

练习 2.17: 执行下面的代码段将输出什么结果？

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

2.3.2 指针

指针（pointer）是“指向（point to）”另外一种类型的复合类型。与引用类似，指针也实现了对其他对象的间接访问。然而指针与引用相比又有很多不同点。其一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。其二，指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。



WARNING 指针通常难以理解，即使是有经验的程序员也常常因为调试指针引发的错误而被备受折磨。

定义指针类型的方法将声明符写成`*d`的形式，其中`d`是变量名。如果在一条语句中定义了几个指针变量，每个变量前面都必须有符号`*`：

```
int *ip1, *ip2; // ip1 和 ip2 都是指向 int 型对象的指针
double dp, *dp2; // dp2 是指向 double 型对象的指针, dp 是 double 型对象
```

获取对象的地址

指针存放某个对象的地址，要想获取该地址，需要使用取地址符（操作符`&`）：

```
int ival = 42;
int *p = &ival; // p 存放变量 ival 的地址，或者说 p 是指向变量 ival 的指针
```

第二条语句把`p`定义为一个指向`int`的指针，随后初始化`p`令其指向名为`ival`的`int`对象。因为引用不是对象，没有实际地址，所以不能定义指向引用的指针。

除了 2.4.2 节（第 56 页）和 15.2.3 节（第 534 页）将要介绍的两种例外情况，其他所有指针的类型都要和它所指向的对象严格匹配：

```
double dval;
double *pd = &dval; // 正确：初始值是 double 型对象的地址
double *pd2 = pd; // 正确：初始值是指向 double 对象的指针

int *pi = pd; // 错误：指针 pi 的类型和 pd 的类型不匹配
pi = &dval; // 错误：试图把 double 型对象的地址赋给 int 型指针
```

因为在声明语句中指针的类型实际上被用于指定它所指向对象的类型，所以二者必须匹配。如果指针指向了一个其他类型的对象，对该对象的操作将发生错误。

指针值

指针的值（即地址）应属下列 4 种状态之一：

1. 指向一个对象。
2. 指向紧邻对象所占空间的下一个位置。
3. 空指针，意味着指针没有指向任何对象。
4. 无效指针，也就是上述情况之外的其他值。



试图拷贝或以其他方式访问无效指针的值都将引发错误。编译器并不负责检查此类错误，

53 ◀ 这一点和试图使用未经初始化的变量是一样的。访问无效指针的后果无法预计，因此程序员必须清楚任意给定的指针是否有效。

尽管第2种和第3种形式的指针是有效的，但其使用同样受到限制。显然这些指针没有指向任何具体对象，所以试图访问此类指针（假定的）对象的行为不被允许。如果这样做了，后果也无法预计。

利用指针访问对象

如果指针指向了一个对象，则允许使用解引用符（操作符*）来访问该对象：

```
int ival = 42;
int *p = &ival; // p存放着变量ival的地址，或者说p是指向变量ival的指针
cout << *p; // 由符号*得到指针p所指向的对象，输出42
```

对指针解引用会得出所指的对象，因此如果给解引用的结果赋值，实际上也就是给指针所指的对象赋值：

```
*p = 0; // 由符号*得到指针p所指向的对象，即可经由p为变量ival赋值
cout << *p; // 输出0
```

如上述程序所示，为*p赋值实际上是为p所指向的对象赋值。



解引用操作仅适用于那些确实指向了某个对象的有效指针。

关键概念：某些符号有多重含义

像&和*这样的符号，既能用作表达式里的运算符，也能作为声明的一部分出现，符号的上下文决定了符号的意义：

```
int i = 42;
int &r = i; // &紧随类型名出现，因此是声明的一部分，r是一个引用
int *p; // *紧随类型名出现，因此是声明的一部分，p是一个指针
p = &i; // &出现在表达式中，是一个取地址符
*p = i; // *出现在表达式中，是一个解引用符
int &r2 = *p; // &是声明的一部分，*是一个解引用符
```

在声明语句中，&和*用于组成复合类型；在表达式中，它们的角色又转变成运算符。在不同场景下出现的虽然是同一个符号，但是由于含义截然不同，所以我们完全可以把它当作不同的符号来看待。

空指针

空指针（null pointer）不指向任何对象，在试图使用一个指针之前代码可以首先检查它是否为空。以下列出几个生成空指针的方法：

54 ◀

```
int *p1 = nullptr; // 等价于int *p1 = 0;
int *p2 = 0; // 直接将p2初始化为字面常量0
// 需要首先#include <cstdlib>
int *p3 = NULL; // 等价于int *p3 = 0;
```



得到空指针最直接的办法就是用字面值**nullptr**来初始化指针，这也是C++11新标准刚刚引入的一种方法。**nullptr**是一种特殊类型的字面值，它可以被转换成（参见2.1.2节，

第 32 页) 任意其他的指针类型。另一种办法就如对 p2 的定义一样, 也可以通过将指针初始化为字面值 0 来生成空指针。

过去的程序还会用到一个名为 NULL 的预处理变量 (preprocessor variable) 来给指针赋值, 这个变量在头文件 `cstdlib` 中定义, 它的值就是 0。

2.6.3 节 (第 68 页) 将稍微介绍一点关于预处理器的知识, 现在只要知道预处理器是运行于编译过程之前的一段程序就可以了。预处理变量不属于命名空间 `std`, 它由预处理器负责管理, 因此我们可以直接使用预处理变量而无须在前面加上 `std::`。

当用到一个预处理变量时, 预处理器会自动地将它替换为实际值, 因此用 NULL 初始化指针和用 0 初始化指针是一样的。在新标准下, 现在的 C++ 程序最好使用 `nullptr`, 同时尽量避免使用 NULL。

把 int 变量直接赋给指针是错误的操作, 即使 int 变量的值恰好等于 0 也不行。

```
int zero = 0;  
pi = zero;           // 错误: 不能把 int 变量直接赋给指针
```

建议: 初始化所有指针

使用未经初始化的指针是引发运行时错误的一大原因。

和其他变量一样, 访问未经初始化的指针所引发的后果也是无法预计的。通常这一行为将造成程序崩溃, 而且一旦崩溃, 要想定位到出错位置将是特别棘手的问题。

在大多数编译器环境下, 如果使用了未经初始化的指针, 则该指针所占内存空间的当前内容将被看作一个地址值。访问该指针, 相当于去访问一个本不存在的位置上的本不存在的对象。糟糕的是, 如果指针所占内存空间中恰好有内容, 而这些内容又被当作了某个地址, 我们就很难分清它到底是合法的还是非法的了。

因此建议初始化所有的指针, 并且在可能的情况下, 尽量等定义了对象之后再定义指向它的指针。如果实在不清楚指针应该指向何处, 就把它初始化为 `nullptr` 或者 0, 这样程序就能检测并知道它没有指向任何具体的对象了。

赋值和指针

指针和引用都能提供对其他对象的间接访问, 然而在具体实现细节上二者有很大不同, 其中最重要的一点就是引用本身并非一个对象。一旦定义了引用, 就无法令其再绑定到另外的对象, 之后每次使用这个引用都是访问它最初绑定的那个对象。

指针和它存放的地址之间就没有这种限制了。和其他任何变量 (只要不是引用) 一样, 给指针赋值就是令它存放一个新的地址, 从而指向一个新的对象:

```
int i = 42;  
int *pi = 0;           // pi 被初始化, 但没有指向任何对象  
int *pi2 = &i;         // pi2 被初始化, 存有 i 的地址  
int *pi3;             // 如果 pi3 定义于块内, 则 pi3 的值是无法确定的  
  
pi3 = pi2;            // pi3 和 pi2 指向同一个对象 i  
pi2 = 0;              // 现在 pi2 不指向任何对象了
```

有时候要想搞清楚一条赋值语句到底是改变了指针的值还是改变了指针所指对象的值不太容易, 最好的办法就是记住赋值永远改变的是等号左侧的对象。当写出如下语句时,

```
pi = &ival;           // pi 的值被改变, 现在 pi 指向了 ival
```

意思是为 `pi` 赋一个新的值，也就是改变了那个存放在 `pi` 内的地址值。相反的，如果写出如下语句，

```
*pi = 0; // ival 的值被改变，指针 pi 并没有改变
```

则 `*pi`（也就是指针 `pi` 指向的那个对象）发生改变。

其他指针操作

只要指针拥有一个合法值，就能将它用在条件表达式中。和采用算术值作为条件（参见 2.1.2 节，第 32 页）遵循的规则类似，如果指针的值是 0，条件取 `false`：

```
int ival = 1024;
int *pi = 0; // pi 合法，是一个空指针
int *pi2 = &ival; // pi2 是一个合法的指针，存放着 ival 的地址
if (pi) // pi 的值是 0，因此条件的值是 false
    ...
if (pi2) // pi2 指向 ival，因此它的值不是 0，条件的值是 true
    ...
// ...
```

任何非 0 指针对应的条件值都是 `true`。

对于两个类型相同的合法指针，可以用相等操作符（`==`）或不相等操作符（`!=`）来比较它们，比较的结果是布尔类型。如果两个指针存放的地址值相同，则它们相等；反之它们不相等。这里两个指针存放的地址值相同（两个指针相等）有三种可能：它们都为空、都指向同一个对象，或者都指向了同一个对象的下一地址。需要注意的是，一个指针指向某对象，同时另一个指针指向另外对象的下一地址，此时也有可能出现这两个指针值相同的情况，即指针相等。

因为上述操作要用到指针的值，所以不论是作为条件出现还是参与比较运算，都必须使用合法指针，使用非法指针作为条件或进行比较都会引发不可预计的后果。

3.5.3 节（第 105 页）将介绍更多关于指针的操作。

56 void* 指针

`void*` 是一种特殊的指针类型，可用于存放任意对象的地址。一个 `void*` 指针存放着一个地址，这一点和其他指针类似。不同的是，我们对该地址中到底是个什么类型的对象并不了解：

```
double obj = 3.14, *pd = &obj; // 正确：void*能存放任意类型对象的地址
void *pv = &obj; // obj 可以是任意类型的对象
pv = pd; // pv 可以存放任意类型的指针
```

利用 `void*` 指针能做的事情儿比较有限：拿它和别的指针比较、作为函数的输入或输出，或者赋给另外一个 `void*` 指针。不能直接操作 `void*` 指针所指的对象，因为我们并不知道这个对象到底是什么类型，也就无法确定能在这个对象上做哪些操作。

概括说来，以 `void*` 的视角来看内存空间也就仅仅是内存空间，没办法访问内存空间中所存的对象，关于这点将在 19.1.1 节（第 726 页）有更详细的介绍，4.11.3 节（第 144 页）将讲述获取 `void*` 指针所存地址的方法。

2.3.2 节练习

练习 2.18: 编写代码分别更改指针的值以及指针所指对象的值。

练习 2.19: 说明指针和引用的主要区别。

练习 2.20: 请叙述下面这段代码的作用。

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

练习 2.21: 请解释下述定义。在这些定义中有非法的吗？如果有，为什么？

```
int i = 0;
(a) double* dp = &i; (b) int *ip = i; (c) int *p = &i;
```

练习 2.22: 假设 p 是一个 int 型指针，请说明下述代码的含义。

```
if (p) // ...
if (*p) // ...
```

练习 2.23: 给定指针 p，你能知道它是否指向了一个合法的对象吗？如果能，叙述判断的思路；如果不能，也请说明原因。

练习 2.24: 在下面这段代码中为什么 p 合法而 lp 非法？

```
int i = 42;           void *p = &i;           long *lp = &i;
```

2.3.3 理解复合类型的声明



如前所述，变量的定义包括一个基本数据类型（base type）和一组声明符。在同一条定义语句中，虽然基本数据类型只有一个，但是声明符的形式却可以不同。也就是说，一条定义语句可能定义出不同类型的变量：

```
// i 是一个 int 型的数，p 是一个 int 型指针，r 是一个 int 型引用
int i = 1024, *p = &i, &r = i;
```



很多程序员容易迷惑于基本数据类型和类型修饰符的关系，其实后者不过是声明符的一部分罢了。

57

定义多个变量



经常有一种观点会误以为，在定义语句中，类型修饰符（*或&）作用于本次定义的全部变量。造成这种错误看法的原因有很多，其中之一是我们可以把空格写在类型修饰符和变量名中间：

```
int* p;           // 合法但是容易产生误导
```

我们说这种写法可能产生误导是因为 int* 放在一起好像是这条语句中所有变量共同的类型一样。其实恰恰相反，基本数据类型是 int 而非 int*。* 仅仅是修饰了 p 而已，对该声明语句中的其他变量，它并不产生任何作用：

```
int* p1, p2;     // p1 是指向 int 的指针，p2 是 int
```

涉及指针或引用的声明，一般有两种写法。第一种把修饰符和变量标识符写在一起：

```
int *p1, *p2; // p1 和 p2 都是指向 int 的指针
```

这种形式着重强调变量具有的复合类型。第二种把修饰符和类型名写在一起，并且每条语句只定义一个变量：

```
int* p1;           // p1 是指向 int 的指针
int* p2;           // p2 是指向 int 的指针
```

这种形式着重强调本次声明定义了一种复合类型。



上述两种定义指针或引用的不同方法没有孰对孰错之分，关键是选择并坚持其中的一种写法，不要总是变来变去。

本书采用第一种写法，将*（或是&）与变量名连在一起。

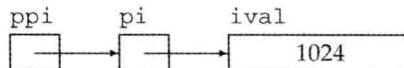
指向指针的指针

一般来说，声明符中修饰符的个数并没有限制。当有多个修饰符连写在一起时，按照其逻辑关系详加解释即可。以指针为例，指针是内存中的对象，像其他对象一样也有自己的地址，因此允许把指针的地址再存放到另一个指针当中。

通过*的个数可以区分指针的级别。也就是说，**表示指向指针的指针，***表示指向指针的指针的指针，以此类推：

```
int ival = 1024;
int *pi = &ival; // pi 指向一个 int 型的数
int **ppi = &pi; // ppi 指向一个 int 型的指针
```

此处 pi 是指向 int 型数的指针，而 ppi 是指向 int 型指针的指针，下图描述了它们之间的关系。



解引用 int 型指针会得到一个 int 型的数，同样，解引用指向指针的指针会得到一个指针。此时为了访问最原始的那个对象，需要对指针的指针做两次解引用：

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

该程序使用三种不同的方式输出了变量 ival 的值：第一种直接输出；第二种通过 int 型指针 pi 输出；第三种两次解引用 ppi，取得 ival 的值。

指向指针的引用

引用本身不是一个对象，因此不能定义指向引用的指针。但指针是对象，所以存在对指针的引用：

```
int i = 42;
int *p;           // p 是一个 int 型指针
int *&r = p;      // r 是一个对指针 p 的引用

r = &i;           // r 引用了一个指针，因此给 r 赋值&i 就是令 p 指向 i
*r = 0;           // 解引用 r 得到 i，也就是 p 指向的对象，将 i 的值改为 0
```

要理解 `r` 的类型到底是什么，最简单的办法是从右向左阅读 `r` 的定义。离变量名最近的符号（此例中是`&r` 的符号`&`）对变量的类型有最直接的影响，因此 `r` 是一个引用。声明符的其余部分用以确定 `r` 引用的类型是什么，此例中的符号`*`说明 `r` 引用的是一个指针。最后，声明的基本数据类型部分指出 `r` 引用的是一个 `int` 指针。



面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

2.3.3 节练习

59

练习 2.25：说明下列变量的类型和值。

(a) `int* ip, i, &r = i;` (b) `int i, *ip = 0;` (c) `int* ip, ip2;`

2.4 const 限定符



有时我们希望定义这样一种变量，它的值不能被改变。例如，用一个变量来表示缓冲区的大小。使用变量的好处是当我们觉得缓冲区大小不再合适时，很容易对其进行调整。另一方面，也应随时警惕防止程序一不小心改变了这个值。为了满足这一要求，可以用关键字 `const` 对变量的类型加以限定：

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

这样就把 `bufSize` 定义成了一个常量。任何试图为 `bufSize` 赋值的行为都将引发错误：

```
bufSize = 512;                  // 错误：试图向 const 对象写值
```

因为 `const` 对象一旦创建后其值就不能再改变，所以 `const` 对象必须初始化。一如既往，初始值可以是任意复杂的表达式：

```
const int i = get_size();         // 正确：运行时初始化
const int j = 42;                // 正确：编译时初始化
const int k;                     // 错误：k 是一个未经初始化的常量
```

初始化和 `const`

正如之前反复提到的，对象的类型决定了其上的操作。与非 `const` 类型所能参与的操作相比，`const` 类型的对象能完成其中大部分，但也不是所有的操作都适合。主要的限制就是只能在 `const` 类型的对象上执行不改变其内容的操作。例如，`const int` 和普通的 `int` 一样都能参与算术运算，也都能转换成一个布尔值，等等。

在不改变 `const` 对象的操作中还有一种是初始化，如果利用一个对象去初始化另外一个对象，则它们是不是 `const` 都无关紧要：

```
int i = 42;
const int ci = i;                // 正确：i 的值被拷贝给了 ci
int j = ci;                     // 正确：ci 的值被拷贝给了 j
```

尽管 `ci` 是整型常量，但无论如何 `ci` 中的值还是一个整型数。`ci` 的常量特征仅仅在执行改变 `ci` 的操作时才会发挥作用。当用 `ci` 去初始化 `j` 时，根本无须在意 `ci` 是不是一个常量。拷贝一个对象的值并不会改变它，一旦拷贝完成，新的对象就和原来的对象没什么关系了。

60 ➤ 默认状态下，const 对象仅在文件内有效

当以编译时初始化的方式定义一个 const 对象时，就如对 bufSize 的定义一样：

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

编译器将在编译过程中把用到该变量的地方都替换成对应的值。也就是说，编译器会找到代码中所有用到 bufSize 的地方，然后用 512 替换。

为了执行上述替换，编译器必须知道变量的初始值。如果程序包含多个文件，则每个用了 const 对象的文件都必须得能访问到它的初始值才行。要做到这一点，就必须在每一个用到变量的文件中都有对它的定义（参见 2.2.2 节，第 41 页）。为了支持这一用法，同时避免对同一变量的重复定义，默认情况下，const 对象被设定为仅在文件内有效。当多个文件中出现了同名的 const 变量时，其实等同于在不同文件中分别定义了独立的变量。

某些时候有这样一种 const 变量，它的初始值不是一个常量表达式，但又确实有必要在文件间共享。这种情况下，我们不希望编译器为每个文件分别生成独立的变量。相反，我们想让这类 const 对象像其他（非常量）对象一样工作，也就是说，只在一个文件中定义 const，而在其他多个文件中声明并使用它。

解决的办法是，对于 const 变量不管是声明还是定义都添加 extern 关键字，这样只需定义一次就可以了：

```
// file_1.cc 定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
// file_1.h 头文件
extern const int bufSize; // 与 file_1.cc 中定义的 bufSize 是同一个
```

如上述程序所示，file_1.cc 定义并初始化了 bufSize。因为这条语句包含了初始值，所以它（显然）是一次定义。然而，因为 bufSize 是一个常量，必须用 extern 加以限定使其被其他文件使用。

file_1.h 头文件中的声明也由 extern 做了限定，其作用是指明 bufSize 并非本文件所独有，它的定义将在别处出现。



i 如果想在多个文件之间共享 const 对象，必须在变量的定义之前添加 extern 关键字。

2.4 节练习

练习 2.26：下面哪些句子是合法的？如果有不合法的句子，请说明为什么？

- | | |
|-------------------------|------------------|
| (a) const int buf; | (b) int cnt = 0; |
| (c) const int sz = cnt; | (d) ++cnt; ++sz; |



2.4.1 const 的引用

可以把引用绑定到 const 对象上，就像绑定到其他对象上一样，我们称之为**对常量的引用**（reference to const）。与普通引用不同的是，对常量的引用不能被用作修改它所绑定的对象：

```
const int ci = 1024;
const int &rl = ci; // 正确：引用及其对应的对象都是常量
```

```
r1 = 42;           // 错误: r1 是对常量的引用
int &r2 = ci;      // 错误: 试图让一个非常量引用指向一个常量对象
```

因为不允许直接为 `ci` 赋值，当然也就不能通过引用去改变 `ci`。因此，对 `r2` 的初始化是错误的。假设该初始化合法，则可以通过 `r2` 来改变它引用对象的值，这显然是不正确的。

术语：常量引用是对 `const` 的引用

C++程序员们经常把词组“对 `const` 的引用”简称为“常量引用”，这一简称还是挺靠谱的，不过前提是你得时刻记得这就是个简称而已。

严格来说，并不存在常量引用。因为引用不是一个对象，所以我们没法让引用本身恒定不变。事实上，由于 C++语言并不允许随意改变引用所绑定的对象，所以从这层意义上理解所有的引用又都算是常量。引用的对象是常量还是非常量可以决定其所能参与的操作，却无论如何都不会影响到引用和对象的绑定关系本身。

初始化和对 `const` 的引用

2.3.1 节（第 46 页）提到，引用的类型必须与其所引用对象的类型一致，但是有两个例外。第一种例外情况就是在初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能转换成（参见 2.1.2 节，第 32 页）引用的类型即可。尤其，允许为一个常量引用绑定非常量的对象、字面值，甚至是个一般表达式：

```
int i = 42; .
const int &r1 = i;      // 允许将 const int& 绑定到一个普通 int 对象上
const int &r2 = 42;      // 正确: r1 是一个常量引用
const int &r3 = r1 * 2;  // 正确: r3 是一个常量引用
int &r4 = r1 * 2;       // 错误: r4 是一个普通的非常量引用
```

要想理解这种例外情况的原因，最简单的办法是弄清楚当一个常量引用被绑定到另外一种类型上时到底发生了什么：

```
double dval = 3.14;
const int &ri = dval;
```

此处 `ri` 引用了一个 `int` 型的数。对 `ri` 的操作应该是整数运算，但 `dval` 却是一个双精度浮点数而非整数。因此为了确保让 `ri` 绑定一个整数，编译器把上述代码变成了如下形式：

```
const int temp = dval;    // 由双精度浮点数生成一个临时的整型常量
const int &ri = temp;     // 让 ri 绑定这个临时量
```

62

在这种情况下，`ri` 绑定了一个临时量（temporary）对象。所谓临时量对象就是当编译器需要一个空间来暂存表达式的求值结果时临时创建的一个未命名的对象。C++程序员们常常把临时量对象简称为临时量。

接下来探讨当 `ri` 不是常量时，如果执行了类似于上面的初始化过程将带来什么样的后果。如果 `ri` 不是常量，就允许对 `ri` 赋值，这样就会改变 `ri` 所引用对象的值。注意，此时绑定的对象是一个临时量而非 `dval`。程序员既然让 `ri` 引用 `dval`，就肯定想通过 `ri` 改变 `dval` 的值，否则为什么要给 `ri` 赋值呢？如此看来，既然大家基本上不会想着把引用绑定到临时量上，C++语言也就把这种行为归为非法。

对 const 的引用可能引用一个并非 const 的对象

必须认识到，常量引用仅对引用可参与的操作做出了限定，对于引用的对象本身是不是一个常量未作限定。因为对象也可能是个非常量，所以允许通过其他途径改变它的值：

```
int i = 42;
int &r1 = i;                      // 引用 r1 绑定对象 i
const int &r2 = i;                // r2 也绑定对象 i，但是不允许通过 r2 修改 i 的值
r1 = 0;                          // r1 并非常量，i 的值修改为 0
r2 = 0;                          // 错误：r2 是一个常量引用
```

r2 绑定（非常量）整数 i 是合法的行为。然而，不允许通过 r2 修改 i 的值。尽管如此，i 的值仍然允许通过其他途径修改，既可以直接给 i 赋值，也可以通过像 r1 一样绑定到 i 的其他引用来修改。



