第 3 章

C++数据类型

本章将概括介绍 C++中预定义的*内置的*、或*基本的*数据类型。本章将以*文字常量*(*literal constant*)开始，如 3.14159 和“pi”，然后介绍*符号变量*(*symbolic variable*)或者*对象*(*object*)的概念。C++程序中的对象必须被定义为某一种特定的类型，本章的余下部分将介绍可以用来声明对象的各种类型。另外，我们还将把 C++内置的对字符串与数组的支持与 C++标准库提供的类抽象进行对比。虽然标准库中的抽象类型不是基本类型，但是它们也是实用 C++程序的基础。我们希望尽早地介绍它们，以此来鼓励和说明它们的使用。我们把这些类型看作是基本内置类型和基本类抽象类型的*扩展基础语言*。

3.1 文 字 常 量

C++预定义了一组数值数据类型，可以用来表示整数、浮点数和单个字符，还预定义了用来表示字符串的字符数组。

- 字符型 char，在典型情况下，用它表示单个字符和小整数。它可以用一个机器字节来表示。
- 整型 int、短整型 short、长整型 long，它们分别代表不同长度的整数值，典型情况下，short 以半个字表示，int 以一个机器字表示，long 为一个或两个机器字。（在 32 位机器中，int 和 long 通常长度相同）。
- 浮点型 float、双精度 double、长双精度 long double，分别表示单精度浮点数、双精度浮点数和扩展精度的浮点数值。典型情况下，float 为一个字，double 是两个字，long double 为三个或四个字。

char、short、int 和 long 称为*有序类型（integral types）*。有序类型可以有符号，也可以

无符号。在有符号类型中，最左边的位是*符号位*，余下的位代表数值。在无符号类型中，所有的位都表示数值。如果符号位被置为 1，数值被解释成负数；如果是 0，则为正数。一个 8 位有符号的 char 可以代表从-128 到 127 的数值；一个无符号的 char 则表示 0 到 255 范围内的数值。

当一个数值，例如 1，出现在程序中时，它被称为*文字常量（literal constant）*：称为“文字”是因为我们只能以它的值的形式提起它；“常量”是因为它的值不能被改变。每个文字都有相应的类型。例如，0 是 int 型，而 3.14159 是 double 型的文字常量。文字常量是*不可寻址的(nonaddressable)*，尽管它的值也存储在机器内存的某个地方，但是我们没有办法访问它们的地址。

整数文字常量可以被写成十进制、八进制或者十六进制的形式。（这不会改变该整数值的位序列）。例如，20 可以被写成下面三种形式的任意一种：

```
20    // 十进制
024   // 八进制
0x14  // 十六进制
```

在整型文字常量前面加一个 0，该值将被解释成一个八进制数。在前面加一个 0x 或 0X，会使一个整型文字常量被解释成十六进制数。（第 20 章“输入/输出流库”将讨论八进值或十六进制形式的输出值）。

在缺省情况下，整型文字常量被当作是一个 int 型的有符号值。我们可以在文字常量后面加一个“L”或“l”(字母 L 的大写形式或者小写形式)，将其指定为 long 类型。一般情况下，我们应该避免使用小写字母，因为它很容易被误当作数字 1。类似地，我们可以在整型文字常量的后面加上“u”或“U”，将其指定为一个无符号数。我们也可以指定无符号 long 型的文字常量。例如：

```
128u  1024UL    1L    8Lu
```

浮点型文字常量可以被写成科学计数法形式或普通的十进制形式。使用科学计数法，指数可写作“e”或“E”。浮点型文字常量在缺省情况下被认为是 double 型，单精度文字常量由值后面的“f”或“F”来标示。类似地，扩展精度由值后面跟的“l”或“L”来指示。（注意“f”、“F”、“l”、“L”后缀只能用在十进制形式中。）例如：

```
3.14159F  0.1f      12.345L  0.0
3e1       1.0E-3    2.       1.0L
```

单词 true 和 false 是 bool 型的文字常量。例如，可以这样写：

```
true   false
```

可打印的文字字符常量可以写成用单引号括起来的形式。例如：

```
'a'       '2'       ','       ' '(空格)
```

一部分不可打印的字符、单引号、双引号以及反斜杠可以用如下的转义序列来表示（转义序列以反斜杠开始）：

```
newline(换行符)          \n
```

```
horizontal tab(水平制表键)  \t
vertical tab(垂直制表键)    \v
backspace(退格键)          \b
carriage return(回车键)       \r
formfeed(进纸键)           \f
alert (bell) (响铃符)         \a
backslash(反斜杠键)       \\
question mark(问号)         \?
single quote(单引号)        \'
double quote(双引号)       \"
```

一般的转义序列采用如下格式：

```
\ooo
```

这里的 ooo 代表三个八进制数字组成的序列。八进制序列的值代表该字符在机器字符集里的数字值。下面的示例使用 ASCII 码字符集表示文字常量：

```
\7 (bell)     \14 (newline)
\0 (null)     \062 ('2')
```

另外，字符文字前面可以加“ L ”，例如：

```
L'a'
```

这称为*宽字符文字*，类型为 wchar_t。宽字符常量支持语言字符集合，如汉语、日语，这些语言中的某些字符不能用单个字符来表示。

字符串文字常量由零个或多个用双引号括起来的字符组成。不可打印字符可以由相应的转义序列来表示。一个字符串文字可以扩展到多行。在一行的最后加上一个反斜杠，表明字符串文字在下一行继续。例如：

```
"" (空字符串)
"a"
"\nCC\toptions\tfile.[cC]\n"
"a multi-line \
string literal signals its \
continuation with a backslash"
```

字符串文字的类型是*常量字符数组*。它由字符串文字本身以及编译器加上的结束空(null)字符构成。例如，

```
'A'
```

代表单个字符‘ A ’，下面的表示则代表单个字符 A 后面跟一个空字符。

```
"A"
```

空字符是 C 和 C++用来标记字符串结束的符号。

正如存在宽字符文字，比如

```
L'a'
```

同样地，也有宽字符串文字，它仍然以“L”开头，如

```
L"a wide string literal"
```

宽字符串文字的类型是*常量宽字符的数组*。它也有一个等价的宽空字符作为结束标志。

如果两个字符串或宽字符串在程序中相邻，C++就会把它们连接在一起，并在最后加上一个空字符。例如：

```
"two" "some"
```

的输出结果是“*twosome*”。如果试图将一个字符串常量与一个宽字符串常量连接起来，会发生什么后果？例如：

```
// this is not a good idea
"two" L"some"
```

结果是未定义的(undefined) —— 即，没有为这两种不同类型的连接定义标准行为。使用未定义行为的程序被称作是*不可移植的*。虽然程序可能在当前编译器下能正确执行，但是不能保证相同的程序在不同的编译器、或当前编译器的以后版本下编译后，仍然能够正确执行。在以前能够运行的程序中跟踪这类问题是一件很令人不快的任务。因此，建议不要使用未定义的程序特性。我们会在合适的时候指出这样的特性。

练习 3.1

说明下列文字常量的区别。

```
(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0xC
(c) 3.14, 3.14f, 3.14L
```

练习3.2

下列语句哪些是非法的？

```
(a) "Who goes with F\144rgus?\014"
(b) 3.14e1L
(c) "two" L"some"
(d) 1024f
(e) 3.14UL
(f) "multiple line
    comment"
```

3.2 变　　量

假设我们有这样一个问题：计算 2 的 10 次方。我们首先想到的可能是：

```
#include <iostream>

int main() {
```

```
    // a first solution
    cout << "2 raised to the power of 10: ";
    cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2;
    cout << endl;
    return 0;
}
```

这样确实能够解决问题，但是，可能需要检查两到三遍，以确保正好有 10 个常数 2 参与乘法。这个程序产生正确的答案 1024。

接着，我们被要求算出 2 的 17 次方和 2 的 23 次方。每次都要修改程序确实很麻烦。更糟糕的是，这样做经常会出错。修改后的程序常常会多乘或少乘了一个 2。最后我们又被要求生成 2 的从 0 到 15 次方的数值的表。使用文字常量需要写 32 行类似下面的格式：

```
cout << "2 raised to the power of X\t";
cout << 2 * ... * 2;
```

这里 X 随每对语句递增 1。

从某种角度来看，这样确实完成了任务。我们的老板不可能去看我们具体的做法，只要我们的结果正确并且及时就可以。实际上，在许多产品环境中，成功的主要测量标准是最后的结果，至于对处理过程的讨论则被视为学术的、不实际的，因而被抛弃。

虽然这种蛮力型的方案也能解决问题，但是它总让人感到不快，而且有些危机感。这种方案的吸引人之处就是简单：我们明白需要做什么，虽然它常常很乏味。复杂的技术方案一般需要很多启动时间，这时常常会感觉什么都没有做。而且因为处理过程是自动的，所以就更有可能出错。

事情不可避免会出错。好处是，在这些错误过程中，不但事情很快被完成，而且扩展了想象的空间。有时候，这个过程也比较有趣。

在本例中，用来取代这种暴力型的计算 2 的幂的方案包括两部分内容：使用有名字的对象来读写每步的计算；引入一个控制流结构，以便在某个条件为真时，可以重复执行一系列语句。下面是一种“技术先进的”计算 2 的 10 次幂的程序：

```
#include <iostream>

int main()
{
    // objects of type int
    int value = 2;
    int pow = 10;

    cout << value << " raised to the power of "
         << pow << ": \t";

    int res = 1; // holds result

    // loop control statement: repeat calculation of res
    // until cnt is greater than pow
```

```
    for ( int cnt=1; cnt <= pow; ++cnt)
          res = res * value;

    cout << res << endl;
}
```

value、pow、res、cnt 是变量，它们允许对数值进行存储、修改和查询。for 循环使计算过程重复执行 pow 次。

虽然这种层次的通用化使程序更加灵活，但是这样的程序仍然是不可重用的。我们必须进一步通用化：把计算指数值的那部分程序代码抽取出来，定义成一个独立的函数，以使其他函数能够调用它。 例如：

```
int
pow( int val, int exp )
{
    for ( int res = 1; exp > 0; --exp )
         res = res * val;
    return res;
}
```

现在，每个需要计算指数值的程序，都可以使用 pow()的实例，而不是重新实现它。我们可以用如下的代码来生成 2 的幂的表；

```
#include <iostream>
extern int pow(int,int);

int main()
{
    int val = 2;
    int exp = 15;

    cout << "The Powers of 2\n";
    for ( int cnt=0; cnt <= exp; ++cnt )
        cout << cnt << ": "
             << pow(val,cnt) << endl;

    return 0;
}
```

实际上，这个 pow()的实现既不够健壮也不够通用。例如，如果指数是负数，该怎么办？如果是 1000,000 呢？对于负数指数，我们的程序总是返回 1。对于一个非常大的指数，变量 int　res 又小得不能够容纳这个结果。因此，对于一个大的指数将返回一个任意的、不正确的值。（在这种情况下，最好的解决方案是将返回值的类型修改为 double 类型）。从通用的角度来说，我们的程序应该能够处理整数和浮点数类型的底数和指数，甚至其他的类型。正如你所看到的，为一个未知的用户组写一个健壮的通用函数，比“实现一个特定的算法来解决眼前的问题”要复杂得多。pow()的实际实现见[PLAUGER92]。

3.2.1　什么是变量？

变量为我们提供了一个有名字的内存存储区，可以通过程序对其进行读、写和处理。C++中的每个符号变量都与一个特定的数据类型相关联，这个类型决定了相关内存的大小、布局、能够存储在该内存区的值的范围以及可以应用在其上面的操作集。我们也可以把变量说成*对象(object)*。下面是 5 个不同类型的变量定义（在后面我们会介绍变量定义的细节情况）：

```
int    student_count;
double salary;
bool   on_loan;
string street_address;
char   delimiter;
```

变量和文字常量都维护一个存储区，并且有相关的类型。区别在于变量是*可寻址的（addressable）*。对于每一个变量，有两个值与其相关联：

1．它的数据值，存储在某个内存地址中。有时这个值也被称为对象的*右值*(rvalue，读做 are-value)。我们可以把右值想像为*被读取的值(read value)*。文字常量和变量都可被用作右值。

2．它的地址值——即，存储数据值的那块内存的地址。它有时被称为变量的*左值*(lvalue，读作 ell-value)。我们可以把左值想像为*位置值(location value)*。文字常量不能被用作左值。

在下面的表达式中

```
ch = ch - '0';
```

变量 ch 同时出现在赋值操作符的左边和右边。右边的实例被读取，与其相关联的内存中的数据值被读出。左边的 ch 用作写入。减操作的结果被存储在 ch 的位置值所指向的内存区中：原来的数据值被覆盖掉。在表达式的右边，ch 和文字字符常量用作右值 。在左边，ch 用作左值。

