**Chapter 4 Compound Types**

The **array**, for example, can **hold several values of the same type**. A particular kind of array can hold a string, which is a series of characters.

* **Introducing Arrays**

An **array** is a data form that can hold several values, **all of one type**. Each value is stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.

short months[12]; // creates array of 12 short

typeName arrayName[arraySize];

***arraySize***, which is the number of elements, must be an **integer constant**, such as 10 or a **const value**, or a **constant expression**, such as 8 \* sizeof (int), **for which all values are known at the time compilation** takes place. In particular, **arraySize cannot be a variable whose value is set while the program is running**.

The compiler does not check to see if you use a valid subscript. For instance, the compiler won’t complain if you assign a value to the nonexistent element months[101].

int yamcosts[3] = {20, 30, 5}; // sizeof yamcosts == 12 bytes;

// sizeof yamcosts[0] == 4 bytes;

**If you don’t initialize an array** that’s defined inside a function, **the element values remain undefined. That means the element takes on whatever value previously resided at that location in memory.**

The ***sizeof*** operator **returns the size, in bytes, of a type or data object**. Note that if you use the sizeof operator **with an array name, you get the number of bytes in the whole array**. But if you use sizeof with an array element, you get the size, in bytes, of the element.

* **Initialization Rules for Arrays**

**You can use the initialization form *only when* defining the array**. You cannot use it later, and you cannot assign one array wholesale to another:

int cards[4] = {3, 6, 8, 10}; // okay

int hand[4]; // okay

hand[4] = {5, 6, 7, 9}; // not allowed

hand = cards; // not allowed

However, you can use subscripts and assign values to the elements of an array individually.

When initializing an array, you can provide fewer values than array elements. For example, the following statement initializes only the first two elements of hotelTips:

float hotelTips[5] = {5.0, 2.5};

**If you partially initialize an array, the compiler sets the remaining elements to zero**. Thus, it’s easy to initialize all the elements of an array to zero—just explicitly initialize the first element to zero and then let the compiler initialize the remaining elements to zero:

long totals[500] = {0}; // all elements are 0;

long totals[500] = {1}; // first element is 1; all other elements are 0;

Note that if you initialize to {1} instead of to {0}, just the first element is set to 1; the rest still get set to 0.

**If you leave the square brackets ([]) empty** when you initialize an array, the **C++ compiler counts the elements for you.** Suppose, for example, that you make this declaration:

short things[] = {1, 5, 3, 8};

**The compiler makes things an array of four elements**.

short things[] = {1, 5, 3, 8};

int num\_elements = sizeof things / sizeof (short);

这样也可以，但是同样的，长度是不能修改的；

* **C++11 Array Initialization**

double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4}; // no =, okay with C++11

unsigned int counts[10] = {}; // all elements set to 0

float balances[100] {}; // all elements set to 0

int plifs[] = {25, 92, 3.0}; // not allowed, floating point can’t be casted

// to int;

char slifs[4] {'h', 'i', 1122011, '\0'}; // not allowed, 1122011 outside range of char

char tlifs[4] {'h', 'i', 112, '\0'}; // allowed, 112 (default int) in range of char

The C++ Standard Template Library (STL) provides an alternative to arrays called the vector template class, and **C++11 adds an array template class**.

* **String**

A string is a series of characters stored in **consecutive** bytes of memory. **C++ has two ways of dealing with strings**. The first, taken from C and often called a **C-style string**, is the first one this chapter examines. Later, this chapter discusses an alternative method based on a **string class library**.

The idea of a series of characters stored in consecutive bytes implies that you can store a string in an array of char.