2.4.2 指针和 const

与引用一样，也可以令指针指向常量或非常量。类似于常量引用（参见 2.4.1 节，第 54 页），指向常量的指针（pointer to const）不能用于改变其所指对象的值。要想存放常量对象的地址，只能使用指向常量的指针：

```
const double pi = 3.14;           // pi 是个常量，它的值不能改变
double *ptr = &pi;               // 错误：ptr 是一个普通指针
const double *cptr = &pi;         // 正确：cptr 可以指向一个双精度常量
*cptr = 42;                      // 错误：不能给*cptr 赋值
```

2.3.2 节（第 47 页）提到，指针的类型必须与其所指对象的类型一致，但是有两个例外。第一种例外情况是允许令一个指向常量的指针指向一个非常量对象：

```
double dval = 3.14;              // dval 是一个双精度浮点数，它的值可以改变
cptr = &dval;                   // 正确：但是不能通过 cptr 改变 dval 的值
```

63

和常量引用一样，指向常量的指针也没有规定其所指的对象必须是一个常量。所谓指向常量的指针仅仅要求不能通过该指针改变对象的值，而没有规定那个对象的值不能通过其他途径改变。



试试这样想吧： 所谓指向常量的指针或引用，不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变所指对象的值。

const 指针

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本身定为常量。常量指针（const pointer）必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能再改变了。把*放在 const 关键字之前用以说明指针是一个常量，这样的书写形式隐含着一层意味，即不变的是指针本身的值而非指向的那个值：

```
int errNumb = 0;
int *const curErr = &errNumb;    // curErr 将一直指向 errNumb
const double pi = 3.14159;
const double *const pip = &pi;    // pip 是一个指向常量对象的常量指针
```

如同 2.3.3 节（第 52 页）所讲的，要想弄清楚这些声明的含义最行之有效的办法是从右向左阅读。此例中，离 curErr 最近的符号是 const，意味着 curErr 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符中的下一个符号是*，意思是 curErr

是一个常量指针。最后，该声明语句的基本数据类型部分确定了常量指针指向的是一个 int 对象。与之相似，我们也能推断出，`pip` 是一个常量指针，它指向的对象是一个双精度浮点型常量。

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指对象的类型。例如，`pip` 是一个指向常量的常量指针，则不论是 `pip` 所指的对象值还是 `pip` 自己存储的那个地址都不能改变。相反的，`curErr` 指向的是一个一般的非常量整数，那么就完全可以用 `curErr` 去修改 `errNumb` 的值：

```
*pip = 2.72;           // 错误：pip 是一个指向常量的指针
                      // 如果 curErr 所指的对象（也就是 errNumb）的值不为 0
if (*curErr) {
    errorHandler();
*curErr = 0;          // 正确：把 curErr 所指的对象的值重置
}
```

2.4.2 节练习

练习 2.27：下面的哪些初始化是合法的？请说明原因。

- | | |
|------------------------------------------------|-------------------------------------------------|
| (a) <code>int i = -1, &r = 0;</code> | (b) <code>int *const p2 = &i2;</code> |
| (c) <code>const int i = -1, &r = 0;</code> | (d) <code>const int *const p3 = &i2;</code> |
| (e) <code>const int *p1 = &i2;</code> | (f) <code>const int &const r2;</code> |
| (g) <code>const int i2 = i, &r = i;</code> | |

练习 2.28：说明下面的这些定义是什么意思，挑出其中不合法的。

- | | |
|---------------------------------------------|---------------------------------------|
| (a) <code>int i, *const cp;</code> | (b) <code>int *p1, *const p2;</code> |
| (c) <code>const int ic, &r = ic;</code> | (d) <code>const int *const p3;</code> |
| (e) <code>const int *p;</code> | |

练习 2.29：假设已有上一个练习中定义的那些变量，则下面的哪些语句是合法的？请说明原因。

- | | |
|--------------------------------|--------------------------------|
| (a) <code>i = ic;</code> | (b) <code>p1 = p3;</code> |
| (c) <code>p1 = &ic;</code> | (d) <code>p3 = &ic;</code> |
| (e) <code>p2 = p1;</code> | (f) <code>ic = *p3;</code> |

2.4.3 顶层 const



如前所述，指针本身是一个对象，它又可以指向另外一个对象。因此，指针本身是不是常量以及指针所指的是不是一个常量就是两个相互独立的问题。用名词**顶层 const** (top-level const) 表示指针本身是个常量，而用名词**底层 const** (low-level const) 表示指针所指的对象是一个常量。

更一般的，顶层 `const` 可以表示任意的对象是常量，这一点对任何数据类型都适用，如算术类型、类、指针等。底层 `const` 则与指针和引用等复合类型的基本类型部分有关。比较特殊的是，指针类型既可以是顶层 `const` 也可以是底层 `const`，这一点和其他类型相比区别明显：

```
int i = 0;
int *const p1 = &i;           // 不能改变 p1 的值，这是一个顶层 const
const int ci = 42;            // 不能改变 ci 的值，这是一个顶层 const
const int *p2 = &ci;          // 允许改变 p2 的值，这是一个底层 const
```

```
const int *const p3 = p2; // 靠右的 const 是顶层 const, 靠左的是底层 const
const int &r = ci; // 用于声明引用的 const 都是底层 const
```

 当执行对象的拷贝操作时, 常量是顶层 const 还是底层 const 区别明显。其中, 顶层 const 不受什么影响:

```
i = ci; // 正确: 拷贝 ci 的值, ci 是一个顶层 const, 对此操作无影响
p2 = p3; // 正确: p2 和 p3 指向的对象类型相同, p3 顶层 const 的部分不影响
```

执行拷贝操作并不会改变被拷贝对象的值, 因此, 拷入和拷出的对象是否是常量都没什么影响。

另一方面, 底层 const 的限制却不能忽视。当执行对象的拷贝操作时, 拷入和拷出的对象必须具有相同的底层 const 资格, 或者两个对象的数据类型必须能够转换。一般来说, 非常量可以转换成常量, 反之则不行:

65 >	int *p = p3; // 错误: p3 包含底层 const 的定义, 而 p 没有
	p2 = p3; // 正确: p2 和 p3 都是底层 const
	p2 = &i; // 正确: int* 能转换成 const int*
	int &r = ci; // 错误: 普通的 int& 不能绑定到 int 常量上
	const int &r2 = i; // 正确: const int& 可以绑定到一个普通 int 上

p3 既是顶层 const 也是底层 const, 拷贝 p3 时可以不在乎它是一个顶层 const, 但是必须清楚它指向的对象得是一个常量。因此, 不能用 p3 去初始化 p, 因为 p 指向的是一个普通的(非常量)整数。另一方面, p3 的值可以赋给 p2, 是因为这两个指针都是底层 const, 尽管 p3 同时也是一个常量指针(顶层 const), 仅就这次赋值而言不会有什么影响。

2.4.3 节练习

练习 2.30: 对于下面的这些语句, 请说明对象被声明成了顶层 const 还是底层 const?

```
const int v2 = 0; int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

练习 2.31: 假设已有上一个练习中所做的那些声明, 则下面的哪些语句是合法的? 请说明顶层 const 和底层 const 在每个例子中有何体现。

```
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
```



2.4.4 constexpr 和常量表达式

常量表达式(const expression)是指值不会改变并且在编译过程就能得到计算结果的表达式。显然, 字面值属于常量表达式, 用常量表达式初始化的 const 对象也是常量表达式。后面将会提到, C++语言中有几种情况下是要用到常量表达式的。

一个对象(或表达式)是不是常量表达式由它的数据类型和初始值共同决定, 例如:

```
const int max_files = 20; // max_files 是常量表达式
const int limit = max_files + 1; // limit 是常量表达式
int staff_size = 27; // staff_size 不是常量表达式
```

```
const int sz = get_size();           // sz 不是常量表达式
```

尽管 `staff_size` 的初始值是个字面值常量，但由于它的数据类型只是一个普通 `int` 而非 `const int`，所以它不属于常量表达式。另一方面，尽管 `sz` 本身是一个常量，但它具体值直到运行时才能获取到，所以也不是常量表达式。

constexpr 变量

在一个复杂系统中，很难（几乎肯定不能）分辨一个初始值到底是不是常量表达式。66
当然可以定义一个 `const` 变量并把它的初始值设为我们认为的某个常量表达式，但在实际使用时，尽管要求如此却常常发现初始值并非常量表达式的情况。可以说，在此种情况下，对象的定义和使用根本就是两回事儿。

C++11 新标准规定，允许将变量声明为 `constexpr` 类型以便由编译器来验证变量的值是否是一个常量表达式。声明为 `constexpr` 的变量一定是一个常量，而且必须用常量表达式初始化：C++
11

```
constexpr int mf = 20;           // 20 是常量表达式
constexpr int limit = mf + 1;    // mf + 1 是常量表达式
constexpr int sz = size();      // 只有当 size 是一个 constexpr 函数时
                                // 才是一条正确的声明语句
```

尽管不能使用普通函数作为 `constexpr` 变量的初始值，但是正如 6.5.2 节（第 214 页）将要介绍的，新标准允许定义一种特殊的 `constexpr` 函数。这种函数应该足够简单以使得编译时就可以计算其结果，这样就能用 `constexpr` 函数去初始化 `constexpr` 变量了。



一般来说，如果你认定变量是一个常量表达式，那就把它声明成 `constexpr` 类型。

字面值类型

常量表达式的值需要在编译时就得到计算，因此对声明 `constexpr` 时用到的类型必须有所限制。因为这些类型一般比较简单，值也显而易见、容易得到，就把它们称为“字面值类型”（literal type）。

到目前为止接触过的数据类型中，算术类型、引用和指针都属于字面值类型。自定义类 `Sales_item`、`IO` 库、`string` 类型则不属于字面值类型，也就不能被定义成 `constexpr`。其他一些字面值类型将在 7.5.6 节（第 267 页）和 19.3 节（第 736 页）介绍。

尽管指针和引用都能定义成 `constexpr`，但它们的初始值却受到严格限制。一个 `constexpr` 指针的初始值必须是 `nullptr` 或者 0，或者是存储于某个固定地址中的对象。

6.1.1 节（第 184 页）将要提到，函数体内定义的变量一般来说并非存放在固定地址中，因此 `constexpr` 指针不能指向这样的变量。相反的，定义于所有函数体之外的对象其地址固定不变，能用来初始化 `constexpr` 指针。同样是在 6.1.1 节（第 185 页）中还将提到，允许函数定义一类有效范围超出函数本身的变量，这类变量和定义在函数体之外的变量一样也有固定地址。因此，`constexpr` 引用能绑定到这样的变量上，`constexpr` 指针也能指向这样的变量。

指针和 `constexpr`

必须明确一点，在 `constexpr` 声明中如果定义了一个指针，限定符 `constexpr` 仅

对指针有效，与指针所指的对象无关：

```
const int *p = nullptr; // p 是一个指向整型常量的指针
constexpr int *q = nullptr; // q 是一个指向整数的常量指针
```

p 和 q 的类型相差甚远，p 是一个指向常量的指针，而 q 是一个常量指针，其中的关键在于 `constexpr` 把它所定义的对象置为了顶层 `const`（参见 2.4.3 节，第 57 页）。

与其他常量指针类似，`constexpr` 指针既可以指向常量也可以指向一个非常量：

```
constexpr int *np = nullptr; // np 是一个指向整数的常量指针，其值为空
int j = 0;
constexpr int i = 42; // i 的类型是整型常量
// i 和 j 都必须定义在函数体之外
constexpr const int *p = &i; // p 是常量指针，指向整型常量 i
constexpr int *p1 = &j; // p1 是常量指针，指向整数 j
```

2.4.4 节练习

练习 2.32：下面的代码是否合法？如果非法，请设法将其修改正确。

```
int null = 0, *p = null;
```

2.5 处理类型

随着程序越来越复杂，程序中用到的类型也越来越复杂，这种复杂性体现在两个方面。一是一些类型难于“拼写”，它们的名字既难记又容易写错，还无法明确体现其真实目的和含义。二是有时候根本搞不清到底需要的类型是什么，程序员不得不回过头去从程序的上下文中寻求帮助。

2.5.1 类型别名

类型别名（type alias）是一个名字，它是某种类型的同义词。使用类型别名有很多好处，它让复杂的类型名字变得简单明了、易于理解和使用，还有助于程序员清楚地知道使用该类型的真实目的。

有两种方法可用于定义类型别名。传统的方法是使用关键字 `typedef`：

```
typedef double wages; // wages 是 double 的同义词
typedef wages base, *p; // base 是 double 的同义词，p 是 double* 的同义词
```

68> 其中，关键字 `typedef` 作为声明语句中的基本数据类型（参见 2.3 节，第 45 页）的一部分出现。含有 `typedef` 的声明语句定义的不再是变量而是类型别名。和以前的声明语句一样，这里的声明符也可以包含类型修饰，从而也能由基本数据类型构造出复合类型来。

新标准规定了一种新的方法，使用**别名声明**（alias declaration）来定义类型的别名：

```
using SI = Sales_item; // SI 是 Sales_item 的同义词
```

这种方法用关键字 `using` 作为别名声明的开始，其后紧跟别名和等号，其作用是把等号左侧的名字规定成等号右侧类型的别名。

类型别名和类型的名字等价，只要是类型的名字能出现的地方，就能使用类型别名：

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

```
SI item; // 等价于 Sales_item item
```

指针、常量和类型别名



如果某个类型别名指代的是复合类型或常量，那么把它用到声明语句里就会产生意想不到的后果。例如下面的声明语句用到了类型 `pstring`，它实际上是类型 `char*` 的别名：

```
typedef char *pstring;
const pstring cstr = 0; // cstr 是指向 char 的常量指针
const pstring *ps; // ps 是一个指针，它的对象是指向 char 的常量指针
```

上述两条声明语句的基本数据类型都是 `const pstring`，和过去一样，`const` 是对给定类型的修饰。`pstring` 实际上是指向 `char` 的指针，因此，`const pstring` 就是指向 `char` 的常量指针，而非指向常量字符的指针。

遇到一条使用了类型别名的声明语句时，人们往往会错误地尝试把类型别名替换成它本来的样子，以理解该语句的含义：

```
const char *cstr = 0; // 是对 const pstring cstr 的错误理解
```

再强调一遍：这种理解是错误的。声明语句中用到 `pstring` 时，其基本数据类型是指针。可是用 `char*` 重写了声明语句后，数据类型就变成了 `char`，`*` 成为了声明符的一部分。这样改写的结果是，`const char` 成了基本数据类型。前后两种声明含义截然不同，前者声明了一个指向 `char` 的常量指针，改写后的形式则声明了一个指向 `const char` 的指针。

2.5.2 auto 类型说明符



编程时常常需要把表达式的值赋给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而要做到这一点并非那么容易，有时甚至根本做不到。为了解决这个问题，C++11 新标准引入了 `auto` 类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应一种特定类型的说明符（比如 `double`）不同，`auto` 让编译器通过初始值来推算变量的类型。显然，`auto` 定义的变量必须有初始值：

```
// 由 val1 和 val2 相加的结果可以推断出 item 的类型
auto item = val1 + val2; // item 初始化为 val1 和 val2 相加的结果
```

此处编译器将根据 `val1` 和 `val2` 相加的结果来推断 `item` 的类型。如果 `val1` 和 `val2` 是类 `Sales_item`（参见 1.5 节，第 17 页）的对象，则 `item` 的类型就是 `Sales_item`；如果这两个变量的类型是 `double`，则 `item` 的类型就是 `double`，以此类推。

使用 `auto` 也能在一条语句中声明多个变量。因为一条声明语句只能有一个基本数据类型，所以该语句中所有变量的初始基本数据类型都必须一样：

```
auto i = 0, *p = &i; // 正确：i 是整数、p 是整型指针
auto sz = 0, pi = 3.14; // 错误：sz 和 pi 的类型不一致
```

复合类型、常量和 auto



C++ 11

69

编译器推断出来的 `auto` 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

首先，正如我们所熟知的，使用引用其实是使用引用的对象，特别是当引用被用作初始值时，真正参与初始化的其实是引用对象的值。此时编译器以引用对象的类型作为 `auto` 的类型：

```
int i = 0, &r = i;
auto a = r;           // a 是一个整数 (r 是 i 的别名, 而 i 是一个整数)
```

其次, `auto` 一般会忽略掉顶层 `const` (参见 2.4.3 节, 第 57 页), 同时底层 `const` 则会保留下来, 比如当初始值是一个指向常量的指针时:

```
const int ci = i, &cr = ci;
auto b = ci;           // b 是一个整数 (ci 的顶层 const 特性被忽略掉了)
auto c = cr;           // c 是一个整数 (cr 是 ci 的别名, ci 本身是一个顶层 const)
auto d = &i;            // d 是一个整型指针 (整数的地址就是指向整数的指针)
auto e = &ci;           // e 是一个指向整数常量的指针 (对常量对象取地址是一种底层 const)
```

如果希望推断出的 `auto` 类型是一个顶层 `const`, 需要明确指出:

```
const auto f = ci;       // ci 的推演类型是 int, f 是 const int
```

还可以将引用的类型设为 `auto`, 此时原来的初始化规则仍然适用:

```
auto &g = ci;           // g 是一个整型常量引用, 绑定到 ci
auto &h = 42;             // 错误: 不能为非常量引用绑定字面值
const auto &j = 42;        // 正确: 可以为常量引用绑定字面值
```

70 设置一个类型为 `auto` 的引用时, 初始值中的顶层常量属性仍然保留。和往常一样, 如果我们给初始值绑定一个引用, 则此时的常量就不是顶层常量了。

要在一条语句中定义多个变量, 切记, 符号`&`和`*`只从属于某个声明符, 而非基本数据类型的一部分, 因此初始值必须是同一种类型:

```
auto k = ci, &l = i;      // k 是整数, l 是整型引用
auto &m = ci, *p = &ci;    // m 是对整型常量的引用, p 是指向整型常量的指针
// 错误: i 的类型是 int 而 &ci 的类型是 const int
auto &n = i, *p2 = &ci;
```

2.5.2 节练习

练习 2.33: 利用本节定义的变量, 判断下列语句的运行结果。

```
a = 42; b = 42; c = 42;
d = 42; e = 42; g = 42;
```

练习 2.34: 基于上一个练习中的变量和语句编写一段程序, 输出赋值前后变量的内容, 你刚才的推断正确吗? 如果不对, 请反复研读本节的示例直到你明白错在何处为止。

练习 2.35: 判断下列定义推断出的类型是什么, 然后编写程序进行验证。

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```



2.5.3 decltype 类型指示符

有时会遇到这种情况: 希望从表达式的类型推断出要定义的变量的类型, 但是不想用该表达式的值初始化变量。为了满足这一要求, C++11 新标准引入了第二种类型说明符 `decltype`, 它的作用是选择并返回操作数的数据类型。在此过程中, 编译器分析表达式并得到它的类型, 却不实际计算表达式的值:

```
decltype(f()) sum = x; // sum 的类型就是函数 f 的返回类型
```



编译器并不实际调用函数 `f`, 而是使用当调用发生时 `f` 的返回值类型作为 `sum` 的类型。换句话说, 编译器为 `sum` 指定的类型是什么呢? 就是假如 `f` 被调用的话将会返回的那个类型。

`decltype` 处理顶层 `const` 和引用的方式与 `auto` 有些许不同。如果 `decltype` 使用的表达式是一个变量, 则 `decltype` 返回该变量的类型(包括顶层 `const` 和引用在内):

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;           // x 的类型是 const int
decltype(cj) y = x;          // y 的类型是 const int&, y 绑定到变量 x
decltype(cj) z;              // 错误: z 是一个引用, 必须初始化
```

因为 `cj` 是一个引用, `decltype(cj)` 的结果就是引用类型, 因此作为引用的 `z` 必须被初始化。

需要指出的是, 引用从来都作为其所指对象的同义词出现, 只有用在 `decltype` 处是一个例外。

decltype 和引用



如果 `decltype` 使用的表达式不是一个变量, 则 `decltype` 返回表达式结果对应的类型。如 4.1.1 节(第 120 页)将要介绍的, 有些表达式将向 `decltype` 返回一个引用类型。一般来说当这种情况发生时, 意味着该表达式的结果对象能作为一条赋值语句的左值:

```
// decltype 的结果可以是引用类型
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;      // 正确: 加法的结果是 int, 因此 b 是一个(未初始化的) int
decltype(*p) c;        // 错误: c 是 int&, 必须初始化
```

因为 `r` 是一个引用, 因此 `decltype(r)` 的结果是引用类型。如果想让结果类型是 `r` 所指的类型, 可以把 `r` 作为表达式的一部分, 如 `r+0`, 显然这个表达式的结果将是一个具体值而非一个引用。

另一方面, 如果表达式的内容是解引用操作, 则 `decltype` 将得到引用类型。正如我们所熟悉的那样, 解引用指针可以得到指针所指的对象, 而且还能给这个对象赋值。因此, `decltype(*p)` 的结果类型就是 `int&`, 而非 `int`。



`decltype` 和 `auto` 的另一处重要区别是, `decltype` 的结果类型与表达式形式密切相关。有一种情况需要特别注意: 对于 `decltype` 所用的表达式来说, 如果变量名加上了一对括号, 则得到的类型与不加括号时会有不同。如果 `decltype` 使用的是一个不加括号的变量, 则得到的结果就是该变量的类型; 如果给变量加上了一层或多层括号, 编译器就会把它当成是一个表达式。变量是一种可以作为赋值语句左值的特殊表达式, 所以这样的 `decltype` 就会得到引用类型:

```
// decltype 的表达式如果是加上了括号的变量, 结果将是引用
decltype((i)) d;      // 错误: d 是 int&, 必须初始化
decltype(i) e;         // 正确: e 是一个(未初始化的) int
```



切记: `decltype((variable))`(注意是双层括号)的结果永远是引用, 而 `decltype(variable)` 结果只有当 `variable` 本身就是一个引用时才是引用。

72

2.5.3 节练习

练习 2.36: 关于下面的代码，请指出每一个变量的类型以及程序结束时它们各自的值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

练习 2.37: 赋值是会产生引用的一类典型表达式，引用的类型就是左值的类型。也就是说，如果 *i* 是 int，则表达式 *i=x* 的类型是 int&。根据这一特点，请指出下面的代码中每一个变量的类型和值。

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

练习 2.38: 说明由 decltype 指定类型和由 auto 指定类型有何区别。请举出一个例子， decltype 指定的类型与 auto 指定的类型一样；再举一个例子， decltype 指定的类型与 auto 指定的类型不一样。



2.6 自定义数据结构

从最基本的层面理解，数据结构是把一组相关的数据元素组织起来然后使用它们的策略和方法。举一个例子，我们的 Sales_item 类把书本的 ISBN 编号、售出量及销售收入等数据组织在了一起，并且提供诸如 isbn 函数、>>、<<、+、+= 等运算在内的一系列操作，Sales_item 类就是一个数据结构。

C++语言允许用户以类的形式自定义数据类型，而库类型 string、istream、ostream 等也都是以类的形式定义的，就像第 1 章的 Sales_item 类型一样。C++语言对类的支持甚多，事实上本书的第 III 部分和第 IV 部分都将大篇幅地介绍与类有关的知识。尽管 Sales_item 类非常简单，但是要想给出它的完整定义可在第 14 章介绍自定义运算符之后。



2.6.1 定义 Sales_data 类型

尽管我们还写不出完整的 Sales_item 类，但是可以尝试着把那些数据元素组织到一起形成一个简单点儿的类。初步的想法是用户能直接访问其中的数据元素，也能实现一些基本的操作。

