第23章　名字空间、转换函数和其他高级主题

本章将介绍名字空间和其他几个高级功能，其中包括转换函数、构造函数、const和volatile成员函数、关键字asm以及连接规范，此外，还将在本章最后讨论C++中基于数组的I/O并总结C与C++的不同之处。

23.1　名字空间

本书曾在前面提到过名字空间，它们是在最近才添加到C++中的，其目的是要将标识符的名字定位在一定范围内以避免发生名字冲突。我们可以从C++编程环境中看出变量名、函数名和类名的暴增。在引入名字空间之前，所有这些名字都在争相获取全局名字空间中的位置，从而引发许多冲突。例如，如果你在程序中定义了一个称为abs( )的函数，该函数（根据它的参数表）会覆盖标准的库函数abs( )，这是因为这两个函数名都存储在全局名字空间中。当在同一个程序中使用两个或多个由第三方提供的库时，各种名字冲突会被混在一起。在这种情况下，有可能（甚至很可能）发生以下情况，即：一个库定义的名字与其他库定义的名字发生冲突。对于类名而言，这种情况尤其令人棘手。例如，如果你在程序中定义了一个称为ThreeDCircle的类并使用了一个库，而且这个库也定义了一个具有同样名字的类，这时就会引发冲突。

关键字namespace的创建就是为了解决这些问题。因为该关键字把名字的可见性定位在声明它的范围内，所以可以允许在不同的上下文中使用相同的名字，而且不会引起冲突。namespace的最明显的受益者或许是C++标准库。在引入namespace之前，整个C++库定义在全局名字空间（当然，它是惟一的名字空间）内。由于namespace的引入，C++库现在定义在自己的称为std的名字空间中，从而减少了发生名字冲突的机会。你还可以在程序中创建自己的名字空间，从而限定有可能引发冲突的名字的可见范围。如果你正在创建类库或函数库，这种方法尤其重要。

23.1.1　名字空间的基础知识

关键字namespace使你可以通过创建一个声明区域来分割全局名字空间。从本质上说，namespace定义了一个作用域，它的一般形式如下所示：

```
namespace name {
  // declarations
}
```

namespace语句内定义的一切都在该名字空间的作用域之内。

下面给出了一个使用namespace的例子，它将用于实现一个简单的倒计数器类的名字限定在相应的作用域内，在该名字空间中定义了counter类和变量upperbound与lowerbound，counter类用于实现计数器功能，而两个变量包含适用于所有计数器的上限和下限。

```
namespace CounterNameSpace {
```

```
  int upperbound;
  int lowerbound;

  class counter {
    int count;
   public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }

    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}
```

其中，upperbound，lowerbound和counter类是CounterNameSpace名字空间定义的作用域的一部分。

在一个名字空间中，所有在该名字空间中声明的标识符都可以被直接引用，它们不受任何名字空间的限制。例如，在CounterNameSpace内，函数run( )可以在语句中直接引用lowerbound。

```
if(count > lowerbound) return count--;
```

然而，由于namespace定义了一个作用域，所以在该名字空间外部必须使用作用域分辨符访问在其内部声明的对象。例如，为了在CounterNameSpace外部把10赋给upperbound，必须使用下面的语句：

```
CounterNameSpace::upperbound = 10;
```

也可以在CounterNameSpace外声明一个counter类型的对象，为此可以使用下面的语句：

```
CounterNameSpace::counter ob;
```

一般来说，为了从名字空间外部访问该名字空间的成员，应在该成员名的前面加上名字空间的名字和作用域分辨符。

下面的程序演示了CounterNameSpace的用途。

```
// Demonstrate a namespace.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
    int count;
  public:
    counter(int n) {
```

```
    if(n <= upperbound) count = n;
    else count = upperbound;
  }

  void reset(int n) {
    if(n <= upperbound) count = n;
  }

  int run() {
    if(count > lowerbound) return count--;
    else return lowerbound;
  }
 };
}

int main()
{
  CounterNameSpace::upperbound = 100;
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  CounterNameSpace::counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  ob2.reset(100);
  CounterNameSpace::lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);

  return 0;
}
```

注意，counter对象的声明和对upperbound与lowerbound的引用受CounterNameSpace的限制。然而，一旦声明了一个counter类型的对象，就不再需要对它和它的成员进行进一步限制。因此，ob1.run()可以被直接调用，因为该名字空间已被确定。

