

## 4.1. 数组

一种数据结构，能够存储多个同类型的数据。

数组声明应指出以下三点：

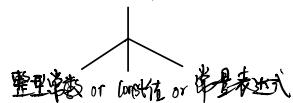
· 存储在每个元素中的值的类型

声明通用格式：

· 数组名

`typeName arrayName[arraySize];`

· 数组中的元素数

  
整型常数 or 常量表达式  
整型常数  
常量表达式  
整型常数 or 常量表达式

Ex:

`short months[12];`

数组之所以被称为复合类型，是因为它是使用其它类型来创建的。

C++ 允许在声明时初始化数组元素

Ex:

`int yearcosts[3] = { 20, 30, 5 };`

· 如果将赋值运算符用于数组，得到的将是整个数组中的空参数。

初始化：

Ex:

`float balance[100] = {};` // all elements set to 0.

## 4.2. 字符串

存储在内存的连续字节中的一系列字符。

C++ 处理字符串的方式有两种：

1. C-风格字符串 (C-style string)

2. 基于 `string` 类库的方法

## C—风格字符串：

以空字符串(null character)结尾，空字符串写作 `\"`，其 ASCII 码为 0，用来标记字符串的结尾。

Ex:

```
char cat[8] = {'f', 'a', 't', 'e', 's', 'l', 'a', '\0'};
```

共 8 个

```
1 #include<iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     char dog[8] = {'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};
8     char cat[8] = {'f', 'a', 't', 'e', 's', 'l', 'a', '\0'};
9
10    cout << dog << endl;
11    cout << cat << endl;
12
13    return 0;
14 }
```

SQL CONSOLE 问题 输出 调试控制台 终端

```
aiden@MacBook-Pro C++ % ./a.out
beaux II
fatessa
aiden@MacBook-Pro C++ %
```

> 两个数组都是 char 数组，但只有第二个数组是字符串。  
dog 数组中因含元 "\0"，所以 cout 语句将输出  
处理字符串中的字符，直到遇到 "\0" 空字符串为止。

## 另外一种更高级的数组初始化字符串方法：

Ex: `char bird[11] = "Mr. Cheeps";`  
`char fish[] = "Bubbles";`

↑  
这种字符串被称为字符串常量(string constant)  
或字符串面值(string literal).

用引号括起来的字符串限制地包括结尾的空字符串，因此不用显示地写在末尾。

\* 字符串常量(使用双引号)不能与字符串常量(使用单引号)互换。

Ex: `char shift_size = 's';` // 字符常量 's' 是字符串编码的简写表示。在 ASCII 系统上，  
's' 只是 83 的另一种写法。



```
1 #include<iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     char short_size = 's';
8     int num = short_size;
9     cout << short_size << endl;
10    cout << num << endl;
11
12    return 0;
13 }
```

SQL CONSOLE 问题 输出 调试控制台 终端  
aiden@MacBook-Pro C++ % ./a.out  
s  
115  
aiden@MacBook-Pro C++ %

但“s”不是字符串，它表示的是两个字符（字符 s 和 \0）组成的字符串。

#### 4.2.2 在数组中使用字符串

- 将数组初始化为字符串常量。
- 将键盘或文件输入读入到数组中。

strlen()：返回的是存储在数组中的字符串的长度，而不是数组本身的长度。

另外，strlen()只计算可见的字符，而不把空字符计算在内。

※ cin 如何确定已完成字符串输入？

cin 使用空白（空格、制表符和换行符）来确定字符串的结束位置。

这意味着 cin 在获取字符数组输入时只读取一个单词，读取该单词后，cin 将该字符串放到数组中，并自动在结尾添加空字符。

## 4.2.4 每次读取一行字符串输入

需采用面向行而不是面向单词的方法

istream 中的类 (如 cm) 提供了一些面向行的类成员函数:

· getline ()    都读取一行输入, 直到到达换行符,

getline()    将跳过换行符。

get ()    将移行符保留在输入序列中。

### 1. 面向行的输入: getline()

通过回车键输入的换行符来确定输入结尾。

如何调用? cm.getline() ← () 里有两个参数。

ex: cm.getline(name, 20);    第一个参数用来存储输入行的数据的名称。  
                                ↑ 第二个参数是需要读取的字符数。

// 这将把一行读入到 name 数组中 —— 如果这行字符串不超过 19 个

~~~~~ 正确地使用 cin 和 cout 喜欢的甜点! getline() 函数每次读取一行换行符。相反, 在存储字符串时, 它用空字符来替换换行符 (参见

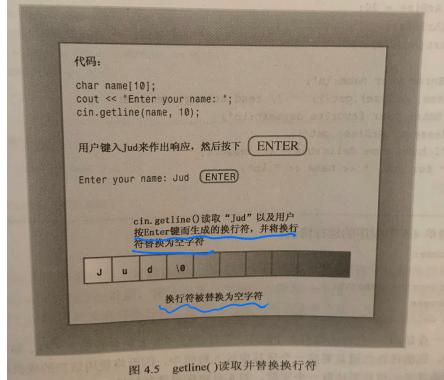


图 4.5 getline() 读取并替换换行符

### 2. 面向行的输入: get()

get() 函数与 getline() 函数类似, 也接受两个参数, 解释参数的方式也相同, 并且都读到行尾。

但 get() 保留换行符

因此, ex:

```

    cin.get(name, ArSize);
    cin.get(dessert, ArSize); // a problem, 第一次调用中换行符留在队列中, 因此第二次
                            // 调用时看到的第一个字符便是换行符, get()认为
                            // 已到达行尾, 而没有发现任何可读取的内容;

```

解决办法: 不需要参数  
 解决办法: cin.get() 可以读取下一个字符(即使是换行符).

所以, 正确做法:

```

    [ ]   cin.get(name, ArSize);      // read first line
          cin.get();                // read new line
    等同于
          cin.get(dessert, ArSize);  // read second line

```

也可用拼接的方法:

```
    cin.get(name, ArSize).get();
```

```

1 #include<iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int size = 100;
8     char name1[size];
9     char name2[size];
10
11    cout << "Enter your name: " << endl;
12    cin.getline(name1, size);
13
14    cout << "Enter your address: " << endl;
15    cin.getline(name2, size);
16
17    cout << "Your name is: " << name1 << endl;
18    cout << "Your address is: " << name2 << endl;
19
20    return 0;
21 }

```

SQL CONSOLE 问题 输出 调试控制台 终端

```

aiden@MacBook-Pro C++ % ./a.out
Enter your name:
ZW
Enter your address:
5054 Sw Technology loop apt.81
Your name is: ZW
Your address is: 5054 Sw Technology loop apt.81

```

```

1 #include<iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int size = 100;
8     char name1[size];
9     char name2[size];
10
11    cout << "Enter your name: " << endl;
12    cin.get(name1, size);
13
14    cout << "Enter your address: " << endl;
15    cin.get(name2, size);
16
17    cout << "Your name is: " << name1 << endl;
18    cout << "Your address is: " << name2 << endl;
19
20    return 0;
21 }

```

SQL CONSOLE 问题 输出 调试控制台 终端

