第7章 常 量

常量概念的建立（由关键字const表示）允许程序员在变化和不变化之间划一条界线。

在C++程序设计项目中提供了安全性和可控性。自从常量问世以来，它就有着很多不同的作用。与此同时，它在C语言中的意义又不一样。开始时，看起来容易混淆。在这一章里，我们将介绍什么时候、为什么和怎样使用关键字 const。最后，讨论volatile,它是const的“兄弟”（因为它们都关系到是否变化，而且语法也一样）。

const的最初动机是取代预处理器 #defines进行值替代。从此它曾被用于指针、函数变量、返回类型、类对象及其成员函数。所有这些用法都稍有区别，但它们在概念上是一致的，我们将在以下各节中说明这些用法。

7.1 值替代

用C语言进行程序设计时，预处理器可以不受限制地建立宏并用它来替代值。因为预处理器只做文本替代，它既没有类型检查思想，也没有类型检查工具，所以预处理器的值替代会产生一些微小的问题，这些问题在C++中可通过使用const而避免。

C语言中预处理器用值替代名字的典型用法是这样的：

```
#define BUFSIZE 100
```

BUFSIZE是一个名字，它不占用存储空间且能放在一个头文件里，目的是为使用它的所有编译单元提供一个值。用值替代而不是用所谓的“不可思议的数”，这对于支持代码维护是非常重要的。如果代码中用到不可思议的数，读者不仅不清楚这个数字来自哪里，而且也不知道它代表什么，进而，当决定改变一个值时，程序员必须执行手动编辑，而且还不能跟踪以保证没有漏掉其中的一个。

多数情况，BUFSIZE的工作方式与普通变量一样但也不都如此；而且这种方法还存在一个类型问题。这就会隐藏一些很难发现的错误。 C++用const把值替代带进编译器领域来解决这些问题。可以这样写：

```
const bufsize=100；
```

或用更清楚的形式：

```
const int bufsize=100；
```

这样就可以在任何编译器需要知道这个值的地方使用 bufsize，同时它还可以执行常量折叠，也就是说，编译器在编译时可以通过必要的计算把一个复杂的常量表达式缩减成简单的。这一点在数组定义里显得尤其重要：

```
char buf[bufsize]；
```

我们可以为所有的内部数据类型（ char、int、float和double型）以及由它们所定义的变量（也可以是类的对象，这将在以后章节里讲到）使用限定符 const。我们应该完全用 const取代#define的值替代。

7.1.1 头文件里的const

与使用#define一样，使用const必须把const定义放进头文件里。这样，通过包含头文件，

可把const定义单独放在一个地方并把它分配给一个编译单元。C++中的const默认为内部连接，也就是说，const仅在const被定义过的文件里才是可见的，而在连接时不能被其他编译单元看到。当定义一个常量（const）时，必须赋一个值给它，除非用extern作了清楚的说明：

```
extern const bufsize；
```

虽然上面的extern强制进行了存储空间分配（另外还有一些情况，如取一个 const的地址，也要进行存储空间分配），但是C++编译器通常并不为const分配存储空间，相反它把这个定义保存在它的符号表里。当const被使用时，它在编译时会进行常量折叠。

当然，绝对不为任何const分配存储是不可能的，尤其对于复杂的结构。这种情况下，编译器建立存储，这会阻止常量折叠。这就是 const为什么必须默认内部连接，即连接仅在特别编译单元内的原因；否则，由于众多的 const在多个cpp文件内分配存储，容易引起连接错误，连接程序在多个对象文件里看到同样的定义就会“抱怨”了。然而，因为 const默认内部连接，所以连接程序不会跨过编译单元连接那些定义，因此不会有冲突。对于在大量场合使用的内部数据类型，包括常量表达式，编译器都能执行常量折叠。

7.1.2 const的安全性

const的作用不限于在常数表达式里代替 #defines。如果用运行期间产生的值初始化一个变量而且知道在那个变量寿命期内它是不变的，用 const限定该变量，程序设计中这是一个很好的做法。如果偶然改变了它，编译器会给出一个出错信息。下面是一个例子：

```
//: SAFECONS.CPP -- Using const for safety
#include <iostream.h>

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
}
```

我们会发现，i是一个编译期间的常量，但j是从i中计算出来的。然而，由于i是一个常量，j的计算值来自一个常数表达式，而它自身也是一个编译期间的常量。紧接下面的一行需要 j的地址，所以迫使编译器给j分配存储空间。即使分配了存储空间，把j值保存在程序的某个地方，由于编译器知道j是常量，而且知道j值是有效的，所以，这仍不会妨碍在决定数组 buf的大小时使用j。