23.1.2 using

可以想像，如果你的程序频繁地引用某个名字空间的成员，那么在每次需要快速引用其中的一个成员时必须指定该名字空间和作用域分辨符是一件很乏味的事。为了解决这个问题，

C++引入了using语句。该语句有两种形式：

```
using namespace name;
using name::member;
```

在第一种形式中，name指定想要访问的名字空间的名字。在指定的名字空间中定义的所有成员都是可见的（也就是说都将成为当前名字空间的一部分），而且可以被不加限制地使用。在第二种形式中，只有一个特定的名字空间成员是可见的。例如，假设名字空间是前面讲到的CounterNameSpace，下面的using语句和赋值语句都是有效的。

```
using CounterNameSpace::lowerbound; // only lowerbound is visible
lowerbound = 10; // OK because lowerbound is visible

using namespace CounterNameSpace; // all members are visible
upperbound = 100; // OK because all members are now visible
```

下面的程序通过改写前面的计数器范例说明了using的用途。

```
// Demonstrate using.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
    int count;
  public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }
    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}

int main()
{
  // use only upperbound from CounterNameSpace
  using CounterNameSpace::upperbound;

  // now, no qualification needed to set upperbound
  upperbound = 100;

  // qualification still needed for lowerbound, etc.
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;
```

```
  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  // now, use entire CounterNameSpace
  using namespace CounterNameSpace;

  counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);
  cout << endl;

  ob2.reset(100);
  lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);

  return 0;
}
```

上面的程序说明了另一个重要的问题：不能用一个名字空间覆盖另一个名字空间。当使一个名字空间成为可见时，它只是把它的名字添加到其他在当前时刻有效的名字空间中。因此，在程序终止之前，std 和 CounterNameSpace 都被添加到全局名字空间中。

23.1.3 未命名的名字空间

由一种特殊类型的名字空间称为未命名名字空间，它使你可以在一个文件内创建惟一的标识符。未命名名字空间也称为匿名名字空间，其一般形式如下：

```
namespace {
  // declarations
}
```

未命名名字空间使你可以创建一些惟一的标识符，这些标识符只在一个文件的作用域内为程序所知。也就是说，在含有未命名名字空间的文件内，这个名字空间的成员可以被不受限制地直接使用。但是在这个文件的外部，这些标识符将不为人所知。

未命名名字空间取消了对一些static存储类修饰符的需要。本书曾在第2章讲到，有一种方法可将全局名字的作用域限制在文件范围，这种方法就是使用static。例如，看一看下面两个文件，它们是同一个程序的不同组成部分。

文件1

```
static int k;
void f1() {
k = 99; // OK
}
```

文件2

```
extern int k;
void f2() {
k = 10; // error
}
```

因为k是在文件1中定义的，所以只能在文件1中使用。而在文件2中，k被指定为extern，这意味着k的名字和类型将为程序所知，但是k本身并没有被真正定义。当这两个文件被连接起来时，如果试图在文件2中使用k，将引发一个错误，这是因为在文件2中没有定义k。在文件1中，由于在k的前面加上了static，从而将k的作用域限制在该文件内，而且不能为文件2所用。

虽然C++依然允许使用static全局声明，但是有一种更好的方法可以达到同样的效果，这种方法就是使用未命名名字空间。例如：

文件1

```
namespace {
  int k;
}
void f1() {
  k = 99; // OK
}
```

文件2

```
extern int k;
void f2() {
  k = 10; // error
}
```

其中，k同样被限制在文件1内。我们建议读者在编写新代码时使用未命名名字空间，而不要使用static。

23.1.4 一些名字空间选项

同一个名字可以有多个名字空间声明，它使得一个名字空间可以被拆分散布在几个文件中，甚至可以在同一个文件内被分割。例如：

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

namespace NS {
  int i;
}

// ...

namespace NS {
  int j;
}

int main()
{
  NS::i = NS::j = 10;

  // refer to NS specifically
  cout << NS::i * NS::j << "\n";

  // use NS namespace
  using namespace NS;

  cout << i * j;

  return 0;
}
```

上面程序的输出如下所示：

```
100
100
```

其中，NS被一分为二。然而，拆分后每个部分的内容仍然在同一个名字空间中，该名字空间就是NS。

一个名字空间必须在所有其他作用域之外被声明，这意味着不能声明被定位于一个函数的名字空间。但是有一种例外情况，即一个名字空间可以被嵌套在另一个名字空间之中。看一看下面的程序：

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

namespace NS1 {
  int i;
  namespace NS2 { // a nested namespace
    int j;
  }
}

int main()
{
  NS1::i = 19;
  // NS2::j = 10; Error, NS2 is not in view
  NS1::NS2::j = 10; // this is right

  cout << NS1::i << " "<< NS1::NS2::j << "\n";

  // use NS1
  using namespace NS1;

/* Now that NS1 is in view, NS2 can be used to
   refer to j. */
  cout << i * NS2::j;

  return 0;
}
```