一般地，赋值操作符的左边总要求一个左值。例如，下列的写法将产生编译错误：

```
// 编译时刻错误：等号左边不是一个左值

// 错误：文字常量不是一个左值
0 = 1;

// 错误：算术表达式不是一个左值
salary + salary * 0.10 = new_salary;
```

在本书中，我们将会看到许多“左值和右值的用法影响程序的语义行为和性能”的情况——尤其在“向函数传递值”或者“从函数返回值”的时候。

变量的定义会引起相关内存的分配。因为一个对象只能有一个位置，所以程序中的每个

对象只能被定义一次。如果在一个文件中定义的对象需要在另一个文件中被访问，这可能会有问题。例如：

```
// file module0.C
// 定义fileName对象
string fileName;
// ... 为fileName赋一个值

// file module1.C
// 需要使用fileName对象

// oops: 编译失败:
// 在module1.C中，fileName未定义
ifstream input_file( fileName );
```

在 C++中，程序在使用对象之前必须先知道该对象。这对“编译器保证对象在使用时的类型正确性”是必需的。引用一个未知的对象将引起编译错误。在本例中，由于在 model1.C 中没有定义 filenName，所以该文件编译失败。

要编译 model1.C，必须让程序知道 fileName，但又不能引入第二个定义。我们可以通过*声明(declaring)* 该变量来做到这一点：

```
// file module1.C
// 需要使用fileName对象

// 声明fileName，也即，让程序知道它，
// 但又不引入第二个定义
extern string fileName;

ifstream input_file( fileName );
```

一个对象的*声明(declaration)*使程序知道该对象的类型和名字。它由关键字 extern 后面跟上对象的类型以及对象的名字构成。（关于 extern 的全面介绍见 8.2 节。）声明不是定义，不会引起内存分配。实际上，它只是说明了在程序之外的某处有这个变量的定义。

虽然一个程序只能包含一个对象的一个定义，但它可以包含任意数目的对象声明。比较好的做法，不是在每个使用对象的文件中都提供一个单独的声明，而是在一个头文件中声明这个对象，然后在需要声明该对象的时候包含这个头文件。按照这种做法，如果需要修改对象的声明，则只需要修改一次，就能维持多个使用该对象的文件中声明的一致性。（8.2 节将对头文件有更多的说明。）

3.2.2　变量名

变量名，即变量的*标识符*，可以由字母、数字以及下划线字符组成。它必须以字母或下划线开头，并且区分大写字母和小写字母。语言本身对变量名的长度没有限制，但是为用户着想，它不应该过长，如

```
gosh_this_is_an_impossibly_long_name_to_type
```

C++保留了一些词用作关键字。关键字标识符不能再作为程序的标识符使用。我们已经见到过 C++语言的许多关键字。表 3.1 列出了 C++关键字全集。

表 3.1　C++关键字

asm	auto	bool	break	Case
catch	char	Class	const	const_cast
continue	default	delete	do	Double
dynamic_cast	else	enum	explicit	Export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

对于命名对象有许多被普遍接受的习惯，主要考虑因素是程序的可读性。

- 对象名一般用小写字母。例如，我们往往写成 index，而不写 INDEX。（一般把 Index 当作类型名，而 INDEX 则一般被看作常量值，通常用预处理器指示符#define 定义。）
- 标识符一般使用助记的名字——即，能够对程序中的用法提供提示的名字，如 on_loan 或 salary。然而，我们写成 table 或者 tbl，这纯粹是风格问题，不是正确性的问题。
- 对于多个词构成的标识符，习惯上，一般在每个词之间加一个下划线，或内嵌的每个词第一个字母大写。例如，一般会写成 student_loan 或 studentLoan，而不是 studentloan(然而，我在这里已经写了所有三种形式)。一般有面向对象背景的人(ObjectOrientedBackground)喜欢用大写字母，而有 C 或过程化背景的人(C_or_procedural_background)则喜欢下划线。（再次说明，使用 isa、isA 或 is_a 只是个风格问题，与正确与否无关。）

3.2.3　对象的定义

一个简单的对象定义由一个类型指示符后面跟一个名字构成，以分号结束。例如：

```
double salary;
double wage;
```

```
int month;
int day;
int year;
unsigned long distance;
```

当同类型的多个标识符被定义的时候，我们可以在类型指示符后面跟一个由逗号分开的标识符列表。这个列表可跨越多行，最后以分号结束。例如，上面的定义可写成：

```
double salary, wage;
int month,
    day, year;
unsigned long distance;
```

一个简单的定义指定了变量的类型和标识符。它并不提供初始值。如果一个变量是在全局域内(global scope)被定义的，那么它被保护性地提供一个初始值 0。在本例中，salary、wage、month、day、year 以及 distance 都被初始化为 0，因为它们都是在全局域内被定义的。如果变量是在局部域内(local scope)被定义的，或是通过 new 表达式动态分配的，则系统不会向它提供初始值 0。 这些对象被称为是*未初始化的*。一个未初始化的对象不是没有值，而是说，它的值是*未定义的*。（与它相关联的内存区中含有一个随机的位串，可能是以前使用的结果。）

因为使用未初始化对象是个常见错误，而且很难发现，所以，一般建议为每个被定义的对象提供一个初始值。（在有些情况下，这不是必需的；然而，在你能够识别这些情况之前，为每个对象提供初始值是个安全的作法。）类机制通过所谓的*缺省构造函数*（2.3 节已经介绍过）提供了类对象的自动初始化。我们将在本章后面部分关于标准库 string 和复数类型（3.11 节和 3.15 节）的讨论中看到这一点。现在，请注意以下代码：

```
int main() {
    // 未初始化的局部对象
    int ival;

    // 通过string的缺省构造函数进行初始化
    string project;

    // ...
}
```

ival 是一个未初始化的局部变量，但 project 是一个已经初始化的类对象——被缺省的 string 类构造函数自动初始化。

初始的第一个值可以在对象的定义中指定。一个被声明了初始值的对象也被称为*已经初始化的*。C++支持两种形式的初始化。第一种形式是使用赋值操作符的显式语法形式：

```
int ival = 1024;
string project = "Fantasia 2000";
```

在隐式形式中，初始值被放在括号中：

```
int ival( 1024 );
string project( "Fantasia 2000" );
```

在这两种情况中，ival 都被初始化为 1,024，而 project 的初始值为“ Fantasia 2000 ”。逗号分隔的标识符列表同样也能为每个对象提供显式的初始值。语法形式如下：

```
double salary = 9999.99, wage = salary + 0.01;
int month = 08,
day = 07, year = 1955;
```

在一个对象的定义中，当对象的标识符在定义中出现后，对象名立即是可见的，因此用一个对象初始化自已是合法的， 只是这样做不太明智。例如：

```
// 合法，但不明智
int bizarre = bizarre;
```

另外，每种内置数据类型都支持一种特殊的构造函数语法，可将对象初始化为 0。例如：

```
// 设置ival为0，dval为0.0
  int ival = int();
   double dval = double();
```

下列定义中，

```
 // int() applied to each of the 10 elements
    vector< int > ivec( 10 );
```

函数 int()被自动应用在 ivec 包含的 10 个元素上。（2.8 节介绍了 vector。3.6 节与第 6 章将有详细讨论。）

对象可以用任意复杂的表达式来初始化，包括函数的返回值。例如：

```
#include <cmath>
#include <string>

double price = 109.99, discount = 0.16;
double sale_price( price * discount );
string pet( "wrinkles" );

extern int get_value();
int val =  get_value();

unsigned abs_val = abs( val );
```

abs()是标准 C 数学库中预定义的函数， 返回其参数的绝对值。get_value()是一个用户定义的函数，它返回一个随机整数值。

练习 3.3

下列定义哪些是非法的？请改正之。

```
(a) int car = 1024, auto = 2048;
(b) int ival = ival;
(c) int ival( int() );
(d) double salary = wage = 9999.99;
(e) cin >> int input_value;
```

练习 3.4
区分左值与右值，并给出它们的例子。
练习 3.5
说明下列 student **和** name **两个实例的区别。**

```
(a) extern string name;
    string name( "exercise 3.5a" );

(b) extern vector<string> students;
    vector<string> students;
```

练习 3.6
下列名字哪些是非法的？请改正之。

```
(a) int double = 3.14159;     (b) vector< int > _;
(c) string namespace;      (d) string catch-22;
(e) char 1_or_2 = '1';      (f) float Float = 3.14f;
```

练习 3.7
下面的全局对象定义和局部对象定义有什么区别(如果你认为有区别的话)？

```
string global_class;
int global_int;
int main() {
    int local_int;
    string local_class;

    // ...
}
```

3.3 指　针　类　型

在 2.2 **节中，我们简要地介绍了指针和动态内存分配。指针持有另一个对象的地址，使我们能够间接地操作这个对象。指针的典型用法是构建一个链接的数据结构，例如树（**tree**）和链表（**list**），并管理在程序执行过程中动态分配的对象，以及作为函数参数类型，主要用来传递数组或大型的类对象。**

每个指针都有一个相关的类型。不同数据类型的指针之间的区别不是在指针的表示上，也不在指针所持有的值（地址）上 —— 对所有类型的指针这两方面都是相同的[1]。不同之处在于指针所指的对象的类型上。指针的类型指示编译器怎样解释特定地址上内存的内容，以及被解释的内存区域应该跨越多少内存单元。

- 如果一个 int 型的指针寻址到 1000 内存处，那么在 32 位机器上，跨越的地址空间是 1000-1003。
- 如果一个 double 型的指针寻址到 1000 内存处，那么在 32 位机器上，跨越的地址空间是 1000-1007。

下面是指针定义的例子：

```
int            *ip1, *ip2;
complex<double> *cp;
string         *pstring;
vector<int>     *pvec;
double         *dp;
```

我们通过在标识符前加一个解引用操作符（*）来定义指针。在逗号分隔的标识符列表中，每个将被用作指针的标识符前都必须加上解引用操作符。在下面的例子中，lp 是一个指向 long 类型对象的指针，而 lp2 则是一个 long 型的数据对象，而不是指针：

```
long *lp, lp2;
```

在下面的例子中，fp 是一个 float 型的数据对象，而 fp2 是一个指向 float 型对象的指针：

```
float fp, *fp2;
```

为清楚起见，最好写成

```
string *ps;
```

而不是

```
string* ps;
```

有可能发生的情况是，当程序员后来想定义第二个字符串指针时，他会错误地修改定义如下：

```
// 喔：ps2不是一个字符串指针
string* ps, ps2;
```

指针可以持有 0 值，表明它没有指向任何对象，或持有一个同类型的数据对象的地址。已知 ival 的定义：

```
int ival = 1024;
```

下面的定义以及对两个指针 pi 和 pi2 的赋值都是合法的。

[1] 这对函数指针并不成立，函数指针指向程序的代码段。函数指针和数据指针是不同的。函数指针将在7.9节说明。

```
// pi被初始化为“没有指向任何对象”
int *pi = 0;

// pi2被初始化为ival的地址
int *pi2 = &ival;

// ok: pi和pi2现在都指向ival
pi = pi2;

// 现在pi2没有指向任何对象
pi2 = 0;
```

指针不能持有非地址值。例如，下面的赋值将导致编译错误：

```
// 错误：pi被赋以int值ival
pi = ival;
```

指针不能被初始化或赋值为其他类型对象的地址值。例如，已知如下定义：

```
double dval;
double *pd = &dval;
```

那么，下列两条语句都会引起编译时刻错误：

```
// 都是编译时刻错误
// 无效的类型赋值：int* <== double*
pi = pd;
pi = &dval;
```

不是说 pi **在物理上不能持有与** dval **相关联内存的地址：它能够。但是不允许，因为，虽然** pi **和** pd **能够持有同样的地址值，但对那块内存的存储布局和内容的解释却完全不同。**

当然，如果我们要做的仅仅是持有地址值（可能是把一个地址同另一个地址作比较），那么指针的实际类型就不重要了。C++**提供了一种特殊的指针类型来支持这种需求：空（**void***）类型指针，它可以被任何数据指针类型的地址值赋值（函数指针不能赋值给它）。**

```
// ok: void* 可以持有任何指针类型的地址值
void *pv = pi;
pv = pd;
```

void***表明相关的值是个地址，但该地址的对象类型不知道。我们不能够操作空类型指针所指向的对象，只能传送该地址值或将它与其他地址值作比较。（在** 4.14 **节我们将会看到更多关于** void***类型的细节。）**

已知一个 int **型指针对象** pi**，当我们写下** pi **时，**

```
// 计算包含在pi内部的地址值
```

```
// 类型：int*
pi;
```

