方法。C++允许在类中包含结构、类和枚举定义。这些嵌套类型的作用域为整个类，这意味着它们被局限于类中，不会与其他地方定义的同名结构、类和枚举发生冲突。

C++为类构造函数提供了一种可用来初始化数据成员的特殊语法。这种语法包括冒号和由逗号分隔的初始化列表，被放在构造函数参数的右括号后，函数体的左括号之前。每一个初始化器都由被初始化的成员的名称和包含初始值的括号组成。从概念上来说，这些初始化操作是在对象创建时进行的，此时函数体中的语句还没有执行。语法如下：

```
queue(int qs) : qsize(qs), items(0), front(NULL), rear(NULL) { }
```

如果数据成员是非静态 const 成员或引用，则必须采用这种格式，但可将 C++11 新增的类内初始化用于非静态 const 成员。

C++11 允许类内初始化，即在类定义中进行初始化：

```
class Queue
{
private:
...
    Node * front = NULL;
    enum {Q_SIZE = 10};
    Node * rear = NULL;
    int items = 0;
    const int qsize = Q_SIZE;
...
};
```

这与使用成员初始化列表等价。然而，使用成员初始化列表的构造函数将覆盖相应的类内初始化。

您可能已经注意到，与简单的 C 结构相比，需要注意的类细节要多得多。作为回报，它们的功能也更强。

12.9 复习题

1．假设 String 类有如下私有成员：

```
class String
{
private:
    char * str;     // points to string allocated by new
    int len;        // holds length of string
//...
};
```

a．下述默认构造函数有什么问题？

```
String::String() {}
```

b．下述构造函数有什么问题？

```
String::String(const char * s)
{
    str = s;
    len = strlen(s);
}
```

c．下述构造函数有什么问题？

```
String::String(const char * s)
{
    strcpy(str, s);
    len = strlen(s);
}
```

2．如果您定义了一个类，其指针成员是使用 new 初始化的，请指出可能出现的 3 个问题以及如何纠正这些问题。

3．如果没有显式提供类方法，编译器将自动生成哪些类方法？请描述这些隐式生成的函数的行为。

4．找出并改正下述类声明中的错误：

```
class nifty
{
// data
    char personality[];
    int talents;
// methods
```

```
    nifty();
    nifty(char * s);
    ostream & operator<<(ostream & os, nifty & n);
}

nifty:nifty()
{
    personality = NULL;
    talents = 0;
}

nifty:nifty(char * s)
{
    personality = new char [strlen(s)];
    personality = s;
    talents = 0;
}
ostream & nifty:operator<<(ostream & os, nifty & n)
{
    os << n;
}
```

5. 对于下面的类声明：

```
class Golfer
{
private:
    char * fullname;        // points to string containing golfer's name
    int games;              // holds number of golf games played
    int * scores;           // points to first element of array of golf scores
public:
    Golfer();
    Golfer(const char * name, int g= 0);
     // creates empty dynamic array of g elements if g > 0
    Golfer(const Golfer & g);
    ~Golfer();
};
```

a. 下列各条语句将调用哪些类方法？

```
Golfer nancy;                       // #1
Golfer lulu ("Little Lulu");        //#2
Golfer roy ("Roy Hobbs", 12);       //#3
Golfer * par = new Golfer;          // #4
Golfer next = lulu;                 // #5
Golfer hazzard = "Weed Thwacker";   // #6
*par = nancy;                       // #7
nancy = "Nancy Putter";             // #8
```

b. 很明显，类需要有另外几个方法才能更有用，但是类需要那些方法才能防止数据被损坏呢？

12.10　编程练习

1. 对于下面的类声明：

```
class Cow {
    char name[20];
    char * hobby;
```

```
    double weight;
public:
    Cow();
    Cow(const char * nm, const char * ho, double wt);
    Cow(const Cow c&);
    ~Cow();
    Cow & operator=(const Cow & c);
    void ShowCow() const;  // display all cow data
};
```

给这个类提供实现，并编写一个使用所有成员函数的小程序。

2．通过完成下面的工作来改进 String 类声明（即将 String1.h 升级为 String2.h）。

a．对+运算符进行重载，使之可将两个字符串合并成 1 个。

b．提供一个 Stringlow( )成员函数，将字符串中所有的字母字符转换为小写（别忘了 cctype 系列字符函数）。

c．提供 String( )成员函数，将字符串中所有字母字符转换成大写。

d．提供一个这样的成员函数，它接受一个 char 参数，返回该字符在字符串中出现的次数。

使用下面的程序来测试您的工作：

```
// pe12_2.cpp
#include <iostream>
using namespace std;
#include "string2.h"
int main()
{
    String s1(" and I am a C++ student.");
    String s2 = "Please enter your name: ";
    String s3;
    cout << s2;                  // overloaded << operator
    cin >> s3;                   // overloaded >> operator
    s2 = "My name is " + s3;     // overloaded =, + operators
    cout << s2 << ".\n";
    s2 = s2 + s1;
    s2.stringup();               // converts string to uppercase
    cout << "The string\n" << s2 << "\ncontains " << s2.has('A')
         << " 'A' characters in it.\n";
    s1 = "red";      // String(const char *),
                     // then String & operator=(const String&)
    String rgb[3] = { String(s1), String("green"), String("blue")};
    cout << "Enter the name of a primary color for mixing light: ";
    String ans;
    bool success = false;
    while (cin >> ans)
    {
        ans.stringlow();         // converts string to lowercase
        for (int i = 0; i < 3; i++)
        {
            if (ans == rgb[i])  // overloaded == operator
            {
                cout << "That's right!\n";
                success = true;
                break;
            }
        }
        if (success)
            break;
        else
```

```
            cout << "Try again!\n";
    }
    cout << "Bye\n";
    return 0;
}
```

输出应与下面相似：

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for mixing light: yellow
Try again!
BLUE
That's right!
Bye
```

3．新编写程序清单 10.7 和程序清单 10.8 描述的 Stock 类，使之使用动态分配的内存，而不是 string 类对象来存储股票名称。另外，使用重载的 operator < <()定义代替 show()成员函数。再使用程序清单 10.9 测试新的定义程序。

4．请看下面程序清单 10.10 定义的 Stack 类的变量：

```
// stack.h -- class declaration for the stack ADT
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};        // constant specific to class
    Item  * pitems;         // holds stack items
    int size;               // number of elements in stack
    int top;                // index for top stack item
public:
    Stack(int n = MAX);     // creates stack with n elements
    Stack(const Stack & st);
    ~Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item); // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);  // pop top into item
    Stack & operator=(const Stack & st);
};
```

正如私有成员表明的，这个类使用动态分配的数组来保存栈项。请重新编写方法，以适应这种新的表示法，并编写一个程序来演示所有的方法，包括复制构造函数和赋值运算符。

5．Heather 银行进行的研究表明，ATM 客户不希望排队时间不超过 1 分钟。使用程序清单 12.10 中的模拟，找出要使平均等候时间为 1 分钟，每小时到达的客户数应为多少（试验时间不短于 100 小时）？

6．Heather 银行想知道，如果再开设一台 ATM，情况将如何。请对模拟进行修改，以包含两个队列。假设当第一台 ATM 前的排队人数少于第二台 ATM 时，客户将排在第一队，否则将排在第二队。然后再找出要使平均等候时间为 1 分钟，每小时到达的客户数应该为多少（注意，这是一个非线性问题，即将 ATM 数量加倍，并不能保证每小时处理的客户数量也翻倍，并确保客户等候的时间少于 1 分钟）？

第 13 章　类继承

本章内容包括：

- is-a 关系的继承。
- 如何以公有方式从一个类派生出另一个类。
- 保护访问。
- 构造函数成员初始化列表。
- 向上和向下强制转换。
- 虚成员函数。
- 早期（静态）联编与晚期（动态）联编。
- 抽象基类。
- 纯虚函数。
- 何时及如何使用公有继承。

面向对象编程的主要目的之一是提供可重用的代码。开发新项目，尤其是当项目十分庞大时，重用经过测试的代码比重新编写代码要好得多。使用已有的代码可以节省时间，由于已有的代码已被使用和测试过，因此有助于避免在程序中引入错误。另外，必须考虑的细节越少，便越能专注于程序的整体策略。

传统的 C 函数库通过预定义、预编译的函数（如 strlen( )和 rand( )，可以在程序中使用这些函数）提供了可重用性。很多厂商都提供了专用的 C 库，这些专用库提供标准 C 库没有的函数。例如，可以购买数据库管理函数库和屏幕控制函数库。然而，函数库也有局限性。除非厂商提供了库函数的源代码（通常是不提供的），否则您将无法根据自己特定的需求，对函数进行扩展或修改，而必须根据库的情况修改自己的程序。即使厂商提供了源代码，在修改时也有一定的风险，如不经意地修改了函数的工作方式或改变了库函数之间的关系。

C++类提供了更高层次的重用性。目前，很多厂商提供了类库，类库由类声明和实现构成。因为类组合了数据表示和类方法，因此提供了比函数库更加完整的程序包。例如，单个类就可以提供用于管理对话框的全部资源。通常，类库是以源代码的方式提供的，这意味着可以对其进行修改，以满足需求。然而，C++提供了比修改代码更好的方法来扩展和修改类。这种方法叫作类继承，它能够从已有的类派生出新的类，而派生类继承了原有类（称为基类）的特征，包括方法。正如继承一笔财产要比自己白手起家容易一样，通过继承派生出的类通常比设计新类要容易得多。下面是可以通过继承完成的一些工作。

- 可以在已有类的基础上添加功能。例如，对于数组类，可以添加数学运算。
- 可以给类添加数据。例如，对于字符串类，可以派生出一个类，并添加指定字符串显示颜色的数据成员。
- 可以修改类方法的行为。例如，对于代表提供给飞机乘客的服务的 Passenger 类，可以派生出提供更高级别服务的 FirstClassPassenger 类。

当然，可以通过复制原始类代码，并对其进行修改来完成上述工作，但继承机制只需提供新特性，甚至不需要访问源代码就可以派生出类。因此，如果购买的类库只提供了类方法的头文件和编译后代码，仍可以使用库中的类派生出新的类。而且可以在不公开实现的情况下将自己的类分发给其他人，同时允许他

们在类中添加新特性。

继承是一种非常好的概念，其基本实现非常简单。但要对继承进行管理，使之在所有情况下都能正常工作，则需要做一些调整。本章将介绍继承简单的一面和复杂的一面。

13.1　一个简单的基类

从一个类派生出另一个类时，原始类称为基类，继承类称为派生类。为说明继承，首先需要一个基类。Webtown 俱乐部决定跟踪乒乓球会会员。作为俱乐部的首席程序员，需要设计一个简单的 TableTennisPlayer 类，如程序清单 13.1 和 13.2 所示。

程序清单 13.1　tabtenn0.h

```
// tabtenn0.h -- a table-tennis base class
#ifndef TABTENN0_H_
#define TABTENN0_H_
#include <string>
using std::string;
// simple base class
class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer (const string & fn = "none",
                       const string & ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; };
    void ResetTable(bool v) { hasTable = v; };
};
#endif
```

程序清单 13.2　tabtenn0.cpp

```
//tabtenn0.cpp -- simple base-class methods
#include "tabtenn0.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
    const string & ln, bool ht) : firstname(fn),
            lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
    std::cout << lastname << ", " << firstname;
}
```

TableTennisPlayer 类只是记录会员的姓名以及是否有球桌。有两点需要说明。首先，这个类使用标准 string 类来存储姓名，相比于使用字符数组，这更方便、更灵活、更安全，而与第 12 章的 String 类相比，这更专业。其次，构造函数使用了第 12 章介绍的成员初始化列表语法，但也可以像下面这样做：

```
TableTennisPlayer::TableTennisPlayer (const string & fn,
                    const string & ln, bool ht)
{
    firstname = fn;
    lastname = ln;
    hasTable = ht;
}
```

这将首先为 firstname 调用 string 的默认构造函数，再调用 string 的赋值运算符将 firstname 设置为 fn，但初始化列表语法可减少一个步骤，它直接使用 string 的复制构造函数将 firstname 初始化为 fn。

程序清单 13.3 使用了这个类。

程序清单 13.3　usett0.cpp

```
// usett0.cpp -- using a base class
#include <iostream>
#include "tabtenn0.h"

int main ( void )
{
    using std::cout;
    TableTennisPlayer player1("Chuck", "Blizzard", true);
    TableTennisPlayer player2("Tara", "Boomdea", false);
    player1.Name();
    if (player1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player2.Name();
    if (player2.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";

    return 0;
}
```

下面是程序清单 13.1.-13.3 组成的程序的输出：

```
Blizzard, Chuck: has a table.
Boomdea, Tara: hasn't a table.
```

注意到该程序实例化对象时将 C-风格字符串作为参数：

```
TableTennisPlayer player1("Chuck", "Blizzard", true);
TableTennisPlayer player2("Tara", "Boomdea", false);
```

但构造函数的形参类型被声明为 const string &。这导致类型不匹配，但与第 12 章创建的 String 类一样，string 类有一个将 const char *作为参数的构造函数，使用 C-风格字符串初始化 string 对象时，将自动调用这个构造函数。总之，可将 string 对象或 C-风格字符串作为构造函数 TableTennisPlayer 的参数；将前者作为参数时，将调用接受 const string &作为参数的 string 构造函数，而将后者作为参数时，将调用接受 const char *作为参数的 string 构造函数。

13.1.1　派生一个类

Webtown 俱乐部的一些成员曾经参加过当地的乒乓球锦标赛，需要这样一个类，它能包括成员在比赛中的比分。与其从零开始，不如从 TableTennisClass 类派生出一个类。首先将 RatedPlayer 类声明为从

TableTennisClass 类派生而来：

```
// RatedPlayer derives from the TableTennisPlayer base class
class RatedPlayer : public TableTennisPlayer
{
...
};
```

冒号指出 RatedPlayer 类的基类是 TableTennisplayer 类。上述特殊的声明头表明 TableTennisPlayer 是一个公有基类，这被称为公有派生。派生类对象包含基类对象。使用公有派生，基类的公有成员将成为派生类的公有成员；基类的私有部分也将成为派生类的一部分，但只能通过基类的公有和保护方法访问（稍后将介绍保护成员）。

上述代码完成了哪些工作呢？Ratedplayer 对象将具有以下特征：

- 派生类对象存储了基类的数据成员（派生类继承了基类的实现）；
- 派生类对象可以使用基类的方法（派生类继承了基类的接口）。

因此，RatedPlayer 对象可以存储运动员的姓名及其是否有球桌。另外，RatedPlayer 对象还可以使用 TableTennisPlayer 类的 Name( )、hasTable( )和 ResetTable( )方法（参见图 13.1）。

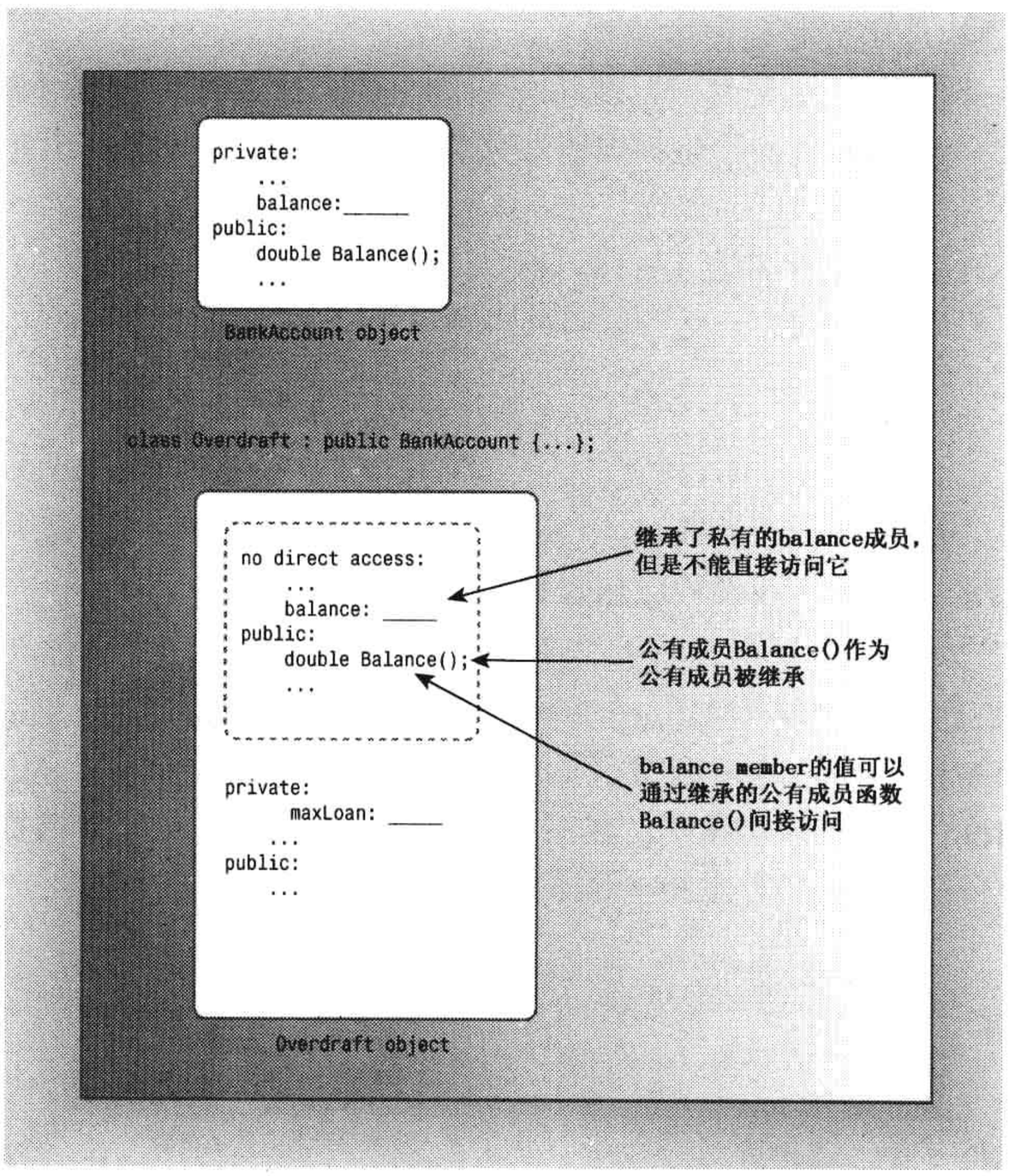


图 13.1　基类对象和派生类对象

需要在继承特性中添加什么呢？

- 派生类需要自己的构造函数。
- 派生类可以根据需要添加额外的数据成员和成员函数。

在这个例子中，派生类需要另一个数据成员来存储比分，还应包含检索比分的方法和重置比分的方法。因此，类声明与下面类似：

```
// simple derived class
class RatedPlayer : public TableTennisPlayer
```

```
{
private:
    unsigned int rating;     // add a data member
public:
    RatedPlayer (unsigned int r = 0, const string & fn = "none",
                 const string & ln = "none", bool ht = false);
    RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating() const { return rating; }   // add a method
    void ResetRating (unsigned int r) {rating = r;} // add a method
};
```

构造函数必须给新成员（如果有的话）和继承的成员提供数据。在第一个 RatedPlayer 构造函数中，每个成员对应一个形参；而第二个 Ratedplayer 构造函数使用一个 TableTennisPlayer 参数，该参数包括 firstname、lastname 和 hasTable。

13.1.2 构造函数：访问权限的考虑

派生类不能直接访问基类的私有成员，而必须通过基类方法进行访问。例如，RatedPlayer 构造函数不能直接设置继承的成员（firstname、lastname 和 hasTable），而必须使用基类的公有方法来访问私有的基类成员。具体地说，派生类构造函数必须使用基类构造函数。

创建派生类对象时，程序首先创建基类对象。从概念上说，这意味着基类对象应当在程序进入派生类构造函数之前被创建。C++使用成员初始化列表语法来完成这种工作。例如，下面是第一个 RatedPlayer 构造函数的代码：

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
    rating = r;
}
```

其中 `:TableTennisPlayer(fn,ln,ht)` 是成员初始化列表。它是可执行的代码，调用 TableTennisPlayer 构造函数。例如，假设程序包含如下声明：

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
```

则 RealPlayer 构造函数将把实参“Mallory”、“Duck”和 true 赋给形参 fn、In 和 ht，然后将这些参数作为实参传递给 TableTennisPlayer 构造函数，后者将创建一个嵌套 TableTennisPlayer 对象，并将数据“Mallory”、“Duck”和 true 存储在该对象中。然后，程序进入 RealPlayer 构造函数体，完成 RealPlayer 对象的创建，并将参数 r 的值（即 1140）赋给 rating 成员（参见图 13.2）。

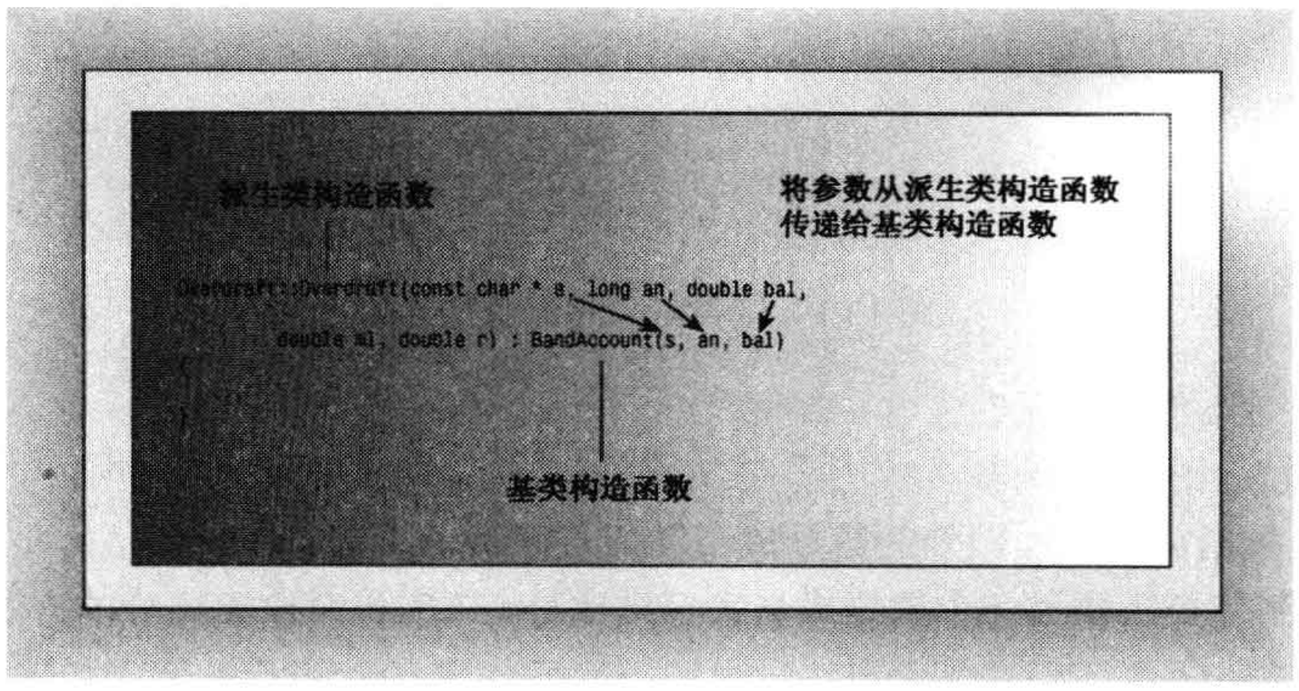


图 13.2 将参数传递给基类构造函数

如果省略成员初始化列表，情况将如何呢？

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) // what if no initializer list?
{
    rating = r;
}
```

必须首先创建基类对象，如果不调用基类构造函数，程序将使用默认的基类构造函数，因此上述代码与下面等效：

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) // : TableTennisPlayer()
{
    rating = r;
}
```

除非要使用默认构造函数，否则应显式调用正确的基类构造函数。

下面来看第二个构造函数的代码：

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp)
{
    rating = r;
}
```

这里也将 TableTennisPlayer 的信息传递给了 TableTennisPlayer 构造函数：

```
TableTennisPlayer(tp)
```

由于 tp 的类型为 TableTennisPlayer &，因此将调用基类的复制构造函数。基类没有定义复制构造函数，但第 12 章介绍过，如果需要使用复制构造函数但又没有定义，编译器将自动生成一个。在这种情况下，执行成员复制的隐式复制构造函数是合适的，因为这个类没有使用动态内存分配（string 成员确实使用了动态内存分配，但本书前面说过，成员复制将使用 string 类的复制构造函数来复制 string 成员）。

如果愿意，也可以对派生类成员使用成员初始化列表语法。在这种情况下，应在列表中使用成员名，而不是类名。所以，第二个构造函数可以按照下述方式编写：

```
// alternative version
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp), rating(r)
{
}
```

有关派生类构造函数的要点如下：

- 首先创建基类对象；
- 派生类构造函数应通过成员初始化列表将基类信息传递给基类构造函数；
- 派生类构造函数应初始化派生类新增的数据成员。

这个例子没有提供显式构造函数，因此将使用隐式构造函数。释放对象的顺序与创建对象的顺序相反，即首先执行派生类的析构函数，然后自动调用基类的析构函数。

注意：创建派生类对象时，程序首先调用基类构造函数，然后再调用派生类构造函数。基类构造函数负责初始化继承的数据成员；派生类构造函数主要用于初始化新增的数据成员。派生类的构造函数总是调用一个基类构造函数。可以使用初始化器列表语法指明要使用的基类构造函数，否则将使用默认的基类构造函数。

派生类对象过期时，程序将首先调用派生类析构函数，然后再调用基类析构函数。

成员初始化列表

派生类构造函数可以使用初始化器列表机制将值传递给基类构造函数。请看下面的例子：

```
derived::derived(type1 x, type2 y) : base(x,y) // initializer list
{
    ...
}
```

其中 derived 是派生类，base 是基类，x 和 y 是基类构造函数使用的变量。例如，如果派生类构造函数接收到参数 10 和 12，则这种机制将把 10 和 12 传递给被定义为接受这些类型的参数的基类构造函数。除虚基类外（参见第 14 章），类只能将值传递回相邻的基类，但后者可以使用相同的机制将信息传递给相邻的基类，依此类推。如果没有在成员初始化列表中提供基类构造函数，程序将使用默认的基类构造函数。成员初始化列表只能用于构造函数。

13.1.3 使用派生类

要使用派生类，程序必须要能够访问基类声明。程序清单 13.4 将这两种类的声明置于同一个头文件中。也可以将每个类放在独立的头文件中，但由于这两个类是相关的，所以把其类声明放在一起更合适。

程序清单 13.4 tabtenn1.h

```
// tabtenn1.h -- a table-tennis base class
#ifndef TABTENN1_H_
#define TABTENN1_H_
#include <string>
using std::string;
// simple base class
class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer (const string & fn = "none",
                       const string & ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; };
    void ResetTable(bool v) { hasTable = v; };
};

// simple derived class
class RatedPlayer : public TableTennisPlayer
{
private:
    unsigned int rating;
public:
    RatedPlayer (unsigned int r = 0, const string & fn = "none",
                 const string & ln = "none", bool ht = false);
    RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating() const { return rating; }
    void ResetRating (unsigned int r) {rating = r;}
};

#endif
```

程序清单 13.5 是这两个类的方法定义。同样，也可以使用不同的文件，但将定义放在一起更简单。

程序清单 13.5　tabtenn1.cpp

```
//tabtenn1.cpp -- simple base-class methods
#include "tabtenn1.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
    const string & ln, bool ht) : firstname(fn),
        lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
    std::cout << lastname << ", " << firstname;
}

// RatedPlayer methods
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
    rating = r;
}

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp), rating(r)
{
}
```

程序清单 13.6 创建了 TableTennisPlayer 类和 RatedPlayer 类的对象。请注意这两个类对象是如何使用 TableTennisPlayer 类的 Name( )和 HasTable( )方法的。

程序清单 13.6　usett1.cpp

```
// usett1.cpp -- using base class and derived class
#include <iostream>
#include "tabtenn1.h"

int main ( void )
{
    using std::cout;
    using std::endl;
    TableTennisPlayer player1("Tara", "Boomdea", false);
    RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
    rplayer1.Name();          // derived object uses base method
    if (rplayer1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player1.Name();           // base object uses base method
    if (player1.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";
    cout << "Name: ";
    rplayer1.Name();
    cout << "; Rating: " << rplayer1.Rating() << endl;
// initialize RatedPlayer using TableTennisPlayer object
```

```
    RatedPlayer rplayer2(1212, player1);
    cout << "Name: ";
    rplayer2.Name();
    cout << "; Rating: " << rplayer2.Rating() << endl;

    return 0;
}
```

下面是程序清单 13.4～程序清单 13.6 组成的程序的输出：

```
Duck, Mallory: has a table.
Boomdea, Tara: hasn't a table.
Name: Duck, Mallory; Rating: 1140
Name: Boomdea, Tara; Rating: 1212
```

13.1.4 派生类和基类之间的特殊关系

派生类与基类之间有一些特殊关系。其中之一是派生类对象可以使用基类的方法，条件是方法不是私有的：

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
rplayer1.Name();  // derived object uses base method
```

另外两个重要的关系是：基类指针可以在不进行显式类型转换的情况下指向派生类对象；基类引用可以在不进行显式类型转换的情况下引用派生类对象：

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();   // invoke Name() with reference
pt->Name();  // invoke Name() with pointer
```

然而，基类指针或引用只能用于调用基类方法，因此，不能使用 rt 或 pt 来调用派生类的 ResetRanking 方法。

通常，C++要求引用和指针类型与赋给的类型匹配，但这一规则对继承来说是例外。然而，这种例外只是单向的，不可以将基类对象和地址赋给派生类引用和指针：

```
TableTennisPlayer player("Betsy", "Bloop", true);
RatedPlayer & rr = player;      // NOT ALLOWED
RatedPlayer * pr = player;      // NOT ALLOWED
```

上述规则是有道理的。例如，如果允许基类引用隐式地引用派生类对象，则可以使用基类引用为派生类对象调用基类的方法。因为派生类继承了基类的方法，所以这样做不会出现问题。如果可以将基类对象赋给派生类引用，将发生什么情况呢？派生类引用能够为基对象调用派生类方法，这样做将出现问题。例如，将 RatedPlayer::Rating( )方法用于 TableTennisPlayer 对象是没有意义的，因为 TableTennisPlayer 对象没有 rating 成员。

如果基类引用和指针可以指向派生类对象，将出现一些很有趣的结果。其中之一是基类引用定义的函数或指针参数可用于基类对象或派生类对象。例如，在下面的函数中：

```
void Show(const TableTennisPlayer & rt)
{
    using std::cout;
    cout << "Name: ";
    rt.Name();
    cout << "\nTable: ";
    if (rt.HasTable())
        cout << "yes\n";
    else
        cout << "no\n";
}
```

形参 rt 是一个基类引用，它可以指向基类对象或派生类对象，所以可以在 Show( )中使用 TableTennis 参数或 Ratedplayer 参数：

```
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Show(player1);   // works with TableTennisPlayer argument
Show(rplayer1);  // works with RatedPlayer argument
```

对于形参为指向基类的指针的函数，也存在相似的关系。它可以使用基类对象的地址或派生类对象的地址作为实参：

```
void Wohs(const TableTennisPlayer * pt);  // function with pointer parameter
...
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Wohs(&player1);   // works with TableTennisPlayer * argument
Wohs(&rplayer1);  // works with RatedPlayer * argument
```

引用兼容性属性也让您能够将基类对象初始化为派生类对象，尽管不那么直接。假设有这样的代码：

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer olaf2(olaf1);
```

要初始化 olaf2，匹配的构造函数的原型如下：

```
TableTennisPlayer(const RatedPlayer &);        // doesn't exist
```

类定义中没有这样的构造函数，但存在隐式复制构造函数：

```
// implicit copy constructor
TableTennisPlayer(const TableTennisPlayer &);
```

形参是基类引用，因此它可以引用派生类。这样，将 olaf2 初始化为 olaf1 时，将要使用该构造函数，它复制 firstname、lastname 和 hasTable 成员。换句话来说，它将 olaf2 初始化为嵌套在 RatedPlayer 对象 olaf1 中的 TableTennisPlayer 对象。

同样，也可以将派生对象赋给基类对象：

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer winner;
winner = olaf1; // assign derived to base object
```

在这种情况下，程序将使用隐式重载赋值运算符：

```
TableTennisPlayer & operator=(const TableTennisPlayer &) const;
```

基类引用指向的也是派生类对象，因此 olaf1 的基类部分被复制给 winner。

13.2　继承：is-a 关系

派生类和基类之间的特殊关系是基于 C++继承的底层模型的。实际上，C++有 3 种继承方式：公有继承、保护继承和私有继承。公有继承是最常用的方式，它建立一种 is-a 关系，即派生类对象也是一个基类对象，可以对基类对象执行的任何操作，也可以对派生类对象执行。例如，假设有一个 Fruit 类，可以保存水果的重量和热量。因为香蕉是一种特殊的水果，所以可以从 Fruit 类派生出 Banana 类。新类将继承原始类的所有数据成员，因此，Banana 对象将包含表示香蕉重量和热量的成员。新的 Banana 类还添加了专门用于香蕉的成员，这些成员通常不用于水果，例如 Banana Institute Peel Index（香蕉机构果皮索引）。因为派生类可以添加特性，所以，将这种关系称为 is-a-kind-of（是一种）关系可能更准确，但是通常使用术语 is-a。

为阐明 is-a 关系，来看一些与该模型不符的例子。公有继承不建立 has-a 关系。例如，午餐可能包括水果，但通常午餐并不是水果。所以，不能通过从 Fruit 类派生出 Lunch 类来在午餐中添加水果。在午餐中加入水果的正确方法是将其作为一种 has-a 关系：午餐有水果。正如将在第 14 章介绍的，最容易的建模方

