第10章　引用和拷贝构造函数

引用是C++的一个特征，它就像能自动被编译器逆向引用的常量型指针一样。

虽然Pascal语言中也有引用，但C++中引用的思想来自于Algol语言。引用是支持C++运算符重载语法的基础（见 第11章），也为函数参数传入和传出的控制提供了便利。

本章首先看一下C和C++的指针的差异，然后介绍引用。但本章的大部分内容将研究令人迷糊的C++新的编程问题：拷贝构造函数 (copy-constructor)。它是特殊的构造函数，需要用引用(&)来实现从现有的相同类型的对象产生新的对象。编译器用拷贝构造函数通过传值的方式来传递和返回对象。

本章最后将阐述有点难以理解的C++的指向成员的指针(pointer-to-member)的概念。

10.1　C++中的指针

C和C++指针的最重要的区别，在于C++是一种类型要求更强的语言。就void*而言，这一点表现得更加突出。C不允许随便地把一个类型的指针指派给另一个类型，但允许通过 void*来实现。例如：

```
bird*  b;
rock*  r;
void*  v;
v = r;
b = v;
```

C++不允许这样做，其编译器将会给出一个出错信息。如果真的想这样做，必须显式地使用映射，通知编译器和读者（见18章C++改进的映射语法）。

10.2　C++中的引用

引用(&)像一个自动能被编译器逆向引用的常量型指针。它通常用于函数的参数表中和函数的返回值，但也可以独立使用。例如：

```
int  x;
int  & r = x;
```

当创建了一个引用时，引用必须被初始化指向一个存在的对象。但也可以这样写：

```
int  & q = 12;
```

这里，编译器分派了一个存储单元，它的值被初始化为 12，这样这个引用就和这个存储单元联系上了。要点是任何引用必须和存储单元联系。但访问引用时，就是在访问那个存储单元。因而，如果这样写：

```
int x=0;
int & a = x;
a++;
```

增加a事实上是增加x。考虑一个引用的最简单的方法是把它当作一个奇特的指针。这个指针的一个优点是不必怀疑它是否被初始化了（编译器强迫它初始化），也不必知道怎样对它逆

向引用（这由编译器做）。

使用引用时有一定的规则：

1) 当引用被创建时，它必须被初始化。（指针则可以在任何时候被初始化。）

2) 一旦一个引用被初始化为指向一个对象，它就不能被改变为对另一个对象的引用。（指针则可以在任何时候指向另一个对象。）

3) 不可能有NULL引用。必须确保引用是和一块合法的存储单元关连。

10.2.1 函数中的引用

最经常看见引用的地方是在函数参数和返回值中。当引用被用作函数参数时，函数内任何对引用的更改将对函数外的参数改变。当然，可以通过传递一个指针来做相同的事情，但引用具有更清晰的语法。（如果愿意的话，可以把引用看作一个使语法更加便利的工具。）

如果从函数中返回一个引用，必须像从函数中返回一个指针一样对待。当函数返回时，无论引用关连的是什么都不应该离开，否则，将不知道指向哪一个内存区域。

这儿有一个例子：

```
//: REFRNCE.CPP -- Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe; x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe; outside this scope
}

int& h() {
  int q;
//!  return q;  // Error
  static int x;
  return x; // Safe; x lives outside scope
}

main() {
  int A = 0;
  f(&A); // Ugly (but explicit)
  g(A);  // Clean (but hidden)
}
```

对函数f( )的调用缺乏使用引用的方便性和清晰性，但很清楚这是传递一个地址。在函数 g ( )的调用中，地址通过引用被传递，但表面上看不出来。

1. 常量引用

仅当在REFRNCE.CPP例程中的参数是非常量对象时，这个引用参数才能工作。如果是常量对象，函数g ( )将不接受这个参数，这样做是一件好事，因为这个函数将改变外部参数。如

果我们知道这函数不妨碍对象的不变性的话，让这个参数是一个常量引用将允许这个函数在任何情况下使用。这意味着，对于内部类型，这个函数不会改变参数，而对于用户定义的类型，该函数只能调用常量成员函数，而且不应当改变任何公共的数据成员。

在函数参数中使用常量引用特别重要。这是因为我们的函数也许会接受临时的对象，这个临时对象是由另一个函数的返回值创立或由函数使用者显式地创立的。临时对象总是不变的，因此如果不使用常量引用，参数将不会被编译器接受。看下面一个非常简单的例子：

```
//: PASCONST.CPP -- Passing references as const

void f(int&) {}
void g(const int&) {}

main() {
//!  f(1); // Error
  g(1);
}
```

调用f(1)会产生一个编译错误，这是因为编译器必须首先建立一个引用。即编译器为一个int类型分派存储单元，同时将其初始化为1并为其产生一个地址和引用捆绑在一起。存储的内容必须是常量，因为改变它将使之变得没有意义。对于所有的临时对象，必须同样假设它们是不可存取的。当改变这种数据的时候，编译器会指出错误，这是非常有用的提示，因为这个改变会导致信息丢失。