既然我们筹划的这个数据结构不带有任何运算功能，不妨把它命名为 Sales_data 以示与 Sales_item 的区别。Sales_data 初步定义如下：

73

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

我们的类以关键字 struct 开始，紧跟着类名和类体（其中类体部分可以为空）。类体由

花括号包围形成了一个新的作用域（参见 2.2.4 节，第 43 页）。类内部定义的名字必须唯一，但是可以与类外部定义的名字重复。

类体右侧的表示结束的花括号后必须写一个分号，这是因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号必不可少：

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;  
// 与上一条语句等价，但可能更好一些  
struct Sales_data { /* ... */ };  
Sales_data accum, trans, *salesptr;
```

分号表示声明符（通常为空）的结束。一般来说，最好不要把对象的定义和类的定义放在一起。这么做无异于把两种不同实体的定义混在了一条语句里，一会儿定义类，一会儿又定义变量，显然这是一种不被建议的行为。



很多新手程序员经常忘了在类定义的最后加上分号。

WARNING

类数据成员

类体定义类的成员，我们的类只有数据成员（data member）。类的数据成员定义了类的对象的具体内容，每个对象都有自己的一份数据成员拷贝。修改一个对象的数据成员，不会影响其他 `Sales_data` 的对象。

定义数据成员的方法和定义普通变量一样：首先说明一个基本类型，随后紧跟一个或多个声明符。我们的类有 3 个数据成员：一个名为 `bookNo` 的 `string` 成员、一个名为 `units_sold` 的 `unsigned` 成员和一个名为 `revenue` 的 `double` 成员。每个 `Sales_data` 的对象都将包括这 3 个数据成员。

C++11 新标准规定，可以为数据成员提供一个类内初始值（in-class initializer）。创建对象时，类内初始值将用于初始化数据成员。没有初始值的成员将被默认初始化（参见 2.2.1 节，第 40 页）。因此当定义 `Sales_data` 的对象时，`units_sold` 和 `revenue` 都将初始化为 0，`bookNo` 将初始化为空字符串。

C++
11

对类内初始值的限制与之前（参见 2.2.1 节，第 39 页）介绍的类似：或者放在花括号里，或者放在等号右边，记住不能使用圆括号。

7.2 节（第 240 页）将要介绍，用户可以使用 C++ 语言提供的另外一个关键字 `class` 来定义自己的数据结构，到时也将说明现在我们使用 `struct` 的原因。在第 7 章学习与 `class` 有关的知识之前，建议读者继续使用 `struct` 定义自己的数据类型。

2.6.1 节练习

< 74

练习 2.39：编译下面的程序观察其运行结果，注意，如果忘记写类定义体后面的分号会发生什么情况？记录下相关信息，以后可能会有用。

```
struct Foo { /* 此处为空 */ } // 注意：没有分号  
int main()  
{  
    return 0;  
}
```

练习 2.40：根据自己的理解写出 `Sales_data` 类，最好与书中的例子有所区别。



2.6.2 使用 Sales_data 类

和 Sales_item 类不同的是，我们自定义的 Sales_data 类没有提供任何操作，Sales_data 类的使用者如果想执行什么操作就必须自己动手实现。例如，我们将参照 1.5.2 节（第 20 页）的例子写一段程序实现求两次交易相加结果的功能。程序的输入是下面这两条交易记录：

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

每笔交易记录着图书的 ISBN 编号、售出数量和售出单价。

添加两个 Sales_data 对象

因为 Sales_data 类没有提供任何操作，所以我们必须自己编码实现输入、输出和相加的功能。假设已知 Sales_data 类定义于 Sales_data.h 文件内，2.6.3 节（第 67 页）将详细介绍定义头文件的方法。

因为程序比较长，所以接下来分成几部分介绍。总的来说，程序的结构如下：

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // 读入 data1 和 data2 的代码
    // 检查 data1 和 data2 的 ISBN 是否相同的代码
    // 如果相同，求 data1 和 data2 的总和
}
```

和原来的程序一样，先把所需的头文件包含进来并且定义变量用于接受输入。和 Sales_item 类不同的是，新程序还包含了 string 头文件，因为我们的代码中将用到 string 类型的成员变量 bookNo。

75

Sales_data 对象读入数据

第 3 章和第 10 章将详细介绍 string 类型的细节，在此之前，我们先了解一点儿关于 string 的知识以便定义和使用我们的 ISBN 成员。string 类型其实就是字符的序列，它的操作有`>>`、`<<`和`==`等，功能分别是读入字符串、写出字符串和比较字符串。这样我们就能书写代码读入第一笔交易了：

```
double price = 0; // 书的单价，用于计算销售收入
// 读入第 1 笔交易： ISBN、销售数量、单价
std::cin >> data1.bookNo >> data1.units_sold >> price;
// 计算销售收入
data1.revenue = data1.units_sold * price;
```

交易信息记录的是书售出的单价，而数据结构存储的是一次交易的销售收入，因此需要将单价读入到 double 变量 price，然后再计算销售收入 revenue。输入语句

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

使用点操作符（参见 1.5.2 节，第 20 页）读入对象 data1 的 bookNo 成员和 units_sold 成员。

最后一条语句把 data1.units_sold 和 price 的乘积赋值给 data1 的 revenue 成员。

接下来程序重复上述过程读入对象 data2 的数据：

```
// 读入第 2 笔交易
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

输出两个 Sales_data 对象的和

剩下的工作就是检查两笔交易涉及的 ISBN 编号是否相同了。如果相同输出它们的和，否则输出一条报错信息：

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // 输出：ISBN、总销售量、总销售额、平均价格
    std::cout << data1.bookNo << " " << totalCnt
        << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0;           // 标示成功
} else {               // 两笔交易的 ISBN 不一样
    std::cerr << "Data must refer to the same ISBN"
        << std::endl;
    return -1;          // 标示失败
}
```

在第一个 if 语句中比较了 data1 和 data2 的 bookNo 成员是否相同。如果相同则执行第一个 if 语句花括号内的操作，首先计算 units_sold 的和并赋给变量 totalCnt，然后计算 revenue 的和并赋给变量 totalRevenue，输出这些值。接下来检查是否确实售出了书籍，如果是，计算并输出每本书的平均价格；如果售量为零，输出一条相应的信息。

< 76

2.6.2 节练习

练习 2.41：使用你自己的 Sales_data 类重写 1.5.1 节（第 20 页）、1.5.2 节（第 21 页）和 1.6 节（第 22 页）的练习。眼下先把 Sales_data 类的定义和 main 函数放在同一个文件里。

2.6.3 编写自己的头文件



尽管如 19.7 节（第 754 页）所讲可以在函数体内定义类，但是这样的类毕竟受到了一些限制。所以，类一般都不定义在函数体内。当在函数体外部定义类时，在各个指定的源文件中可能只有一处该类的定义。而且，如果要在不同文件中使用同一个类，类的定义就必须保持一致。

为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应与类的名字一样。例如，库类型 string 在名为 string 的头文件中定义。又如，我们应该把 Sales_data 类定义在名为 Sales_data.h 的头文件中。

头文件通常包含那些只能被定义一次的实体，如类、const 和 constexpr 变量（参见 2.4 节，第 54 页）等。头文件也经常用到其他头文件的功能。例如，我们的 Sales_data 类包含有一个 string 成员，所以 Sales_data.h 必须包含 string.h 头文件。同时，使用 Sales_data 类的程序为了能操作 bookNo 成员需要再一次包含 string.h 头文件。

这样，事实上使用 `Sales_data` 类的程序就先后两次包含了 `string.h` 头文件：一次是直接包含的，另有一次是随着包含 `Sales_data.h` 被隐式地包含进来的。有必要在书写头文件时做适当处理，使其遇到多次包含的情况也能安全和正常地工作。



头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

预处理器概述

77 确保头文件多次包含仍能安全工作的常用技术是预处理器（preprocessor），它由 C++ 语言从 C 语言继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前已经用到了一项预处理功能`#include`，当预处理器看到`#include` 标记时就会用指定的头文件的内容代替`#include`。

C++程序还会用到的一项预处理功能是头文件保护符（header guard），头文件保护符依赖于预处理变量（参见 2.3.2 节，第 48 页）。预处理变量有两种状态：已定义和未定义。`#define` 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：`#ifdef` 当且仅当变量已定义时为真，`#ifndef` 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直至遇到`#endif` 指令为止。

使用这些功能就能有效地防止重复包含的发生：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

第一次包含 `Sales_data.h` 时，`#ifndef` 的检查结果为真，预处理器将顺序执行后面的操作直至遇到`#endif` 为止。此时，预处理变量 `SALES_DATA_H` 的值将变为已定义，而且 `Sales_data.h` 也会被拷贝到我们的程序中来。后面如果再一次包含 `Sales_data.h`，则`#ifndef` 的检查结果将为假，编译器将忽略`#ifndef` 到`#endif` 之间的部分。



预处理变量无视 C++ 语言中关于作用域的规则。

WARNING

整个程序中的预处理变量包括头文件保护符必须唯一，通常的做法是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。为了避免与程序中的其他实体发生名字冲突，一般把预处理变量的名字全部大写。



头文件即使（目前还）没有被包含在任何其他头文件中，也应该设置保护符。

头文件保护符很简单，程序员只要习惯性地加上就可以了，没必要太在乎你的程序到底需不需要。

2.6.3 节练习

练习 2.42：根据你自己的理解重写一个 `Sales_data.h` 头文件，并以此为基础重做 2.6.2 节（第 67 页）的练习。

小结

78

类型是 C++ 编程的基础。

类型规定了其对象的存储要求和所能执行的操作。C++ 语言提供了一套基础内置类型，如 `int` 和 `char` 等，这些类型与实现它们的机器硬件密切相关。类型分为非常量和常量，一个常量对象必须初始化，而且一旦初始化其值就不能再改变。此外，还可以定义复合类型，如指针和引用等。复合类型的定义以其他类型为基础。

C++ 语言允许用户以类的形式自定义类型。C++ 库通过类提供了一套高级抽象类型，如输入输出和 `string` 等。

术语表

地址 (address) 是一个数字，根据它可以找到内存中的一个字节。

别名声明 (alias declaration) 为另外一种类型定义一个同义词：使用“名字=类型”的格式将名字作为该类型的同义词。

算术类型 (arithmetic type) 布尔值、字符、整数、浮点数等内置类型。

数组 (array) 是一种数据结构，存放着一组未命名的对象，可以通过索引来访问这些对象。3.5 节将详细介绍数组的知识。

auto 是一个类型说明符，通过变量的初始值来推断变量的类型。

基本类型 (base type) 是类型说明符，可用 `const` 修饰，在声明语句中位于声明符之前。基本类型提供了最常见的数据类型，以此为基础构建声明符。

绑定 (bind) 令某个名字与给定的实体关联在一起，使用该名字也就是使用该实体。例如，引用就是将某个名字与某个对象绑定在一起。

字节 (byte) 内存中可寻址的最小单元，大多数机器的字节占 8 位。

类成员 (class member) 类的组成部分。

复合类型 (compound type) 是一种类型，它的定义以其他类型为基础。

const 是一种类型修饰符，用于说明永不改变的对象。`const` 对象一旦定义就无法再

赋新值，所以必须初始化。

常量指针 (const pointer) 是一种指针，它的值永不改变。

常量引用 (const reference) 是一种习惯叫法，含义是指向常量的引用。

常量表达式 (const expression) 能在编译时计算并获取结果的表达式。

constexpr 是一种函数，用于代表一条常量表达式。6.5.2 节（第 214 页）将介绍 `constexpr` 函数。

转换 (conversion) 一种类型的值转变成另外一种类型值的过程。C++ 语言支持内置类型之间的转换。

数据成员 (data member) 组成对象的数据元素，类的每个对象都有类的数据成员的一份拷贝。数据成员可以在类内部声明的同时初始化。

声明 (declaration) 声称存在一个变量、函数或是别处定义的类型。名字必须在定义或声明之后才能使用。

79

声明符 (declarator) 是声明的一部分，包括被定义的名字和类型修饰符，其中类型修饰符可以有也可以没有。

decltype 是一个类型说明符，从变量或表达式推断得到类型。

默认初始化 (default initialization) 当对象未被显式地赋予初始值时执行的初始化行

为。由类本身负责执行的类对象的初始化行为。全局作用域的内置类型对象初始化为 0；局部作用域的对象未被初始化即拥有未定义的值。

定义 (definition) 为某一特定类型的变量申请存储空间，可以选择初始化该变量。名字必须在定义或声明之后才能使用。

转义序列 (escape sequence) 字符特别是那些不可打印字符的替代形式。转义以反斜线开头，后面紧跟一个字符，或者不多于 3 个八进制数字，或者字母 x 加上 1 个十六进制数。

全局作用域 (global scope) 位于其他所有作用域之外的作用域。

头文件保护符 (header guard) 使用预处理变量以防止头文件被某个文件重复包含。

标识符 (identifier) 组成名字的字符序列，标识符对大小写敏感。

类内初始值 (in-class initializer) 在声明类的数据成员时同时提供的初始值，必须置于等号右侧或花括号内。

在作用域内 (in scope) 名字在当前作用域内可见。

被初始化 (initialized) 变量在定义的同时被赋予初始值，变量一般都应该被初始化。

内层作用域 (inner scope) 嵌套在其他作用域之内的作用域。

整型 (integral type) 参见算术类型。

列表初始化 (list initialization) 利用花括号把一个或多个初始值放在一起的初始化形式。

字面值 (literal) 是一个不能改变的值，如数字、字符、字符串等。单引号内的是字符字面值，双引号内的是字符串字面值。

局部作用域 (local scope) 是块作用域的习惯叫法。

底层 const (low-level const) 一个不属于顶层的 const，类型如果由底层常量定义，

则不能被忽略。

成员 (member) 类的组成部分。

不可打印字符 (nonprintable character) 不具有可见形式的字符，如控制符、退格、换行符等。

空指针 (null pointer) 值为 0 的指针，空指针合法但是不指向任何对象。

nullptr 是表示空指针的字面值常量。

对象 (object) 是内存的一块区域，具有某种类型，变量是命名了的对象。

外层作用域 (outer scope) 嵌套着别的作用域的作用域。

指针 (pointer) 是一个对象，存放着某个对象的地址，或者某个对象存储区域之后的下一地址，或者 0。

指向常量的指针 (pointer to const) 是一个指针，存放着某个常量对象的地址。指向常量的指针不能用来改变它所指对象的值。

预处理器 (preprocessor) 在 C++ 编译过程中执行的一段程序。

预处理变量 (preprocessor variable) 由预处理器管理的变量。在程序编译之前，预处理器负责将程序中的预处理变量替换成它的真实值。

引用 (reference) 是某个对象的别名。

80 对常量的引用 (reference to const) 是一个引用，不能用来改变它所绑定对象的值。对常量的引用可以绑定常量对象，或者非常量对象，或者表达式的结果。

作用域 (scope) 是程序的一部分，在其中某些名字有意义。C++ 有几级作用域：

全局 (global) ——名字定义在所有其他作用域之外。

类 (class) ——名字定义在类内部。

命名空间 (namespace) ——名字定义在命名空间内部。

块 (block) ——名字定义在块内部。

名字从声明位置开始直至声明语句所在的作用域末端为止都是可用的。

分离式编译 (separate compilation) 把程序分割为多个单独文件的能力。

带符号类型 (signed) 保存正数、负数或 0 的整型。

字符串 (string) 是一种库类型，表示可变长字符序列。

struct 是一个关键字，用于定义类。

临时值 (temporary) 编译器在计算表达式结果时创建的无名对象。为某表达式创建了一个临时值，则此临时值将一直存在直到包含有该表达式的最大的表达式计算完成为止。

顶层 const (top-level const) 是一个 **const**，规定某对象的值不能改变。

类型别名 (type alias) 是一个名字，是另外一个类型的同义词，通过关键字 **typedef** 或别名声明语句来定义。

类型检查 (type checking) 是一个过程，编译器检查程序使用某给定类型对象的方式与该类型的定义是否一致。

类型说明符 (type specifier) 类型的名字。

typedef 为某类型定义一个别名。当关键字 **typedef** 作为声明的基本类型出现时，声明中定义的名字就是类型名。

未定义 (undefined) 即 C++ 语言没有明确规定的情况。不论是否有意为之，未定义行为都可能引发难以追踪的运行时错误、安全问题和可移植性问题。

未初始化 (uninitialized) 变量已定义但未被赋予初始值。一般来说，试图访问未初始化变量的值将引发未定义行为。

无符号类型 (unsigned) 保存大于等于 0 的整型。

变量 (variable) 命名的对象或引用。C++ 语言要求变量要先声明后使用。

void* 可以指向任意非常量的指针类型，不能执行解引用操作。

void 类型 是一种有特殊用处的类型，既无操作也无值。不能定义一个 **void** 类型的变量。

字 (word) 在指定机器上进行整数运算的自然单位。一般来说，字的空间足够存放地址。32 位机器上的字通常占据 4 个字节。

& 运算符 (& operator) 取地址运算符。

*** 运算符 (* operator)** 解引用运算符。解引用一个指针将返回该指针所指的对象，为解引用的结果赋值也就是为指针所指的对象赋值。

#define 是一条预处理指令，用于定义一个预处理变量。

#endif 是一条预处理指令，用于结束一个 **#ifdef** 或 **#ifndef** 区域。

#ifdef 是一条预处理指令，用于判断给定的变量是否已经定义。

#ifndef 是一条预处理指令，用于判断给定的变量是否尚未定义。

第3章 字符串、向量和数组

内容

3.1 命名空间的 using 声明	74
3.2 标准库类型 string	75
3.3 标准库类型 vector	86
3.4 迭代器介绍	95
3.5 数组	101
3.6 多维数组	112
小结	117
术语表	117

除了第2章介绍的内置类型之外,C++语言还定义了一个内容丰富的抽象数据类型库。其中, `string` 和 `vector` 是两种最重要的标准库类型, 前者支持可变长字符串, 后者则表示可变长的集合。还有一种标准库类型是迭代器, 它是 `string` 和 `vector` 的配套类型, 常被用于访问 `string` 中的字符或 `vector` 中的元素。

内置数组是一种更基础的类型, `string` 和 `vector` 都是对它的某种抽象。本章将分别介绍数组以及标准库类型 `string` 和 `vector`。

82 第2章介绍的内置类型是由C++语言直接定义的。这些类型，比如数字和字符，体现了大多数计算机硬件本身具备的能力。标准库定义了另外一组具有更高级性质的类型，它们尚未直接实现到计算机硬件中。

本章将介绍两种最重要的标准库类型：`string` 和 `vector`。`string` 表示可变长的字符串序列，`vector` 存放的是某种给定类型对象的可变长序列。本章还将介绍内置数组类型，和其他内置类型一样，数组的实现与硬件密切相关。因此相较于标准库类型 `string` 和 `vector`，数组在灵活性上稍显不足。

在开始介绍标准库类型之前，先来学习一种访问库中名字的简单方法。

3.1 命名空间的 `using` 声明

目前为止，我们用到的库函数基本上都属于命名空间 `std`，而程序也显式地将这一点标示了出来。例如，`std::cin` 表示从标准输入中读取内容。此处使用作用域操作符 (`::`) (参见 1.2 节，第 7 页) 的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧那个名字。因此，`std::cin` 的意思就是要使用命名空间 `std` 中的名字 `cin`。

上面的方法显得比较烦琐，然而幸运的是，通过更简单的途径也能使用到命名空间中的成员。本节将学习其中一种最安全的方法，也就是使用 **using 声明** (using declaration)，18.2.2 节 (第 702 页) 会介绍另一种方法。

有了 `using` 声明就无须专门的前缀 (形如命名空间`::`) 也能使用所需的名字了。`using` 声明具有如下的形式：

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

一旦声明了上述语句，就可以直接访问命名空间中的名字：

```
#include <iostream>
// using 声明，当我们使用名字 cin 时，从命名空间 std 中获取它
using std::cin;

int main()
{
    int i;
    cin >> i;           // 正确：cin 和 std::cin 含义相同
    cout << i;          // 错误：没有对应的 using 声明，必须使用完整的名字
    std::cout << i;    // 正确：显式地从 std 中使用 cout
    return 0;
}
```

每个名字都需要独立的 `using` 声明

按照规定，每个 `using` 声明引入命名空间中的一个成员。例如，可以把要用到的标准库中的名字都以 `using` 声明的形式表示出来，重写 1.2 节 (第 5 页) 的程序如下：

83

```
#include <iostream>
// 通过下列 using 声明，我们可以使用标准库中的名字
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
```

```
int v1, v2;
cin >> v1 >> v2;
cout << "The sum of " << v1 << " and " << v2
    << " is " << v1 + v2 << endl;
return 0;
}
```

在上述程序中，一开始就有对 `cin`、`cout` 和 `endl` 的 `using` 声明，这意味着我们不用再添加 `std::` 形式的前缀就能直接使用它们。C++ 语言的形式比较自由，因此既可以一行只放一条 `using` 声明语句，也可以一行放上多条。不过要注意，用到的每个名字都必须有自己的声明语句，而且每句话都得以分号结束。

头文件不应包含 `using` 声明

位于头文件的代码（参见 2.6.3 节，第 67 页）一般来说不应该使用 `using` 声明。这是因为头文件的内容会拷贝到所有引用它的文件中去，如果头文件里有某个 `using` 声明，那么每个使用了该头文件的文件就都会有这个声明。对于某些程序来说，由于不经意间包含了一些名字，反而可能产生始料未及的名字冲突。

一点注意事项

经本节所述，后面的所有例子将假设，但凡用到的标准库中的名字都已经使用 `using` 语句声明过了。例如，我们将在代码中直接使用 `cin`，而不再使用 `std::cin`。

为了让书中的代码尽量简洁，今后将不会再把所有 `using` 声明和 `#include` 指令一一标出。附录 A 中的表 A.1（第 766 页）列出了本书涉及的所有标准库中的名字及对应的头文件。



读者请注意：在编译及运行本书的示例前请为代码添加必要的 `#include` 指令和 `using` 声明。

3.1 节练习

练习 3.1：使用恰当的 `using` 声明重做 1.4.1 节（第 11 页）和 2.6.2 节（第 67 页）的练习。

3.2 标准库类型 `string`



标准库类型 `string` 表示可变长的字符序列，使用 `string` 类型必须首先包含 `string` 头文件。作为标准库的一部分，`string` 定义在命名空间 `std` 中。接下来的示例都假定已包含了下述代码：

```
#include <string>
using std::string;
```

本节描述最常用的 `string` 操作，9.5 节（第 320 页）还将介绍另外一些。



C++ 标准一方面对库类型所提供的操作做了详细规定，另一方面也对库的实现者做出一些性能上的需求。因此，标准库类型对于一般应用场合来说有足够的效率。



3.2.1 定义和初始化 string 对象

如何初始化类的对象是由类本身决定的。一个类可以定义很多种初始化对象的方式，只不过这些方式之间必须有所区别：或者是初始值的数量不同，或者是初始值的类型不同。

表 3.1 列出了初始化 string 对象最常用的一些方式，下面是几个例子：

```
string s1;                      // 默认初始化，s1 是一个空字符串
string s2 = s1;                  // s2 是 s1 的副本
string s3 = "hiya";              // s3 是该字符串字面值的副本
string s4(10, 'c');              // s4 的内容是 cccccccccc
```

可以通过默认的方式（参见 2.2.1 节，第 40 页）初始化一个 string 对象，这样就会得到一个空的 string，也就是说，该 string 对象中没有任何字符。如果提供了一个字符串字面值（参见 2.1.3 节，第 36 页），则该字面值中除了最后那个空字符外其他所有的字符都被拷贝到新创建的 string 对象中去。如果提供的是一个数字和一个字符，则 string 对象的内容是给定字符连续重复若干次后得到的序列。