式是，将 Fruit 对象作为 Lunch 类的数据成员（参见图 13.3）。

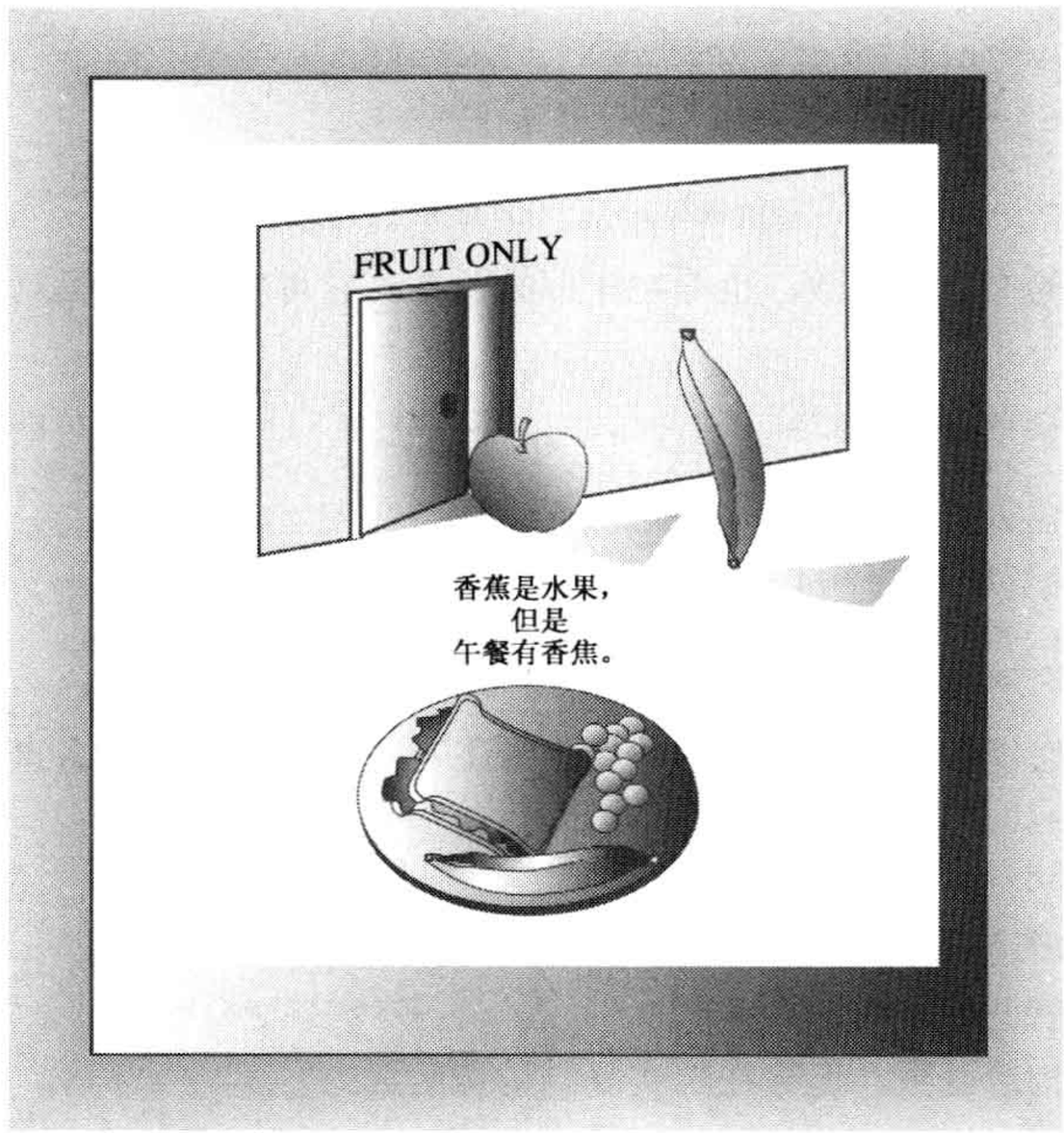


图 13.3　is-a 关系和 has-a 关系

公有继承不能建立 is-like-a 关系，也就是说，它不采用明喻。人们通常说律师就像鲨鱼，但律师并不是鲨鱼。例如，鲨鱼可以在水下生活。所以，不应从 Shark 类派生出 Lawyer 类。继承可以在基类的基础上添加属性，但不能删除基类的属性。在有些情况下，可以设计一个包含共有特征的类，然后以 is-a 或 has-a 关系，在这个类的基础上定义相关的类。

公有继承不建立 is-implemented-as-a（作为……来实现）关系。例如，可以使用数组来实现栈，但从 Array 类派生出 Stack 类是不合适的，因为栈不是数组。例如，数组索引不是栈的属性。另外，可以以其他方式实现栈，如链表。正确的方法是，通过让栈包含一个私有 Array 对象成员来隐藏数组实现。

公有继承不建立 uses-a 关系。例如，计算机可以使用激光打印机，但从 Computer 类派生出 Printer 类（或反过来）是没有意义的。然而，可以使用友元函数或类来处理 Printer 对象和 Computer 对象之间的通信。

在 C++中，完全可以使用公有继承来建立 has-a、is-implemented-as-a 或 uses-a 关系；然而，这样做通常会导致编程方面的问题。因此，还是坚持使用 is-a 关系吧。

13.3　多态公有继承

RatedPlayer 继承示例很简单。派生类对象使用基类的方法，而未做任何修改。然而，可能会遇到这样的情况，即希望同一个方法在派生类和基类中的行为是不同的。换句话来说，方法的行为应取决于调用该方法的对象。这种较复杂的行为称为多态——具有多种形态，即同一个方法的行为随上下文而异。有两种重要的机制可用于实现多态公有继承：

- 在派生类中重新定义基类的方法。
- 使用虚方法。

现在来看另一个例子。由于 Webtown 俱乐部的工作经历，您成了 Pontoon 银行的首席程序员。银行要求您完成的第一项工作是开发两个类。一个类用于表示基本支票账户——Brass Account，另一个类用于表示代表 Brass Plus 支票账户，它添加了透支保护特性。也就是说，如果用户签出一张超出其存款余额的支

票——但是超出的数额并不是很大，银行将支付这张支票，对超出的部分收取额外的费用，并追加罚款。可以根据要保存的数据以及允许执行的操作来确定这两种账户的特征。

下面是用于 Brass Account 支票账户的信息：

- 客户姓名；
- 账号；
- 当前结余。

下面是可以执行的操作：

- 创建账户；
- 存款；
- 取款；
- 显示账户信息。

Pontoon 银行希望 Brass Plus 支票账户包含 Brass Account 的所有信息及如下信息：

- 透支上限；
- 透支贷款利率；
- 当前的透支总额。

不需要新增操作，但有两种操作的实现不同：

- 对于取款操作，必须考虑透支保护；
- 显示操作必须显示 Brass Plus 账户的其他信息。

假设将第一个类命名为 Brass，第二个类为 BrassPlus。应从 Brass 公有派生出 BrassPlus 吗？要回答这个问题，必须先回答另一个问题：BrassPlus 类是否满足 is-a 条件？当然满足。对于 Brass 对象是正确的事情，对于 BrassPlus 对象也是正确的。它们都将保存客户姓名、账号以及结余。使用这两个类都可以存款、取款和显示账户信息。请注意，is-a 关系通常是不可逆的。也就是说，水果不是香蕉；同样，Brass 对象不具备 BrassPlus 对象的所有功能。

13.3.1　开发 Brass 类和 BrassPlus 类

Brass Account 类的信息很简单，但是银行没有告诉您有关透支系统的细节。当您向友好的 Pontoon 银行代表询问时，他提供了如下信息：

- Brass Plus 账户限制了客户的透支款额。默认为 500 元，但有些客户的限额可能不同；
- 银行可以修改客户的透支限额；
- Brass Plus 账户对贷款收取利息。默认为 11.125%，但有些客户的利率可能不同；
- 银行可以修改客户的利率；
- 账户记录客户所欠银行的金额（透支数额加利息）。用户不能通过常规存款或从其他账户转账的方式偿付，而必须以现金的方式交给特定的银行工作人员。如果有必要，工作人员可以找到该客户。欠款偿还后，欠款金额将归零。

最后一种特性是银行出于做生意的考虑而采用的，这种方法有它有利的一面——使编程更简单。

上述列表表明，新的类需要构造函数，而且构造函数应提供账户信息，设置透支上限（默认为 500 元）和利率（默认为 11.125%）。另外，还应有重新设置透支限额、利率和当前欠款的方法。要添加到 Brass 类中的就是这些，这将在 BrassPlus 类声明中声明。

有关这两个类的信息声明，类声明应类似于程序清单 13.7。

程序清单 13.7　brass.h

```
// brass.h  -- bank account classes
#ifndef BRASS_H_
#define BRASS_H_
#include <string>
// Brass Account Class
```

```
class Brass
{
private:
    std::string fullName;
    long acctNum;
    double balance;
public:
    Brass(const std::string & s = "Nullbody", long an = -1,
                double bal = 0.0);
    void Deposit(double amt);
    virtual void Withdraw(double amt);
    double Balance() const;
    virtual void ViewAcct() const;
    virtual ~Brass() {}
};

//Brass Plus Account Class
class BrassPlus : public Brass
{
private:
    double maxLoan;
    double rate;
    double owesBank;
public:
    BrassPlus(const std::string & s = "Nullbody", long an = -1,
            double bal = 0.0, double ml = 500,
            double r = 0.11125);
    BrassPlus(const Brass & ba, double ml = 500,
                            double r = 0.11125);
    virtual void ViewAcct()const;
    virtual void Withdraw(double amt);
    void ResetMax(double m) { maxLoan = m; }
    void ResetRate(double r) { rate = r; };
    void ResetOwes() { owesBank = 0; }
};

#endif
```

对于程序清单 13.7，需要说明的有下面几点：

- BrassPlus 类在 Brass 类的基础上添加了 3 个私有数据成员和 3 个公有成员函数；
- Brass 类和 BrassPlus 类都声明了 ViewAcct( )和 Withdraw( )方法，但 BrassPlus 对象和 Brass 对象的这些方法的行为是不同的；
- Brass 类在声明 ViewAcct( )和 Withdraw( )时使用了新关键字 virtual。这些方法被称为虚方法（virtual method）；
- Brass 类还声明了一个虚析构函数，虽然该析构函数不执行任何操作。

第一点没有什么新鲜的。RatedPlayer 类在 TableTennisPlayer 类的基础上添加新数据成员和 2 个新方法的方式与此类似。

第二点介绍了声明如何指出方法在派生类的行为的不同。两个 ViewAcct( )原型表明将有 2 个独立的方法定义。基类版本的限定名为 Brass::ViewAcct( )，派生类版本的限定名为 BrassPlus::ViewAcct( )。程序将使用对象类型来确定使用哪个版本：

```
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
```

```
dom.ViewAcct();        // use Brass::ViewAcct()
dot.ViewAcct();        // use BrassPlus::ViewAcct()
```

同样，Withdraw( )也有 2 个版本，一个供 Brass 对象使用，另一个供 BrassPlus 对象使用。对于在两个类中行为相同的方法（如 Deposit( )和 Balance( )），则只在基类中声明。

第三点（使用 virtual）比前两点要复杂。如果方法是通过引用或指针而不是对象调用的，它将确定使用哪一种方法。如果没有使用关键字 virtual，程序将根据引用类型或指针类型选择方法；如果使用了 virtual，程序将根据引用或指针指向的对象的类型来选择方法。如果 ViewAcct( )不是虚的，则程序的行为如下：

```
// behavior with non-virtual ViewAcct()
// method chosen according to reference type
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();       // use Brass::ViewAcct()
b2_ref.ViewAcct();       // use Brass::ViewAcct()
```

引用变量的类型为 Brass，所以选择了 Brass::ViewAccount( )。使用 Brass 指针代替引用时，行为将与此类似。

如果 ViewAcct( )是虚的，则行为如下：

```
// behavior with virtual ViewAcct()
// method chosen according to object type
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();       // use Brass::ViewAcct()
b2_ref.ViewAcct();       // use BrassPlus::ViewAcct()
```

这里两个引用的类型都是 Brass，但 b2_ref 引用的是一个 BrassPlus 对象，所以使用的是 BrassPlus::ViewAcct( )。使用 Brass 指针代替引用时，行为将类似。

稍后您将看到，虚函数的这种行为非常方便。因此，经常在基类中将派生类会重新定义的方法声明为虚方法。方法在基类中被声明为虚的后，它在派生类中将自动成为虚方法。然而，在派生类声明中使用关键字 virtual 来指出哪些函数是虚函数也不失为一个好办法。

第四点是，基类声明了一个虚析构函数。这样做是为了确保释放派生对象时，按正确的顺序调用析构函数。本章后面将详细介绍这个问题。

注意：如果要在派生类中重新定义基类的方法，通常应将基类方法声明为虚的。这样，程序将根据对象类型而不是引用或指针的类型来选择方法版本。为基类声明一个虚析构函数也是一种惯例。

1. 类实现

接下来需要实现类，其中的部分工作已由头文件中的内联函数定义完成了。程序清单 13.8 列出了其他方法的定义。注意，关键字 virtual 只用于类声明的方法原型中，而没有用于程序清单 13.8 的方法定义中。

程序清单 13.8　brass.cpp

```
// brass.cpp -- bank account class methods
#include <iostream>
#include "brass.h"
using std::cout;
using std::endl;
using std::string;

// formatting stuff
```

```
typedef std::ios_base::fmtflags format;
typedef std::streamsize precis;
format setFormat();
void restore(format f, precis p);

// Brass methods

Brass::Brass(const string & s, long an, double bal)
{
    fullName = s;
    acctNum = an;
    balance = bal;
}

void Brass::Deposit(double amt)
{
    if (amt < 0)
        cout << "Negative deposit not allowed; "
             << "deposit is cancelled.\n";
    else
        balance += amt;
}

void Brass::Withdraw(double amt)
{
    // set up ###.## format
    format initialState = setFormat();
    precis prec = cout.precision(2);

    if (amt < 0)
        cout << "Withdrawal amount must be positive; "
             << "withdrawal canceled.\n";
    else if (amt <= balance)
        balance -= amt;
    else
        cout << "Withdrawal amount of $" << amt
             << " exceeds your balance.\n"
             << "Withdrawal canceled.\n";
    restore(initialState, prec);
}
double Brass::Balance() const
{
    return balance;
}

void Brass::ViewAcct() const
{
     // set up ###.## format
    format initialState = setFormat();
    precis prec = cout.precision(2);
    cout << "Client: " << fullName << endl;
    cout << "Account Number: " << acctNum << endl;
    cout << "Balance: $" << balance << endl;
    restore(initialState, prec); // restore original format
}
```

```
// BrassPlus Methods
BrassPlus::BrassPlus(const string & s, long an, double bal,
          double ml, double r) : Brass(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
          : Brass(ba)   // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

// redefine how ViewAcct() works
void BrassPlus::ViewAcct() const
{
    // set up ###.## format
    format initialState = setFormat();
    precis prec = cout.precision(2);

    Brass::ViewAcct();   // display base portion
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout.precision(3);  // ###.### format
    cout << "Loan Rate: " << 100 * rate << "%\n";
    restore(initialState, prec);
}

// redefine how Withdraw() works
void BrassPlus::Withdraw(double amt)
{
    // set up ###.## format
    format initialState = setFormat();
    precis prec = cout.precision(2);

    double bal = Balance();
    if (amt <= bal)
        Brass::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit(advance);
        Brass::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
    restore(initialState, prec);
}
```

```
format setFormat()
{
    // set up ###.## format
    return cout.setf(std::ios_base::fixed,
                     std::ios_base::floatfield);
}

void restore(format f, precis p)
{
    cout.setf(f, std::ios_base::floatfield);
    cout.precision(p);
}
```

介绍程序清单 13.8 的具体细节（如一些方法的格式化处理）之前，先来看一下与继承直接相关的方面。记住，派生类并不能直接访问基类的私有数据，而必须使用基类的公有方法才能访问这些数据。访问的方式取决于方法。构造函数使用一种技术，而其他成员函数使用另一种技术。

派生类构造函数在初始化基类私有数据时，采用的是成员初始化列表语法。RatedPlayer 类构造函数和 BrassPlus 构造函数都使用这种技术：

```
BrassPlus::BrassPlus(const string & s, long an, double bal,
                     double ml, double r) : Brass(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
                     : Brass(ba)   // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

这几个构造函数都使用成员初始化列表语法，将基类信息传递给基类构造函数，然后使用构造函数体初始化 BrassPlus 类新增的数据项。

非构造函数不能使用成员初始化列表语法，但派生类方法可以调用公有的基类方法。例如，BrassPlus 版本的 ViewAcct( )核心内容如下（忽略了格式方面）：

```
// redefine how ViewAcct() works
void BrassPlus::ViewAcct() const
{
...
    Brass::ViewAcct();  // display base portion
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout << "Loan Rate: " << 100 * rate << "%\n";
...
}
```

换句话说，BrassPlus::ViewAcct( )显示新增的 BrassPlus 数据成员，并调用基类方法 Brass::ViewAcct( ) 来显示基类数据成员。在派生类方法中，标准技术是使用作用域解析运算符来调用基类方法。

代码必须使用作用域解析运算符。假如这样编写代码：

```
// redefine erroneously how ViewAcct() works
void BrassPlus::ViewAcct() const
{
...
    ViewAcct();   // oops! recursive call
...
}
```

如果代码没有使用作用域解析运算符，编译器将认为 ViewAcct( )是 BrassPlus::ViewAcct( )，这将创建一个不会终止的递归函数——这可不好。

接下来看 BrassPlus::Withdraw( )方法。如果客户提取的金额超过了结余，该方法将安排贷款。它可以使用 Brass::Withdraw( )来访问 balance 成员，但如果取款金额超过了结余，Brass::Withdraw( )将发出一个错误消息。这种实现使用 Deposit( )方法进行放贷，然后在得到了足够的结余后调用 Brass::Withdraw，从而避免了错误消息：

```
// redefine how Withdraw() works
void BrassPlus::Withdraw(double amt)
{
...
    double bal = Balance();
    if (amt <= bal)
        Brass::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit(advance);
        Brass::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
...
}
```

该方法使用基类的 Balance( )函数来确定结余。因为派生类没有重新定义该方法，代码不必对 Balance( )使用作用域解析运算符。

方法 ViewAcct( )和 Withdraw( )使用格式化方法 setf( )和 precision( )将浮点值的输出模式设置为定点，即包含两位小数。设置模式后，输出的模式将保持不变，因此该方法将格式模式重置为调用前的状态。这与程序清单 8.8 和程序清单 10.5 类似。为避免代码重复，该程序将设置格式的代码放在辅助函数中：

```
// formatting stuff
typedef std::ios_base::fmtflags format;
typedef std::streamsize precis;
format setFormat();
void restore(format f, precis p);
```

函数 setFormat( )设置定点表示法并返回以前的标记设置：

```
format setFormat()
{
    // set up ###.## format
    return cout.setf(std::ios_base::fixed,
                     std::ios_base::floatfield);
}
```

而函数 restore( )重置格式和精度：

```
void restore(format f, precis p)
{
    cout.setf(f, std::ios_base::floatfield);
    cout.precision(p);
}
```

有关设置输出格式的更详细信息，请参阅第 17 章。

2. 使用 Brass 和 BrassPlus 类

清单 13.9 使用了一个 Brass 对象和一个 BrassPlus 对象来测试类定义。

程序清单 13.9　usebrass1.cpp

```
// usebrass1.cpp -- testing bank account classes
// compile with brass.cpp
#include <iostream>
#include "brass.h"

int main()
{
    using std::cout;
    using std::endl;

    Brass Piggy("Porcelot Pigg", 381299, 4000.00);
    BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);
    Piggy.ViewAcct();
    cout << endl;
    Hoggy.ViewAcct();
    cout << endl;
    cout << "Depositing $1000 into the Hogg Account:\n";
    Hoggy.Deposit(1000.00);
    cout << "New balance: $" << Hoggy.Balance() << endl;
    cout << "Withdrawing $4200 from the Pigg Account:\n";
    Piggy.Withdraw(4200.00);
    cout << "Pigg account balance: $" << Piggy.Balance() << endl;
    cout << "Withdrawing $4200 from the Hogg Account:\n";
    Hoggy.Withdraw(4200.00);
    Hoggy.ViewAcct();

    return 0;
}
```

下面是程序清单 13.9 所示程序的输出，请注意为何 Hogg 受透支限制，而 Pigg 没有：

```
Client: Porcelot Pigg
Account Number: 381299
Balance: $4000.00

Client: Horatio Hogg
Account Number: 382288
Balance: $3000.00
Maximum loan: $500.00
Owed to bank: $0.00
Loan Rate: 11.125%

Depositing $1000 into the Hogg Account:
```

```
New balance: $4000
Withdrawing $4200 from the Pigg Account:
Withdrawal amount of $4200.00 exceeds your balance.
Withdrawal canceled.
Pigg account balance: $4000
Withdrawing $4200 from the Hogg Account:
Bank advance: $200.00
Finance charge: $22.25
Client: Horatio Hogg
Account Number: 382288
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $222.25
Loan Rate: 11.125%
```

3. 演示虚方法的行为

在程序清单 13.9 中，方法是通过对象（而不是指针或引用）调用的，没有使用虚方法特性。下面来看一个使用了虚方法的例子。假设要同时管理 Brass 和 BrassPlus 账户，如果能使用同一个数组来保存 Brsss 和 BrassPlus 对象，将很有帮助，但这是不可能的。数组中所有元素的类型必须相同，而 Brass 和 BrassPlus 是不同的类型。然而，可以创建指向 Brass 的指针数组。这样，每个元素的类型都相同，但由于使用的是公有继承模型，因此 Brass 指针既可以指向 Brass 对象，也可以指向 BrassPlus 对象。因此，可以使用一个数组来表示多种类型的对象。这就是多态性，程序清单 13.10 是一个简单的例子。

程序清单 13.10　usebrass2.cpp

```
// usebrass2.cpp -- polymorphic example
// compile with brass.cpp
#include <iostream>
#include <string>
#include "brass.h"
const int CLIENTS = 4;

int main()
{
   using std::cin;
   using std::cout;
   using std::endl;

   Brass * p_clients[CLIENTS];
   std::string temp;
   long tempnum;
   double tempbal;
   char kind;

   for (int i = 0; i < CLIENTS; i++)
   {
       cout << "Enter client's name: ";
       getline(cin,temp);
       cout << "Enter client's account number: ";
       cin >> tempnum;
       cout << "Enter opening balance: $";
       cin >> tempbal;
       cout << "Enter 1 for Brass Account or "
            << "2 for BrassPlus Account: ";
```

```
        while (cin >> kind && (kind != '1' && kind != '2'))
            cout <<"Enter either 1 or 2: ";
        if (kind == '1')
            p_clients[i] = new Brass(temp, tempnum, tempbal);
        else
        {
            double tmax, trate;
            cout << "Enter the overdraft limit: $";
            cin >> tmax;
            cout << "Enter the interest rate "
                 << "as a decimal fraction: ";
            cin >> trate;
            p_clients[i] = new BrassPlus(temp, tempnum, tempbal,
                                         tmax, trate);
        }
        while (cin.get() != '\n')
            continue;
    }
    cout << endl;
    for (int i = 0; i < CLIENTS; i++)
    {
        p_clients[i]->ViewAcct();
        cout << endl;
    }

    for (int i = 0; i < CLIENTS; i++)
    {
        delete p_clients[i];  // free memory
    }
    cout << "Done.\n";
    return 0;
}
```

程序清单 13.10 根据用户的输入来确定要添加的账户类型，然后使用 new 创建并初始化相应类型的对象。您可能还记得，getline（cin, temp）从 cin 读取一行输入，并将其存储到 string 对象 temp 中。

下面是该程序的运行情况：

```
Enter client's name: Harry Fishsong
Enter client's account number: 112233
Enter opening balance: $1500
Enter 1 for Brass Account or 2 for BrassPlus Account: 1
Enter client's name: Dinah Otternoe
Enter client's account number: 121213
Enter opening balance: $1800
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $350
Enter the interest rate as a decimal fraction: 0.12
Enter client's name: Brenda Birdherd
Enter client's account number: 212118
Enter opening balance: $5200
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $800
Enter the interest rate as a decimal fraction: 0.10
Enter client's name: Tim Turtletop
Enter client's account number: 233255
```

```
Enter opening balance: $688
Enter 1 for Brass Account or 2 for BrassPlus Account: 1

Client: Harry Fishsong
Account Number: 112233
Balance: $1500.00

Client: Dinah Otternoe
Account Number: 121213
Balance: $1800.00
Maximum loan: $350.00
Owed to bank: $0.00
Loan Rate: 12.00%

Client: Brenda Birdherd
Account Number: 212118
Balance: $5200.00
Maximum loan: $800.00
Owed to bank: $0.00
Loan Rate: 10.00%

Client: Tim Turtletop
Account Number: 233255
Balance: $688.00

Done.
```

多态性是由下述代码提供的：

```
for (i = 0; i < CLIENTS; i++)
{
    p_clients[i]->ViewAcct();
    cout << endl;
}
```

如果数组成员指向的是 Brass 对象，则调用 Brass::ViewAcct( )；如果指向的是 BrassPlus 对象，则调用 BrassPlus::ViewAcct( )。如果 Brass::ViewAcct( )被声明为虚的，则在任何情况下都将调用 Brass::ViewAcct( )。

4. 为何需要虚析构函数

在程序清单 13.10 中，使用 delete 释放由 new 分配的对象的代码说明了为何基类应包含一个虚析构函数，虽然有时好像并不需要析构函数。如果析构函数不是虚的，则将只调用对应于指针类型的析构函数。对于程序清单 13.10，这意味着只有 Brass 的析构函数被调用，即使指针指向的是一个 BrassPlus 对象。如果析构函数是虚的，将调用相应对象类型的析构函数。因此，如果指针指向的是 BrassPlus 对象，将调用 BrassPlus 的析构函数，然后自动调用基类的析构函数。因此，使用虚析构函数可以确保正确的析构函数序列被调用。对于程序清单 13.10，这种正确的行为并不是很重要，因为析构函数没有执行任何操作。然而，如果 BrassPlus 包含一个执行某些操作的析构函数，则 Brass 必须有一个虚析构函数，即使该析构函数不执行任何操作。

13.4　静态联编和动态联编

程序调用函数时，将使用哪个可执行代码块呢？编译器负责回答这个问题。将源代码中的函数调用解释为执行特定的函数代码块被称为函数名联编（binding）。在 C 语言中，这非常简单，因为每个函数名都对应一个不同的函数。在 C++中，由于函数重载的缘故，这项任务更复杂。编译器必须查看函数参数以及函数名才能确定使用哪个函数。然而，C/C++编译器可以在编译过程完成这种联编。在编译过程中进行联

编被称为静态联编（static binding），又称为早期联编（early binding）。然而，虚函数使这项工作变得更困难。正如在程序清单 13.10 所示的那样，使用哪一个函数是不能在编译时确定的，因为编译器不知道用户将选择哪种类型的对象。所以，编译器必须生成能够在程序运行时选择正确的虚方法的代码，这被称为动态联编（dynamic binding），又称为晚期联编（late binding）。

知道虚方法的行为后，下面深入地探讨这一过程，首先介绍 C++如何处理指针和引用类型的兼容性。

13.4.1 指针和引用类型的兼容性

在 C++中，动态联编与通过指针和引用调用方法相关，从某种程度上说，这是由继承控制的。公有继承建立 is-a 关系的一种方法是如何处理指向对象的指针和引用。通常，C++不允许将一种类型的地址赋给另一种类型的指针，也不允许一种类型的引用指向另一种类型：

```
double x = 2.5;
int * pi = &x;   // invalid assignment, mismatched pointer types
long & rl = x;   // invalid assignment, mismatched reference type
```

然而，正如您看到的，指向基类的引用或指针可以引用派生类对象，而不必进行显式类型转换。例如，下面的初始化是允许的：

```
BrassPlus dilly ("Annie Dill", 493222, 2000);
Brass * pb = &dilly;   // ok
Brass & rb = dilly;    // ok
```

将派生类引用或指针转换为基类引用或指针被称为向上强制转换（upcasting），这使公有继承不需要进行显式类型转换。该规则是 is-a 关系的一部分。BrassPlus 对象都是 Brass 对象，因为它继承了 Brass 对象所有的数据成员和成员函数。所以，可以对 Brass 对象执行的任何操作，都适用于 BrassPlus 对象。因此，为处理 Brass 引用而设计的函数可以对 BrassPlus 对象执行同样的操作，而不必担心会导致任何问题。将指向对象的指针作为函数参数时，也是如此。向上强制转换是可传递的，也就是说，如果从 BrassPlus 派生出 BrassPlusPlus 类，则 Brass 指针或引用可以引用 Brass 对象、BrassPlus 对象或 BrassPlusPlus 对象。

相反的过程——将基类指针或引用转换为派生类指针或引用——称为向下强制转换（downcasting）。如果不使用显式类型转换，则向下强制转换是不允许的。原因是 is-a 关系通常是不可逆的。派生类可以新增数据成员，因此使用这些数据成员的类成员函数不能应用于基类。例如，假设从 Employee 类派生出 Singer 类，并添加了表示歌手音域的数据成员和用于报告音域的值的成员函数 range( )，则将 range( )方法应用于 Employee 对象是没有意义的。但如果允许隐式向下强制转换，则可能无意间将指向 Singer 的指针设置为一个 Employee 对象的地址，并使用该指针来调用 range( )方法（参见图 13.4）。

对于使用基类引用或指针作为参数的函数调用，将进行向上转换。请看下面的代码段，这里假定每个函数都调用虚方法 ViewAcct( )：

```
void fr(Brass & rb); // uses rb.ViewAcct()
void fp(Brass * pb); // uses pb->ViewAcct()
void fv(Brass b);    // uses b.ViewAcct()
int main()
{
    Brass b("Billy Bee", 123432, 10000.0);
    BrassPlus bp("Betty Beep", 232313, 12345.0);
    fr(b);  // uses Brass::ViewAcct()
    fr(bp); // uses BrassPlus::ViewAcct()
    fp(b);  // uses Brass::ViewAcct()
    fp(bp); // uses BrassPlus::ViewAcct()
    fv(b);  // uses Brass::ViewAcct()
    fv(bp); // uses Brass::ViewAcct()
...
}
```

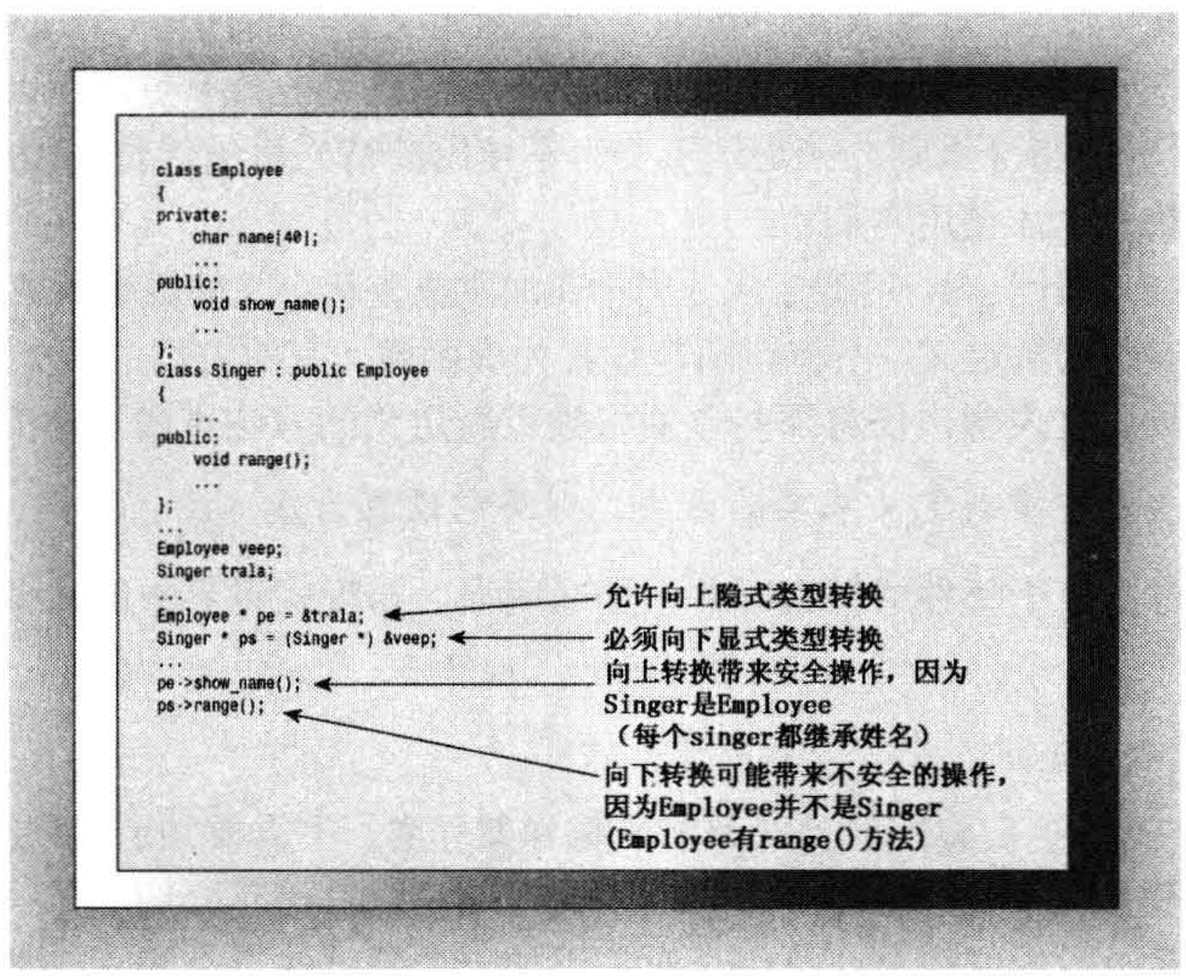


图 13.4　向上强制转换和向下强制转换

按值传递导致只将 BrassPlus 对象的 Brass 部分传递给函数 fv( )。但随引用和指针发生的隐式向上转换导致函数 fr( )和 fp( )分别为 Brass 对象和 BrassPlus 对象使用 Brass::ViewAcct( )和 BrassPlus::ViewAcct( )。

隐式向上强制转换使基类指针或引用可以指向基类对象或派生类对象，因此需要动态联编。C++使用虚成员函数来满足这种需求。

13.4.2　虚成员函数和动态联编

来回顾一下使用引用或指针调用方法的过程。请看下面的代码：

```
BrassPlus ophelia;     // derived-class object
Brass * bp;            // base-class pointer
bp = &ophelia;         // Brass pointer to BrassPlus object
bp->ViewAcct();        // which version?
```

正如前面介绍的，如果在基类中没有将 ViewAcct( )声明为虚的，则 bp->ViewAcct( )将根据指针类型（Brass *）调用 Brass::ViewAcct( )。指针类型在编译时已知，因此编译器在编译时，可以将 ViewAcct( )关联到 Brass::ViewAcct( )。总之，编译器对非虚方法使用静态联编。

然而，如果在基类中将 ViewAcct( )声明为虚的，则 bp->ViewAcct( )根据对象类型（BrassPlus）调用 BrassPlus::ViewAcct( )。在这个例子中，对象类型为 BrassPlus，但通常（如程序清单 13.10 所示）只有在运行程序时才能确定对象的类型。所以编译器生成的代码将在程序执行时，根据对象类型将 ViewAcct( )关联到 Brass::ViewAcct( )或 BrassPlus::ViewAcct( )。总之，编译器对虚方法使用动态联编。

在大多数情况下，动态联编很好，因为它让程序能够选择为特定类型设计的方法。因此，您可能会问：

- 为什么有两种类型的联编？
- 既然动态联编如此之好，为什么不将它设置成默认的？
- 动态联编是如何工作的？

下面来看看这些问题的答案。

1. 为什么有两种类型的联编以及为什么默认为静态联编

如果动态联编让您能够重新定义类方法，而静态联编在这方面很差，为何不摒弃静态联编呢？原因有两个——效率和概念模型。

首先来看效率。为使程序能够在运行阶段进行决策，必须采取一些方法来跟踪基类指针或引用指向的对象类型，这增加了额外的处理开销（稍后将介绍一种动态联编方法）。例如，如果类不会用作基类，则不

需要动态联编。同样，如果派生类（如 RatedPlayer）不重新定义基类的任何方法，也不需要使用动态联编。在这些情况下，使用静态联编更合理，效率也更高。由于静态联编的效率更高，因此被设置为 C++的默认选择。Strousstrup 说，C++的指导原则之一是，不要为不使用的特性付出代价（内存或者处理时间）。仅当程序设计确实需要虚函数时，才使用它们。

接下来看概念模型。在设计类时，可能包含一些不在派生类重新定义的成员函数。例如，Brass::Balance( ) 函数返回账户结余，不应该重新定义。不将该函数设置为虚函数，有两方面的好处：首先效率更高；其次，指出不要重新定义该函数。这表明，仅将那些预期将被重新定义的方法声明为虚的。

提示：如果要在派生类中重新定义基类的方法，则将它设置为虚方法；否则，设置为非虚方法。

当然，设计类时，方法属于哪种情况有时并不那么明显。与现实世界中的很多方面一样，类设计并不是一个线性过程。

2. 虚函数的工作原理

C++规定了虚函数的行为，但将实现方法留给了编译器作者。不需要知道实现方法就可以使用虚函数，但了解虚函数的工作原理有助于更好地理解概念，因此，这里对其进行介绍。

通常，编译器处理虚函数的方法是：给每个对象添加一个隐藏成员。隐藏成员中保存了一个指向函数地址数组的指针。这种数组称为虚函数表（virtual function table，vtbl）。虚函数表中存储了为类对象进行声明的虚函数的地址。例如，基类对象包含一个指针，该指针指向基类中所有虚函数的地址表。派生类对象将包含一个指向独立地址表的指针。如果派生类提供了虚函数的新定义，该虚函数表将保存新函数的地址；如果派生类没有重新定义虚函数，该 vtbl 将保存函数原始版本的地址。如果派生类定义了新的虚函数，则该函数的地址也将被添加到 vtbl 中（参见图 13.5）。注意，无论类中包含的虚函数是 1 个还是 10 个，都只需要在对象中添加 1 个地址成员，只是表的大小不同而已。