这将计算 pi 当前持有的地址值。当我们写下&pi 时，

```
// 计算pi的实际地址
// 类型： int**
&pi;
```

这将计算指针对象 pi 被存储的位置的地址。那么，怎样访问 pi 指向的对象呢？

在缺省情况下，我们没有办法访问 pi 指向的对象，以对这个对象进行读或者写的操作。为了访问指针所指向的对象，我们必须解除指针的引用。C++提供了一个*解引用操作符（*）（dereference operator）*来间接地读和写指针所指向的对象。例如，已知下列定义：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival;
```

下面给出了怎样解引用 pi 以便间接访问 ival：

```
// 解除pi的引用，为它所指向的对象ival
// 赋以ival2的值
*pi = ival2;

// 对于右边的实例，读取pi所指对象的值
// 对于左边的实例，则把右边的表达式赋给对象

*pi = abs( *pi ); // ival = abs(ival);
*pi = *pi + 1;    // ival = ival + 1;
```

我们知道，当取一个 int 型对象的地址时，

```
int *pi = &ival;
```

结果是 int* —— 即指向 int 的指针。当我们取指向 int 型的指针的地址时：

```
int **ppi = &pi;
```

结果是 int** —— 即指向 int 指针的指针。当我们解引用 ppi 时：

```
int *pi2 = *ppi;
```

我们获得指针 ppi 持有的地址值 —— 在本例中，即 pi 持有的值，而 pi 又是 ival 的地址。为了实际地访问到 ival，我们需要两次解引用 ppi。例如：

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

下面两条赋值语句的行为截然不同，但它们都是合法的。第一条语句增加了 pi 指向的数据对象的值；而第二条语句增加了 pi 包含的地址的值。

```
int i, j, k;
int *pi = &i;

// i加2 (i = i + 2)
*pi = *pi + 2;

// 加到pi包含的地址上
pi = pi + 2;
```

一个指针可以让它的地址值增加或减少一个整数值。这类指针操作，被称为*指针的算术运算*。这种操作初看上去并不直观，我们总认为是数据对象的加法，而不是离散的十进制数值的加法。指针加 2 意味着给指针持有的地址值增加了该类型两个对象的长度。例如，假设一个 char 是一个字节，一个 int 是 4 个字节，double 是 8 个字节，那么指针加 2 是给其持有的地址值增加 2、8、还是 16，取决于指针的类型是 char、int 还是 double。

实际上，只有指针指向数组元素时，我们才能保证较好地运用指针的算术运算。在前面的例子中，我们不能保证三个整数变量连续存储在内存中；因此，ip+2 可能、也可能不产生一个有效的地址，这取决于在该位置上实际存储的是什么。指针算术运算的典型用法是遍历一个数组。例如：

```
int ia[ 10 ];
int *iter = &ia[0];
int *iter_end = &ia[10];

while ( iter != iter_end ) {
    do_something_with_value( *iter );
    ++iter; // 现在iter指向下一个元素
}
```

练习 3.8

已知下列定义

```
int ival = 1024, ival2 = 2048;
int *pi1 = &ival, *pi2 = &ival2, **pi3 = 0;
```

说明下列赋值将产生什么后果？哪些是错误的?

```
(a) ival = *pi3;    (e) pi1 = *pi3;
(b) *pi2 = *pi3;    (f) ival = *pi1;
(c) ival = pi2;     (g) pi1 = ival;
(d) pi2 = *pi1;     (h) pi3 = &pi2;
```

练习 3.9

指针是 C 和 C++程序设计一个很重要的方面，也是程序错误的常见起源。例如：

```
pi = &ival2;
pi = pi + 1024;
```

几乎可以保证，pi 会指向内存的一个随机区域。这个赋值在做什么？什么时候它不是一个错误？

练习 3.10

类似地，下面的小程序的行为是未定义的，可能在运行时失败：

```
int foobar( int *pi ) {
    *pi = 1024;
    return *pi;
}

int main()
{
    int *pi2 = 0;
    int ival = foobar( pi2 );
    return 0;
}
```

问题出在哪里？怎样改正它？

练习 3.11

在前面两个练习中，出现错误是因为缺少在运行时刻对指针使用的检查。如果指针在 C++程序设计中起重要作用，你认为为什么没有为指针的使用增加更多的安全性？你能想到哪些指导规则能使指针的使用更加安全？

3.4 字 符 串 类 型

C++提供了两种字符串的表示：C 风格的字符串和标准 C++引入的 string 类类型。一般我们建议使用 string 类，但实际上在许多程序的情形中，我们有必要理解和使用老式的 C 风格字符串。在第 7 章我们会看到一个例子，它处理命令行选项，这些选项被作为 C 风格的字符串数组传递给 main() 函数。

3.4.1 C 风格字符串

C 风格的字符串起源于 C 语言，并在 C++中继续得到支持。（实际上，在标准 C++之前，除了第三方字符串库类之外，它是唯一一种被支持的字符串。）

字符串被存储在一个字符数组中，一般通过一个 char*类型的指针来操纵它。标准 C 库为操纵 C 风格的字符串提供了一组函数。例如：

```
// 返回字符串的长度
int strlen( const char* );
```

```
// 比较两个字符串是否相等
int strcmp( const char*, const char* );

// 把第二个字符串拷贝到第一个字符串中
char* strcpy(char*, const char* );
```

（标准 C 库作为标准的 C++的一部分被包含在其中。）为使用这些函数，我们必须包含相关的 C 头文件，

```
#include <cstring>
```

指向 C 风格字符串的字符指针总是指向一个相关联的字符数组。即使当我们写一个字符串常量时，如：

```
const char *st = "The expense of spirit\n";
```

系统在内部把字符串常量存储在一个字符串数组中。然后，st 指向该数组的第一个元素。那么，我们怎样以字符串的形式来操纵 st 呢？

一般地，我们用指针的算术运算来遍历 C 风格的字符串，每次指针增加 1，直到到达终止空字符为止。例如：

```
while ( *st++ ) { ... }
```

char*类型的指针被解除引用，并且测试指向的字符是 true 还是 false。true 值是除了空字符外的任意字符。++是增加运算符，它使指针 st 指向数组中的下一个字符。

一般来说，当我们使用一个指针时，在解除指针的引用之前，测试它是否指向某个对象是必要的。否则，程序很可能会失败。例如：

```
int
string_length( const char *st )
{
int cnt = 0;
if ( st )
     while ( *st++ )
            ++cnt;
return cnt;
}
```

C 风格字符串的长度可以为 0（因而被视为空串），有两种方式：字符指针被置为 0，因而它不指向任何对象；或者，指针已经被设置，但是它指向的数组只包含一个空字符。如

```
// pc1不指向任何一个数组对象
char *pc1 = 0;

// pc2指向空字符
const char *pc2 = "";
```

由于 C 风格字符串的底层(low-level)特性，C 或 C++的初学者很容易在这上面出错。在下面的一系列程序中，我们罗列了一些初学者易犯的错误。程序的任务很简单：计算 st 的长

度。不幸的是，第一个尝试就是错误的。你能看到问题所在吗？

```
#include <iostream>
const char *st = "The expense of spirit\n";
int main() {
    int len = 0;
    while ( st++ ) ++len;

    cout << len << ": " << st;
    return 0;
}
```

程序失败是因为 st 没有被解除引用，即

```
st++
```

测试的是 st 中的地址是否为零，而不是它指向的字符是否为空。这个条件将一直为真，因为循环的每次迭代都给 st 中的地址加 1。程序将永远执行下去或者直到系统终止它。这样的循环被称作*无限循环*(*infinite loop*)。

我们的第二个版本改正了这个错误。它能执行到完成 。不幸的是，输出的结果是错误的。你能发现我们这次犯的错误吗？

```
#include <iostream>
const char *st = "The expense of spirit\n";

int main()
{
    int len = 0;
    while ( *st++ ) ++len;

    cout << len << ": " << st << endl;
    return 0;
}
```

这次的错误是 st 已经不再指向字符串文字常量。st 已经前进到终止空字符之后的字符上去了。（程序的输出结果取决于 st 所指向的内存单元的内容。）下面是一种可能的解决办法：

```
st = st - len;
cout << len << ": " << st;
```

编译并执行程序。但是，输出仍然是不正确的。它产生如下结果：

```
22: he expense of spirit
```

这反映了程序设计某些本质的方面。你能看到这次我们犯的错误吗？

在计算字符串的长度的时候，空字符并没有被考虑在内。st 必须被重新定位到字符串长度*加* 1 的位置。下列代码是正确的：

```
st = st - len - 1;
```

编译并执行，程序最终产生正确的结果如下：

```
22: The expense of spirit
```

现在程序是正确的了，但是，从程序风格的角度来说，它还有些不太雅致。语句

```
st = st - len - 1;
```

被加进来，以便改正由直接递增 st 引起的错误。st 的赋值不符合程序的原始逻辑，而且，现在的程序有些难以理解。

像这样的程序修正通常被称作*补丁(patch)*——把某些东西伸展开以便补上现有程序中的洞。我们通过补偿原始设计中的逻辑错误来补救我们的程序。较好的解决方案是修正原始设计中的漏洞。一种方案是定义第二个指针，用 st 对它初始化。例如：

```
const char *p = st;
```

现在可以用 p 来计算 st 的长度， 而 st 不变：

```
while ( *p++ )
```

3.4.2　字符串类型

正如我们前面所看到的，因为字符指针的底层特性，用它表示字符串很容易出错。为了将程序员从许多“与使用 C 风格字符串相关的错误”中解脱出来，每个项目、部门或公司都提供了自己的字符串类 —— 实际上，本书的前两个版本就是这样做的。问题是，如果每个人都提供自己的字符串实现，那么程序的可移植性和兼容性就变得非常困难。C++标准库提供了字符串类抽象的一个公共实现。

你希望字符串类有哪些操作呢？最小的基本行为集合由什么构成呢？

1．支持用字符序列或第二个字符串对象来初始化一个字符串对象。C 风格的字符串不支持用另外一个字符串初始化一个字符串。

2．支持字符串之间的拷贝。C 风格字符串通过使用库函数 strcpy()来实现。

3．支持读写访问单个字符。对于 C 风格字符串，单个字符访问由下标操作符或直接解除指针引用来实现。

4．支持两个字符串的相等比较。对于 C 风格字符串，字符串比较通过库函数 strcmp()来实现。

5．支持两个字符串的连接：把一个字符串接到另一个字符串上，或将两个字符串组合起来形成第三个字符串。对于 C 风格的字符串，连接由库函数 strcat()来实现。把两个字符串连接起来形成第三个字符串的实现是，用 strcpy()把一个字符串拷贝到一个新实例中，然后用 strcat()把另一个字符串连接到新的实例上。

6．支持对字符串长度的查询。对于 C 风格字符串，字符串长度由库函数 strlen()返回。

7．支持字符串是否为空的判断。对于 C 风格字符串，通过下面两步条件测试来完成

```
char *str = 0;
//...
```

```
if ( ! str || ! *str )
    return;
```

标准 C++提供了支持这些操作的 string 类（在第 6 章我们会看到更多的操作）。本小节我们来看 string 类型怎样支持这些操作。

要使用 string 类型，必须先包含相关的头文件：

```
#include <string>
```

例如，下面是上一小节定义的字符数组：

```
#include <string>
string st( "The expense of spirit\n" );
```

st 的长度由 size()操作返回（不包含终止空字符）：

```
cout << "The size of "
     << st
     << " is " << st.size()
     << " characters, including the newline\n";
```

string 构造函数的第二种形式定义了一个空字符串。例如：

```
string st2; // 空字符串
```

我们怎样能保证它是空的？当然，一种办法是测试 size()是否为 0：

```
if ( ! st.size() )
    // ok: 空
```

更直接的办法是使用 empty()操作：

```
if ( st.empty() )
    // ok: 空
```

如果字符串中不含有字符，则 empty()返回布尔常量 true；否则，返回 false。

第三种形式的构造函数，用一个 string 对象来初始化另一个 string 对象。例如：

```
string st3( st );
```

将 st3 初始化成 st 的一个拷贝。怎样验证呢？等于操作符比较两个 string 对象，如果相等则返回 true：

```
if ( st == st3 )
    // 初始化成功
```

怎样拷贝一个字符串呢？最简单的办法是使用赋值操作符。例如，

```
st2 = st3; // 把st3拷贝到st2中
```

首先将与 st2 相关联的字符存储区释放掉，然后再分配足够存储与 st3 相关联的字符的

存储区，最后将与 st3 相关联的字符拷贝到该存储区中。

我们可以使用加操作符“ + ”或看起来有点怪异的复合赋值操作符“ += ”，将两个或多个字符串连接起来。例如，给出两个字符串

```
string s1( "hello, " );
string s2( "world\n" );
```

我们可以按如下方式将两个字符串连接起来形成第三个字符串：

```
string s3 = s1 + s2;
```

如果希望直接将 s2 附加在 s1 后面，那么可使用“ += ”操作符：

```
s1 += s2;
```

s1 和 s2 的初始化包含了一个空格、一个逗号、一个换行，这多少有些不方便。它们的存在限制了对这些 string 对象的重用，尽管它满足了眼前的需要。一种替代做法就是混合使用 C 风格的字符串与 string 对象，如下所示：