表 3.1：初始化 string 对象的方式

string s1	默认初始化，s1 是一个空串
string s2(s1)	s2 是 s1 的副本
string s2 = s1	等价于 s2(s1)，s2 是 s1 的副本
string s3("value")	s3 是字面值" value "的副本，除了字面值最后的那个空字符外
string s3 = "value"	等价于 s3("value")，s3 是字面值" value "的副本
string s4(n, 'c')	把 s4 初始化为由连续 n 个字符 c 组成的串

直接初始化和拷贝初始化

由 2.2.1 节（第 39 页）的学习可知，C++ 语言有几种不同的初始化方式，通过 string 我们可以清楚地看到在这些初始化方式之间到底有什么区别和联系。如果使用等号 (=) 初始化一个变量，实际上执行的是 **拷贝初始化** (copy initialization)，编译器把等号右侧的初始值拷贝到新创建的对象中去。与之相反，如果不使用等号，则执行的是 **直接初始化** (direct initialization)。

当初始值只有一个时，使用直接初始化或拷贝初始化都行。如果像上面的 s4 那样初始化要用到的值有多个，一般来说只能使用直接初始化的方式：

```
string s5 = "hiya";          // 拷贝初始化
string s6("hiya");          // 直接初始化
string s7(10, 'c');          // 直接初始化，s7 的内容是 cccccccccc
```

对于用多个值进行初始化的情况，非要用拷贝初始化的方式来处理也不是不可以，不过需要显式地创建一个（临时）对象用于拷贝：

```
string s8 = string(10, 'c'); // 拷贝初始化，s8 的内容是 cccccccccc
```

s8 的初始值是 string(10, 'c')，它实际上是用数字 10 和字符 c 两个参数创建出来的一个 string 对象，然后这个 string 对象又拷贝给了 s8。这条语句本质上等价于下面的两条语句：

```
string temp(10, 'c');        // temp 的内容是 cccccccccc
string s8 = temp;            // 将 temp 拷贝给 s8
```

其实我们可以看到，尽管初始化 s8 的语句合法，但和初始化 s7 的方式比较起来可读性较差，也没有任何补偿优势。

3.2.2 string 对象上的操作



一个类除了要规定初始化其对象的方式外，还要定义对象上所能执行的操作。其中，类既能定义通过函数名调用的操作，就像 Sales_item 类的 isbn 函数那样（参见 1.5.2 节，第 20 页），也能定义<<、+等各种运算符在该类对象上的新含义。表 3.2 中列举了 string 的大多数操作。

表 3.2: string 的操作

os<<s	将 s 写到输出流 os 当中，返回 os
is>>s	从 is 中读取字符串赋给 s，字符串以空白分隔，返回 is
getline(is, s)	从 is 中读取一行赋给 s，返回 is
s.empty()	s 为空返回 true，否则返回 false
s.size()	返回 s 中字符的个数
s[n]	返回 s 中第 n 个字符的引用，位置 n 从 0 计起
s1+s2	返回 s1 和 s2 连接后的结果
s1=s2	用 s2 的副本代替 s1 中原来的字符
s1==s2	如果 s1 和 s2 中所含的字符完全一样，则它们相等；string 对象的相等性判断对字母的大小写敏感
s1!=s2	等性判断对字母的大小写敏感
<, <=, >, >=	利用字符在字典中的顺序进行比较，且对字母的大小写敏感

读写 string 对象

第 1 章曾经介绍过，使用标准库中的 `iostream` 来读写 `int`、`double` 等内置类型的值。同样，也可以使用 IO 操作符读写 `string` 对象：

```
// 注意：要想编译下面的代码还需要适当的#include 语句和 using 声明
int main()
{
    string s;           // 空字符串
    cin >> s;          // 将 string 对象读入 s，遇到空白停止
    cout << s << endl; // 输出 s
    return 0;
}
```

这段程序首先定义一个名为 s 的空 `string`，然后将标准输入的内容读取到 s 中。在执行读取操作时，`string` 对象会自动忽略开头的空白（即空格符、换行符、制表符等）并从第一个真正的字符开始读起，直到遇见下一处空白为止。 ◀86

如上所述，如果程序的输入是“Hello World!”（注意开头和结尾处的空格），则输出将是“Hello”，输出结果中没有任何空格。

和内置类型的输入输出操作一样，`string` 对象的此类操作也是返回运算符左侧的运算对象作为其结果。因此，多个输入或者多个输出可以连写在一起：

```
string s1, s2;
cin >> s1 >> s2;           // 把第一个输入读到 s1 中，第二个输入读到 s2 中
cout << s1 << s2 << endl; // 输出两个 string 对象
```

假设给上面这段程序输入与之前一样的内容“**Hello World!**”，输出将是“**HelloWorld!**”。

读取未知数量的 string 对象

1.4.3 节（第 13 页）的程序可以读入数量未知的整数，下面编写一个类似的程序用于读取 string 对象：

```
int main()
{
    string word;
    while (cin >> word)           // 反复读取，直至到达文件末尾
        cout << word << endl;      // 逐个输出单词，每个单词后面紧跟一个换行
    return 0;
}
```

在该程序中，读取的对象是 string 而非 int，但是 while 语句的条件部分和之前版本的程序是一样的。该条件负责在读取时检测流的情况，如果流有效，也就是说没遇到文件结束标记或非法输入，那么执行 while 语句内部的操作。此时，循环体将输出刚刚从标准输入读取的内容。重复若干次之后，一旦遇到文件结束标记或非法输入循环也就结束了。

使用 `getline` 读取一整行

有时我们希望能在最终得到的字符串中保留输入时的空白符，这时应该用 `getline` 函数代替原来的`>>`运算符。`getline` 函数的参数是一个输入流和一个 string 对象，函数从给定的输入流中读入内容，直到遇到换行符为止（注意换行符也被读进来了），然后把所读的内容存入到那个 string 对象中去（注意不存换行符）。`getline` 只要一遇到换行符就结束读取操作并返回结果，哪怕输入的一开始就是换行符也是如此。如果输入真的一开始就是换行符，那么所得的结果是个空 string。

和输入运算符一样，`getline` 也会返回它的流参数。因此既然输入运算符能作为判断的条件（参见 1.4.3 节，第 13 页），我们也能用 `getline` 的结果作为条件。例如，可以通过改写之前的程序让它一次输出一整行，而不再是每行输出一个词：

```
int main()
{
    string line;
    // 每次读入一整行，直至到达文件末尾
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

因为 `line` 中不包含换行符，所以我们手动地加上换行操作符。和往常一样，使用 `endl` 结束当前行并刷新显示缓冲区。



触发 `getline` 函数返回的那个换行符实际上被丢弃掉了，得到的 string 对象中并不包含该换行符。

string 的 `empty` 和 `size` 操作

顾名思义，`empty` 函数根据 string 对象是否为空返回一个对应的布尔值（参见第

2.1 节, 30 页)。和 Sales_item 类(参见 1.5.2 节, 第 20 页)的 isbn 成员一样, empty 也是 string 的一个成员函数。调用该函数的方法很简单, 只要使用点操作符指明是哪个对象执行了 empty 函数就可以了。

通过改写之前的程序, 可以做到只输出非空的行:

```
// 每次读入一整行, 遇到空行直接跳过
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

在上面的程序中, if 语句的条件部分使用了逻辑非运算符 (!), 它返回与其运算对象相反的结果。此例中, 如果 str 不为空则返回真。

size 函数返回 string 对象的长度(即 string 对象中字符的个数), 可以使用 size 88 函数只输出长度超过 80 个字符的行:

```
string line;
// 每次读入一整行, 输出其中超过 80 个字符的行
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

string::size_type 类型

对于 size 函数来说, 返回一个 int 或者如前面 2.1.1 节(第 31 页)所述的那样返回一个 unsigned 似乎都是合情合理的。但其实 size 函数返回的是一个 string::size_type 类型的值, 下面就对这种新的类型稍作解释。

string 类及其他大多数标准库类型都定义了几种配套的类型。这些配套类型体现了标准库类型与机器无关的特性, 类型 **size_type** 即是其中的一种。在具体使用的时候, 通过作用域操作符来表明名字 size_type 是在类 string 中定义的。

尽管我们不太清楚 string::size_type 类型的细节, 但有一点是肯定的: 它是一个无符号类型的值(参见 2.1.1 节, 第 30 页)而且能足够存放下任何 string 对象的大小。所有用于存放 string 类的 size 函数返回值的变量, 都应该是 string::size_type 类型的。

过去, string::size_type 这种类型有点儿神秘, 不太容易理解和使用。在 C++11 新标准中, 允许编译器通过 auto 或者 decltype(参见 2.5.2 节, 第 61 页)来推断变量的类型: C++ 11

```
auto len = line.size(); // len 的类型是 string::size_type
```

由于 size 函数返回的是一个无符号整型数, 因此切记, 如果在表达式中混用了带符号数和无符号数将可能产生意想不到的结果(参见 2.1.2 节, 第 33 页)。例如, 假设 n 是一个具有负值的 int, 则表达式 s.size()<n 的判断结果几乎肯定是 true。这是因为负值 n 会自动地转换成一个比较大的无符号值。



如果一条表达式中已经有了 size() 函数就不要再使用 int 了, 这样可以避免混用 int 和 unsigned 可能带来的问题。

比较 string 对象

string 类定义了几种用于比较字符串的运算符。这些比较运算符逐一比较 string

对象中的字符，并且对大小写敏感，也就是说，在比较时同一个字母的大写形式和小写形式是不同的。

相等性运算符（`==`和`!=`）分别检验两个 `string` 对象相等或不相等，`string` 对象相等意味着它们的长度相同而且所包含的字符也全都相同。关系运算符`<`、`<=`、`>`、`>=`分别检验一个 `string` 对象是否小于、小于等于、大于、大于等于另外一個 `string` 对象。上述这些运算符都依照（大小写敏感的）字典顺序：

- 89> 1. 如果两个 `string` 对象的长度不同，而且较短 `string` 对象的每个字符都与较长 `string` 对象对应位置上的字符相同，就说较短 `string` 对象小于较长 `string` 对象。
- 2. 如果两个 `string` 对象在某些对应的位置上不一致，则 `string` 对象比较的结果其实是 `string` 对象中第一对相异字符比较的结果。

下面是 `string` 对象比较的一个示例：

```
string str = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

根据规则 1 可判断，对象 `str` 小于对象 `phrase`；根据规则 2 可判断，对象 `slang` 既大于 `str` 也大于 `phrase`。

为 `string` 对象赋值

一般来说，在设计标准库类型时都力求在易用性上向内置类型看齐，因此大多数库类型都支持赋值操作。对于 `string` 类而言，允许把一个对象的值赋给另外一个对象：

```
string st1(10, 'c'), st2;// st1 的内容是 cccccccccc; st2 是一个空字符串
st1 = st2;                // 赋值：用 st2 的副本替换 st1 的内容
                           // 此时 st1 和 st2 都是空字符串
```

两个 `string` 对象相加

两个 `string` 对象相加得到一个新的 `string` 对象，其内容是把左侧的运算对象与右侧的运算对象串接而成。也就是说，对 `string` 对象使用加法运算符（`+`）的结果是一个新的 `string` 对象，它所包含的字符由两部分组成：前半部分是加号左侧 `string` 对象所含的字符、后半部分是加号右侧 `string` 对象所含的字符。另外，复合赋值运算符（`+=`）（参见 1.4.1 节，第 10 页）负责把右侧 `string` 对象的内容追加到左侧 `string` 对象的后面：

```
string s1 = "hello, ", s2 = "world\n";
string s3 = s1 + s2;      // s3 的内容是 hello, world\n
s1 += s2;                // 等价于 s1 = s1 + s2
```

字面值和 `string` 对象相加

如 2.1.2 节（第 33 页）所讲的，即使一种类型并非所需，我们也可以使用它，不过前提是该种类型可以自动转换成所需的类型。因为标准库允许把字符字面值和字符串字面值（参见 2.1.3 节，第 36 页）转换成 `string` 对象，所以在需要 `string` 对象的地方就可以使用这两种字面值来替代。利用这一点将之前的程序改写为如下形式：

```
string s1 = "hello", s2 = "world"; // 在 s1 和 s2 中都没有标点符号
string s3 = s1 + ", " + s2 + '\n';
```

当把 `string` 对象和字符串字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算符（`+`）的两侧的运算对象至少有一个是 `string`：

```
string s4 = s1 + ", " ; // 正确：把一个 string 对象和一个字面值相加
string s5 = "hello" + ", " ; // 错误：两个运算对象都不是 string
// 正确：每个加法运算符都有一个运算对象是 string
string s6 = s1 + ", " + "world";
string s7 = "hello" + ", " + s2; // 错误：不能把字面值直接相加
```

90

`s4` 和 `s5` 初始化时只用到了一个加法运算符，因此很容易判断是否合法。`s6` 的初始化形式之前没有出现过，但其实它的工作机理和连续输入连续输出（参见 1.2 节，第 6 页）是一样的，可以用如下的形式分组：

```
string s6 = (s1 + ", ") + "world";
```

其中子表达式 `s1 + ", "` 的结果是一个 `string` 对象，它同时作为第二个加法运算符的左侧运算对象，因此上述语句和下面的两个语句是等价的：

```
string tmp = s1 + ", " ; // 正确：加法运算符有一个运算对象是 string
s6 = tmp + "world"; // 正确：加法运算符有一个运算对象是 string
```

另一方面，`s7` 的初始化是非法的，根据其语义加上括号后就成了下面的形式：

```
string s7 = ("hello" + ", ") + s2; // 错误：不能把字面值直接相加
```

很容易看到，括号内的子表达式试图把两个字符串字面值加在一起，而编译器根本没法做到这一点，所以这条语句是错误的。



因为某些历史原因，也为了与 C 兼容，所以 C++ 语言中的字符串字面值并不是标准库类型 `string` 的对象。切记，字符串字面值与 `string` 是不同的类型。

3.2.2 节练习

练习 3.2： 编写一段程序从标准输入中一次读入一整行，然后修改该程序使其一次读入一个词。

练习 3.3： 请说明 `string` 类的输入运算符和 `getline` 函数分别是如何处理空白字符的。

练习 3.4： 编写一段程序读入两个字符串，比较其是否相等并输出结果。如果不相等，输出较大的那个字符串。改写上述程序，比较输入的两个字符串是否等长，如果不等长，输出长度较大的那个字符串。

练习 3.5： 编写一段程序从标准输入中读入多个字符串并将它们连接在一起，输出连接成的大字符串。然后修改上述程序，用空格把输入的多个字符串分隔开来。

3.2.3 处理 `string` 对象中的字符



我们经常需要单独处理 `string` 对象中的字符，比如检查一个 `string` 对象是否包含空白，或者把 `string` 对象中的字母改成小写，再或者查看某个特定的字符是否出现等。

这类处理的一个关键问题是如何获取字符本身。有时需要处理 `string` 对象中的每一个字符，另外一些时候则只需处理某个特定的字符，还有些时候遇到某个条件处理就要停

91

下来。以往的经验告诉我们，处理这些情况常常要涉及语言和库的很多方面。

另一个关键问题是想知道能改变某个字符的特性。在 `cctype` 头文件中定义了一组标准库函数处理这部分工作，表 3.3 列出了主要的函数名及其含义。

表 3.3: `cctype` 头文件中的函数

<code>isalnum(c)</code>	当 c 是字母或数字时为真
<code>isalpha(c)</code>	当 c 是字母时为真
<code>iscntrl(c)</code>	当 c 是控制字符时为真
<code>isdigit(c)</code>	当 c 是数字时为真
<code>isgraph(c)</code>	当 c 不是空格但可打印时为真
<code>islower(c)</code>	当 c 是小写字母时为真
<code>isprint(c)</code>	当 c 是可打印字符时为真（即 c 是空格或 c 具有可视形式）
<code>ispunct(c)</code>	当 c 是标点符号时为真（即 c 不是控制字符、数字、字母、可打印空白中的一种）
<code>isspace(c)</code>	当 c 是空白时为真（即 c 是空格、横向制表符、纵向制表符、回车符、换行符、进纸符中的一种）
<code>isupper(c)</code>	当 c 是大写字母时为真
<code>isxdigit(c)</code>	当 c 是十六进制数字时为真
<code>tolower(c)</code>	如果 c 是大写字母，输出对应的小写字母；否则原样输出 c
<code>toupper(c)</code>	如果 c 是小写字母，输出对应的大写字母；否则原样输出 c

建议：使用 C++ 版本的 C 标准库头文件

C++ 标准库中除了定义 C++ 语言特有的功能外，也兼容了 C 语言的标准库。C 语言的头文件形如 `name.h`，C++ 则将这些文件命名为 `cname`。也就是去掉了 `.h` 后缀，而在文件名 `name` 之前添加了字母 `c`，这里的 `c` 表示这是一个属于 C 语言标准库的头文件。

因此，`cctype` 头文件和 `ctype.h` 头文件的内容是一样的，只不过从命名规范上来讲更符合 C++ 语言的要求。特别的，在名为 `cname` 的头文件中定义的名字从属于命名空间 `std`，而定义在名为 `.h` 的头文件中的则不然。

一般来说，C++ 程序应该使用名为 `cname` 的头文件而不使用 `name.h` 的形式，标准库中的名字总能在命名空间 `std` 中找到。如果使用 `.h` 形式的头文件，程序员就不得不时刻牢记哪些是从 C 语言那儿继承过来的，哪些又是 C++ 语言所独有的。

处理每个字符？使用基于范围的 for 语句

C++ 11 如果想对 `string` 对象中的每个字符做点儿什么操作，目前最好的办法是使用 C++11 新标准提供的一种语句：范围 `for`（range `for`）语句。这种语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，其语法形式是：

```
for (declaration : expression)
    statement
```

其中，`expression` 部分是一个对象，用于表示一个序列。`declaration` 部分负责定义一个变量，该变量将被用于访问序列中的基础元素。每次迭代，`declaration` 部分的变量会被初始化为 `expression` 部分的下一个元素值。

一个 `string` 对象表示一个字符的序列，因此 `string` 对象可以作为范围 `for` 语句

中的 *expression* 部分。举一个简单的例子，我们可以使用范围 for 语句把 string 对象中的字符每行一个输出出来：

```
string str("some string");
// 每行输出 str 中的一个字符。
for (auto c : str)           // 对于 str 中的每个字符
    cout << c << endl;      // 输出当前字符，后面紧跟一个换行符
```

for 循环把变量 c 和 str 联系了起来，其中我们定义循环控制变量的方式与定义任意一个普通变量是一样的。此例中，通过使用 auto 关键字（参见 2.5.2 节，第 61 页）让编译器来决定变量 c 的类型，这里 c 的类型是 char。每次迭代，str 的下一个字符被拷贝给 c，因此该循环可以读作“对于字符串 str 中的每个字符 c，”执行某某操作。此例中的“某某操作”即输出一个字符，然后换行。

< 92

举个稍微复杂一点的例子，使用范围 for 语句和 ispunct 函数来统计 string 对象中标点符号的个数：

```
string s("Hello World!!!");
// punct_cnt 的类型和 s.size 的返回类型一样；参见 2.5.3 节（第 62 页）
decltype(s.size()) punct_cnt = 0;
// 统计 s 中标点符号的数量
for (auto c : s)           // 对于 s 中的每个字符
    if (ispunct(c))        // 如果该字符是标点符号
        ++punct_cnt;        // 将标点符号的计数值加 1
cout << punct_cnt
    << " punctuation characters in " << s << endl;
```

程序的输出结果将是：

```
3 punctuation characters in Hello World!!!
```

这里我们使用 decltype 关键字（参见 2.5.3 节，第 62 页）声明计数变量 punct_cnt，它的类型是 s.size 函数返回值的类型，也就是 string::size_type。使用范围 for 语句处理 string 对象中的每个字符并检查其是否是标点符号。如果是，使用递增运算符（参见 1.4.1 节，第 10 页）给计数变量加 1。最后，待范围 for 语句结束后输出统计结果。

< 93

使用范围 for 语句改变字符串中的字符

如果想要改变 string 对象中字符的值，必须把循环变量定义成引用类型（参见 2.3.1 节，第 45 页）。记住，所谓引用只是给定对象的一个别名，因此当使用引用作为循环控制变量时，这个变量实际上被依次绑定到了序列的每个元素上。使用这个引用，我们就能改变它绑定的字符。

新的例子不再是统计标点符号的个数了，假设我们想要把字符串改写为大写字母的形式。为了做到这一点可以使用标准库函数 toupper，该函数接收一个字符，然后输出其对应的大写形式。这样，为了把整个 string 对象转换成大写，只要对其中的每个字符调用 toupper 函数并将结果再赋给原字符就可以了：

```
string s("Hello World!!!");
// 转换成大写形式。
for (auto &c : s)           // 对于 s 中的每个字符（注意：c 是引用）
    c = toupper(c);          // c 是一个引用，因此赋值语句将改变 s 中字符的值
cout << s << endl;
```

上述代码的输出结果将是：

```
HELLO WORLD!!!
```

每次迭代时，变量 c 引用 string 对象 s 的下一个字符，赋值给 c 也就是在改变 s 中对应字符的值。因此当执行下面的语句时，

```
c = toupper(c); // c 是一个引用，因此赋值语句将改变 s 中字符的值
```

实际上改变了 c 绑定的字符的值。整个循环结束后，str 中的所有字符都变成了大写形式。

只处理一部分字符？

如果要处理 string 对象中的每一个字符，使用范围 for 语句是个好主意。然而，有时我们需要访问的只是其中一个字符，或者访问多个字符但遇到某个条件就要停下来。例如，同样是将字符改为大写形式，不过新的要求不再是对整个字符串都这样做，而仅仅把 string 对象中的第一个字母或第一个单词大写化。

要想访问 string 对象中的单个字符有两种方式：一种是使用下标，另外一种是使用迭代器，其中关于迭代器的内容将在 3.4 节（第 95 页）和第 9 章中介绍。

下标运算符 ([]) 接收的输入参数是 string::size_type 类型的值（参见 3.2.2 节，第 79 页），这个参数表示要访问的字符的位置；返回值是该位置上字符的引用。

string 对象的下标从 0 计起。如果 string 对象 s 至少包含两个字符，则 s[0] 是第 1 个字符、s[1] 是第 2 个字符、s[s.size()-1] 是最后一个字符。

string 对象的下标必须大于等于 0 而小于 s.size()。



使用超出此范围的下标将引发不可预知的结果，以此推断，使用下标访问空 string 也会引发不可预知的结果。

94

下标的值称作“下标”或“索引”，任何表达式只要它的值是一个整型值就能作为索引。不过，如果某个索引是带符号类型的值将自动转换成由 string::size_type（参见 2.1.2 节，第 33 页）表达的无符号类型。

下面的程序使用下标运算符输出 string 对象中的第一个字符：

```
if (!s.empty())           // 确保确实有字符需要输出
    cout << s[0] << endl;   // 输出 s 的第一个字符
```

在访问指定字符之前，首先检查 s 是否为空。其实不管什么时候只要对 string 对象使用了下标，都要确认在那个位置上确实有值。如果 s 为空，则 s[0] 的结果将是未定义的。

只要字符串不是常量（参见 2.4 节，第 53 页），就能为下标运算符返回的字符赋新值。例如，下面的程序将字符串的首字符改成了大写形式：

```
string s("some string");
if (!s.empty())           // 确保 s[0] 的位置确实有字符
    s[0] = toupper(s[0]);  // 为 s 的第一个字符赋一个新值
```

程序的输出结果将是：

```
Some string
```

使用下标执行迭代

另一个例子是把 s 的第一个词改成大写形式：

```
// 依次处理 s 中的字符直至我们处理完全部字符或者遇到一个空白
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
    s[index] = toupper(s[index]); // 将当前字符改成大写形式
```

程序的输出结果将是：

SOME string

在上述程序中，`for` 循环使用变量 `index` 作为 `s` 的下标，`index` 的类型是由 `decltype` 关键字决定的。首先把 `index` 初始化为 0，这样第一次迭代就会从 `s` 的首字符开始；之后每次迭代将 `index` 加 1 以得到 `s` 的下一个字符。循环体负责将当前的字母改写为大写形式。