上面这个程序的输出如下所示：

```
19 10
190
```

其中，名字空间NS2嵌套在NS1之内，因此当程序开始执行时，为了引用j，必须用NS1和NS2这两个名字空间对其进行限定，而只用NS2是不够的。当执行了下面的语句之后，就可以直接引用NS2了，这是因为using语句使NS1变成可见的了：

```
using namespace NS1;
```

一般来说，对大多数中小型程序而言不需要创建名字空间。然而，如果要创建可重用的代码库或是要最大程度地确保程序的可移植性，应该考虑把代码包装在一个名字空间内。

23.2 std名字空间

标准C++把整个库定义在自己的名字空间中，该名字空间称为std。这也是本书的大多数

程序都包含下列语句的原因所在：

```
using namespace std;
```

上面的语句将std名字空间引入当前名字空间，从而使你可以直接访问库中定义的函数名和类名，而且不必使用限定符std::。

当然，如果愿意，也可以利用std::显式地限定每一个名字。例如，下面的程序没有把库引入全局名字空间：

```
// Use explicit std:: qualification.
#include <iostream>

int main()
{
  int val;

  std::cout << "Enter a number: ";

  std::cin >> val;

  std::cout << "This is your number: ";
  std::cout << std::hex << val;

  return 0;
}
```

其中，cout，cin和操作算子hex通过它们的名字空间被显式地限定。也就是说，为了把数据写到标准输出上，必须指定std::cout；为了从标准输入读入数据，必须使用std::cin；hex操作算子必须被引用为std::hex。

如果你的程序只是有限地使用标准C++库，那么你或许不想把该库引入到全局名字空间中。然而，如果你的程序包含几百个对库名字的引用，那么把std包含在当前名字空间中比分别对每一个名字加以限定要容易得多。

如果只使用标准库中的几个名字，那么对每一个名字指定一个using语句或许是更可取的方法。这个方法的优点是：不用限定符std::仍然可以使用这些名字，同时又不用将整个标准库引入全局名字空间。例如：

```
// Bring only a few names into the global namespace.
#include <iostream>

// gain access to cout, cin, and hex
using std::cout;
using std::cin;
using std::hex;

int main()
{
  int val;
  cout << "Enter a number: ";

  cin >> val;
  cout << "This is your number: ";
  cout << hex << val;
  return 0;
}
```

其中，cin，cout 和 hex 可以被直接使用，但是 std 名字空间的其他内容没有改为可见的。

就像前面讲到的那样，最初的C++库是在全局名字空间中定义的。如果要转换较早的C++程序，则要么需要包含一个 using namespace std 语句，要么需要用 std::限定对库成员的每一个引用。如果用新式的头文件取代了老式的.H头文件，这个问题尤其重要。记住，老式的.H头文件把文件的内容放入全局名字空间，新式的头文件把文件内容放入 std 名字空间。

23.3　创建转换函数

在某些情况下，你或许想在包括其他类型数据的表达式中使用一个类的对象，重载操作符函数有时可以为此提供一些方法。然而，在其他情况下，需要的只是把某种类类型转换为目标类型的简单类型转换。为了处理这些情况，C++ 允许你创建定制转换函数。转换函数可以把你创建的类转换成与表达式中其他内容兼容的类型，该函数的一般形式如下：

operator *type*( ) { return *value*; }

其中，type 是要转换成的目标类型，value 是转换后的类的值。转换函数返回 type 类型的数据，而且不允许其他返回类型的限定符。另外，该函数不能包括其他参数。转换函数必须是这个类的一个成员（转换函数就是为此类定义的）。转换函数可以被继承，也可以是虚函数。

下面是一个使用转换函数的程序，其中使用的 stack 类最早在第 11 章中引入。假设你想在一个整型表达式内使用 stack 类型的对象，而且在整型表达式中使用的 stack 对象值是当前堆栈中存放的值的个数（例如，如果你正在用一些stack对象进行仿真并正在监测这些堆栈的填充速度，那么你或许想做些与此类似的事情）。要到达这个目的，有一种方法是把一个 stack 类型的对象转换为表示堆栈中的数据项个数的整型数。为此，可以使用如下所示的转换函数：

```
operator int() { return tos; }
```