在主函数main()里，对于标识符c中有另一种const，因为其值在编译期间是不知道的。这意味着需要存储空间，而编译器不想在符号表里保留任何东西（和 C的方式一样）。初始化必须发生在定义的地方，而且一旦初始化，其值不能改变。我们看到 c2由c的值计算出来，也会看到这类常量的作用域与其他任何类型常量的作用域是一样的——这是对#define用法的另一种

改进。

实际上，如果想一个值不变，就应该使之成为常量（ const)。这不仅为防止意外的更改提供安全措施，也消除了存储和读内存操作，使编译器产生的代码更有效。

7.1.3 集合

const可以用于集合，但编译器不能把一个集合存放在它的符号表里，所以必须分配内存。在这种情况下，const意味着“ 不能改变的一块存储 ”。然而，其值在编译时不能被使用，因为编译器在编译时不需要知道存储的内容。这样，我们不能写：

```
//: CONSTAG.CPP -- Constants and aggregates

const int i[] = { 1, 2, 3, 4 };

//! float f[i[3]]; // Illegal

struct s { int i, j; };

const s S[] = { { 1, 2 }, { 3, 4 } };

//! double d[S[1].j]; // Illegal

main() {}
```

在一个数组定义里，编译器必须能产生这样的移动存储数组的栈指针代码。在上面这两种非法定义里，编译器给出“ 提示 ”是因为它不能在数组定义里找到一个常数表达式。

7.1.4 与C语言的区别

常量引进是在早期的C++版本中，当时标准C规范正在制订。那时，常量被看作是一个好的思想而被包含在C中。但是，C中的const意思是“ 一个不能被改变的普通变量 ”，在C中，它总是占用存储而且它的名字是全局符。C编译器不能把const看成一个编译期间的常量。在C中，如果写：

```
const bufsize=100；
char buf[bufsize]；
```

尽管看起来好像做了一件合理的事，但这将得到一个错误结果。因为 bufsize占用存储的某个地方，所以C编译器不知道它在编译时的值。在C语言中可以选择这样书写：

```
const bufsize；
```

这样写在C++中是不对的，而C编译器则把它作为一个声明，这个声明指明在别的地方有存储分配。因为C默认const是外部连接的，C++默认const是内部连接的，这样，如果在C++中想完成与C中同样的事情，必须用extern把连接改成外部连接：

```
extern const bufsize;//declaration only
```

这种方法也可用在C语言中。

在C语言中使用限定符const不是很有用，即使是在常数表达式里（必须在编译期间被求出）想使用一个已命名的值，使用const也不是很有用的。C迫使程序员在预处理器里使用#define。

7.2 指针

我们还可以使指针成为const指针。当处理const指针时，编译器仍将努力阻止存储分配并进行常量折叠，但在这种情况下，这些特征似乎很少有用。更重要的是，如果程序员以后想在程序代码中改变这种指针的使用，编译器将给出通知。这大大增加了安全性。

当使用带有指针的const时，有两种选择：或者const修饰指针正指向对象，或者const修饰存储在指针本身的地址里。这些语法在开始时有点使人混淆，但练习之后就好了。

7.2.1 指向const的指针

使用指针定义的技巧，正如任何复杂的定义一样，是在标识符的开始处读它并从里向外读。const指定那个“最靠近”的。这样，如果要使正指向的元素不发生改变，我们得写一个像这样的定义：

```
const int* x；
```

从标识符开始，是这样读的：“x是一个指针，它指向一个const int。”这里不需要初始化，因为说x可以指向任何东西（那是说，它不是一个const），但它所指的东西是不能被改变的。

这是一个容易混淆的部分。有人可能认为：要想指针本身不变，即包含在指针 x里的地址不变，可简单地像这样把const从一边移向另一边：

```
int const* x；
```

并非所有的人都很肯定地认为：应该读成“ x是一个指向int的const指针”。然而，实际上应读成“x是一个指向恰好是const的int普通指针”。即const又把它自己与int结合在一起，结果与前面定义一样。两个定义是一样的，这一点容易使人混淆。为使程序更具有可读性，我们应该坚持用第一种形式。