```
const char *pc = ", ";
string s1( "hello" );
string s2( "world" );

string s3 = s1 + pc + s2 + "\n";
```

这种连接策略比较受欢迎，因为它使 s1 和 s2 处于一种更容易被重用的形式。这种方法能够生效是由于 string 类型能够自动将 C 风格的字符串转换成 string 对象。例如，这使我们可以将一个 C 风格的字符串赋给一个 string 对象：

```
string s1;
const char *pc = "a character array";

s1 = pc; // ok
```

但是，反向的转换不能自动执行。对隐式地将 string 对象转换成 C 风格的字符串，string 类型没有提供支持。例如，下面试图用 s1 初始化 str，就会在编译时刻失败：

```
char *str = s1; // 编译时刻类型错误
```

为实现这种转换，必须显式地调用名为 c_str()的操作：

```
char *str = s1.c_str(); // 几乎是正确的，但是还差一点
```

名字 c_str()代表了 string 类型与 C 风格字符串两种表示法之间的关系。字面意思是：给我一个 C 风格的字符串表示——即，指向字符数组起始处的字符指针。

但是，这个初始化还是失败了。这次是由于另外一个不同的原因：为了防止字符数组被程序直接处理，c_str()返回了一个指向常量数组的指针（下一节将解释常量修饰符 const。）

```
const char*
```

str 被定义为非常量指针，所以这个赋值被标记为类型违例。正确的初始化如下：

```
const char *str = s1.c_str(); //ok
```

string 类型支持通过下标操作符访问单个字符。例如，在下面的代码段中，字符串中的所有句号被下划线代替：

```
string str( "fa.disney.com" );
int size = str.size();
for ( int ix = 0; ix < size; ++ix )
      if ( str[ ix ] == '.' )
          str[ ix ] = '_';
```

对 string 类型的介绍现在就讲这些，尽管我们还有许多内容要说。例如，上面代码段的实现可用如下语句替代：

```
replace( str.begin(), str.end(), '.', '_' );
```

replace()是 2.8 节中简要介绍的泛型算法中的一个（第 12 章将详细介绍泛型算法，本书附录按字母顺序给出了泛型算法及其用法的例子）。

begin()和 end()操作返回指向 string 开始和结束处的迭代器(iterator)。迭代器是指针的类抽象，由标准库提供（在 2.8 节中我们简要地介绍了迭代器，在第 6 章和第 12 章将详细介绍）。

replace()扫描 begin()和 end()之间的字符。对于每个等于句号的字符，都被替换成下划线。

练习 3.12

下列语句哪些是错误的？

```
(a) char ch = "The long, winding road";
(b) int ival = &ch;
(c) char *pc = &ival;
(d) string st( &ch );
(e) pc = 0;     (i) pc = '0';
(f) st = pc;    (j) st = &ival;
(g) ch = pc[0]; (k) ch = *pc;
(h) pc = st;    (l) *pc = ival;
```

练习 3.13

解释下面两个 while 循环的区别。

```
while ( st++ )
       ++cnt;

while ( *st++ )
       ++cnt;
```

练习 3.14

考虑下面两个语义上等价的程序，一个使用 C 风格字符串，另一个使用 string 类型。

```
// ***** C-style character string implementation *****

#include <iostream>
#include <cstring>

int main()
{
    int errors = 0;
    const char *pc = "a very long literal string";

    for ( int ix = 0; ix < 1000000; ++ix )
    {
       int len = strlen( pc );
       char *pc2 = new char[ len + 1 ];
       strcpy( pc2, pc );

       if ( strcmp( pc2, pc ))
       ++errors;

       delete [] pc2;
    }
    cout << "C-style character strings: "
        << errors << " errors occurred.\n";
}

// ***** string implementation *****

#include <iostream>
#include <string>

int main()
{
    int errors = 0;
    string str( "a very long literal string" );

    for ( int ix = 0; ix < 1000000; ++ix )
    {
       int len = str.size();
       string str2 = str;
       if ( str != str2 )
           ++errors;
    }
    cout << "string class: "
        << errors << " errors occurred.\n";
}
```

a) **说明程序完成了什么功能；**

b) **平均来说，**string **类型实现的执行速度是** C **风格字符串的两倍，在** UNIX **的** timex **命**

令下显示的执行时间如下：

```
user       0.96   # string class
user       1.98   # C-style character string
```

你是这样预想的吗？说明原因。

练习 3.15

C++的 string 类型是基于对象的类抽象的一个例子。对于本节中所介绍的关于它的用法及操作集，你有什么希望改变的吗？你认为还有哪些其他操作是必需的？有用的？请说明。

3.5　const 限定修饰符

下面的循环有两个问题，都是由于使用 512 作为循环上限引起的：

```
for ( int index = 0; index < 512; ++index )
    ... ;
```

第一个问题是可读性。用 512 来测试 index 是什么意思呢？循环在做什么呢——即 512，是什么意思？（在本例中，512 被称作*魔数(magic number)*，它的重要性在上下文中没有体现出来，就好像这个数是凭空出现的。）

第二个问题是可维护性。想像程序有 10,000 行，512 在 4%的代码中出现。在这 400 个出现中，80%必须要被改成 1024。为了做到这一点，我们必须明白哪些 512 是要被转换的，而哪些不是。即使只有一个地方弄错了，也会中断程序，要我们回头全部重新检查一遍。

这两个问题的解决方案就是使用一个被初始化为 512 的对象。通过选一个助记名，可能是 bufSize，使程序更具可读性。现在，条件测试变成与对象作比较，而不是与一个文字常量作比较：

```
index < bufSize
```

我们不需要再把 320 个出现 512 的地方一一找出来，只需改变 bufSize 的值就行了。我们只需改变 bufSize 被初始化的那一行。这种方法不仅只需要很少的工作量，而且大大减少了出错的可能性。这种方案的代价是一个额外的变量。现在 512 被称为是*局部化的(localized)*。

```
int bufSize = 512; // 缓冲区大小
// ...

for ( int index = 0; index < bufSize; ++index )
     // ...
```

这种方案的问题是，bufSize 是一个左值。在程序中 bufSize 有可能被偶然修改。例如，下面是一个常见的程序错误：

```
// 偶尔改变了bufSize的值
if ( bufSize = 1 )
```

```
    // ...
```

在 C++中，“ = ”是赋值操作符，而“ == ”是等于操作符。程序员不小心将 bufSize 的值改成 1，这导致了一个很难跟踪的错误。（这种错误很难被发现，因为程序员一般不会认为这行代码是错的，这就是为什么许多编译器会对此类的赋值表达式生成警告的原因。）

const 类型限定修饰符提供了一个解决方案。它把一个对象转换成一个*常量（constant）*。例如：

```
const int bufSize = 512; // 缓冲区大小
```

定义 bufSize 是一个常量，并将其初始化为 512。在程序中任何改变这个值的企图都将导致编译错误。因此，它被称为是*只读的*。例如：

```
// 错误：企图写入const对象
if ( bufsize = 0 ) ...
```

因为常量在定义后就不能被修改，所以它必须被初始化。未初始化的常量定义将导致编译错误。

```
const double pi; // 错误：未初始化的常量
```

一旦一个常量被定义了，我们就不能改变与 const 对象相关联的值。另一方面，我们能把它的地址赋值给一个指针吗？例如， 下面代码是否可行？

```
const double minWage = 9.60;

// ok? error?
double *ptr = &minWage;
```

这是否可行呢？minWage 是一个常量对象，因此它不能被改写为一个新的值。但是 ptr 是一个普通指针，没有什么能阻止我们写出这样的代码：

```
*ptr += 1.40; // 修改了minWage!
```

一般编译器不能跟踪指针在程序中任意一点指向的对象。（这种薄记工作要求数据流分析功能，通常由单独的*优化器(optimizer)*组件来完成。）允许非 const 对象的指针指向一个常量对象，把“ 试图通过该指针间接地改变对象值 ”的动作标记为非法的，这对编译器来说是不可行的。因而任何试图将一个非 const 对象的指针指向一个常量对象的动作都将引起编译错误。

这并不意味着我们不能间接地指向一个 const 对象，只意味着我们必须声明一个指向常量的指针来做这件事。例如：

```
const double *cptr;
```

cptr 是一个指向 double 类型的 const 对象的指针。（我们可以从右往左把这个定义读为“ cptr 是一个指向 double 类型的、被定义成 const 的对象的指针。 ”）微妙之处在于 cptr 本身不是常量。我们可以重新赋值 cptr，使其指向不同的对象，但不能修改 cptr 指向的对象。例如：

```
const double *pc = 0;
const double minWage = 9.60;

// ok: 不能通过pc修改minWage
pc = &minWage;

double dval = 3.14;

// ok: 不能通过pc修改dval
// 虽然dval本身不是一个常量
pc = &dval; // ok

dval = 3.14159; // ok
*pc = 3.14159;  // 错误
```

const 对象的地址只能赋值给指向 const 对象的指针，例如 pc。但是，指向 const 对象的指针可以被赋以一个非 const 对象的地址，例如，

```
pc = &dval;
```

虽然 dval 不是常量，但试图通过 pc 修改它的值，仍会导致编译错误（因为在运行程序的任意一点上，编译器不能确定指针所指的实际对象）。

在实际的程序中，指向 const 的指针常被用作函数的形式参数。它作为一个约定来保证：被传递给函数的实际对象在函数中不会被修改。例如：

```
// 在实际的程序中，指向常量的指针
// 往往被用作函数参数
int strcmp( const char *str1, const char *str2 );
```

（在第 7 章关于函数的讨论中我们会更多地讨论指向 const 对象的指针。）

我们可以定义一个 const 指针指向一个 const 或一个非 const 对象。例如：

```
int errNumb = 0;
int *const curErr = &errNumb;
```

curErr 是指向一个非 const 对象的 const 指针。（我们可以从右往左把定义读作“curErr 是一个指向 int 类型对象的 const 指针。”）这意味着不能赋给 curErr 其他的地址值，但可以修改 curErr 指向的值。

下面的代码说明我们可以怎样使用 curErr：

```
do_something();

if ( *curErr ) {
     errorHandler();
     *curErr = 0; // ok: 重置指针所指的对象
}
```

试图给 const 指针赋值会在编译时刻被标记为错误：

```
curErr = &myErrNumb; // 错误
```

指向 const 对象的 const 指针的定义就是将前面两种定义结合起来。例如：

```
const double pi = 3.14159;
const double *const pi_ptr = &pi;
```

在这种情况下， pi_ptr 指向的对象的值以及它的地址本身都不能被改变。（我们可以从右往左将定义读作“ pi_ptr 是指向被定义为 const 的 double 类型对象的 const 指针。”）

练习 3.16

解释下列五个定义的意思。并指出其中任何非法定义。

```
(a) int i;            (d) int *const cpi;
(b) const int ic;     (e) const int *const cpic;
(c) const int *pic;
```

练习 3.17

下列哪些初始化是合法的？为什么？

```
(a) int i = -1;
(b) const int ic = i;
(c) const int *pic = &ic;
(d) int *const cpi = &ic;
(e) const int *const cpic = &ic;
```

练习 3.18

根据上个练习的定义，下列哪些赋值是合法的？为什么？

```
(a) i = ic;        (d) pic = cpic;
(b) pic = &ic;     (e) cpic = &ic;
(c) cpi = pic;      (f) ic = *cpic;
```

3.6 引　用　类　型

引用(reference)有时候被称为*别名(alias)*，它可以用作一个对象的替代名字。引用使得我们可以间接地操纵对象，使用方式类似于指针，但是不需要用到指针的语法。在实际的程序中，引用主要被用作函数的形式参数——通常将类对象传递给一个函数。但是现在我们用独立的对象来介绍并示范引用的用法。

引用类型由类型标识符和一个取地址操作符来定义。引用必须被初始化。例如：

```
int ival = 1024;

// ok: refVal是一个指向ival的引用
```

```
int &refVal = ival;

// 错误：引用必须被初始化为指向一个对象
int &refVal2;
```

虽然引用也被用作一种指针，但是如同指针的情形那样，用一个对象的地址来初始化引用是错误的。然而，我们可以定义一个指针引用，例如：

```
int ival = 1024;

// 错误：refVal是int类型，不是int*
int &refVal = &ival;

int *pi = &ival;

// ok: refPtr是一个指向指针的引用
int *&ptrVal2 = pi;
```

一旦引用已经被定义了，它就不能再指向其他的对象（这是它为什么必须要被初始化的原因）。例如，下列的赋值不会使 refVal 指向 min_val，而是会使 refVal 指向的对象 ival 的值被设置为 min_val 的值。

```
int min_val = 0;

// ival被设置为min_val的值
// refVal并没有引用到min_val上
refVal = min_val;
```

引用的所有操作实际上都被应用在它所指的对象身上，包括取地址操作符。例如：