下面的程序说明了该转换函数的工作原理：

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

const int SIZE=100;

// this creates the class stack
class stack {
  int stck[ SIZE];
  int tos;
public:
  stack() { tos=0; }
  void push(int i);
  int pop(void);
  operator int() { return tos; } // conversion of stack to int
};

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
```

```
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack stck;
  int i, j;

  for(i=0; i<20; i++) stck.push(i);

  j = stck; // convert to integer

  cout << j << " items on stack.\n";

  cout << SIZE - stck << " spaces open.\n";
  return 0;
}
```

上面这个程序的输出如下所示：

```
20 items on stack.
80 spaces open.
```

就像该程序说明的那样，当在整型表达式中使用一个stack对象时（例如，j = stck），将对该对象使用转换函数。在这种情况下，转换函数返回值为20。另外，当从SIZE中减去stck时，也会调用转换函数。

下面是另一个关于转换函数的例子。这个程序创建一个称为pwr( )的类，这个类可以存储和计算某数的幂并把求幂的结果存储为double型。通过给类型double提供一个转换函数并返回计算结果，可以在包括其他double值的表达式中使用类型pwr的对象。

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

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  pwr operator+(pwr o) {
    double base;
```

```
    int exp;
    base = b + o.b;
    exp = e + o.e;

    pwr temp(base, exp);
    return temp;
  }

  operator double() { return val; } // convert to double
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2);
  double a;

  a = x; // convert to double
  cout << x + 100.2; // convert x to double and add 100.2
  cout << "\n";

  pwr y(3.3, 3), z(0, 0);

  z = x + y; // no conversion
  a = z; // convert to double
  cout << a;

  return 0;
}
```

上面这个程序的输出如下所示：

```
116.2
20730.7
```

可以看到，当x用在表达式x + 100.2中时，转换函数用于产生double值。还要注意，因为在表达式x + y中只包括pwr类型的对象，所以没有用到转换操作。

我们可以根据前面的例子做出以下推断：在许多情况下，创建一个用于类的转换函数是一种有益之举。当类对象与内置类型混在一起时，转换函数可以提供更为自然的语法。确切地说，在pwr类的情况下，因为可以有效地将其转换为double值，从而使得在“普通”数学表达式中使用这个类的对象的操作既简单又容易理解。

你可以创建不同的转换函数以满足不同的需要，例如，可以定义另一个目标类型为long的转换。根据表达式的类型，可以自动确定使用哪一种转换。

23.4 const成员函数与mutable

类成员函数可以被声明为const，这使得this可以被当做一个const指针，从而使得该函数不能修改调用它的对象。另外，一个const对象不能调用一个非const成员函数。然而，一个const成员函数既可以被const对象也可以被非const对象调用。

为了把一个成员函数指定为const，可使用下面例子中的形式：

```
class X {
  int some_var;
public:
  int f1() const; // const member function
};
```

可以看到，const跟在函数声明之后。

把一个成员函数声明为const的目的是为了防止修改调用它的对象。例如，看一看下面的程序：

```
/*
    Demonstrate const member functions.
    This program won't compile.
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // error!
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

因为seti( )被声明为const，所以这个程序将不能编译。这意味着不允许它修改调用对象。因为该程序试图改变i，所以导致出错。相反，由于geti( )没有试图修改i，所以没有错误。有时，虽然不想让cons函数修改其他成员，但却想使它能够修改某个类的一个或多个成员。可以通过使用mutable达到这个目的，它覆盖const函数的成分，也就是说，mutable成员可以被const成员函数修改。例如：

```
// Demonstrate mutable.
#include <iostream>
using namespace std;

class Demo {
  mutable int i;
  int j;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // now, OK.
  }

/* The following function won't compile.
  void setj(int x) const {
    j = x; // Still Wrong!
  }
*/
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

其中，i被指定为mutable，因此可以被seti( )函数改变。然而，j不是mutable，所以不能被setj( )修改。

23.5　volatile 成员函数

类成员函数可以被声明为volatile，这将使得this被当做一个volatile指针。为了把一个成员函数指定为volatile，可使用下面例子中显示的形式：

```
class X {
public:
  void f2(int a) volatile; // volatile member function
};
```

23.6　explicit 构造函数

就像第12章介绍的那样，只要构造函数只需要一个参数，就可以利用ob(x)或ob = x 来初始化一个对象，这是因为当创建只有一个参数的构造函数时，你还隐含地创建了一个从参数类型到类类型的转换。但有时你并不想让它自动进行转换，为此，C++定义了关键字explicit。要想理解该关键字的作用，请看下面的程序。