7.2.2 const指针

使指针本身成为一个const指针，必须把const标明的部分放在*的右边，如：

```
int d=1；
int* const x=&d；
```

现在它读成“ x是一个指针，这个指针是指向int的const指针”。因为现在指针本身是const指针，编译器要求给它一个初始化值，这个值在指针寿命期间不变。然而要改变它所指向的值是可以的，可以写*x=2；

也可以使用下面两种合法形式中的任何一种形式把一个const指针变为一个const对象：

```
int d=1；
const int* const x=&d； // (1)
int const* const x2=&d； // (2)
```

现在，指针和对象都不能改变。

一些人认为第二种形式更好。因为const总是放在被修改者的右边。但对于特定的代码类型来讲，程序员得自己决定哪一种形式更清楚。

• 格式

这本书主张，不管何时在一行里仅放一个指针定义，且在定义的地方初始化每个指针。正因为这一点，才可以把‘ * ’“ 附于 ”数据类型上：

```
int* u=&w；
```

int*本身好像是离散型的。这使代码更容易懂，可惜的是，事情并非如此。事实上，‘ * ’与标识符结合，而不是与类型结合。它可以被放在类型名和标识符之间的任何地方。所以，可以这样做：

```
int* u=&w, v=0;
```

它建立一个int* u和一个非指针int v。由于读者时常混淆这一点，所以最好用本书里所用的表示形式（即一行里只定义一个指针）。

7.2.3 赋值和类型检查

C++关于类型检查有其特别之处，这一点也扩展到指针赋值。我们可以把一个非 const对象的地址赋给一个const指针，因为也许有时不想改变某些可以改变的东西。然而，不能把一个const对象的地址赋给一个非 const指针，因为这样做可能通过被赋值指针改变这个 const指针。当然，总能用类型转换强制进行这样的赋值，但是，这不是一个好的程序设计习惯，因为这样就打破了对象的const属性以及由const提供的安全性。例如：

```
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
int* v = &e; // illegal -- e const
int* w = (int*)&e; // legal but bad practice
```

虽然C++有助于防止错误发生，但如果程序员自己打破了这种安全机制，它也是无能为力的。

• 串字面值

限定词const是很严格的，const没被强调的地方是有关串字面值。也许有人写：

```
char* cp="howdy";
```

编译器将接受它而不报告错误。从技术上讲，这是一个错误，因为串字面值（这里是“ howdy ”）是被编译器作为一个常量串建立的，所引用串的结果是它在内存里的首地址。

所以串字面值实际上是常量串。然而，编译器把它们作为非常量看待，这是因为有许多现有的C代码是这样做的。当然，改变串字面值的做法还未被定义，虽然可能在很多机器上是这样做的。

7.3 函数参数和返回值

用const限定函数参数及返回值是常量概念另一个容易被混淆的地方。如果以值传递对象时，对用户来讲，用const限定没有意义（它意味着传递的参数在函数里是不能被修改的）。如果以常量返回用户定义类型的一个对象的值，这意味着返回值不能被修改。如果传递并返回地址，const将保证该地址内容不会被改变。

7.3.1 传递const值

如果函数是以值传递的，可用const限定函数参数，如：

```
void f1(const int i) {
  i++; // illegal -- compile-time error
}
```

这是什么意思呢？这是作了一个约定：变量初值不会被函数x()改变。然而，由于参数是以值传递的，因此要立即制作原变量的副本，这个约定对用户来说是隐藏的。

在函数里，const有这样的意义：参数不能被改变。所以它其实是函数创建者的工具，而不是函数调用者的工具。

为了不使调用者混淆，在函数内部用const限定参数优于在参数表里用const限定参数。可以用一个指针这样做，但更好的语法形式是“引用”，这是第10章讨论的主题。简言之，引用像一个被自动逆向引用的常指针，它的作用是成为对象的别名。为建立一个引用，在定义里使用&。所以，不引起混淆的函数定义看来像这样的：

```
void f2(int ic) {
  const int& i = ic;
  i++;  // illegal -- compile-time error
}
```

这又会得到一个错误信息，但这时对象的常量性（const）不是函数特征标志的部分；它仅对函数实现有意义，所以它对用户来说是不可见的。

7.3.2 返回const值

对返回值来讲，存在一个类似的道理，即如果从一个函数中返回值，这个值作为一个常量：

```
const int g()；
```

约定了函数框架里的原变量不会被修改。正如前面讲的，返回这个变量的值，因为这个变量被制成副本，所以初值不会被修改。

首先，这使const看起来没有什么意义。可以从这个例子中看到：返回常量值明显失去意义：

```
//: CONSTVAL.CPP -- Returning consts by value
// Has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
}
```