```

aiden@MacBook-Pro C++ % ./a.out
Enter your name:
ZW
Enter your address:
5054 Sw Technology loop apt.81
Your name is: ZW
Your address is: 5054 Sw Technology loop apt.81

```

如果给 cout 提供一个字符串的地址, 则它将从该字符串开始打印, 直到遇到空字符串为止.

在 C++ 中, 用引号括起的字符串像数组名一样, 也是第一个元素的地址.

ex: cout << " s are red\n";

该字符串代表第一个元素的地址.

### 4.3. string 类简介

ISO/ANSI C++98 标准通过添加 string 类扩展了 C++ 库,

因此现在可以使用类型的变量(使用 C++ 的话说是对象)而不是字符串数组来存储字符串.

\* 要使用 string 类，必须在程序中包含头文件 string.

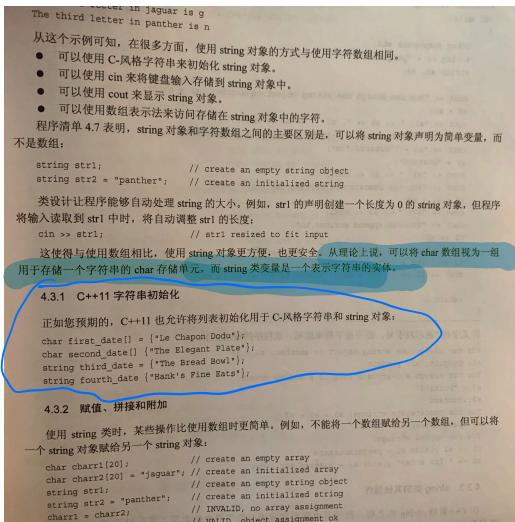
#include <string>

string 类定义隐藏了字符串的数据性质，可以让我们像处理普通变量那样处理字符串。

Ex: string str1;  
string str2 = "panther";

在很多方面，使用 string 对象的方式与使用字符数组相同

- 可以使用 C-风格字符串来初始化 string 对象
- 可以使用 cin 来将键盘输入存储到 string 对象中。 // cin >> str1;
- 可以使用 cout 来显示 string 对象。 // cout << str1;
- 可以使用数组表示法来访问存储在 string 对象中的字符。 // cout << str[2];



## 4.3.2. 赋值、拼接和附加

string str1;  
string str2 = "panther";

`str1 = str2;`

`string` 类简化了字符串合并操作。可以使用运算符`+`将两个`string`对象合并起来。

ex: `str3 = str1 + str2;`

`str1 += str2;`

#### 4.3.3. `string` 类的其他操作

对于 C-风格字符串，程序员使用 C 语言库中的函数来完成这些任务。

头文件 `cstring` (以前为 `string.h`) 提供了这些函数。

#include <cstring>

ex: `char char1[20];`  
`char char2[20] = "Jaguar";`

`strcpy()`: 将字符串复制到字符数组中。

`strcpy(char1, char2);`

`strcat()`: 将字符串附加到字符串数组末尾。

`strcat(char1, " juice");`

如何返回字符串包含的字节数？

- C-风格

`strlen()` → 一个常规函数

ex: `strlen(char1);`

- `string` 类

`size()` → `size()` 是 `string` 类的一个方法。

ex: `str1.size();`

## 4.3.4. Struct 类 I/O

### 4.4. 结构简介

结构也是 C++ OOP 堡垒(类)的基础

结构是用户定义的类型，而结构声明定义了这种类型的数据属性。

创建结构分两步：

- ① 定义结构描述 —— 描述并标记了能够存储在结构中的各种数据类型。
- ② 按描述创建结构变量 (结构数据对象)

创建完结构变量后，可以使用成员运算符(.) 来访问各个成员。

```
Ex: struct Inflatable
{
    char name[20];
    float volume;
    double price;
};

Inflatable hat;
hat.volume = ...;
```

两种初始化方式：

```
Inflatable ghost =
{
    "Glorious Glorin",
    1.8,
    29.99
};
```

#### 4.4.4 结构数组

→ 一个包含 100 个 Inflatable 结构的数组。

ex: Inflatable gifts[100];

gifts 将是一个 Inflatable 数组，其中的每个元素（如 gifts[0] 或 gifts[99]）都是 Inflatable 对象。

gifts 本身是一个数组，而不是结构，所以像 gifts.price 这样的表达是无效的。

这样，gifts 将是一个 inflatable 数组，其中的每个元素（如 gifts[0] 或 gifts[99]）都是 inflatable 对象。与成员运算符一起使用，以访问单个对象的成员。