```
refVal += 2;
```

将 refVal 指向的对象 ival 加 2。类似地，

```
int ii = refVal;
```

把与 ival 相关联的值赋给 ii，而

```
int *pi = &refVal;
```

用 ival 的地址初始化 pi。

每个引用的定义必须以取地址操作符开始。（这与前面我们对指针的讨论是同样的问题。）例如：

```
// 定义两个int类型的对象
int ival = 1024, ival2 = 2048;

// 定义一个引用和一个对象
int &rval = ival, rval2 = ival2;
```

```
// 定义一个对象、一个指针和一个引用
int ival3 = 1024, *pi = &ival3, &ri = ival3;

// 定义两个引用
int &rval3 = ival3, &rval4 = ival2;
```

const 引用可以用不同类型的对象初始化（只要能从一种类型转换到另一种类型即可），也可以是不可寻址的值，如文字常量。例如：

```
double dval = 3.14159;

// 仅对于const引用才是合法的
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;
```

同样的初始化对于非 const 引用是不合法的，将导致编译错误。原因有些微妙，需要适当作些解释。

引用在内部维护的是一个对象的地址，它是该对象的别名。对于不可寻址的值，如文字常量，以及不同类型的对象，编译器为了实现引用，必须生成一个临时对象，引用实际上指向该对象，但用户不能访问它。例如，当我们写：

```
double dval = 1024;
const int &ri = dval;
```

编译器将其转换成:

```
int temp = dval;
const int &ri = temp;
```

如果我们给 ri 赋一个新值，则这样做不会改变 dval，而是改变 temp。对用户来说，就好像修改动作没有生效(这对于用户来说，并不总是好的)。

const 引用不会暴露这个问题，因为它们是只读的。不允许非 const 引用指向需要临时对象的对象或值，一般来说这比“允许定义这样的引用，但实际上不会生效”的方案要好得多。

下面给出的例子很难在第一次就能正确声明。我们希望用一个 const 对象的地址来初始化一个引用。非 const 引用定义是非法的，将导致编译时刻错误：

```
const int ival = 1024;

// 错误：要求一个const引用
int *&pi_ref = &ival;
```

下面是我们首先想到的修正 pi_ref 定义的做法，但是它不能生效 —— 你能看出来这是为什么吗？

```
const int ival = 1024;

// 仍然错误
const int *&pi_ref = &ival;
```

如果我们从右向左读这个定义，会发现 pi_ref 是一个指向定义为 const 的 int 型对象的指针。我们的引用不是指向一个常量，而是指向一个非常量指针，指针指向一个 const 对象。正确的定义如下：

```
const int ival = 1024;

// ok: 这是可以被编译器接受的
const int *const &pi_ref = &ival;
```

指针和引用有两个主要区别：引用必须总是指向一个对象；如果用一个引用给另一个引用赋值，那么改变的是被引用的对象而不是引用本身。我们来看几个例子。当我们这样写：

```
int *pi = 0;
```

用 0 初始化 pi——即，pi 当前不指向任何对象。但当我们写

```
const int &ri = 0;
```

时，在内部，发生了以下转换：

```
int temp = 0;
const int &ri = temp;
```

引用之间的赋值是第二个不同。当给出以下代码：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
```

我们写

```
pi = pi2;
```

pi 指向的对象 ival 并没有被改变；而是 pi 被赋值为指向 pi2 所指的对象——在本例中即 ival2。重要的是，现在 pi 和 pi2 都指向同一对象。（这是一个重要的错误源：如果我们把一个类对象拷贝给另一个类对象，而该类有一个或多个成员是指针。我们将在第 14 章详细讨论这个问题。）

但是，假定有下列代码：

```
int &ri = ival, &ri2 = ival2;
```

然后，我们写出这样的赋值语句

```
ri = ri2;
```

那么，改变的是 ival，而不是引用本身。赋值之后，两个引用仍然指向原来的对象。

实际的 C++程序很少使用指向独立对象的引用类型。引用类型主要被用作函数的形式参数，例如：

```
// 在实际的例子中，引用是如何被使用的

// return status of access. Place value in parameter
bool get_next_value( int &next_value );

// overloaded addition operator
Matrix operator+( const Matrix&, const Matrix& );
```

这些引用的用法和我们讨论的指向独立对象的引用类型有什么联系呢？在下面这样的调用中：

```
int ival;
while ( get_next_value( ival )) ...
```

实际参数（本例中为 ival）同形式参数 next_value 的绑定，等价于下面的独立对象定义：

```
int &next_value = ival;
```

（引用作为函数参数的用法将在第 7 章中详细讨论。）

练习 3.19

下列定义，哪些是无效的？为什么？怎样改正？

```
(a) int ival = 1.01;       (b) int &rval1 = 1.01;
(c) int &rval2 = ival;     (d) int &rval3 = &ival;
(e) int *pi = &ival;       (f) int &rval4 = pi;
(g) int &rval5 = *pi;      (h) int &*prval1 = pi;
(i) const int &ival2 = 1;  (j) const int &*prval2 = &ival;
```

练习 3.20

已知上面的定义，下列赋值哪些是无效的？

```
(a) rval1 = 3.14159;
(b) prval1 = prval2;
(c) prval2 = rval1;
(d) *prval2 = ival2;
```

练习 3.21

(a)中的定义有什么区别？(b)中的赋值又有什么区别？哪些是非法的？

```
(a) int ival = 0;
    const int *pi = 0;
    const int &ri = 0;

(b) pi = &ival;
```

```
    ri = &ival;
    pi = &rval;
```

3.7 布　尔　类　型

布尔型对象可以被赋以文字值 true **或** false**。例如：**

```
// 初始化一个string对象，用来存放搜索的结果
string search_word = get_word();

// 把一个bool变量初始化为false
bool found = false;

string next_word;
while ( cin >> next_word )
  if ( next_word == search_word )
        found = true;
// ...

// 缩写，相当于：if ( found == true )
if ( found )
      cout << "ok, we found the word\n";
 else cout << "nope, the word was not present.\n";
```

虽然布尔类型的对象也被看作是一种整数类型的对象，但是它不能被声明为 signed、unsigned、short **或** long**。例如，下列代码是非法的**:

```
// 错误：不能指定bool为short
short bool found = false;
```

当表达式需要一个算术值时，布尔对象(如 found)**和布尔文字都被隐式地提升成** int(**正如下面的例子**)**：**false **变成** 0**，而** true **变成** 1**。例如：**

```
bool found = false;
int occurrence_count = 0;

while ( /* mumble  */ )
{
  found = look_for( /* something */ );

  // found的值被提升为0或者1
  occurrence_count += found;
}
```

正如文字 false **和** true **能自动转换成整数值** 0 **和** 1 **一样，如果有必要，算术值和指针值也能隐式地被转换成布尔类型的值。**0 **或空指针被转换成** false**，所有其他的值都被转换成**

true。例如：

```
// returns count of occurrences
extern int find( const string& );
bool found = false;
if ( found = find( "rosebud" ))
 // ok: found == true

// returns pointer to item if present
extern int* find( int value );

if ( found = find( 1024 ))
 // ok: found == true
```

3.8 枚 举 类 型

我们在写程序的时候，常常需要定义一组与对象相关的属性。例如，一个文件可能会以三种状态(输入、输出和追加)之一被打开。

典型情况下，我们通过把每个属性和一个唯一的 const 值相关联，来记录这些状态值。因此，我们可能会这样写：

```
const int input = 1;
const int output = 2;
const int append = 3;
```

并按如下方式使用这些常量：

```
bool open_file( string file_name, int open_mode);

// ...
open_file( "Phoenix_and_the_Crane", append );
```

尽管这样做也能奏效，但是它有许多缺点。一个主要的缺点是，我们没有办法限制传递给函数的值只能是 input、output 和 append 之一。

枚举*(enumeration)*提供了一种替代的方法，它不但定义了整数常量，而且把它们组成一个集合。例如：

```
enum open_modes{ input = 1, output, append };
```

open_modes 是一个枚举类型。每个被命名的枚举定义了一个唯一的类型，它可以被用作类型标识符，例如：

```
void open_file( string file_name, open_modes om );
```

input、output 和 append 是***枚举成员****(enumerators)*。它们代表了能用来初始化和赋值 open_modes 类型变量的值的全集。例如：

```
open_file( "Phoenix and the Crane", append );
```

如果我们试图向 open_file()传递一个 input、output、append 之外的值，就会产生编译错误。而且，如果像下面这样传递一个相等的整数值，编译器仍然会将其标记为错误。

```
// 错误：1不是open_modes的枚举成员 ...
open_file( "Jonah", 1 );
```

此外，我们还可以声明枚举类型对象，如

```
open_modes om = input;
// ...
om = append;
```

并用 om 代替一个枚举成员：

```
open_file( "TailTell", om );
```

我们不能做到的是打印枚举成员的实际枚举名。当我们这样写的时候：

```
cout << input << " " << om << endl;
```

它输出：

```
1 3
```

一种解决方案是定义一个由枚举成员的值索引的字符串数组。因此，我们可以这样写：

```
cout << open_modes_table[ input ] << " "
     << open_modes_table[ om ]    << endl;
```

产生输出：

```
input append
```

第二件不能做的事情是，我们不能使用枚举成员进行迭代，如

```
// not supported
for ( open_modes iter = input; iter != append; ++iter )
      // ...
```

C++不支持在枚举成员之间的前后移动。

枚举类型用关键字 enum，加上一个自选的枚举类型名来定义，类型名后面跟一个用花括号括起来的枚举成员列表，枚举成员之间用逗号分开。在缺省情况下，第一个枚举成员被赋以值 0，后面的每个枚举成员依次比前面的大 1。在前面的例子中，赋给 input 值 1，output 值 2，append 值 3。下面的枚举成员 shape 与 0 相关，sphere 是 1，cylinder 是 2，polygon 是 3。

```
// shape == 0, sphere == 1, cylinder == 2, polygon == 3
enum Forms{ shape, sphere, cylinder, polygon };
```

我们也可以显式地把一个值赋给一个枚举成员。这个值不必是唯一的。下面的例子中，

point2d 被赋值为 2，在缺省情况下，point2w 等于 point2d 加 1 为 3，point3d 被显式地赋值为 3，point3w 在缺省情况下是 4。

```
// point2d == 2, point2w == 3, point3d == 3, point3w == 4
enum Points { point2d = 2, point2w, point3d = 3, point3w };
```

我们可以定义枚举类型的对象，它可以参与表达式运算，也可以被作为参数传递给函数。枚举类型的对象能够被初始化，但是它只能被一个相同枚举类型的对象或枚举成员集中的某个值初始化或赋值。例如，虽然 3 是一个与 Points 相关联的合法值，但是它不能被显式地赋给一个 Points 类型的对象：

```
void mumble() {
    Points pt3d = point3d; // ok: pt3d == 3

    // 错误：pt2w被初始化为一个int整数
    Points pt2w = 3;

    // 错误：polygon不是Points的枚举成员
    pt2w = polygon;

    // ok: pt2w和pt3d都是Points枚举类型
    pt2w = pt3d;
}
```

但是，在必要时，枚举类型会自动被提升成算术类型。例如：

```
const int array_size = 1024;

// ok: pt2w被提升成int类型
int chunk_size = array_size * pt2w;
```

3.9 数　组　类　型

正如我们在 2.1 节中所看到的，数组是一个单一数据类型对象的集合。其中单个对象并没有被命名 ，但是我们可以通过它在数组中的位置对它进行访问。这种访问形式被称作*索引访问(indexing)*或*下标访问(subscripting)*。例如：

```
int ival;
```

声明了一个 int 型对象。而

```
int ia[ 10 ];
```

声明了一个包含 10 个 int 对象的数组。每个对象被称作是 ia 的一个*元素*。因此

```
ival = ia[ 2 ];
```

将 ia 中由 2 索引的元素的值赋给 ival。类似地，

```
ia[ 7 ] = ival;
```

把 ival 的值赋给 ia 的由 7 索引的元素。

数组定义由一个类型名、一个标识符和一个维数组成。维数指定数组中包含的元素的数目，它被写在一对方括号里边。我们必须为数组指定一个大于等于 1 的维数。维数值必须是常量表达式——即，必须能在编译时刻计算出它的值。这意味着一个非 const 的变量不能被用来指定数组的维数。下面的例子包含合法的和非法的数组定义：

```
extern int get_size();

// buf_size和max_files都是const
const int buf_size = 512, max_files = 20;
int staff_size = 27;

// ok: const变量
char input_buffer[ buf_size ];

// ok：常量表达式：20 - 3
char *fileTable[ max_files - 3 ];

// 错误：非const变量
double salaries[ staff_size ];

// 错误：非const表达式
int test_scores[ get_size() ];
```