对于内部数据类型来说，返回值是否是常量并没有关系，所以返回一个内部数据类型的值时，应该去掉const从而使用户程序员不混淆。

处理用户定义的类型时，返回值为常量是很重要的。如果一个函数返回一个类对象的值，其值是常量，那么这个函数的返回值不能是一个左值（即它不能被赋值，也不能被修改）。例如：

```
//: CONSTRET.CPP -- Constant return by value
// Result cannot be used as an lvalue
class X {
```

```
  int i;
public:
  X(int I = 0) : i(I) {}
  void modify() { i++; }
};

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Pass by non-const reference
  x.modify();
}

main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
  f7(f5()); // OK
  // Causes compile-time errors:
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
}
```

f5()返回一个非const X对象，然而f6()返回一个const X对象。只有非const返回值能作为一个左值使用。换句话说，如果不让对象的返回值作为一个左值使用，当返回一个对象的值时，应该使用const。

返回一个内部数据类型的值时，const没有意义的原因是：编译器已经不让它成为一个左值（因为它总是一个值而不是一个变量）。仅当返回用户定义的类型对象的值时，才会出现上述问题。

函数f7()把它的参数作为一个非const引用（C++中另一种处理地址的办法，这是第10章讨论的主题）。从效果上讲，这与取一个非const指针一样，只是语法不同。

• 临时变量

有时候，在求表达式值期间，编译器必须建立临时对象。像其他任何对象一样，它们需要存储空间而且必须被构造和删除。区别是我们从来看不到它们——编译器负责决定它们的去留以及它们存在的细节。这里有一个关于临时变量的情况：它们自动地成为常量。因为我们通常接触不到临时对象，不能使用与之相关的信息，所以告诉临时对象做一些改变几乎肯定会出错。当程序员犯那样的错误时，由于使所有的临时变量自动地成为常量，编译器会向他发出错误警告。

类对象常量是怎样保存起来的，将在这一章的后面介绍。

7.3.3 传递和返回地址

如果传递或返回一个指针（或一个引用），用户取指针并修改初值是可能的。如果使这个指针成为常（const）指针，就会阻止这类事的发生，这是非常重要的。事实上，无论什么时候传递一个地址给一个函数，我们都应该尽可能用 const修饰它，如果不这样做，就使得带有指向const的指针函数不具备可用性。

是否选择返回一个指向const的指针，取决于我们想让用户用它干什么。下面这个例子表明了如何使用const指针作为函数参数和返回值：

```
//: CONSTP.CPP -- Constant pointer arg/return

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static string:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
}
```

函数t()把一个普通的非const指针作为一个参数，而函数u()把一个const指针作为参数。在

函数u()里，我们会看到试图修改const指针的内容是非法的。当然，我们可以把信息拷进一个非const变量。编译器也不允许使用存储在const指针里的地址来建立一个非const指针。

函数v()和w()测试返回的语义值。函数v()返回一个从串字面值中建立的const char*。在编译器建立了它并把它存储在静态存储区之后，这个声明实际上产生串字面值的地址。像前面提到的一样，从技术上讲，这个串是一个常量，这个常量由函数v()的返回值正确地表示。

w()的返回值要求这个指针及这个指针所指向的对象均为常量。像函数 v()一样，仅仅因为它是静态的，所以在函数返回后由 w()返回的值是有效的。函数不能返回指向局部栈变量的指针，这是因为在函数返回后它们是无效的，而且栈也被清理了。可返回的另一个普通指针是在堆中分配的存储地址，在函数返回后它仍然有效。

在函数main()中，函数被各种参数测试。函数t()将接受一个非const指针参数。但是，如果我们想传给它一个指向const的指针，那么就将不能防止t()丢下这个指针所指的内容不管，所以编译器会给出一个错误信息。函数 u()带一个const指针，所以它接受两种类型的参数。这样，带const指针函数比不带const指针函数更具一般性。

正如所期望的，函数v()返回值只可以被赋给一个const指针。编译器拒绝把函数w()的返回值赋给一个非const指针，而接受一个const int* const，但令人吃惊的是它也接受一个const int*，这与返回类型不匹配。正如前面所讲的，因为这个值（包含在指针中的地址）正被拷贝，所以自动保持这样的约定：原始变量不能被触动。因此，只有把 const int*const中的第二个const当作一个左值使用时（编译器会阻止这种情况），它才能显示其意义。