2. 指针引用

在C语言中，如果想改变指针本身而不是它所指向的内容，函数声明可能像这样：

```
void f (int**);
```

传递它时，必须取得指针的地址，像下面的例子：

```
int I = 47;
int* ip = &I;
f (&ip);
```

对于C++中的引用，语法清晰多了。函数参数变成指针的引用，用不着取得指针的地址。通过运行下面的程序，将会看到指针本身增加了，而不是它指向的内容增加了。

```
//: REFPTR.CPP -- Reference to pointer
#include <iostream.h>

void increment(int*& i) { i++; }

main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
}
```

10.2.2 参数传递准则

当给函数传递参数时，人们习惯上应该是通过常量引用来传递。虽然最初看起来似乎仅出

于对效率的考虑（通常在设计和装配程序时并不考虑效率），但像本章以后部分介绍的，这里将会存在很多的危险。拷贝构造函数需要通过值来传递对象，但这并不总是可行的。

这种简单习惯可以大大提高效率：传值方式需要调用构造函数和析构函数，然而如果不想改变参数，则可通过常量引用传递，它仅需要将地址压栈。

事实上，只有一种情况不适合用传递地址方式，这就是当传值是唯一安全的途径，否则将会破坏对象时（而不是修改外部对象，这不是调用者通常期望的）。这是下一节的主题。

10.3 拷贝构造函数

介绍了C++中的引用的基本概念后，我们将讲述一个更令人混淆的概念：拷贝构造函数，它常被称为X(X&)（“X引用的X”）。在函数调用时，这个构造函数是控制通过传值方式传递和返回用户定义类型的根本所在。

10.3.1 传值方式传递和返回

为了理解拷贝构造函数的需要，我们来看一下C语言在调用函数时处理通过传值方式传递和返回变量的方法。

```
int f (int x, char c);
int g = f (a,b);
```

编译器如何知道怎样传递和返回这些变量？其实它天生就知道！因为它必须处理的类型范围是如此之小（char，int，float，double和它们的变量），这些信息都被内置在编译器中。

如果能了解编译器怎样产生汇编代码和怎样确定调用函数 f ( )产生语句的，我们就能得到下面的等价物：

```
push b
push a
call f ( )
add sp,4
mov g, register a
```

这个代码已被认真整理过，使之具有普遍意义——b和a的表达式根据变量是全局变量（在这种情况下它们是_b和_a）或局部变量（编译器将在堆栈指针上对其索引）将有差异。 g表达式也是这样。对f( )调用的形式取决于我们的name-mangling方案，“寄存器 a”取决于CPU寄存器在我们的汇编程序中是如何命名的。但不管代码如何，逻辑是相同的。

在C和C++中，参数是从右向左进栈，然后调用函数，调用代码负责清理栈中的参数（这一点说明了add sp,4的作用）。但是要注意，通过传值方式传递参数时，编译器简单地将参数拷贝压栈——编译器知道拷贝有多大，并知道为压栈的参数产生正确的拷贝。

f( )的返回值放在寄存器中。编译器同样知道返回值的类型，因为这个类型是内置于语言中的，于是编译器可以通过把返回值放在寄存器中返回它。拷贝这个值的比特位等同于拷贝对象。

1. 传递和返回大对象

现在来考虑用户定义的类型。如果创建了一个类，我们希望传递该类的一个对象，编译器怎么知道做什么？这是编译器的作者所不知的非内部数据类型，是别人创建的类型。

为了研究这个问题，我们首先从一个简单的结构开始，这个结构太大以至于不能在寄存器中返回。

```
//: PASSTRUC.CPP -- Passing a big structure

struct big {
  char buf[100];
  int i;
  long d;
} B, B2;

big bigfun(big b) {
  b.i = 100; // Do something to the argument
  return b;
}

main() {
  B2 = bigfun(B);
}
```

在这里列出汇编代码输出有点复杂，因为大多数编译器使用“ helper ”函数而不是设置所有功能性内置。在main()函数中，正如我们猜测的，首先调用函数 bigfun()，整个B的内容被压栈。（我们可能发现有些编译器把B的地址和大小装入寄存器，然后调用helper函数把它压栈。）

在先前的例子中，调用函数之前要把参数压栈。然而，在 PASSTRUC.CPP中，将看到附加的动作：在函数调用之前， B2的地址压栈，虽然它明显不是一个参数。为了理解这里发生的事，必须了解当编译器调用函数时对编译器的约束。