`for` 语句的条件部分涉及一点新知识，该条件使用了逻辑与运算符（`&&`）。如果参与运算的两个运算对象都为真，则逻辑与结果为真；否则结果为假。对这个运算符来说最重要的一点是，C++语言规定只有当左侧运算对象为真时才会检查右侧运算对象的情况。如上例所示，这条规定确保了只有当下标取值在合理范围之内时才会真的用此下标去访问字符串。也就是说，只有在 `index` 达到 `s.size()` 之前才会执行 `s[index]`。随着 `index` 的增加，它永远也不可能超过 `s.size()` 的值，所以可以确保 `index` 比 `s.size()` 小。

提示：注意检查下标的合法性

95

使用下标时必须确保其在合理范围之内，也就是说，下标必须大于等于 0 而小于字符串的 `size()` 的值。一种简便易行的方法是，总是设下标的类型为 `string::size_type`，因为此类型是无符号数，可以确保下标不会小于 0。此时，代码只需保证下标小于 `size()` 的值就可以了。



C++标准并不要求标准库检测下标是否合法。一旦使用了一个超出范围的下标，就会产生不可预知的结果。

使用下标执行随机访问

在之前的示例中，我们让字符串的下标每次加 1 从而按顺序把所有字符改写成了大写形式。其实也能通过计算得到某个下标值，然后直接获取对应位置的字符，并不是每次都得从前往后依次访问。

例如，想要编写一个程序把 0 到 15 之间的十进制数转换成对应的十六进制形式，只需初始化一个字符串令其存放 16 个十六进制“数字”：

```
const string hexdigits = "0123456789ABCDEF"; // 可能的十六进制数字
cout << "Enter a series of numbers between 0 and 15"
     << " separated by spaces. Hit ENTER when finished: "
     << endl;
string result; // 用于保存十六进制的字符串
string::size_type n; // 用于保存从输入流读取的数
while (cin >> n)
    if (n < hexdigits.size()) // 忽略无效输入
        result += hexdigits[n]; // 得到对应的十六进制数字
```

```
cout << "Your hex number is: " << result << endl;
```

假设输入的内容如下：

```
12 0 5 15 8 15
```

程序的输出结果将是：

```
Your hex number is: C05F8F
```

上述程序的执行过程是这样的：首先初始化变量 `hexdigits` 令其存放从 0 到 F 的十六进制数字，注意我们把 `hexdigits` 声明成了常量（参见 2.4 节，第 53 页），这是因为在后面的程序中不打算再改变它的值。在循环内部使用输入值 `n` 作为 `hexdigits` 的下标，`hexdigits[n]` 的值就是 `hexdigits` 内位置 `n` 处的字符。例如，如果 `n` 是 15，则结果是 F；如果 `n` 是 12，则结果是 C，以此类推。把得到的十六进制数字添加到 `result` 内，最后一并输出。

无论何时用到字符串的下标，都应该注意检查其合法性。在上面的程序中，下标 `n` 是 `string::size_type` 类型，也就是无符号类型，所以 `n` 可以确保大于或等于 0。在实际使用时，还需检查 `n` 是否小于 `hexdigits` 的长度。

96

3.2.3 节练习

练习 3.6：编写一段程序，使用范围 `for` 语句将字符串内的所有字符用 X 替换。

练习 3.7：就上一题完成的程序而言，如果将循环控制变量的类型设为 `char` 将发生什么？先估计一下结果，然后实际编程进行验证。

练习 3.8：分别用 `while` 循环和传统的 `for` 循环重写第一题的程序，你觉得哪种形式更好呢？为什么？

练习 3.9：下面的程序有何作用？它合法吗？如果不合法，为什么？

```
string s;
cout << s[0] << endl;
```

练习 3.10：编写一段程序，读入一个包含标点符号的字符串，将标点符号去除后输出字符串剩余的部分。

练习 3.11：下面的范围 `for` 语句合法吗？如果合法，`c` 的类型是什么？

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```



3.3 标准库类型 `vector`

标准库类型 `vector` 表示对象的集合，其中所有对象的类型都相同。集合中的每个对象都有一个与之对应的索引，索引用于访问对象。因为 `vector` “容纳着”其他对象，所以它也常被称作容器（container）。第 II 部将对容器进行更为详细的介绍。

要想使用 `vector`，必须包含适当的头文件。在后续的例子中，都将假定做了如下 `using` 声明：

```
#include <vector>
using std::vector;
```

C++语言既有类模板（class template），也有函数模板，其中 `vector` 是一个类模板。只有对 C++有了相当深入的理解才能写出模板，事实上，我们直到第 16 章才会学习如何自定义模板。幸运的是，即使还不会创建模板，我们也可以先试着用用它。

模板本身不是类或函数，相反可以将模板看作为编译器生成类或函数编写的一份说明。编译器根据模板创建类或函数的过程称为实例化（instantiation），当使用模板时，需要指出编译器应把类或函数实例化成何种类型。

对于类模板来说，我们通过提供一些额外信息来指定模板到底实例化成什么样的类，需要提供哪些信息由模板决定。提供信息的方式总是这样：即在模板名字后面跟一对尖括号，在括号内放上信息。

以 `vector` 为例，提供的额外信息是 `vector` 内所存放对象的类型：

```
vector<int> ivec;           // ivec 保存 int 类型的对象
vector<Sales_item> Sales_vec; // 保存 Sales_item 类型的对象
vector<vector<string>> file; // 该向量的元素是 vector 对象
```

97

在上面的例子中，编译器根据模板 `vector` 生成了三种不同的类型：`vector<int>`、`vector<Sales_item>` 和 `vector<vector<string>>`。



`vector` 是模板而非类型，由 `vector` 生成的类型必须包含 `vector` 中元素的类型，例如 `vector<int>`。

C++11

`vector` 能容纳绝大多数类型的对象作为其元素，但是因为引用不是对象（参见 2.3.1 节，第 45 页），所以不存在包含引用的 `vector`。除此之外，其他大多数（非引用）内置类型和类类型都可以构成 `vector` 对象，甚至组成 `vector` 的元素也可以是 `vector`。

需要指出的是，在早期版本的 C++ 标准中如果 `vector` 的元素还是 `vector`（或者其他模板类型），则其定义的形式与现在的 C++11 新标准略有不同。过去，必须在外层 `vector` 对象的右尖括号和其元素类型之间添加一个空格，如应该写成 `vector<vector<int>>` 而非 `vector<vector<int>>`。



某些编译器可能仍需以老式的声明语句来处理元素为 `vector` 的 `vector` 对象，如 `vector<vector<int>>`。

3.3.1 定义和初始化 vector 对象



和任何一种类类型一样，`vector` 模板控制着定义和初始化向量的方法。表 3.4 列出了定义 `vector` 对象的常用方法。

表 3.4：初始化 `vector` 对象的方法

<code>vector<T> v1</code>	<code>v1</code> 是一个空 <code>vector</code> ，它潜在的元素是 <code>T</code> 类型的，执行默认初始化
<code>vector<T> v2(v1)</code>	<code>v2</code> 中包含有 <code>v1</code> 所有元素的副本
<code>vector<T> v2 = v1</code>	等价于 <code>v2(v1)</code> ， <code>v2</code> 中包含有 <code>v1</code> 所有元素的副本
<code>vector<T> v3(n, val)</code>	<code>v3</code> 包含了 <code>n</code> 个重复的元素，每个元素的值都是 <code>val</code>
<code>vector<T> v4(n)</code>	<code>v4</code> 包含了 <code>n</code> 个重复地执行了值初始化的对象
<code>vector<T> v5{a, b, c...}</code>	<code>v5</code> 包含了初始值个数的元素，每个元素被赋予相应的初始值
<code>vector<T> v5={a, b, c...}</code>	等价于 <code>v5(a, b, c...)</code>

可以默认初始化 `vector` 对象（参见 2.2.1 节，第 40 页），从而创建一个指定类型的空 `vector`：

```
vector<string> svec; // 默认初始化，svec 不含任何元素
```

看起来空 `vector` 好像没什么用，但是很快我们就会知道程序在运行时可以很高效地往 `vector` 对象中添加元素。事实上，最常见的方式就是先定义一个空 `vector`，然后当运行时获取到元素的值后再逐一添加。

当然也可以在定义 `vector` 对象时指定元素的初始值。例如，允许把一个 `vector` 对象的元素拷贝给另外一个 `vector` 对象。此时，新 `vector` 对象的元素就是原 `vector` 对象对应元素的副本。注意两个 `vector` 对象的类型必须相同：

```
vector<int> ivec; // 初始状态为空
// 在此处给 ivec 添加一些值
vector<int> ivec2(ivec); // 把 ivec 的元素拷贝给 ivec2
vector<int> ivec3 = ivec; // 把 ivec 的元素拷贝给 ivec3
vector<string> svec(ivec2); // 错误：svec 的元素是 string 对象，不是 int
```

98 ◀ 列表初始化 `vector` 对象

C++ 11 C++ 新标准还提供了另外一种为 `vector` 对象的元素赋初值的方法，即列表初始化（参见 2.2.1 节，第 39 页）。此时，用花括号括起来的 0 个或多个初始元素值被赋给 `vector` 对象：

```
vector<string> articles = {"a", "an", "the"};
```

上述 `vector` 对象包含三个元素：第一个是字符串“a”，第二个是字符串“an”，最后一个也是字符串“the”。

之前已经讲过，C++ 语言提供了几种不同的初始化方式（参见 2.2.1 节，第 39 页）。在大多数情况下这些初始化方式可以相互等价地使用，不过也并非一直如此。目前已经介绍过的两种例外情况是：其一，使用拷贝初始化时（即使用 = 时）（参见 3.2.1 节，第 76 页），只能提供一个初始值；其二，如果提供的是一个类内初始值（参见 2.6.1 节，第 64 页），则只能使用拷贝初始化或使用花括号的形式初始化。第三种特殊的要求是，如果提供的是初始元素值的列表，则只能把初始值都放在花括号里进行列表初始化，而不能放在圆括号里：

```
vector<string> v1{"a", "an", "the"}; // 列表初始化
vector<string> v2("a", "an", "the"); // 错误
```

创建指定数量的元素

还可以用 `vector` 对象容纳的元素数量和所有元素的统一初始值来初始化 `vector` 对象：

```
vector<int> ivec(10, -1); // 10 个 int 类型的元素，每个都被初始化为 -1
vector<string> svec(10, "hi!"); // 10 个 string 类型的元素，
// 每个都被初始化为 "hi!"
```

值初始化

通常情况下，可以只提供 `vector` 对象容纳的元素数量而不用略去初始值。此时库会创建一个值初始化的（value-initialized）元素初值，并把它赋给容器中的所有元素。这个初值由 `vector` 对象中元素的类型决定。

如果 `vector` 对象的元素是内置类型，比如 `int`，则元素初始值自动设为 0。如果元素是某种类类型，比如 `string`，则元素由类默认初始化：

```
vector<int> ivec(10);           // 10 个元素，每个都初始化为 0
vector<string> svec(10);        // 10 个元素，每个都是空 string 对象
```

对这种初始化的方式有两个特殊限制：其一，有些类要求必须明确地提供初始值（参见 2.2.1 节，第 40 页），如果 `vector` 对象中元素的类型不支持默认初始化，我们就必须提供初始的元素值。对这种类型的对象来说，只提供元素的数量而不设定初始值无法完成初始化工作。

其二，如果只提供了元素的数量而没有设定初始值，只能使用直接初始化：

```
vector<int> vi = 10; // 错误：必须使用直接初始化的形式指定向量大小
```

99

这里的 10 是用来说明如何初始化 `vector` 对象的，我们用它的本意是想创建含有 10 个值初始化了的元素的 `vector` 对象，而非把数字 10 “拷贝”到 `vector` 中。因此，此时不宜使用拷贝初始化，7.5.4 节（第 265 页）将对这一点做更详细的介绍。

列表初始值还是元素数量？



在某些情况下，初始化的真实含义依赖于传递初始值时用的是花括号还是圆括号。例如，用一个整数来初始化 `vector<int>` 时，整数的含义可能是 `vector` 对象的容量也可能元素的值。类似的，用两个整数来初始化 `vector<int>` 时，这两个整数可能一个是 `vector` 对象的容量，另一个是元素的初值，也可能它们是容量为 2 的 `vector` 对象中两个元素的初值。通过使用花括号或圆括号可以区分上述这些含义：

```
vector<int> v1(10);           // v1 有 10 个元素，每个的值都是 0
vector<int> v2{10};            // v2 有 1 个元素，该元素的值是 10

vector<int> v3(10, 1);         // v3 有 10 个元素，每个的值都是 1
vector<int> v4{10, 1};          // v4 有 2 个元素，值分别是 10 和 1
```

如果用的是圆括号，可以说提供的值是用来构造（construct）`vector` 对象的。例如，`v1` 的初始值说明了 `vector` 对象的容量；`v3` 的两个初始值则分别说明了 `vector` 对象的容量和元素的初值。

如果用的是花括号，可以表述成我们想列表初始化（list initialize）该 `vector` 对象。也就是说，初始化过程会尽可能地把花括号内的值当成是元素初始值的列表来处理，只有在无法执行列表初始化时才会考虑其他初始化方式。在上例中，给 `v2` 和 `v4` 提供的初始值都能作为元素的值，所以它们都会执行列表初始化，`vector` 对象 `v2` 包含一个元素而 `vector` 对象 `v4` 包含两个元素。

另一方面，如果初始化时使用了花括号的形式但是提供的值又不能用来列表初始化，就要考虑用这样的值来构造 `vector` 对象了。例如，要想列表初始化一个含有 `string` 对象的 `vector` 对象，应该提供能赋给 `string` 对象的初值。此时不难区分到底是要列表初始化 `vector` 对象的元素还是用给定的容量值来构造 `vector` 对象：

```
vector<string> v5{"hi"}; // 列表初始化：v5 有一个元素
vector<string> v6("hi"); // 错误：不能使用字符串字面值构建 vector 对象
vector<string> v7{10};      // v7 有 10 个默认初始化的元素
vector<string> v8{10, "hi"}; // v8 有 10 个值为"hi"的元素
```

100

尽管在上面的例子中除了第二条语句之外都用了花括号，但其实只有 `v5` 是列表初始化。要想列表初始化 `vector` 对象，花括号里的值必须与元素类型相同。显然不能用 `int` 初始化 `string` 对象，所以 `v7` 和 `v8` 提供的值不能作为元素的初始值。确认无法执行列表初始化后，编译器会尝试用默认值初始化 `vector` 对象。

3.3.1 节练习

练习 3.12: 下列 `vector` 对象的定义有不正确的吗？如果有，请指出来。对于正确的，描述其执行结果；对于不正确的，说明其错误的原因。

- (a) `vector<vector<int>> ivec;`
- (b) `vector<string> svec = ivec;`
- (c) `vector<string> svec(10, "null");`

练习 3.13: 下列的 `vector` 对象各包含多少个元素？这些元素的值分别是多少？

- | | |
|-----------------------------------------------------|-----------------------------------------------|
| (a) <code>vector<int> v1;</code> | (b) <code>vector<int> v2(10);</code> |
| (c) <code>vector<int> v3(10, 42);</code> | (d) <code>vector<int> v4{10};</code> |
| (e) <code>vector<int> v5{10, 42};</code> | (f) <code>vector<string> v6{10};</code> |
| (g) <code>vector<string> v7{10, "hi"};</code> | |



3.3.2 向 `vector` 对象中添加元素

对 `vector` 对象来说，直接初始化的方式适用于三种情况：初始值已知且数量较少、初始值是另一个 `vector` 对象的副本、所有元素的初始值都一样。然而更常见的情况是：创建一个 `vector` 对象时并不清楚实际所需的元素个数，元素的值也经常无法确定。还有些时候即使元素的初值已知，但如果这些值总量较大而各不相同，那么在创建 `vector` 对象的时候执行初始化操作也会显得过于烦琐。

举个例子，如果想创建一个 `vector` 对象令其包含从 0 到 9 共 10 个元素，使用列表初始化的方法很容易做到这一点；但如果 `vector` 对象包含的元素是从 0 到 99 或者从 0 到 999 呢？这时通过列表初始化把所有元素都一一罗列出来就不太合适了。对此例来说，更好的处理方法是先创建一个空 `vector`，然后在运行时再利用 `vector` 的成员函数 `push_back` 向其中添加元素。`push_back` 负责把一个值当成 `vector` 对象的尾元素“压到（push）”`vector` 对象的“尾端（back）”。例如：

```
101> vector<int> v2;           // 空 vector 对象
    for (int i = 0; i != 100; ++i)
        v2.push_back(i); // 依次把整数值放到 v2 尾端
    // 循环结束后 v2 有 100 个元素，值从 0 到 99
```

在上例中，尽管知道 `vector` 对象最后会包含 100 个元素，但在一开始还是把它声明成空 `vector`，在每次迭代时才顺序地把下一个整数作为 `v2` 的新元素添加给它。

同样的，如果直到运行时才能知道 `vector` 对象中元素的确切个数，也应该使用刚刚这种方法创建 `vector` 对象并为其赋值。例如，有时需要实时读入数据然后将其赋予 `vector` 对象：

```
// 从标准输入中读取单词，将其作为 vector 对象的元素存储
string word;
vector<string> text;           // 空 vector 对象
while (cin >> word) {
    text.push_back(word);     // 把 word 添加到 text 后面
}
```

和之前的例子一样，本例也是先创建一个空 `vector`，之后依次读入未知数量的值并保存到 `text` 中。

关键概念：vector 对象能高效增长

C++ 标准要求 `vector` 应该能在运行时高效快速地添加元素。因此既然 `vector` 对象能高效地增长，那么在定义 `vector` 对象的时候设定其大小也就没什么必要了，事实上如果这么做性能可能更差。只有一种例外情况，就是所有 (all) 元素的值都一样。一旦元素的值有所不同，更有效的办法是先定义一个空的 `vector` 对象，再在运行时向其中添加具体值。此外，9.4 节（第 317 页）将介绍，`vector` 还提供了方法，允许我们进一步提升动态添加元素的性能。

开始的时候创建空的 `vector` 对象，在运行时再动态添加元素，这一做法与 C 语言及其他大多数语言中内置数组类型的用法不同。特别是如果用惯了 C 或者 Java，可以预计在创建 `vector` 对象时顺便指定其容量是最好的。然而事实上，通常的情况是恰恰相反。

向 `vector` 对象添加元素蕴含的编程假定

由于能高效便捷地向 `vector` 对象中添加元素，很多编程工作被极大简化了。然而，这种简便性也伴随着一些对编写程序更高的要求：其中一条就是必须要确保所写的循环正确无误，特别是在循环有可能改变 `vector` 对象容量的时候。

随着对 `vector` 的更多使用，我们还会逐渐了解到其他一些隐含的要求，其中一条是现在就要指出的：如果循环体内部包含有向 `vector` 对象添加元素的语句，则不能使用范围 `for` 循环，具体原因将在 5.4.3 节（第 168 页）详细解释。



范围 `for` 语句体内不应改变其所遍历序列的大小。

3.3.2 节练习

102

练习 3.14： 编写一段程序，用 `cin` 读入一组整数并把它们存入一个 `vector` 对象。

练习 3.15： 改写上题的程序，不过这次读入的是字符串。

3.3.3 其他 `vector` 操作



除了 `push_back` 之外，`vector` 还提供了几种其他操作，大多数都和 `string` 的相关操作类似，表 3.5 列出了其中比较重要的一些。

表 3.5: `vector` 支持的操作

<code>v.empty()</code>	如果 <code>v</code> 不含有任何元素，返回真；否则返回假
<code>v.size()</code>	返回 <code>v</code> 中元素的个数
<code>v.push_back(t)</code>	向 <code>v</code> 的尾端添加一个值为 <code>t</code> 的元素
<code>v[n]</code>	返回 <code>v</code> 中第 <code>n</code> 个位置上元素的引用
<code>v1 = v2</code>	用 <code>v2</code> 中元素的拷贝替换 <code>v1</code> 中的元素
<code>v1 = {a, b, c...}</code>	用列表中元素的拷贝替换 <code>v1</code> 中的元素
<code>v1 == v2</code>	<code>v1</code> 和 <code>v2</code> 相等当且仅当它们的元素数量相同且对应位置的元素值都相同
<code>v1 != v2</code>	
<code><, <=, >, >=</code>	顾名思义，以字典顺序进行比较

访问 `vector` 对象中元素的方法和访问 `string` 对象中字符的方法差不多，也是通过元素在 `vector` 对象中的位置。例如，可以使用范围 `for` 语句处理 `vector` 对象中的所有元素：

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)           // 对于 v 中的每个元素（注意：i 是一个引用）
    i *= i;                 // 求元素值的平方
for (auto i : v)            // 对于 v 中的每个元素
    cout << i << " ";      // 输出该元素
cout << endl;
```

第一个循环把控制变量 `i` 定义成引用类型，这样就能通过 `i` 给 `v` 的元素赋值，其中 `i` 的类型由 `auto` 关键字指定。这里用到了一种新的复合赋值运算符（参见 1.4.1 节，第 10 页）。如我们所知，`+=` 把左侧运算对象和右侧运算对象相加，结果存入左侧运算对象；类似的，`*=` 把左侧运算对象和右侧运算对象相乘，结果存入左侧运算对象。最后，第二个循环输出所有元素。

`vector` 的 `empty` 和 `size` 两个成员与 `string` 的同名成员（参见 3.2.2 节，第 78 页）功能完全一致：`empty` 检查 `vector` 对象是否包含元素然后返回一个布尔值；`size` 则返回 `vector` 对象中元素的个数，返回值的类型是由 `vector` 定义的 `size_type` 类型。



要使用 `size_type`，需首先指定它是由哪种类型定义的。`vector` 对象的类型总是包含着元素的类型（参见 3.3 节，第 87 页）：

<code>vector<int>::size_type</code>	<code>// 正确</code>
<code>vector::size_type</code>	<code>// 错误</code>

各个相等性运算符和关系运算符也与 `string` 的相应运算符（参见 3.2.2 节，第 79 页）功能一致。两个 `vector` 对象相等当且仅当它们所含的元素个数相同，而且对应位置的元素值也相同。关系运算符依照字典顺序进行比较：如果两个 `vector` 对象的容量不同，但是在相同位置上的元素值都一样，则元素较少的 `vector` 对象小于元素较多的 `vector` 对象；若元素的值有区别，则 `vector` 对象的大小关系由第一对相异的元素值的大小关系决定。

103

只有当元素的值可比较时，`vector` 对象才能被比较。一些类，如 `string` 等，确实定义了自己的相等性运算符和关系运算符；另外一些，如 `Sales_item` 类支持的运算已经全都罗列在 1.5.1 节（第 17 页）中了，显然并不支持相等性判断和关系运算等操作。因此，不能比较两个 `vector<Sales_item>` 对象。

计算 `vector` 内对象的索引

使用下标运算符（参见 3.2.3 节，第 84 页）能获取到指定的元素。和 `string` 一样，`vector` 对象的下标也是从 0 开始计起，下标的类型是相应的 `size_type` 类型。只要 `vector` 对象不是一个常量，就能向下标运算符返回的元素赋值。此外，如 3.2.3 节（第 85 页）所述的那样，也能通过计算得到 `vector` 内对象的索引，然后直接获取索引位置上的元素。

举个例子，假设有一组成绩的集合，其中成绩的取值是从 0 到 100。以 10 分为一个分数段，要求统计各个分数段各有多少个成绩。显然，从 0 到 100 总共有 101 种可能的成绩取值，这些成绩分布在 11 个分数段上：每 10 个分数构成一个分数段，这样的分数段有 10 个，额外还有一个分数段表示满分 100 分。这样第一个分数段将统计成绩在 0 到 9 之间的数量；第二个分数段将统计成绩在 10 到 19 之间的数量，以此类推。最后一个分数段统计满分 100 分的数量。

按照上面的描述，如果输入的成绩如下：