• 标准参数传递

在C语言中，值传递是很普通的，但是当我们想传递地址时，只能使用指针。然而，在C++中却不使用这两种方法。在C++中，传递一个参数时，先选择通过引用传递，而且是通过常量（const）引用。对程序员来说，这样做的语法与值传递是一样的，所以在指针方面没有混淆之处的——他们甚至不必考虑这个问题。对于类的创建者来说，传递地址总比传递整个类对象更有效，如通过常量（ const）引用来传递，这意味着函数将不改变该地址所指的内容，从用户程序员的观点来看，效果恰好与值传递一样。

由于引用的语法（看起来像值传递）的原因，传递一个临时对象给带有一个引用的函数，是可能的，但不能传递一个临时对象给带有一个指针的函数——因为它必须清楚地带有地址。所以，通过引用传递会产生一个在C中不会出现的新情形：一个总是常量的临时变量，它的地址可以被传递给一个函数。这就是为什么临时变量通过引用被传递给一个函数时，这个函数的参数一定是常量（const）引用。下面的例子说明了这一点：

```
//: CONSTTMP.CPP -- Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference
main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
}
```

函数f()返回类X的一个对象的值。这意味着立即取 f()的返回值并把它传递给其他函数时（正如g1()和g2()函数的调用），建立了一个临时变量，那个临时变量是常量。这样，函数g1()中的调用是错误的，因为g1()不带一个常量（const）引用，但是函数g2()中的调用是正确的。

7.4 类

这一部分介绍了const用于类的两种办法。程序员可能想在一个类里建立一个局部常量，将它用在常数表达式里，这个常数表达式在编译期间被求值。然而， const的意思在类里是不同的，所以必须使用另一技术——枚举，以达到同样的效果。

我们还可以建立一个类对象常量（const）（正如我们刚刚看到的，编译器总是建立临时类对象常量）。但是，要保持类对象为常量却比较复杂。编译器能保证一个内部数据类型为常量，但不能控制一个类中错综复杂的事物。为了保证一个类对象为常量，引进了 const成员函数：对于一个常量对象，只能调用const成员函数。

7.4.1 类里的const和enum

常数表达式使用常量的情况之一是在类里。典型的例子是在一个类里建立一个数组，并用const代替#define建立数组大小以及用于有关数组的计算。并把数组大小一直隐藏在类里，这样，如果用size表示数组大小，就可以把 size这个名字用在另一个类里而不发生冲突。然而预处理器从这些#define被定义的时起就把它们看成全程的，所以如用 #define就不会得到预期的效果。

起初读者可能认为合乎逻辑的选择是把一个const放在类里。但这不会产生预期的结果。在一个类里， const恢复它在C中的一部分意思。它在每个类对象里分配存储并代表一个值，这个值一旦被初始化以后就不能改变。在一个类里使用 const的意思是“在这个对象寿命期内，这是一个常量”。然而，对这个常量来讲，每个不同的对象可以含一个不同的值。

这样，在一个类里建立一个const时，不能给它初值。这个初始化工作必须发生在构造函数里，并且，要在构造函数的某个特别的地方。因为 const必须在建立它的地方被初始化，所以在构造函数的主体里， const必须已初始化了，否则，就只有等待，直到在构造函数主体以后的某个地方给它初始化，这意味着过一会儿才给 const初始化。当然，无法防止在在构造函数主体的不同地方改变const的值。

1. 构造函数初始化表达式表

构造函数有个特殊的初始化方法，称为构造函数初始化表达式表，起初用在继承里（继承是以后章节中有关面向对象的主题）。构造函数初始化表达式表——顾名思义，是出现在构造函数的定义里的——是一个出现在函数参数表和冒号后，但在构造函数主体开头的花括号前的“函数调用表”。这提醒人们，表里的初始化发生在构造函数的任何代码执行之前。这是把所有的const初始化的地方，所以类里的const正确形式是：

```
class fred {
  const size;
public:
  fred();
};

fred::fred() : size(100) {}
```

开始时，上面显示的构造函数初始化表达式表的形式容易使人们混淆，因为人们不习惯看到一个内部数据类型有一个构造函数。

2. 内部数据类型“构造函数”

随着语言的发展和人们为使用户定义类型像内部数据类型所作的努力，有时似乎使内部数据类型看起来像用户定义类型更好。在构造函数初始化表达式表里，可以把一个内部数据类型看成好像它有一个构造函数，就像下面这样：

```
class B {
  int i;
public:
  B(int I);
};

B::B(int I) : i(I) {}
```