```
cin >> gifts[0].volume; // use volume member of first struct
cout << gifts[99].price << endl; // display price member of last struct
记住，gifts 本身是一个数组，而不是结构，因此像 gifts.price 这样的表达是无效的。
要初始化结构数组，可以结合使用初始化数组的规则（用逗号分隔每个元素的值，并将这些值括在花括号内）和初始化结构的规则（用逗号分隔每个成员的值，并将这些值用花括号括起）。由于数组中的元素都是结构，因此可以使用结构初始化的方式来提供它的值。因此，最终结果为一个被括在花括号内并用逗号分隔的值列表，其中每个值本身又是一个被括在花括号中并用逗号分隔的值列表：
```

```
inflatable guests[2] = {
    {"Bambi", 0.5, 21.99}, // first structure in array
    {"Godzilla", 2000, 565.99} // next structure in array
};
可以按自己喜欢的方式来格式化它们。例如，两个初始化位于同一行，而每个结构成员的初始值位于一行。
程序清单 4.13 是一个使用结构数组的简短示例。由于 guests 是一个 inflatable 数组，因此 guests[0] 的类型为 inflatable，可以使用它和句点运算符来访问相应 inflatable 结构的成员。
```

程序清单 4.13 arrstruc.cpp

```
// arrstruc.cpp -- an array of structures
#include <iostream>
```

#### 4.4.5 结构中的位运算

#### 4.5. 共同体

#### 4.6. 枚举

#### 4.7. 指针和自由存储空间

指针是一个变量，其存储的是值的地址，而不是值本身。

\* 运算符被称为间接值 (indirect value) 或解引用 (dereferencing) 运算符，将其应用于指针，可以得到该地址存储的新值。

运用地址运算符(&),就可获得变量地址.

ex: int donuts = 5;  
cout << &donuts; // 地址输出

\* int\* p1, p2; // 声明了一个指针(p1)和一个地址(p2)

对每个指针变量,都需要使用一个\*

指针变量不仅仅是指针,而且是指向特定类型的指针

#### 4.7.2. 指针的危险.

- \* C++中创建指针时,计算机将分配用来存储地址的内存,但不会分配用来存储指针所指向的基本型的内存.
- \* 一定要在对指针应用解释引用运算符(\*)之前,将指针初始化为一个有效的、适当的地址.

#### 4.7.3. 指针和动态内存

#### 4.7.4. 使用 new 来分配内存

前面我们都说将指针初始化为变量的地址; 变量是在编译时分配的有名存内存,而指针只是可以通过名直接访问的内存提供了一个别名.

指针真正用或之地: 在运行阶段分配未命名的内存以存储值。  
在这种情况,只能通过指针来访问内存.

在运行阶段为一个 int 值分配未命名的内存,并使用指针来访问这个值:

int\* pn = new int;  
// new 将找到一个长真正确的内存块,并返回该内存块的地址,  
// 负责的职责是将该地址赋给一个指针.

new int 告诉程序,需要适合存储 int 的内存。new 运算符根据类型来确定需要多少字节的内存。然后,它找到这样的内存,并返回其地址。

比较:

① int\* pn = new int; 与 ② int higgins;  
                        int\* pt = &higgins;

①,②都是将一个 int 变量的地址赋给了指针.

② 可以通过名称 `hingers` 来访问该指针

① R能通过该指针进行访问

问题：P指向的内存没有名称，如何称呼？

P指向一个数据对象，此“对象”不是面向对象编程中的对象，而是一样“东西”。

术语“数据对象”比“变量”更通用，它指的是为数据项分配的内存块。因此，变量也是数据对象，但 P 指向的内存不是变量。

为一个数据对象（可以是结构，也可以是基本类型）获得并指定分配内存的通用格式如下：

```
typeName* pointer-name = new typeName;
```

\*: new 分配的内存块通常与常规变量声明分配的内存块不同。

变量的值都存储在被称为栈(stack)的内存区域中，

new 从被称为堆(heap)或自由存储区(free store)的内存区域分配内存。

\* C++ 提供了检测并处理内存分配失败的工具

#### 4.7.5 使用 `delete` 释放内存

使用完内存后，通过 `delete` 运算符，归还或释放(`free`)所分配的内存可供程序的其他部分使用。

使用 `delete` 时，后面需加上指向内存块的指针