```
42 65 95 100 39 67 95 76 88 76 83 92 76 93
```

则输出的结果应该是：

```
0 0 0 1 1 0 2 3 2 4 1
```

结果显示：成绩在 30 分以下的没有、30 分至 39 分有 1 个、40 分至 49 分有 1 个、50 分至 59 分没有、60 分至 69 分有 2 个、70 分至 79 分有 3 个、80 分至 89 分有 2 个、90 分至 99 分有 4 个，还有 1 个是满分。

在具体实现时使用一个含有 11 个元素的 `vector` 对象，每个元素分别用于统计各个分数段上出现的成绩个数。对于某个成绩来说，将其除以 10 就能得到对应的分数段索引。注意：两个整数相除，结果还是整数，余数部分被自动忽略掉了。例如， $42/10=4$ 、 $65/10=6$ 、 $100/10=10$ 等。一旦计算得到了分数段索引，就能用它作为 `vector` 对象的下标，进而获取该分数段的计数值并加 1：

```
// 以 10 分为一个分数段统计成绩的数量：0~9, 10~19, ..., 90~99, 100
vector<unsigned> scores(11, 0); // 11 个分数段，全都初始化为 0
unsigned grade;
while (cin >> grade) { // 读取成绩
    if (grade <= 100) // 只处理有效的成绩
        ++scores[grade/10]; // 将对应分数段的计数值加 1
}
```

在上面的程序中，首先定义了一个 `vector` 对象存放各个分数段上成绩的数量。此例中，由于初始状态下每个元素的值都相同，所以我们为 `vector` 对象申请了 11 个元素，并把所有元素的初始值都设为 0。`while` 语句的条件部分负责读入成绩，在循环体内部首先检查读入的成绩是否合法（即是否小于等于 100 分），如果合法，将成绩对应的分数段的计数值加 1。

执行计数值累加的那条语句很好地体现了 C++ 程序代码的简洁性。表达式

```
++scores[grade/10]; // 将当前分数段的计数值加 1
```

等价于

```
auto ind = grade/10; // 得到分数段索引
scores[ind] = scores[ind] + 1; // 将计数值加 1
```

上述语句的含义是：用 `grade` 除以 10 来计算成绩所在的分数段，然后将所得的结果作为变量 `scores` 的下标。通过运行下标运算获取该分数段对应的计数值，因为新出现了一个属于该分数段的成绩，所以将计数值加 1。

如前所述，使用下标的时候必须清楚地知道它是否在合理范围之内（参见 3.2.3 节，第 85 页）。在这个程序里，我们事先确认了输入的成绩确实在 0 到 100 之间，这样计算所得的下标就一定在 0 到 10 之间，属于 0 到 `scores.size()-1` 规定的有效范围，一定是合法的。

不能用下标形式添加元素

刚接触 C++ 语言的程序员也许会认为可以通过 `vector` 对象的下标形式来添加元素，事实并非如此。下面的代码试图为 `vector` 对象 `ivec` 添加 10 个元素：

```
vector<int> ivec; // 空 vector 对象
```

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix; // 严重错误：ivec 不包含任何元素
```

然而，这段代码是错误的：ivec 是一个空 vector，根本不包含任何元素，当然也就不能通过下标去访问任何元素！如前所述，正确的方法是使用 push_back：

105 >

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // 正确：添加一个新元素，该元素的值是 ix
```



vector 对象（以及 string 对象）的下标运算符可用于访问已存在的元素，而不能用于添加元素。

提示：只能对可知已存在的元素执行下标操作！

关于下标必须明确的一点是：只能对可知已存在的元素执行下标操作。例如，

```
vector<int> ivec;           // 空 vector 对象
cout << ivec[0];           // 错误：ivec 不包含任何元素

vector<int> ivec2(10);     // 含有 10 个元素的 vector 对象
cout << ivec2[10];         // 错误：ivec2 元素的合法索引是从 0 到 9
```

试图用下标的形式去访问一个不存在的元素将引发错误，不过这种错误不会被编译器发现，而是在运行时产生一个不可预知的值。

不幸的是，这种通过下标访问不存在的元素的行为非常常见，而且会产生很严重的后果。所谓的缓冲区溢出（buffer overflow）指的就是这类错误，这也是导致 PC 及其他设备上应用程序出现安全问题的一个重要原因。



确保下标合法的一种有效手段就是尽可能使用范围 for 语句。

3.3.3 节练习

练习 3.16：编写一段程序，把练习 3.13 中 vector 对象的容量和具体内容输出出来。检验你之前的回答是否正确，如果不对，回过头重新学习 3.3.1 节（第 87 页）直到弄明白错在何处为止。

练习 3.17：从 cin 读入一组词并把它们存入一个 vector 对象，然后设法把所有词都改写为大写形式。输出改变后的结果，每个词占一行。

练习 3.18：下面的程序合法吗？如果不合法，你准备如何修改？

```
vector<int> ivec;
ivec[0] = 42;
```

练习 3.19：如果想定义一个含有 10 个元素的 vector 对象，所有元素的值都是 42，请列举出三种不同的实现方法。哪种方法更好呢？为什么？

练习 3.20：读入一组整数并把它们存入一个 vector 对象，将每对相邻整数的和输出出来。改写你的程序，这次要求先输出第 1 个和最后 1 个元素的和，接着输出第 2 个和倒数第 2 个元素的和，以此类推。

3.4 迭代器介绍



我们已经知道可以使用下标运算符来访问 `string` 对象的字符或 `vector` 对象的元素，还有另外一种更通用的机制也可以实现同样的目的，这就是 **迭代器**（iterator）。在第 II 部分中将要介绍，除了 `vector` 之外，标准库还定义了其他几种容器。所有标准库容器都可以使用迭代器，但是其中只有少数几种才同时支持下标运算符。严格来说，`string` 对象不属于容器类型，但是 `string` 支持很多与容器类型类似的操作。`vector` 支持下标运算符，这点和 `string` 一样；`string` 支持迭代器，这也和 `vector` 是一样的。

106

类似于指针类型（参见 2.3.2 节，第 47 页），迭代器也提供了对对象的间接访问。就迭代器而言，其对象是容器中的元素或者 `string` 对象中的字符。使用迭代器可以访问某个元素，迭代器也能从一个元素移动到另外一个元素。迭代器有有效和无效之分，这一点和指针差不多。有效的迭代器或者指向某个元素，或者指向容器中尾元素的下一位置；其他所有情况都属于无效。

3.4.1 使用迭代器



和指针不一样的是，获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。比如，这些类型都拥有名为 `begin` 和 `end` 的成员，其中 `begin` 成员负责返回指向第一个元素（或第一个字符）的迭代器。如有下述语句：

```
// 由编译器决定 b 和 e 的类型；参见 2.5.2 节（第 61 页）
// b 表示 v 的第一个元素，e 表示 v 尾元素的下一位置
auto b = v.begin(), e = v.end(); // b 和 e 的类型相同
```

`end` 成员则负责返回指向容器（或 `string` 对象）“尾元素的下一位置（one past the end）”的迭代器，也就是说，该迭代器指示的是容器的一个本不存在的“尾后（off the end）”元素。这样的迭代器没什么实际含义，仅是个标记而已，表示我们已经处理完了容器中的所有元素。`end` 成员返回的迭代器常被称作 **尾后迭代器**（off-the-end iterator）或者简称为尾迭代器（end iterator）。特殊情况下如果容器为空，则 `begin` 和 `end` 返回的是同一个迭代器。

 Note

如果容器为空，则 `begin` 和 `end` 返回的是同一个迭代器，都是尾后迭代器。

一般来说，我们不清楚（不在意）迭代器准确的类型到底是什么。在上面的例子中，使用 `auto` 关键字定义变量 `b` 和 `e`（参见 2.5.2 节，第 61 页），这两个变量的类型也就是 `begin` 和 `end` 的返回值类型，第 97 页将对相关内容做更详细的介绍。

迭代器运算符

表 3.6 列举了迭代器支持的一些运算。使用 `==` 和 `!=` 来比较两个合法的迭代器是否相等，如果两个迭代器指向的元素相同或者都是同一个容器的尾后迭代器，则它们相等；否则就说这两个迭代器不相等。

表 3.6: 标准容器迭代器的运算符

*iter	返回迭代器 iter 所指元素的引用
iter->mem	解引用 iter 并获取该元素的名为 mem 的成员，等价于 (*iter).mem
++iter	令 iter 指示容器中的下一个元素
--iter	令 iter 指示容器中的上一个元素
iter1 == iter2	判断两个迭代器是否相等（不相等），如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之，不相等
iter1 != iter2	

和指针类似，也能通过解引用迭代器来获取它所指示的元素，执行解引用的迭代器必须合法并确实指示着某个元素（参见 2.3.2 节，第 48 页）。试图解引用一个非法迭代器或者尾后迭代器都是未被定义的行为。

举个例子，3.2.3 节（第 84 页）中的程序利用下标运算符把 string 对象的第一个字母改为了大写形式，下面利用迭代器实现同样的功能：

```
string s("some string");
if (s.begin() != s.end()) { // 确保 s 非空
    auto it = s.begin(); // it 表示 s 的第一个字符
    *it = toupper(*it); // 将当前字符改成大写形式
}
```

本例和原来的程序一样，首先检查 s 是否为空，显然通过检查 begin 和 end 返回的结果是否一致就能做到这一点。如果返回的结果一样，说明 s 为空；如果返回的结果不一样，说明 s 不为空，此时 s 中至少包含一个字符。

我们在 if 内部，声明了一个迭代器变量 it 并把 begin 返回的结果赋给它，这样就得到了指示 s 中第一个字符的迭代器，接下来通过解引用运算符将第一个字符更改为大写形式。和原来的程序一样，输出结果将是：

Some string

将迭代器从一个元素移动到另外一个元素

迭代器使用递增（++）运算符（参见 1.4.1 节，第 11 页）来从一个元素移动到下一个元素。从逻辑上来说，迭代器的递增和整数的递增类似，整数的递增是在整数值上“加 1”，迭代器的递增则是将迭代器“向前移动一个位置”。



因为 end 返回的迭代器并不实际指示某个元素，所以不能对其进行递增或解引用的操作。

之前有一个程序把 string 对象中第一个单词改写为大写形式，现在利用迭代器及其递增运算符可以实现相同的功能：

```
// 依次处理 s 的字符直至我们处理完全部字符或者遇到空白
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // 将当前字符改成大写形式
```

和 3.2.3 节（第 84 页）的那个程序一样，上面的循环也是遍历 s 的字符直到遇到空白字符为止，只不过之前的程序用的是下标运算符，现在这个程序用的是迭代器。

循环首先用 s.begin 的返回值来初始化 it，意味着 it 指示的是 s 中的第一个字符（如果有的话）。条件部分检查是否已到达 s 的尾部，如果尚未到达，则将 it 解引用的结

果传入 `isspace` 函数检查是否遇到了空白。每次迭代的最后，执行 `++it` 令迭代器前移一个位置以访问 `s` 的下一个字符。

循环体内部和上一个程序 `if` 语句内的最后一句话一样，先解引用 `it`，然后将结果传入 `toupper` 函数得到该字母对应的大写形式，再把这个大写字母重新赋值给 `it` 所指示的字符。

关键概念：泛型编程

原来使用 C 或 Java 的程序员在转而使用 C++ 语言之后，会对 `for` 循环中使用 `!=` 而非 `<` 进行判断有点儿奇怪，比如上面的这个程序以及 85 页的那个。C++ 程序员习惯性地使用 `!=`，其原因和他们更愿意使用迭代器而非下标的原因一样：因为这种编程风格在标准库提供的所有容器上都有效。

之前已经说过，只有 `string` 和 `vector` 等一些标准库类型有下标运算符，而并非全都如此。与之类似，所有标准库容器的迭代器都定义了 `==` 和 `!=`，但是它们中的大多数都没有定义 `<` 运算符。因此，只要我们养成使用迭代器和 `!=` 的习惯，就不用太在意用的到底是哪种容器类型。

迭代器类型

就像不知道 `string` 和 `vector` 的 `size_type` 成员（参见 3.2.2 节，第 79 页）到底是什么类型一样，一般来说我们也不知道（其实是无须知道）迭代器的精确类型。而实际上，那些拥有迭代器的标准库类型使用 `iterator` 和 `const_iterator` 来表示迭代器的类型：

```
vector<int>::iterator it;      // it 能读写 vector<int> 的元素  
string::iterator it2;         // it2 能读写 string 对象中的字符  
  
vector<int>::const_iterator it3; // it3 只能读元素，不能写元素  
string::const_iterator it4;    // it4 只能读字符，不能写字符  
  
const_iterator 和常量指针（参见 2.4.2 节，第 56 页）差不多，能读取但不能修改它所指的元素值。相反，iterator 的对象可读可写。如果 vector 对象或 string 对象是一个常量，只能使用 const_iterator；如果 vector 对象或 string 对象不是常量，那么既能使用 iterator 也能使用 const_iterator。
```

术语：迭代器和迭代器类型

< 109

迭代器这个名词有三种不同的含义：可能是迭代器概念本身，也可能是指容器定义的迭代器类型，还可能是指某个迭代器对象。

重点是理解存在一组概念上相关的类型，我们认定某个类型是迭代器当且仅当它支持一套操作，这套操作使得我们能访问容器的元素或者从某个元素移动到另外一个元素。

每个容器类定义了一个名为 `iterator` 的类型，该类型支持迭代器概念所规定的一套操作。

begin 和 end 运算符

`begin` 和 `end` 返回的具体类型由对象是否是常量决定，如果对象是常量，`begin` 和 `end` 返回 `const_iterator`；如果对象不是常量，返回 `iterator`：

```
vector<int> v;
```

```
const vector<int> cv;
auto it1 = v.begin();    // it1 的类型是 vector<int>::iterator
auto it2 = cv.begin();   // it2 的类型是 vector<int>::const_iterator
```

有时候这种默认的行为并非我们所要。在 6.2.3 节（第 191 页）中将会看到，如果对象只需读操作而无须写操作的话最好使用常量类型（比如 `const_iterator`）。为了便于专门得到 `const_iterator` 类型的返回值，C++11 新标准引入了两个新函数，分别是 `cbegin` 和 `cend`：

```
auto it3 = v.cbegin(); // it3 的类型是 vector<int>::const_iterator
```

类似于 `begin` 和 `end`，上述两个新函数也分别返回指示容器第一个元素或最后元素下一位置的迭代器。有所不同的是，不论 `vector` 对象（或 `string` 对象）本身是否是常量，返回值都是 `const_iterator`。

结合解引用和成员访问操作

解引用迭代器可获得迭代器所指的对象，如果该对象的类型恰好是类，就有可能希望进一步访问它的成员。例如，对于一个由字符串组成的 `vector` 对象来说，要想检查其元素是否为空，令 `it` 是该 `vector` 对象的迭代器，只需检查 `it` 所指字符串是否为空就可以了，其代码如下所示：

```
(*it).empty()
```

注意，`(*it).empty()` 中的圆括号必不可少，具体原因将在 4.1.2 节（第 121 页）介绍，该表达式的含义是先对 `it` 解引用，然后解引用的结果再执行点运算符（参见 1.5.2 节，第 20 页）。如果不加圆括号，点运算符将由 `it` 来执行，而非 `it` 解引用的结果：

```
(*it).empty()    // 解引用 it，然后调用结果对象的 empty 成员
*it.empty()      // 错误：试图访问 it 的名为 empty 的成员，但 it 是个迭代器，
                  // 没有 empty 成员
```

110 上面第二个表达式的含义是从名为 `it` 的对象中寻找其 `empty` 成员，显然 `it` 是一个迭代器，它没有哪个成员是叫 `empty` 的，所以第二个表达式将发生错误。

为了简化上述表达式，C++语言定义了箭头运算符（`->`）。箭头运算符把解引用和成员访问两个操作结合在一起，也就是说，`it->mem` 和 `(*it).mem` 表达的意思相同。

例如，假设用一个名为 `text` 的字符串向量存放文本文件中的数据，其中的元素或者是一句话或者是一个用于表示段落分隔的空字符串。如果要输出 `text` 中第一段的内容，可以利用迭代器写一个循环令其遍历 `text`，直到遇到空字符串的元素为止：

```
// 依次输出 text 的每一行直至遇到第一个空白行为止
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

我们首先初始化 `it` 令其指向 `text` 的第一个元素，循环重复执行直至处理完了 `text` 的所有元素或者发现某个元素为空。每次迭代时只要发现还有元素并且尚未遇到空元素，就输出当前正在处理的元素。值得注意的是，因为循环从头到尾只是读取 `text` 的元素而未向其中写值，所以使用了 `cbegin` 和 `cend` 来控制整个迭代过程。

某些对 `vector` 对象的操作会使迭代器失效

3.3.2 节（第 90 页）曾经介绍过，虽然 `vector` 对象可以动态地增长，但是也会有一

些副作用。已知的一个限制是不能在范围 `for` 循环中向 `vector` 对象添加元素。另外一个限制是任何一种可能改变 `vector` 对象容量的操作，比如 `push_back`，都会使该 `vector` 对象的迭代器失效。9.3.6 节（第 315 页）将详细解释迭代器是如何失效的。



谨记，但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。

WARNING

3.4.1 节练习

练习 3.21：请使用迭代器重做 3.3.3 节（第 94 页）的第一个练习。

练习 3.22：修改之前那个输出 `text` 第一段的程序，首先把 `text` 的第一段全都改成大写形式，然后再输出它。

练习 3.23：编写一段程序，创建一个含有 10 个整数的 `vector` 对象，然后使用迭代器将所有元素的值都变成原来的两倍。输出 `vector` 对象的内容，检验程序是否正确。

3.4.2 迭代器运算



迭代器的递增运算令迭代器每次移动一个元素，所有的标准库容器都有支持递增运算的迭代器。类似的，也能用`==`和`!=`对任意标准库类型的两个有效迭代器（参见 3.4 节，第 95 页）进行比较。

◀ 111

`string` 和 `vector` 的迭代器提供了更多额外的运算符，一方面可使得迭代器的每次移动跨过多个元素，另外也支持迭代器进行关系运算。所有这些运算被称作**迭代器运算** (iterator arithmetic)，其细节由表 3.7 列出。

表 3.7：`vector` 和 `string` 迭代器支持的运算

<code>iter + n</code>	迭代器加上一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
<code>iter - n</code>	迭代器减去一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向后移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
<code>iter1 += n</code>	迭代器加法的复合赋值语句，将 <code>iter1</code> 加 <code>n</code> 的结果赋给 <code>iter1</code>
<code>iter1 -= n</code>	迭代器减法的复合赋值语句，将 <code>iter1</code> 减 <code>n</code> 的结果赋给 <code>iter1</code>
<code>iter1 - iter2</code>	两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动差值个元素后将得到左侧的迭代器。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置
<code>>、>=、<、<=</code>	迭代器的关系运算符，如果某迭代器指向的容器位置在另一个迭代器所指位置之前，则说前者小于后者。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置

迭代器的算术运算

可以令迭代器和一个整数值相加（或相减），其返回值是向前（或向后）移动了若干个位置的迭代器。执行这样的操作时，结果迭代器或者指示原 `vector` 对象（或 `string` 对象）内的一个元素，或者指示原 `vector` 对象（或 `string` 对象）尾元素的下一位置。

举个例子，下面的代码得到一个迭代器，它指向某 `vector` 对象中间位置的元素：

```
// 计算得到最接近 vi 中间元素的一个迭代器
auto mid = vi.begin() + vi.size() / 2;
```

如果 `vi` 有 20 个元素，`vi.size() / 2` 得 10，此例中即令 `mid` 等于 `vi.begin() + 10`。已知下标从 0 开始，则迭代器所指的元素是 `vi[10]`，也就是从首元素开始向前相隔 10 个位置的那个元素。

对于 `string` 或 `vector` 的迭代器来说，除了判断是否相等，还能使用关系运算符(<、<=、>、>=) 对其进行比较。参与比较的两个迭代器必须合法而且指向的是同一个容器的元素（或者尾元素的下一位置）。例如，假设 `it` 和 `mid` 是同一个 `vector` 对象的两个迭代器，可以用下面的代码来比较它们所指的位置孰前孰后：

```
if (it < mid)
    // 处理 vi 前半部分的元素
```

112

只要两个迭代器指向的是同一个容器中的元素或者尾元素的下一位置，就能将其相减，所得结果是两个迭代器的距离。所谓距离指的是右侧的迭代器向前移动多少位置就能追上左侧的迭代器，其类型是名为 `difference_type` 的带符号整型数。`string` 和 `vector` 都定义了 `difference_type`，因为这个距离可正可负，所以 `difference_type` 是带符号类型的。

使用迭代器运算

使用迭代器运算的一个经典算法是二分搜索。二分搜索从有序序列中寻找某个给定的值。二分搜索从序列中间的位置开始搜索，如果中间位置的元素正好就是要找的元素，搜索完成；如果不是，假如该元素小于要找的元素，则在序列的后半部分继续搜索；假如该元素大于要找的元素，则在序列的前半部分继续搜索。在缩小的范围内计算一个新的中间元素并重复之前的过程，直至最终找到目标或者没有元素可供继续搜索。

下面的程序使用迭代器完成了二分搜索：

```
// text 必须是有序的
// beg 和 end 表示我们搜索的范围
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg) / 2; // 初始状态下的中间点
// 当还有元素尚未检查并且我们还没有找到 sought 时执行循环
while (mid != end && *mid != sought) {
    if (sought < *mid) // 我们要找的元素在前半部分吗？
        end = mid; // 如果是，调整搜索范围使得忽略掉后半部分
    else
        beg = mid + 1; // 我们要找的元素在后半部分
    mid = beg + (end - beg) / 2; // 新的中间点
}
```

程序一开始定义了三个迭代器：`beg` 指向搜索范围内的第一个元素、`end` 指向尾元素的下一位置、`mid` 指向中间的那个元素。初始状态下，搜索范围是名为 `text` 的 `vector<string>` 的全部范围。

循环部分先检查搜索范围是否为空，如果 `mid` 和 `end` 的当前值相等，说明已经找遍了所有元素。此时条件不满足，循环终止。当搜索范围不为空时，可知 `mid` 指向了某个元素，检查该元素是否就是我们所要搜索的，如果是，也终止循环。

当进入到循环体内部后，程序通过某种规则移动 beg 或者 end 来缩小搜索的范围。如果 mid 所指的元素比要找的元素 sought 大，可推测若 text 含有 sought，则必出现在 mid 所指元素的前面。此时，可以忽略 mid 后面的元素不再查找，并把 mid 赋给 end 即可。另一种情况，如果 *mid 比 sought 小，则要找的元素必出现在 mid 所指元素的后面。此时，通过令 beg 指向 mid 的下一个位置即可改变搜索范围。因为已经验证过 mid 不是我们要找的对象，所以在接下来的搜索中不必考虑它。

循环过程终止时，mid 或者等于 end 或者指向要找的元素。如果 mid 等于 end，说明 text 中没有我们要找的元素。

3.4.2 节练习

113

练习 3.24：请使用迭代器重做 3.3.3 节（第 94 页）的最后一个练习。

练习 3.25：3.3.3 节（第 93 页）划分分数段的程序是使用下标运算符实现的，请利用迭代器改写该程序并实现完全相同的功能。

练习 3.26：在 100 页的二分搜索程序中，为什么用的是 $mid = beg + (end - beg) / 2$ ，而非 $mid = (beg + end) / 2$ ？