**C-style strings have a special feature**: **The last character of every string is the null character**. This character, written \0, is the character with ASCII code 0, and it serves to mark the string’s end. For example, consider the following two declarations of **C-style**:

char dog[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'}; // **not a string!**

char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'}; **// a string!**

**Both of these arrays are arrays of char, but only the second is a string**. The null character plays a fundamental role in C-style strings. For example, C++ has many functions that handle strings, including those used by cout. **They all work by processing a string character- by-character until they reach the null character**. If you ask cout to display a nice string like cat in the preceding example, it displays the first seven characters, detects the null character, and stops. But if you are ungracious enough to tell cout to display the dog array from the preceding example, which is not a string, **cout prints the eight letters in the array and then keeps marching through memory byte-by-byte**, interpreting each byte as a character to print, until it reaches a null character. **Because null characters, which really are bytes set to zero, tend to be common in memory, the damage is usually contained quickly**; nonetheless, you should not treat non-string character arrays as strings.

There is a better way to initialize a character array to a string**. Just use a quoted string**, called a **string constant or string literal**, as in the following:

char bird[11] = "Mr. Cheeps"; **// the \0 is understood**

char fish[] = "Bubbles"; **// let the compiler count**

**Quoted strings always include the terminating null character implicitly**, so you don’t have to spell it out. Also the various **C++ input facilities for reading a string from keyboard input into a char array automatically add the terminating null character for you**.

Of course, you should make sure the array is large enough to hold all the characters of the string, including the null character. There is no harm, other than wasted space, in making an array larger than the string. That’s because functions that work with strings are guided by the location of the null character, not by the size of the array. C++ imposes no limits on the length of a string.

**Note that a string constant (with double quotes) is not interchangeable with a character constant (with single quotes)**. A character constant, such as **'S'**, is a shorthand notation for the code for a character. On an ASCII system, **'S' is just another way of writing 83. Thus, the following statement assigns the value 83 to shirt\_size**:

char shirt\_size = 'S'; **// this is fine**

**But "S" is not a character constant; it represents the string consisting of two characters**, the S and the \0 characters. **Even worse, "S" actually represents the memory address at which the string is stored. So a statement like the following attempts to assign a memory address to shirt\_size**:

char shirt\_size = "S"; **// illegal type mismatch**

Because an address is a separate type in C++, a C++ compiler won’t allow this sort of nonsense.

* **Get string length**

#include <cstring> // for the strlen() function

int main()

{

using namespace std;

**const int Size = 15;**

char name1[Size]; // empty array

char name2[Size] = "C++owboy"; // initialized array

cout << "Howdy! I'm " << name2;

cout << "! What's your name?\n";

**cin >> name1;**

cout << "Well, " << name1 << ", your name has ";

cout << **strlen(name1)** << " letters and is stored\n";

cout << "in an array of " << **sizeof(name1)** << " bytes.\n";

cout << "Your initial is " << name1[0] << ".\n";

name2[3] = '\0'; // set to null character

cout << "Here are the first 3 characters of my name: ";

**cout << name2 << endl;**

return 0;

}

一个示例下的运行结果：

Howdy! I'm C++owboy! What's your name?

**Basicman**

Well, Basicman, your name has **8 letters** and is stored

in an array of **15 bytes**.

Your initial is B.

Here are the first 3 characters of my name: **C++**

The program also uses the **standard C library function *strlen()*** to get the length of a string.

1. 可以直接用cin >> arrayname或者cout << arrayname的形式直接输入输出字符串；
2. 实例中name1和name2都是string;
3. ***sizeof*** **operator** **gives the size of the entire array, 15 bytes**, but the ***strlen()*** **function** **returns the size of the string stored in the array and not the size of the array itself**. Also ***strlen()* counts just the visible characters and not the null character**. Thus, it returns a value of 8, not 9, for the length of Basicman. If cosmic is a string, the minimum array size for holding that string is strlen(cosmic) + 1.

char shirt\_size = "String"; **// illegal type mismatch**

char shirt\_size[8] = "String"; **// Fine**

char shirt\_size[] = "String"; **// Fine**

**注意：在将用“”形成的string给别的数据类型赋值时，这个string其实给出的是个地址;**

* **Adventures in String Input**

**<List4-3.cpp>**

#include <iostream>

int main()