2. 函数调用栈的框架

当编译器为函数调用产生代码时，它首先把所有的参数压栈，然后调用函数。在函数内部，产生代码，向下移动栈指针为函数局部变量提供存储单元。（在这里“ 下 ”是相对的，在压栈时，机器栈指针可能增加也可能减小。）在汇编语言CALL中，CPU把程序代码中的函数调用指令的地址压栈，所以汇编语言 RETURN可以使用这个地址返回到调用点。当然，这个地址是非常神圣的，因为没有它程序将迷失方向。这儿提供一个在 CALL后栈框架的样子，此时在函数中已为局部变量分配了存储单元。

函数参数
返回地址
局部变量

函数的其他部分产生的代码完全按照这个方法安排内存，因此它可以谨慎地从函数参数和局部变量中存取而不触及返回地址。我称函数调用过程中被函数使用的这块内存为函数框架(function frame)。另外，试图从栈中得到返回值是合乎道理的。因为编译器简单地把返回值压栈，函数可以返回一个偏移值，它告诉返回值的开始在栈中所处的位置。

3. 重入

因为在C和C++的函数支持中断，所以这将出现语言重入的难题。同时，它们也支持函数递归调用。这就意味着在程序执行的任何时候，中断都可以发生而不打乱程序。当然，编写中断服务程序（ISR）的作者负责存储和还原他所使用的所有的寄存器，但如果 ISR需要使用深

入堆栈的内存时，就要小心了。（可以把ISR看成没有参数和返回值是void的普通函数，它存储和还原CPU的状态。有些硬件事件触发一个ISR函数的调用，而不是在程序中显式地调用。）

现在来想象一下，如果调用函数试图从普通函数中返回堆栈中的值将会发生什么。因为不能触及堆栈返回地址以上任何部分，所以函数必须在返回地址下将值压栈。但当汇编语言RETURN执行时，堆栈指针必须指向返回地址（或正好位于它下面，这取决于机器。），所以恰好在RETURN语句之前，函数必须将堆栈指针向上移动，以便清除所有局部变量。如果我们试图从堆栈中的返回地址下返回数值，因为中断可能此时发生，此时是我们最易被攻击的时候。这个时候ISR将向下移动堆栈指针，保存返回地址和局部变量，这样就会覆盖掉我们的返回值。

为了解决这个问题，在调用函数之前，调用者应负责在堆栈中为返回值分配额外的存储单元。然而，C不是按照这种方法设计的，C++也一样。正如我们不久将看到的，C++编译器使用更有效的方案。

我们的下一个想法可能是在全局数据区域返回数值，但这不可行。重入意味着任何函数可以中断任何其他的函数，包括与之相同的函数。因此，如果把返回值放在全局区域，我们可能又返回到相同的函数中，这将重写返回值。对于递归也是同样道理。

唯一安全的返回场所是寄存器，问题是当寄存器没有足够大用于存放返回值时该怎么做。答案是把返回值的地址像一个函数参数一样压栈，让函数直接把返回值信息拷贝到目的地。这样做不仅解决了问题，而且效率更高。这也是在PASSTRUC.CCP中main( )中bigfun( ) 调用之前将B2的地址压栈的原因。如果看了bigfun( )的汇编输出，可以看到它存在这个隐藏的参数并在函数内完成向目的地的拷贝。

4. 位拷贝（bitcopy）与初始化

迄今为止，一切都很顺利。对于传递和返回大的简单结构有了可使用的方法。但注意我们所用的方法是从一个地方向另一个地方拷贝比特位，这对于 C着眼于变量的原始方法当然进行得很好。但在C++中，对象比一组比特位要丰富得多，因为对象具有含义。这个含义也许不能由它具有的位拷贝来很好地反映。

下面来考虑一个简单的例子：一个类在任何时候知道它存在多少个对象。从第 9章，我们了解到可以通过包含一个静态(static)数据成员的方法来做到这点。

```
//: HOWMANY.CPP -- Class counts its objects
#include <fstream.h>
ofstream out("howmany.out");

class howmany {
  static int object_count;
public:
  howmany() {
    object_count++;
  }
  static void print(const char* msg = 0) {
    if(msg) out << msg << ": ";
    out << "object_count = "
        << object_count << endl;
  }
  ~howmany() {
    object_count--;
```

```
    print("~howmany()");
  }
};

int howmany::object_count = 0;

// Pass and return BY VALUE:
howmany f(howmany x) {
  x.print("x argument inside f()");
  return x;
}

main() {
  howmany h;
  howmany::print("after construction of h");
  howmany h2 = f(h);
  howmany::print("after call to f()");
}
```

howmany类包括一个静态int类型变量和一个用以报告这个变量的静态成员函数print( )，这个函数有一个可选择的消息参数。每当一个对象产生时，构造函数增加记数，而对象销毁时，析构函数减小记数。

然而，输出并不是我们所期望的那样：

```
after construction of h: object_count = 1
x argument inside f(): object_count = 1
~howmany(): object_count = 0
after call to f(): object_count = 0
~howmany(): object_count = -1
~howmany(): object_count = -2
```