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

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
  myclass ob = 4; // automatically converted into myclass(4)

  cout << ob.geta();

  return 0;
}
```

其中，构造函数myclass只有一个参数。我们现在看一看ob是怎样在main( )中被声明的。下面的语句以4为参数被自动转换为对myclass构造函数的调用。

```
myclass ob = 4; // automatically converted into myclass(4)
```

也就是说，计算机处理上面的语句时认为它与下面的语句相同：

```
myclass ob(4);
```

如果不想进行这种隐式的转换，则可以利用explicit防止转换的发生。限定符explicit只适用于构造函数。只有当一个初始化使用普通构造函数语法时才将构造函数说明为explicit。这种构造函数不执行自动转换。例如，把myclass构造函数声明为explicit将不提供自动转换。下面的程序将myclass( )声明为explicit。

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

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

现在，只有如下形式的构造函数才被允许。

```
myclass ob(4);
```

下面的语句是无效语句。

```
myclass ob = 4; // now in error
```

23.7 成员初始化语法

前面各章的范例代码都是在成员所在类的构造函数内部对成员变量进行初始化的。例如，下面的程序包含一个MyClass类，该类有两个整型数据成员，它们是numA和numB。这些成

员变量是在构造函数 MyClass 内部被初始化的。

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

class MyClass {
  int numA;
  int numB;
public:
  /* Initialize numA and numB inside the MyClass constructor
     using normal syntax. */
  MyClass(int x, int y) {
    numA = x;
    numB = y;
  }

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

像 MyClass( )一样，在构造函数内部把初始值赋给成员变量 numA 和 numB 是惯用的方法，同时也是对许多的类成员进行初始化的方法。然而，这种方法不能适用于所有情况，例如，如果 numA 和 numB 被声明为 const（如下所示），则不能由构造函数 MyClass 对其赋值。

```
class MyClass {
  const int numA; // const member
  const int numB; // const member
```

这是因为 const 变量必须被初始化，而且在初始化之后不能被赋值。当使用必须被初始化的引用成员和使用没有默认构造函数的类成员时，也会出现类似问题。为了解决此类问题，C++ 提供了另一种成员初始化语法，这种语法用于在创建一个类的对象时给类成员分配一个初始值。成员初始化语法与调用基类构造函数所用的语法相似，下面是该语法的一般形式：

```
constructor(arg-list) : member1(initializer),
                        member2(initializer),
                        // ...
                        memberN(initializer)
{
 // body of constructor
}
```

想要初始化的成员应在函数体之前声明，而且在构造函数名和参数表之间要用冒号分隔。

你可以把对基类构造函数的调用与成员的初始化在同一个列表中混合使用。

下面的程序重新编写了MyClass，其中，numA 和numB是利用成员初始化语法赋值的const成员。

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

class MyClass {
  const int numA; // const member
  const int numB; // const member
public:
  // Initialize numA and numB using initialization syntax.
  MyClass(int x, int y) : numA(x), numB(y) { }

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

我们可以看到程序是怎样利用如下语句对 numA 和 numB 进行初始化的：

```
MyClass(int x, int y) : numA(x), numB(y) { }
```

其中，numA利用x中传递的值初始化，numB利用y中传递的值初始化。虽然numA和numB现在是const型，但由于使用了成员初始化语法，所以当创建一个MyClass对象时可以给numA和numB分配初值。

成员初始化语法特别适用于没有默认构造函数的类的类成员。为什么呢？让我们看一看下面这个略有变异的MyClass版本，该版本试图把两个整型值存储在一个IntPair类型的对象中。由于IntPair没有默认构造函数，下面的程序将出现错误而且不能编译。

```
// This program is in error and won't compile.
#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) { }
};

class MyClass {
```

```
  IntPair nums; // Error: no default constructor for IntPair!
public:
  // This won't work!
  MyClass(int x, int y) {
    nums.a = x;
    nums.b = y;
  }

  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);
  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

上面的程序不能编译的原因是IntPair只有一个构造函数，而且该构造函数需要两个参数。然而，nums在MyClass内部被声明为不带任何变量，而且a和b的值也在MyClass的构造函数内部被设定。这个程序之所以出现错误，是因为它暗指一个默认（即无参数）构造函数可以在初始时创建一个IntPair对象，但情况并非如此。