这在初始化const数据成员时尤为典型，因为它们必须在进入函数体前被初始化。

我们还可以把这个内部数据类型的“构造函数”（仅指赋值）扩展为一般的情形，可以写：

```
float pi (3.14159);
```

把一个内部数据类型封装在一个类里以保证用构造函数初始化，是很有用的。例如，下面是一个integer类：

```
class integer {
  int i;
public:
  integer(int I = 0);
};

integer::integer(int I) : i(I) {}
```

现在，如果建立一个integer数组，它们都被自动初始化为零：

```
integer I[100];
```

与for循环和memset()相比，这种初始化不必付出更多的开销。很多编译器可以很容易地把它优化成一个很快的过程。

7.4.2 编译期间类里的常量

因为在类对象里进行了存储空间分配，编译器不能知道const的内容是什么，所以不能把它用作编译期间的常量。这意味着对于类里的常数表达式来说，const就像它在C中一样没有作用。我们不能这样写：

```
class bob {
  const size = 100;  // illegal
  int array[size];  // illegal
//...
```

在类里的const意思是“在这个特定对象的寿命期内，而不是对于整个类来说，这个值是不变的（const）”。那么怎样建立一个可以用在常数表达式里的类常量呢？一个普通的办法是使用一个不带实例的无标记的 enum。枚举的所有值必须在编译时建立，它对类来说是局部的，但常数表达式能得到它的值，这样，我们一般会看到：

```
class bunch {
  enum { size = 1000 };
  int i[size];
};
```

使用enum是不会占用对象中的存储空间的，枚举常量在编译时被全部求值。我们也可以明确地建立枚举常量的值：

```
enum { one=1,two=2,three};
```

对于整型enum，编译器从最后一个值继续计数，所以枚举常量three将取值3。

下面这个例子表明了在一个串指针栈里的enum的用法：

```
//: SSTACK.CPP -- Enums inside classes
#include <string.h>
#include <iostream.h>

class StringStack {
  enum { size = 100 };
  const char* stack[size];
  int index;
public:
   StringStack();
   void push(const char* s);
   const char* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(char*));
}

void StringStack::push(const char* s) {
  if(index < size)
    stack[index++] = s;
}
const char* StringStack::pop() {
  if(index > 0) {
    const char* rv = stack[--index];
    stack[index] = 0;
    return rv;
  }
  return 0;
}

const char* iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
```

```
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const ICsz = sizeof iceCream/sizeof *iceCream;

main() {
  StringStack SS;
  for(int i = 0; i < ICsz; i++)
    SS.push(iceCream[i]);
  const char* cp;
  while((cp = SS.pop()) != 0)
    cout << cp << endl;
}
```

注意push()带一个const char*参数，pop()返回一个const char*，stack保存const char*。如果不是这样，就不能用stringstack保存iceCream里的指针。然而，它不让程序员做任何事情以改变包含在Stringstack里的对象。当然，不是所有的串指针栈都有这个限制。

虽然会经常在以前的程序代码里看到使用enum技术，但C++还有一个静态常量static const，它在一个类里产生一个更灵活的编译期间的常量。这一点在将第9章描述。

• 枚举的类型检查

C中的枚举是相当原始的，只涉及整型值和名字，但不提供类型检查。在C++里，正如我们现在所期望的，类型概念是十分重要的，枚举正是这样要求的。我们建立了一个已命名的枚举时，我们就已经有效地建立了一个新的类型，就像一个类一样：在编译单元被翻译期间，枚举名字将成为一个保留字。

另外，C++中的枚举有一个比C中更严格的类型检查。假如我们有一个称为a的枚举类型color，就会注意这一点。在C中可以写a++，但在C++中不能这样写。这是因为枚举自增正在执行两个类型转换，其中一个类型在C++中是合法的，另一个是不合法的。首先，枚举的值隐蔽地从color转换到int，然后值增1，然后int又转回到color。在C++中，这样做是不允许的，因为color是一个与int不同的类型，无法知道blue加1恰好出现在颜色表里。如果要对color加1，那么它应该是一个类（有自增操作），而不是一个enum。不论什么时候写出了隐含对enum进行类型转换的代码，编译器都把它标记成危险的活动。