在h生成以后，对象数是1，这是对的。我们希望在f( )调用后对象数是2，因为h2也在范围内。然而，对象数是0，这意味着发生了严重的错误。这从结尾两个析构函数执行后使得对象数变为负数的事实得到确认，有些事根本就不应该发生。

让我们来看一下函数f( )通过传值方式传入参数那一处。原来的对象h存在于函数框架之外，同时在函数体内又增加了一个对象，这个对象是传值方式传入的对象的拷贝。然而，参数的传递是使用C的原始的位拷贝的概念，但C++ howmany类需要真正的初始化来维护它的完整性。所以，缺省的位拷贝不能达到预期的效果。

当局部对象出了调用的函数f( )范围时，析构函数就被调用，析构函数使object_count减小。所以，在函数外面，object_count等于0。h2对象的创建也是用位拷贝产生的，所以，构造函数在这里也没有调用。当对象h和h2出了它们的作用范围时，它们的析构函数又使object_count值变为负值。

10.3.2 拷贝构造函数

上述问题的出现是因为编译器对如何从现有的对象产生新的对象作了假定。当通过传值的

方式传递一个对象时，就创立了一个新对象，函数体内的对象是由函数体外的原来存在的对象传递的。从函数返回对象也是同样的道理。在表达式中：

```
howmany h2 = f(h);
```

先前未创立的对象h2是由函数f( )的返回值创建的，所以又从一个现有的对象中创建了一个新对象。

编译器假定我们想使用位拷贝（bitcopy）来创建对象。在许多情况下，这是可行的。但在howmany类中就行不通，因为初始化不仅仅是简单的拷贝。如果类中含有指针又将出现问题：它们指向什么内容，是否拷贝它们或它们是否与一些新的内存块相连？

幸运的是，我们可以介入这个过程，并可以防止编译器进行位拷贝 (bitcopy)。每当编译器需要从现有的对象创建新对象时，我们可以通过定义我们自己的函数做这些事。因为我们是在创建新对象，所以，这个函数应该是构造函数，并且传递给这个函数的单一参数必须是我们创立的对象的源对象。但是这个对象不能传入构造函数，因为我们试图定义处理传值方式的函数按句法构造传递一个指针是没有意义的，毕竟我们正在从现有的对象创建新对象。这里，引用就起作用了，可以使用源对象的引用。这个函数被称为拷贝构造函数，它经常被提及为 X(X&)（它是被称为X的类的外在表现）。

如果设计了拷贝构造函数，当从现有的对象创建新对象时，编译器将不使用位拷贝(bitcopy)。编译器总是调用我们的拷贝构造函数。所以，如果我们没有设计拷贝函数，编译器将做一些判断，但我们完全可以接管这个过程的控制。

现在我们可以关注HOWMANY.CPP中的问题了：

```
//: HOWMANY2.CPP -- The copy-constructor
#include <fstream.h>
#include <string.h>
ofstream out("howmany2.out");

class howmany2 {
  enum { bufsize = 30 };
  char id[bufsize]; // Object identifier
  static int object_count;
public:
  howmany2(const char* ID = 0) {
    if(ID) strncpy(id, ID, bufsize);
    else *id = 0;
    ++object_count;
    print("howmany2()");
  }
  // The copy-constructor:
  howmany2(const howmany2& h) {
    strncpy(id, h.id, bufsize);
    strncat(id, " copy", bufsize - strlen(id));
    ++object_count;
    print("howmany2(howmany2&)");
  }
  // Can't be static (printing id):
```

```
  void print(const char* msg = 0) const {
    if(msg) out << msg << endl;
    out << '\t' << id << ": "
        << "object_count = "
        << object_count << endl;
  }
  ~howmany2() {
    --object_count;
    print("~howmany2()");
  }
};

int howmany2::object_count = 0;

// Pass and return BY VALUE:
howmany2 f(howmany2 x) {
  x.print("x argument inside f()");
  out << "returning from f()" << endl;
  return x;
}

main() {
  howmany2 h("h");
  out << "entering f()" << endl;
  howmany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "call f(), no return value" << endl;
  f(h);
  out << "after call to f()" << endl;
}
```

这儿有一些新的手法，要很好地理解。首先，字符缓冲器 id起着对象识别作用，所以可以判断被打印的信息是哪一个对象的。在构造函数内，可以设置一个标识符（通常是对象的名字）。使用标准C库函数strncpy( )把标识符拷贝给id。strncpy( )只拷贝一定数目的字符，这是为防止超出缓冲器的限度。

其次是拷贝构造函数howmany2(howmany2&)。拷贝构造函数可以仅从现有的对象创立新对象，所以，现有对象的名字被拷贝给id，id后面跟着单词“ copy ”，这样我们就能了解它是从哪里拷贝来的。注意，使用标准C库函数strncat( )拷贝字符给id也得防止超过缓冲器的限度。