3.5 数组

数组是一种类似于标准库类型 `vector`（参见 3.3 节，第 86 页）的数据结构，但是在性能和灵活性的权衡上又与 `vector` 有所不同。与 `vector` 相似的地方是，数组也是存放类型相同的对象的容器，这些对象本身没有名字，需要通过其所在位置访问。与 `vector` 不同的地方是，数组的大小确定不变，不能随意向数组中增加元素。因为数组的大小固定，因此对某些特殊的应用来说程序的运行时性能较好，但是相应地也损失了一些灵活性。



如果不清楚元素的确切个数，请使用 `vector`。

Tip

3.5.1 定义和初始化内置数组

数组是一种复合类型（参见 2.3 节，第 45 页）。数组的声明形如 `a[d]`，其中 `a` 是数组的名字，`d` 是数组的维度。维度说明了数组中元素的个数，因此必须大于 0。数组中元素的个数也属于数组类型的一部分，编译的时候维度应该是已知的。也就是说，维度必须是一个常量表达式（参见 2.4.4 节，第 58 页）：

```
unsigned cnt = 42;           // 不是常量表达式
constexpr unsigned sz = 42;   // 常量表达式，关于 constexpr，参见 2.4.4 节（第 59 页）
int arr[10];                 // 含有 10 个整数的数组
int *parr[sz];               // 含有 42 个整型指针的数组
string bad[cnt];             // 错误：cnt 不是常量表达式
string strs[get_size()];     // 当 get_size 是 constexpr 时正确；否则错误
```

默认情况下，数组的元素被默认初始化（参见 2.2.1 节，第 40 页）。



和内置类型的变量一样，如果在函数内部定义了某种内置类型的数组，那么默认初始化会令数组含有未定义的值。

定义数组的时候必须指定数组的类型，不允许用 auto 关键字由初始值的列表推断类型。另外和 vector 一样，数组的元素应为对象，因此不存在引用的数组。

114 显式初始化数组元素

可以对数组的元素进行列表初始化（参见 3.3.1 节，第 88 页），此时允许忽略数组的维度。如果在声明时没有指明维度，编译器会根据初始值的数量计算并推测出来；相反，如果指明了维度，那么初始值的总数量不应该超出指定的大小。如果维度比提供的初始值数量大，则用提供的初始值初始化靠前的元素，剩下的元素被初始化成默认值（参见 3.3.1 节，第 88 页）：

```
const unsigned sz = 3;
int ia1[sz] = {0, 1, 2};           // 含有 3 个元素的数组，元素值分别是 0, 1, 2
int a2[] = {0, 1, 2};              // 维度是 3 的数组
int a3[5] = {0, 1, 2};             // 等价于 a3[] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"};       // 等价于 a4[] = {"hi", "bye", ""}
int a5[2] = {0, 1, 2};             // 错误：初始值过多
```

字符数组的特殊性

字符数组有一种额外的初始化形式，我们可以用字符串字面值（参见 2.1.3 节，第 36 页）对此类数组初始化。当使用这种方式时，一定要注意字符串字面值的结尾处还有一个空字符，这个空字符也会像字符串的其他字符一样被拷贝到字符数组中去：

```
char a1[] = {'C', '+', '+'};        // 列表初始化，没有空字符
char a2[] = {'C', '+', '+', '\0'};    // 列表初始化，含有显式的空字符
char a3[] = "C++";                  // 自动添加表示字符串结束的空字符
const char a4[6] = "Daniel";        // 错误：没有空间可存放空字符！
```

a1 的维度是 3，a2 和 a3 的维度都是 4，a4 的定义是错误的。尽管字符串字面值"Daniel"看起来只有 6 个字符，但是数组的大小必须至少是 7，其中 6 个位置存放字面值的内容，另外 1 个存放结尾处的空字符。

不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值：

```
int a[] = {0, 1, 2};               // 含有 3 个整数的数组
int a2[] = a;                      // 错误：不允许使用一个数组初始化另一个数组
a2 = a;                            // 错误：不能把一个数组直接赋值给另一个数组
```



一些编译器支持数组的赋值，这就是所谓的编译器扩展（compiler extension）。但一般来说，最好避免使用非标准特性，因为含有非标准特性的程序很可能在其他编译器上无法正常工作。

理解复杂的数组声明

和 vector 一样，数组能存放大多数类型的对象。例如，可以定义一个存放指针的数组。又因为数组本身就是对象，所以允许定义数组的指针及数组的引用。在这几种情况中，定义存放指针的数组比较简单和直接，但是定义数组的指针或数组的引用就稍微复杂一点了：

```
115 int *ptrs[10];                  // ptrs 是含有 10 个整型指针的数组
      int &refs[10] = /* ? */;          // 错误：不存在引用的数组
      int (*Parray)[10] = &arr;         // Parray 指向一个含有 10 个整数的数组
      int (&arrRef)[10] = arr;          // arrRef 引用一个含有 10 个整数的数组
```

默认情况下，类型修饰符从右向左依次绑定。对于 `ptrs` 来说，从右向左（参见 2.3.3 节，第 52 页）理解其含义比较简单：首先知道我们定义的是一个大小为 10 的数组，它的名字是 `ptrs`，然后知道数组中存放的是指向 `int` 的指针。

但是对于 `Parray` 来说，从右向左理解就不太合理了。因为数组的维度是紧跟着被声明的名字的，所以就数组而言，由内向外阅读要比从右向左好多了。由内向外的顺序可帮助我们更好地理解 `Parray` 的含义：首先是圆括号括起来的部分，`*Parray` 意味着 `Parray` 是个指针，接下来观察右边，可知 `Parray` 是个指向大小为 10 的数组的指针，最后观察左边，知道数组中的元素是 `int`。这样最终的含义就明白无误了，`Parray` 是一个指针，它指向一个 `int` 数组，数组中包含 10 个元素。同理，`(&arrRef)` 表示 `arrRef` 是一个引用，它引用的对象是一个大小为 10 的数组，数组中元素的类型是 `int`。

当然，对修饰符的数量并没有特殊限制：

```
int *(&arry)[10] = ptrs; // arry 是数组的引用，该数组含有 10 个指针
```

按照由内向外的顺序阅读上述语句，首先知道 `arry` 是一个引用，然后观察右边知道，`arry` 引用的对象是一个大小为 10 的数组，最后观察左边知道，数组的元素类型是指向 `int` 的指针。这样，`arry` 就是一个含有 10 个 `int` 型指针的数组的引用。



要想理解数组声明的含义，最好的办法是从数组的名字开始按照由内向外的顺序阅读。

3.5.1 节练习

练习 3.27：假设 `txt_size` 是一个无参数的函数，它的返回值是 `int`。请回答下列哪个定义是非法的？为什么？

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

练习 3.28：下列数组中元素的值是什么？

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int ia2[10];
}
```

练习 3.29：相比于 `vector` 来说，数组有哪些缺点，请列举一些。

3.5.2 访问数组元素

116

与标准库类型 `vector` 和 `string` 一样，数组的元素也能使用范围 `for` 语句或下标运算符来访问。数组的索引从 0 开始，以一个包含 10 个元素的数组为例，它的索引从 0 到 9，而非从 1 到 10。

在使用数组下标的时候，通常将其定义为 `size_t` 类型。`size_t` 是一种机器相关的无符号类型，它被设计得足够大以便能表示内存中任意对象的大小。在 `cstddef` 头文件中定义了 `size_t` 类型，这个文件是 C 标准库 `stddef.h` 头文件的 C++ 语言版本。

数组除了大小固定这一特点外，其他用法与 `vector` 基本类似。例如，可以用数组来记录各分数段的成绩个数，从而实现与 3.3.3 节（第 93 页）的程序一样的功能：

```
// 以 10 分为一个分数段统计成绩的数量：0~9, 10~19, ..., 90~99, 100
unsigned scores[11] = {}； // 11 个分数段，全部初始化为 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // 将当前分数段的计数值加 1
}
```

与 93 页的程序相比，上面程序最大的不同是 `scores` 的声明。这里 `scores` 是一个含有 11 个无符号元素的数组。另外一处不太明显的区别是，本例所用的下标运算符是由 C++ 语言直接定义的，这个运算符能用在数组类型的运算对象上。93 页的那个程序所用的下标运算符是库模板 `vector` 定义的，只能用于 `vector` 类型的运算对象。

与 `vector` 和 `string` 一样，当需要遍历数组的所有元素时，最好的办法也是使用范围 `for` 语句。例如，下面的程序输出所有的 `scores`：

```
for (auto i : scores) // 对于 scores 中的每个计数值
    cout << i << " "; // 输出当前的计数值
cout << endl;
```

因为维度是数组类型的一部分，所以系统知道数组 `scores` 中有多少个元素，使用范围 `for` 语句可以减轻人为控制遍历过程的负担。

检查下标的值

与 `vector` 和 `string` 一样，数组的下标是否在合理范围之内由程序员负责检查，所谓合理就是说下标应该大于等于 0 而且小于数组的大小。要想防止数组下标越界，除了小心谨慎注意细节以及对代码进行彻底的测试之外，没有其他好办法。对于一个程序来说，即使顺利通过编译并执行，也不能肯定它不包含此类致命的错误。



大多数常见的安全问题都源于缓冲区溢出错误。当数组或其他类似数据结构的下标越界并试图访问非法内存区域时，就会产生此类错误。

117

3.5.2 节练习

练习 3.30：指出下面代码中的索引错误。

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
    ia[ix] = ix;
```

练习 3.31：编写一段程序，定义一个含有 10 个 `int` 的数组，令每个元素的值就是其下标值。

练习 3.32：将上一题刚刚创建的数组拷贝给另外一个数组。利用 `vector` 重写程序，实现类似的功能。

练习 3.33：对于 104 页的程序来说，如果不初始化 `scores` 将发生什么？

3.5.3 指针和数组

在 C++ 语言中，指针和数组有非常紧密的联系。就如即将介绍的，使用数组的时候编译器一般会把它转换成指针。

通常情况下，使用取地址符（参见 2.3.2 节，第 47 页）来获取指向某个对象的指针，取地址符可以用于任何对象。数组的元素也是对象，对数组使用下标运算符得到该数组指定位置的元素。因此像其他对象一样，对数组的元素使用取地址符就能得到指向该元素的指针：

```
string nums[] = {"one", "two", "three"}; // 数组的元素是 string 对象
string *p = &nums[0]; // p 指向 nums 的第一个元素
```

然而，数组还有一个特性：在很多用到数组名字的地方，编译器都会自动地将其替换为一个指向数组首元素的指针：

```
string *p2 = nums; // 等价于 p2 = &nums[0]
```



在大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针。

由上可知，在一些情况下数组的操作实际上是指针的操作，这一结论有很多隐含的意思。其中一层意思是当使用数组作为一个 auto（参见 2.5.2 节，第 61 页）变量的初始值时，推断得到的类型是指针而非数组：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
auto ia2(ia); // ia2 是一个整型指针，指向 ia 的第一个元素
ia2 = 42; // 错误：ia2 是一个指针，不能用 int 值给指针赋值
```

尽管 ia 是由 10 个整数构成的数组，但当使用 ia 作为初始值时，编译器实际执行的初始化过程类似于下面的形式：

```
auto ia2(&ia[0]); // 显然 ia2 的类型是 int*
```

118

必须指出的是，当使用 decltype 关键字（参见 2.5.3 节，第 62 页）时上述转换不会发生，decltype(ia) 返回的类型是由 10 个整数构成的数组：

```
// ia3 是一个含有 10 个整数的数组
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p; // 错误：不能用整型指针给数组赋值
ia3[4] = i; // 正确：把 i 的值赋给 ia3 的一个元素
```

指针也是迭代器

与 2.3.2 节（第 47 页）介绍的内容相比，指向数组元素的指针拥有更多功能。vector 和 string 的迭代器（参见 3.4 节，第 95 页）支持的运算，数组的指针全都支持。例如，允许使用递增运算符将指向数组元素的指针向前移动到下一个位置上：

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p 指向 arr 的第一个元素
++p; // p 指向 arr[1]
```

就像使用迭代器遍历 vector 对象中的元素一样，使用指针也能遍历数组中的元素。当然，这样做的前提是先得获取到指向数组第一个元素的指针和指向数组尾元素的下一个位置的指针。之前已经介绍过，通过数组名字或者数组中首元素的地址都能得到指向首元素的指针；不过获取尾后指针就要用到数组的另外一个特殊性质了。我们可以设法获取数组

尾元素之后的那个并不存在的元素的地址：

```
int *e = &arr[10]; // 指向 arr 尾元素的下一位置的指针
```

这里显然使用下标运算符索引了一个不存在的元素，arr 有 10 个元素，尾元素所在位置的索引是 9，接下来那个不存在的元素唯一的用处就是提供其地址用于初始化 e。就像尾后迭代器（参见 3.4.1 节，第 95 页）一样，尾后指针也不指向具体的元素。因此，不能对尾后指针执行解引用或递增的操作。

利用上面得到的指针能重写之前的循环，令其输出 arr 的全部元素：

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // 输出 arr 的元素
```

标准库函数 begin 和 end

尽管能计算得到尾后指针，但这种用法极易出错。为了让指针的使用更简单、更安全，C++11 新标准引入了两个名为 begin 和 end 的函数。这两个函数与容器中的两个同名成员（参见 3.4.1 节，第 95 页）功能类似，不过数组毕竟不是类类型，因此这两个函数不是成员函数。正确的使用形式是将数组作为它们的参数：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
int *beg = begin(ia);           // 指向 ia 首元素的指针
int *last = end(ia);           // 指向 arr 尾元素的下一位置的指针
```

119 begin 函数返回指向 ia 首元素的指针，end 函数返回指向 ia 尾元素下一位置的指针，这两个函数定义在 iterator 头文件中。

使用 begin 和 end 可以很容易地写出一个循环并处理数组中的元素。例如，假设 arr 是一个整型数组，下面的程序负责找到 arr 中的第一个负数：

```
// pbeg 指向 arr 的首元素，pend 指向 arr 尾元素的下一位置
int *pbeg = begin(arr), *pend = end(arr);
// 寻找第一个负值元素，如果已经检查完全部元素则结束循环
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

首先定义了两个名为 pbeg 和 pend 的整型指针，其中 pbeg 指向 arr 的第一个元素，pend 指向 arr 尾元素的下一位置。while 语句的条件部分通过比较 pbeg 和 pend 来确保可以安全地对 pbeg 解引用，如果 pbeg 确实指向了一个元素，将其解引用并检查元素值是否为负值。如果是，条件失效、退出循环；如果不是，将指针向前移动一位继续考查下一个元素。



一个指针如果指向了某种内置类型数组的尾元素的“下一位置”，则其具备与 vector 的 end 函数返回的与迭代器类似的功能。特别要注意，尾后指针不能执行解引用和递增操作。

指针运算

指向数组元素的指针可以执行表 3.6（第 96 页）和表 3.7（第 99 页）列出的所有迭代器运算。这些运算，包括解引用、递增、比较、与整数相加、两个指针相减等，用在指针和用在迭代器上意义完全一致。

给（从）一个指针加上（减去）某整数值，结果仍是指针。新指针指向的元素与原来

的指针相比前进了（后退了）该整数值个位置：

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr;           // 等价于 int *ip = &arr[0]
int *ip2 = ip + 4;      // ip2 指向 arr 的尾元素 arr[4]
```

ip 加上 4 所得的结果仍是一个指针，该指针所指的元素与 ip 原来所指的元素相比前进了 4 个位置。

给指针加上一个整数，得到的新指针仍需指向同一数组的其他元素，或者指向同一数组的尾元素的下一位置：

```
// 正确：arr 转换成指向它首元素的指针；p 指向 arr 尾元素的下一位置
int *p = arr + sz;        // 使用警告：不要解引用！
int *p2 = arr + 10;       // 错误：arr 只有 5 个元素，p2 的值未定义
```

当给 arr 加上 sz 时，编译器自动地将 arr 转换成指向数组 arr 中首元素的指针。执行加法后，指针从首元素开始向前移动了 sz（这里是 5）个位置，指向新位置的元素。也就是说，它指向了数组 arr 尾元素的下一位置。如果计算所得的指针超出了上述范围就将产生错误，而且这种错误编译器一般发现不了。◀120

和迭代器一样，两个指针相减的结果是它们之间的距离。参与运算的两个指针必须指向同一个数组当中的元素：

```
auto n = end(arr) - begin(arr); // n 的值是 5，也就是 arr 中元素的数量
```

两个指针相减的结果的类型是一种名为 **ptrdiff_t** 的标准库类型，和 **size_t** 一样，**ptrdiff_t** 也是一种定义在 **cstddef** 头文件中的机器相关的类型。因为差值可能为负值，所以 **ptrdiff_t** 是一种带符号类型。

只要两个指针指向同一个数组的元素，或者指向该数组的尾元素的下一位置，就能利用关系运算符对其进行比较。例如，可以按照如下的方式遍历数组中的元素：

```
int *b = arr, *e = arr + sz;
while (b < e) {
    // 使用 *b
    ++b;
}
```

如果两个指针分别指向不相关的对象，则不能比较它们：

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
// 未定义的：p 和 e 无关，因此比较毫无意义！
while (p < e)
```

尽管作用可能不是特别明显，但必须说明的是，上述指针运算同样适用于空指针（参见 2.3.2 节，第 48 页）和所指对象并非数组的指针。在后一种情况下，两个指针必须指向同一个对象或该对象的下一位置。如果 p 是空指针，允许给 p 加上或减去一个值为 0 的整型常量表达式（参见 2.4.4 节，第 58 页）。两个空指针也允许彼此相减，结果当然是 0。

解引用和指针运算的交互

指针加上一个整数所得的结果还是一个指针。假设结果指针指向了一个元素，则允许解引用该结果指针：

```
int ia[] = {0,2,4,6,8}; // 含有 5 个整数的数组
int last = *(ia + 4); // 正确：把 last 初始化成 8，也就是 ia[4] 的值
```

表达式 $\star(\text{ia}+4)$ 计算 ia 前进 4 个元素后的新地址，解引用该结果指针的效果等价于表达式 ia[4]。

回忆一下在 3.4.1 节（第 98 页）中介绍过如果表达式含有解引用运算符和点运算符，最好在必要的地方加上圆括号。类似的，此例中指针加法的圆括号也不可缺少。如果写成下面的形式：

```
last = *ia + 4; // 正确：last = 4 等价于 ia[0] + 4
```

含义就与之前完全不同了，此时先解引用 ia，然后给解引用的结果再加上 4。4.1.2 节（第 121 页）将对这一问题做进一步分析。



下标和指针

121

如前所述，在很多情况下使用数组的名字其实用的是一个指向数组首元素的指针。一个典型的例子是当对数组使用下标运算符时，编译器会自动执行上述转换操作。给定

```
int ia[] = {0,2,4,6,8}; // 含有 5 个整数的数组
```

此时，ia[0]是一个使用了数组名字的表达式，对数组执行下标运算其实是对指向数组元素的指针执行下标运算：

```
int i = ia[2];           // ia 转换成指向数组首元素的指针
                         // ia[2] 得到 (ia + 2) 所指的元素
int *p = ia;             // p 指向 ia 的首元素
i = *(p + 2);           // 等价于 i = ia[2]
```

只要指针指向的是数组中的元素（或者数组中尾元素的下一位置），都可以执行下标运算：

```
int *p = &ia[2];         // p 指向索引为 2 的元素
int j = p[1];            // p[1] 等价于 *(p + 1)，就是 ia[3] 表示的那个元素
int k = p[-2];           // p[-2] 是 ia[0] 表示的那个元素
```

虽然标准库类型 string 和 vector 也能执行下标运算，但是数组与它们相比还是有所不同。标准库类型限定使用的下标必须是无符号类型，而内置的下标运算无此要求，上面的最后一个例子很好地说明了这一点。内置的下标运算符可以处理负值，当然，结果地址必须指向原来的指针所指同一数组中的元素（或是同一数组尾元素的下一位置）。



内置的下标运算符所用的索引值不是无符号类型，这一点与 vector 和 string 不一样。

3.5.3 节练习

练习 3.34：假定 p1 和 p2 指向同一个数组中的元素，则下面程序的功能是什么？什么情况下该程序是非法的？

```
p1 += p2 - p1;
```

练习 3.35：编写一段程序，利用指针将数组中的元素置为 0。

练习 3.36：编写一段程序，比较两个数组是否相等。再写一段程序，比较两个 vector 对象是否相等。

3.5.4 C 风格字符串



尽管 C++ 支持 C 风格字符串，但在 C++ 程序中最好还是不要使用它们。这是因为 C 风格字符串不仅使用起来不太方便，而且极易引发程序漏洞，是诸多安全问题的根本原因。

<122

字符串字面值是一种通用结构的实例，这种结构即是 C++ 由 C 继承而来的 C 风格字符串 (C-style character string)。C 风格字符串不是一种类型，而是为了表达和使用字符串而形成的一种约定俗成的写法。按此习惯书写的字符串存放在字符数组中并以空字符结束 (null terminated)。以空字符结束的意思是在字符串最后一个字符后面跟着一个空字符 ('\0')。一般利用指针来操作这些字符串。

C 标准库 String 函数

表 3.8 列举了 C 语言标准库提供的一组函数，这些函数可用于操作 C 风格字符串，它们定义在 `cstring` 头文件中，`cstring` 是 C 语言头文件 `string.h` 的 C++ 版本。

表 3.8: C 风格字符串的函数

<code>strlen(p)</code>	返回 p 的长度，空字符不计算在内
<code>strcmp(p1, p2)</code>	比较 p1 和 p2 的相等性。如果 $p1 == p2$ ，返回 0；如果 $p1 > p2$ ，返回一个正值；如果 $p1 < p2$ ，返回一个负值
<code>strcat(p1, p2)</code>	将 p2 附加到 p1 之后，返回 p1
<code>strcpy(p1, p2)</code>	将 p2 拷贝给 p1，返回 p1



表 3.8 所列的函数不负责验证其字符串参数。

WARNING

传入此类函数的指针必须指向以空字符作为结束的数组：

```
char ca[] = {'C', '+', '+'};      // 不以空字符结束
cout << strlen(ca) << endl;      // 严重错误：ca 没有以空字符结束
```

此例中，ca 虽然也是一个字符数组但它不是以空字符作为结束的，因此上述程序将产生未定义的结果。`strlen` 函数将有可能沿着 ca 在内存中的位置不断向前寻找，直到遇到空字符才停下来。

比较字符串

比较两个 C 风格字符串的方法和之前学习过的比较标准库 `string` 对象的方法大相径庭。比较标准库 `string` 对象的时候，用的是普通的关系运算符和相等性运算符：

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2) // false: s2 小于 s1
```

如果把这些运算符用在两个 C 风格字符串上，实际比较的将是指针而非字符串本身：

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) // 未定义的：试图比较两个无关地址
```

<123

谨记之前介绍过的，当使用数组的时候其实真正用的是指向数组首元素的指针（参见 3.5.3 节，第 105 页）。因此，上面的 if 条件实际上比较的是两个 `const char*` 的值。这两个

指针指向的并非同一对象，所以将得到未定义的结果。

要想比较两个 C 风格字符串需要调用 `strcmp` 函数，此时比较的就不再是指针了。如果两个字符串相等，`strcmp` 返回 0；如果前面的字符串较大，返回正值；如果后面的字符串较大，返回负值：

```
if (strcmp(ca1, ca2) < 0) // 和两个 string 对象的比较 s1 < s2 效果一样
```

目标字符串的大小由调用者指定

连接或拷贝 C 风格字符串也与标准库 `string` 对象的同类操作差别很大。例如，要想把刚刚定义的那两个 `string` 对象 `s1` 和 `s2` 连接起来，可以直接写成下面的形式：