```
Ex: int* ps = new int;  
...  
delete ps;
```

\* 只能用 `delete` 并释放使用 `new` 分配的内存。然而，对空指针使用 `delete` 是错误的。

#### 4.7.6 使用 `new` 来创建动态数组

通常，对于大型数据（如矩阵、字符串和结构），应使用 `new`。

**静态联编 (static binding)**: 在编译时给数据对象分配内存，意味着数据对象是在编译时加入到内存中的。

必须在编写程序时指定数据对象的长度、大小。

**动态联编 (dynamic binding)**: 数据对象在运行时创建的。（在运行时创建的数据对象是动态数组(dynamic array)）。

程序将在运行时确定数据对象的大小、长短。

即在运行时为数组分配空间，其长度将在运行时设置。

##### 1. 使用 `new` 创建动态数组。

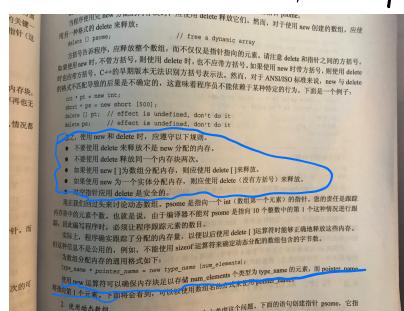
只需将数组的元素类型和元素数目告诉 `new` 即可，必须在类型名后加上方括号，其中包含元素数目。

ex:

`int* psome = new int[10]; // get a block of 10 ints.`

new 运算符返回第一个元素的地址。

记得释放，对于使用 new 分配的数组，应使用另一种形式的 delete 来释放。  
`delete [] psome; // free a dynamic array.`



## 2. 使用动态数组

如何访问动态数组中的其它元素?

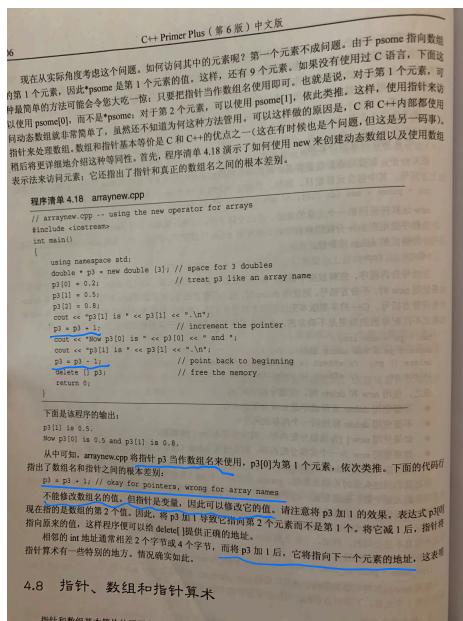
只要把指针当作数组名使用即可

ex:

`psome[0]; // 第一个元素`

`psome[1]; // 第二个元素`

`⋮`



### 4.8 指针、数组和指针算术

指针和数组基本操作的原因在于指针本身。

#### 4.8. 指针、数组和指针算术

指针和数组基址等价的原因在于指针算术 (pointer arithmetic) 和 C++ 内部处理数据的方式。

将整数表是加工后,其值将增加1:

将指针变量加工后，增加的量等于它指向的类型的字节数。

例：将指向 double 类的指针加工后，如果系统对 double 使用 8 个字节存储，则数组将增加 8。

\* C++ 将数组名看作地址。

使用高精度数组表示法 等同于 对指针解除了限制

Ex: `taus[0] = *taus`  
`taus[3] = *(taus+3)`

第4章 集合类

187

程序清单 4.19 *adapters.cpp*