在拷贝构造函数内部，对象数目会像普通构造函数一样的增加。这意味着当参数传递和返回时，我们能得到准确的对象数目。

print( )函数已经被修改，用于打印消息、对象标识符和对象数目。现在 print( )函数必须存取具体对象的id数据，所以不再是static成员函数。

在main( )函数内部，可以看到又增加了一次函数 f( ) 的调用。但这次使用了普通的C语言调用方式，忽略了函数的返回值。既然现在知道了值是如何返回的（即在函数体内，代码处理返回过程并把结果放在目的地，目的地地址作为一个隐藏的参数传递。），我们可能想知道返回

值被忽略将会发生什么事情。程序的输出将对此作出解释。

在显示输出之前，这儿提供了一个小程序，这个小程序使用 iostreams为文件加入行号：

```
//: LINENUM.CPP -- Add line numbers
#include <fstream.h>
#include <strstrea.h>
#include <stdlib.h>
#include "..\allege.h"

main(int argc, char* argv[]) {
  if(argc < 2) {
    cerr << "usage: linenum file\n"
         << "adds line numbers to file"
         << endl;
    exit(1);
  }
  strstream text;
  {
    ifstream in(argv[1]);
    allegefile(in);
    text << in.rdbuf(); // Read in whole file
  } // Close file
  ofstream out(argv[1]); // Overwrite file
  const bsz = 100;
  char buf[bsz];
  int line = 0;
  while(text.getline(buf, bsz)) {
    out.setf(ios::right, ios::adjustfield);
    out.width(2);
    out << ++line << ") " << buf << endl;
  }
}
```

整个文件被读入strstream（我们可以从中写和读），ifstream在其范围内被关闭。然后为相同的文件创建一个ofstream, ofstream重写这个文件。getline( )从strstream中一次取一行，加上行号后写回文件。

行号以右对齐方式按 2个字段宽打印，所以输出仍然按原来的方式排列。可以改变程序，在程序中加入可选择的第 2个命令行参数。这个命令行参数可以让用户选择字段宽，或可以做得更聪明些，通过计算文件行数自动决定字段宽度。

当LINENUM.CPP被应用于HOWMANY2.OUT时，结果如下：

```
1) howmany2()
2)    h: object_count = 1
3) entering f()
4) howmany2(howmany2&)
5)    h copy: object_count = 2
6) x argument inside f()
```

```
 7)    h copy: object_count = 2
 8) returning from f()
 9) howmany2(howmany2&)
10)    h copy copy: object_count = 3
11) ~howmany2()
12)    h copy: object_count = 2
13) h2 after call to f()
14)    h copy copy: object_count = 2
15) call f(), no return value
16) howmany2(howmany2&)
17)    h copy: object_count = 3
18) x argument inside f()
19)    h copy: object_count = 3
20) returning from f()
21) howmany2(howmany2&)
22)    h copy copy: object_count = 4
23) ~howmany2()
24)    h copy: object_count = 3
25) ~howmany2()
26)    h copy copy: object_count = 2
27) after call to f()
28) ~howmany2()
29)    h copy copy: object_count = 1
30) ~howmany2()
31)    h: object_count = 0
```

正如我们所希望的，第一件发生的事是为h调用普通的构造函数，对象数增加为1。但在进入函数f( )时，拷贝构造函数被编译器调用完成传值过程。在f( )内创建了一个新对象，它是h的拷贝（因此被称为“ h拷贝 ”），所以对象数变成2，这是拷贝构造函数作用的结果。

第8行显示了从f( )返回的开始情况。但在局部变量“ h拷贝 ”销毁以前（在函数结尾这个局部变量便出了范围），它必须被拷入返回值，也就是h2。先前未创建的对象（h2）是从现有的对象（在函数f( )内的局部变量）创建的，所以第9行拷贝构造函数当然又被使用。现在，对于h2的标识符，名字变成了“ h拷贝的拷贝 ”。因为它是从拷贝拷过来的，这个拷贝是函数 f( ) 内部对象。在对象返回之后，函数结束之前，对象数暂时变为 3，但此后内部对象“ h拷贝 ”被销毁。在13行完成对f( )调用后，仅有2个对象h和h2 。这时我们可以看到h2最终是“ h拷贝的拷贝 ”。

• 临时对象

15行开始调用f(h)，这次调用忽略了返回值。在16行可以看到恰好在参数传入之前，拷贝构造函数被调用。和前面一样， 21行显示了为返回值而调用拷贝构造函数。但是，拷贝构造函数必须有一个作为它的目的地（this指针）的工作地址。对象返回到哪里？

每当编译器需要正确地计算一个表达式时，编译器可以创建一个临时对象。在这种情况下，编译器创建一个我们甚至看不见的对象作为函数 f( )忽略了的返回值的目的地地址。这个临时对象的生存期应尽可能地短，这样，空间就不会被这些等待被销毁且占珍贵资源的临时对象搞乱。在一些情况下，临时对象可能立即传递给另外的函数。但在现在这种情况下，在函数调用