```
class Scientist{
{
    ...
    char name[40];
public:
    virtual void show_name();
    virtual void show_all();
    ...
};
class Physicist : public Scientist
{
    ...
    char field[40];
public;
    void show_all(); // redefined
    virtual void show_field(); // new
    ...
};
```

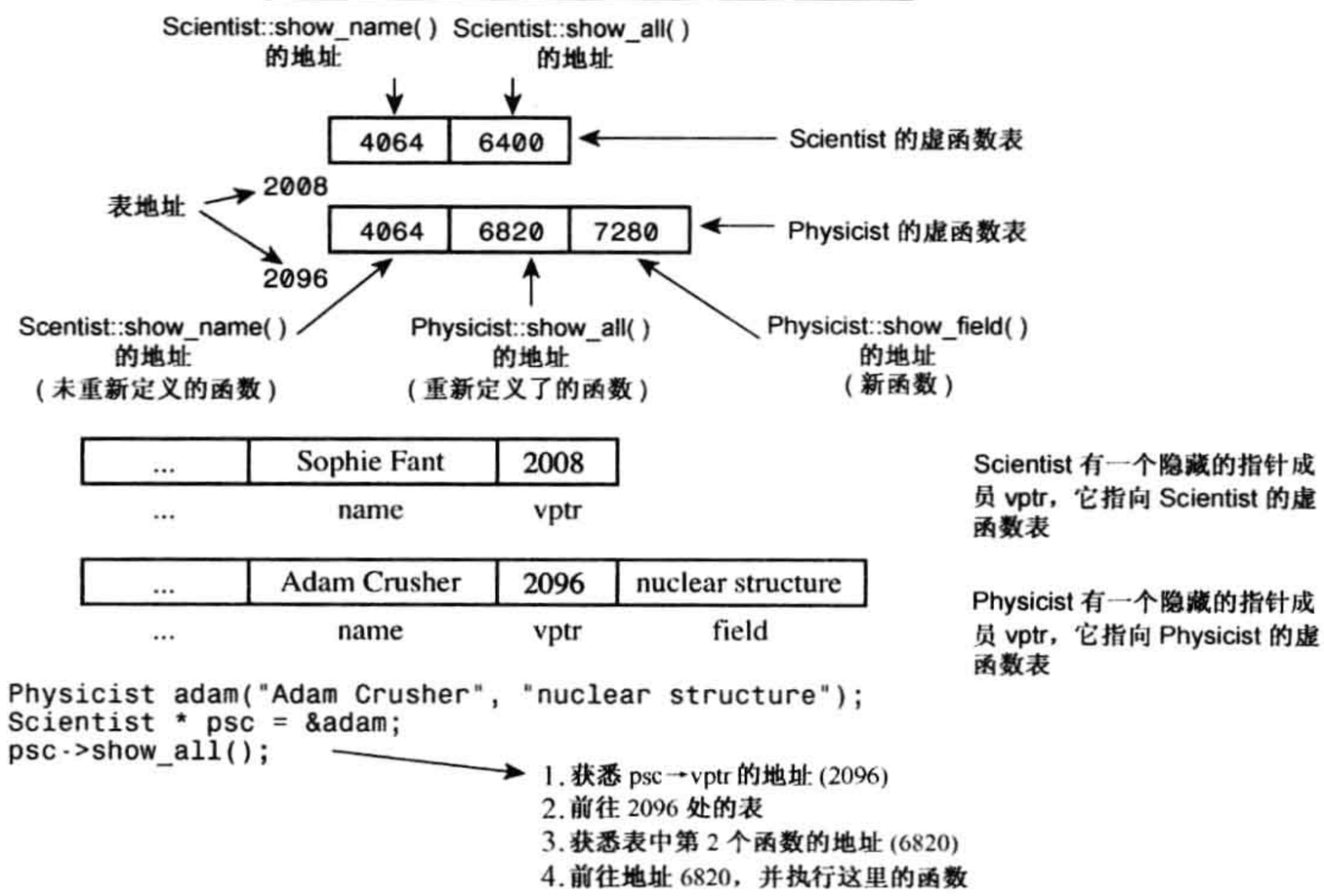


图 13.5 一种虚函数机制

调用虚函数时，程序将查看存储在对象中的 vtbl 地址，然后转向相应的函数地址表。如果使用类声明中定义的第一个虚函数，则程序将使用数组中的第一个函数地址，并执行具有该地址的函数。如果使用类声明中的第三个虚函数，程序将使用地址为数组中第三个元素的函数。

总之，使用虚函数时，在内存和执行速度方面有一定的成本，包括：

- 每个对象都将增大，增大量为存储地址的空间；
- 对于每个类，编译器都创建一个虚函数地址表（数组）；
- 对于每个函数调用，都需要执行一项额外的操作，即到表中查找地址。

虽然非虚函数的效率比虚函数稍高，但不具备动态联编功能。

13.4.3　有关虚函数注意事项

我们已经讨论了虚函数的一些要点。

- 在基类方法的声明中使用关键字 virtual 可使该方法在基类以及所有的派生类（包括从派生类派生出来的类）中是虚的。
- 如果使用指向对象的引用或指针来调用虚方法，程序将使用为对象类型定义的方法，而不使用为引用或指针类型定义的方法。这称为动态联编或晚期联编。这种行为非常重要，因为这样基类指针或引用可以指向派生类对象。
- 如果定义的类将被用作基类，则应将那些要在派生类中重新定义的类方法声明为虚的。

对于虚方法，还需要了解其他一些知识，其中有的已经介绍过。下面来看看这些内容。

1. 构造函数

构造函数不能是虚函数。创建派生类对象时，将调用派生类的构造函数，而不是基类的构造函数，然后，派生类的构造函数将使用基类的一个构造函数，这种顺序不同于继承机制。因此，派生类不继承基类的构造函数，所以将类构造函数声明为虚的没什么意义。

2. 析构函数

析构函数应当是虚函数，除非类不用做基类。例如，假设 Employee 是基类，Singer 是派生类，并添加一个 char *成员，该成员指向由 new 分配的内存。当 Singer 对象过期时，必须调用~Singer( )析构函数来释放内存。

请看下面的代码：

```
Employee * pe = new Singer; // legal because Employee is base for Singer
...
delete pe;                  // ~Employee() or ~Singer()?
```

如果使用默认的静态联编，delete 语句将调用~Employee( )析构函数。这将释放由 Singer 对象中的 Employee 部分指向的内存，但不会释放新的类成员指向的内存。但如果析构函数是虚的，则上述代码将先调用~Singer 析构函数释放由 Singer 组件指向的内存，然后，调用～Employee( )析构函数来释放由 Employee 组件指向的内存。

这意味着，即使基类不需要显式析构函数提供服务，也不应依赖于默认构造函数，而应提供虚析构函数，即使它不执行任何操作：

```
virtual ~BaseClass() { }
```

顺便说一句，给类定义一个虚析构函数并非错误，即使这个类不用做基类；这只是一个效率方面的问题。

提示：通常应给基类提供一个虚析构函数，即使它并不需要析构函数。

3. 友元

友元不能是虚函数，因为友元不是类成员，而只有成员才能是虚函数。如果由于这个原因引起了设计

问题，可以通过让友元函数使用虚成员函数来解决。

4. 没有重新定义

如果派生类没有重新定义函数，将使用该函数的基类版本。如果派生类位于派生链中，则将使用最新的虚函数版本，例外的情况是基类版本是隐藏的（稍后将介绍）。

5. 重新定义将隐藏方法

假设创建了如下所示的代码：

```
class Dwelling
{
public:
      virtual void showperks(int a) const;
...
};
class Hovel : public Dwelling
{
public:
      virtual void showperks() const;
...
};
```

这将导致问题，可能会出现类似于下面这样的编译器警告：

```
Warning: Hovel::showperks(void) hides Dwelling::showperks(int)
```

也可能不会出现警告。但不管结果怎样，代码将具有如下含义：

```
Hovel trump;
trump.showperks();      // valid
trump.showperks(5);     // invalid
```

新定义将 showperks( )定义为一个不接受任何参数的函数。重新定义不会生成函数的两个重载版本，而是隐藏了接受一个 int 参数的基类版本。总之，重新定义继承的方法并不是重载。如果在派生类中重新定义函数，将不是使用相同的函数特征标覆盖基类声明，而是隐藏同名的基类方法，不管参数特征标如何。

这引出了两条经验规则：第一，如果重新定义继承的方法，应确保与原来的原型完全相同，但如果返回类型是基类引用或指针，则可以修改为指向派生类的引用或指针（这种例外是新出现的）。这种特性被称为返回类型协变（covariance of return type），因为允许返回类型随类类型的变化而变化：

```
class Dwelling
{
public:
// a base method
      virtual Dwelling & build(int n);
      ...
};
class Hovel : public Dwelling
{
public:
// a derived method with a covariant return type
      virtual Hovel & build(int n);  // same function signature
      ...
};
```

注意，这种例外只适用于返回值，而不适用于参数。

第二，如果基类声明被重载了，则应在派生类中重新定义所有的基类版本。

```
class Dwelling
{
public:
// three overloaded showperks()
      virtual void showperks(int a) const;
      virtual void showperks(double x) const;
      virtual void showperks() const;
      ...
};
class Hovel : public Dwelling
{
public:
// three redefined showperks()
      virtual void showperks(int a) const;
      virtual void showperks(double x) const;
      virtual void showperks() const;
      ...
};
```

如果只重新定义一个版本，则另外两个版本将被隐藏，派生类对象将无法使用它们。注意，如果不需要修改，则新定义可只调用基类版本：

```
void Hovel::showperks() const {Dwelling::showperks();}
```

13.5　访问控制：protected

到目前为止，本书的类示例已经使用了关键字 public 和 private 来控制对类成员的访问。还存在另一个访问类别，这种类别用关键字 protected 表示。关键字 protected 与 private 相似，在类外只能用公有类成员来访问 protected 部分中的类成员。private 和 protected 之间的区别只有在基类派生的类中才会表现出来。派生类的成员可以直接访问基类的保护成员，但不能直接访问基类的私有成员。因此，对于外部世界来说，保护成员的行为与私有成员相似；但对于派生类来说，保护成员的行为与公有成员相似。

例如，假如 Brass 类将 balance 成员声明为保护的：

```
class Brass
{
protected:
    double balance;
...
};
```

在这种情况下，BrassPlus 类可以直接访问 balance，而不需要使用 Brass 方法。例如，可以这样编写 BrassPlus::Withdraw( )的核心：

```
void BrassPlus::Withdraw(double amt)
{
    if (amt < 0)
        cout << "Withdrawal amount must be positive; "

             << "withdrawal canceled.\n";
    else if (amt <= balance)       // access balance directly
        balance -= amt;
    else if ( amt <= balance + maxLoan - owesBank)
    {
        double advance = amt - balance;
```

```
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit(advance);
        balance -= amt;
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
}
```

使用保护数据成员可以简化代码的编写工作，但存在设计缺陷。例如，继续以 BrassPlus 为例，如果 balance 是受保护的，则可以按下面的方式编写代码：

```
void BrassPlus::Reset(double amt)
{
    balance = amt;
}
```

Brass 类被设计成只能通过 Deposit( )和 Withdraw( )才能修改 balance。但对于 BrassPlus 对象，Reset( )方法将忽略 Withdraw( )中的保护措施，实际上使 balance 成为公有变量，。

警告：最好对类数据成员采用私有访问控制，不要使用保护访问控制；同时通过基类方法使派生类能够访问基类数据。

然而，对于成员函数来说，保护访问控制很有用，它让派生类能够访问公众不能使用的内部函数。

13.6 抽象基类

至此，介绍了简单继承和较复杂的多态继承。接下来更为复杂的是抽象基类（abstract base class，ABC）。我们来看一些可使用 ABC 的编程情况。

有时候，使用 is-a 规则并不是看上去的那样简单。例如，假设您正在开发一个图形程序，该程序会显示圆和椭圆等。圆是椭圆的一个特殊情况——长轴和短轴等长的椭圆。因此，所有的圆都是椭圆，可以从 Ellipse 类派生出 Circle 类。但涉及到细节时，将发现很多问题。

首先考虑 Ellipse 类包含的内容。数据成员可以包括椭圆中心的坐标、半长轴（长轴的一半）、短半轴（短轴的一半）以及方向角（水平坐标轴与长轴之间的角度）。另外，还可以包括一些移动椭圆、返回椭圆面积、旋转椭圆以及缩放长半轴和短半轴的方法：

```
class Ellipse
{
private:
    double x;     // x-coordinate of the ellipse's center
    double y;     // y-coordinate of the ellipse's center
    double a;     // semimajor axis
    double b;     // semiminor axis
    double angle; // orientation angle in degrees
    ...
public:
    ...
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual double Area() const { return 3.14159 * a * b; }
    virtual void Rotate(double nang) { angle += nang; }
    virtual void Scale(double sa, double sb)  { a *= sa; b *= sb; }
    ...
};
```

现在假设从 Ellipse 类派生出一个 Circle 类：

```
class Circle : public Ellipse
{
    ...
};
```

虽然圆是一种椭圆，但是这种派生是笨拙的。例如，圆只需要一个值（半径）就可以描述大小和形状，并不需要有长半轴（a）和短半轴（b）。Circle 构造函数可以通过将同一个值赋给成员 a 和 b 来照顾这种情况，但将导致信息冗余。angle 参数和 Rotate( )方法对圆来说没有实际意义；而 Scale( )方法（顾名思义）会将两个轴作不同的缩放，将圆变成椭圆。可以使用一些技巧来修正这些问题，例如在 Circle 类中的私有部分包含重新定义的 Rotate( )方法，使 Rotate( )不能以公有方式用于圆。但总的来说，不使用继承，直接定义 Circle 类更简单：

```
class Circle      // no inheritance
{
private:
    double x;     // x-coordinate of the circle's center
    double y;     // y-coordinate of the circle's center
    double r;     // radius
    ...
public:
    ...
    void Move(int nx, ny) { x = nx; y = ny; }
    double Area() const { return 3.14159 * r * r; }
    void Scale(double sr)  { r *= sr; }
    ...
};
```

现在，类只包含所需的成员。但这种解决方法的效率也不高。Circle 和 Ellipse 类有很多共同点，将它们分别定义则忽略了这一事实。

还有一种解决方法，即从 Ellipse 和 Circle 类中抽象出它们的共性，将这些特性放到一个 ABC 中。然后从该 ABC 派生出 Circle 和 Ellipse 类。这样，便可以使用基类指针数组同时管理 Circle 和 Ellipse 对象，即可以使用多态方法）。在这个例子中，这两个类的共同点是中心坐标、Move( )方法（对于这两个类是相同的）和 Area( )方法（对于这两个类来说，是不同的）。确实，甚至不能在 ABC 中实现 Area( )方法，因为它没有包含必要的数据成员。C++通过使用纯虚函数（pure virtual function）提供未实现的函数。纯虚函数声明的结尾处为=0，参见 Area( )方法：

```
class BaseEllipse  // abstract base class
{
private:
    double x;   // x-coordinate of center
    double y;   // y-coordinate of center
    ...
public:
    BaseEllipse(double x0 = 0, double y0 = 0) : x(x0),y(y0) {}
    virtual ~BaseEllipse() {}
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual double Area() const = 0; // a pure virtual function
   ...
}
```

当类声明中包含纯虚函数时，则不能创建该类的对象。这里的理念是，包含纯虚函数的类只用作基类。要成为真正的 ABC，必须至少包含一个纯虚函数。原型中的=0 使虚函数成为纯虚函数。这里的方法 Area( )没有定义，但 C++甚至允许纯虚函数有定义。例如，也许所有的基类方法都与 Move( )一样，可以在基类

中进行定义，但您仍需要将这个类声明为抽象的。在这种情况下，可以将原型声明为虚的：

```
void Move(int nx, ny) = 0;
```

这将使基类成为抽象的，但您仍可以在实现文件中提供方法的定义：

```
void BaseEllipse::Move(int nx, ny) { x = nx; y = ny; }
```

总之，在原型中使用=0 指出类是一个抽象基类，在类中可以不定义该函数。

现在，可以从 BaseEllipse 类派生出 Ellips 类和 Circle 类，添加所需的成员来完成每个类。需要注意的一点是，Circle 类总是表示圆，而 Ellipse 类总是表示椭圆——也可以是圆。然而，Ellipse 类圆可被重新缩放为非圆，而 Ciecle 类圆必须始终为圆。

使用这些类的程序将能够创建 Ellipse 对象和 Circle 对象，但是不能创建 BaseEllipse 对象。由于 Circle 和 Ellipse 对象的基类相同，因此可以用 BaseEllipse 指针数组同时管理这两种对象。像 Circle 和 Ellipse 这样的类有时被称为具体（concrete）类，这表示可以创建这些类型的对象。

总之，ABC 描述的是至少使用一个纯虚函数的接口，从 ABC 派生出的类将根据派生类的具体特征，使用常规虚函数来实现这种接口。

13.6.1 应用 ABC 概念

您可能希望看到一个完整的 ABC 示例，因此这里将这一概念用于 Brass 和 BrassPlus 账户，首先定义一个名为 AcctABC 的 ABC。这个类包含 Brass 和 BrassPlus 类共有的所有方法和数据成员，而那些在 BrassPlus 类和 Brass 类中的行为不同的方法应被声明为虚函数。至少应有一个虚函数是纯虚函数，这样才能使 AcctABC 成为抽象类。

程序清单 13.11 的头文件声明了 AcctABC 类（ABC）、Brass 类和 BrassPlus 类（两者都是具体类）。为帮助派生类访问基类数据，AcctABC 提供了一些保护方法；派生类方法可以调用这些方法，但它们并不是派生类对象的公有接口的组成部分。AcctABC 还提供一个保护成员函数，用于处理格式化（以前是使用非成员函数处理的）。另外，AcctABC 类还有两个纯虚函数，所以它确实是抽象类。

程序清单 13.11 acctabc.h

```
// acctabc.h  -- bank account classes
#ifndef ACCTABC_H_
#define ACCTABC_H_
#include <iostream>
#include <string>

// Abstract Base Class
class AcctABC
{
private:
    std::string fullName;
    long acctNum;
    double balance;
protected:
    struct Formatting
    {
        std::ios_base::fmtflags flag;
        std::streamsize pr;
    };
    const std::string & FullName() const {return fullName;}
    long AcctNum() const {return acctNum;}
    Formatting SetFormat() const;
    void Restore(Formatting & f) const;
public:
```

```
    AcctABC(const std::string & s = "Nullbody", long an = -1,
                double bal = 0.0);
    void Deposit(double amt) ;
    virtual void Withdraw(double amt) = 0; // pure virtual function
    double Balance() const {return balance;};
    virtual void ViewAcct() const = 0;     // pure virtual function
    virtual ~AcctABC() {}
};

// Brass Account Class
class Brass :public AcctABC
{
public:
    Brass(const std::string & s = "Nullbody", long an = -1,
          double bal = 0.0) : AcctABC(s, an, bal) { }
    virtual void Withdraw(double amt);
    virtual void ViewAcct() const;
    virtual ~Brass() {}
};
//Brass Plus Account Class
class BrassPlus : public AcctABC
{
private:
    double maxLoan;
    double rate;
    double owesBank;
public:
    BrassPlus(const std::string & s = "Nullbody", long an = -1,
            double bal = 0.0, double ml = 500,
            double r = 0.10);
    BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
    virtual void ViewAcct()const;
    virtual void Withdraw(double amt);
    void ResetMax(double m) { maxLoan = m; }
    void ResetRate(double r) { rate = r; };
    void ResetOwes() { owesBank = 0; }
};
#endif
```

接下来需要实现那些不是内联函数的方法，如程序清单 13.12 所示。

程序清单 13.12　acctABC.cpp

```
// acctabc.cpp -- bank account class methods
#include <iostream>
#include "acctabc.h"
using std::cout;
using std::ios_base;
using std::endl;
using std::string;

// Abstract Base Class
AcctABC::AcctABC(const string & s, long an, double bal)
{
    fullName = s;
    acctNum = an;
```

```
    balance = bal;
}

void AcctABC::Deposit(double amt)
{
    if (amt < 0)
        cout << "Negative deposit not allowed; "
             << "deposit is cancelled.\n";
    else
        balance += amt;
}

void AcctABC::Withdraw(double amt)
{
    balance -= amt;
}

// protected methods for formatting
AcctABC::Formatting AcctABC::SetFormat() const
{
 // set up ###.## format
    Formatting f;
    f.flag =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    f.pr = cout.precision(2);
    return f;
}

void AcctABC::Restore(Formatting & f) const
{
    cout.setf(f.flag, ios_base::floatfield);
    cout.precision(f.pr);
}

// Brass methods
void Brass::Withdraw(double amt)
{
    if (amt < 0)
        cout << "Withdrawal amount must be positive; "
             << "withdrawal canceled.\n";
    else if (amt <= Balance())
        AcctABC::Withdraw(amt);
    else
        cout << "Withdrawal amount of $" << amt
             << " exceeds your balance.\n"
             << "Withdrawal canceled.\n";
}

void Brass::ViewAcct() const
{

    Formatting f = SetFormat();
    cout << "Brass Client: " << FullName() << endl;
    cout << "Account Number: " << AcctNum() << endl;
    cout << "Balance: $" << Balance() << endl;
```

```
    Restore(f);
}

// BrassPlus Methods
BrassPlus::BrassPlus(const string & s, long an, double bal,
          double ml, double r) : AcctABC(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
          : AcctABC(ba)   // uses implicit copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

void BrassPlus::ViewAcct() const
{
    Formatting f = SetFormat();

    cout << "BrassPlus Client: " << FullName() << endl;
    cout << "Account Number: " << AcctNum() << endl;
    cout << "Balance: $" << Balance() << endl;
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout.precision(3);
    cout << "Loan Rate: " << 100 * rate << "%\n";
    Restore(f);
}

void BrassPlus::Withdraw(double amt)
{
    Formatting f = SetFormat();

    double bal = Balance();
    if (amt <= bal)
        AcctABC::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit(advance);
        AcctABC::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
    Restore(f);
}
```

保护方法 FullName( )和 AcctNum( )提供了对数据成员 fullName 和 acctNum 的只读访问，使得可以进一步定制每个派生类的 ViewAcct( )。

这个版本在设置输出格式方面做了两项改进。前一个版本使用两个函数调用来设置输出格式，并使用一个函数调用来恢复格式：

```
format initialState = setFormat();
precis prec = cout.precision(2);
...
restore(initialState, prec); // restore original format
```

这个版本定义了一个结构，用于存储两项格式设置；并使用该结构来设置和恢复格式，因此只需两个函数调用：

```
struct Formatting
{
     std::ios_base::fmtfglas flag;
     std::streamsize pr;
};
...
Formatting f = SetFormat();
...
Restore(f);
```

因此代码更整洁。

旧版本存在的问题是，setFormat( )和 restore( )都是独立的函数，这些函数与客户定义的同名函数发生冲突。解决这种问题的方式有多种，一种方式是将这些函数声明为静态的，这样它们将归文件 brass.cpp 及其继任 acctabc.cpp 私有。另一种方式是，将这些函数以及结构 Formatting 放在一个独立的名称空间中。但这个示例探讨的主题之一是保护访问权限，因此将这些结构和函数放在了类定义的保护部分。这使得它们对基类和派生类可用，同时向外隐藏了它们。

对于 Brass 和 BrassPlus 账户的这种新实现，使用方式与旧实现相同，因为类方法的名称和接口都与以前一样。例如，为使程序清单 13.10 能够使用新的实现，需要采取下面的步骤将 usebrass2.cpp 转换为 usebrass3.cpp：

- 使用 acctabc.cpp 而不是 brass.cpp 来链接 usebrass2.cpp。
- 包含文件 acctabc.h，而不是 brass.h。
- 将下面的代码：

```
Brass * p_clients[CLIENTS];
```

替换为：

```
AcctABC * p_clients[CLIENTS];
```

程序清单 13.13 是修改后的文件，并将其重命名为 usebrass3.cpp。

程序清单 13.13　usebrass3.cpp

```
// usebrass3.cpp -- polymorphic example using an abstract base class
// compile with acctacb.cpp
#include <iostream>
#include <string>
#include "acctabc.h"
const int CLIENTS = 4;

int main()
{
   using std::cin;
   using std::cout;
```

```
using std::endl;

AcctABC * p_clients[CLIENTS];
std::string temp;
long tempnum;
double tempbal;
char kind;

for (int i = 0; i < CLIENTS; i++)
{
    cout << "Enter client's name: ";
    getline(cin,temp);
    cout << "Enter client's account number: ";
    cin >> tempnum;
    cout << "Enter opening balance: $";
    cin >> tempbal;
    cout << "Enter 1 for Brass Account or "
         << "2 for BrassPlus Account: ";
    while (cin >> kind && (kind != '1' && kind != '2'))
        cout <<"Enter either 1 or 2: ";
    if (kind == '1')
        p_clients[i] = new Brass(temp, tempnum, tempbal);
    else
    {
        double tmax, trate;
        cout << "Enter the overdraft limit: $";
        cin >> tmax;
        cout << "Enter the interest rate "
             << "as a decimal fraction: ";
        cin >> trate;
        p_clients[i] = new BrassPlus(temp, tempnum, tempbal,
                                     tmax, trate);
    }
    while (cin.get() != '\n')
        continue;
}
cout << endl;
for (int i = 0; i < CLIENTS; i++)
{
    p_clients[i]->ViewAcct();
    cout << endl;
}

for (int i = 0; i < CLIENTS; i++)
{
    delete p_clients[i];  // free memory
}
cout << "Done.\n";

return 0;
}
```

该程序本身的行为与非抽象基类版本相同，因此如果输入与给程序清单 13.10 提供的输入相同，输出也将相同。

13.6.2 ABC 理念

在处理继承的问题上，RatedPlayer 示例使用的方法比较随意，而 ABC 方法比它更具系统性、更规范。设计 ABC 之前，首先应开发一个模型——指出编程问题所需的类以及它们之间相互关系。一种学院派思想认为，如果要设计类继承层次，则只能将那些不会被用作基类的类设计为具体的类。这种方法的设计更清晰，复杂程度更低。

可以将 ABC 看作是一种必须实施的接口。ABC 要求具体派生类覆盖其纯虚函数——迫使派生类遵循 ABC 设置的接口规则。这种模型在基于组件的编程模式中很常见，在这种情况下，使用 ABC 使得组件设计人员能够制定“接口约定”，这样确保了从 ABC 派生的所有组件都至少支持 ABC 指定的功能。

13.7 继承和动态内存分配

继承是怎样与动态内存分配（使用 new 和 delete）进行互动的呢？例如，如果基类使用动态内存分配，并重新定义赋值和复制构造函数，这将怎样影响派生类的实现呢？这个问题的答案取决于派生类的属性。如果派生类也使用动态内存分配，那么就需要学习几个新的小技巧。下面来看看这两种情况。

13.7.1 第一种情况：派生类不使用 new

假设基类使用了动态内存分配：

```
//  Base Class Using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
...
};
```

声明中包含了构造函数使用 new 时需要的特殊方法：析构函数、复制构造函数和重载赋值运算符。

现在，从 baseDMA 派生出 lackDMA 类，而后者不使用 new，也未包含其他一些不常用的、需要特殊处理的设计特性：

```
// derived class without DMA
class lacksDMA :public baseDMA
{
private:
    char color[40];
public:
...
};
```

是否需要为 lackDMA 类定义显式析构函数、复制构造函数和赋值运算符呢？不需要。

首先，来看是否需要析构函数。如果没有定义析构函数，编译器将定义一个不执行任何操作的默认构造函数。实际上，派生类的默认构造函数总是要进行一些操作：执行自身的代码后调用基类析构函数。因为我们假设 lackDMA 成员不需执行任何特殊操作，所以默认析构函数是合适的。

接着来看复制构造函数。第 12 章介绍过，默认复制构造函数执行成员复制，这对于动态内存分配来说是不合适的，但对于新的 lacksDMA 成员来说是合适的。因此只需考虑继承的 baseDMA 对象。要知道，成员复制将根据数据类型采用相应的复制方式，因此，将 long 复制到 long 中是通过使用常规赋值完成的；但复制类成员或继承的类组件时，则是使用该类的复制构造函数完成的。所以，lacksDMA 类的默认复制构造函数使用显式 baseDMA 复制构造函数来复制 lacksDMA 对象的 baseDMA 部分。因此，默认复制构造函数对于新的 lacksDMA 成员来说是合适的，同时对于继承的 baseDMA 对象来说也是合适的。

对于赋值来说，也是如此。类的默认赋值运算符将自动使用基类的赋值运算符来对基类组件进行赋值。因此，默认赋值运算符也是合适的。

派生类对象的这些属性也适用于本身是对象的类成员。例如，第 10 章介绍过，实现 Stock 类时，可以使用 string 对象而不是 char 数组来存储公司名称。标准 string 类和本书前面创建的 String 类一样，也采用动态内存分配。现在，读者知道了为何这不会引发问题。Stock 的默认复制构造函数将使用 string 的复制构造函数来复制对象的 company 成员；Stock 的默认赋值运算符将使用 string 的赋值运算符给对象的 company 成员赋值；而 Stock 的析构函数（默认或其他析构函数）将自动调用 string 的析构函数。

13.7.2　第二种情况：派生类使用 new

假设派生类使用了 new：

```
// derived class with DMA
class hasDMA :public baseDMA
{
private:
    char * style;  // use new in constructors
public:
...
};
```

在这种情况下，必须为派生类定义显式析构函数、复制构造函数和赋值运算符。下面依次考虑这些方法。

派生类析构函数自动调用基类的析构函数，故其自身的职责是对派生类构造函数执行工作的进行清理。因此，hasDMA 析构函数必须释放指针 style 管理的内存，并依赖于 baseDMA 的析构函数来释放指针 label 管理的内存。

```
baseDMA::~baseDMA()  // takes care of baseDMA stuff
{
   delete [] label;
}

hasDMA::~hasDMA()       // takes care of hasDMA stuff
{
    delete [] style;
}
```

接下来看复制构造函数。BaseDMA 的复制构造函数遵循用于 char 数组的常规模式，即使用 strlen( )来获悉存储 C-风格字符串所需的空间、分配足够的内存（字符数加上存储空字符所需的 1 字节）并使用函数 strcpy( )将原始字符串复制到目的地：

```
baseDMA::baseDMA(const baseDMA & rs)
{
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
}
```

hasDMA 复制构造函数只能访问 hasDMA 的数据，因此它必须调用 baseDMA 复制构造函数来处理共享的 baseDMA 数据：

```
hasDMA::hasDMA(const hasDMA & hs)
        : baseDMA(hs)
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}
```

需要注意的一点是，成员初始化列表将一个 hasDMA 引用传递给 baseDMA 构造函数。没有参数类型为 hasDMA 引用的 baseDMA 构造函数，也不需要这样的构造函数。因为复制构造函数 baseDMA 有一个 baseDMA 引用参数，而基类引用可以指向派生类型。因此，baseDMA 复制构造函数将使用 hasDMA 参数的 baseDMA 部分来构造新对象的 baseDMA 部分。

接下来看赋值运算符。BaseDMA 赋值运算符遵循下述常规模式：

```
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
    if (this == &rs)
        return *this;
    delete [] label;
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
    return *this;
}
```

由于 hasDMA 也使用动态内存分配，所以它也需要一个显式赋值运算符。作为 hasDMA 的方法，它只能直接访问 hasDMA 的数据。然而，派生类的显式赋值运算符必须负责所有继承的 baseDMA 基类对象的赋值，可以通过显式调用基类赋值运算符来完成这项工作，如下所示：

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;
    baseDMA::operator=(hs);  // copy base portion
    delete [] style;         // prepare for new style
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
    return *this;
}
```

下述语句看起来有点奇怪：

```
baseDMA::operator=(hs);  // copy base portion
```

但通过使用函数表示法，而不是运算符表示法，可以使用作用域解析运算符。实际上，该语句的含义如下：

```
*this = hs;  // use baseDMA::operator=()
```

当然编译器将忽略注释，所以使用后面的代码时，编译器将使用 hasDMA ::operator=( )，从而形成递归调用。使用函数表示法使得赋值运算符被正确调用。

总之，当基类和派生类都采用动态内存分配时，派生类的析构函数、复制构造函数、赋值运算符都必须使用相应的基类方法来处理基类元素。这种要求是通过三种不同的方式来满足的。对于析构函数，这是自动完成的；对于构造函数，这是通过在初始化成员列表中调用基类的复制构造函数来完成的；如果不这样做，将自动调用基类的默认构造函数。对于赋值运算符，这是通过使用作用域解析运算符显式地调用基类的赋值运算符来完成的。

13.7.3　使用动态内存分配和友元的继承示例

为演示这些有关继承和动态内存分配的概念，我们将刚才介绍过的 baseDMA、lacksDMA 和 hasDMA 类集成到一个示例中。程序清单 13.14 是这些类的头文件。除了前面介绍的内容外，这个头文件还包含一个友元函数，以说明派生类如何访问基类的友元。

程序清单 13.14　dma.h

```
// dma.h  -- inheritance and dynamic memory allocation
#ifndef DMA_H_
#define DMA_H_
#include <iostream>

//  Base Class Using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
    friend std::ostream & operator<<(std::ostream & os,
                                     const baseDMA & rs);
};

// derived class without DMA
// no destructor needed
// uses implicit copy constructor
// uses implicit assignment operator
class lacksDMA :public baseDMA
{
private:
    enum { COL_LEN = 40};
    char color[COL_LEN];
public:
    lacksDMA(const char * c = "blank", const char * l = "null",
             int r = 0);
    lacksDMA(const char * c, const baseDMA & rs);
    friend std::ostream & operator<<(std::ostream & os,
                                     const lacksDMA & rs);
};
// derived class with DMA
class hasDMA :public baseDMA
{
private:
    char * style;
public:
    hasDMA(const char * s = "none", const char * l = "null",
           int r = 0);
    hasDMA(const char * s, const baseDMA & rs);
```

```
    hasDMA(const hasDMA & hs);
    ~hasDMA();
    hasDMA & operator=(const hasDMA & rs);
    friend std::ostream & operator<<(std::ostream & os,
                                     const hasDMA & rs);
};

#endif
```