{

using namespace std;

const int ArSize = 20;

**char name[ArSize];**

**char dessert[ArSize];**

cout << "Enter your name:\n";

cin >> name;

cout << "Enter your favorite dessert:\n";

cin >> dessert;

cout << "I have some delicious " << dessert;

cout << " for you, " << name << ".\n";

return 0;

}

一个示例下的运行结果：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./a.out

Enter your name:

***Jian Lao***

Enter your favorite dessert: **// 这里根本没让你输入就直接跳过去了**

I have some delicious **Lao** for you, **Jian**. **// Lao前面只有程序本身加的空格，没有额外的**

The problem lies with how cin determines when you’ve finished entering a string. **You can’t enter the null character from the keyboard,** so cin needs some other means for locating the end of a string. **The cin technique is to use whitespace - *spaces, tabs, and newlines* - to delineate a string**. **This means cin reads just one word when it gets input for a character array**. After it reads this word, cin automatically adds the terminating null character **when it places the string into the array**.

The practical result in this example is that **cin reads Jian as the entire first string** **and puts it into the name array**. This leaves **poor Lao still sitting in the input queue**. When **cin searches the input queue for the response to the favorite dessert question, it finds Lao still there**. **这里cin第二次执行时实际上是看见了一个space符和后面的Lao, cin会跳过它每次执行时开头的whitespace符直接从非whitespace读起.** Then cin gobbles up Lao and puts it into the dessert array.

**Cin的另一个问题是它不会做长度检查**，**比如把上面的ArSize改为8**：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./a.out

Enter your name:

***ABCDEFGHIJKLMN OPQRSTUVWXYZ* （这是键盘上的输入）**

Enter your favorite dessert:

I have some delicious **OPQRSTUVWXYZ** for you, **ABCDEFGHIJKLMN**.

\*\*\* stack smashing detected \*\*\*: ./a.out terminated

Aborted (core dumped)

momo@HMI:~/C++PrimerPlus/Chapter4$

明明指定了两个数组长度是8，但最后都输出了10多个字符;

* **Reading String Input a Line at a Time**

Specifically, **you need a line-oriented method** instead of a word-oriented method. You are in luck, for the **istream** class, **of which cin is an example**, has some line-oriented class member functions: **getline()** and **get()**. **Both read an entire input line** - that is, **up until a newline character**. However, **getline() then discards the newline character**, whereas **get() leaves it in the input queue**.

1. **Line-Oriented Input with getline()**

**cin.getline()** takes two arguments. The first argument is the **name of the target** (that is, the array destined to hold the line of input), and the **second argument is a limit on the number of characters to be read**. If **this limit is, say, 20, the function reads no more than 19 characters, leaving room to automatically add the null character at the end**. The getline() member function **stops reading input when** it reaches this numeric limit or when it reads a newline character, whichever comes first. It reads input through the newline character marking the end of the line, but **it doesn’t leave the newline character (in input queue). Instead, it replaces it with a null character when storing the string** .

键盘输入最后敲得回车键就是newline符;

键盘连续输入的字符一定是string, 除非把char array拆开一个一个输入;

**<List4-4.cpp>**

#include <iostream>

int main()

{

using namespace std;

const int ArSize = **6**;

char name[ArSize];

char dessert[ArSize];

cout << "Enter your name:\n";

**cin.getline(name, ArSize);**

cout << "Enter your favorite dessert:\n";

**cin.getline(dessert, ArSize);**

cout << "I have some delicious " << dessert;

cout << " for you, " << name << ".\n";

return 0;

}

运行结果为：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-4

Enter your name:

***ABCDEFG*（这是键盘上的输入）**

Enter your favorite dessert: **// 这里根本没让你输入就直接跳过去了，因为溢出而输入关闭了**

I have some delicious for you, **ABCDE**.