之后，不需要临时对象，所以一旦函数调用以对内部对象调用析构函数（23和24行）的方式结束，临时对象就被销毁（25和26行）。

在28-31行，对象h2被销毁了，接着对象h被销毁。对象记数非常正确地回到了0。

10.3.3 缺省拷贝构造函数

因为拷贝构造函数实现传值方式的参数传递和返回，所以在这种简单结构情况下，编译器将有效地创建一个缺省拷贝构造函数，这非常重要。在C中也是这样。然而，直到目前所看到的一切都是缺省的原始行为：位拷贝（bitcopy）。

当包括更复杂的类型时，如果没有创建拷贝构造函数，C++编译器也将自动地为我们创建拷贝构造函数。这是因为在这里用位拷贝是没有意义的，它并不能达到我们的目的。

这儿有一个例子显示编译器采取的更具智能的方法。设想我们创建了一个包括几个现有类的对象的新类。这个创建类的方法被称为组合（composition），它是从现有类创建新类的方法之一。现在，假设我们用这个方法快速创建一个新类来解决某个问题。因为我们还不知道拷贝构造函数，所以没有创建它。下面的例子演示了当编译器为我们的新类创建缺省拷贝构造函数时编译器干了那些事。

```
//: AUTOCC.CPP -- Automatic copy-constructor
#include <iostream.h>
#include <string.h>

class withCC { // With copy-constructor
public:
  // Explicit default constructor required:
  withCC() {}
  withCC(const withCC&) {
    cout << "withCC(withCC&)" << endl;
  }
};

class woCC { // Without copy-constructor
  enum { bsz = 30 };
  char buf[bsz];
public:
  woCC(const char* msg = 0 ) {
    memset(buf, 0, bsz);
    if(msg) strncpy(buf, msg, bsz);
  }
  void print(const char* msg = 0) const {
    if(msg) cout << msg << ": ";
    cout << buf << endl;
  }
};

class composite {
  withCC WITHCC; // Embedded objects
```

```
  woCC WOCC;
public:
  composite() : WOCC("composite()") {}
  void print(const char* msg = 0) {
    WOCC.print(msg);
  }
};

main() {
  composite c;
  c.print("contents of c");
  cout << "calling composite copy-constructor"
       << endl;
  composite c2 = c;  // Calls copy-constructor
  c2.print("contents of c2");
}
```

类withCC有一个拷贝构造函数，这个函数只是简单地宣布它被调用。在类 composite中，使用缺省的构造函数创建一个 withCC类的对象。如果在类 withCC中根本没有构造函数，编译器将自动地创建一个缺省的构造函数。不过在这种情况下，这个构造函数什么也不做。然而，如果我们加了一个拷贝构造函数，我们就告诉了编译器我们将自己处理构造函数的创建，编译器将不再为我们创建缺省的构造函数。并且除非我们显式地创建一个缺省的构造函数，就如同为类withCC所做的那样，否则，编译器会指示出错。

类woCC没有拷贝构造函数，但它的构造函数将在内部缓冲器存储一个信息，这个信息可以使用print( )函数打印出来。这个构造函数在类composite构造函数的初始化表达式表（初始化表达式表已在第7章简单地介绍过了，并将在第 13章中全面介绍）中被显式地调用。这样做的原因在以后将会明白。

类composite既含有 withCC类的成员对象又含有 woCC类的成员对象（注意内嵌的对象WOCC在构造函数初始化表达式表中被初始化）。类composite没有显式地定义拷贝构造函数。然而，在main( )函数中，按下面的定义使用拷贝构造函数创建了一个对象。

```
composite c2 = c ;
```

类composite的拷贝构造函数由编译器自动创建，程序的输出显示了它是如何被创建的。

为了对使用组合（和继承的方法，将在第 13章介绍）的类创建拷贝构造函数，编译器递归地为所有的成员对象和基本类调用拷贝构造函数。如果成员对象也含有别的对象，那么后者的拷贝构造函数也将被调用。所以，在这里，编译器也为类 withCC调用拷贝构造函数。程序的输出显示了这个构造函数被调用。因为 woCC没有拷贝构造函数，编译器为它创建一个，它是缺省的位拷贝（bitcopy）的行为，编译器在类 composite的拷贝构造函数内部调用这个缺省的拷贝构造函数，于是在main中调用的composite::print( )显示的c2.WOCC的内容与c.WOCC内容将是相同的。编译器获得一个拷贝构造函数的过程被称为 memberwise initialization。

最好的方法是创建自己的拷贝构造函数而不让编译器创建。这样就能保证程序在我们的控制之下。

10.3.4 拷贝构造函数方法的选择

现在，我们可能已头晕了。我们可能想，怎样才能不必了解拷贝构造函数就能写一个具有

一定功能的类。但是我们别忘了：仅当准备用传值的方式传递类对象时，才需要拷贝构造函数。如果不需要这么做，就不要拷贝构造函数。

1. 防止传值方式传递

我们也许会说：“如果我自己不写拷贝构造函数，编译器将为我创建。所以，我怎么能保证一个对象永远不会被通过传值方式传递呢？”