程序清单 13.15 列出了类 baseDMA、lackDMA 和 hasDMA 的方法定义。

程序清单 13.15　dma.cpp

```
// dma.cpp --dma class methods

#include "dma.h"
#include <cstring>

// baseDMA methods
baseDMA::baseDMA(const char * l, int r)
{
    label = new char[std::strlen(l) + 1];
    std::strcpy(label, l);
    rating = r;
}

baseDMA::baseDMA(const baseDMA & rs)
{
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
}

baseDMA::~baseDMA()
{
    delete [] label;
}
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
    if (this == &rs)
        return *this;
    delete [] label;
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
    return *this;
}

std::ostream & operator<<(std::ostream & os, const baseDMA & rs)
{
    os << "Label: " << rs.label << std::endl;
    os << "Rating: " << rs.rating << std::endl;
    return os;
}

// lacksDMA methods
```

```
lacksDMA::lacksDMA(const char * c, const char * l, int r)
    : baseDMA(l, r)
{
    std::strncpy(color, c, 39);
    color[39] = '\0';
}

lacksDMA::lacksDMA(const char * c, const baseDMA & rs)
    : baseDMA(rs)
{
    std::strncpy(color, c, COL_LEN - 1);
    color[COL_LEN - 1] = '\0';
}

std::ostream & operator<<(std::ostream & os, const lacksDMA & ls)
{
    os << (const baseDMA &) ls;
    os << "Color: " << ls.color << std::endl;
    return os;
}

// hasDMA methods
hasDMA::hasDMA(const char * s, const char * l, int r)
        : baseDMA(l, r)
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}
hasDMA::hasDMA(const char * s, const baseDMA & rs)
        : baseDMA(rs)
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}

hasDMA::hasDMA(const hasDMA & hs)
        : baseDMA(hs)  // invoke base class copy constructor
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}

hasDMA::~hasDMA()
{
    delete [] style;
}

hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;
    baseDMA::operator=(hs);  // copy base portion
    delete [] style;         // prepare for new style
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
```

```
    return *this;
}

std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << std::endl;
    return os;
}
```

在程序清单 13.14 和程序清单 13.15 中，需要注意的新特性是，派生类如何使用基类的友元。例如，请考虑下面这个 hasDMa 类的友元：

```
friend std::ostream & operator<<(std::ostream & os,
                                 const hasDMA & rs);
```

作为 hasDMA 类的友元，该函数能够访问 style 成员。然而，还存在一个问题：该函数如不是 baseDMA 类的友元，那它如何访问成员 lable 和 rating 呢？答案是使用 baseDMA 类的友元函数 operator<<( )。下一个问题是，因为友元不是成员函数，所以不能使用作用域解析运算符来指出要使用哪个函数。这个问题的解决方法是使用强制类型转换，以便匹配原型时能够选择正确的函数。因此，代码将参数 const hasDMA & 转换成类型为 const baseDMA &的参数：

```
std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
//  type cast to match operator<<(ostream & , const baseDMA &)
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << endl;
    return os;
}
```

程序清单 13.16 是一个测试类 baseDMA、lackDMA 和 hasDMA 的小程序。

程序清单 13.16　usedma.cpp

```
// usedma.cpp -- inheritance, friends, and DMA
// compile with dma.cpp
#include <iostream>
#include "dma.h"
int main()
{
    using std::cout;
    using std::endl;

    baseDMA shirt("Portabelly", 8);
    lacksDMA balloon("red", "Blimpo", 4);
    hasDMA map("Mercator", "Buffalo Keys", 5);
    cout << "Displaying baseDMA object:\n";
    cout << shirt << endl;
    cout << "Displaying lacksDMA object:\n";
    cout << balloon << endl;
    cout << "Displaying hasDMA object:\n";
    cout << map << endl;
    lacksDMA balloon2(balloon);
    cout << "Result of lacksDMA copy:\n";
    cout << balloon2 << endl;
    hasDMA map2;
    map2 = map;
```

```
    cout << "Result of hasDMA assignment:\n";
    cout << map2 << endl;
    return 0;
}
```

程序清单 13.14～程序清单 13.16 组成的程序的输出如下：

```
Displaying baseDMA object:
Label: Portabelly
Rating: 8

Displaying lacksDMA object:
Label: Blimpo
Rating: 4
Color: red

Displaying hasDMA object:
Label: Buffalo Keys
Rating: 5
Style: Mercator

Result of lacksDMA copy:
Label: Blimpo
Rating: 4
Color: red

Result of hasDMA assignment:
Label: Buffalo Keys
Rating: 5
Style: Mercator
```

13.8　类设计回顾

C++可用于解决各种类型的编程问题，但不能将类设计简化成带编号的例程。然而，有些常用的指导原则，下面复习并拓展前面的讨论，以介绍这些原则。

13.8.1　编译器生成的成员函数

第 12 章介绍过，编译器会自动生成一些公有成员函数——特殊成员函数。这表明这些特殊成员函数很重要，下面回顾其中的一些。

1. 默认构造函数

默认构造函数要么没有参数，要么所有的参数都有默认值。如果没有定义任何构造函数，编译器将定义默认构造函数，让您能够创建对象。例如，假设 Star 是一个类，则下述代码需要使用默认构造函数：

```
Star rigel;         // create an object without explicit initialization
Star pleiades[6];   // create an array of objects
```

自动生成的默认构造函数的另一项功能是，调用基类的默认构造函数以及调用本身是对象的成员所属类的默认构造函数。

另外，如果派生类构造函数的成员初始化列表中没有显式调用基类构造函数，则编译器将使用基类的默认构造函数来构造派生类对象的基类部分。在这种情况下，如果基类没有构造函数，将导致编译阶段错误。

如果定义了某种构造函数，编译器将不会定义默认构造函数。在这种情况下，如果需要默认构造函数，则必须自己提供。

提供构造函数的动机之一是确保对象总能被正确地初始化。另外，如果类包含指针成员，则必须初始化这些成员。因此，最好提供一个显式默认构造函数，将所有的类数据成员都初始化为合理的值。

2. 复制构造函数

复制构造函数接受其所属类的对象作为参数。例如，Star 类的复制构造函数的原型如下：

```
Star(const Star &);
```

在下述情况下，将使用复制构造函数：

- 将新对象初始化为一个同类对象；
- 按值将对象传递给函数；
- 函数按值返回对象；
- 编译器生成临时对象。

如果程序没有使用（显式或隐式）复制构造函数，编译器将提供原型，但不提供函数定义；否则，程序将定义一个执行成员初始化的复制构造函数。也就是说，新对象的每个成员都被初始化为原始对象相应成员的值。如果成员为类对象，则初始化该成员时，将使用相应类的复制构造函数。

在某些情况下，成员初始化是不合适的。例如，使用 new 初始化的成员指针通常要求执行深复制（参见 baseDMA 类示例），或者类可能包含需要修改的静态变量。在上述情况下，需要定义自己的复制构造函数。

3. 赋值运算符

默认的赋值运算符用于处理同类对象之间的赋值。不要将赋值与初始化混淆了。如果语句创建新的对象，则使用初始化；如果语句修改已有对象的值，则是赋值：

```
Star sirius;
Star alpha = sirius;      // initialization (one notation)
Star dogstar;
dogstar = sirius;         // assignment
```

默认赋值为成员赋值。如果成员为类对象，则默认成员赋值将使用相应类的赋值运算符。如果需要显式定义复制构造函数，则基于相同的原因，也需要显式定义赋值运算符。Star 类的赋值运算符的原型如下：

```
Star & Star::operator=(const Star &);
```

赋值运算符函数返回一个 Star 对象引用。baseDMA 类演示了一个典型的显式赋值运算符函数示例。

编译器不会生成将一种类型赋给另一种类型的赋值运算符。如果希望能够将字符串赋给 Star 对象，则方法之一是显式定义下面的运算符：

```
Star & Star::operator=(const char *) {...}
```

另一种方法是使用转换函数（参见下一节中的“转换”小节）将字符串转换成 Star 对象，然后使用将 Star 赋给 Star 的赋值函数。第一种方法的运行速度较快，但需要的代码较多，而使用转换函数可能导致编译器出现混乱。

第 18 章将讨论 C++11 新增的两个特殊方法：移动构造函数和移动赋值运算符。

13.8.2 其他的类方法

定义类时，还需要注意其他几点。下面的几小节将分别介绍。

1. 构造函数

构造函数不同于其他类方法，因为它创建新的对象，而其他类方法只是被现有的对象调用。这是构造函数不被继承的原因之一。继承意味着派生类对象可以使用基类的方法，然而，构造函数在完成其工作之前，对象并不存在。

2. 析构函数

一定要定义显式析构函数来释放类构造函数使用 new 分配的所有内存，并完成类对象所需的任何特殊的清理工作。对于基类，即使它不需要析构函数，也应提供一个虚析构函数。

3. 转换

使用一个参数就可以调用的构造函数定义了从参数类型到类类型的转换。例如，下述 Star 类的构造函数原型：

```
Star(const char *);                        // converts char * to Star
Star(const Spectral &, int members = 1); // converts Spectral to Star
```

将可转换的类型传递给以类为参数的函数时，将调用转换构造函数。例如，在如下代码中：

```
Star north;
north = "polaris";
```

第二条语句将调用 Star::operator = （const Star *）函数，使用 Star::star（const char *）生成一个 Star 对象，该对象将被用作上述赋值运算符函数的参数。这里假设没有定义将 char *赋给 Star 的赋值运算符。

在带一个参数的构造函数原型中使用 explicit 将禁止进行隐式转换，但仍允许显式转换：

```
class Star
{
...
public:
    explicit Star(const char *);
...
};
...
Star north;
north = "polaris";         // not allowed
north = Star("polaris");  // allowed
```

要将类对象转换为其他类型，应定义转换函数（参见第 11 章）。转换函数可以是没有参数的类成员函数，也可以是返回类型被声明为目标类型的类成员函数。即使没有声明返回类型，函数也应返回所需的转换值。下面是一些示例：

```
Star::Star double() {...}           // converts star to double
Star::Star const char * () {...}    // converts to const char
```

应理智地使用这样的函数，仅当它们有帮助时才使用。另外，对于某些类，包含转换函数将增加代码的二义性。例如，假设已经为第 11 章的 Vector 类型定义了 double 转换，并编写了下面的代码：

```
Vector ius(6.0, 0.0);
Vector lux = ius + 20.2;         // ambiguous
```

编译器可以将 ius 转换成 double 并使用 double 加法，或将 20.2 转换成 veotor（使用构造函数之一）并使用 vector 加法。但除了指出二义性外，它什么也不做。

C++11 支持将关键字 explicit 用于转换函数。与构造函数一样，explicit 允许使用强制类型转换进行显式转换，但不允许隐式转换。

4. 按值传递对象与传递引用

通常，编写使用对象作为参数的函数时，应按引用而不是按值来传递对象。这样做的原因之一是为了提高效率。按值传递对象涉及到生成临时拷贝，即调用复制构造函数，然后调用析构函数。调用这些函数需要时间，复制大型对象比传递引用花费的时间要多得多。如果函数不修改对象，应将参数声明为 const 引用。

按引用传递对象的另外一个原因是，在继承使用虚函数时，被定义为接受基类引用参数的函数可以接受派生类。这在本章前面介绍过（同时请参见本章后面的“虚方法”一节）。

5. 返回对象和返回引用

有些类方法返回对象。您可能注意到了，有些成员函数直接返回对象，而另一些则返回引用。有时方法必须返回对象，但如果可以不返回对象，则应返回引用。来具体看一下。

首先，在编码方面，直接返回对象与返回引用之间唯一的区别在于函数原型和函数头：

```
Star nova1(const Star &);      // returns a Star object
Star & nova2(const Star &);    // returns a reference to a Star
```

其次，应返回引用而不是返回对象的的原因在于，返回对象涉及生成返回对象的临时副本，这是调用函数的程序可以使用的副本。因此，返回对象的时间成本包括调用复制构造函数来生成副本所需的时间和调用析构函数删除副本所需的时间。返回引用可节省时间和内存。直接返回对象与按值传递对象相似：它们都生成临时副本。同样，返回引用与按引用传递对象相似：调用和被调用的函数对同一个对象进行操作。

然而，并不总是可以返回引用。函数不能返回在函数中创建的临时对象的引用，因为当函数结束时，临时对象将消失，因此这种引用将是非法的。在这种情况下，应返回对象，以生成一个调用程序可以使用的副本。

通用的规则是，如果函数返回在函数中创建的临时对象，则不要使用引用。例如，下面的方法使用构造函数来创建一个新对象，然后返回该对象的副本：

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

如果函数返回的是通过引用或指针传递给它的对象，则应按引用返回对象。例如，下面的代码按引用返回调用函数的对象或作为参数传递给函数的对象：

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
       return s;            // argument object
    else
       return *this;        // invoking object
}
```

6. 使用 const

使用 const 时应特别注意。可以用它来确保方法不修改参数：

```
Star::Star(const char * s) {...} // won't change the string to which s points
```

可以使用 const 来确保方法不修改调用它的对象：

```
void Star::show() const {...} // won't change invoking object
```

这里 const 表示 const Star * this，而 this 指向调用的对象。

通常，可以将返回引用的函数放在赋值语句的左侧，这实际上意味着可以将值赋给引用的对象。但可以使用 const 来确保引用或指针返回的值不能用于修改对象中的数据：

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
       return s;            // argument object
    else
       return *this;        // invoking object
}
```

该方法返回对 this 或 s 的引用。因为 this 和 s 都被声明为 const，所以函数不能对它们进行修改，这意味着返回的引用也必须被声明为 const。

注意，如果函数将参数声明为指向 const 的引用或指针，则不能将该参数传递给另一个函数，除非后者也确保了参数不会被修改。

13.8.3　公有继承的考虑因素

通常，在程序中使用继承时，有很多问题需要注意。下面来看其中的一些问题。

1. is-a 关系

要遵循 is-a 关系。如果派生类不是一种特殊的基类，则不要使用公有派生。例如，不应从 Brain 类派生出 Programmer 类。如果要指出程序员有大脑，应将 Brain 类对象作为 Programmer 类的成员。

在某些情况下，最好的方法可能是创建包含纯虚函数的抽象数据类，并从它派生出其他的类。

请记住，表示 is-a 关系的方式之一是，无需进行显式类型转换，基类指针就可以指向派生类对象，基类引用可以引用派生类对象。另外，反过来是行不通的，即不能在不进行显式类型转换的情况下，将派生类指针或引用指向基类对象。这种显式类型转换（向下强制转换）可能有意义，也可能没有，这取决于类声明（参见图 13.4）。

2. 什么不能被继承

构造函数是不能继承的，也就是说，创建派生类对象时，必须调用派生类的构造函数。然而，派生类构造函数通常使用成员初始化列表语法来调用基类构造函数，以创建派生对象的基类部分。如果派生类构造函数没有使用成员初始化列表语法显式调用基类构造函数，将使用基类的默认构造函数。在继承链中，每个类都可以使用成员初始化列表将信息传递给相邻的基类。C++11 新增了一种让您能够继承构造函数的机制，但默认仍不继承构造函数。

析构函数也是不能继承的。然而，在释放对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数。如果基类有默认析构函数，编译器将为派生类生成默认析构函数。通常，对于基类，其析构函数应设置为虚的。

赋值运算符是不能继承的，原因很简单。派生类继承的方法的特征标与基类完全相同，但赋值运算符的特征标随类而异，这是因为它包含一个类型为其所属类的形参。赋值运算符确实有一些有趣的特征，下面介绍它们。

3. 赋值运算符

如果编译器发现程序将一个对象赋给同一个类的另一个对象，它将自动为这个类提供一个赋值运算符。这个运算符的默认或隐式版本将采用成员赋值，即将原对象的相应成员赋给目标对象的每个成员。然而，如果对象属于派生类，编译器将使用基类赋值运算符来处理派生对象中基类部分的赋值。如果显式地为基类提供了赋值运算符，将使用该运算符。与此相似，如果成员是另一个类的对象，则对于该成员，将使用其所属类的赋值运算符。

正如多次提到的，如果类构造函数使用 new 来初始化指针，则需要提供一个显式赋值运算符。因为对于派生对象的基类部分，C++将使用基类的赋值运算符，所以不需要为派生类重新定义赋值运算符，除非它添加了需要特别留意的数据成员。例如，baseDMA 类显式地定义了赋值，但派生类 lackDMA 使用为它生成的隐式赋值运算符。

然而，如果派生类使用了 new，则必须提供显式赋值运算符。必须给类的每个成员提供赋值运算符，而不仅仅是新成员。HasDMA 类演示了如何完成这项工作：

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;
    baseDMA::operator=(hs);  // copy base portion
    delete [] style;         // prepare for new style
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
    return *this;
}
```

将派生类对象赋给基类对象将会如何呢？（注意，这不同于将基类引用初始化为派生类对象。）请看下面的例子：

```
Brass blips;                                                  // base class
BrassPlus snips("Rafe Plosh", 91191,3993.19, 600.0, 0.12); // derived class
blips = snips;                        // assign derived object to base object
```

这将使用哪个赋值运算符呢？赋值语句将被转换成左边的对象调用的一个方法：

```
blips.operator=(snips);
```

其中左边的对象是 Brass 对象，因此它将调用 Brass ::operator =（const Brass &）函数。is-a 关系允许 Brass 引用指向派生类对象，如 Snips。赋值运算符只处理基类成员，所以上述赋值操作将忽略 Snips 的 maxLoan 成员和其他 BrassPlus 成员。总之，可以将派生对象赋给基类对象，但这只涉及基类的成员。

相反的操作将如何呢？即可以将基类对象赋给派生类对象吗？请看下面的例子：

```
Brass gp("Griff Hexbait", 21234, 1200);    // base class
BrassPlus temp;                            // derived class
temp = gp;    // possible?
```

上述赋值语句将被转换为如下所示：

```
temp.operator=(gp);
```

左边的对象是 BrassPlus 对象，所以它调用 BrassPlus ::operator=（const BrassPlus &）函数。然而，派生类引用不能自动引用基类对象，因此上述代码不能运行，除非有下面的转换构造函数：

```
BrassPlus(const Brass &);
```

与 BrassPlus 类的情况相似，转换构造函数可以接受一个类型为基类的参数和其他参数，条件是其他参数有默认值：

```
BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
```

如果有转换构造函数，程序将通过它根据 gp 来创建一个临时 BrassPlus 对象，然后将它用作赋值运算符的参数。

另一种方法是，定义一个用于将基类赋给派生类的赋值运算符：

```
BrassPlus & BrassPlus ::operator=(const Brass &) {...}
```

该赋值运算符的类型与赋值语句完全匹配，因此无需进行类型转换。

总之，问题“是否可以将基类对象赋给派生对象？”的答案是“也许”。如果派生类包含了这样的构造函数，即对将基类对象转换为派生类对象进行了定义，则可以将基类对象赋给派生对象。如果派生类定义了用于将基类对象赋给派生对象的赋值运算符，则也可以这样做。如果上述两个条件都不满足，则不能这样做，除非使用显式强制类型转换。

4. 私有成员与保护成员

对派生类而言，保护成员类似于公有成员；但对于外部而言，保护成员与私有成员类似。派生类可以直接访问基类的保护成员，但只能通过基类的成员函数来访问私有成员。因此，将基类成员设置为私有的可以提高安全性，而将它们设置为保护成员则可简化代码的编写工作，并提高访问速度。Stroustrup 在其《The Design and Evolution of C++》一书中指出，使用私用数据成员比使用保护数据成员更好，但保护方法很有用。

5. 虚方法

设计基类时，必须确定是否将类方法声明为虚的。如果希望派生类能够重新定义方法，则应在基类中将方法定义为虚的，这样可以启用晚期联编（动态联编）；如果不希望重新定义方法，则不必将其声明为虚的，这样虽然无法禁止他人重新定义方法，但表达了这样的意思：您不希望它被重新定义。

请注意，不适当的代码将阻止动态联编。例如，请看下面的两个函数：

```
void show(const Brass & rba)
{
    rba.ViewAcct();
    cout << endl;
}
void inadequate(Brass ba)
{
    ba.ViewAcct();
    cout << endl;
}
```

第一个函数按引用传递对象，第二个按值传递对象。

现在，假设将派生类参数传递给上述两个函数：

```
BrassPlus buzz("Buzz Parsec", 00001111, 4300);
show(buzz);
inadequate(buzz);
```

show( )函数调用使 rba 参数成为 BrassPlus 对象 buzz 的引用，因此，rba.ViewAcct( )被解释为 BrassPlus 版本，正如应该的那样。但在 inadequate( )函数中（它是按值传递对象的），ba 是 Brass（const Brass &）构造函数创建的一个 Brass 对象（自动向上强制转换使得构造函数参数可以引用一个 BrassPlus 对象）。因此，在 inadequate( )中，ba.ViewAcct( )是 Brass 版本，所以只有 buzz 的 Brass 部分被显示。

6. 析构函数

正如前面介绍的，基类的析构函数应当是虚的。这样，当通过指向对象的基类指针或引用来删除派生对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数，而不仅仅是调用基类的析构函数。

7. 友元函数

由于友元函数并非类成员，因此不能继承。然而，您可能希望派生类的友元函数能够使用基类的友元函数。为此，可以通过强制类型转换将，派生类引用或指针转换为基类引用或指针，然后使用转换后的指针或引用来调用基类的友元函数：

```
ostream & operator<<(ostream & os, const hasDMA & hs)
{
//  type cast to match operator<<(ostream & , const baseDMA &)
    os << (const baseDMA &) hs;
    os << "Style: " << hs.style << endl;
    return os;
}
```

也可以使用第 15 章将讨论的运算符 dynamic_cast<>来进行强制类型转换：

```
os << dynamic_cast<const baseDMA &> (hs);
```

鉴于第 15 章将讨论的原因，这是更佳的强制类型转换方式。

8. 有关使用基类方法的说明

以公有方式派生的类的对象可以通过多种方式来使用基类的方法。

- 派生类对象自动使用继承而来的基类方法，如果派生类没有重新定义该方法。
- 派生类的构造函数自动调用基类的构造函数。
- 派生类的构造函数自动调用基类的默认构造函数，如果没有在成员初始化列表中指定其他构造函数。
- 派生类构造函数显式地调用成员初始化列表中指定的基类构造函数。
- 派生类方法可以使用作用域解析运算符来调用公有的和受保护的基类方法。
- 派生类的有元函数可以通过强制类型转换，将派生类引用或指针转换为基类引用或指针，然后使

用该引用或指针来调用基类的友元函数。

13.8.4 类函数小结

C++类函数有很多不同的变体，其中有些可以继承，有些不可以。有些运算符函数既可以是成员函数，也可以是友元，而有些运算符函数只能是成员函数。表 13.1（摘自《The Annotated C++ Reference Manual》）总结了这些特征，其中 op=表示诸如+=、*=等格式的赋值运算符。注意，op=运算符的特征与“其他运算符”类别并没有区别。单独列出 op=旨在指出这些运算符与=运算符的行为是不同的。

表 13.1 成员函数属性

函数	能否继承	成员还是友元	默认能否生成	能否为虚函数	是否可以有返回类型
构造函数	否	成员	能	否	否
析构函数	否	成员	能	能	否
=	否	成员	能	能	能
&	能	任意	能	能	能
转换函数	能	成员	否	能	否
( )	能	成员	否	能	能
[]	能	成员	否	能	能
->	能	成员	否	能	能
op=	能	任意	否	能	能
new	能	静态成员	否	否	void*
delete	能	静态成员	否	否	void
其他运算符	能	任意	否	能	能
其他成员	能	成员	否	能	能
友元	否	友元	否	否	能

13.9 总结

继承通过使用已有的类（基类）定义新的类（派生类），使得能够根据需要修改编程代码。公有继承建立 is-a 关系，这意味着派生类对象也应该是某种基类对象。作为 is-a 模型的一部分，派生类继承基类的数据成员和大部分方法，但不继承基类的构造函数、析构函数和赋值运算符。派生类可以直接访问基类的公有成员和保护成员，并能够通过基类的公有方法和保护方法访问基类的私有成员。可以在派生类中新增数据成员和方法，还可以将派生类用作基类，来做进一步的开发。每个派生类都必须有自己的构造函数。程序创建派生类对象时，将首先调用基类的构造函数，然后调用派生类的构造函数；程序删除对象时，将首先调用派生类的析构函数，然后调用基类的析构函数。

如果要将类用作基类，则可以将成员声明为保护的，而不是私有的，这样，派生类将可以直接访问这些成员。然而，使用私有成员通常可以减少出现编程问题的可能性。如果希望派生类可以重新定义基类的方法，则可以使用关键字 virtual 将它声明为虚的。这样对于通过指针或引用访问的对象，能够根据对象类型来处理，而不是根据引用或指针的类型来处理。具体地说，基类的析构函数通常应当是虚的。

可以考虑定义一个 ABC：只定义接口，而不涉及实现。例如，可以定义抽象类 Shape，然后使用它派生出具体的形状类，如 Circle 和 Square。ABC 必须至少包含一个纯虚方法，可以在声明中的分号前面加上=0 来声明纯虚方法。

```
virtual double area() const = 0;
```

不一定非得定义纯虚方法。对于包含纯虚成员的类，不能使用它来创建对象。纯虚方法用于定义派生类的通用接口。

13.10　复习题

1．派生类从基类那里继承了什么？

2．派生类不能从基类那里继承什么？

3．假设 baseDMA ::operator=( )函数的返回类型为 void，而不是 baseDMA &，这将有什么后果？如果返回类型为 baseDMA，而不是 baseDMA &，又将有什么后果？

4．创建和删除派生类对象时，构造函数和析构函数调用的顺序是怎样的？

5．如果派生类没有添加任何数据成员，它是否需要构造函数？

6．如果基类和派生类定义了同名的方法，当派生类对象调用该方法时，被调用的将是哪个方法？

7．在什么情况下，派生类应定义赋值运算符？

8．可以将派生类对象的地址赋给基类指针吗？可以将基类对象的地址赋给派生类指针吗？

9．可以将派生类对象赋给基类对象吗？可以将基类对象赋给派生类对象吗？

10．假设定义了一个函数，它将基类对象的引用作为参数。为什么该函数也可以将派生类对象作为参数？

11．假设定义了一个函数，它将基类对象作为参数（即函数按值传递基类对象）。为什么该函数也可以将派生类对象作为参数？

12．为什么通常按引用传递对象比按值传递对象的效率更高？

13．假设 Corporation 是基类，PublicCorporation 是派生类。再假设这两个类都定义了 head( )函数，ph 是指向 Corporation 类型的指针，且被赋给了一个 PublicCorporation 对象的地址。如果基类将 head( )定义为：

a．常规非虚方法；

b．虚方法；

则 ph->head( )将被如何解释？

14．下述代码有什么问题？

```
class Kitchen
{
private:
    double kit_sq_ft;
public:
    Kitchen() {kit_sq_ft = 0.0; }
    virtual double area() const { return kit_sq_ft * kit_sq_ft; }
};
class House : public Kitchen
{
private:
    double all_sq_ft;
public:
    House() {all_sq_ft += kit_sq_ft;}
    double area(const char *s) const { cout << s; return all_sq_ft; }
};
```

13.11　编程练习

1．以下面的类声明为基础：

```
// base class
class Cd {  // represents a CD disk
```

```
private:
    char performers[50];
    char label[20];
    int selections;   // number of selections
    double playtime;  // playing time in minutes
public:
    Cd(char * s1, char * s2, int n, double x);
    Cd(const Cd & d);
    Cd();
    ~Cd();
    void Report() const;  // reports all CD data
    Cd & operator=(const Cd & d);
};
```

派生出一个 Classic 类，并添加一组 char 成员，用于存储指出 CD 中主要作品的字符串。修改上述声明，使基类的所有函数都是虚的。如果上述定义声明的某个方法并不需要，则请删除它。使用下面的程序测试您的产品：

```
#include <iostream>
using namespace std;
#include "classic.h"     // which will contain #include cd.h
void Bravo(const Cd & disk);
int main()
{
    Cd c1("Beatles", "Capitol", 14, 35.5);
    Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C",
                         "Alfred Brendel", "Philips", 2, 57.17);
    Cd *pcd = &c1;

    cout << "Using object directly:\n";
    c1.Report();    // use Cd method
    c2.Report();    // use Classic method

    cout << "Using type cd * pointer to objects:\n";
    pcd->Report();  // use Cd method for cd object
    pcd = &c2;
    pcd->Report();  // use Classic method for classic object

    cout << "Calling a function with a Cd reference argument:\n";
    Bravo(c1);
    Bravo(c2);

    cout << "Testing assignment: ";
    Classic copy;
    copy = c2;
    copy.Report()

    return 0;
}

void Bravo(const Cd & disk)
{
    disk.Report();
}
```

2．完成练习 1，但让两个类使用动态内存分配而不是长度固定的数组来记录字符串。

3．修改 baseDMA-lacksDMA-hasDMA 类层次，让三个类都从一个 ABC 派生而来，然后使用与程序清

单 13.10 相似的程序对结果进行测试。也就是说，它应使用 ABC 指针数组，并让用户决定要创建的对象类型。在类定义中添加 virtual View( )方法以处理数据显示。

4．Benevolent Order of Programmers 用来维护瓶装葡萄酒箱。为描述它，BOP Portmaster 设置了一个 Port 类，其声明如下：

```
#include <iostream>
using namespace std;
class Port
{
private:
    char * brand;
    char style[20]; // i.e., tawny, ruby, vintage
    int bottles;
public:
    Port(const char * br = "none", const char * st = "none", int b = 0);
    Port(const Port & p);                // copy constructor
    virtual ~Port() { delete [] brand; }
    Port & operator=(const Port & p);
    Port & operator+=(int b);            // adds b to bottles
    Port & operator-=(int b);            // subtracts b from bottles, if
available
    int BottleCount() const { return bottles; }
    virtual void Show() const;
    friend ostream & operator<<(ostream & os, const Port & p);
};
```

show( )方法按下面的格式显示信息：

```
Brand: Gallo
Kind: tawny
Bottles: 20
```

operator<<( )函数按下面的格式显示信息（末尾没有换行符）：

```
Gallo, tawny, 20
```

PortMaster 完成了 Port 类的方法定义后派生了 VintagePort 类，然后被解职——因为不小心将一瓶 45 度 Cockburn 泼到了正在准备烤肉调料的人身上，VintagePort 类如下所示：

```
class VintagePort : public Port // style necessarily = "vintage"
{
private:
    char * nickname;            // i.e., "The Noble" or "Old Velvet", etc.
    int year;                   // vintage year
public:
    VintagePort();
    VintagePort(const char * br, int b, const char * nn, int y);
    VintagePort(const VintagePort & vp);
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator=(const VintagePort & vp);
    void Show() const;
    friend ostream & operator<<(ostream & os, const VintagePort & vp);
};
```

您被指定负责完成 VintagePort。

a．第一个任务是重新创建 Port 方法定义，因为前任被开除时销毁了方法定义。

b．第二个任务是解释为什么有的方法重新定义了，而有些没有重新定义。

c．第三个任务是解释为何没有将 operator=( )和 operator<<( )声明为虚的。

d．第四个任务是提供 VintagePort 中各个方法的定义。

第 14 章　C++中的代码重用

本章内容包括：

- has-a 关系。
- 包含对象成员的类。
- 模板类 valarray。
- 私有和保护继承。
- 多重继承。
- 虚基类。
- 创建类模板。
- 使用类模板。
- 模板的具体化。

C++的一个主要目标是促进代码重用。公有继承是实现这种目标的机制之一，但并不是唯一的机制。本章将介绍其他方法，其中之一是使用这样的类成员：本身是另一个类的对象。这种方法称为包含（containment）、组合（composition）或层次化（layering）。另一种方法是使用私有或保护继承。通常，包含、私有继承和保护继承用于实现 has-a 关系，即新的类将包含另一个类的对象。例如，HomeTheater 类可能包含一个 BluRayPlayer 对象。多重继承使得能够使用两个或更多的基类派生出新的类，将基类的功能组合在一起。

第 10 章介绍了函数模板，本章将介绍类模板——另一种重用代码的方法。类模板使我们能够使用通用术语定义类，然后使用模板来创建针对特定类型定义的特殊类。例如，可以定义一个通用的栈模板，然后使用该模板创建一个用于表示 int 值栈的类和一个用于表示 double 值栈的类，甚至可以创建一个这样的类，即用于表示由栈组成的栈。

14.1　包含对象成员的类

首先介绍包含对象成员的类。有一些类（如 string 类和第 16 章将介绍的标准 C++类模板）为表示类中的组件提供了方便的途径。下面来看一个具体的例子。

学生是什么？入学者？参加研究的人？残酷现实社会的避难者？有姓名和一系列考试分数的人？显然，最后一个定义完全没有表示出人的特征，但非常适合于简单的计算机表示。因此，让我们根据该定义来开发 Student 类。

将学生简化成姓名和一组考试分数后，可以使用一个包含两个成员的类来表示它：一个成员用于表示姓名，另一个成员用于表示分数。对于姓名，可以使用字符数组来表示，但这将限制姓名的长度。当然，也可以使用 char 指针和动态内存分配，但正如第 12 章指出的，这将要求提供大量的支持代码。一种更好的方法是，使用一个由他人开发好的类的对象来表示。例如，可以使用一个 String 类（参见第 12 章）或标准 C++ string 类的对象来表示姓名。较简单的选择是使用 string 类，因为 C++库提供了这个类的所有实现

代码，且其实现更完美。要使用 String 类，您必须在项目中包含实现文件 string1.cpp。

对于考试分数，存在类似的选择。可以使用一个定长数组，这限制了数组的长度；可以使用动态内存分配并提供大量的支持代码；也可以设计一个使用动态内存分配的类来表示该数组；还可以在标准 C++库中查找一个能够表示这种数据的类。

自己开发这样的类一点问题也没有。开发简单的版本并不那么难，因为 double 数组与 char 数组有很多相似之处，因此可以根据 String 类来设计表示 double 数组的类。事实上，本书以前的版本就这样做过。

当然，如果 C++库提供了合适的类，实现起来将更简单。C++库确实提供了一个这样的类，它就是 valarray。

14.1.1　valarray 类简介

valarray 类是由头文件 valarray 支持的。顾名思义，这个类用于处理数值（或具有类似特性的类），它支持诸如将数组中所有元素的值相加以及在数组中找出最大和最小的值等操作。valarray 被定义为一个模板类，以便能够处理不同的数据类型。本章后面将介绍如何定义模板类，但就现在而言，您只需知道如何使用模板类即可。

模板特性意味着声明对象时，必须指定具体的数据类型。因此，使用 valarray 类来声明一个对象时，需要在标识符 valarray 后面加上一对尖括号，并在其中包含所需的数据类型：