虽然 staff_size 被一个文字常量初始化，但是 staff_size 本身是一个非 const 对象。系统只能在运行时刻访问它的值，因此，它作为数组维数是非法的。另一方面，表达式

```
max_files - 3
```

是常量表达式，因为 max_files 是用 20 作初始值的 const 变量。这个表达式在编译时刻被计算成 17。

正如我们在 2.1 节所看到的，数组元素是从 0 开始计数的。对一个包含 10 个元素的数组，正确的索引值是从 0 到 9，而不是从 1 到 10。下面的例子中，一个 for 循环遍历数组的 10 个元素，并用它们的索引值作初始值：

```
int main()
{
    const int array_size = 10;
    int ia[ array_size ];

    for ( int ix = 0; ix < array_size; ++ix )
         ia[ ix ] = ix;
}
```

数组可以被显式地用一组数初始化，这组数用逗号分开，被写在大括号中。例如：

```
const int array_size = 3;
int ia[ array_size ] = { 0, 1, 2 };
```

被显式初始化的数组不需要指定维数值。编译器会根据列出来的元素的个数来确定数组的维数：

```
// 维数为3的数组
int ia[] = { 0, 1, 2 };
```

如果指定了维数，那么初始化列表提供的元素的个数不能超过这个值。否则，将导致编译错误。如果指定的维数大于给出的元素的个数，那么没有被显式初始化的元素将被置为 0。

```
// ia ==> { 0, 1, 2, 0, 0 }
const int array_size = 5;
int ia[ array_size ] = { 0, 1, 2 };
```

字符数组可以用一个由逗号分开的字符文字列表初始化，文字列表用花括号括起来，或者用一个字符串文字初始化。但是，注意这两种形式不是等价的，字符串常量包含一个额外的终止空字符。例如：

```
const char ca1[] = { 'C', '+', '+' };
const char ca2[] = "C++";
```

ca1 的维数是 3，ca2 的维数是 4。下面的声明将被标记为错误：

```
// 错误："Daniel"是7个元素
const char ch3[ 6 ] = "Daniel";
```

一个数组不能被另外一个数组初始化，也不能被赋值给另外一个数组。而且，C++不允许声明一个引用数组(即由引用组成的数组)。

```
const int array_size = 3;
int ix, jx, kx;

// ok: 类型为int*的指针的数组
int *iap [] = { &ix, &jx, &kx };

// 错误：不允许引用数组
int &iar[] = { ix, jx, kx };

// 错误：不能用另一个数组来初始化一个数组
int ia2[] = ia; // 错误

int main()
{
    int ia3[ array_size ]; // ok
```

```
    // 错误：不能把一个数组赋给另一个数组
    ia3 = ia;
    return 0;
}
```

要把一个数组拷贝到另一个中去，必须按顺序拷贝每个元素。例如：

```
const int array_size = 7;
int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };

int main()
{
    int ia2[ array_size ];

    for ( int ix = 0; ix < array_size; ++ix )
         ia2[ ix ] = ia1[ ix ];

    return 0;
}
```

任意一个结果为整数值的表达式都可以用来索引数组。例如：

```
int someVal, get_index();
ia2[ get_index() ] = someVal;
```

但是用户必须清楚，C++没有提供编译时刻或运行时刻对数组下标的范围检查。除了程序员自己注意细节，并彻底地测试自己的程序之外，没有别的办法可防止数组越界。能够通过编译并执行的程序仍然存在致命的错误，这不是不可能的。

练习 3.22

下面哪些数组定义是非法的？为什么？

```
int get_size();
int buf_size = 1024;
(a) int ia[ buf_size ];     (d) int ia[ 2 * 7 - 14 ];
(b) int ia[ get_size() ];   (e) char st[ 11 ] = "fundamental";
(c) int ia[ 4 * 7 - 14 ];
```

练习 3.23

下面代码试图用数组中每个元素的索引值来初始化该元素。它包含一些索引错误。请把它们指出来

```
int main() {
    const int array_size = 10;
    int ia[ array_size ];

    for ( int ix = 1; ix <= array_size; ++ix )
         ia[ ix ] = ix;
```

```
    // ...
}
```

3.9.1　多维数组

我们也可以定义多维数组。每一维用一个方括号对来指定，例如：

```
int ia[ 4 ][ 3 ];
```

定义了一个二维数组。第一维被称作*行(row)*维，第二维称作*列(column)*维。ia 是一个二维数组，它有 4 行，每行 3 个元素。多维数组也可以被初始化。

```
int ia[ 4 ][ 3 ] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
};
```

用来指示行的花括号，即被内嵌在里边的花括号是可选的。下面的初始化与前面的是等价的，只是有点不清楚。

```
int ia[4][3] = { 0,1,2,3,4,5,6,7,8,9,10,11 };
```

下面的定义只初始化了每行的第一个元素。其余的元素被初始化为 0。

```
int ia[ 4 ][ 3 ] = { {0}, {3}, {6}, {9} };
```

如果省略了花括号，结果会完全不同。下面的定义

```
int ia[ 4 ][ 3 ] = { 0, 3, 6, 9 };
```

初始化了第一行的 3 个元素和第二行的第一个元素，其余元素都被初始化为 0。为了索引到一个多维数组中，每一维都需要一个方括号对。例如，下面的一对嵌套 for 循环初始化了一个二维数组。

```
int main()
{
    const int rowSize = 4;
    const int colSize = 3;
    int ia[ rowSize ][ colSize ];

    for ( int i = 0; i < rowSize; ++i )
          for ( int j = 0; j < colSize; ++j )
               ia[ i ][ j ] = i + j;
}
```

虽然表达式

```
ia[ 1, 2 ]
```

在 C++中是合法的结构，但它的意思可能不是程序员所希望的：ia[1,2]等价于 ia[2]，因为“1,2”是一个逗号表达式，它的结果是一个单值 2。（逗号表达式将在 4.10 节中讨论）。这将访问 ia 的第三行的第一个元素。程序员希望的可能是 ia[1][2]。

在 C++中，多维数组的索引访问要求对程序员希望访问的每个索引都有一对方括号。

3.9.2　数组与指针类型的关系

已知下面的数组定义

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

那么，只简单写

```
ia;
```

意味着什么呢？

数组标识符代表数组中第一个元素的地址。它的类型是数组元素类型的指针。在 ia 这个例子中，它的类型是 int*。因此，下面两种形式是等价的，它们都返回数组的第一个元素的地址。

```
ia;
&ia[0];
```

类似地，为了访问相应的值，我们可以取下列两种方式之一：

```
// 两者都得到第一个元素的值
*ia;
ia[0];
```

我们知道怎样用下标操作符来访问第二个元素的地址：

```
&ia[1];
```

同样，下面这个表达式

```
ia+1;
```

也能得到第二个元素的地址，等等。类似地，下面两个表达式都可以访问第二个元素的值：

```
*(ia+1);
ia[1];
```

但是，如下的表达式

```
*ia+1;
```

完全不同于下面的表达式

```
*(ia+1);
```

解引用操作符比加法运算符的优先级高（我们将在 4.13 节中讨论优先级），所以它先被计算。解引用 ia 将返回数组的第一个元素的值。然后对其加 1。如果在表达式里加上括号，那么 ia 将先被加 1，然后解引用新的地址值。对 ia 加 1 将使 ia 增加其元素类型的大小，ia+1 指向数组中的下一个元素。

数组元素遍历则可以通过下标操作符来实现，到目前为止我们一直这样做，或者我们也可以通过直接操作指针来实现数组元素遍历。例如：

```
#include <iostream>
int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    int *pbegin = ia;
    int *pend = ia + 9;

    while ( pbegin != pend ) {
           cout << *pbegin << ' ';
          ++pbegin;
    }
}
```

pbegin 被初始化指向数组的第一个元素。在 while 循环的每次迭代中它都被递增以指向数组的下一个元素。最难的是判断何时停止。在本例中，我们将 pend 初始化指向数组最末元素的下一个地址。当 pbegin 等于 pend 时，我们知道已经迭代了整个数组。

如果我们把这一对指向数组头和最末元素下一位置的指针，抽取到一个独立的函数中，那么，现在我们就有了一个能够迭代整个数组的工具，却无须知道数组的实际大小（当然，调用函数的程序员必须知道）。例如：

```
#include <iostream>

void ia_print( int *pbegin, int *pend )
{
     while ( pbegin != pend ) {
        cout << *pbegin << ' ';
       ++pbegin;
     }
}

int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    ia_print( ia, ia + 9 );

}
```

当然，这是有限制的：它只支持指向整型数组的指针。我们可以通过把 ia_print()转换成模板函数来消除这个限制（在 2.5 节我们简要地介绍了模板）。例如：

```
#include <iostream>

template <class elemType>
void print( elemType *pbegin, elemType *pend )
{
      while ( pbegin != pend ) {
        cout << *pbegin << ' ';
       ++pbegin;
      }
}
```

现在我们可以给通用的函数 print()传递一对指向任意类型数组的指针，只要该类型的输出操作符已经被定义即可，例如：

```
int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    double da[4] = { 3.14, 6.28, 12.56, 25.12 };
    string sa[3] = { "piglet", "eeyore", "pooh" };

    print( ia, ia+9 );
    print( da, da+4 );
    print( sa, sa+3 );
}
```

这种程序设计形式被称为*泛型程序设计*(*generic programming*)，标准库提供了一组泛型算法（我们在 2.8 节和 3.4 节结束的时候简要地介绍了这些算法），它们通过一对标记元素范围的开始/结束指针来遍历其中的元素。例如，我们可以如下调用泛型算法 sort()：

```
#include <algorithm>
int main()
{
    int ia[6] = { 107, 28, 3, 47, 104, 76 };
    string sa[3] = { "piglet", "eeyore", "pooh" };

   sort( ia, ia+6 );
    sort( sa, sa+3 );
}
```

我们将在第 12 章详细讨论泛型算法。本书附录以字母顺序给出这些算法以及用法示例。

更一般化的是，标准库提供了一组类，它们封装了容器和指针的抽象。在 2.8 节我们已经对其进行了简要的介绍。在下一节中，我们将讨论 vector 容器类型，它为内置数组提供了一个基于对象的替代品。

3.10　vector 容器类型

vector 类为内置数组提供了一种替代表示（在 2.8 节中我们简要介绍了 vector），通常我们建议使用 vector。（但是仍然有许多程序环境必须使用内置数组，例如处理命令行选项——我们将在 7.8 节中可以看到）。与 string 类一样，vector 类是随标准 C++引入的标准库的一部分。

为了使用 vector，我们必须包含相关的头文件：

```
#include <vector>
```

使用 vector 有两种不同的形式，即所谓的*数组习惯*和 *STL 习惯*。在数组习惯用法中，我们模仿内置数组的用法：我们定义一个已知长度的 vector。

```
vector< int > ivec( 10 );
```

这与如下定义一个包含十个元素的内置数组相似：

```
int ia[ 10 ];
```

我们可以用下标操作符访问 vector 的元素，与访问内置数组的元素的方式一样。例如：

```
void simple_example()
{
   const int elem_size = 10;
   vector< int > ivec( elem_size );
   int ia[ elem_size ];

   for ( int ix = 0; ix < elem_size; ++ix )
         ia[ ix ] = ivec[ ix ];

  // ...
}
```

我们可以用 size()查询 vector 的大小，也可以用 empty()测试它是否为空。例如：

```
void print_vector( vector<int> ivec )
{
  if ( ivec.empty() )
     return;

  for ( int ix = 0; ix < ivec.size(); ++ix )
        cout << ivec[ ix ] << ' ';
}
```

vector 的元素被初始化为与其类型相关的缺省值。算术和指针类型的缺省值是 0。对于 class 类型，缺省值可通过调用该类的缺省构造函数获得（关于缺省构造函数的介绍见 2.3 节）。我们还可以为每个元素提供一个显式的初始值来完成初始化。例如：

```
vector< int > ivec( 10, -1 );
```

定义了 ivec，它包含十个 int 型的元素，每个元素都被初始化为-1。

对于内置数组，我们可以显式地把数组的元素初始化为一组常量值。例如：

```
int ia[ 6 ] = { -2, -1, 0, 1, 2, 1024 };
```

我们不能用同样的方法显式地初始化 vector。但是，我们可以将 vector 初始化为一个已有数组的全部或一部分，只需指定希望被用来初始化 vector 的数组的开始地址以及数组最末元素的下一位置来实现。例如：

```
// 把ia的6个元素拷贝到ivec中
vector< int > ivec( ia, ia+6 );
```

被传递给 ivec 的两个指针标记了用来初始化对象的值的范围。第二个指针总是指向要被拷贝的末元素的下一位置。标记出来的元素范围也可以是数组的一个子集。例如：

```
// 拷贝3个元素：ia[2], ia[3], ia[4]
vector< int > ivec( &ia[ 2 ], &ia[ 5 ] );
```