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-4

Enter your name:

**ABCDE**

Enter your favorite dessert:

**HIJKLM**

I have some delicious **HIJKL** for you, **ABCDE**.

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-4

Enter your name:

**(什么也不输入，直接回车)**

Enter your favorite dessert:

ABC

I have some delicious ABC for you, .

momo@HMI:~/C++PrimerPlus/Chapter4$

1. **Line-Oriented Input with get()**

**cin.get()** takes the same arguments, **interprets them the same way**, and reads to the end of a line. **But rather than read and discard the newline character, get() leaves that character in the input queue**.

**意思是说, cin.get()在把input queue里的数据copy到目标数组里的时候，会把结尾的newline character留在input queue里；**

cin.get(name, ArSize);

cin.get(dessert, Arsize); **// a problem**

Because **the first call leaves the newline character in the input queue**, that newline character is the first character the second call sees. **Thus, get() concludes that it has reached the end of line without having found anything to read.** Without help, get() just can’t get past that newline character.

**Solution:**

cin.get(name, ArSize); // read first line

**cin.get();** **// read newline**

cin.get(dessert, Arsize); // read second line

Another way to use get() is to concatenate, or join, the two class member functions, as follows:

cin.get(name, ArSize)**.get()**; // concatenate member functions

What makes this possible is that cin.get(name, ArSize) returns the cin object, which is then used as the object that invokes the get() function.

cout << "Enter your name:\n";

**cin.get(name, ArSize).get();** **// read string, newline**

cout << "Enter your favorite dessert:\n";

**cin.get(dessert, ArSize).get();**

cin.get() itself, without any parameters, reads one character.

上面的getline()也可以这么用：

cin.getline(name1, ArSize).getline(name2, ArSize);

**Why use get() instead of getline() at all**? First, older implementations may not have getline(). Second, **get() lets you be a bit more careful**. Suppose, for example, you used get() to read a line into an array. **How can you tell if it read the whole line rather than stopped because the array was filled? Look at the next input character.** If it is a newline character, then the whole line was read. If it is not a newline character, then there is still more input on that line. Chapter 17 investigates this technique. In short, getline() is a little simpler to use, but get() makes error checking simpler. You can use either one to read a line of input; just keep the slightly different behaviors in mind.

**A potential problem** is that the **input string could be longer than the allocated space**. If the input line is longer than the number of characters specified, **both getline() and get() leave the remaining characters in the input queue. However, getline() additionally sets the failbit and turns off further input**.

* **Empty Lines and Other Problems**

The current practice is that after **get() (but not getline())** reads an empty line, it sets something called the failbit. The implications of this act are that further input is blocked, but you can restore input with the following command:

cin.clear();

* **cin, cin.getline()和cin.get()的总结**

键盘输入的数据进入input queue这个过程不受这三者任何一个控制; 这三者都只从input queue中读数据，差别在于如何从input queue中读出数据;

1. cin

* cin在从input queue中读取字符串时，以whitespace - spaces, tabs, and newlines, 作为一次调用的结束标志；

~~如果这个结束标志是spaces或者tabs，它会在把数据copy到目标字符串中时把这两个结束标志从input queue中提走，换成Null character添加到目标结尾处；如果这个结束标志是newline，它会在把数据copy到目标字符串中时把newline标志留在input queue中，自己添加Null character到被提走的数据结尾；~~

**拷贝数据时会把这3种whitespace留在input queue里，自己添加Null character到被提走的数据结尾；**

* cin在读数据时，如果发现input queue头部是个spaces, tabs, 或者newlines，会直接跳过读取下一个数据，或者触发用户输入；他必须读到非whitespace的字符才算一次有效输入；cin不把开头的whitespace当作字符的输入；
* 在cin往目标字符串中填数据时，它不会管目标字符串是否够大，直接把它从input queue中带来的数据全部往里填；

1. cin.getline()