```
valarray<int> q_values;      // an array of int
valarray<double> weights;    // an array of double
```

第 4 章介绍 vector 和 array 类时，您见过这种语法，它非常简单。这些类也可用于存储数字，但它们提供的算术支持没有 valarray 多。

这是您需要学习的唯一新语法，它非常简单。

类特性意味着要使用 valarray 对象，需要了解这个类的构造函数和其他类方法。下面是几个使用其构造函数的例子：

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;          // an array of double, size 0
valarray<int> v2(8);          // an array of 8 int elements
valarray<int> v3(10,8);       // an array of 8 int elements,
                              // each set to 10
valarray<double> v4(gpa, 4); // an array of 4 elements
             // initialized to the first 4 elements of gpa
```

从中可知，可以创建长度为零的空数组、指定长度的空数组、所有元素度被初始化为指定值的数组、用常规数组中的值进行初始化的数组。在 C++11 中，也可使用初始化列表：

```
valarray<int> v5 = {20, 32, 17, 9};  // C++11
```

下面是这个类的一些方法。

- operator[ ]( )：让您能够访问各个元素。
- size( )：返回包含的元素数。
- sum( )：返回所有元素的总和。
- max( )：返回最大的元素。
- min( )：返回最小的元素。

还有很多其他的方法，其中的一些将在第 16 章介绍；但就这个例子而言，上述方法足够了。

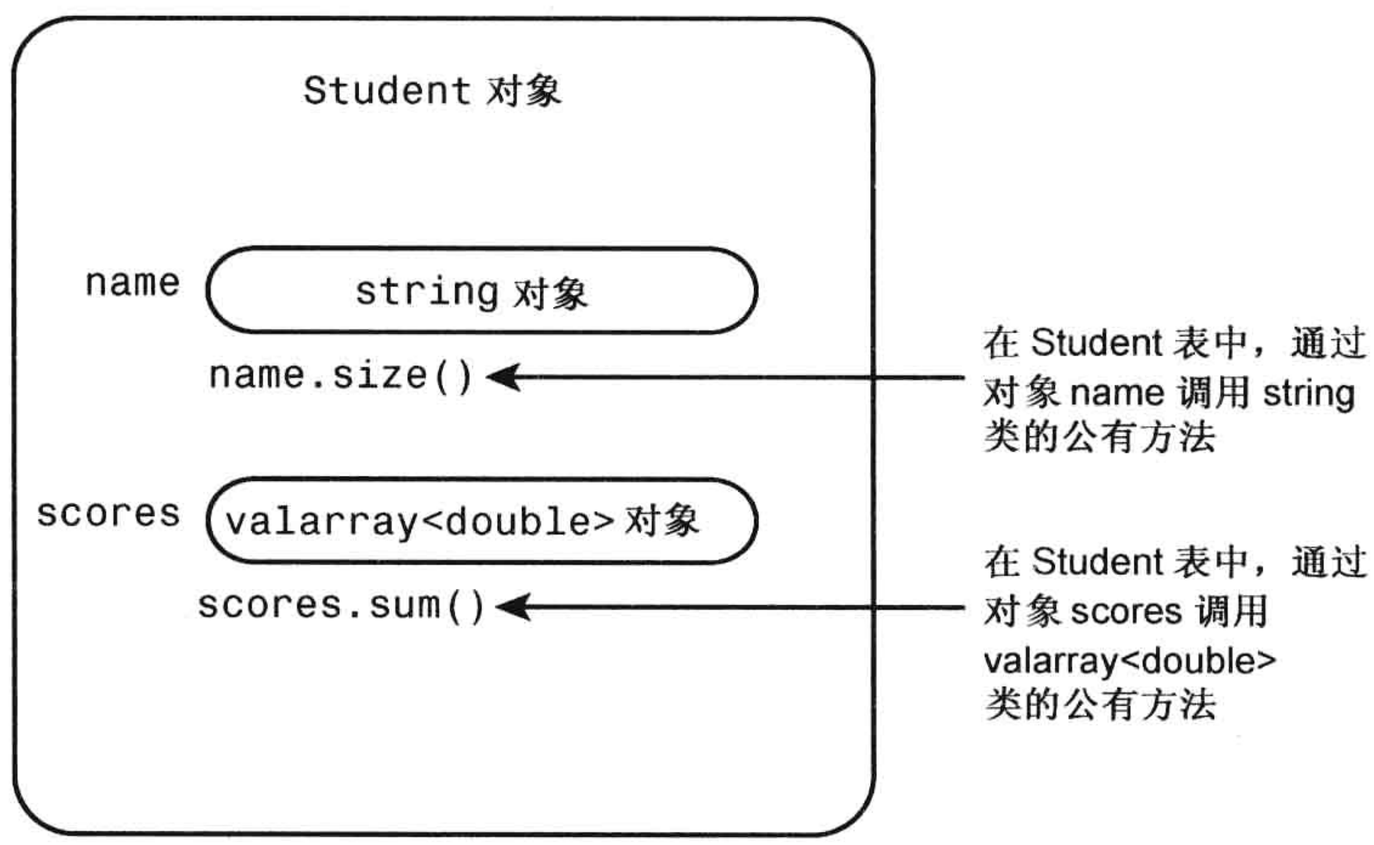
14.1.2　Student 类的设计

至此，已经确定了 Student 类的设计计划：使用一个 string 对象来表示姓名，使用一个 valarray<double> 来表示考试分数。那么如何设计呢？您可能想以公有的方式从这两个类派生出 Student 类，这将是多重公有继承，C++允许这样做，但在这里并不合适，因为学生与这些类之间的关系不是 is-a 模型。学生不

是姓名，也不是一组考试成绩。这里的关系是 has-a，学生有姓名，也有一组考试分数。通常，用于建立 has-a 关系的 C++技术是组合（包含），即创建一个包含其他类对象的类。例如，可以将 Student 类声明为如下所示：

```
class Student
{
private:
    string name;              // use a string object for name
    valarray<double> scores;  // use a valarray<double> object for scores
    ...
};
```

同样，上述类将数据成员声明为私有的。这意味着 Student 类的成员函数可以使用 string 和 valarray<double>类的公有接口来访问和修改 name 和 scores 对象，但在类的外面不能这样做，而只能通过 Student 类的公有接口访问 name 和 score（请参见图 14）。对于这种情况，通常被描述为 Student 类获得了其成员对象的实现，但没有继承接口。例如，Student 对象使用 string 的实现，而不是 char * name 或 char name [26]实现来保存姓名。但 Student 对象并不是天生就有使用函数 string operator+=( )的能力。




```
class Student
{
private
    string name;
    valarray<double> scores;
    …
};
```

图 14.1 对象中的对象：包含

接口和实现

使用公有继承时，类可以继承接口，可能还有实现（基类的纯虚函数提供接口，但不提供实现）。获得接口是 is-a 关系的组成部分。而使用组合，类可以获得实现，但不能获得接口。不继承接口是 has-a 关系的组成部分。

对于 has-a 关系来说，类对象不能自动获得被包含对象的接口是一件好事。例如，string 类将+运算符重载为将两个字符串连接起来；但从概念上说，将两个 Student 对象串接起来是没有意义的。这也是这里不使用公有继承的原因之一。另一方面，被包含的类的接口部分对新类来说可能是有意义的。例如，可能希望使用 string 接口中的 operator<( )方法将 Student 对象按姓名进行排序，为此可以定义 Student::Operator<( ) 成员函数，它在内部使用函数 string::Operator<( )。下面介绍一些细节。

14.1.3　Student 类示例

现在需要提供 Student 类的定义，当然它应包含构造函数以及一些用作 Student 类接口的方法。程序清单 14.1 是 Student 类的定义，其中所有构造函数都被定义为内联的；它还提供了一些用于输入和输出的友元函数。

程序清单 14.1　studentc.h

```
// studentc.h -- defining a Student class using containment
#ifndef STUDENTC_H_
#define STUDENTC_H_

#include <iostream>
#include <string>
#include <valarray>
class Student
{
private:
    typedef std::valarray<double> ArrayDb;
    std::string name;       // contained object
    ArrayDb scores;         // contained object
    // private method for scores output
    std::ostream & arr_out(std::ostream & os) const;
public:
    Student() : name("Null Student"), scores() {}
    explicit Student(const std::string & s)
        : name(s), scores() {}
    explicit Student(int n) : name("Nully"), scores(n) {}
    Student(const std::string & s, int n)
        : name(s), scores(n) {}
    Student(const std::string & s, const ArrayDb & a)
        : name(s), scores(a) {}
    Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
    ~Student() {}
    double Average() const;
    const std::string & Name() const;
    double & operator[](int i);
    double operator[](int i) const;
// friends
    // input
    friend std::istream & operator>>(std::istream & is,
                                     Student & stu);  // 1 word
    friend std::istream & getline(std::istream & is,
                                  Student & stu);     // 1 line
    // output
    friend std::ostream & operator<<(std::ostream & os,
                                     const Student & stu);
};

#endif
```

为简化表示，Student 类的定义中包含下述 typedef:

```
typedef std::valarray<double> ArrayDb;
```

这样，在以后的代码中便可以使用表示 ArrayDb，而不是 std::valarray<double>，因此类方法和友元函数可以使用 ArrayDb 类型。将该 typedef 放在类定义的私有部分意味着可以在 Student 类的实现中使用它，但在 Student 类外面不能使用。

请注意关键字 explicit 的用法：

```
explicit Student(const std::string & s)
    : name(s), scores() {}
explicit Student(int n) : name("Nully"), scores(n) {}
```

本书前面说过，可以用一个参数调用的构造函数将用作从参数类型到类类型的隐式转换函数；但这通常不是好主意。在上述第二个构造函数中，第一个参数表示数组的元素个数，而不是数组中的值，因此将一个构造函数用作 int 到 Student 的转换函数是没有意义的，所以使用 explicit 关闭隐式转换。如果省略该关键字，则可以编写如下所示的代码：

```
Student doh("Homer", 10);  // store "Homer", create array of 10 elements
doh = 5;   // reset name to "Nully", reset to empty array of 5 elements
```

在这里，马虎的程序员键入了 doh 而不是 doh[0]。如果构造函数省略了 explicit，则将使用构造函数调用 Student（5）将 5 转换为一个临时 Student 对象，并使用“Nully”来设置成员 name 的值。因此赋值操作将使用临时对象替换原来的 doh 值。使用了 explicit 后，编译器将认为上述赋值运算符是错误的。

C++和约束

C++包含让程序员能够限制程序结构的特性——使用 explicit 防止单参数构造函数的隐式转换，使用 const 限制方法修改数据，等等。这样做的根本原因是：在编译阶段出现错误优于在运行阶段出现错误。

1. 初始化被包含的对象

构造函数全都使用您熟悉的成员初始化列表语法来初始化 name 和 score 成员对象。在前面的一些例子中，构造函数用这种语法来初始化内置类型的成员：

```
Queue::Queue(int qs) : qsize(qs)  {...} // initialize qsize to qs
```

上述代码在成员初始化列表中使用的是数据成员的名称（qsize）。另外，前面介绍的示例中的构造函数还使用成员初始化列表初始化派生对象的基类部分：

```
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs) {...}
```

对于继承的对象，构造函数在成员初始化列表中使用类名来调用特定的基类构造函数。对于成员对象，构造函数则使用成员名。例如，请看程序清单 14.3 的最后一个构造函数：

```
Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
```

因为该构造函数初始化的是成员对象，而不是继承的对象，所以在初始化列表中使用的是成员名，而不是类名。初始化列表中的每一项都调用与之匹配的构造函数，即 name(str)调用构造函数 string(const char *)，scores(pd, n)调用构造函数 ArrayDb(const double *, int)。

如果不使用初始化列表语法，情况将如何呢？C++要求在构建对象的其他部分之前，先构建继承对象的所有成员对象。因此，如果省略初始化列表，C++将使用成员对象所属类的默认构造函数。

初始化顺序

当初始化列表包含多个项目时，这些项目被初始化的顺序为它们被声明的顺序，而不是它们在初始化列表中的顺序。例如，假设 Student 构造函数如下：

```
Student(const char * str, const double * pd, int n)
        : scores(pd, n),  name(str) {}
```

则 name 成员仍将首先被初始化，因为在类定义中它首先被声明。对于这个例子来说，初始化顺序并

不重要，但如果代码使用一个成员的值作为另一个成员的初始化表达式的一部分时，初始化顺序就非常重要了。

2. 使用被包含对象的接口

被包含对象的接口不是公有的，但可以在类方法中使用它。例如，下面的代码说明了如何定义一个返回学生平均分数的函数：

```
double Student::Average() const
{
    if (scores.size() > 0)
        return scores.sum()/scores.size();
    else
        return 0;
}
```

上述代码定义了可由 Student 对象调用的方法，该方法内部使用了 valarray 的方法 size( )和 sum( )。这是因为 scores 是一个 valarray 对象，所以它可以调用 valarray 类的成员函数。总之，Student 对象调用 Student 的方法，而后者使用被包含的 valarray 对象来调用 valarray 类的方法。

同样，可以定义一个使用 string 版本的<<运算符的友元函数：

```
// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    ...
}
```

因为 stu.name 是一个 string 对象，所以它将调用函数 operatot<<(ostream &, const string &)，该函数位于 string 类中。注意，operator<<(ostream & os, const Student & stu)必须是 Student 类的友元函数，这样才能访问 name 成员。另一种方法是，在该函数中使用公有方法 Name( )，而不是私有数据成员 name。

同样，该函数也可以使用 valarray 的<<实现来进行输出，不幸的是没有这样的实现；因此，Student 类定义了一个私有辅助方法来处理这种任务：

```
// private method
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = scores.size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << scores[i] << " ";
            if (i % 5 == 4)
                os << endl;
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}
```

通过使用这样的辅助方法，可以将零乱的细节放在一个地方，使得友元函数的编码更为整洁：

```
// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    stu.arr_out(os);  // use private method for scores
    return os;
}
```

辅助函数也可用作其他用户级输出函数的构建块——如果您选择提供这样的函数的话。

程序清单 14.2 是 Student 类的类方法文件，其中包含了让您能够使用[ ]运算符来访问 Student 对象中各项成绩的方法。

程序清单 14.2　student.cpp

```
// studentc.cpp -- Student class using containment
#include "studentc.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;
//public methods
double Student::Average() const
{
    if (scores.size() > 0)
        return scores.sum()/scores.size();
    else
        return 0;
}

const string & Student::Name() const
{
    return name;
}

double & Student::operator[](int i)
{
    return scores[i];          // use valarray<double>::operator[]()
}

double Student::operator[](int i) const
{
    return scores[i];
}

// private method
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = scores.size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << scores[i] << " ";
            if (i % 5 == 4)
                os << endl;
```

```
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}

// friends
// use string version of operator>>()
istream & operator>>(istream & is, Student & stu)
{
    is >> stu.name;
    return is;
}

// use string friend getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
    getline(is, stu.name);
    return is;
}

// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    stu.arr_out(os);  // use private method for scores
    return os;
}
```

除私有辅助方法外，程序清单 14.2 并没有新增多少代码。使用包含让您能够充分利用已有的代码。

3. 使用新的 Student 类

下面编写一个小程序来测试这个新的 Student 类。出于简化的目的，该程序将使用一个只包含 3 个 Student 对象的数组，其中每个对象保存 5 个考试成绩。另外还将使用一个不复杂的输入循环，该循环不验证输入，也不让用户中途退出。程序清单 14.3 列出了该测试程序，请务必将该程序与 Student.cpp 一起进行编译。

程序清单 14.3　use_stuc.cpp

```
// use_stuc.cpp -- using a composite class
// compile with studentc.cpp
#include <iostream>
#include "studentc.h"
using std::cin;
using std::cout;
using std::endl;

void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;

int main()
```

```
{
    Student ada[pupils] =
        {Student(quizzes), Student(quizzes), Student(quizzes)};

    int i;
    for (i = 0; i < pupils; ++i)
        set(ada[i], quizzes);
    cout << "\nStudent List:\n";
    for (i = 0; i < pupils; ++i)
        cout << ada[i].Name() << endl;
    cout << "\nResults:";
    for (i = 0; i < pupils; ++i)
    {
        cout << endl << ada[i];
        cout << "average: " << ada[i].Average() << endl;
    }
    cout << "Done.\n";
    return 0;
}

void set(Student & sa, int n)
{
    cout << "Please enter the student's name: ";
    getline(cin, sa);
    cout << "Please enter " << n << " quiz scores:\n";
    for (int i = 0; i < n; i++)
        cin >> sa[i];
    while (cin.get() != '\n')
        continue;
}
```

下面是程序清单 14.1～程序清单 14.3 组成的程序的运行情况：

```
Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
92 94 96 93 95
Please enter the student's name: Pat Roone
Please enter 5 quiz scores:
83 89 72 78 95
Please enter the student's name: Fleur O'Day
Please enter 5 quiz scores:
92 89 96 74 64
Student List:
Gil Bayts
Pat Roone
Fleur O'Day

Results:
Scores for Gil Bayts:
92 94 96 93 95
average: 94

Scores for Pat Roone:
83 89 72 78 95
average: 83.4
```

```
Scores for Fleur O'Day:
92 89 96 74 64
average: 83
Done.
```

14.2　私有继承

C++还有另一种实现 has-a 关系的途径——私有继承。使用私有继承，基类的公有成员和保护成员都将成为派生类的私有成员。这意味着基类方法将不会成为派生对象公有接口的一部分，但可以在派生类的成员函数中使用它们。

下面更深入地探讨接口问题。使用公有继承，基类的公有方法将成为派生类的公有方法。总之，派生类将继承基类的接口；这是 is-a 关系的一部分。使用私有继承，基类的公有方法将成为派生类的私有方法。总之，派生类不继承基类的接口。正如从被包含对象中看到的，这种不完全继承是 has-a 关系的一部分。

使用私有继承，类将继承实现。例如，如果从 String 类派生出 Student 类，后者将有一个 String 类组件，可用于保存字符串。另外，Student 方法可以使用 String 方法来访问 String 组件。

包含将对象作为一个命名的成员对象添加到类中，而私有继承将对象作为一个未被命名的继承对象添加到类中。我们将使用术语子对象（subobject）来表示通过继承或包含添加的对象。

因此私有继承提供的特性与包含相同：获得实现，但不获得接口。所以，私有继承也可以用来实现 has-a 关系。接下来介绍如何使用私有继承来重新设计 Student 类。

14.2.1　Student 类示例（新版本）

要进行私有继承，请使用关键字 private 而不是 public 来定义类（实际上，private 是默认值，因此省略访问限定符也将导致私有继承）。Student 类应从两个类派生而来，因此声明将列出这两个类：

```
class Student : private std::string, private std::valarray<double>
{
public:
    ...
};
```

使用多个基类的继承被称为多重继承（multiple inheritance，MI）。通常，MI 尤其是公有 MI 将导致一些问题，必须使用额外的语法规则来解决它们，这将在本章后面介绍。但在这个示例中，MI 不会导致问题。

新的 Student 类不需要私有数据，因为两个基类已经提供了所需的所有数据成员。包含版本提供了两个被显式命名的对象成员，而私有继承提供了两个无名称的子对象成员。这是这两种方法的第一个主要区别。

1. 初始化基类组件

隐式地继承组件而不是成员对象将影响代码的编写，因为再也不能使用 name 和 scores 来描述对象了，而必须使用用于公有继承的技术。例如，对于构造函数，包含将使这样的构造函数：

```
Student(const char * str, const double * pd, int n)
   : name(str), scores(pd, n) {}      // use object names for containment
```

对于继承类，新版本的构造函数将使用成员初始化列表语法，它使用类名而不是成员名来标识构造函数：

```
Student(const char * str, const double * pd, int n)
   : std::string(str), ArrayDb(pd, n) {}  // use class names for inheritance
```

在这里，ArrayDb 是 std::valarray<double>的别名。成员初始化列表使用 std::string(str)，而不是 name(str)。这是包含和私有继承之间的第二个主要区别。

程序清单 14.4 列出了新的类定义。唯一不同的地方是，省略了显式对象名称，并在内联构造函数中使用了类名，而不是成员名。

程序清单 14.4 studenti.h

```
// studenti.h -- defining a Student class using private inheritance
#ifndef STUDENTC_H_
#define STUDENTC_H_

#include <iostream>
#include <valarray>
#include <string>
class Student : private std::string, private std::valarray<double>
{
private:
    typedef std::valarray<double> ArrayDb;
    // private method for scores output
    std::ostream & arr_out(std::ostream & os) const;
public:
    Student() : std::string("Null Student"), ArrayDb() {}
    explicit Student(const std::string & s)
            : std::string(s), ArrayDb() {}
    explicit Student(int n) : std::string("Nully"), ArrayDb(n) {}
    Student(const std::string & s, int n)
            : std::string(s), ArrayDb(n) {}
    Student(const std::string & s, const ArrayDb & a)
            : std::string(s), ArrayDb(a) {}
    Student(const char * str, const double * pd, int n)
            : std::string(str), ArrayDb(pd, n) {}
    ~Student() {}
    double Average() const;
    double & operator[](int i);
    double operator[](int i) const;
    const std::string & Name() const;
// friends
    // input
    friend std::istream & operator>>(std::istream & is,
                                     Student & stu);  // 1 word
    friend std::istream & getline(std::istream & is,
                                  Student & stu);     // 1 line
    // output
    friend std::ostream & operator<<(std::ostream & os,
                                     const Student & stu);
};

#endif
```

2. *访问基类的方法*

使用私有继承时，只能在派生类的方法中使用基类的方法。但有时候可能希望基类工具是公有的。例如，在类声明中提出可以使用 average( )函数。和包含一样，要实现这样的目的，可以在公有 Student::average( )函数中使用私有 Student::Average( )函数（参见图 14.2）。包含使用对象来调用方法：

```
double Student::Average() const
{
    if (scores.size() > 0)
        return scores.sum()/scores.size();
    else
```

```
        return 0;
}
```

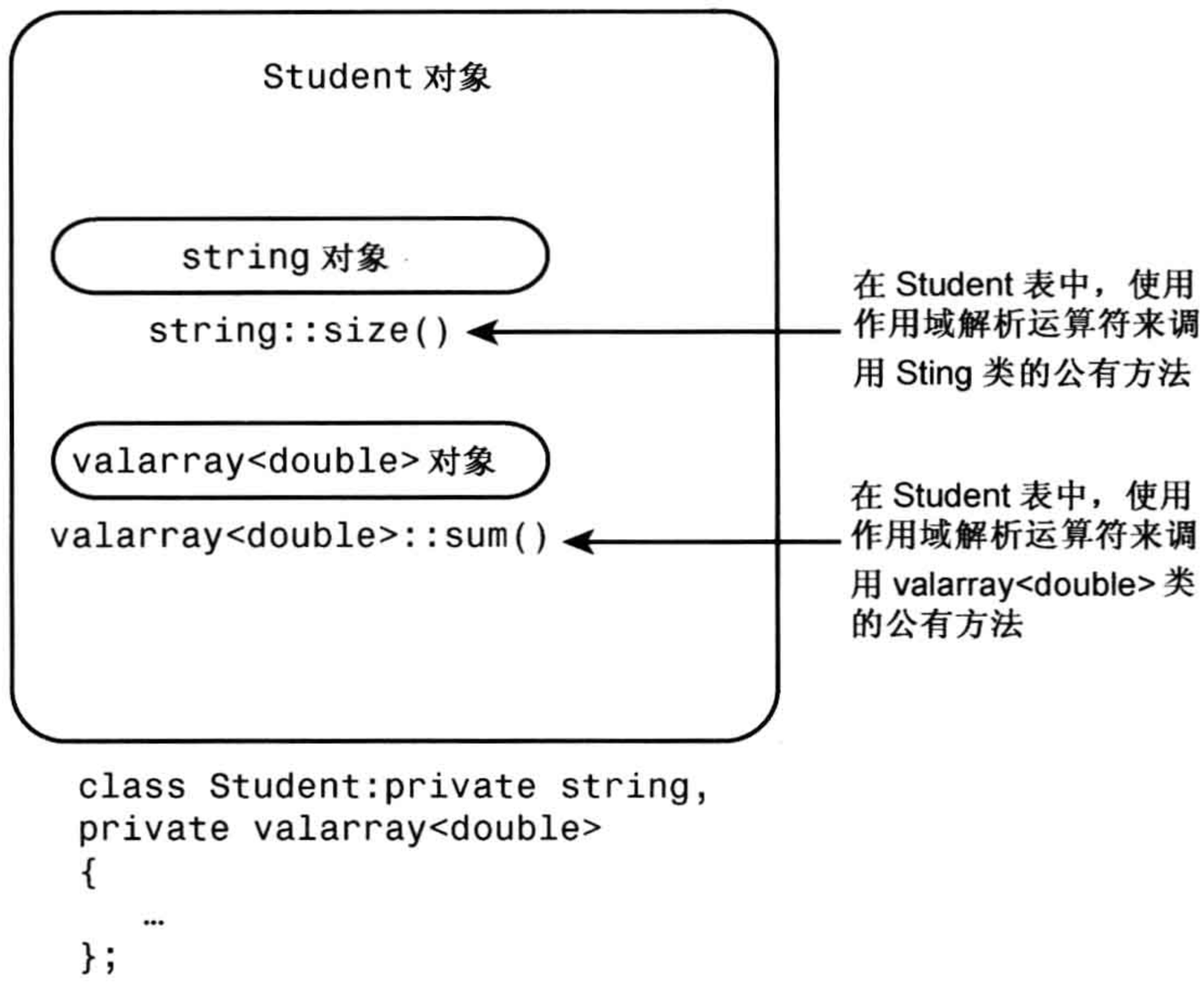


图 14.2　对象中的对象：私有继承

然而，私有继承使得能够使用类名和作用域解析运算符来调用基类的方法：

```
double Student::Average() const
{
    if (ArrayDb::size() > 0)
        return ArrayDb::sum()/ArrayDb::size();
    else
        return 0;
}
```

总之，使用包含时将使用对象名来调用方法，而使用私有继承时将使用类名和作用域解析运算符来调用方法。

3. 访问基类对象

使用作用域解析运算符可以访问基类的方法，但如果要使用基类对象本身，该如何做呢？例如，Student 类的包含版本实现了 Name( )方法，它返回 string 对象成员 name；但使用私有继承时，该 string 对象没有名称。那么，Student 类的代码如何访问内部的 string 对象呢？

答案是使用强制类型转换。由于 Student 类是从 string 类派生而来的，因此可以通过强制类型转换，将 Student 对象转换为 string 对象；结果为继承而来的 string 对象。本书前面介绍过，指针 this 指向用来调用方法的对象，因此*this 为用来调用方法的对象，在这个例子中，为类型为 Student 的对象。为避免调用构造函数创建新的对象，可使用强制类型转换来创建一个引用：

```
const string & Student::Name() const
{
    return (const string &) *this;
}
```

上述方法返回一个引用，该引用指向用于调用该方法的 Student 对象中的继承而来的 string 对象。

4. 访问基类的友元函数

用类名显式地限定函数名不适合于友元函数，这是因为友元不属于类。然而，可以通过显式地转换为基类来调用正确的函数。例如，对于下面的友元函数定义：

```
ostream & operator<<(ostream & os, const Student & stu)
{
      os << "Scores for " << (const String &) stu << ":\n";
...

}
```

如果 plato 是一个 Student 对象，则下面的语句将调用上述函数，stu 将是指向 plato 的引用，而 os 将是指向 cout 的引用：

```
cout << plato;
```

下面的代码：

```
os << "Scores for " << (const String &) stu << ":\n";
```

显式地将 stu 转换为 string 对象引用，进而调用函数 operator<<(ostream &, const String &)。

引用 stu 不会自动转换为 string 引用。根本原因在于，在私有继承中，在不进行显式类型转换的情况下，不能将指向派生类的引用或指针赋给基类引用或指针。

然而，即使这个例子使用的是公有继承，也必须使用显式类型转换。原因之一是，如果不使用类型转换，下述代码将与友元函数原型匹配，从而导致递归调用：

```
os << stu;
```

另一个原因是，由于这个类使用的是多重继承，编译器将无法确定应转换成哪个基类，如果两个基类都提供了函数 operator<<( )。程序清单 14.5 列出了除内联函数之外的所有 Student 类方法。

程序清单 14.5　student.cpp

```
// studenti.cpp -- Student class using private inheritance
#include "studenti.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;

// public methods
double Student::Average() const
{
    if (ArrayDb::size() > 0)
        return ArrayDb::sum()/ArrayDb::size();
    else
        return 0;
}

const string & Student::Name() const
{
    return (const string &) *this;
}

double & Student::operator[](int i)
{
    return ArrayDb::operator[](i);          // use ArrayDb::operator[]()
}
```

```
double Student::operator[](int i) const
{
    return ArrayDb::operator[](i);
}

// private method
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = ArrayDb::size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << ArrayDb::operator[](i) << " ";
            if (i % 5 == 4)
                os << endl;
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}

// friends
// use String version of operator>>()
istream & operator>>(istream & is, Student & stu)
{
    is >> (string &)stu;
    return is;
}

// use string friend getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
    getline(is, (string &)stu);
    return is;
}

// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << (const string &) stu  << ":\n";
    stu.arr_out(os);  // use private method for scores
    return os;
}
```

同样，由于这个示例也重用了 string 和 valarray 类的代码，因此除私有辅助方法外，它包含的新代码很少。

5. 使用修改后的 Student 类

接下来也需要测试这个新类。注意到两个版本的 Student 类的公有接口完全相同，因此可以使用同一个程序测试它们。唯一不同的是，应包含 studenti.h 而不是 studentc.h，应使用 studenti.cpp 而不是 studentc.cpp

来链接程序。程序清单 14.6 列出列该程序，请将其与 studenti.cpp 一起编译。

程序清单 14.6 use_stui.cpp

```
// use_stui.cpp -- using a class with private inheritance
// compile with studenti.cpp
#include <iostream>
#include "studenti.h"
using std::cin;
using std::cout;
using std::endl;

void set(Student & sa, int n);

const int pupils = 3;
const int quizzes = 5;

int main()
{
    Student ada[pupils] =
        {Student(quizzes), Student(quizzes), Student(quizzes)};

    int i;
    for (i = 0; i < pupils; i++)
        set(ada[i], quizzes);
    cout << "\nStudent List:\n";
    for (i = 0; i < pupils; ++i)
        cout << ada[i].Name() << endl;
    cout << "\nResults:";
    for (i = 0; i < pupils; i++)
    {
        cout << endl << ada[i];
        cout << "average: " << ada[i].Average() << endl;
    }
    cout << "Done.\n";
    return 0;
}
void set(Student & sa, int n)
{
    cout << "Please enter the student's name: ";
    getline(cin, sa);
    cout << "Please enter " << n << " quiz scores:\n";
    for (int i = 0; i < n; i++)
        cin >> sa[i];
    while (cin.get() != '\n')
        continue;
}
```

下面是该程序的运行情况：

```
Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
92 94 96 93 95
Please enter the student's name: Pat Roone
Please enter 5 quiz scores:
83 89 72 78 95
```

```
Please enter the student's name: Fleur O'Day
Please enter 5 quiz scores:
92 89 96 74 64

Student List:
Gil Bayts
Pat Roone
Fleur O'Day

Results:
Scores for Gil Bayts:
92 94 96 93 95
average: 94

Scores for Pat Roone:
83 89 72 78 95
average: 83.4

Scores for Fleur O'Day:
92 89 96 74 64
average: 83
Done.
```

输入与前一个测试程序相同，输出也相同。

14.2.2　使用包含还是私有继承

由于既可以使用包含，也可以使用私有继承来建立 has-a 关系，那么应使用种方式呢？大多数 C++程序员倾向于使用包含。首先，它易于理解。类声明中包含表示被包含类的显式命名对象，代码可以通过名称引用这些对象，而使用继承将使关系更抽象。其次，继承会引起很多问题，尤其从多个基类继承时，可能必须处理很多问题，如包含同名方法的独立的基类或共享祖先的独立基类。总之，使用包含不太可能遇到这样的麻烦。另外，包含能够包括多个同类的子对象。如果某个类需要 3 个 string 对象，可以使用包含声明 3 个独立的 string 成员。而继承则只能使用一个这样的对象（当对象都没有名称时，将难以区分）。

然而，私有继承所提供的特性确实比包含多。例如，假设类包含保护成员（可以是数据成员，也可以是成员函数），则这样的成员在派生类中是可用的，但在继承层次结构外是不可用的。如果使用组合将这样的类包含在另一个类中，则后者将不是派生类，而是位于继承层次结构之外，因此不能访问保护成员。但通过继承得到的将是派生类，因此它能够访问保护成员。

另一种需要使用私有继承的情况是需要重新定义虚函数。派生类可以重新定义虚函数，但包含类不能。使用私有继承，重新定义的函数将只能在类中使用，而不是公有的。

提示：通常，应使用包含来建立 has-a 关系；如果新类需要访问原有类的保护成员，或需要重新定义虚函数，则应使用私有继承。

14.2.3　保护继承

保护继承是私有继承的变体。保护继承在列出基类时使用关键字 protected：

```
class Student : protected std::string,
                protected std::valarray<double>
{...};
```

使用保护继承时，基类的公有成员和保护成员都将成为派生类的保护成员。和私有私有继承一样，基类的接口在派生类中也是可用的，但在继承层次结构之外是不可用的。当从派生类派生出另一个类时，私有继承和保护继承之间的主要区别便呈现出来了。使用私有继承时，第三代类将不能使用基类的接口，这

是因为基类的公有方法在派生类中将变成私有方法；使用保护继承时，基类的公有方法在第二代中将变成受保护的，因此第三代派生类可以使用它们。

表 14.1 总结了公有、私有和保护继承。隐式向上转换（implicit upcasting）意味着无需进行显式类型转换，就可以将基类指针或引用指向派生类对象。

表 14.1　　各种继承方式

特征	公有继承	保护继承	私有继承
公有成员变成	派生类的公有成员	派生类的保护成员	派生类的私有成员
保护成员变成	派生类的保护成员	派生类的保护成员	派生类的私有成员
私有成员变成	只能通过基类接口访问	只能通过基类接口访问	只能通过基类接口访问
能否隐式向上转换	是	是（但只能在派生类中）	否

14.2.4　使用 using 重新定义访问权限

使用保护派生或私有派生时，基类的公有成员将成为保护成员或私有成员。假设要让基类的方法在派生类外面可用，方法之一是定义一个使用该基类方法的派生类方法。例如，假设希望 Student 类能够使用 valarray 类的 sum( )方法，可以在 Student 类的声明中声明一个 sum( )方法，然后像下面这样定义该方法：

```
double Student::sum() const     // public Student method
{
    return std::valarray<double>::sum(); // use privately-inherited method
}
```

这样 Student 对象便能够调用 Student::sum( )，后者进而将 valarray<double>::sum( )方法应用于被包含的 valarray 对象（如果 ArrayDb typedef 在作用域中，也可以使用 ArrayDb 而不是 std::valarray<double>）。

另一种方法是，将函数调用包装在另一个函数调用中，即使用一个 using 声明（就像名称空间那样）来指出派生类可以使用特定的基类成员，即使采用的是私有派生。例如，假设希望通过 Student 类能够使用 valarray 的方法 min( )和 max( )，可以在 studenti.h 的公有部分加入如下 using 声明：

```
class Student : private std::string, private std::valarray<double>
{
...
public:
    using std::valarray<double>::min;
    using std::valarray<double>::max;
    ...
};
```