与内置数组不同，一个 vector 可以被另一个 vector 初始化，或被赋给另一个 vector。例如：

```
vector< string > svec;

void init_and_assign()
{
    // 用另一个vector初始化一个vector
    vector< string > user_names( svec );
    // ...

    // 把一个vector拷贝给另一个vector
    svec = user_names;
}
```

在 STL 习惯[2]中，vector 的用法完全不同。我们不是定义一个已知大小的 vector，而是定义一个空 vector：

```
vector< string > text;
```

我们向 vector 中插入元素，而不再是索引元素，以及向元素赋值。例如，push_back()操作，就是在 vector 的后面插入一个元素。下面的 while 循环从标准输入读入一个字符串序列，并每次将一个字符串插入到 vector 中：

```
string word;
while ( cin >> word ) {
```

[2] STL 表示标准模板库（Standard Template Library）。在被纳入到标准 C++中之前，vector 与泛型算法是独立库 STL 的一部分（见[MUSSER96]）。

```
        text.push_back( word );
        // ...
    }
```

虽然我们仍可以用下标操作符来迭代访问元素：

```
    cout << "words read are: \n";
    for ( int ix = 0; ix < text.size(); ++ix )
          cout << text[ ix ] << ' ';
    cout << endl;
```

但是，更典型的做法是使用 vector 操作集中的 begin()和 end()返回的迭代器(iterator)对：

```
    cout << "words read are: \n";
    for ( vector<string>::iterator it = text.begin();
          it != text.end(); ++it )
             cout << *it << ' ';
    cout << endl;
```

iterator 是标准库中的类，它具有指针的功能。

```
    *it;
```

对迭代器解引用，并访问其指向的实际对象。

```
    ++it;
```

向前移动迭代器 it，使其指向下一个元素。（在第 6 章，我们将非常详细地讨论 iterator、vector 和一般的 STL 习惯用法）。

注意不要混用这两种习惯用法。例如，下面的定义

```
    vector<int> ivec;
```

定义了一个空 vector，再写这样的语句

```
    ivec[0] = 1024;
```

是错误的，因为 ivec 还没有第一个元素。我们只能索引 vector 中已经存在的元素。size()操作返回 vector 包含的元素的个数。

类似地，当我们用一个给定的大小定义一个 vector 时，例如

```
    vector<int> ia( 10 );
```

任何一个插入操作都将增加 vector 的大小，而不是覆盖掉某个现有的元素。这看起来好像是很显然的，但是，下面的错误在初学者中并不少见：

```
    const int size = 7;
    int ia[ size ] = { 0, 1, 1, 2, 3, 5, 8 };
    vector< int > ivec( size );
    for ( int ix = 0; ix < size; ++ix )
          ivec.push_back( ia[ ix ]);
```

程序结束时 ivec 包含 14 个元素，ia 的元素从第八个元素开始插入。

另外，在 STL 习惯用法下，vector 的一个或多个元素可以被删除。（我们将在第 6 章讨论。）

练习 3.24

下列 vector 定义，哪些是错误的？

```
int ia[ 7 ] = { 0, 1, 1, 2, 3, 5, 8 };

(a) vector< vector< int > > ivec;
(b) vector< int >    ivec = { 0, 1, 1, 2, 3, 5, 8 };
(c) vector< int >    ivec( ia, ia+7 );
(d) vector< string > svec = ivec;
(e) vector< string > svec( 10, string( "null" ));
```

练习 3.25

已知下面的函数声明

```
bool is_equal( const int*ia, int ia_size,
               const vector<int> &ivec );
```

请实现下列行为：如果两个容器大小不同，则比较相同大小部分的元素。一旦某个元素不相等，则返回 false。如果所有元素都相等，则返回 true。请用 iterator 迭代访问 vector ——可以以本节中的例子为模型。并且写一个 main()函数来测试 is_equal()函数。

3.11　复　数　类　型

复数（complex number）类是标准库的一部分。为了要使用它，我们必须包含其相关的头文件：

```
#include <complex>
```

每个复数都有两部分：实数部分和虚数部分。虚数代表负数的平方根。这个术语是由笛卡儿首创的。复数的一般表示法如下：

```
2 + 3i
```

这里 2 代表实数部分，而 3i 表示虚数部分。这两部分合起来表示单个复数。

复数对象的定义一般可以使用以下形式：

```
// 纯虚数：0 + 7i
complex< double > purei( 0, 7 );

// 虚数部分缺省为0：3 + 0i
complex< float > real_num( 3 );

// 实部和虚部均缺省为0：0 + 0i
complex< long double > zero;
```

```
// 用另一个复数对象来初始化一个复数对象
complex< double > purei2( purei );
```

这里，复数对象有 float、double 或 long double 几种表示。我们也可以声明复数对象的数组：

```
complex< double > conjugate[ 2 ] = {
        complex< double >( 2, 3 ),
         complex< double >( 2, -3 )
};
```

我们也可以声明指针或引用：

```
complex< double > *ptr = &conjugate[0];
complex< double > &ref = *ptr;
```

复数支持加、减、乘、除和相等比较。另外，它也支持对实部和虚部的访问。这些操作将在 4.6 节中详细介绍。

3.12　Typedef 名字

typedef 机制是一种通用的类型定义设施，通过它我们可以为内置的或用户定义的数据类型引入助记符号。例如：

```
typedef double          wages;
typedef vector<int>      vec_int;
typedef vec_int         test_scores;
typedef bool       in_attendance;
typedef int        *Pint;
```

这些 typedef 名字在程序中可被用作类型标识符：

```
// double hourly, weekly;
wages hourly, weekly;

// vector<int> vec1( 10 );
vec_int vec1( 10 );

// vector<int> test0( class_size );
const int class_size = 34;
test_scores test0( class_size );

// vector< bool > attendance;
vector< in_attendance > attendance( class_size );
```

```
// int *table[ 10 ];
Pint table[ 10 ];
```

typedef 定义以关键字 typedef 开始，后面是数据类型和标识符。这里的标识符即 typedef 名字，它并没有引入一种新的类型，而只是为现有类型引入了一个助记符号。typedef 名字可以出现在任何类型名能够出现的地方。

typedef 名字可以被用作程序文档的辅助说明。它也能够降低声明的复杂度。例如，在典型情况下，typedef 名字可以用来增强“复杂模板声明的定义”的可读性（见 3.14 节例子），增强“指向函数的指针”（将在 7.9 节中讨论）以及“指向类的成员函数的指针”（将在 13.6 节中讨论）的可读性。

下面是一个几乎所有人刚开始时都会答错的问题。错误在于将 typedef 当作宏扩展。已知下面的 typedef

```
typedef char *cstring;
```

在以下声明中，cstr 的类型是什么？

```
extern const cstring cstr;
```

第一个回答差不多都是：

```
const char *cstr
```

即指向 const 字符的指针。但是，这是不正确的。const 修饰 cstr 的类型。cstr 是一个指针，因此，这个定义声明了 cstr 是一个指向字符的 const 指针（见 3.5 节关于 const 指针类型的讨论。）：

```
char *const cstr;
```

3.13　volatile 限定修饰符

当一个对象的值可能会在编译器的控制或监测之外被改变时，例如一个被系统时钟更新的变量，那么该对象应该被声明成 volatile。因此，编译器执行的某些例行优化行为不能应用在被程序员指定为 volatile 的对象上。

volatile 限定修饰符的用法同 const 非常相似 —— 都是作为类型的附加修饰符。例如：

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[ max_size ];
volatile Screen bitmap_buf;
```

display_register 是一个 int 型的 volatile 对象。curr_task 是一个指向 volatile 的 Task 类对象的指针。ixa 是一个 volatile 的整型数组。数组的每个元素被认为是 volatile 的。bitmap_buf 是一个 volatile 的 Screen 类对象，它的每个数据成员都被视为 volatile 的。

volatile 修饰符的主要目的是提示编译器，该对象的值可能在编译器未监测到的情况下被改变。因此编译器不能武断地对引用这些对象的代码作优化处理。

3.14　pair 类型

pair 类也是标准库的一部分，它使得我们可以在单个对象内部把相同类型或不同类型的两个值关联起来。为了使用 pair 类，我们必须包含下面的头文件：

```
#include <utility>
```

例如：

```
pair< string, string > author( "James", "Joyce" );
```

创建了一个 pair 对象 author，它包含两个字符串，分别被初始化为"James"和"Joyce"。

我们可以用*成员访问符号(member access notation)*访问 pair 中的单个元素，它们的名字为 first 和 second。例如：

```
string firstBook;

if ( author.first == "James" &&
     author.second == "Joyce" )
     firstBook = "Stephen Hero";
```

如果我们希望定义大量相同 pair 类型的对象，那么最方便的做法就是用 typedef，如下：

```
typedef pair< string, string > Authors;

Authors proust( "marcel", "proust" );
Authors joyce( "james", "joyce" );
Authors musil( "robert", "musil" );
```

下面是第二个 pair。一个元素持有对象的名字，另一个元素持有指向其符号表入口的指针：

```
// 前向声明(forward declaration)
class EntrySlot;
extern EntrySlot* look_up( string );

typedef pair< string, EntrySlot* > SymbolEntry;

SymbolEntry current_entry( "author", look_up( "author" ));
// ...

if ( EntrySlot *it = look_up( "editor" ))
{
     current_entry.first = "editor";
     current_entry.second = it;
```

```
}
```

我们将在第 6 章讨论标准库容器类型、以及第 12 章讨论标准库泛型算法的时候，再次看到 pair 类型。

3.15　类（class）类型

类机制支持新类型的设计，如本章讨论的基于对象的 string、vector、complex、pair 类型，以及第 1 章介绍的面向对象的 iostream 类层次结构。在第 2 章中，我们通过一个 Array 类抽象的实现和进化过程，将支持面向对象的与基于对象的类设计的基本概念和机制快速浏览了一遍。在本节中，我们将简要地介绍一个简单的基于对象的 String 类抽象的设计与实现。它将得益于我们前面给出的、对 C 风格字符串以及标准库 string 类型的讨论。这个实现将着重说明 C++对*操作符重载（operator overloading）*的支持，2.3 节曾简单介绍过这方面的知识。（从第 13 章到第 15 章将详细介绍类。我们在本书的开始部分先介绍类的某些方面，是为了使我们能够在本书 13 章之前就可以提供一些更有意义的、并且用到了类的例子。初次阅读本书的读者可跳过本节，在对后面章节有了更多的了解后，再回头来看。）

现在我们对 String 类应该做些什么已经很清楚：我们需要支持 String 对象的初始化和赋值，包括用字符串文字、C 风格字符串、以及另外一个 String 对象进行初始化或者赋值，我们将通过特定的构造函数以及*类特定的*[3]赋值操作符实例来实现这样的功能。

我们需要支持用索引访问 String 中的单个字符，以便与 C 风格字符串和标准库 string 类型具有相同的方式。我们将提供一个类特定的下标操作符实例来做到这一点。

另外，我们还想支持这样一些操作，如确定 String 长度的 size()、两个 String 对象的相等比较，或者 String 同 C 风格字符串的比较、读写一个 String 对象等等。我们将提供等于、iostream 输入、iostream 输出操作符的实例，以实现后两个操作。最后，我们也需要访问底层的 C 风格字符串。

类的定义由关键字 class 开始，后面是一个标识符，该标识符也被用作类的类型指示符，如 complex、vector、Array 等等。一般地，一个类包括公有的(public)操作部分和私有的(private)数据部分。这些操作被称为该类的*成员函数(member functions)*或*方法(methods)*，它们定义了类的*公有接口(public interface)*——即，用户可以在该类对象上执行的操作的集合。我们的 String 类的私有数据包括：_string，一个指向动态分配的字符数组的 char*类型的指针；和 _size，记录 String 中字符串长度的 int 型变量。下面是我们的定义：

```
#include <iostream>

class String;
istream& operator>>( istream&, String& );
ostream& operator<<( ostream&, const String& );
```

[3] 译注：这里的“类特定的”，即 class-specific，是指相应的操作符属于 String 这个类，也就是说与 String 相关联，而不是系统全局缺省的操作符实例。

```
class String {
public:
// 一组重载的构造函数
// 提供自动初始化功能
// String str1;             // String()
// String str2( "literal" ); // String( const  char* );
// String str3( str2 );      // String( const String& );

String();
String( const char* );
String( const String& );

// 析构函数：自动析构
~String();

// 一组重载的赋值操作符
// str1 = str2
// str3 = "a string literal"

String& operator=( const String& );
String& operator=( const char* );

// 一组重载的等于操作符
// str1 == str2;
// str3 == “a string literal”;

bool operator==( const String& );
bool operator==( const char* );

// 重载的下标操作符
// str1[ 0 ] = str2[ 0 ];

char& operator[]( int );

// 成员访问函数
int   size()  { return _size;   }
char* c_str() { return _string; }

private:
int   _size;
char *_string;