* cin.getline()在从input queue中读取字符串时，以newlines(回车键)作为一次调用的结束标志，并且它会在把数据copy到目标字符串中时把这个结束标志从input queue中提走，自己添加Null character到被提走的数据结尾；
* cin.getline()在碰到空白输入(只敲击了回车键)时，会把这当作空白赋给目标数组，然后进行下一次输入；
* cin.getline()会关注它从input queue中提出的数据长度是否大于目标长度，如果真的大了，本次赋值仍然进行，按照目标的大小赋值，多余的留在input queue中，并将input关闭；

1. cin.get()

* cin.get()在从input queue中读取字符串时，以newlines(回车键)作为一次调用的结束标志，但是，它在把数据copy到目标字符串中时，会把这个结束标志留在input queue中，自己再在已经提出来的数据后面加上Null character；
* cin.get()在从input queue中读取字符串时，**如果一开始就是newline**，直接将这个newline从input queue中提出扔掉，然后结束本次操作；
* cin.get()在碰到空白输入(只敲击了回车键)时，会设置failbit，并将输入关闭；
* cin.get()会关注它从input queue中提出的数据长度是否大于目标长度，如果真的大了，本次赋值仍然进行，按照目标的大小赋值，多余的留在input queue中，但不会将input关闭；

这三者都是在判断input queue中无数据时才会触发用户键盘输入，就是终端处于输入状态，只不过他们判断input queue是否为空的标准不同；

Listing 4.6

#include <iostream>

int main()

{

using namespace std;

cout << "What year was your house built?\n";

**int year;**

cin >> year;

cout << "What is its street address?\n";

char address[80];

**cin.getline**(address, 80);

cout << "Year built: " << year << endl;

cout << "Address: " << address << endl;

cout << "Done!\n";

return 0;

}

运行结果为：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-6

What year was your house built?

**1966**

What is its street address?

Year built: 1966

Address: **// 这里根本没让你输入就直接跳过去了**

Done!

momo@HMI:~/C++PrimerPlus/Chapter4$

这里的原因是，键盘敲入了1966和一个回车键，cin根据类型int只读入1966的部分，**把回车符留在了input queue里了(这是书上的说法)**；而下一个输入又正好是getline()，是以newline符为结束依据的，所以直接跳过了；

**如果把cin.getline(address, 80)换成cin >> address**:

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-6

What year was your house built?

1966

What is its street address?

**1386 turrett dr**

Year built: 1966

Address: **1386**

Done!

momo@HMI:~/C++PrimerPlus/Chapter4$

说白了，cin一定要以非whitespace符为开始，如果开头是whitespace符，它就跳过或者触发用户输入；

开始后一遇到那3个whitespace符就结束；**拷贝数据时会把这3中whitespace留在input queue里；**

getline()和get()一看见newline就结束，不管在哪里看见的；

#include <iostream>

int main()

{

using namespace std;

cout << "What year was your house built?\n";

char year[50];

cin >> year;

cout << "What is its street address?\n";

char address[80];

cin.getline(address, 80);

cout << "Year built:" << year << endl;

cout << "Address:" << address << endl;

cout << "Done!\n";

return 0;

}

运行结果：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-6

What year was your house built?

1234 ABCD

What is its street address?

Year built:1234

Address: ABCD **// 这里开头有留下的space符**

Done!

momo@HMI:~/C++PrimerPlus/Chapter4$ cat List4-6.cpp

Solution:

cin >> year;

cin.get();

or:

(cin >> year).get();

再看个例子：

<TestGet.cpp>

#include <iostream>

int main()

{

using namespace std;

char str1[10];

char str2[10];

cout << "Input multiple-word string str1:" << endl;

cin >> str1;

cout << "str1 is:" << str1 << endl;

cout << "cin.get() multiple-word string str2:" << endl;

cin.get(str2, 10);

cout << "str2 is:" << str2 << endl;

return 0;