```
// adapters.cpp -- practice addition
#include <iostream>
#include <vector>
using namespace std;
class Doubles {
public:
    Doubles() : start(1.0), end(1000.0, 0.0001, 1) {
        short stack[10] = {1, 2, 3, 4};
        for (short i = 0; i < 10; ++i)
            cout << stack[i] << endl;
    }
    void add() {
        Doubles::iterator p1 = start, p2 = end;
        Doubles::iterator p3 = p1 + 10;
        while (p1 != p3) {
            *p1 += *p2;
            p1++;
            p2++;
        }
        cout << "The sum of the first 10 elements is: " << *p1 << endl;
        cout << "The sum of the last 10 elements is: " << *p2 << endl;
        cout << "The sum of all elements is: " << *p3 << endl;
    }
private:
    Doubles::iterator start;
    Doubles::iterator end;
};

int main() {
    Doubles d;
    d.add();
    return 0;
}
```

Wages <-- wages[0]

wages[1] = wages[0] + wages[1]

wages[2] = wages[0] + wages[1] + wages[2]

wages[3] = wages[0] + wages[1] + wages[2] + wages[3]

wages[4] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4]

wages[5] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5]

wages[6] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6]

wages[7] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7]

wages[8] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7] + wages[8]

wages[9] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7] + wages[8] + wages[9]

wages[10] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7] + wages[8] + wages[9] + wages[10]

wages[11] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7] + wages[8] + wages[9] + wages[10] + wages[11]

wages[12] = wages[0] + wages[1] + wages[2] + wages[3] + wages[4] + wages[5] + wages[6] + wages[7] + wages[8] + wages[9] + wages[10] + wages[11] + wages[12]

下面是函数的输出结果：

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

```
int main() {
    Doubles d;
    d.add();
    return 0;
}
```

```
</
```

4. 在很多情况下，可以相同的方式使用指针名和数组名。

区别之一在于，可以修改滑块的值，数组不是常量，不能修改数组。

`potentname = potentname + 1; // valid.`

另一个区别

对数组应用索引时运算符得到的是数组的长度

对指针应用sizeof得到的是指针的长度，即使指针指向的是一个数组

## 数组的地址

```
1 #include<iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     short tell[10];
8     cout << tell << endl;
9     cout << &tell << endl;
10
11
12 }
```

#### 4.8.4. 使用 new 创建动态结构

通过使用 new，可以创建动态结构。

“动态”意味着内存是在运行时，而不是编译时分配的。

new 用于结构由两步组成：

- 创建结构
- 访问其成员

Ex:

要创建一个未命名的 inflatable 类型，并将其地址赋给一个指针：

```
inflatable * ps = new inflatable; // 把以存储 inflatable 结构的一块可用  
内存的地址赋给 ps.
```

创建 动态结构 时，不能将成员运算符句点用于结构名，因为这种结构没有名称，  
只是知道它的地址。

箭头成员运算符 ( $\rightarrow$ )：可用于指向结构的指针，就像点运算符可用于结构名一样。

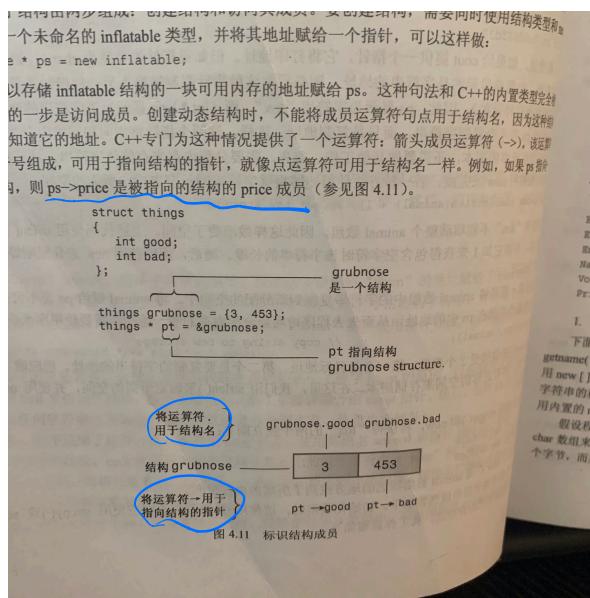


图 4.11 标识结构成员

字符串可能会影响被认为是独立的、位于其他地方的字符串。无论如何，由于 bird 指针被声明为 const，编译器将禁止改变 bird 指向的位置中的内容。

试图将信息读入 ps 指向的位置将更糟。由于 ps 没有被初始化，因此并不知道信息将被存储在哪里，这甚至可能改写内存中的信息。幸运的是，要避免这种问题很容易——只要使用足够大的 char 数组来接收输入即可。请不要使用字符串常量或未被初始化的指针来接收输入。为避免这些问题，也可以使用 std::string 对象，而不是数组。

警告：在将字符串读入程序时，应使用已分配的内存地址。该地址可以是数组名，也可以是使用 new 初始化过的指针。