有一个简单的技术防止通过传值方式传递：声明一个私有（private）拷贝构造函数。我们甚至不必去定义它，除非我们的成员函数或友元（friend）函数需要执行传值方式的传递。如果用户试图用传值方式传递或返回对象，编译器将会发出一个出错信息。这是因为拷贝构造函数是私有的。因为我们已显式地声明我们接管了这项工作，所以编译器不再创建缺省的拷贝构造函数。

这儿提供了一个例子：

```
//: STOPCC.CPP -- Preventing copy-construction

class noCC {
  int i;
  noCC(const noCC&); // No definition
public:
  noCC(int I = 0) : i(I) {}
};

void f(noCC);

main() {
  noCC n;
//! f(n); // Error: copy-constructor called
//! noCC n2 = n; // Error: c-c called
//! noCC n3(n); // Error: c-c called
}
```

注意使用更普通的形式

```
noCC(const noCC&);
```

这里使用了const。

2. 改变外部对象的函数

一般来讲，引用语法比指针语法更好，然而对于读者来说，它使得意思变得模糊。例如，在iostreams库函数中，一个重载版函数get( )是用一个char&作为参数，函数通过插入get( )的结果而改变它的参数。然而，当我们阅读使用这个函数的代码时，我们不会立即明白外面的对象正被改变：

```
char c;
cin.get(c);
```

事实上函数调用看起来像一个传值传递，暗示着外部对象没有被改变。

正因为如此，当传递一个可被修改的参数时，从代码维护的观点看，使用指针可能安全些。所以除非我们打算通过地址修改外部对象（这个地址通过非const指针传递），要不然都用const引用传递地址。因为这样，读者更容易读懂我们的代码。

10.4 指向成员的指针（简称成员指针）

指针是指向一些内存地址的变量，既可以是数据的地址也可以是函数的地址。所以，可以在运行时改变指针指向的内容。除了C++的成员指针(pointer-to-member)选择的内容是在类之外，C++的成员指针遵从同样的原则。困难的是所有的指针需要一个地址，但在类内部没有地址；选择一个类的成员意味着在类中偏移。只有把这个偏移和具体对象的开始地址结合，才能得到实际地址。成员指针的语法要求选择一个对象的同时逆向引用成员指针。

为了理解这个语法，先来考虑一个简单的结构：

```
struct simple { int a ; };
```

如果有一个这个结构的指针sp和对象so，可以通过下面方法选择成员：

```
sp->a ;
so.a ;
```

现在，假设有一个普通的指向integer的指针ip。为了取得ip指向的内容，用一个*号逆向引用指针的引用。

```
*ip = 4 ;
```

最后，考虑如果有一个指向一个类对象成员，甚至假设它代表对象内一定的偏移，将会发生什么？为了取得指针指向的内容，必须用 *号逆向引用。但是，它只是一个对象内的偏移，所以还必须要指定那个对象。因此， *号要和逆向引用的对象结合。像下面使用 simple类的例子：

```
sp->*pm = 47 ;
so.*pm = 47 ;
```

所以，对于指向一个对象的指针新的语法变为->*，对于一个对象或引用则为.*。现在，让我们看看定义pm的语法是什么？其实它像任何一个指针，必须说出它指向什么类型。并且，在定义中也要使用一个‘ * ’号。唯一的区别只是必须说出这个成员指针使用什么类的对象。当然，这是用类名和全局操作符实现的。因此，可表示如下：

```
int simple::*pm ;
```

当我们定义它时（或任何别的时间），也可以初始化成员指针：

```
int simple::*pm = &simple::a ;
```

因为引用到一个类而非那个类的对象，所以没有 simple::a的确切“地址”。因而，&simple::a仅可作为成员指针的语法表示。

函数

这里提供一个为成员函数产生成员指针（pointer-to-member）的例子。指向函数的指针定义像下面的形式：

```
int (*fp)(float) ;
```

(*fp)的圆括号用来迫使编译器正确判断定义。没有圆括号，这个表达式就是一个返回 int*值的函数。

为了定义和使用一个成员函数的指针，圆括号扮演同样重要的角色。假设在一个结构内有一个函数：

```
struct simple2 { int f(float); } ;
```