为了解决这个问题，可以给IntPair添加一个默认构造函数。然而，它只能在访问该类的源代码之后才能奏效，而实际情况并非总是如此。一种比较好的解决方案是使用成员初始化语法，如下面这个正确的程序版本所示。

```
// This program is now correct.
#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) { }
};

class MyClass {
  IntPair nums; // now OK
public:
  // Initialize nums object using initialization syntax.
  MyClass(int x, int y) : nums(x,y) { }

  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};
```

```
int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

其中，nums是创建MyClass对象时分配的初始值，所以不需要默认构造函数。

最后要说的是：类成员的构造顺序和初始化顺序与它们在类中被声明的顺序相同，与它们的初始部分的顺序无关。

23.8 利用关键字asm

虽然C++是功能丰富的编程语言，但是仍然有少数几个不能处理的特殊情况（例如，C++中没有禁止中断的语句）。为了处理这些特殊情况，C++提供了一个“天窗”（trap door），使你可以随时略过C++编译器进入汇编代码。这个“天窗”就是asm语句。利用asm把汇编语言直接嵌入C++程序。该汇编代码无需任何修改就可以编译，而且在asm语句处成为程序代码的一部分。

关键字asm的一般形式如下所示：

```
asm ("op-code");
```

其中，op-code是将被嵌入程序中的汇编语言指令。然而，有几种编译器也允许使用下面的asm语句形式：

```
asm instruction ;
asm instruction newline
asm {
  instruction sequence
}
```

其中，instruction是任何有效的汇编语言指令。要了解asm属性的具体实现，必须查阅你所使用的编译器的文档以了解详细情况。

笔者在撰写本书时，微软的Visual C++使用__asm嵌入汇编代码。它在其他方面与asm类似。

下面是一个使用关键字asm的简单（而且相当安全）例子：

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

int main()
{
  asm int 5; // generate intertupt 5

  return 0;
}
```

这个程序在 DOS 环境下运行时将产生一个指令——INT 5，该指令调用屏幕打印函数。

警告：要全面掌握用汇编语言编程的知识，必须会使用 asm 语句。如果你对汇编语言不够精通，最好不要使用 asm，因为它有可能产生非常严重的错误。

23.9　连接说明

可以在C++中指定如何将一个函数连接到你的程序中。默认情况下，这些函数作为C++函数被连接。然而，通过使用连接说明，可以连接一个不同类型语言的函数。连接说明符的一般形式如下：

extern "*language*" *function-prototype*

其中，language 表示希望连接的语言。所有C++编译器都支持C和C++连接，有些编译器还允许使用连接到Fortran，Pascal或BASIC的连接说明符（使用时需要查阅所使用的编译器文档）。

下面的程序使得 myCfunc( )被连接为一个 C 函数。

```
#include <iostream>
using namespace std;
extern "C" void myCfunc();

int main()
{
  myCfunc();

  return 0;
}

// This will link as a C function.
void myCfunc()
{
  cout << "This links as a C function.\n";
}
```

注意：关键字 extern 是连接说明必须包含的一部分，而且，由于连接说明不能在函数内部使用，所以该说明必须是全局的。

可以在使用这种形式的连接说明时指定多个函数：

```
extern "language" {
 prototypes
}
```

23.10　基于数组的 I/O

除控制台和文件I/O外，C++中基于流的I/O系统还允许基于数组的I/O。基于数组的I/O既可以把一个字符数组用做输入设备或输出设备，也可以将其同时用做输入和输出设备。基于数组的I/O可以通过标准C++流被执行。事实上，你所了解的有关C++ I/O的一切都适用于基于数组的I/O。基于数组的I/O所独具的特点是：连接到流的设备是一个字符数组。连接到字符数组的流通常被称为char *流。为了在程序中使用基于数组的I/O，必须包括头文件<strstream>。

注意：标准C++不赞成使用本节描述的基于字符的流类。这意味着它们仍然有效，但是却不提倡在新代码中使用。我们在此对其进行简单的介绍以帮助使用较老代码的读者。

23.10.1 基于数组的类

基于数组的I/O类有istrstream，ostrstream和strstream。这些类可分别用于输入流、输出流和输入/输出流。另外，istrstream是istream的派生类，ostrstream是ostream的派生类。因此，所有基于数组的类都从ios派生而来，而且可以访问的成员函数与“普通”I/O类访问的相同。

23.10.2 创建基于数组的输出流

为了执行一个到数组的输出，必须使用ostrstream构造函数把数组连接到一个流：

```
ostrstream ostr(char *buf, streamsize size, openmode mode=ios::out);
```