};
```

String 类定义了三个构造函数。正如在 2.3 节中简要讨论的那样，重载函数机制允许同一函数名或操作符引用到多个实例，只要通过参数表能区分开每个实例就行。我们的三个构造函数形成了一个有效的重载函数集合，首先由参数个数，然后由参数类型来区分它们。第一个构造函数

```
String();
```

被称做*缺省构造函数*，因为它不需要任何显式的初始值。当我们这样写时：

```
String str1;
```

缺省构造函数将被应用到 str1 上。

另外两个 String 构造函数都有一个参数。当我们写

```
String str2( "a string literal" );
```

时，根据参数类型，构造函数

```
String( const char* );
```

被应用在 str2 上。类似地，当我们写

```
String str3( str2 );
```

时，构造函数

```
String( const String& );
```

被应用在 str3 上 —— 这是根据被传递给构造函数的参数类型来判断的。这种构造函数被称为*拷贝构造函数(copy constructor)*，因为它用另一个对象的拷贝来初始化一个对象。当我们写：

```
String str4( 1024 );
```

时，实参的类型与构造函数集期望的参数类型都不匹配，因此，str4 的定义导致一个编译错误。

被重载的操作符采用下面的一般形式

```
return_type operator op ( parameter_list );
```

这里 operator 是关键字，op 是一个预定义的操作符，如“+”、“=”、“==”、“[]”，等等（第 15 章有确切的规则）。下面的声明

```
char& operator[]( int );
```

声明了一个下标操作符的重载实例，它带有一个 int 型的参数，返回指向 char 的引用。重载的操作符还可以被重载，只要每个实例的参数表能够被区分开即可。例如，我们为 String 类提供了两个不同的赋值与等于操作符的实例。

有名字的成员函数可以通过成员访问符号来调用。例如，已知下列 String 定义：

```
String object( "Danny" );
String *ptr = new String( "Anna" );
String array[2];
```

我们可以如下调用成员函数 size()，它们分别返回长度值 5、4 和 0（一会儿我们会看到 String 类的实现）。

```
vector<int> sizes( 3 );

// 针对对象的点成员访问符号 .
```

```
// object has a size of 5
sizes[ 0 ] = object.size();

// 针对指针的箭头成员访问符号->
// ptr has a size of 4
sizes[ 1 ] = ptr->size();

// 再次使用点成员访问符号
// array[0] has a size of 0
sizes[ 2 ] = array[0].size();
```

被重载的操作符也可以直接应用在类对象上。例如：

```
String name1( "Yadie" );
String name2( "Yodie" );

// 应用：bool operator==(const String&)
if ( name1 == name2 )
    return;
else
// 应用：String& operator=( const String& )
    name1 = name2;
```

一个类的成员函数可以被定义在类的定义中，也可以定义在外面。（例如，size()和c_str()，都是在 String 类的定义中被定义的。）在类定义之外定义的成员函数不但要告诉编译器它们的名字、返回类型、参数表，而且还要说明它们所属的类。我们应该把成员函数的定义放到一个程序文本文件中——例如，String.C——并且把含有该类定义的头文件(本例中为 String.h)包含进来。例如：

```
// this is placed in a program text file: String.C

// 包含String类的定义
#include "String.h"

// 包含strcmp()函数的声明
// cstring是标准C库的头文件
#include <cstring>
bool              // 返回类型
String::              // 说明这是String类的一个成员
operator==             // 函数的名字：等于操作符
(const String &rhs)   // 参数列表
{
    if ( _size != rhs._size )
        return false;
    return strcmp( _string, rhs._string ) ? false : true;
}
```

strcmp()是 C 标准库函数。它比较两个 C 风格的字符串。如果相等则返回 0，否则返回非 0。*条件操作符* (?:)测试问号前面的条件，如果为 true，选择问号与冒号之间的表达式，

如果为 false，选择冒号后面的表达式。在本例中，如果 strcmp()返回非 0 值，条件操作符返回 false，否则返回 true。（4.7 节将详细讨论条件操作符。）

因为等于操作符是个可能被频繁调用的小函数，因此把它声明成内联(inline)函数是个好办法。内联函数在每个调用点上被展开，因此，这样做可以消除函数调用相关的额外消耗。只要该函数被调用足够多次（7.6 节将详细介绍内联函数），内联函数就能够显著地提高性能。在类定义内部定义的成员函数，如 size()，在缺省情况下被设置为 inline。在类外面定义的成员函数必须显式地声明为 inline：

```
inline bool
String::operator==(const String &rhs)
{
    // 如前
}
```

在类体外定义的内联成员函数，应该被包含在含有该类定义的头文件中。我们在重新定义了等于操作符之后，应当把它的定义从 String.C 移到 String.h 中。

下面是比较 String 对象和 C 风格字符串的等于操作符（它也被定义成内联函数，因而被放在 String.h 头文件中）。

```
inline bool
String::operator==(const char *s)
{
    return strcmp( _string, s ) ? false : true;
}
```

构造函数的名字与类名相同。我们不能在它的声明或构造函数体中指定返回值。它的一个或多个实例都可以被声明成 inline。

```
#include <cstring>

// 缺省构造函数
inline String::String()
{
    _size = 0;
    _string = 0;
}

inline String::String( const char *str )
{
      if ( ! str ) {
          _size = 0; _string = 0;
  }
  else {
  _size = strlen( str );
  _string = new char[ _size + 1 ];
  strcpy( _string, str );
```

```
  }
}

// 拷贝构造函数
inline String::String( const String &rhs )
{
     _size = rhs._size;
     if ( ! rhs._string )
          _string = 0;
     else {
          _string = new char[ _size + 1 ];
          strcpy( _string, rhs._string );
     }
}
```

因为我们用 new 表达式动态地分配内存来保留字符串，所以当不再需要该字符串对象的时候，我们必须用 delete 表达式释放该内存区。这可以通过定义类的析构函数自动实现，把 delete 表达式放在构析函数中。如果类的析构函数存在，那么在每个类的生命期结束时它会被自动调用（第 8 章将解释一个对象的三种可能的生命期）。析构函数由类名前面加一个波浪号（~）来标识。下面是 String 类的析构函数的定义：

```
inline String::~String() { delete [] _string; }
```

两个被重载的赋值操作符引用了一个特殊的关键字 this。当我们写

```
String name1( "orville" ), name2( "wilbur" );
name1 = "Orville Wright";
```

在赋值操作符中，this 指向 name1。

更一般的情况，在类成员函数中 this 指针被自动设置为指向左侧的类对象（我们通过该对象调用这个成员函数）。当我们写

```
ptr->size();
obj[ 1024 ];
```

在 size()中，this 指针指向 ptr；在下标操作符中，this 指针指向 obj。当我们写*this 时，我们在访问 this 所指的实际对象（13.4 节将详细讨论 this 指针）。

```
inline String&
String::operator=( const char *s )
{
    if ( ! s ) {
   _size = 0;
   delete [] _string;
   _string = 0;
    }
    else {
   _size = strlen( s );
```

```
    delete [] _string;
    _string = new char[ _size + 1 ];
    strcpy( _string, s );
     }
     return *this;
}
```

当我们把一个类对象拷贝给另一个时，最常犯的错误是忘了先测试这两个类对象实际上是否是同一个对象。这个错误最典型的发生时机是：当一个或两个对象都是通过解除一个指针的引用而来的。此时，this **指针将再次发挥作用，以支持这种测试。例如：**

```
inline String&
String::operator=( const String &rhs )
{
    // 在表达式name1 = *pointer_to_string中，
    // this指向name1,
    // rhs代表*pointer_to_string.
if ( this != &rhs ) {
```

下面是完整的实现：

```
inline String&
String::operator=( const String &rhs )
{
if ( this != &rhs )
{
    delete [] _string;
    _size = rhs._size;

    if ( ! rhs._string )
_string = 0;
    else {
_string = new char[ _size + 1 ];
strcpy( _string, rhs._string );
    }
}
return *this;
}
```

下标操作符几乎与 2.3 **节中** Array **类的实现相同：**

```
#include <cassert>

inline char&
String::operator[]( int elem )
{
    assert( elem >= 0 && elem < _size );
    return _string[ elem ];
```

```
}
```

输入操作符和输出操作符是作为非成员函数实现的（原因将在 15.2 节中讨论。20.4 节与 20.5 节将对重载 iostream 输入和输出操作符进行详细讨论。）我们的输入操作符最多读入 4095 个字符。setw()是一个预定义的 iostream 操纵符。它读入的字符数最多为传递给它的参数减 1。因此，我们可以保证不会溢出 inBuf 字符数组。为了要使用它，我们必须包含 iomanip 头文件。（第 20 章将详细讨论 setw()。）

```
#include <iomanip>

inline istream&
operator>>( istream &io, String &s )
{
    // 人工限制最多4096个字符
    const int limit_string_size = 4096;
    char inBuf[ limit_string_size ];

    // setw()是iostream库的一部分
    // 限制被读取的字符个数为limit_string_size-1
    io >> setw( limit_string_size ) >> inBuf;

    s = inBuf; // String::operator=( const char* );
    return io;
}
```

为了显示 String，输出操作符需要访问内部的 char*表示。但是，因为它不是类的成员函数，所以它没有访问_string 的权限。有两种可能的解决方案：一种是给输出操作符赋予一个特殊的访问许可（把它声明成类的*友元(friend)* —— 我们将在 15.2 节中看到）；第二种方法是提供一个内联的访问函数——在本例中为 c_str()，这是以标准库 string 类提供的解决方案为模型的。下面是实现：

```
inline ostream&
operator<<( ostream& os, String &s )
{
    return os << s.c_str();
}
```

下面的小程序练习了 String 类的实现。它从标准输入读入一个 String 序列，然后顺序访问 String，并记录出现的元音字母。

```
#include <iostream>
#include "String.h"
int main()
{
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0,
        theCnt = 0, itCnt = 0, wdCnt = 0, notVowel = 0;
```

```
    // 为了使用operator==( const char* )
    // 我们并不定义The( "The" )和It( "It" )
    String buf, the( "the" ), it( "it" );

    // 调用operator>>( istream&, String& )
    while ( cin >> buf ) {
       ++wdCnt;

       // 调用operator<<( ostream&, const String& )
       cout << buf << ' ';

       if ( wdCnt % 12 == 0 )
           cout << endl;

       // 调用String::operator==(const String&) and
       //        String::operator==( const char* );
       if ( buf == the || buf == "The" )
              ++theCnt;
       else
       if ( buf == it || buf == "It" )
              ++itCnt;

       // 调用String::size()
       for ( int ix = 0; ix < buf.size(); ++ix )
       {
          // 调用String::operator[](int)
          switch( buf[ ix ] )
          {
                 case 'a': case 'A': ++aCnt; break;
                 case 'e': case 'E': ++eCnt; break;
                 case 'i': case 'I': ++iCnt; break;
                 case 'o': case 'O': ++oCnt; break;
                 case 'u': case 'U': ++uCnt; break;
                 default: ++notVowel; break;
          }
       }
    }

    // 调用operator<<( ostream&, const String& )
    cout << "\n\n"
         << "Words read: " << wdCnt  << "\n\n"
         << "the/The: "    << theCnt << '\n'
         << "it/It: "      << itCnt  << "\n\n"
         << "non-vowels read: " << notVowel << "\n\n"
         << "a: " << aCnt << '\n'
         << "e: " << eCnt << '\n'
         << "i: " << iCnt << '\n'
         << "o: " << oCnt << '\n'
         << "u: " << uCnt << endl;
}
```

程序的输入是 Stan 写的儿童故事中的一段话（在第 6 章我们会再次看到）。编译并执行程序，它产生如下输出：

```
Alice Emma has long flowing red hair. Her Daddy says when the
wind blows through her hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he tells her, magical but
untamed. "Daddy, shush, there is no such thing," she tells him, at
the same time wanting him to tell her more. Shyly, she asks,
"I mean, Daddy, is there?"

Words read: 65

the/The: 2
it/It: 1

non-vowels read: 190

a: 22
e: 30
i: 24
o: 10
u: 7
```

练习 3.26

在 String 类的构造函数和赋值操作符的实现中，有大量的重复代码。请使用 2.3 节中展示的模型，把这些公共代码抽取成独立的私有成员函数。用它们重新实现构造函数和赋值操作符，并重新运行程序以确保其正确。

练习 3.27

修改程序，使其也能够记下辅音字母 b、d、f、s 和 t 的个数。

练习 3.28

实现能够返回 String 中某个字符出现次数的成员函数。声明如下：

```
class String {
public:
    // ...
    int count( char ch ) const;
    // ...
};
```

练习 3.29

实现一个成员操作符函数，它能把一个 String 与另一个连接起来，并返回一个新的 String。声明如下：

```
class String {
public:
    // ...
    String operator+( const 2String &rhs ) const;
    // ...
};
```