```
(char*) char 数组: char animal[10];
(choose) 读下来，请注意下述代码完成的工作:
    ps = animal; // set ps to point to string
    ...
cout << animal << endl;
cout << ps << endl;
```

它将生成下面的输出：

```
fox at 0x0065f430
fox at 0x0065f430
```

一般来说，如果给 cout 提供一个指针，它将打印地址。但如果指针的类型为 char\*，则 cout 将显示指向的字符串（如果要显示的是字符串的地址，则必须将这种指针强制转换为另一种指针类型，例如 int\* 上面的代码就是这样做的）。因此，ps 显示为字符串“Fox”，而“cout”ps 显示为该字符串的地址。注意，将 animal 赋给 ps 并不会复制字符串，而是复制地址。这样，这两个指针将指向相同的内存单元和字符串。要获得字符串的副本，还需要做其他工作。首先，需要分配内存来存储该字符串，这可以通过声明另一个数组或使用 new 来完成。后一种方法便能够根据字符串的长度来指定所需的空间：

```
ps = new char[strlen(animal) + 1]; // get new storage
字符串“fox”不能填满整个 animal 数组，因此这样做浪费了空间。上述代码使用 strlen() 来确定字符串的长度，并将它加 1 来获得包含空字符时该字符串的长度。随后，程序使用 new 来分配刚好足够存储该字符串的空间。
```

接下来，需要将 animal 数组中的字符串复制到新分配的空间中。将 animal 赋给 ps 是不可行的，因为这样只能修改存储在 ps 中的地址，从而失去程序访问新分配内存的唯一途径。需要使用库函数 strcpy()：