}

运行结果是：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./TestGet

Input multiple-word string str1:

***ABC DEF***

str1 is:ABC

cin.get() multiple-word string str2:

str2 is: DEF **// 这里开头有留下的space符**

momo@HMI:~/C++PrimerPlus/Chapter4$ ./TestGet

Input multiple-word string str1:

***ABC***

str1 is:ABC

cin.get() multiple-word string str2:

str2 is: **// get()碰到的第一个字符就是newline，直接扔掉退出；**

momo@HMI:~/C++PrimerPlus/Chapter4$

* **Introducing the string Class**

The ISO/ANSI C++98 Standard expanded the C++ library by adding a string class. To use the string class, a program has to include the **string header file**. **The string class is part of the std namespace**, so you have to provide a using directive or declaration or else refer to the class as std::string.

#include <iostream>

**#include <string> // make string class available**

int main()

{

using namespace std;

char charr1[20];

char charr2[20] = "jaguar"; // create an initialized array

**string str1**; // create an empty string object

**string str2 = "panther";** // create an initialized string

cout << "Enter a kind of feline: ";

cin >> charr1;

cout << "Enter another kind of feline: ";

**cin >> str1;** // use cin for input

cout << "Here are some felines:\n";

cout << charr1 << " " << charr2 << " "

<< str1 << " " << str2

<< endl;

cout << "The third letter in " << charr2 << " is "

<< charr2[2] << endl;

cout << "The third letter in " << str2 << " is "

<< str2[2] << endl;

return 0;

}

You should learn from this example that, in many ways, you can use a string object in the same manner as a character array.

The **main difference** between string objects and character arrays shown above is that you declare a **string object as a simple variable, not as an array.** The class design allows the program to handle the sizing automatically. For instance, the declaration for str1 creates a string object of length zero, but the program automatically resizes str1 when it reads input into str1. This makes using a string object both more convenient and safer than using an array. A string class variable is a **single entity** representing the string.

* **C++11 String Initialization**

C++11 enable list-initialization for **C-style strings** and string objects:

char first\_date[] = {"Le Chapon Dodu"};

char second\_date[] {"The Elegant Plate"};

string third\_date = {"The Bread Bowl"};

string fourth\_date {"Hank's Fine Eats"};

1. **You can’t simply assign one array to another. But you can assign one string object to another:**

char charr1[20]; // create an empty array

char charr2[20] = "jaguar"; // create an initialized array

string str1; // create an empty string object

string str2 = "panther"; // create an initialized string

charr1 = charr2; **// INVALID, no array assignment**

str1 = str2; **// VALID, object assignment ok**

1. **You can use the + operator to add two string objects together and the += operator to tack on a string to the end of an existing string object.** S1 += S2相当于S1 = S1 + S2;

using namespace std;

string s1 = "penguin";

string s2, s3;

s2 = "buzzard";

s3 = s1 + s2; **// s3 = penguinbuzzard**

s1 += s2; **// s1 = penguinbuzzard**

s2 += " for a day"; **// s2 = buzzard for a day**

1. **For C-style strings**, they used functions from the C library for these tasks. The cstring header file (formerly string.h) supports these functions. For example, you can use the strcpy() function to copy a string to a character array, and you can use the strcat() function to append a string to a character array:

strcpy(charr1, charr2); **// copy charr2 to charr1**

strcat(charr1, charr2); **// append contents of charr2 to char1**

下面的例子比较了C和C++相关的几个操作：

#include <iostream>

#include <string> // make string class available

#include <cstring> **// C-style string library**

int main()

{

using namespace std;

char charr1[20];

char charr2[20] = "jaguar";

string str1;

string str2 = "panther";

// assignment for string objects and character arrays

str1 = str2; // copy str2 to str1

strcpy(charr1, charr2); // copy charr2 to charr1

// appending for string objects and character arrays

str1 += " paste"; // add paste to end of str1

strcat(charr1, " juice"); // add juice to end of charr1

// finding the length of a string object and a C-style string

**int len1 = str1.size(); // obtain length of str1**

**int len2 = strlen(charr1); // obtain length of charr1**

cout << "The string " << str1 << " contains "

<< len1 << " characters.\n";

cout << "The string " << charr1 << " contains "

<< len2 << " characters.\n";

return 0;

}

运行结果是：

The string panther paste contains 13 characters.