共用数据类型有类似的附加类型检查。

7.4.3 const对象和成员函数

可以用const限定类成员函数，这是什么意思呢？为了搞清楚这一点，必须首先掌握const对象的概念。

用户定义类型和内部数据类型一样，都可以定义一个const对象。例如：

```
const int i=1；
const blob B(2)；
```

这里，B是类型blob的一个const对象。它的构造函数被调用，且其参数为“ 2 ”。由于编译器强调对象为const的，因此它必须保证对象的数据成员在对象寿命期内不被改变。可以很容易地保证公有数据不被改变，但是怎么知道哪个成员函数会改变数据？又怎么知道哪个成员函数对于const对象来说是“ 安全 ”的呢？

如果声明一个成员函数为 const函数，则等于告诉编译器可以为一个 const对象调用这个函数。一个没有被特别声明为 const的成员函数被看成是将要修改对象中数据成员的函数，而且编译器不允许为一个const对象调用这个函数。

然而，不能就此为止。仅仅声明一个函数在类定义里是 const的，不能保证成员函数也如此定义，所以编译器迫使程序员在定义函数时要重申 const说明。(const已成为函数识别符的一部分，所以编译器和连接程序都要检查 const)。为确保函数的常量性，在函数定义中，如果我们改变对象中的任何成员或调用一个非 const成员函数，编译器都将发出一个出错信息，强调在函数定义期间函数被定义成 const函数。这样，可以保证声明为 const的任何成员函数能够按定义方式运行。

const放在函数声明前意味着返回值是常量，但这不合语法。必须把 const标识符放在参数表后。例如：

```
class X {
  int i;
public:
  int f() const;
};
```

关键字const必须用同样的方式重复出现在定义里，否则编译器把它看成一个不同的函数：

```
int X::f() const {return i ; }
```

如果f()试图用任何方式改变i或调用另一个非const成员函数，编译器把它标记成一个错误。

任何不修改成员数据的函数应该声明为const函数，这样它可以由const对象使用。

下面是一个比较const和非const成员函数的例子：

```
//: QUOTER.CPP -- Random quote selection
#include <iostream.h>
#include <stdlib.h> // Random number generator
#include <time.h> // To seed random generator

class quoter {
  int lastquote;
public:
  quoter();
  int Lastquote() const;
  const char* quote();
};

quoter::quoter(){
  lastquote = -1;
  time_t t;
  srand((unsigned) time(&t)); // Seed generator
}
```

```
int quoter::Lastquote() const {
  return lastquote;
}

const char* quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
  };
  const qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

main() {
  quoter q;
  const quoter cq;
  cq.Lastquote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
}
```

构造函数和析构函数都不是const成员函数，因为它们在初始化和清理时，总是对对象作些修改。quote()成员函数也不能是const函数，因为它在返回说明里修改数据成员lastquote。然而Lastquote()没做修改，所以它可以成为const函数，而且也可以被const对象cq安全地调用。

• 按位和与按成员 const

如果我们想要建立一个const成员函数，但仍然想在对象里改变某些数据，这时该怎么办呢？这关系到按位const和按成员const的区别。按位const意思是对象中的每个位是固定的，所以对象的每个位映像从不改变。按成员 const意思是，虽然整个对象从概念上讲是不变的，但是某个成员可能有变化。当编译器被告知一个对象是 const对象时，它将保护这个对象。这里我们要介绍在const成员函数里改变数据成员的两种方法。

第一种方法已成为过去，称为“强制转换const”。它以相当奇怪的方式执行。取this（这个关键字产生当前对象的地址）并把它强制转换成指向当前类型对象的指针。看来 this已经是我们所需的指针，但它是一个const指针，所以，还应把它强制转换成一个普通指针，这样就可以在运算中去掉常量性。下面是一个例子：

```
//: CASTAWAY.CPP -- "Casting away" constness

class Y {
  int i, j;
public:
  Y() { i = j = 0; }
  void f() const;
};

void Y::f() const {
//!   i++; // Error -- const member function
    ((Y*)this)->j++; //OK: cast away const-ness
}

main() {
  const Y yy;
  yy.f(); // Actually changes it!
}
```

这种方法可行，在过去的程序代码里可以看到这种用法，但这不是首选的技术。问题是：this没有用const修饰，这在一个对象的成员函数里被隐藏，这样，如果用户不能见到源代码（并找到用这种方法的地方），就不知道发生了什么。为解决所有这些问题，应该在类声明里使用关键字mutable，以指定一个特定的数据成员可以在一个const对象里被改变。

```
//: MUTABLE.CPP -- The "mutable" keyword

class Y {
  int i;
  mutable int j;
public:
  Y() { i = j = 0; }
  void f() const;
};

void Y::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

main() {
  const Y yy;
  yy.f(); // Actually changes it!
}
```