上述 using 声明使得 valarray<double>::min( )和 valarray<double>::max( )可用，就像它们是 Student 的公有方法一样：

```
cout << "high score: " << ada[i].max() << endl;
```

注意，using 声明只使用成员名——没有圆括号、函数特征标和返回类型。例如，为使 Student 类可以使用 valarray 的 operator[ ]( )方法，只需在 Student 类声明的公有部分包含下面的 using 声明：

```
using std::valarray<double>::operator[];
```

这将使两个版本（const 和非 const）都可用。这样，便可以删除 Student::operator[] ( )的原型和定义。using 声明只适用于继承，而不适用于包含。

有一种老式方式可用于在私有派生类中重新声明基类方法，即将方法名放在派生类的公有部分，如下所示：

```
class Student : private std::string, private std::valarray<double>
{
public:
```

```
    std::valarray<double>::operator[];  // redeclare as public, just use name
    ...

};
```

这看起来像不包含关键字 using 的 using 声明。这种方法已被摒弃，即将停止使用。因此，如果编译器支持 using 声明，应使用它来使派生类可以使用私有基类中的方法。

14.3　多重继承

MI 描述的是有多个直接基类的类。与单继承一样，公有 MI 表示的也是 is-a 关系。例如，可以从 Waiter 类和 Singer 类派生出 SingingWaiter 类：

```
class SingingWaiter : public Waiter, public Singer {...};
```

请注意，必须使用关键字 public 来限定每一个基类。这是因为，除非特别指出，否则编译器将认为是私有派生：

```
class SingingWaiter : public Waiter, Singer {...}; // Singer is a private base
```

正如本章前面讨论的，私有 MI 和保护 MI 可以表示 has-a 关系。Student 类的 studenti.h 实现就是一个这样的示例。下面将重点介绍公有 MI。

MI 可能会给程序员带来很多新问题。其中两个主要的问题是：从两个不同的基类继承同名方法；从两个或更多相关基类那里继承同一个类的多个实例。为解决这些问题，需要使用一些新规则和不同的语法。因此，与使用单继承相比，使用 MI 更困难，也更容易出现问题。由于这个原因，很多 C++用户强烈反对使用 MI，一些人甚至希望删除 MI；而喜欢 MI 的人则认为，对一些特殊的工程来说，MI 很有用，甚至是必不可少的；也有一些人建议谨慎、适度地使用 MI。

下面来看一个例子，并介绍有哪些问题以及如何解决它们。要使用 MI，需要几个类。我们将定义一个抽象基类 Worker，并使用它派生出 Waiter 类和 Singer 类。然后，便可以使用 MI 从 Waiter 类和 Singer 类派生出 SingingWaiter 类（参见图 14.3）。这里使用两个独立的派生来使基类（Worker）被继承，这将导致 MI 的大多数麻烦。首先声明 Worker、Waiter 和 Singer 类，如程序清单 14.7 所示。

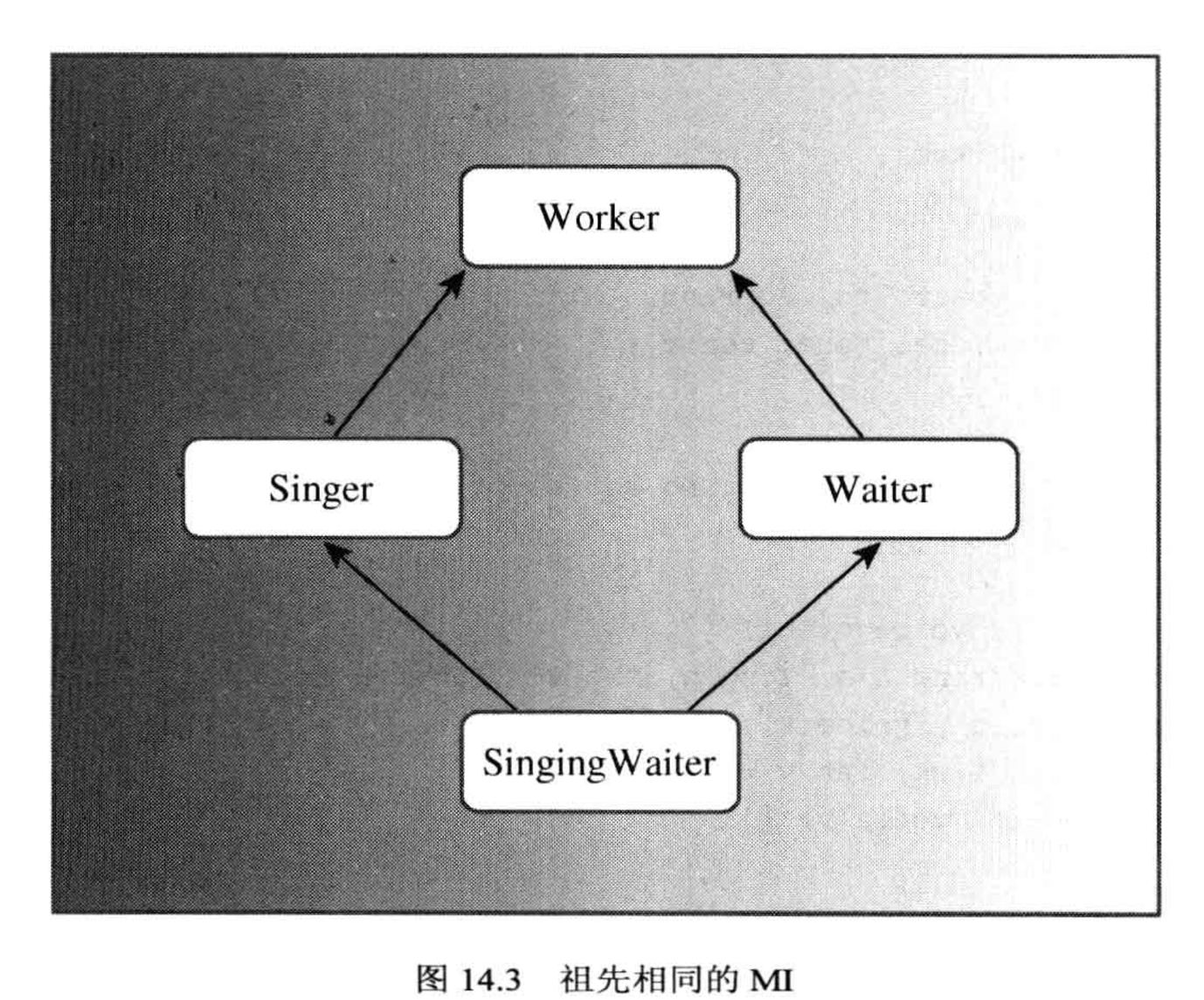


图 14.3　祖先相同的 MI

程序清单 14.7　Worker0.h

```
// worker0.h  -- working classes
#ifndef WORKER0_H_
#define WORKER0_H_

#include <string>

class Worker   // an abstract base class
{
private:
    std::string fullname;
    long id;
public:
    Worker() : fullname("no one"), id(0L) {}
    Worker(const std::string & s, long n)
            : fullname(s), id(n) {}
    virtual ~Worker() = 0;   // pure virtual destructor
    virtual void Set();
    virtual void Show() const;
};

class Waiter : public Worker
{
private:
    int panache;
public:
    Waiter() : Worker(), panache(0) {}
    Waiter(const std::string & s, long n, int p = 0)
            : Worker(s, n), panache(p) {}
    Waiter(const Worker & wk, int p = 0)
            : Worker(wk), panache(p) {}
    void Set();
    void Show() const;
};

class Singer : public Worker
{
protected:
    enum {other, alto, contralto, soprano,
                    bass, baritone, tenor};
    enum {Vtypes = 7};
private:
    static char *pv[Vtypes];    // string equivs of voice types
    int voice;
public:
    Singer() : Worker(), voice(other) {}
    Singer(const std::string & s, long n, int v = other)
            : Worker(s, n), voice(v) {}
    Singer(const Worker & wk, int v = other)
            : Worker(wk), voice(v) {}
    void Set();
    void Show() const;
};
```

```
#endif
```

程序清单 14.7 的类声明中包含一些表示声音类型的内部常量。一个枚举用符号常量 alto、contralto 等表示声音类型，静态数组 pv 存储了指向相应 C-风格字符串的指针，程序清单 14.8 初始化了该数组，并提供了方法的定义。

程序清单 14.8　worker0.cpp

```
// worker0.cpp -- working class methods
#include "worker0.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
// Worker methods

// must implement virtual destructor, even if pure
Worker::~Worker() {}

void Worker::Set()
{
    cout << "Enter worker's name: ";
    getline(cin, fullname);
    cout << "Enter worker's ID: ";
    cin >> id;
    while (cin.get() != '\n')
        continue;
}

void Worker::Show() const
{
    cout << "Name: " << fullname << "\n";
    cout << "Employee ID: " << id << "\n";
}

// Waiter methods
void Waiter::Set()
{
    Worker::Set();
    cout << "Enter waiter's panache rating: ";
    cin >> panache;
    while (cin.get() != '\n')
        continue;
}

void Waiter::Show() const
{
    cout << "Category: waiter\n";
    Worker::Show();
    cout << "Panache rating: " << panache << "\n";
}

// Singer methods
```

```
// Singer methods

char * Singer::pv[] = {"other", "alto", "contralto",
            "soprano", "bass", "baritone", "tenor"};

void Singer::Set()
{
    Worker::Set();
    cout << "Enter number for singer's vocal range:\n";
    int i;
    for (i = 0; i < Vtypes; i++)
    {
        cout << i << ": " << pv[i] << "   ";
        if ( i % 4 == 3)
            cout << endl;
    }
    if (i % 4 != 0)
        cout << endl;
    while (cin >>  voice && (voice < 0 || voice >= Vtypes) )
        cout << "Please enter a value >= 0 and < " << Vtypes << endl;

    while (cin.get() != '\n')
        continue;
}

void Singer::Show() const
{
    cout << "Category: singer\n";
    Worker::Show();
    cout << "Vocal range: " << pv[voice] << endl;
}
```

程序清单 14.9 是一个简短的程序，它使用一个多态指针数组对这些类进行了测试。

程序清单 14.9　worktest.cpp

```
// worktest.cpp -- test worker class hierarchy
#include <iostream>
#include "worker0.h"
const int LIM = 4;
int main()
{
    Waiter bob("Bob Apple", 314L, 5);
    Singer bev("Beverly Hills", 522L, 3);
    Waiter w_temp;
    Singer s_temp;

    Worker * pw[LIM] = {&bob, &bev, &w_temp, &s_temp};

    int i;
    for (i = 2; i < LIM; i++)
        pw[i]->Set();
    for (i = 0; i < LIM; i++)
    {
        pw[i]->Show();
```

```
        std::cout << std::endl;
    }

    return 0;
}
```

下面是程序清单 14.7～程序清单 14.9 组成的程序的输出：

```
Enter waiter's name: Waldo Dropmaster
Enter worker's ID: 442
Enter waiter's panache rating: 3
Enter singer's name: Sylvie Sirenne
Enter worker's ID: 555
Enter number for singer's vocal range:
0: other   1: alto   2: contralto   3: soprano
4: bass   5: baritone   6: tenor
3
Category: waiter
Name: Bob Apple
Employee ID: 314
Panache rating: 5

Category: singer
Name: Beverly Hills
Employee ID: 522
Vocal range: soprano

Category: waiter
Name: Waldo Dropmaster
Employee ID: 442
Panache rating: 3

Category: singer
Name: Sylvie Sirenne
Employee ID: 555
Vocal range: soprano
```

这种设计看起来是可行的：使用 Waiter 指针来调用 Waiter::Show( )和 Waiter::Set( )；使用 Singer 指针来调用 Singer::Show( )和 Singer::Set( )。然后，如果添加一个从 Singer 和 Waiter 类派生出的 SingingWaiter 类后，将带来一些问题。具体地说，将出现以下问题。

- 有多少 Worker？
- 哪个方法？

14.3.1　有多少 Worker

假设首先从 Singer 和 Waiter 公有派生出 SingingWaiter：

```
class SingingWaiter: public Singer, public  Waiter {...};
```

因为 Singer 和 Waiter 都继承了一个 Worker 组件，因此 SingingWaiter 将包含两个 Worker 组件（参见图 14.4）。

正如预期的，这将引起问题。例如，通常可以将派生类对象的地址赋给基类指针，但现在将出现二义性：

```
SingingWaiter ed;
Worker * pw = &ed;   //ambiguous
```

通常，这种赋值将把基类指针设置为派生对象中的基类对象的地址。但 ed 中包含两个 Worker 对象，有两个地址可供选择，所以应使用类型转换来指定对象：

```
Worker * pw1 = (Waiter *) &ed;   // the Worker in Waiter
Worker * pw2 = (Singer *) &ed;   // the Worker in Singer
```

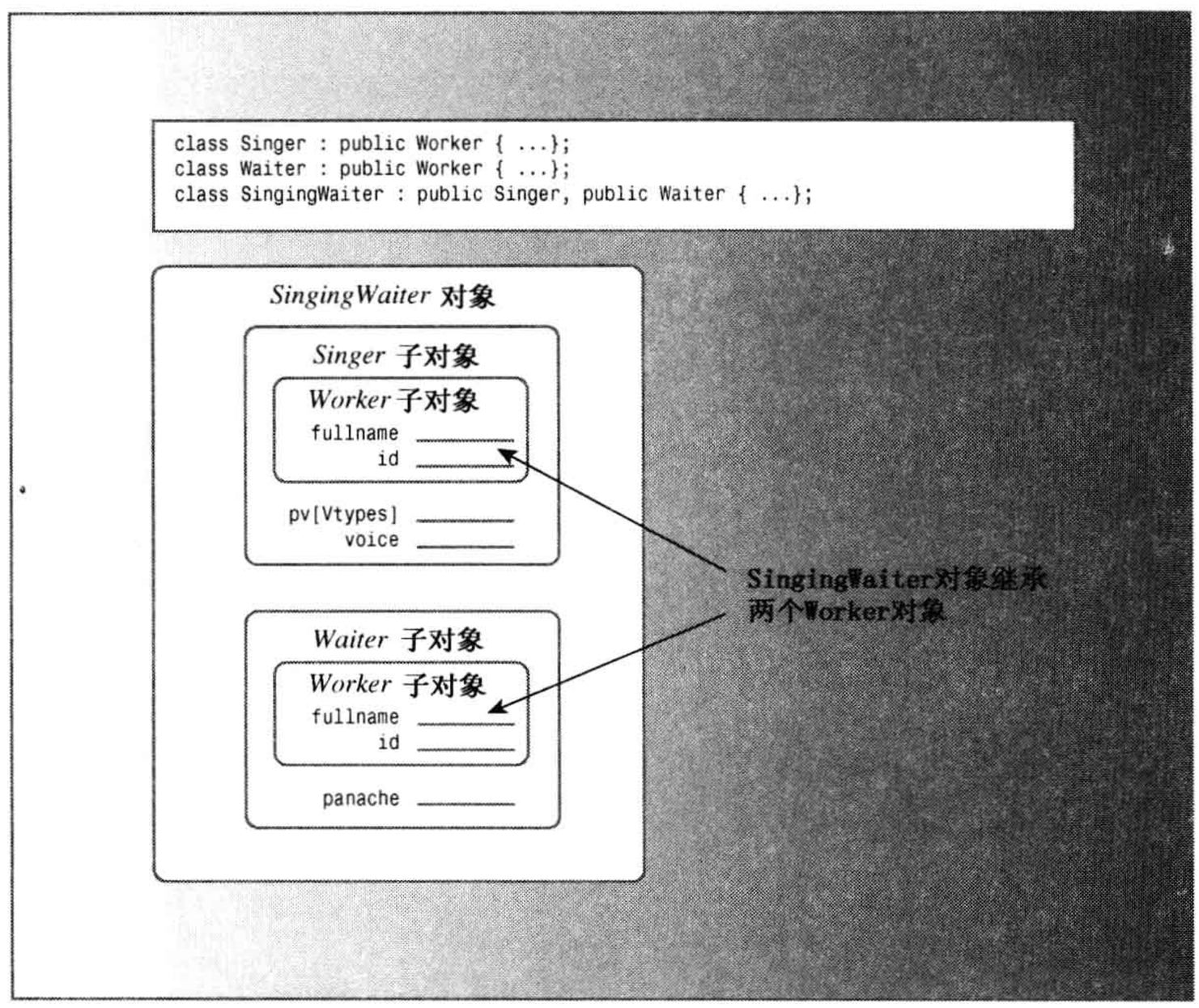


图 14.4 继承两个基类对象

这将使得使用基类指针来引用不同的对象（多态性）复杂化。

包含两个 Worker 对象拷贝还会导致其他的问题。然而，真正的问题是：为什么需要 Worker 对象的两个拷贝？唱歌的侍者和其他 Worker 对象一样，也应只包含一个姓名和一个 ID。C++引入多重继承的同时，引入了一种新技术——虚基类（virtual base class），使 MI 成为可能。

1. 虚基类

虚基类使得从多个类（它们的基类相同）派生出的对象只继承一个基类对象。例如，通过在类声明中使用关键字 virtual，可以使 Worker 被用作 Singer 和 Waiter 的虚基类（virtual 和 public 的次序无关紧要）：

```
class Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

然后，可以将 SingingWaiter 类定义为：

```
class SingingWaiter: public Singer, public  Waiter {...};
```

现在，SingingWaiter 对象将只包含 Worker 对象的一个副本。从本质上说，继承的 Singer 和 Waiter 对象共享一个 Worker 对象，而不是各自引入自己的 Worker 对象副本（请参见图 14.5）。因为 SingingWaiter 现在只包含了一个 Worker 子对象，所以可以使用多态。

您可能会有这样的疑问：

- 为什么使用术语“虚”？
- 为什么不抛弃将基类声明为虚的这种方式，而使虚行为成为多 MI 的准则呢？
- 是否存在麻烦呢？

首先，为什么使用术语虚？毕竟，在虚函数和虚基类之间并不存在明显的联系。C++用户强烈反对引入新的关键字，因为这将给他们带来很大的压力。例如，如果新关键字与重要程序中的重要函数或变量的名称相同，这将非常麻烦。因此，C++对这种新特性也使用关键字 virtual——有点像关键字重载。

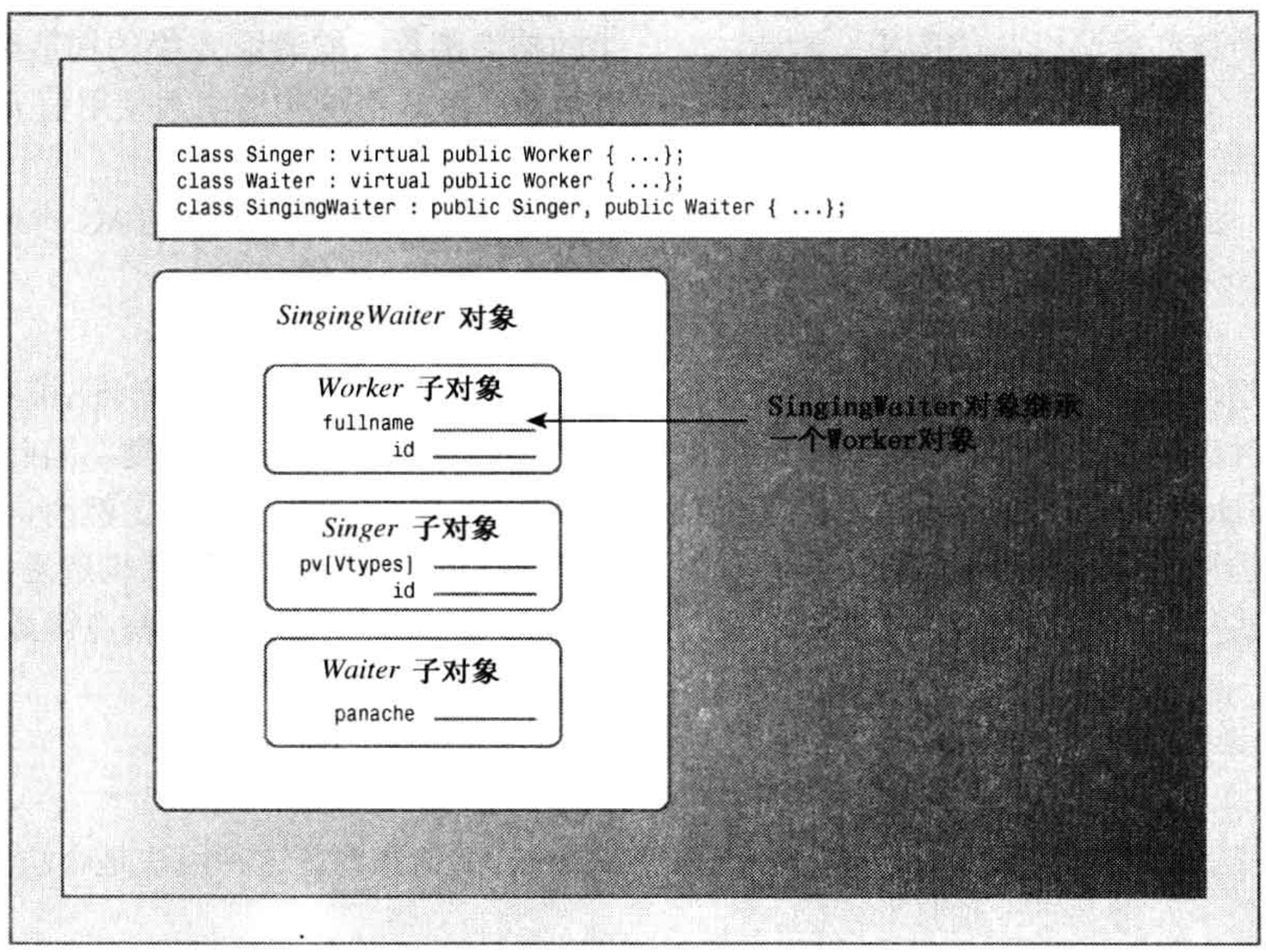


图 14.5　虚基类继承

其次，为什么不抛弃将基类声明为虚的这种方式，而使虚行为成为 MI 的准则呢？第一，在一些情况下，可能需要基类的多个拷贝；第二，将基类作为虚的要求程序完成额外的计算，为不需要的工具付出代价是不应当的；第三，这样做有其缺点，将在下一段介绍。

最后，是否存在麻烦？是的。为使虚基类能够工作，需要对 C++规则进行调整，必须以不同的方式编写一些代码。另外，使用虚基类还可能需要修改已有的代码。例如，将 SingingWaiter 类添加到 Worker 集成层次中时，需要在 Singer 和 Waiter 类中添加关键字 virtual。

2. 新的构造函数规则

使用虚基类时，需要对类构造函数采用一种新的方法。对于非虚基类，唯一可以出现在初始化列表中的构造函数是即时基类构造函数。但这些构造函数可能需要将信息传递给其基类。例如，可能有下面一组构造函数：

```
class A
{
    int a;
public:
    A(int n = 0) : a(n) {}
    ...
};
class B: public A
{
    int b;
public:
    B(int m = 0, int n = 0) : A(n), b(m)  {}
    ...
};
class C : public B
{
    int c;
public:
    C(int q = 0, int m = 0, int n = 0) : B(m, n), c(q) {}
    ...
};
```

C 类的构造函数只能调用 B 类的构造函数，而 B 类的构造函数只能调用 A 类的构造函数。这里，C 类的构造函数使用值 q，并将值 m 和 n 传递给 B 类的构造函数；而 B 类的构造函数使用值 m，并将值 n 传递给 A 类的构造函数。

如果 Worker 是虚基类，则这种信息自动传递将不起作用。例如，对于下面的 MI 构造函数：

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                    : Waiter(wk,p), Singer(wk,v) {}  // flawed
```

存在的问题是，自动传递信息时，将通过 2 条不同的途径（Waiter 和 Singer）将 wk 传递给 Worker 对象。为避免这种冲突，C++在基类是虚的时，禁止信息通过中间类自动传递给基类。因此，上述构造函数将初始化成员 panache 和 voice，但 wk 参数中的信息将不会传递给子对象 Waiter。然而，编译器必须在构造派生对象之前构造基类对象组件；在上述情况下，编译器将使用 Worker 的默认构造函数。

如果不希望默认构造函数来构造虚基类对象，则需要显式地调用所需的基类构造函数。因此，构造函数应该是这样：

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                    : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
```

上述代码将显式地调用构造函数 worker（const Worker &）。请注意，这种用法是合法的，对于虚基类，必须这样做；但对于非虚基类，则是非法的。

警告：如果类有间接虚基类，则除非只需使用该虚基类的默认构造函数，否则必须显式地调用该虚基类的某个构造函数。

14.3.2 哪个方法

除了修改类构造函数规则外，MI 通常还要求调整其他代码。假设要在 SingingWaiter 类中扩展 Show( ) 方法。因为 SingingWaiter 对象没有新的数据成员，所以可能会认为它只需使用继承的方法即可。这引出了第一个问题。假设没有在 SingingWaiter 类中重新定义 Show( )方法，并试图使用 SingingWaiter 对象调用继承的 Show( )方法：

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Show();  // ambiguous
```

对于单继承，如果没有重新定义 Show( )，则将使用最近祖先中的定义。而在多重继承中，每个直接祖先都有一个 Show( )函数，这使得上述调用是二义性的。

警告：多重继承可能导致函数调用的二义性。例如，BadDude 类可能从 Gunslinger 类和 PokerPlayer 类那里继承两个完全不同的 Draw( )方法。

可以使用作用域解析运算符来澄清编程者的意图：

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Singer::Show();  // use Singer version
```

然而，更好的方法是在 SingingWaiter 中重新定义 Show( )，并指出要使用哪个 Show( )。例如，如果希望 SingingWaiter 对象使用 Singer 版本的 Show( )，则可以这样做：

```
void SingingWaiter::Show()
{
    Singer::Show();
}
```

对于单继承来说，让派生方法调用基类的方法是可以的。例如，假设 HeadWaiter 类是从 Waiter 类派生而来的，则可以使用下面的定义序列，其中每个派生类使用其基类显示信息，并添加自己的信息：

```
void Worker::Show() const
{
```

```
    cout << "Name: " << fullname << "\n";
    cout << "Employee ID: " << id << "\n";
}

void Waiter::Show() const
{
    Worker::Show();
    cout << "Panache rating: " << panache << "\n";
}
void HeadWaiter::Show() const
{
    Waiter::Show();
    cout << "Presence rating: " << presence << "\n";
}
```

然而，这种递增的方式对 SingingWaiter 示例无效。下面的方法将无效，因为它忽略了 Waiter 组件：

```
void SingingWaiter::Show()
{
    Singer::Show();
}
```

可以通过同时调用 Waiter 版本的 Show( )来补救：

```
void SingingWaiter::Show()
{
      Singer::Show();
      Waiter::Show();
}
```

然而，这将显示姓名和 ID 两次，因为 Singer::Show( )和 Waiter::Show( )都调用了 Worker::Show( )。

如果解决呢？一种办法是使用模块化方式，而不是递增方式，即提供一个只显示 Worker 组件的方法和一个只显示 Waiter 组件或 Singer 组件（而不是 Waiter 和 Worker 组件）的方法。然后，在 SingingWaiter::Show( )方法中将组件组合起来。例如，可以这样做：

```
void Worker::Data() const
{
    cout << "Name: " << fullname << "\n";
    cout << "Employee ID: " << id << "\n";
}

void Waiter::Data() const
{
    cout << "Panache rating: " << panache << "\n";
}

void Singer::Data() const
{
    cout << "Vocal range: " << pv[voice] << "\n";
}

void SingingWaiter::Data() const
{
    Singer::Data();
    Waiter::Data();
}

void SingingWaiter::Show() const
```

```
{
    cout << "Category: singing waiter\n";
    Worker::Data();
    Data();
}
```

与此相似，其他 Show( )方法可以组合适当的 Data( )组件。

采用这种方式，对象仍可使用 Show( )方法。而 Data( )方法只在类内部可用，作为协助公有接口的辅助方法。然而，使 Data( )方法成为私有的将阻止 Waiter 中的代码使用 Worker::Data( )，这正是保护访问类的用武之地。如果 Data( )方法是保护的，则只能在继承层次结构中的类中使用它，在其他地方则不能使用。

另一种办法是将所有的数据组件都设置为保护的，而不是私有的，不过使用保护方法（而不是保护数据）将可以更严格地控制对数据的访问。

Set( )方法取得数据，以设置对象值，该方法也有类似的问题。例如，SingingWaiter::Set( )应请求 Worker 信息一次，而不是两次。对此，可以使用前面的解决方法。可以提供一个受保护的 Get( )方法，该方法只请求一个类的信息，然后将使用 Get( )方法作为构造块的 Set( )方法集合起来。

总之，在祖先相同时，使用 MI 必须引入虚基类，并修改构造函数初始化列表的规则。另外，如果在编写这些类时没有考虑到 MI，则还可能需要重新编写它们。程序清单 14.10 列出了修改后的类声明，程序清单 14.11 列出实现。

程序清单 14.10　workermi.h

```
// workermi.h  -- working classes with MI
#ifndef WORKERMI_H_
#define WORKERMI_H_

#include <string>

class Worker   // an abstract base class
{
private:
    std::string fullname;
    long id;
protected:
    virtual void Data() const;
    virtual void Get();
public:
    Worker() : fullname("no one"), id(0L) {}
    Worker(const std::string & s, long n)
            : fullname(s), id(n) {}
    virtual ~Worker() = 0; // pure virtual function
    virtual void Set() = 0;
    virtual void Show() const = 0;
};

class Waiter : virtual public Worker
{
private:
    int panache;
protected:
    void Data() const;
    void Get();
```

```
public:
    Waiter() : Worker(), panache(0) {}
    Waiter(const std::string & s, long n, int p = 0)
            : Worker(s, n), panache(p) {}
    Waiter(const Worker & wk, int p = 0)
            : Worker(wk), panache(p) {}
    void Set();
    void Show() const;
};

class Singer : virtual public Worker
{
protected:
enum {other, alto, contralto, soprano,
                    bass, baritone, tenor};
    enum {Vtypes = 7};
    void Data() const;
    void Get();
private:
    static char *pv[Vtypes];    // string equivs of voice types
    int voice;
public:
    Singer() : Worker(), voice(other) {}
    Singer(const std::string & s, long n, int v = other)
            : Worker(s, n), voice(v) {}
    Singer(const Worker & wk, int v = other)
            : Worker(wk), voice(v) {}
    void Set();
    void Show() const;
};

// multiple inheritance
class SingingWaiter : public Singer, public Waiter
{
protected:
    void Data() const;
    void Get();
public:
    SingingWaiter()  {}
    SingingWaiter(const std::string & s, long n, int p = 0,
                          int v = other)
            : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) {}
    SingingWaiter(const Worker & wk, int p = 0, int v = other)
            : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
    SingingWaiter(const Waiter & wt, int v = other)
            : Worker(wt),Waiter(wt), Singer(wt,v) {}
    SingingWaiter(const Singer & wt, int p = 0)
            : Worker(wt),Waiter(wt,p), Singer(wt) {}
    void Set();
    void Show() const;
};

#endif
```

程序清单 14.11　workermi.cpp

```
// workermi.cpp -- working class methods with MI
#include "workermi.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
// Worker methods
Worker::~Worker() { }

// protected methods
void Worker::Data() const
{
    cout << "Name: " << fullname << endl;
    cout << "Employee ID: " << id << endl;
}

void Worker::Get()
{
    getline(cin, fullname);
    cout << "Enter worker's ID: ";
    cin >> id;
    while (cin.get() != '\n')
        continue;
}

// Waiter methods
void Waiter::Set()
{
    cout << "Enter waiter's name: ";
    Worker::Get();
    Get();
}

void Waiter::Show() const
{
    cout << "Category: waiter\n";
    Worker::Data();
    Data();
}

// protected methods
void Waiter::Data() const
{
    cout << "Panache rating: " << panache << endl;
}
void Waiter::Get()
{
    cout << "Enter waiter's panache rating: ";
    cin >> panache;
    while (cin.get() != '\n')
        continue;
}
```

```
// Singer methods

char * Singer::pv[Singer::Vtypes] = {"other", "alto", "contralto",
            "soprano", "bass", "baritone", "tenor"};

void Singer::Set()
{
    cout << "Enter singer's name: ";
    Worker::Get();
    Get();
}

void Singer::Show() const
{
    cout << "Category: singer\n";
    Worker::Data();
    Data();
}

// protected methods
void Singer::Data() const
{
    cout << "Vocal range: " << pv[voice] << endl;
}

void Singer::Get()
{
    cout << "Enter number for singer's vocal range:\n";
    int i;
    for (i = 0; i < Vtypes; i++)
    {
        cout << i << ": " << pv[i] << "   ";
        if ( i % 4 == 3)
            cout << endl;
    }
    if (i % 4 != 0)
        cout << '\n';
    cin >>  voice;
    while (cin.get() != '\n')
        continue;
}

// SingingWaiter methods
void SingingWaiter::Data() const
{
    Singer::Data();
    Waiter::Data();
}

void SingingWaiter::Get()
{
    Waiter::Get();
    Singer::Get();
}
```

```
void SingingWaiter::Set()
{
    cout << "Enter singing waiter's name: ";
    Worker::Get();
    Get();
}

void SingingWaiter::Show() const
{
    cout << "Category: singing waiter\n";
    Worker::Data();
    Data();
}
```

当然，好奇心要求我们测试这些类，程序清单 14.12 提供了测试代码。注意，该程序使用了多态属性，将各种类的地址赋给基类指针。另外，该程序还在下面的检测中使用了 C-风格字符串库函数 strchr( )：

```
while (strchr("wstq", choice) == NULL)
```

该函数返回参数 choice 指定的字符在字符串“wstq”中第一次出现的地址，如果没有这样的字符，则返回 NULL 指针。使用这种检测比使用 if 语句将 choice 指定的字符同每个字符进行比较简单。

请将程序清单 14.12 与 workermi.cpp 一起编译。

程序清单 14.12　workmi.cpp

```
// workmi.cpp -- multiple inheritance
// compile with workermi.cpp
#include <iostream>
#include <cstring>
#include "workermi.h"
const int SIZE = 5;

int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   using std::strchr;

   Worker * lolas[SIZE];

    int ct;
    for (ct = 0; ct < SIZE; ct++)
    {
        char choice;
        cout << "Enter the employee category:\n"
            << "w: waiter  s: singer  "
            << "t: singing waiter  q: quit\n";
        cin >> choice;
        while (strchr("wstq", choice) == NULL)
        {
            cout << "Please enter a w, s, t, or q: ";
            cin >> choice;
        }
        if (choice == 'q')
            break;
```

```
    switch(choice)
    {
        case 'w':   lolas[ct] = new Waiter;
                    break;
        case 's':   lolas[ct] = new Singer;
                    break;
        case 't':   lolas[ct] = new SingingWaiter;
                    break;
    }
    cin.get();
        lolas[ct]->Set();
    }

    cout << "\nHere is your staff:\n";
    int i;
    for (i = 0; i < ct; i++)
    {
        cout << endl;
        lolas[i]->Show();
    }
    for (i = 0; i < ct; i++)
        delete lolas[i];
    cout << "Bye.\n";
    return 0;
}
```

下面是程序清单 14.10～程序清单 14.12 组成的程序的运行情况：