The string jaguar juice contains 12 characters.

For example, the C library equivalent of

str3 = str1 + str2;

is this:

strcpy(charr3, charr1);

strcat(charr3, charr2);

Furthermore, **with arrays, there is always the danger of the destination array being too small to hold the information**, as in this example:

char site[10] = "house";

strcat(site, " of pancakes"); // memory problem

**The strcat() function would attempt to copy all 12 characters into the site array, thus overrunning adjacent memory**. This might cause the program to abort, or the program might continue running but with corrupted data. **The string class, with its automatic resizing as necessary, avoids this sort of problem**. The C library does provide cousins to strcat() and strcpy(), called strncat() and strncpy(), that work more safely by taking a third argument to indicate the maximum allowed size of the target array, but using them adds another layer of complexity in writing programs.

#include <iostream>

#include <string> // make string class available

#include <cstring> // C-style string library

int main()

{

using namespace std;

char charr[20];

string str;

cout << "Length of string in charr before input: "

<< strlen(charr) << endl;

cout << "Length of string in str before input: "

<< str.size() << endl;

cout << "Enter a line of text:\n";

**cin.getline(charr, 20);**

cout << "You entered: " << charr << endl;

cout << "Enter another line of text:\n";

**getline(cin, str);** **// cin now an argument; no length specifier**

cout << "You entered: " << str << endl;

cout << "Length of string in charr after input: "

<< strlen(charr) << endl;

cout << "Length of string in str after input: "

<< str.size() << endl;

return 0;

}

书上示例中的测试结果为：

**Length of string in charr before input: 27**

Length of string in str before input: 0

Enter a line of text:

peanut butter

You entered: peanut butter

Enter another line of text:

blueberry jam

You entered: blueberry jam

Length of string in charr after input: 13

Length of string in str after input: 13

1. **Note that the program says the length of the C-style string in the array charr before input is 27, which is larger than the size of the array!** Two things are going on here. The first is that the contents of an uninitialized array are undefined. The second is that the **strlen() function works by starting at the first element of the array and counting bytes until it reaches a null character**. In this case, the first null character doesn’t appear until several bytes after the end of the array. Where the first null character appears in uninitialized data is essentially random, so you very well could get a different numeric result using this program.

Also note that the **length of the string in str before input is 0. That’s because an uninitialized string object is automatically set to zero size**.

1. This is the code for reading a line into an **array**:

cin.getline(charr, 20);

这里的getline()是cin的一个member function;

This is the code for reading a line **into a string object**:

**getline(cin, str);**

这里的getline()不是成员函数; so it takes cin as an argument that tells it where to find the input. **Also there isn’t an argument for the size of the string because the string object automatically resizes to fit the string**.

**So why is one getline() an istream class method and the other getline() not?**

The **istream class was part of C++ long before the string class was added**. So the istream design recognizes basic C++ types such as double and int, **but it is ignorant of the string type**. Therefore, there are istream class methods for processing double, int, and the other basic types, **but there are no istream class methods for processing string objects. 所以需要第二个getline函数来单独处理string的输入.**

Because there are no istream class methods for processing string objects, you might wonder why code like this works:

cin >> str; // read a word into the str string object

It turns out that code like the following does (in disguised notation) use a member function of the istream class:

cin >> x; // read a value into a basic C++ type

**But the string class equivalent uses a friend function** (also in disguised notation) of the string class. You’ll have to wait until Chapter 11 to see what a friend function is and how this technique works. In the meantime, you can use cin and cout with string objects and not worry about the inner workings.