```
// 将 largeStr 初始化成 s1、一个空格和 s2 的连接
string largeStr = s1 + " " + s2;
```

同样的操作如果放到 `ca1` 和 `ca2` 这两个数组身上就会产生错误了。表达式 `ca1 + ca2` 试图将两个指针相加，显然这样的操作没什么意义，也肯定是非常非法的。

正确的方法是使用 `strcat` 函数和 `strcpy` 函数。不过要想使用这两个函数，还必须提供一个用于存放结果字符串的数组，该数组必须足够大以便容纳下结果字符串及末尾的空字符。下面的代码虽然很常见，但是充满了安全风险，极易引发严重错误：

```
// 如果我们计算错了 largeStr 的大小将引发严重错误
strcpy(largeStr, ca1);           // 把 ca1 拷贝给 largeStr
strcat(largeStr, " ");           // 在 largeStr 的末尾加上一个空格
strcat(largeStr, ca2);           // 把 ca2 连接到 largeStr 后面
```

一个潜在的问题是，我们在估算 `largeStr` 所需的空间时不容易估准，而且 `largeStr` 所存的内容一旦改变，就必须重新检查其空间是否足够。不幸的是，这样的代码到处都是，程序员根本没法照顾周全。这类代码充满了风险而且经常导致严重的安全泄漏。



对大多数应用来说，使用标准库 `string` 要比使用 C 风格字符串更安全、更高效。

124 >

3.5.4 节练习

练习 3.37：下面的程序是何含义，程序的输出结果是什么？

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

练习 3.38：在本节中我们提到，将两个指针相加不但是非法的，而且也没什么意义。请问为什么两个指针相加没什么意义？

练习 3.39：编写一段程序，比较两个 `string` 对象。再编写一段程序，比较两个 C 风格字符串的内容。

练习 3.40：编写一段程序，定义两个字符数组并用字符串字面值初始化它们；接着再定义一个字符数组存放前两个数组连接后的结果。使用 `strcpy` 和 `strcat` 把前两个数组的内容拷贝到第三个数组中。

3.5.5 与旧代码的接口

很多 C++ 程序在标准库出现之前就已经写成了，它们肯定没用到 `string` 和 `vector` 类型。而且，有一些 C++ 程序实际上是与 C 语言或其他语言的接口程序，当然也无法使用 C++ 标准库。因此，现代的 C++ 程序不得不与那些充满了数组和/或 C 风格字符串的代码衔接，为了使这一工作简单易行，C++ 专门提供了一组功能。

混用 `string` 对象和 C 风格字符串



3.2.1 节（第 76 页）介绍过允许使用字符串字面值来初始化 `string` 对象：

```
string s("Hello World"); // s 的内容是 Hello World
```

更一般的情况是，任何出现字符串字面值的地方都可以用以空字符结束的字符数组来替代：

- 允许使用以空字符结束的字符数组来初始化 `string` 对象或为 `string` 对象赋值。
- 在 `string` 对象的加法运算中允许使用以空字符结束的字符数组作为其中一个运算对象（不能两个运算对象都是）；在 `string` 对象的复合赋值运算中允许使用以空字符结束的字符数组作为右侧的运算对象。

上述性质反过来就不成立了：如果程序的某处需要一个 C 风格字符串，无法直接用 `string` 对象来代替它。例如，不能用 `string` 对象直接初始化指向字符的指针。为了完成该功能，`string` 专门提供了一个名为 `c_str` 的成员函数：

```
char *str = s; // 错误：不能用 string 对象初始化 char*
const char *str = s.c_str(); // 正确
```

顾名思义，`c_str` 函数的返回值是一个 C 风格的字符串。也就是说，函数的返回结果是一个指针，该指针指向一个以空字符结束的字符数组，而这个数组所存的数据恰好与那个 `string` 对象的一样。结果指针的类型是 `const char*`，从而确保我们不会改变字符数组的内容。

< 125

我们无法保证 `c_str` 函数返回的数组一直有效，事实上，如果后续的操作改变了 `s` 的值就可能让之前返回的数组失去效用。



如果执行完 `c_str()` 函数后程序想一直都能使用其返回的数组，最好将该数组重新拷贝一份。

使用数组初始化 `vector` 对象

3.5.1 节（第 102 页）介绍过不允许使用一个数组为另一个内置类型的数组赋初值，也不允许使用 `vector` 对象初始化数组。相反的，允许使用数组来初始化 `vector` 对象。要实现这一目的，只需指明要拷贝区域的首元素地址和尾后地址就可以了：

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
// ivec 有 6 个元素，分别是 int_arr 中对应元素的副本
vector<int> ivec(begin(int_arr), end(int_arr));
```

在上述代码中，用于创建 `ivec` 的两个指针实际上指明了用来初始化的值在数组 `int_arr` 中的位置，其中第二个指针应指向待拷贝区域尾元素的下一位置。此例中，使用标准库函数 `begin` 和 `end`（参见 3.5.3 节，第 106 页）来分别计算 `int_arr` 的首指针和尾后指针。在最终的结果中，`ivec` 将包含 6 个元素，它们的次序和值都与数组 `int_arr` 完全

一样。 ▶

用于初始化 `vector` 对象的值也可能仅是数组的一部分：

```
// 拷贝三个元素：int_arr[1]、int_arr[2]、int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

这条初始化语句用 3 个元素创建了对象 `subVec`，3 个元素的值分别来自 `int_arr[1]`、`int_arr[2]` 和 `int_arr[3]`。

建议：尽量使用标准库类型而非数组

使用指针和数组很容易出错。一部分原因是概念上的问题：指针常用于底层操作，因此容易引发一些与烦琐细节有关的错误。其他问题则源于语法错误，特别是声明指针时的语法错误。

现代的 C++ 程序应当尽量使用 `vector` 和迭代器，避免使用内置数组和指针；应该尽量使用 `string`，避免使用 C 风格的基于数组的字符串。

3.5.5 节练习

练习 3.41： 编写一段程序，用整型数组初始化一个 `vector` 对象。

练习 3.42： 编写一段程序，将含有整数元素的 `vector` 对象拷贝给一个整型数组。



3.6 多维数组

严格来说，C++ 语言中没有多维数组，通常所说的多维数组其实是数组的数组。谨记这一点，对今后理解和使用多维数组大有益处。

当一个数组的元素仍然是数组时，通常使用两个维度来定义它：一个维度表示数组本身大小，另外一个维度表示其元素（也是数组）大小：

```
int ia[3][4]; // 大小为 3 的数组，每个元素是含有 4 个整数的数组
// 大小为 10 的数组，它的每个元素都是大小为 20 的数组，
// 这些数组的元素是含有 30 个整数的数组
int arr[10][20][30] = {0}; // 将所有元素初始化为 0
```

如 3.5.1 节（第 103 页）所介绍的，按照由内而外的顺序阅读此类定义有助于更好地理解其真实含义。在第一条语句中，我们定义的名字是 `ia`，显然 `ia` 是一个含有 3 个元素的数组。接着观察右边发现，`ia` 的元素也有自己的维度，所以 `ia` 的元素本身又都是含有 4 个元素的数组。再观察左边知道，真正存储的元素是整数。因此最后可以明确第一条语句的含义：它定义了一个大小为 3 的数组，该数组的每个元素都是含有 4 个整数的数组。

使用同样的方式理解 `arr` 的定义。首先 `arr` 是一个大小为 10 的数组，它的每个元素都是大小为 20 的数组，这些数组的元素又都是含有 30 个整数的数组。实际上，定义数组时对下标运算符的数量并没有限制，因此只要愿意就可以定义这样一个数组：它的元素还是数组，下一级数组的元素还是数组，再下一级数组的元素还是数组，以此类推。

对于二维数组来说，常把第一个维度称作行，第二个维度称作列。

多维数组的初始化

允许使用花括号括起来的一组值初始化多维数组，这点和普通的数组一样。下面的初始化形式中，多维数组的每一行分别用花括号括了起来：

```
int ia[3][4] = {           // 三个元素，每个元素都是大小为 4 的数组
    {0, 1, 2, 3},          // 第 1 行的初始值
    {4, 5, 6, 7},          // 第 2 行的初始值
    {8, 9, 10, 11}         // 第 3 行的初始值
};
```

其中内层嵌套着的花括号并非必需的，例如下面的初始化语句，形式上更为简洁，完成的功能和上面这段代码完全一样：

```
// 没有标识每行的花括号，与之前的初始化语句是等价的
int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

类似于一维数组，在初始化多维数组时也并非所有元素的值都必须包含在初始化列表之内。如果仅仅想初始化每一行的第一个元素，通过如下的语句即可：

```
// 显式地初始化每行的首元素
int ia[3][4] = {{0}, {4}, {8}};
```

<127>

其他未列出的元素执行默认值初始化，这个过程和一维数组（参见 3.5.1 节，第 102 页）一样。在这种情况下如果再省略掉内层的花括号，结果就大不一样了。下面的代码

```
// 显式地初始化第 1 行，其他元素执行值初始化
int ix[3][4] = {0, 3, 6, 9};
```

含义发生了变化，它初始化的是第一行的 4 个元素，其他元素被初始化为 0。

多维数组的下标引用

可以使用下标运算符来访问多维数组的元素，此时数组的每个维度对应一个下标运算符。

如果表达式含有的下标运算符数量和数组的维度一样多，该表达式的结果将是给定类型的元素；反之，如果表达式含有的下标运算符数量比数组的维度小，则表达式的结果将是给定索引处的一个内层数组：

```
// 用 arr 的首元素为 ia 最后一行的最后一个元素赋值
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[1]; // 把 row 绑定到 ia 的第二个 4 元素数组上
```

在第一个例子中，对于用到的两个数组来说，表达式提供的下标运算符数量都和它们各自的维度相同。在等号左侧，`ia[2]` 得到数组 `ia` 的最后一行，此时返回的是表示 `ia` 最后一行的那个一维数组而非任何实际元素；对这个一维数组再取下标，得到编号为 [3] 的元素，也就是这一行的最后一个元素。

类似的，等号右侧的运算对象包含 3 个维度。首先通过索引 0 得到最外层的数组，它是一个大小为 20 的（多维）数组；接着获取这 20 个元素数组的第一个元素，得到一个大小为 30 的一维数组；最后再取出其中的第一个元素。

在第二个例子中，把 `row` 定义成一个含有 4 个整数的数组的引用，然后将其绑定到 `ia` 的第 2 行。

再举一个例子，程序中经常会用到两层嵌套的 `for` 循环来处理多维数组的元素：

```
constexpr size_t rowCount = 3, colCount = 4;
```

```

int ia[rowCnt][colCnt]; // 12 个未初始化的元素
// 对于每一行
for (size_t i = 0; i != rowCnt; ++i) {
    // 对于行内的每一列
    for (size_t j = 0; j != colCnt; ++j) {
        // 将元素的位置索引作为它的值
        ia[i][j] = i * colCnt + j;
    }
}

```

外层的 `for` 循环遍历 `ia` 的所有元素，注意这里的元素是一维数组；内层的 `for` 循环则遍历那些一维数组的整数元素。此例中，我们将元素的值设为该元素在整个数组中的序号。

使用范围 `for` 语句处理多维数组

由于在 C++11 新标准中新增了范围 `for` 语句，所以前一个程序可以简化为如下形式：

```

size_t cnt = 0;
for (auto &row : ia)           // 对于外层数组的每一个元素
    for (auto &col : row) {     // 对于内层数组的每一个元素
        col = cnt;             // 将下一个值赋给该元素
        ++cnt;                  // 将 cnt 加 1
    }
}

```

这个循环赋给 `ia` 元素的值和之前那个循环是完全相同的，区别之处是通过使用范围 `for` 语句把管理数组索引的任务交给了系统来完成。因为要改变元素的值，所以得把控制变量 `row` 和 `col` 声明成引用类型（参见 3.2.3 节，第 83 页）。第一个 `for` 循环遍历 `ia` 的所有元素，这些元素是大小为 4 的数组，因此 `row` 的类型就应该是含有 4 个整数的数组的引用。第二个 `for` 循环遍历那些 4 元素数组中的某一个，因此 `col` 的类型是整数的引用。每次迭代把 `cnt` 的值赋给 `ia` 的当前元素，然后将 `cnt` 加 1。

在上面的例子中，因为要改变数组元素的值，所以我们选用引用类型作为循环控制变量，但其实还有一个深层次的原因促使我们这么做。举一个例子，考虑如下的循环：

```

for (const auto &row : ia) // 对于外层数组的每一个元素
    for (auto col : row)   // 对于内层数组的每一个元素
        cout << col << endl;

```

这个循环中并没有任何写操作，可是我们还是将外层循环的控制变量声明成了引用类型，这是为了避免数组被自动转成指针（参见 3.5.3 节，第 105 页）。假设不用引用类型，则循环如下述形式：

```

for (auto row : ia)
    for (auto col : row)

```

程序将无法通过编译。这是因为，像之前一样第一个循环遍历 `ia` 的所有元素，注意这些元素实际上是大小为 4 的数组。因为 `row` 不是引用类型，所以编译器初始化 `row` 时会自动将这些数组形式的元素（和其他类型的数组一样）转换成指向该数组内首元素的指针。这样得到的 `row` 的类型就是 `int*`，显然内层的循环就不合法了，编译器将试图在一个 `int*` 内遍历，这显然和程序的初衷相去甚远。



要使用范围 `for` 语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型。

指针和多维数组

当程序使用多维数组的名字时，也会自动将其转换成指向数组首元素的指针。



定义指向多维数组的指针时，千万别忘了这个多维数组实际上是数组的数组。

◀ 129

因为多维数组实际上是数组的数组，所以由多维数组名转换得来的指针实际上是指向第一个内层数组的指针：

```
int ia[3][4];           // 大小为 3 的数组，每个元素是含有 4 个整数的数组
int (*p)[4] = ia;       // p 指向含有 4 个整数的数组
p = &ia[2];             // p 指向 ia 的尾元素
```

根据 3.5.1 节（第 103 页）提出的策略，我们首先明确 (**p*) 意味着 *p* 是一个指针。接着观察右边发现，指针 *p* 所指的是一个维度为 4 的数组；再观察左边知道，数组中的元素是整数。因此，*p* 就是指向含有 4 个整数的数组的指针。



在上述声明中，圆括号必不可少：

```
int *ip[4];           // 整型指针的数组
int (*ip)[4];         // 指向含有 4 个整数的数组
```

随着 C++11 新标准的提出，通过使用 `auto` 或者 `decltype`（参见 2.5.2 节，第 61 页）就能尽可能地避免在数组前面加上一个指针类型了：

```
// 输出 ia 中每个元素的值，每个内层数组各占一行
// p 指向含有 4 个整数的数组
for (auto p = ia; p != ia + 3; ++p) {
    // q 指向 4 个整数数组的首元素，也就是说，q 指向一个整数
    for (auto q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
```

外层的 `for` 循环首先声明一个指针 *p* 并令其指向 *ia* 的第一个内层数组，然后依次迭代直到 *ia* 的全部 3 行都处理完为止。其中递增运算 `++p` 负责将指针 *p* 移动到 *ia* 的下一行。

内层的 `for` 循环负责输出内层数组所包含的值。它首先令指针 *q* 指向 *p* 当前所在行的第一个元素。`*p` 是一个含有 4 个整数的数组，像往常一样，数组名被自动地转换成指向该数组首元素的指针。内层 `for` 循环不断迭代直到我们处理完了当前内层数组的所有元素为止。为了获取内层 `for` 循环的终止条件，再一次解引用 *p* 得到指向内层数组首元素的指针，给它加上 4 就得到了终止条件。

当然，使用标准库函数 `begin` 和 `end`（参见 3.5.3 节，第 106 页）也能实现同样的功能，而且看起来更简洁一些：

```
// p 指向 ia 的第一个数组
for (auto p = begin(ia); p != end(ia); ++p) {
    // q 指向内层数组的首元素
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' '; // 输出 q 所指的整数值
    cout << endl;
}
```

C++
11

130> 在这一版本的程序中，循环终止条件由 `end` 函数负责判断。虽然我们也能推断出 `p` 的类型是指向含有 4 个整数的数组的指针，`q` 的类型是指向整数的指针，但是使用 `auto` 关键字我们就不必再烦心这些类型到底是什么了。

类型别名简化多维数组的指针

读、写和理解一个指向多维数组的指针是一个让人不胜其烦的工作，使用类型别名（参见 2.5.1 节，第 60 页）能让这项工作变得简单一点儿，例如：

```
using int_array = int[4]; // 新标准下类型别名的声明，参见 2.5.1 节（第 60 页）
typedef int int_array[4]; // 等价的 typedef 声明，参见 2.5.1 节（第 60 页）

// 输出 ia 中每个元素的值，每个内层数组各占一行
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
```

程序将类型“4 个整数组成的数组”命名为 `int_array`，用类型名 `int_array` 定义外层循环的控制变量让程序显得简洁明了。

3.6 节练习

练习 3.43：编写 3 个不同版本的程序，令其均能输出 `ia` 的元素。版本 1 使用范围 `for` 语句管理迭代过程；版本 2 和版本 3 都使用普通的 `for` 语句，其中版本 2 要求用下标运算符，版本 3 要求用指针。此外，在所有 3 个版本的程序中都要直接写出数据类型，而不能使用类型别名、`auto` 关键字或 `decltype` 关键字。

练习 3.44：改写上一个练习中的程序，使用类型别名来代替循环控制变量的类型。

练习 3.45：再一次改写程序，这次使用 `auto` 关键字。

小结

131

`string` 和 `vector` 是两种最重要的标准库类型。`string` 对象是一个可变长的字符序列，`vector` 对象是一组同类型对象的容器。

迭代器允许对容器中的对象进行间接访问，对于 `string` 对象和 `vector` 对象来说，可以通过迭代器访问元素或者在元素间移动。

数组和指向数组元素的指针在一个较低的层次上实现了与标准库类型 `string` 和 `vector` 类似功能。一般来说，应该优先选用标准库提供的类型，之后再考虑 C++ 语言内置的低层的替代品数组或指针。

术语表

`begin` 是 `string` 和 `vector` 的成员，返回指向第一个元素的迭代器。也是一个标准库函数，输入一个数组，返回指向该数组首元素的指针。

缓冲区溢出 (buffer overflow) 一种严重的程序故障，主要的原因是试图通过一个越界的索引访问容器内容，容器类型包括 `string`、`vector` 和数组等。

C 风格字符串 (C-style string) 以空字符结束的字符数组。字符串字面值是 C 风格字符串，C 风格字符串容易出错。

类模板 (class template) 用于创建具体类类型的模板。要想使用类模板，必须提供关于类型的辅助信息。例如，要定义一个 `vector` 对象需要指定元素的类型：`vector<int>` 包含 `int` 类型的元素。

编译器扩展 (compiler extension) 某个特定的编译器为 C++ 语言额外增加的特性。基于编译器扩展编写的程序不易移植到其他编译器上。

容器 (container) 是一种类型，其对象容纳了一组给定类型的对象。`vector` 是一种容器类型。

拷贝初始化 (copy initialization) 使用赋值号 (=) 的初始化形式。新创建的对象是初始值的一个副本。

difference_type 由 `string` 和 `vector` 定义的一种带符号整数类型，表示两个迭代

器之间的距离。

直接初始化 (direct initialization) 不使用赋值号 (=) 的初始化形式。

`empty` 是 `string` 和 `vector` 的成员，返回一个布尔值。当对象的大小为 0 时返回真，否则返回假。

`end` 是 `string` 和 `vector` 的成员，返回一个尾后迭代器。也是一个标准库函数，输入一个数组，返回指向该数组尾元素的下一位置的指针。

`getline` 在 `string` 头文件中定义的一个函数，以一个 `istream` 对象和一个 `string` 对象为输入参数。该函数首先读取输入流的内容直到遇到换行符停止，然后将读入的数据存入 `string` 对象，最后返回 `istream` 对象。其中换行符读入但是不保留。

索引 (index) 是下标运算符使用的值。表示要在 `string` 对象、`vector` 对象或者数组中访问的一个位置。

实例化 (instantiation) 编译器生成一个指定的模板类或函数的过程。

迭代器 (iterator) 是一种类型，用于访问容器中的元素或者在元素之间移动。

迭代器运算 (iterator arithmetic) 是 `string` 或 `vector` 的迭代器的运算：迭代器与整数相加或相减得到一个新的迭代器，与原来的迭代器相比，新迭代器向前

或向后移动了若干个位置。两个迭代器相减得到它们之间的距离，此时它们必须指向同一个容器的元素或该容器尾元素的下一个位置。

[132] > 以空字符结束的字符串 (null-terminated string) 是一个字符串，它的最后一个字符后面还跟着一个空字符 ('\0')。

尾后迭代器 (off-the-end iterator) `end` 函数返回的迭代器，指向一个并不存在的元素，该元素位于容器尾元素的下一个位置。

指针运算 (pointer arithmetic) 是指针类型支持的算术运算。指向数组的指针所支持的运算种类与迭代器运算一样。

`ptrdiff_t` 是 `cstdint` 头文件中定义的一种与机器实现有关的带符号整数类型，它的空间足够大，能够表示数组中任意两个指针之间的距离。

`push_back` 是 `vector` 的成员，向 `vector` 对象的末尾添加元素。

范围 for 语句 (range for) 一种控制语句，可以在值的一个特定集合内迭代。

`size` 是 `string` 和 `vector` 的成员，分别返回字符的数量或元素的数量。返回值的类型是 `size_type`。

`size_t` 是 `cstdint` 头文件中定义的一种与机器实现有关的无符号整数类型，它的空间足够大，能够表示任意数组的大小。

`size_type` 是 `string` 和 `vector` 定义的类型的名字，能存放下任意 `string` 对象或 `vector` 对象的大小。在标准库中，`size_type` 被定义为无符号类型。

`string` 是一种标准库类型，表示字符的序列。

using 声明 (using declaration) 令命名空间中的某个名字可被程序直接使用。

```
using 命名空间 :: 名字;
```

上述语句的作用是令程序可以直接使用名字，而无须写它的前缀部分 `命名空间::`。

值初始化 (value initialization) 是一种初始化过程。内置类型初始化为 0，类类型由

类的默认构造函数初始化。只有当类包含默认构造函数时，该类的对象才会被值初始化。对于容器的初始化来说，如果只说明了容器的大小而没有指定初始值的话，就会执行值初始化。此时编译器会生成一个值，而容器的元素被初始化为该值。

`vector` 是一种标准库类型，容纳某指定类型的一组元素。

++运算符 (++ operator) 是迭代器和指针定义的递增运算符。执行“加 1”操作使得迭代器指向下一个元素。

[]运算符 ([] operator) 下标运算符。`obj[j]` 得到容器对象 `obj` 中位置 `j` 的那个元素。索引从 0 开始，第一个元素的索引是 0，尾元素的索引是 `obj.size() - 1`。下标运算符的返回值是一个对象。如果 `p` 是指针、`n` 是整数，则 `p[n]` 与 `* (p+n)` 等价。

->运算符 (-> operator) 箭头运算符，该运算符综合了解引用操作和点操作。`a->b` 等价于 `(*a).b`。

<<运算符 (<< operator) 标准库类型 `string` 定义的输出运算符，负责输出 `string` 对象中的字符。

>>运算符 (>> operator) 标准库类型 `string` 定义的输入运算符，负责读入一组字符，遇到空白停止，读入的内容赋给运算符右侧的运算对象，该运算对象应该是一个 `string` 对象。

!运算符 (! operator) 逻辑非运算符，将它的运算对象的布尔值取反。如果运算对象是假，则结果为真，如果运算对象是真，则结果为假。

&&运算符 (&& operator) 逻辑与运算符，如果两个运算对象都是真，结果为真。只有当左侧运算对象为真时才会检查右侧运算对象。

||运算符 (|| operator) 逻辑或运算符，任何一个运算对象是真，结果就为真。只有当左侧运算对象为假时才会检查右侧运算对象。