```
Enter the employee category:
w: waiter  s: singer  t: singing waiter  q: quit
w
Enter waiter's name: Wally Slipshod
Enter worker's ID: 1040
Enter waiter's panache rating: 4
Enter the employee category:
w: waiter  s: singer  t: singing waiter  q: quit
s
Enter singer's name: Sinclair Parma
Enter worker's ID: 1044
Enter number for singer's vocal range:
0: other   1: alto   2: contralto   3: soprano
4: bass   5: baritone   6: tenor
5
Enter the employee category:
w: waiter  s: singer  t: singing waiter  q: quit
t
Enter singing waiter's name: Natasha Gargalova
Enter worker's ID: 1021
Enter waiter's panache rating: 6
Enter number for singer's vocal range:
0: other   1: alto   2: contralto   3: soprano
4: bass   5: baritone   6: tenor
3
Enter the employee category:
w: waiter  s: singer  t: singing waiter  q: quit
q
```

```
Here is your staff:

Category: waiter
Name: Wally Slipshod
Employee ID: 1040
Panache rating: 4

Category: singer
Name: Sinclair Parma
Employee ID: 1044
Vocal range: baritone

Category: singing waiter
Name: Natasha Gargalova
Employee ID: 1021
Vocal range: soprano
Panache rating: 6
Bye.
```

下面介绍其他一些有关 MI 的问题。

1. 混合使用虚基类和非虚基类

再来看一下通过多种途径继承一个基类的派生类的情况。如果基类是虚基类，派生类将包含基类的一个子对象；如果基类不是虚基类，派生类将包含多个子对象。当虚基类和非虚基类混合时，情况将如何呢？例如，假设类 B 被用作类 C 和 D 的虚基类，同时被用作类 X 和 Y 的非虚基类，而类 M 是从 C、D、X 和 Y 派生而来的。在这种情况下，类 M 从虚派生祖先（即类 C 和 D）那里共继承了一个 B 类子对象，并从每一个非虚派生祖先（即类 X 和 Y）分别继承了一个 B 类子对象。因此，它包含三个 B 类子对象。当类通过多条虚途径和非虚途径继承某个特定的基类时，该类将包含一个表示所有的虚途径的基类子对象和分别表示各条非虚途径的多个基类子对象。

2. 虚基类和支配

使用虚基类将改变 C++解析二义性的方式。使用非虚基类时，规则很简单。如果类从不同的类那里继承了两个或更多的同名成员（数据或方法），则使用该成员名时，如果没有用类名进行限定，将导致二义性。但如果使用的是虚基类，则这样做不一定会导致二义性。在这种情况下，如果某个名称优先于（dominates）其他所有名称，则使用它时，即便不使用限定符，也不会导致二义性。

那么，一个成员名如何优先于另一个成员名呢？派生类中的名称优先于直接或间接祖先类中的相同名称。例如，在下面的定义中：

```
class B
{
public:
      short q();
      ...
};

class C : virtual public B
{
public:
      long q();
      int omg()
      ...
};
```

```
class D : public C
{
      ...
};

class E : virtual public B
{
private:
      int omg();
      ...
};

class F:  public D, public E
{
     ...
};
```

类 C 中的 q( )定义优先于类 B 中的 q( )定义，因为类 C 是从类 B 派生而来的。因此，F 中的方法可以使用 q( )来表示 C::q( )。另一方面，任何一个 omg( )定义都不优先于其他 omg( )定义，因为 C 和 E 都不是对方的基类。所以，在 F 中使用非限定的 omg( )将导致二义性。

虚二义性规则与访问规则无关，也就是说，即使 E::omg( )是私有的，不能在 F 类中直接访问，但使用 omg( )仍将导致二义性。同样，即使 C::q( )是私有的，它也将优先于 D::q( )。在这种情况下，可以在类 F 中调用 B::q( )，但如果不限定 q( )，则将意味着要调用不可访问的 C::q( )。

14.3.3　MI 小结

首先复习一下不使用虚基类的 MI。这种形式的 MI 不会引入新的规则。然而，如果一个类从两个不同的类那里继承了两个同名的成员，则需要在派生类中使用类限定符来区分它们。即在从 GunSlinger 和 PokerPlayer 派生而来的 BadDude 类中，将分别使用 Gunslinger::draw( )和 PokerPlayer::draw( )来区分从这两个类那里继承的 draw( )方法。否则，编译器将指出二义性。

如果一个类通过多种途径继承了一个非虚基类，则该类从每种途径分别继承非虚基类的一个实例。在某些情况下，这可能正是所希望的，但通常情况下，多个基类实例都是问题。

接下来看一看使用虚基类的 MI。当派生类使用关键字 virtual 来指示派生时，基类就成为虚基类：

```
class marketing : public virtual reality { ... };
```

主要变化（同时也是使用虚基类的原因）是，从虚基类的一个或多个实例派生而来的类将只继承了一个基类对象。为实现这种特性，必须满足其他要求：

- 有间接虚基类的派生类包含直接调用间接基类构造函数的构造函数，这对于间接非虚基类来说是非法的；
- 通过优先规则解决名称二义性。

正如您看到的，MI 会增加编程的复杂程度。然而，这种复杂性主要是由于派生类通过多条途径继承同一个基类引起的。避免这种情况后，唯一需要注意的是，在必要时对继承的名称进行限定。

14.4　类模板

继承（公有、私有或保护）和包含并不总是能够满足重用代码的需要。例如，Stack 类（参见第 10 章）和 Queue 类（参见第 12 章）都是容器类（container class），容器类设计用来存储其他对象或数据类型。例如，第 10 章的 Stack 类设计用于存储 unsigned long 值。可以定义专门用于存储 double 值或 string 对象的 Stack 类，

除了保存的对象类型不同外，这两种 Stack 类的代码是相同的。然而，与其编写新的类声明，不如编写一个泛型（即独立于类型的）栈，然后将具体的类型作为参数传递给这个类。这样就可以使用通用的代码生成存储不同类型值的栈。第 10 章的 Stack 示例使用 typedef 处理这种需求。然而，这种方法有两个缺点：首先，每次修改类型时都需要编辑头文件；其次，在每个程序中只能使用这种技术生成一种栈，即不能让 typedef 同时代表两种不同的类型，因此不能使用这种方法在同一个程序中同时定义 int 栈和 string 栈。

C++的类模板为生成通用的类声明提供了一种更好的方法（C++最初不支持模板，但模板被引入后，就一直在演化，因此有的编译器可能不支持这里介绍的所有特性）。模板提供参数化（parameterized）类型，即能够将类型名作为参数传递给接收方来建立类或函数。例如，将类型名 int 传递给 Queue 模板，可以让编译器构造一个对 int 进行排队的 Queue 类。

C++库提供了多个模板类，本章前面使用了模板类 valarray，第 4 章介绍了模板类 vector 和 array，而第 16 章将讨论的 C++标准模板库（STL）提供了几个功能强大而灵活的容器类模板实现。本章将介绍如何设计一些基本的特性。

14.4.1 定义类模板

下面以第 10 章的 Stack 类为基础来建立模板。原来的类声明如下：

```
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};     // constant specific to class
    Item items[MAX];     // holds stack items
    int top;             // index for top stack item
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);   // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);          // pop top into item
};
```

采用模板时，将使用模板定义替换 Stack 声明，使用模板成员函数替换 Stack 的成员函数。和模板函数一样，模板类以下面这样的代码开头：

```
template <class Type>
```

关键字 template 告诉编译器，将要定义一个模板。尖括号中的内容相当于函数的参数列表。可以把关键字 class 看作是变量的类型名，该变量接受类型作为其值，把 Type 看作是该变量的名称。

这里使用 class 并不意味着 Type 必须是一个类；而只是表明 Type 是一个通用的类型说明符，在使用模板时，将使用实际的类型替换它。较新的 C++实现允许在这种情况下使用不太容易混淆的关键字 typename 代替 class：

```
template <typename Type>  // newer choice
```

可以使用自己的泛型名代替 Type，其命名规则与其他标识符相同。当前流行的选项包括 T 和 Type，我们将使用后者。当模板被调用时，Type 将被具体的类型值（如 int 或 string）取代。在模板定义中，可以使用泛型名来标识要存储在栈中的类型。对于 Stack 来说，这意味着应将声明中所有的 typedef 标识符 Item 替换为 Type。例如，

```
Item items[MAX];  // holds stack items
```

应改为：

```
Type items[MAX];  // holds stack items
```

同样，可以使用模板成员函数替换原有类的类方法。每个函数头都将以相同的模板声明打头：

```
template <class Type>
```

同样应使用泛型名 Type 替换 typedef 标识符 Item。另外，还需将类限定符从 Stack::改为 Stack<Type>::。例如，

```
bool Stack::push(const Item & item)
{
...
}
```

应该为：

```
template <class Type>                        // or template <typename Type>
bool Stack<Type>::push(const Type & item)
{
...
}
```

如果在类声明中定义了方法（内联定义），则可以省略模板前缀和类限定符。

程序清单 14.13 列出了类模板和成员函数模板。知道这些模板不是类和成员函数定义至关重要。它们是 C++编译器指令，说明了如何生成类和成员函数定义。模板的具体实现——如用来处理 string 对象的栈类——被称为实例化（instantiation）或具体化（specialization）。不能将模板成员函数放在独立的实现文件中（以前，C++标准确实提供了关键字 export，让您能够将模板成员函数放在独立的实现文件中，但支持该关键字的编译器不多；C++11 不再这样使用关键字 export，而将其保留用于其他用途）。由于模板不是函数，它们不能单独编译。模板必须与特定的模板实例化请求一起使用。为此，最简单的方法是将所有模板信息放在一个头文件中，并在要使用这些模板的文件中包含该头文件。

程序清单 14.13　stacktp.h

```
// stacktp.h -- a stack template
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>
class Stack
{
private:
    enum {MAX = 10};     // constant specific to class
    Type items[MAX];     // holds stack items
    int top;             // index for top stack item
public:
    Stack();
    bool isempty();
    bool isfull();
    bool push(const Type & item); // add item to stack
    bool pop(Type & item);        // pop top into item
};

template <class Type>
Stack<Type>::Stack()
{
    top = 0;
}

template <class Type>
bool Stack<Type>::isempty()
```

```
{
    return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
    return top == MAX;
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}

#endif
```

14.4.2 使用模板类

仅在程序包含模板并不能生成模板类，而必须请求实例化。为此，需要声明一个类型为模板类的对象，方法是使用所需的具体类型替换泛型名。例如，下面的代码创建两个栈，一个用于存储 int，另一个用于存储 string 对象：

```
Stack<int> kernels;       // create a stack of ints
Stack<string> colonels;   // create a stack of string objects
```

看到上述声明后，编译器将按 Stack<Type>模板来生成两个独立的类声明和两组独立的类方法。类声明 Stack<int>将使用 int 替换模板中所有的 Type，而类声明 Stack<string>将用 string 替换 Type。当然，使用的算法必须与类型一致。例如，Stack 类假设可以将一个项目赋给另一个项目。这种假设对于基本类型、结构和类来说是成立的（除非将赋值运算符设置为私有的），但对于数组则不成立。

泛型标识符——例如这里的 Type——称为类型参数（type parameter），这意味着它们类似于变量，但赋给它们的不能是数字，而只能是类型。因此，在 kernel 声明中，类型参数 Type 的值为 int。

注意，必须显式地提供所需的类型，这与常规的函数模板是不同的，因为编译器可以根据函数的参数类型来确定要生成哪种函数：

```
template <class T>
void simple(T t) { cout << t << '\n';}
...
simple(2);        // generate void simple(int)
simple("two");    // generate void simple(const char *)
```

程序清单 14.14 修改了原来的栈测试程序（程序清单 11.12），使用字符串而不是 unsigned long 值作为订单 ID。

程序清单 14.14　stacktem.cpp

```
// stacktem.cpp -- testing the template stack class
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using std::cin;
using std::cout;

int main()
{
    Stack<std::string> st;   // create an empty stack
    char ch;
    std::string po;
    cout << "Please enter A to add a purchase order,\n"
         << "P to process a PO, or Q to quit.\n";
    while (cin >> ch && std::toupper(ch) != 'Q')
    {
        while (cin.get() != '\n')
            continue;
        if (!std::isalpha(ch))
        {
            cout << '\a';
            continue;
        }
        switch(ch)
        {
            case 'A':
            case 'a': cout << "Enter a PO number to add: ";
                      cin >> po;
                      if (st.isfull())
                          cout << "stack already full\n";
                      else
                          st.push(po);
                      break;
            case 'P':
            case 'p': if (st.isempty())
                          cout << "stack already empty\n";
                      else {
                          st.pop(po);
                          cout << "PO #" << po << " popped\n";
                          break;
                      }
        }
        cout << "Please enter A to add a purchase order,\n"
             << "P to process a PO, or Q to quit.\n";
```

```
    }
    cout << "Bye\n";
    return 0;
}
```

程序清单 14.14 所示程序的运行情况如下：

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: red911porsche
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: blueR8audi
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: silver747boeing
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #silver747boeing popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #blueR8audi popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #red911porsche popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

14.4.3 深入探讨模板类

可以将内置类型或类对象用作类模板 Stack<Type>的类型。指针可以吗？例如，可以使用 char 指针替换程序清单 14.14 中的 string 对象吗？毕竟，这种指针是处理 C-风格字符串的内置方式。答案是可以创建指针栈，但如果不对程序做重大修改，将无法很好地工作。编译器可以创建类，但使用效果如何就因人而异了。下面解释程序清单 14.14 不太适合使用指针栈的原因，然后介绍一个指针栈很有用的例子。

1. 不正确地使用指针栈

我们将简要地介绍 3 个试图对程序清单 14.14 进行修改，使之使用指针栈的简单（但有缺陷的）示例。这几个示例揭示了设计模板时应牢记的一些教训，切忌盲目使用模板。这 3 个示例都以完全正确的 Stack<Type>模板为基础：

```
Stack<char *> st; // create a stack for pointers-to-char
```

版本 1 将程序清单 14.14 中的：

```
string po;
```

替换为：

```
char * po;
```

这旨在用 char 指针而不是 string 对象来接收键盘输入。这种方法很快就失败了，因为仅仅创建指针，没有创建用于保存输入字符串的空间（程序将通过编译，但在 cin 试图将输入保存在某些不合适的内存单元中时崩溃）。

版本 2 将

```
string po;
```

替换为：

```
char po[40];
```

这为输入的字符串分配了空间。另外，po 的类型为 char *，因此可以被放在栈中。但数组完全与 pop( ) 方法的假设相冲突：

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

首先，引用变量 item 必须引用某种类型的左值，而不是数组名。其次，代码假设可以给 item 赋值。即使 item 能够引用数组，也不能为数组名赋值。因此这种方法失败了。

版本 3 将

```
string po;
```

替换为：

```
char * po = new char[40];
```

这为输入的字符串分配了空间。另外，po 是变量，因此与 pop( )的代码兼容。然而，这里将会遇到最基本的问题：只有一个 pop 变量，该变量总是指向相同的内存单元。确实，在每当读取新字符串时，内存的内容都将发生改变，但每次执行压入操作时，加入到栈中的的地址都相同。因此，对栈执行弹出操作时，得到的地址总是相同的，它总是指向读入的最后一个字符串。具体地说，栈并没有保存每一个新字符串，因此没有任何用途。

2. 正确使用指针栈

使用指针栈的方法之一是，让调用程序提供一个指针数组，其中每个指针都指向不同的字符串。把这些指针放在栈中是有意义的，因为每个指针都将指向不同的字符串。注意，创建不同指针是调用程序的职责，而不是栈的职责。栈的任务是管理指针，而不是创建指针。

例如，假设我们要模拟下面的情况。某人将一车文件夹交付给了 Plodson。如果 Plodson 的收取篮（in-basket）是空的，他将取出车中最上面的文件夹，将其放入收取篮；如果收取篮是满的，Plodson 将取出篮中最上面的文件，对它进行处理，然后放入发出篮（out-basket）中。如果收取篮既不是空的也不是满的，Plodson 将处理收取篮中最上面的文件，也可能取出车中的下一个文件，把它放入收取篮。他采取了自认为是比较鲁莽的行动——扔硬币来决定要采取的措施。下面来讨论他的方法对原始文件处理顺序的影响。

可以用一个指针数组来模拟这种情况，其中的指针指向表示车中文件的字符串。每个字符串都包含文件所描述的人的姓名。可以用栈表示收取篮，并使用第二个指针数组来表示发出篮。通过将指针从输入数组压入到栈中来表示将文件添加到收取篮中，同时通过从栈中弹出项目，并将它添加到发出篮中来表示处理文件。

应考虑该问题的各个方面，因此栈的大小必须是可变的。程序清单 14.15 重新定义了 Stack<Type>类，使 Stack 构造函数能够接受一个可选大小的参数。这涉及到在内部使用动态数组，因此，Stack 类需要包含一个析构函数、一个复制构造函数和一个赋值运算符。另外，通过将多个方法作为内联函数，精减了代码。

程序清单 14.15　stcktp1.h

```
// stcktp1.h -- modified Stack template
#ifndef STCKTP1_H_
#define STCKTP1_H_

template <class Type>
class Stack
{
private:
    enum {SIZE = 10};    // default size
    int stacksize;
    Type * items;        // holds stack items
    int top;             // index for top stack item
public:
    explicit Stack(int ss = SIZE);
    Stack(const Stack & st);
    ~Stack() { delete [] items; }
    bool isempty() { return top == 0; }
    bool isfull() { return top == stacksize; }
    bool push(const Type & item);   // add item to stack
    bool pop(Type & item);          // pop top into item
    Stack & operator=(const Stack & st);
};

template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
    items = new Type [stacksize];
}

template <class Type>
Stack<Type>::Stack(const Stack & st)
{
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
}
template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < stacksize)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
```

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}

template <class Type>
Stack<Type> & Stack<Type>::operator=(const Stack<Type> & st)
{
    if (this == &st)
        return *this;
    delete [] items;
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
    return *this;
}

#endif
```

原型将赋值运算符函数的返回类型声明为 Stack 引用，而实际的模板函数定义将类型定义为 Stack<Type>。前者是后者的缩写，但只能在类中使用。即可以在模板声明或模板函数定义内使用 Stack，但在类的外面，即指定返回类型或使用作用域解析运算符时，必须使用完整的 Stack<Type>。

程序清单 14.16 中的程序使用新的栈模板来实现 Plodson 模拟，它像以前介绍的模拟那样使用 rand( )、srand( )和 time( )来生成随机数，这里是随机生成 0 和 1，来模拟掷硬币的结果。

程序清单 14.16　stkoptr1.cpp

```
// stkoptr1.cpp -- testing stack of pointers
#include <iostream>
#include <cstdlib>     // for rand(), srand()
#include <ctime>       // for time()
#include "stcktp1.h"
const int Num = 10;
int main()
{
    std::srand(std::time(0)); // randomize rand()
    std::cout << "Please enter stack size: ";
    int stacksize;
    std::cin >> stacksize;
// create an empty stack with stacksize slots
    Stack<const char *> st(stacksize);

// in basket
    const char * in[Num] = {
            " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
```

```
            " 3: Betty Rocker", " 4: Ian Flagranti",
            " 5: Wolfgang Kibble", " 6: Portia Koop",
            " 7: Joy Almondo", " 8: Xaverie Paprika",
            " 9: Juan Moore", "10: Misha Mache"
            };
// out basket
    const char * out[Num];

    int processed = 0;
    int nextin = 0;
    while (processed < Num)
    {
        if (st.isempty())
            st.push(in[nextin++]);
        else if (st.isfull())
            st.pop(out[processed++]);
        else if (std::rand() % 2  && nextin < Num)   // 50-50 chance
            st.push(in[nextin++]);
        else
            st.pop(out[processed++]);
    }
    for (int i = 0; i < Num; i++)
        std::cout << out[i] << std::endl;

    std::cout << "Bye\n";
    return 0;
}
```

下面是程序清单 14.16 所示程序的两次运行情况。注意，由于使用了随机特性，每次运行时，文件最后的顺序都可能不同，即使栈大小保持不变。

```
Please enter stack size: 5
 2: Kiki Ishtar
 1: Hank Gilgamesh
 3: Betty Rocker
 5: Wolfgang Kibble
 4: Ian Flagranti
 7: Joy Almondo
 9: Juan Moore
 8: Xaverie Paprika
 6: Portia Koop
10: Misha Mache
Bye

Please enter stack size: 5
 3: Betty Rocker
 5: Wolfgang Kibble
 6: Portia Koop
 4: Ian Flagranti
 8: Xaverie Paprika
 9: Juan Moore
10: Misha Mache
 7: Joy Almondo
 2: Kiki Ishtar
 1: Hank Gilgamesh
Bye
```

程序说明

在程序清单 14.16 中，字符串本身永远不会移动。把字符串压入栈实际上是新建一个指向该字符串的指针，即创建一个指针，该指针的值是现有字符串的地址。从栈弹出字符串将把地址值复制到 out 数组中。

该程序使用的类型是 const char *，因为指针数组将被初始化为一组字符串常量。

栈的析构函数对字符串有何影响呢？没有。构造函数使用 new 创建一个用于保存指针的数组，析构函数删除该数组，而不是数组元素指向的字符串。

14.4.4　数组模板示例和非类型参数

模板常用作容器类，这是因为类型参数的概念非常适合于将相同的存储方案用于不同的类型。确实，为容器类提供可重用代码是引入模板的主要动机，所以我们来看看另一个例子，深入探讨模板设计和使用的其他几个方面。具体地说，将探讨一些非类型（或表达式）参数以及如何使用数组来处理继承族。

首先介绍一个允许指定数组大小的简单数组模板。一种方法是在类中使用动态数组和构造函数参数来提供元素数目，最后一个版本的 Stack 模板采用的就是这种方法。另一种方法是使用模板参数来提供常规数组的大小，C++11 新增的模板 array 就是这样做的。程序清单 14.17 演示了如何做。

程序清单 14.17　arraytp.h

```
//arraytp.h  -- Array Template
#ifndef ARRAYTP_H_
#define ARRAYTP_H_

#include <iostream>
#include <cstdlib>

template <class T, int n>
class ArrayTP
{
private:
    T ar[n];
public:
    ArrayTP() {};
    explicit ArrayTP(const T & v);
    virtual T & operator[](int i);
    virtual T operator[](int i) const;
};

template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
    for (int i = 0; i < n; i++)
        ar[i] = v;
}
template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i
            << " is out of range\n";
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
```

```
}

template <class T, int n>
T ArrayTP<T,n>::operator[](int i) const
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i
            << " is out of range\n";
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
}

#endif
```

请注意程序清单 14.17 中的模板头：

```
template <class T, int n>
```

关键字 class（或在这种上下文中等价的关键字 typename）指出 T 为类型参数，int 指出 n 的类型为 int。这种参数（指定特殊的类型而不是用作泛型名）称为非类型（non-type）或表达式（expression）参数。假设有下面的声明：

```
ArrayTP<double, 12> eggweights;
```

这将导致编译器定义名为 ArrayTP<double, 12>的类，并创建一个类型为 ArrayTP<double, 12>的 eggweight 对象。定义类时，编译器将使用 double 替换 T，使用 12 替换 n。

表达式参数有一些限制。表达式参数可以是整型、枚举、引用或指针。因此，double m 是不合法的，但 double * rm 和 double * pm 是合法的。另外，模板代码不能修改参数的值，也不能使用参数的地址。所以，在 ArrayTP 模板中不能使用诸如 n++和&n 等表达式。另外，实例化模板时，用作表达式参数的值必须是常量表达式。

与 Stack 中使用的构造函数方法相比，这种改变数组大小的方法有一个优点。构造函数方法使用的是通过 new 和 delete 管理的堆内存，而表达式参数方法使用的是为自动变量维护的内存栈。这样，执行速度将更快，尤其是在使用了很多小型数组时。

表达式参数方法的主要缺点是，每种数组大小都将生成自己的模板。也就是说，下面的声明将生成两个独立的类声明：

```
ArrayTP<double, 12> eggweights;
ArrayTP<double, 13> donuts;
```

但下面的声明只生成一个类声明，并将数组大小信息传递给类的构造函数：

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

另一个区别是，构造函数方法更通用，这是因为数组大小是作为类成员（而不是硬编码）存储在定义中的。这样可以将一种尺寸的数组赋给另一种尺寸的数组，也可以创建允许数组大小可变的类。

14.4.5 模板多功能性

可以将用于常规类的技术用于模板类。模板类可用作基类，也可用作组件类，还可用作其他模板的类型参数。例如，可以使用数组模板实现栈模板，也可以使用数组模板来构造数组——数组元素是基于栈模板的栈。即可以编写下面的代码：

```
template <typename T>   // or <class T>
class Array
{
```

```
private:
    T entry;
    ...
};

template <typename Type>
class GrowArray : public Array<Type> {...};  // inheritance

template <typename Tp>
class Stack
{
        Array<Tp> ar;        // use an Array<> as a component
       ...
};
...
Array < Stack<int> > asi;  // an array of stacks of int
```

在最后一条语句中，C++98 要求使用至少一个空白字符将两个>符号分开，以免与运算符>>混淆。C++11 不要求这样做。

1. 递归使用模板

另一个模板多功能性的例子是，可以递归使用模板。例如，对于前面的数组模板定义，可以这样使用它：

```
ArrayTP< ArrayTP<int,5>, 10> twodee;
```

这使得 twodee 是一个包含 10 个元素的数组，其中每个元素都是一个包含 5 个 int 元素的数组。与之等价的常规数组声明如下：

```
int twodee[10][5];
```

请注意，在模板语法中，维的顺序与等价的二维数组相反。程序清单 14.18 使用了这种方法，同时使用 ArrayTP 模板创建了一维数组，来分别保存这 10 个组（每组包含 5 个数）的总数和平均值。方法调用 cout.width(2)以两个字符的宽度显示下一个条目（如果整个数字的宽度不超过两个字符）。

程序清单 14.18　twod.cpp

```
// twod.cpp -- making a 2-d array
#include <iostream>
#include "arraytp.h"
int main(void)
{
    using std::cout;
    using std::endl;
    ArrayTP<int, 10> sums;
    ArrayTP<double, 10> aves;
    ArrayTP< ArrayTP<int,5>, 10> twodee;


    int i, j;

    for (i = 0; i < 10; i++)
    {
        sums[i] = 0;
        for (j = 0; j < 5; j++)
        {
            twodee[i][j] = (i + 1) * (j + 1);
```

```
            sums[i] += twodee[i][j];
        }
        aves[i] = (double) sums[i] / 10;
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 5; j++)
        {
            cout.width(2);
            cout << twodee[i][j] << ' ';
        }
        cout << ": sum = ";
        cout.width(3);
        cout  << sums[i] << ", average = " << aves[i] << endl;
    }

    cout << "Done.\n";

    return 0;
}
```

下面是程序清单 14.18 所示程序的输出。在 twodee 的 10 个元素（每个元素又是一个包含 5 个元素的数组）中，每个元素对应于 1 行：列出了每个元素包含的值、这些值的总和以及平均值。

```
 1  2  3  4  5 : sum =  15, average = 1.5
 2  4  6  8 10 : sum =  30, average = 3
 3  6  9 12 15 : sum =  45, average = 4.5
 4  8 12 16 20 : sum =  60, average = 6
 5 10 15 20 25 : sum =  75, average = 7.5
 6 12 18 24 30 : sum =  90, average = 9
 7 14 21 28 35 : sum = 105, average = 10.5
 8 16 24 32 40 : sum = 120, average = 12
 9 18 27 36 45 : sum = 135, average = 13.5
10 20 30 40 50 : sum = 150, average = 15
Done.
```

2. 使用多个类型参数

模板可以包含多个类型参数。例如，假设希望类可以保存两种值，则可以创建并使用 Pair 模板来保存两个不同的值（标准模板库提供了类似的模板，名为 pair）。程序清单 14.19 所示的小程序是一个这样的示例。其中，方法 first( ) const 和 second( ) const 报告存储的值，由于这两个方法返回 Pair 数据成员的引用，因此让您能够通过赋值重新设置存储的值。

程序清单 14.19　pairs.cpp

```
// pairs.cpp -- defining and using a Pair template
#include <iostream>
#include <string>
template <class T1, class T2>
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first();
    T2 & second();
```

```
    T1 first() const { return a; }
    T2 second() const { return b; }
    Pair(const T1 & aval, const T2 & bval) : a(aval), b(bval) { }
    Pair() {}
};

template<class T1, class T2>
T1 & Pair<T1,T2>::first()
{
    return a;
}
template<class T1, class T2>
T2 & Pair<T1,T2>::second()
{
    return b;
}

int main()
{
    using std::cout;
    using std::endl;
    using std::string;
    Pair<string, int> ratings[4] =
    {
        Pair<string, int>("The Purpled Duck", 5),
        Pair<string, int>("Jaquie's Frisco Al Fresco", 4),
        Pair<string, int>("Cafe Souffle", 5),
        Pair<string, int>("Bertie's Eats", 3)
    };

    int joints = sizeof(ratings) / sizeof (Pair<string, int>);
    cout << "Rating:\t Eatery\n";
    for (int i = 0; i < joints; i++)
        cout << ratings[i].second() << ":\t "
             << ratings[i].first() << endl;
    cout << "Oops! Revised rating:\n";
    ratings[3].first() = "Bertie's Fab Eats";
    ratings[3].second() = 6;
    cout << ratings[3].second() << ":\t "
         << ratings[3].first() << endl;
   return 0;
}
```

对于程序清单 14.19，需要注意的一点是，在 main( )中必须使用 Pair<string, int>来调用构造函数，并将它作为 sizeof 的参数。这是因为类名是 Pair<string, int>，而不是 Pair。另外，Pair<char *, double>是另一个完全不同的类的名称。

下面是程序清单 14.19 所示程序的输出：

```
Rating:  Eatery
5:       The Purpled Duck
4:       Jaquie's Frisco Al Fresco
5:       Cafe Souffle
3:       Bertie's Eats
Oops! Revised rating:
6:       Bertie's Fab Eats
```

3. 默认类型模板参数

类模板的另一项新特性是，可以为类型参数提供默认值：

```
template <class T1, class T2 = int> class Topo {...};
```

这样，如果省略 T2 的值，编译器将使用 int：

```
Topo<double, double> m1; // T1 is double, T2 is double
Topo<double> m2;         // T1 is double, T2 is int
```

第 16 章将讨论的标准模板库经常使用该特性，将默认类型设置为类。

虽然可以为类模板类型参数提供默认值，但不能为函数模板参数提供默认值。然而，可以为非类型参数提供默认值，这对于类模板和函数模板都是适用的。

14.4.6 模板的具体化

类模板与函数模板很相似，因为可以有隐式实例化、显式实例化和显式具体化，它们统称为具体化（specialization）。模板以泛型的方式描述类，而具体化是使用具体的类型生成类声明。

1. 隐式实例化

到目前为止，本章所有的模板示例使用的都是隐式实例化（implicit instantiation），即它们声明一个或多个对象，指出所需的类型，而编译器使用通用模板提供的处方生成具体的类定义：

```
ArrayTP<int, 100> stuff; // implicit instantiation
```

编译器在需要对象之前，不会生成类的隐式实例化：

```
ArrayTP<double, 30> * pt;     // a pointer, no object needed yet
pt = new ArrayTP<double, 30>; // now an object is needed
```

第二条语句导致编译器生成类定义，并根据该定义创建一个对象。

2. 显式实例化

当使用关键字 template 并指出所需类型来声明类时，编译器将生成类声明的显式实例化（explicit instantiation）。声明必须位于模板定义所在的名称空间中。例如，下面的声明将 ArrayTP<string, 100>声明为一个类：

```
template class ArrayTP<string, 100>; // generate ArrayTP<string, 100> class
```

在这种情况下，虽然没有创建或提及类对象，编译器也将生成类声明（包括方法定义）。和隐式实例化一样，也将根据通用模板来生成具体化。

3. 显式具体化

显式具体化（explicit specialization）是特定类型（用于替换模板中的泛型）的定义。有时候，可能需要在为特殊类型实例化时，对模板进行修改，使其行为不同。在这种情况下，可以创建显式具体化。例如，假设已经为用于表示排序后数组的类（元素在加入时被排序）定义了一个模板：

```
template <typename T>
class SortedArray
{
     ...// details omitted
};
```

另外，假设模板使用>运算符来对值进行比较。对于数字，这管用；如果 T 表示一种类，则只要定义了 T::operator>( )方法，这也管用；但如果 T 是由 const char *表示的字符串，这将不管用。实际上，模板倒是可以正常工作，但字符串将按地址（按照字母顺序）排序。这要求类定义使用 strcmp( )，而不是>来对值进行比较。在这种情况下，可以提供一个显式模板具体化，这将采用为具体类型定义的模板，而不是为泛型定义的模板。当具体化模板和通用模板都与实例化请求匹配时，编译器将使用具体化版本。

具体化类模板定义的格式如下：

```
template <> class Classname<specialized-type-name> { ... };
```

早期的编译器可能只能识别早期的格式，这种格式不包括前缀 template<>：

```
class Classname<specialized-type-name> { ... };
```

要使用新的表示法提供一个专供 const char *类型使用的 SortedArray 模板，可以使用类似于下面的代码：

```
template <> class SortedArray<const char char *>
{
    ...// details omitted
};
```

其中的实现代码将使用 strcmp( )（而不是>）来比较数组值。现在，当请求 const char *类型的 SortedArray 模板时，编译器将使用上述专用的定义，而不是通用的模板定义：

```
SortedArray<int> scores;          // use general definition
SortedArray<const char *> dates;  // use specialized definition
```

4. 部分具体化

C++还允许部分具体化（partial specialization），即部分限制模板的通用性。例如，部分具体化可以给类型参数之一指定具体的类型：

```
// general template
    template <class T1, class T2> class Pair {...};
// specialization with T2 set to int
    template <class T1> class Pair<T1, int> {...};
```

关键字 template 后面的<>声明的是没有被具体化的类型参数。因此，上述第二个声明将 T2 具体化为 int，但 T1 保持不变。注意，如果指定所有的类型，则<>内将为空，这将导致显式具体化：

```
// specialization with T1 and T2 set to int
    template <> class Pair<int, int> {...};
```

如果有多个模板可供选择，编译器将使用具体化程度最高的模板。给定上述三个模板，情况如下：

```
Pair<double, double> p1; // use general Pair template
Pair<double, int> p2;    // use Pair<T1, int> partial specialization
Pair<int, int> p3;       // use Pair<int, int> explicit specialization
```

也可以通过为指针提供特殊版本来部分具体化现有的模板：

```
template<class T>        // general version
class Feeb { ... };
template<class T*>       // pointer partial specialization
class Feeb { ... };      // modified code
```

如果提供的类型不是指针，则编译器将使用通用版本；如果提供的是指针，则编译器将使用指针具体化版本：

```
Feeb<char> fb1;          // use general Feeb template, T is char
Feeb<char *> fb2;        // use Feeb T* specialization, T is char
```

如果没有进行部分具体化，则第二个声明将使用通用模板，将 T 转换为 char *类型。如果进行了部分具体化，则第二个声明将使用具体化模板，将 T 转换为 char。

部分具体化特性使得能够设置各种限制。例如，可以这样做：

```
// general template
    template <class T1, class T2, class T3> class Trio{...};
// specialization with T3 set to T2
    template <class T1, class T2> class Trio<T1, T2, T2> {...};
// specialization with T3 and T2 set to T1*
    template <class T1> class Trio<T1, T1*, T1*> {...};
```