现在类用户可从声明里看到哪个成员能够在一个const成员函数里被修改。

7.4.4 只读存储能力

如果一个对象被定义成const对象，它就成为被放进只读存储器（ROM）中的一个候选，

这经常是嵌入式程序设计中要考虑的重要事情。然而，只建立一个 const对象是不够的——只读存储能力的条件非常严格。当然，这个对象还应是按位 const的，而不是按成员const的。如果只通过关键字mutable实现按成员常量化的话，就容易看出这一点。如果在一个 const成员函数里的const被强制转换了，编译器可能检测不到这个。另外，

1) class或struct必须没有用户定义的构造函数或析构函数。

2) 这里不能有基类（将在关于继承的章节里谈到），也不能有包含用户定义的构造函数或析构函数的成员对象。

在只读存储能力类型的const对象中的任何部分上，有关写操做的影响没有定义。虽然适当形式的对象可被放进ROM里，但是目前还没有什么对象需要放进ROM里。

7.5 可变的（volatile）

volatile的语法与const是一样的，但是volatile的意思是“在编译器认识的范围外，这个数据可以被改变”。不知何故，环境正在改变数据（可能通过多任务处理），所以，volatile告诉编译器不要擅自做出有关数据的任何假定——在优化期间这是特别重要的。如果编译器说：“我已经把数据读进寄存器，而且再没有与寄存器接触”。一般情况下，它不需要再读这个数据。但是，如果数据是volatile修饰的，编译器不能作出这样的假定，因为可能被其他进程改变了，它必须重读这个数据而不是优化这个代码。

就像建立const对象一样，程序员也可以建立volatile对象，甚至还可以建立const volatile对象，这个对象不能被程序员改变，但可通过外面的工具改变。下面是一个例子，它代表一个类，这个类涉及到硬件通信：

```
//: VOLATILE.CPP -- The volatile keyword
class comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  comm();
  void isr() volatile;
  char read(int Index) const;
};

comm::comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// As an interrupt service routine:
void comm::isr() volatile {
  if(flag) flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}
```

```
char comm::read(int Index) const {
  if(Index < 0 || Index >= bufsize)
    return 0;
  return buf[Index];
}

main() {
  volatile comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Not OK;
                 // read() not volatile
}
```

就像const一样，我们可以对数据成员、成员函数和对象本身使用 volatile，可以并且也只能为volatile对象调用volatile成员函数。

函数isr()不能像中断服务程序那样使用的原因是：在一个成员函数里，当前对象（ this）的地址必须被秘密地传递，而中断服务程序 ISR一般根本不要参数。为解决这个问题，可以使isr()成为静态成员函数，这是下面章节讨论的主题。

volatile的语法与const是一样的，所以经常把它们俩放在一起讨论。为表示可以选择两个中的任何一个，它们俩通称为c-v限定词。

7.6 小结

关键字const能将对象、函数参数、返回值及成员函数定义为常量，并能消除预处理器的值替代而不对预处理器有任何影响。所有这些都为程序设计提供了非常好的类型检查形式以及安全性。使用所谓的const correctness（在可能的任何地方使用const）已成为项目的救星。

对于忽视const而继续使用老的C代码的程序员， 第10、11两章将改变他们的做法，在那里将开始大量使用引用，那时将看到对函数参数使用 const是多么关键。

7.7 练习

1. 建立一个具有成员函数 fly()的名为bird的类和一个不含 fly()的名为rock的类。建立一个rock对象，取它的地址，把它赋给一个 void*。现在取这个void*，把它赋给一个bird*，通过那个指针调用函数fly()。C语言允许公开地通过void*赋值是C语言中的一个“ 缺陷 ”，为什么呢？您知道吗？

2. 建立一个包含const成员的类，在构造函数初始化表达式表里初始化这个 const成员，建立一个无标记的枚举，用它决定一个数组的大小。

3. 建立一个类，该类具有 const和非const成员函数。建立这个类的const和非const对象，试着为不同类型的对象调用不同类型的成员函数。

4. 创建一个函数，这个函数带有一个常量值参数。然后试着在函数体内改变这个参数。

5. 请自行证明C和C++编译器对于const的处理是不同的。创建一个全局的 const并将它用于一个常量表达式中；然后在C和C++下编译它。