通过给普通函数插入类名和全局操作符就可以定义一个指向成员函数的指针：

```
int (simple2::*fp) (float) ;
```

当创建它时或其他任何时候，可以对它初始化：

```
int (simple2::*fp) (float) = &simple2::f ;
```

和其他普通函数一样，&号是可选的；可以用不带参数表的函数标识符来表示地址：

```
fp = simple2::f ;
```

• 一个例子

在程序运行时，我们可以改变指针所指的内容。因此在运行时我们就可以通过指针选择来改变我们的行为，这就为程序设计提供了重要的灵活性。成员指针也一样，它允许在运行时选择一个成员。特别的，当我们的类只有公有 (public)成员函数（数据成员通常被认为是内部实现的一部分）时，就可以用指针在运行时选择成员函数，下面的例子正是这样：

```
//: PMEM.CPP -- Pointers to members

class widget {
public:
  void f(int);
  void g(int);
  void h(int);
  void i(int);
};

void widget::h(int) {}

main() {
  widget w;
  widget* wp = &w;
  void (widget::*pmem)(int) = &widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
}
```

当然，期望一般用户创建如此复杂的表达式不是特别合乎情理。如果用户必须直接操作成员指针，那么typedef是适合的。为了安排得当，可以使用成员指针作为内部执行机制的一部分。这儿提供一个前述的在类内使用成员指针的例子。用户所要做的是传递一个数字以选择一个函数。[1]

```
//: PMEM2.CPP -- Pointers to members
#include <iostream.h>
class widget {
  void f(int) const {cout << "widget::f()\n";}
  void g(int) const {cout << "widget::g()\n";}
  void h(int) const {cout << "widget::h()\n";}
  void i(int) const {cout << "widget::i()\n";}
```

[1] 感谢Owen Mortensen提供这个例子。

```
    enum { count = 4 };
    void (widget::*fptr[count])(int) const;
  public:
    widget() {
      fptr[0] = &widget::f; // Full spec required
      fptr[1] = &widget::g;
      fptr[2] = &widget::h;
      fptr[3] = &widget::i;
    }
    void select(int I, int J) {
      if(I < 0 || I >= count) return;
      (this->*fptr[I])(J);
    }
    int Count() { return count; }
  };

  main() {
    widget w;
    for(int i = 0; i < w.Count(); i++)
      w.select(i, 47);
  }
```

在类接口和main( )函数里，可以看到，包括函数本身在内的整个实现被隐藏了。代码甚至必须请求Count( )函数。用这个方法，类的实现者可以改变大量函数而不影响使用这个类的代码。

在构造函数中，成员指针的初始化似乎被过分地指定了。是否可以这样写：

```
fptr[1] = &g ;
```

因为名字g在成员函数中出现，是否可以自动地认为在这个类范围内呢？问题是这不符合成员函数的语法，它的语法要求每个人，尤其编译器，能够判断将要进行什么。相似地，当成员函数被逆向引用时，它看起来像这样：

```
(this->*fptr[i]) (j);
```

它仍被过分地指定了，this似乎多余。正如前面所讲的，当它被逆向引用时，语法也需要成员指针总是和一个对象绑定在一起。

10.5 小结

C++的指针和C中的指针是非常相似的，这是非常好的。否则，许多C代码在C++中将不会被正确地编译。仅在出现危险赋值的地方，编译器会产生出错信息。假设我们确实想这样赋值，编译器的出错可以用简单的（和显式的！）映射（cast）清除。

C++还从Algol和Pascal中引进引用(reference)概念，引用就像一个能自动被编译器逆向引用的常量指针一样。引用占一个地址，但我们可以把它看成一个对象。引用是操作符重载语法（下一章的主题）的重点，它也为普通函数值传递和返回对象增加了语法的便利。

拷贝构造函数采用相同类型的对象引用作为它的参数，它可以被用来从现有的类创建新类。当用传值方式传递或返回一个对象时，编译器自动调用这个拷贝构造函数。虽然，编译器将自

动地创建一个拷贝构造函数，但是，如果认为需要为我们的类创建一个拷贝构造函数，应该自己定义它以确保正确的操作。如果不想通过传值方式传递和返回对象，应该创建一个私有的(private)拷贝构造函数。

成员指针和普通指针一样具有相同的功能：可以在运行时选取特定存储单元(数据或函数)。成员指针只和类成员一起工作而不和全局数据或函数一起工作。通过使用成员指针，我们的程序设计可以在运行时灵活地改变。

10.6 练习

1. 写一个函数，这个函数用一个char&作参数并且修改该参数。在main( )函数里，打印一个char变量，使用这个变量做参数，调用我们设计的函数。然后，再次打印此变量以证明它已被改变。这样做是如何影响程序的可读性的？

2. 写一个有拷贝构造函数的类，在拷贝构造函数里用 cout自我声明。现在，写一个函数，这个函数通过传值方式传入我们新类的对象。写另一个函数，在这个函数内创建这个新类的局部对象，通过传值方式返回这个对象。调用这些函数以证明通过传值方式传递和返回对象时，拷贝构造函数确实悄悄地被调用了。

3. 努力发现如何使得我们的编译器产生汇编语言，并请为PASSTRUC.CPP产生汇编代码。跟踪和揭示我们的编译器为传递和返回大结构产生代码的方法。