```
strcpy(ps, animal); // copy string to new storage
```

strcpy() 函数接受 2 个参数。第一个是目标地址，第二个是要复制的字符串的地址。您应确定，分配的空间有足够的空间来存储副本。在这里，我们用 strlen() 来确定所需的 storage。通过使用 strcpy() 和 new，将获得：

## 4.8.5 动态存储、静态存储和动态存储

C++ 有 3 种管理数据内存的方式：

- ① 动态存储
- ② 静态存储
- ③ 动态存储（自动存储空间成堆）

自动存储：

在函数内部定义的常规变量使用自动存储空间，被称为自动变量（automatic variable），这意味着它们在所属的函数被调用时自动产生，在该函数结束时消失。

实际上，自动变量是一个局部变量，其作用域为包含它的代码块。

自动变量通常存储在栈中。这意味着执行代码块时，其中的变量将依次加入到栈中，而在离开代码块时，将按相反的顺序释放这些变量。

因此，在程序执行过程中，栈将不断地增大或减小。

## 静态存储

静态存储是整个程序执行期间都存在的存储方式。

变量成为静态的方式有两种：

① 在函数外面定义它

② 在声明变量时使用关键字 static。

Ex: static double fee = 16.50;

自动存储和静态存储的关键在于：这些方法严格地限制了变量的寿命。

变量可能在程序的整个生命周期（静态变量），

也可能只是在特定函数被执行时存在（自动变量）。

## 动态存储：

new, delete.

它们管理了一个内存池。这在 C++ 中被称为自由存储空间 (free store) 或堆 (heap)。

该内存池同用于静态变量和自动变量的内存是分开的。

在线中，自动添加和删除机制使得占用的内存总是连续的，

但 new 和 delete 的相互影响可能导致占用的自由存储区不连续，

这使得跟踪新分配内存的位置更困难。

## 4.9. 类型组合

### 4.10. 数组和指代品

模板类 vector 和 array 是数组的指代品。

#### 4.10.1 模板类 vector

模板类 vector 类似于 array 类，也是一种动态数组。

① 要使用 vector 对象，必须包含头文件 vector。Ex: #include <vector>

② vector 包含在名称空间 std 中，因此可以使用 std 编译指令、using 声明或 std::vector。

③ 模板使用不同的语法来指出其中有值的数据类型。

④ vector 类使用不同的语法来指定元素数。

一般地，下面的声明创建一个名为 vi 的 vector 对象，它包含 n\_elem 个类型为 typeName 的元素：

vector<typeName> vi(n\_elem); 其中，参数 n\_elem 可以是整型常量，  
也可以是整型变量。

ex: `vector<int> vi;` // 一个初始长度为 0 的类型为 int 的动态数组。

`vector<double> vd(n);` // create an array of n doubles

#### 4.10.2. 模板类 array (C++11)

如果需要的是长度固定的数组，使用模板类 array 是更好的选择。

需 `#include <array>`

与数组一样，array 对象的长度也是固定的，也使用栈（静态内存分配），  
而不是自由存储区，因此效率与数组相同，但更方便，更容易。

一般的，下面的声明创建一个名为 arr 的 array 对象，它包含 n\_elem 个类型为  
typeName 的元素：

array<typeName, n\_elem> arr; n\_elem 不能是变量。

无论是数组、vector 对象还是 array 对象，都可以使用索引表示法来访问各个元素。

ex: `arr[2] = ...;`

### 4.11. 总结

C++ 的 3 种复合类型 {  
    数组  
    结构  
    指针

[2]

数组可以在一个数据对象中存储多个同种类型的值。通过使用索引或下标，  
可以访问数组中各个元素。

中括号表示法或数组的成员函数，都可以在一个数据对象中存储多个不同类型的值。  
但是，这种表示法或成员函数只能处理一种类型的值。如果要处理多种类型的值，就需要使用另一种表达这种类型的语句。例如，  
如果要处理多种类型的值，那么就无法使用中括号或成员函数。这时，可以使用指针或引用。但是，这两种方法都需要手动管理内存，  
而且，它们的实现方式非常复杂。因此，建议使用 std::vector 或 std::array。这两种方法都提供了自动管理内存的功能，  
并且，它们的实现方式相对简单。

具体来说，可以创建一个数组，但是这个数组只能存储一种类型的值，而且必须手动管理内存。而使用 std::vector 或 std::array，  
则可以自动管理内存。也就是说，不需要手动管理内存，只需要提供一个指向内存的指针，或者是一个引用。这样，就可以  
在使用时直接使用，而不需要手动管理内存。这对于处理多种类型的值来说，非常方便。

对于数组，如果要处理多种类型的值，那么就需要手动管理内存。这是因为，数组的大小是固定的，而且不能动态调整。  
而 std::vector 和 std::array 则可以处理这种情况。它们的大小是动态的，可以根据需要增加或减少。这样，就可以避免手动管理内存的麻烦。

对于指针和引用，如果要处理多种类型的值，那么就需要手动管理内存。因为，它们的类型是不同的。如果要处理多种类型的值，  
那么就需要手动转换类型。而 std::vector 和 std::array 则可以自动处理这种情况。这样，就可以避免手动转换类型的麻烦。

总的来说，std::vector 和 std::array 是处理多种类型的值的最佳选择。它们提供了自动管理内存的功能，而且实现方式相对简单。  
因此，建议在处理多种类型的值时，优先使用 std::vector 或 std::array。这样，就可以避免手动管理内存的麻烦。

自动变量在程序执行到其所属的代码块(通常函数体内)时产生，  
在离开该代码块时终止。

静态变量在整个程序周期内都存在。

模板类 `vector` 是动态数组的替代品。

模板类 `array` 是固定大小的替代品。