其中，buf是一个指向数组的指针，该指针所指的数组用于收集写入ostr流的字符，数组的大小通过参数size传递。默认情况下，这个流以一般输出方式打开，但是你可以利用或（OR）运算把其他各种选项和它和一同使用以创建所需的模式。例如，可以通过包括ios::app把输出写到数组中已有数据的末尾。大多数情况下，mode将使用默认模式。

一旦打开了基于数组的输出流，所有定向到这个流的输出都被转移到该数组。然而，输出不能写到数组边界之外，否则将会引发错误。

下面是一个演示基于数组的输出流的简单程序。

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

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "C++ array-based I/O. ";
  outs << 1024 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ' ' << 99.789 << ends;
  cout << str; // display string on console

  return 0;
}
```

上面程序将显示以下信息：

```
C++ array-based I/O. 1024 0x64 99.789
```

记住，outs是一个流，就像任何其他的流一样，它的能力也与前面介绍的其他类型的流相同，惟一不同的是：它所连接的设备是一个字符数组。因为outs是一个流，所以像hex和ends这样的操作算子是完全有效的，而且像setf( )这样的ostream成员函数也完全可用。

这个程序利用ends操作算子手工地用null作为数组的结尾。无论数组是自动地以null字符结束，还是依赖于具体实现，如果这对你的应用来说非常重要，那么最好手工地用null字符结

束数组。

可以通过调用 pcount( )成员函数来确定输出数组中的字符数，该函数的原型如下：

```
streamsize pcount( );
```

如果数组有空终止符的话，pcount( )函数的返回值也包括该字符。

下面的程序演示了 pcount( )函数，它指出 outs 含有 18 个字符——17 个字符和 1 个 null 终止符。

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

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "abcdefg ";
  outs << 27 << " " << 890.23;
  outs << ends; // null terminate
  cout << outs.pcount(); // display how many chars in outs

  cout << " " << str;

  return 0;
}
```

23.10.3　将数组用做输入

为了把输入流连接到一个数组，可使用如下的 istrstream 构造函数：

```
istrstream istr(const char *buf);
```

其中，buf 是一个数组指针，它所指向的数组在每次对 istr 执行输入时将用做一个字符源。buf 所指的数组必须以 null 字符结束。然而，这个 null 终止符永远不会从数组中读出。

下面是一个将字符串用做输入的例子。

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

int main()
{
  char s[] = "10 Hello 0x75 42.73 OK";

  istrstream ins(s);

  int i;
  char str[80];
  float f;

  // reading: 10 Hello
  ins >> i;
  ins >> str;
```

```
  cout << i << " " << str << endl;

  // reading 0x75 42.73 OK
  ins >> hex >> i;
  ins >> f;
  ins >> str;

  cout << hex << i << " " << f << " " << str;

  return 0;
}
```

如果只想将字符串的一部分用做输入，可使用如下形式的 istrstream 函数：

istrstream istr(const char *buf, streamsize size);

这里，在 buf 所指的数组中，只有前 size 个数组元素会被使用。因为字符串的大小由 size 的值决定，所以该字符串不必以 null 字符结束。

连接到存储器的流就像连接到其他设备的流一样。例如，下面的程序演示了任何一个字符数组是怎样被读取的。当到达数组尾（如同文件尾）时，ins 将为 false。

```
/* This program shows how to read the contents of any
   array that contains text. */
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10.23 this is a test <<>><<?!\n";

  istrstream ins(s);

  char ch;

  /* This will read and display the contents
     of any text array. */

  ins.unsetf(ios::skipws); // don't skip spaces
  while (ins) { // false when end of array is reached
    ins >> ch;
    cout << ch;
  }

  return 0;
}
```

23.10.4 输入/输出基于数组的流

为了创建一个既可以执行输入又可以执行输出的基于数组的流，可使用下面的strstream构造函数：

strstream *iostr*(char *buf, streamsize *size*, openmode *mode* = ios::in | ios::out);

其中，buf 指向一个用于 I/O 操作的字符串；size 的值指定数组大小；mode 的值决定 iostr 的操作方式。对一般的输入/输出操作而言，mode 将是 ios::in | ios::out。对于输入来说，数组

必须以 null 结束。

下面的程序使用一个数组执行输入和输出操作。