给定上述声明，编译器将作出如下选择：

```
Trio<int, short, char *> t1; // use general template
Trio<int, short> t2; // use Trio<T1, T2, T2>
Trio<char, char *, char *> t3; use Trio<T1, T1*, T1*>
```

14.4.7 成员模板

模板可用作结构、类或模板类的成员。要完全实现 STL 的设计，必须使用这项特性。程序清单 14.20 是一个简短的模板类示例，该模板类将另一个模板类和模板函数作为其成员。

程序清单 14.20 tempmemb.cpp

```
// tempmemb.cpp -- template members
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class beta
{
private:
    template <typename V>  // nested template class member
    class hold
    {
    private:
        V val;
    public:
        hold(V v  = 0) : val(v) {}
        void show() const { cout << val << endl; }
        V Value() const { return val; }
    };
    hold<T> q;              // template object
    hold<int> n;            // template object
public:
    beta( T t, int i) : q(t), n(i) {}
    template<typename U>   // template method
    U blab(U u, T t) { return (n.Value() + q.Value()) * u / t; }
    void Show() const { q.show(); n.show();}
};

int main()
{
    beta<double> guy(3.5, 3);
    cout << "T was set to double\n";
    guy.Show();
    cout << "V was set to T, which is double, then V was set to int\n";
    cout << guy.blab(10, 2.3) << endl;
    cout << "U was set to int\n";
    cout << guy.blab(10.0, 2.3) << endl;
    cout << "U was set to double\n";
    cout << "Done\n";
    return 0;
}
```

在程序清单 14.20 中，hold 模板是在私有部分声明的，因此只能在 beta 类中访问它。beta 类使用 hold 模板声明了两个数据成员：

```
hold<T> q;              // template object
hold<int> n;            // template object
```

n 是基于 int 类型的 hold 对象，而 q 成员是基于 T 类型（beta 模板参数）的 hold 对象。在 main( )中，下述声明使得 T 表示的是 double，因此 q 的类型为 hold<double>：

```
beta<double> guy(3.5, 3);
```

blab( )方法的 U 类型由该方法被调用时的参数值显式确定，T 类型由对象的实例化类型确定。在这个例子中，guy 的声明将 T 的类型设置为 double，而下述方法调用的第一个参数将 U 的类型设置为 int（参数 10 对应的类型）：

```
cout << guy.blab(10, 2.5) << endl;
```

因此，虽然混合类型引起的自动类型转换导致 blab( )中的计算以 double 类型进行，但返回值的类型为 U（即 int），因此它被截断为 28，如下面的程序输出所示：

```
T was set to double
3.5
3
V was set to T, which is double, then V was set to int
28
U was set to int
28.2609
U was set to double
Done
```

注意到调用 guy.blab( )时，使用 10.0 代替了 10，因此 U 被设置为 double，这使得返回类型为 double，因此输出为 28.2608。

正如前面指出的，guy 对象的声明将第二个参数的类型设置为 double。与第一个参数不同的是，第二个参数的类型不是由函数调用设置的。例如，下面的语句仍将 blah( )实现为 blah(int, double)，并根据常规函数原型规则将 3 转换为类型 double：

```
cout << guy.blab(10, 3) << endl;
```

可以在 beta 模板中声明 hold 类和 blah 方法，并在 beta 模板的外面定义它们。然而，很老的编译器根本不接受模板成员，而另一些编译器接受模板成员（如程序清单 14.20 所示），但不接受类外面的定义。然而，如果所用的编译器接受类外面的定义，则在 beta 模板之外定义模板方法的代码如下：

```
template <typename T>
class beta
{
private:
    template <typename V>  // declaration
    class hold;
    hold<T> q;
    hold<int> n;
public:
    beta( T t, int i) : q(t), n(i) {}
    template<typename U>   // declaration
    U blab(U u, T t);
    void Show() const { q.show(); n.show();}
};

// member definition
template <typename T>
  template<typename V>
    class beta<T>::hold
    {
```

```
{
private:
    V val;
public:
    hold(V v  = 0) : val(v) {}
    void show() const { std::cout << val << std::endl; }
    V Value() const { return val; }
};

// member definition
template <typename T>
  template <typename U>
    U beta<T>::blab(U u, T t)
    {
       return (n.Value() + q.Value()) * u / t;
    }
```

上述定义将 T、V 和 U 用作模板参数。因为模板是嵌套的，因此必须使用下面的语法：

```
template <typename T>
  template <typename V>
```

而不能使用下面的语法：

```
template<typename T, typename V>
```

定义还必须指出 hold 和 blab 是 beta<T>类的成员，这是通过使用作用域解析运算符来完成的。

14.4.8 将模板用作参数

您知道，模板可以包含类型参数（如 typename T）和非类型参数（如 int n）。模板还可以包含本身就是模板的参数，这种参数是模板新增的特性，用于实现 STL。

在程序清单 14.21 所示的示例中，开头的代码如下：

```
template <template <typename T> class Thing>
class Crab
```

模板参数是 template <typename T>class Thing，其中 template <typename T>class 是类型，Thing 是参数。这意味着什么呢？假设有下面的声明：

```
Crab<King> legs;
```

为使上述声明被接受，模板参数 King 必须是一个模板类，其声明与模板参数 Thing 的声明匹配：

```
template <typename T>
class King {...};
```

在程序清单 14.21 中，Crab 的声明声明了两个对象：

```
Thing<int> s1;
Thing<double> s2;
```

前面的 legs 声明将用 King<int>替换 Thing<int>，用 King<double>替换 Thing<double>。然而，程序清单 14.21 包含下面的声明：

```
Crab<Stack> nebula;
```

因此，Thing<int>将被实例化为 Stack<int>，而 Thing<double>将被实例化为 Stack<double>。总之，模板参数 Thing 将被替换为声明 Crab 对象时被用作模板参数的模板类型。

Crab 类的声明对 Thing 代表的模板类做了另外 3 个假设，即这个类包含一个 push( )方法，包含一个 pop( )方法，且这些方法有特定的接口。Crab 类可以使用任何与 Thing 类型声明匹配，并包含方法 push( )和 pop( )的模板类。本章恰巧有一个这样的类——stacktp.h 中定义的 Stack 模板，因此这个例子将使用它。

程序清单 14.21　tempparm.cpp

```
// tempparm.cpp - templates as parameters
#include <iostream>
#include "stacktp.h"

template <template <typename T> class Thing>
class Crab
{
private:
    Thing<int> s1;
    Thing<double> s2;
public:
    Crab() {};
    // assumes the thing class has push() and pop() members
    bool push(int a, double x) { return s1.push(a) && s2.push(x); }
    bool pop(int & a, double & x){ return s1.pop(a) && s2.pop(x); }
};

int main()
{
    using std::cout;
    using std::cin;
    using std::endl;
    Crab<Stack> nebula;
// Stack must match template <typename T> class thing
    int ni;
    double nb;
    cout << "Enter int double pairs, such as 4 3.5 (0 0 to end):\n";
    while (cin>> ni >> nb && ni > 0 && nb > 0)
    {
        if (!nebula.push(ni, nb))
            break;
    }

    while (nebula.pop(ni, nb))
           cout << ni << ", " << nb << endl;
    cout << "Done.\n";

    return 0;
}
```

下面是程序清单 14.21 所示程序的运行情况：

```
Enter int double pairs, such as 4 3.5 (0 0 to end):
50 22.48
25 33.87
60 19.12
0 0
60, 19.12
25, 33.87
50, 22.48
Done.
```

可以混合使用模板参数和常规参数，例如，Crab 类的声明可以像下面这样打头：

```
template <template <typename T> class Thing, typename U, typename V>
class Crab
```

```
{
private:
    Thing<U> s1;
    Thing<V> s2;
...
```

现在，成员 s1 和 s2 可存储的数据类型为泛型，而不是用硬编码指定的类型。这要求将程序中 nebula 的声明修改成下面这样：

```
Crab<Stack, int, double> nebula; // T=Stack, U=int, V=double
```

模板参数 T 表示一种模板类型，而类型参数 U 和 V 表示非模板类型。

14.4.9 模板类和友元

模板类声明也可以有友元。模板的友元分 3 类：

- 非模板友元；
- 约束（bound）模板友元，即友元的类型取决于类被实例化时的类型；
- 非约束（unbound）模板友元，即友元的所有具体化都是类的每一个具体化的友元。

下面分别介绍它们。

1. 模板类的非模板友元函数

在模板类中将一个常规函数声明为友元：

```
template <class T>
class HasFriend
{
public:
    friend void counts();      // friend to all HasFriend instantiations
    ...
};
```

上述声明使 counts( )函数成为模板所有实例化的友元。例如，它将是类 hasFriend<int>和 HasFriend<string>的友元。

counts( )函数不是通过对象调用的（它是友元，不是成员函数），也没有对象参数，那么它如何访问 HasFriend 对象呢？有很多种可能性。它可以访问全局对象；可以使用全局指针访问非全局对象；可以创建自己的对象；可以访问独立于对象的模板类的静态数据成员。

假设要为友元函数提供模板类参数，可以如下所示来进行友元声明吗？

```
friend void report(HasFriend &);    // possible?
```

答案是不可以。原因是不存在 HasFriend 这样的对象，而只有特定的具体化，如 HasFriend<short>。要提供模板类参数，必须指明具体化。例如，可以这样做：

```
template <class T>
class HasFriend
{
    friend void report(HasFriend<T> &); // bound template friend
    ...
};
```

为理解上述代码的功能，想想声明一个特定类型的对象时，将生成的具体化：

```
HasFriend<int> hf;
```

编译器将用 int 替代模板参数 T，因此友元声明的格式如下：

```
class HasFriend<int>
{
    friend void report(HasFriend<int> &); // bound template friend
    ...
};
```

也就是说，带 HasFriend<int>参数的 report( )将成为 HasFriend<int>类的友元。同样，带 HasFriend<double>参数的 report( )将是 report( )的一个重载版本——它是 Hasfriend<double>类的友元。

注意，report( )本身并不是模板函数，而只是使用一个模板作参数。这意味着必须为要使用的友元定义显式具体化：

```
void report(HasFriend<short> &) {...}; // explicit specialization for short
void report(HasFriend<int> &) {...};   // explicit specialization for int
```

程序清单 14.22 说明了上面几点。HasFriend 模板有一个静态成员 ct。这意味着这个类的每一个特定的具体化都将有自己的静态成员。count( )方法是所有 HasFriend 具体化的友元，它报告两个特定的具体化（HasFriend<int>和 HasFriend<double>）的 ct 的值。该程序还提供两个 report( )函数，它们分别是某个特定 HasFriend 具体化的友元。

程序清单 14.22　frnd2tmp.cpp

```
// frnd2tmp.cpp -- template class with non-template friends
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class HasFriend
{
private:
    T item;
    static int ct;
public:
    HasFriend(const T & i) : item(i) {ct++;}
    ~HasFriend()  {ct--; }
    friend void counts();
    friend void reports(HasFriend<T> &); // template parameter
};

// each specialization has its own static data member
template <typename T>
int HasFriend<T>::ct = 0;

// non-template friend to all HasFriend<T> classes
void counts()
{
    cout << "int count: " << HasFriend<int>::ct << "; ";
    cout << "double count: " << HasFriend<double>::ct << endl;
}

// non-template friend to the HasFriend<int> class
void reports(HasFriend<int> & hf)
{
    cout <<"HasFriend<int>: " << hf.item << endl;
}

// non-template friend to the HasFriend<double> class
void reports(HasFriend<double> & hf)
{
    cout <<"HasFriend<double>: " << hf.item << endl;
}

int main()
```

```
{
    cout << "No objects declared: ";
    counts();
    HasFriend<int> hfi1(10);
    cout << "After hfi1 declared: ";
    counts();
    HasFriend<int> hfi2(20);
    cout << "After hfi2 declared: ";
    counts();
    HasFriend<double> hfdb(10.5);
    cout << "After hfdb declared: ";
    counts();
    reports(hfi1);
    reports(hfi2);
    reports(hfdb);

    return 0;
}
```

有些编译器将对您使用非模板友元发出警告。下面是程序清单 14.22 所示程序的输出：

```
No objects declared: int count: 0; double count: 0
After hfi1 declared: int count: 1; double count: 0
After hfi2 declared: int count: 2; double count: 0
After hfdb declared: int count: 2; double count: 1
HasFriend<int>: 10
HasFriend<int>: 20
HasFriend<double>: 10.5
```

2. 模板类的约束模板友元函数

可以修改前一个示例，使友元函数本身成为模板。具体地说，为约束模板友元作准备，要使类的每一个具体化都获得与友元匹配的具体化。这比非模板友元复杂些，包含以下 3 步。

首先，在类定义的前面声明每个模板函数。

```
template <typename T> void counts();
template <typename T> void report(T &);
```

然后，在函数中再次将模板声明为友元。这些语句根据类模板参数的类型声明具体化：

```
template <typename TT>
class HasFriendT
{
...
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};
```

声明中的<>指出这是模板具体化。对于 report( )，<>可以为空，因为可以从函数参数推断出如下模板类型参数：

```
HasFriendT<TT>
```

然而，也可以使用：

```
report<HasFriendT<TT> >(HasFriendT<TT> &)
```

但 counts( )函数没有参数，因此必须使用模板参数语法（<TT>）来指明其具体化。还需要注意的是，TT 是 HasFriendT 类的参数类型。

同样，理解这些声明的最佳方式也是设想声明一个特定具体化的对象时，它们将变成什么样。例如，假设声明了这样一个对象：

```
HasFriendT<int> squack;
```

编译器将用 int 替换 TT，并生成下面的类定义：

```
class HasFriendT<int>
{
...
    friend void counts<int>();
    friend void report<>(HasFriendT<int> &);
};
```

基于 TT 的具体化将变为 int，基于 HasFriend<TT>的具体化将变为 HasFriend<int>。因此，模板具体化 counts<int>( )和 report<HasFriendT<int> >( )被声明为 HasFriendT<int>类的友元。

程序必须满足的第三个要求是，为友元提供模板定义。程序清单 14.23 说明了这 3 个方面。请注意，程序清单 14.22 包含 1 个 count( )函数，它是所有 HasFriend 类的友元；而程序清单 14.23 包含两个 count( )函数，它们分别是某个被实例化的类类型的友元。因为 count( )函数调用没有可被编译器用来推断出所需具体化的函数参数，所以这些调用使用 count<int>和 coount<double>( )指明具体化。但对于 report( )调用，编译器可以从参数类型推断出要使用的具体化。使用<>格式也能获得同样的效果：

```
report<HasFriendT<int> >(hfi2);  // same as report(hfi2);
```

程序清单 14.23　tmp2tmp.cpp

```
// tmp2tmp.cpp -- template friends to a template class
#include <iostream>
using std::cout;
using std::endl;
// template prototypes
template <typename T> void counts();
template <typename T> void report(T &);

// template class
template <typename TT>
class HasFriendT
{
private:
    TT item;
    static int ct;
public:
    HasFriendT(const TT & i) : item(i) {ct++;}
    ~HasFriendT() { ct--; }
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};

template <typename T>
int HasFriendT<T>::ct = 0;

// template friend functions definitions
template <typename T>
void counts()
{
    cout << "template size: " << sizeof(HasFriendT<T>) << "; ";
    cout << "template counts(): " << HasFriendT<T>::ct << endl;
}

template <typename T>
void report(T & hf)
{
```

```
    cout << hf.item << endl;
}

int main()
{
    counts<int>();
    HasFriendT<int> hfi1(10);
    HasFriendT<int> hfi2(20);
    HasFriendT<double> hfdb(10.5);
    report(hfi1);  // generate report(HasFriendT<int> &)
    report(hfi2);  // generate report(HasFriendT<int> &)
    report(hfdb);  // generate report(HasFriendT<double> &)
    cout << "counts<int>() output:\n";
    counts<int>();
    cout << "counts<double>() output:\n";
    counts<double>();

    return 0;
}
```

下面是程序清单 14.23 所示程序的输出：

```
template size: 4; template counts(): 0
10
20
10.5
counts<int>() output:
template size: 4; template counts(): 2
counts<double>() output:
template size: 8; template counts(): 1
```

正如您看到的，counts<double>和 counts<int>报告的模板大小不同，这表明每种 T 类型都有自己的友元函数 count( )。

3. 模板类的非约束模板友元函数

前一节中的约束模板友元函数是在类外面声明的模板的具体化。int 类具体化获得 int 函数具体化，依此类推。通过在类内部声明模板，可以创建非约束友元函数，即每个函数具体化都是每个类具体化的友元。对于非约束友元，友元模板类型参数与模板类类型参数是不同的：

```
template <typename T>
class ManyFriend
{
...
    template <typename C, typename D> friend void show2(C &, D &);
};
```

程序清单 14.24 是一个使用非约束友元的例子。其中，函数调用 show2（hfi1，hfi2）与下面的具体化匹配：

```
void show2<ManyFriend<int> &, ManyFriend<int> &>
          (ManyFriend<int> & c, ManyFriend<int> & d);
```

因为它是所有 ManyFriend 具体化的友元，所以能够访问所有具体化的 item 成员，但它只访问了 ManyFriend<int>对象。

同样，show2(hfd, hfi2)与下面具体化匹配：

```
void show2<ManyFriend<double> &, ManyFriend<int> &>
          (ManyFriend<double> & c, ManyFriend<int> & d);
```

它也是所有 ManyFriend 具体化的友元，并访问了 ManyFriend<int>对象的 item 成员和 ManyFriend<double>对象的 item 成员。

程序清单 14.24　manyfrnd.cpp

```
// manyfrnd.cpp -- unbound template friend to a template class
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class ManyFriend
{
private:
    T item;
public:
    ManyFriend(const T & i) : item(i) {}
    template <typename C, typename D> friend void show2(C &, D &);
};

template <typename C, typename D> void show2(C & c, D & d)
{
    cout << c.item << ", " << d.item << endl;
}

int main()
{
    ManyFriend<int> hfi1(10);
    ManyFriend<int> hfi2(20);
    ManyFriend<double> hfdb(10.5);
    cout << "hfi1, hfi2: ";
    show2(hfi1, hfi2);
    cout << "hfdb, hfi2: ";
    show2(hfdb, hfi2);

    return 0;
}
```

程序清单 14.24 所示程序的输出如下：

```
hfi1, hfi2: 10, 20
hfdb, hfi2: 10.5, 20
```

14.4.10　模板别名（C++11）

如果能为类型指定别名，将很方便，在模板设计中尤其如此。可使用 typedef 为模板具体化指定别名：

```
// define three typedef aliases
typedef std::array<double, 12> arrd;
typedef std::array<int, 12> arri;
typedef std::array<std::string, 12> arrst;
arrd gallons;  // gallons is type std::array<double, 12>
arri days;     // days is type std::array<int, 12>
arrst months;  // months is type std::array<std::string, 12>
```

但如果您经常编写类似于上述 typedef 的代码，您可能怀疑要么自己忘记了可简化这项任务的 C++功能，要么 C++没有提供这样的功能。C++11 新增了一项功能——使用模板提供一系列别名，如下所示：

```
template<typename T>
  using arrtype = std::array<T,12>;  // template to create multiple aliases
```

这将 arrtype 定义为一个模板别名，可使用它来指定类型，如下所示：

```
arrtype<double> gallons;       // gallons is type std::array<double, 12>
arrtype<int> days;             // days is type std::array<int, 12>
arrtype<std::string> months;   // months is type std::array<std::string, 12>
```

总之，arrtype<T>表示类型 std::array<T, 12>。

C++11 允许将语法 using =用于非模板。用于非模板时，这种语法与常规 typedef 等价：

```
typedef const char * pc1;        // typedef syntax
using pc2 = const char *;        // using = syntax
typedef const int *(*pa1)[10];   // typedef syntax
using pa2 = const int *(*)[10]; // using = syntax
```

习惯这种语法后，您可能发现其可读性更强，因为它让类型名和类型信息更清晰。

C++11 新增的另一项模板功能是可变参数模板（variadic template），让您能够定义这样的模板类和模板函数，即可接受可变数量的参数。这个主题将在第 18 章介绍。

14.5 总结

C++提供了几种重用代码的手段。第 13 章介绍的公有继承能够建立 is-a 关系，这样派生类可以重用基类的代码。私有继承和保护继承也使得能够重用基类的代码，但建立的是 has-a 关系。使用私有继承时，基类的公有成员和保护成员将成为派生类的私有成员；使用保护继承时，基类的公有成员和保护成员将成为派生类的保护成员。无论使用哪种继承，基类的公有接口都将成为派生类的内部接口。这有时候被称为继承实现，但并不继承接口，因为派生类对象不能显式地使用基类的接口。因此，不能将派生对象看作是一种基类对象。由于这个原因，在不进行显式类型转换的情况下，基类指针或引用将不能指向派生类对象。

还可以通过开发包含对象成员的类来重用类代码。这种方法被称为包含、层次化或组合，它建立的也是 has-a 关系。与私有继承和保护继承相比，包含更容易实现和使用，所以通常优先采用这种方式。然而，私有继承和保护继承比包含有一些不同的功能。例如，继承允许派生类访问基类的保护成员；还允许派生类重新定义从基类那里继承的虚函数。因为包含不是继承，所以通过包含来重用类代码时，不能使用这些功能。另一方面，如果需要使用某个类的几个对象，则用包含更适合。例如，State 类可以包含一组 County 对象。

多重继承（MI）使得能够在类设计中重用多个类的代码。私有 MI 或保护 MI 建立 has-a 关系，而公有 MI 建立 is-a 关系。MI 会带来一些问题，即多次定义同一个名称，继承多个基类对象。可以使用类限定符来解决名称二义性的问题，使用虚基类来避免继承多个基类对象的问题。但使用虚基类后，就需要为编写构造函数初始化列表以及解决二义性问题引入新的规则。

类模板使得能够创建通用的类设计，其中类型（通常是成员类型）由类型参数表示。典型的模板如下：

```
template <class T>
class Ic
{
      T v;
      ...
public:
      Ic(const T & val) : v(val) { }
      ...
};
```

其中，T 是类型参数，用作以后将指定的实际类型的占位符（这个参数可以是任意有效的 C++名称，

但通常使用 T 和 Type）。在这种环境下，也可以使用 typename 代替 class：

```
template <typename T>  // same as template <class T>
class Rev {...} ;
```

类定义（实例化）在声明类对象并指定特定类型时生成。例如，下面的声明导致编译器生成类声明，用声明中的实际类型 short 替换模板中的所有类型参数 T：

```
class Ic<short> sic;    // implicit instantiation
```

这里，类名为 Ic<short>，而不是 Ic。Ic<short>称为模板具体化。具体地说，这是一个隐式实例化。

使用关键字 template 声明类的特定具体化时，将发生显式实例化：

```
template class IC<int>;  // explicit instantiation
```

在这种情况下，编译器将使用通用模板生成一个 int 具体化——Ic<int>，虽然尚未请求这个类的对象。

可以提供显式具体化——覆盖模板定义的具体类声明。方法是以 template<>打头，然后是模板类名称，再加上尖括号（其中包含要具体化的类型）。例如，为字符指针提供专用 Ic 类的代码如下：

```
template <> class Ic<char *>.
{
      char * str;
      ...
public:
      Ic(const char * s) : str(s) { }
      ...
};
```

这样，下面这样的声明将为 chic 使用专用定义，而不是通用模板：

```
class Ic<char *> chic;
```

类模板可以指定多个泛型，也可以有非类型参数：

```
template <class T, class TT, int n>
class Pals {...};
```

下面的声明将生成一个隐式实例化，用 double 代替 T，用 string 代替 TT，用 6 代替 n：

```
Pals<double, string, 6> mix;
```

类模板还可以包含本身就是模板的参数：

```
template < template <typename T> class CL, typename U, int z>
class Trophy {...};
```

其中 z 是一个 int 值，U 为类型名，CL 为一个使用 template<typename, T>声明的类模板。

类模板可以被部分具体化：

```
template <class T> Pals<T, T, 10> {...};
template <class T, class TT> Pals<T, TT, 100> {...};
template <class T, int n> Pals <T, T*, n> {...};
```

第一个声明为两个类型相同，且 n 的值为 6 的情况创建了一个具体化。同样，第二个声明为 n 等于 100 的情况创建一个具体化；第三个声明为第二个类型是指向第一个类型的指针的情况创建了一个具体化。

模板类可用作其他类、结构和模板的成员。

所有这些机制的目的都是为了让程序员能够重用经过测试的代码，而不用手工复制它们。这样可以简化编程工作，提供程序的可靠性。

14.6　复习题

1．以 A 栏的类为基类时，B 栏的类采用公有派生还是私有派生更合适。

A	B
class Bear	class PolarBear
class Kitchen	class Home
class Person	class Programmer
class Person	class HorseAndJockey
class Person, class Automobile	class Driver

2．假设有下面的定义：

```
class Frabjous {
private:
      char fab[20];
public:
      Frabjous(const char * s = "C++") : fab(s) { }
      virtual void tell() { cout << fab; }
};

class Gloam {
private:
      int glip;
      Frabjous fb;
public:
      Gloam(int g = 0, const char * s = "C++");
      Gloam(int g, const Frabjous & f);
      void tell();
};
```

假设 Gloam 版本的 tell( )应显示 glip 和 fb 的值，请为这 3 个 Gloam 方法提供定义。

3．假设有下面的定义：

```
class Frabjous {
private:
      char fab[20];
public:
      Frabjous(const char * s = "C++") : fab(s) { }
      virtual void tell() { cout << fab; }
};

class Gloam : private Frabjous{
private:
      int glip;
public:
      Gloam(int g = 0, const char * s = "C++");
      Gloam(int g, const Frabjous & f);
      void tell();
};
```

假设 Gloam 版本的 tell( )应显示 glip 和 fab 的值，请为这 3 个 Gloam 方法提供定义。

4．假设有下面的定义，它是基于程序清单 14.13 中的 Stack 模板和程序清单 14.10 中的 Woker 类的：

```
Stack<Worker *> sw;
```

请写出将生成的类声明。只实现类声明，不实现非内联类方法。

5．使用本章中的模板定义对下面的内容进行定义：

- string 对象数组；
- double 数组栈；

- 指向 Worker 对象的指针的栈数组。

程序清单 14.18 生成了多少个模板类定义？

6．指出虚基类与非虚基类之间的区别。

14.7　编程练习

1．Wine 类有一个 string 类对象成员（参见第 4 章）和一个 Pair 对象（参见本章）；其中前者用于存储葡萄酒的名称，而后者有 2 个 valarray<int>对象（参见本章），这两个 valarray<int>对象分别保存了葡萄酒的酿造年份和该年生产的瓶数。例如，Pair 的第 1 个 valarray<int>对象可能为 1988、1992 和 1996 年，第 2 个 valarray<int>对象可能为 24、48 和 144 瓶。Wine 最好有 1 个 int 成员用于存储年数。另外，一些 typedef 可能有助于简化编程工作：

```
typedef std::valarray<int> ArrayInt;
typedef Pair<ArrayInt, ArrayInt> PairArray;
```

这样，PairArray 表示的是类型 Pair<std::valarray<int>, std::valarray<int> >。使用包含来实现 Wine 类，并用一个简单的程序对其进行测试。Wine 类应该有一个默认构造函数以及如下构造函数：

```
// initialize label to l, number of years to y,
// vintage years to yr[], bottles to bot[]
Wine(const char * l, int y, const int yr[], const int bot[]);
// initialize label to l, number of years to y,
// create array objects of length y
Wine(const char * l, int y);
```

Wine 类应该有一个 GetBottles( )方法，它根据 Wine 对象能够存储几种年份（y），提示用户输入年份和瓶数。方法 Label( )返回一个指向葡萄酒名称的引用。sum( )方法返回 Pair 对象中第二个 valarray<int>对象中的瓶数总和。

测试程序应提示用户输入葡萄酒名称、元素个数以及每个元素存储的年份和瓶数等信息。程序将使用这些数据来构造一个 Wine 对象，然后显示对象中保存的信息。

下面是一个简单的测试程序：

```
// pe14-1.cpp  -- using Wine class with containment
#include <iostream>
#include "winec.h"

int main ( void )
{
    using std::cin;
    using std::cout;
    using std::endl;

    cout << "Enter name of wine: ";
    char lab[50];
    cin.getline(lab, 50);
    cout << "Enter number of years: ";
    int yrs;
    cin >> yrs;

    Wine holding(lab, yrs); // store label, years, give arrays yrs elements
    holding.GetBottles();   // solicit input for year, bottle count
    holding.Show();         // display object contents

    const int YRS = 3;
```

```
    int y[YRS] = {1993, 1995, 1998};
    int b[YRS] = { 48, 60, 72};
    // create new object, initialize using data in arrays y and b
    Wine more("Gushing Grape Red",YRS, y, b);
    more.Show();
    cout << "Total bottles for " << more.Label() // use Label() method
         << ": " << more.sum() << endl;          // use sum() method
    cout << "Bye\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Enter name of wine: Gully Wash
Enter number of years: 4
Enter Gully Wash data for 4 year(s):
Enter year: 1988
Enter bottles for that year: 42
Enter year: 1994
Enter bottles for that year: 58
Enter year: 1998
Enter bottles for that year: 122
Enter year: 2001
Enter bottles for that year: 144
Wine: Gully Wash
        Year    Bottles
        1988    42
        1994    58
        1998    122
        2001    144
Wine: Gushing Grape Red
        Year    Bottles
        1993    48
        1995    60
        1998    72
Total bottles for Gushing Grape Red: 180
Bye
```

2．采用私有继承而不是包含来完成编程练习 1。同样，一些 typedef 可能会有所帮助，另外，您可能还需要考虑诸如下面这样的语句的含义：

```
PairArray::operator=(PairArray(ArrayInt(),ArrayInt()));
cout  << (const string &)(*this);
```

您设计的类应该可以使用编程练习 1 中的测试程序进行测试。

3．定义一个 QueueTp 模板。然后在一个类似于程序清单 14.12 的程序中创建一个指向 Worker 的指针队列（参见程序清单 14.10 中的定义），并使用该队列来测试它。

4．Person 类保存人的名和姓。除构造函数外，它还有 Show( )方法，用于显示名和姓。Gunslinger 类以 Person 类为虚基类派生而来，它包含一个 Draw( )成员，该方法返回一个 double 值，表示枪手的拔枪时间。这个类还包含一个 int 成员，表示枪手枪上的刻痕数。最后，这个类还包含一个 Show( )函数，用于显示所有这些信息。

PokerPlayer 类以 Person 类为虚基类派生而来。它包含一个 Draw( )成员，该函数返回一个 1～52 的随机数，用于表示扑克牌的值（也可以定义一个 Card 类，其中包含花色和面值成员，然后让 Draw( )返回一个 Card 对象）。PokerPlayer 类使用 Person 类的 show( )函数。BadDude( )类从 Gunslinger 和 PokerPlayer 类公有派生而来。它包含 Gdraw( )成员（返回坏蛋拔枪的时间）和 Cdraw( )成员（返回下一张扑克牌），另外

还有一个合适的 Show( )函数。请定义这些类和方法以及其他必要的方法（如用于设置对象值的方法），并使用一个类似于程序清单 14.12 的简单程序对它们进行测试。

5．下面是一些类声明：

```
// emp.h -- header file for abstr_emp class and children

#include <iostream>
#include <string>

class abstr_emp
{
private:
    std::string fname;    // abstr_emp's first name
    std::string lname;    // abstr_emp's last name
    std::string job;
public:
    abstr_emp();
    abstr_emp(const std::string & fn, const std::string &  ln,
             const std::string &  j);
    virtual void ShowAll() const;    // labels and shows all data
    virtual void SetAll();        // prompts user for values
    friend std::ostream &
             operator<<(std::ostream & os, const abstr_emp & e);
    // just displays first and last name
    virtual ~abstr_emp() = 0;         // virtual base class
};

class employee : public abstr_emp
{
public:
    employee();
    employee(const std::string & fn, const std::string &  ln,
             const std::string &  j);
    virtual void ShowAll() const;
    virtual void SetAll();
};

class manager:  virtual public abstr_emp
{
private:
    int inchargeof;        // number of abstr_emps managed
protected:
    int InChargeOf() const { return inchargeof; } // output
    int & InChargeOf(){ return inchargeof; }      // input
public:
    manager();
    manager(const std::string & fn, const std::string & ln,
            const std::string & j, int ico = 0);
    manager(const abstr_emp & e, int ico);
    manager(const manager & m);
    virtual void ShowAll() const;
    virtual void SetAll();
};

class fink: virtual public abstr_emp
```

```
{
private:
    std::string reportsto;        // to whom fink reports
protected:
    const std::string ReportsTo() const { return reportsto; }
    std::string & ReportsTo(){ return reportsto; }
public:
    fink();
    fink(const std::string & fn, const std::string & ln,
         const std::string & j, const std::string & rpo);
    fink(const abstr_emp & e, const std::string & rpo);
    fink(const fink & e);
    virtual void ShowAll() const;
    virtual void SetAll();
};

class highfink: public manager, public fink // management fink
{
public:
    highfink();
    highfink(const std::string & fn, const std::string & ln,
             const std::string & j, const std::string & rpo,
             int ico);
    highfink(const abstr_emp & e, const std::string & rpo, int ico);
    highfink(const fink & f, int ico);
    highfink(const manager & m, const std::string & rpo);
    highfink(const highfink & h);
    virtual void ShowAll() const;
    virtual void SetAll();
};
```

注意，该类层次结构使用了带虚基类的 MI，所以要牢记这种情况下用于构造函数初始化列表的特殊规则。还需要注意的是，有些方法被声明为保护的。这可以简化一些 highfink 方法的代码（例如，如果 highfink::ShowAll( )只是调用 fink::ShowAll( )和 manager::ShwAll( )，则它将调用 abstr_emp::ShowAll( )两次）。请提供类方法的实现，并在一个程序中对这些类进行测试。下面是一个小型测试程序：

```
// pe14-5.cpp
// useemp1.cpp -- using the abstr_emp classes

#include <iostream>
using namespace std;
#include "emp.h"

int main(void)
{
    employee em("Trip", "Harris", "Thumper");
    cout << em << endl;
    em.ShowAll();
    manager ma("Amorphia", "Spindragon", "Nuancer", 5);
    cout << ma << endl;
    ma.ShowAll();


    fink fi("Matt", "Oggs", "Oiler", "Juno Barr");
    cout << fi << endl;
```

```
    fi.ShowAll();
    highfink hf(ma, "Curly Kew");  // recruitment?
    hf.ShowAll();
    cout << "Press a key for next phase:\n";
    cin.get();
    highfink hf2;
    hf2.SetAll();

    cout << "Using an abstr_emp * pointer:\n";
    abstr_emp  * tri[4] = {&em, &fi, &hf, &hf2};
    for (int i = 0; i < 4; i++)
        tri[i]->ShowAll();

    return 0;
}
```

为什么没有定义赋值运算符？

为什么要将 ShowAll( )和 SetAll( )定义为虚的？

为什么要将 abstr_emp 定义为虚基类？

为什么 highfink 类没有数据部分？

为什么只需要一个 operator<<( )版本？

如果使用下面的代码替换程序的结尾部分，将会发生什么情况？

```
abstr_emp  tri[4] = {em, fi, hf, hf2};
for (int i = 0; i < 4; i++)
      tri[i].ShowAll();
```