```
// Perform both input and output.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iostr[80];

  strstream strio(iostr, sizecf(iostr), ios::in | ios::out);

  int a, b;
  char str[80];

  strio << "10 20 testing ";
  strio >> a >> b >> str;
  cout << a << " " << b << " " << str << endl;

  return 0;
}
```

上面的程序先是把 10 20 testing 写到数组中，然后再将其读回。

23.10.5　使用动态数组

在前面的例子中，当把一个流连接到一个输出数组时，该数组和数组的大小都被传递给 ostrstream构造函数。如果你知道将要把输出定向到的数组可容纳的最大字符数，这种方法还是很不错的。然而，如果不知道输出数组的大小，那么会怎么样呢？要解决这个问题，可以使用另一种形式的 ostrstream 构造函数，这种构造函数如下所示：

```
ostrstream();
```

当使用这种构造函数时，ostrstream创建并维护一个动态分配的数组，这种数组的长度可以自动增加以适应存储输出的需要。

为了访问动态分配的数组，必须使用另一个称为 str( )的函数，该函数的原型如下：

```
char *str();
```

这个函数可以“冻结”数组并返回一个指向该数组的指针。利用 str( )返回的指针可以像字符串一样访问该动态数组。一旦动态数组被冻结，该数组就不能再次被用做输出，除非数组被解冻（参见下面的内容）。因此，你一定不想在把字符全部输出到数组之前冻结该数组。

下面是一个使用动态输出数组的程序。

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

int main()
{
  char *p;

  ostrstream outs; // dynamically allocate array
```

```
  outs << "C++ array-based I/O ";
  outs << -10 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ends;

  p = outs.str(); // Freeze dynamic buffer and return
                  // pointer to it.
  cout << p;

  return 0;
}
```

还可以利用strstream类使用动态I/O数组，这样既可以对一个数组执行输入，也可以对其执行输出。

通过调用freeze( )函数，可以对一个动态数组进行冻结或解冻。该函数的原型如下所示：

```
void freeze(bool action = true);
```

如果action为true，数组被冻结；如果action为false，数组被解冻。

23.10.6 使用基于数组的流的二进制I/O

记住，基于数组的I/O具备一般I/O的全部功能。因此，连接到基于数组的流的数组也可以包含二进制信息。当读取二进制信息时，有可能需要使用eof( )函数确定何时数组结束。例如，下面的程序说明了如何利用get( )函数读取数组（二进制数组或文本数组）的内容。

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

int main()
{
  char *p = "this is a test\1\2\3\4\5\6\7";

  istrstream ins(p);

  char ch;

  // read and display binary info
  while (!ins.eof()) {
    ins.get(ch);
    cout << hex << (int) ch << ' ';

}
  return 0;
}
```

在上面的例子中，由\1\2\3等构成的值是非打印值。

为了输出二进制字符，可使用put( )函数。如果需要读取存放二进制数据的缓冲区，可以使用read( )成员函数。为了将数据写入二进制数据的缓冲区，可以使用write( )函数。

23.11 C与C++的区别

在很大程度上讲，标准C++是标准C的超集，实际上，所有C程序也是C++程序。然而，

二者之间存在着少许区别，对这些区别的讨论贯穿在本书第一部分和第二部分之中。我们在这里对最重要的区别做个总结。

在C++中，局部变量可以在一个块中的任何地方声明；而在C中，局部变量必须在块的开始处（即在所有“动作”语句之前）声明（C99已经取消了这种限制）。

在C中，以如下方式声明的函数对函数的参数未加任何说明。

```
int f();
```

也就是说，如果在函数名后面的括号中没有指定任何参数，那么在C中则意味着对参数未做任何声明。该函数可能有参数，也可能没有参数。然而在C++中，像这样的函数声明则意味着该函数没有参数。也就是说，在C++中，下面两个声明具有同样的作用：

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

在C++中，参数列表中的void是可选的。许多C++程序员使用它们是为了向阅读程序的人清楚地表明某个函数没有任何参数，但是从技术上讲，void不是必须的。

在C++中，所有函数都必须被设计为原型，但在C中这只是一种选择（尽管编程经验告诉我们，在C程序中也应该采用原型设计）。在C和C++之间还有一个重要且细微的差别：在C中，字符常量自动被计算为整型，而在C++中则不然。

在C中，多次声明一个全局变量虽然不可取，但是并不算错，而在C++中将引发错误。

在C中，一个标识符至少含有31个有效字符。在C++中，所有字符都是有效字符。然而，从实用的角度看，过长的标识符并不实用而且很少用到。

在C中，虽然在程序内部调用main( )的情况不常见，但这种做法是允许的。在C++中，则不允许这样做。

在C中，不能获得register变量的地址；而在C++中则可以获得这个地址。

在C中，如果在一些类型声明语句中没有类型说明符，那么该类型被假设为int。这种默认规则不再适用于C++。