* **Other Forms of String Literals**

cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << '\n';

This would display the following:

Jim "King" Tutt uses \n instead of endl.

* **Introducing Structures**

struct inflatable

{

char name[20]**;**

float volume**;**

double price**;**

};

struct inflatable goose; **// keyword struct required in C**

inflatable vincent; **// keyword struct not required in C++**

inflatable guest =

{

"Glorious Gloria"**,** // name value

1.88, // volume value

29.99 // price value

};

struct perks

{

int key\_number;

char car[12];

} mr\_smith, ms\_jones; // two perks variables

You even can initialize a variable you create in this fashion:

struct perks

{

int key\_number;

char car[12];

} mr\_glitz =

{

7,

"Packard"

}

inflatable guests[2] = // initializing an array of structs

{

{"Bambi", 0.5, 21.99}, // first structure in array

{"Godzilla", 2000, 565.99} // next structure in array

};

结构体定义时每个变量后面是分号，实例化时是逗号.

When a declaration occurs outside any function, it’s called an external declaration. The external declaration can be used by all the functions following it, whereas the internal declaration can be used only by the function in which the declaration is found.

C++ practices discourage the use of external variables but encourage the use of external structure declarations.

* **C++11 Structure Initialization**

As with arrays, C++11 extends the features of list-initialization. The = sign is optional:

inflatable duck {"Daphne", 0.12, 9.98}; // can omit the = in C++11

Next, **empty braces result in the individual members being set to 0**. For example, the following declaration results in mayor.volume and mayor.price being set to 0 and **all the bytes in *mayor.name* being set to 0**:

inflatable mayor {};

**Can a Structure Use a string Class Member? Yes.**

#include <string>

struct inflatable

{

**std::string name;**

float volume;

double price;

};

* **Bit Fields in Structures**

C++, like C, enables you to specify structure members that occupy a particular number of bits. The field type should be an integral or enumeration type (enumerations are discussed later in this chapter), and a colon followed by a number indicates the actual number of bits to be used.

struct torgle\_register

{

unsigned int SN : 4; // 4 bits for SN value

unsigned int : 4; // 4 bits unused

bool goodIn : 1; // valid input (1 bit)

bool goodTorgle : 1; // successful torgling

};

You can initialize the fields in the usual manner, and you use standard structure notation to access bit fields:

torgle\_register tr = { 14, true, false };

if (tr.goodIn) // if statement covered in Chapter 6

...

* **Union**

A union is a data format that can hold different data types but only one type at a time. That is, whereas a structure can hold, say, an int and a long and a double, a union can hold an int **or** a long **or** a double.

union one4all

{

int int\_val;

long long\_val;

double double\_val;

};

You can use a one4all variable to hold an int, a long, or a double, just as long as you do so at different times:

one4all pail;

pail.int\_val = 15; // store an int

cout << pail.int\_val;

pail.double\_val = 1.38; // store a double, int value is lost

cout << pail.double\_val;

Thus, pail can serve as an int variable on one occasion and as a double variable at another time. The member name identifies the capacity in which the variable is acting. **Because a union holds only one value at a time, it has to have space enough to hold its largest member. Hence, the size of the union is the size of its largest member.**

One use for a union is to save space when a data item can use two or more formats **but never simultaneously**. For example, suppose you manage a mixed inventory of widgets, some of which have an integer ID, and some of which have a string ID. In that case, you could use the following:

struct widget

{

char brand[20];

int type;

**union id** // format depends on widget type

{

long id\_num; // type 1 widgets

char id\_char[20]; // other widgets

} id\_val;

};

...

widget prize;

...

if (prize.type == 1) // if-else statement (Chapter 6)

cin >> prize.id\_val.id\_num; // use member name to indicate mode

else

cin >> prize.id\_val.id\_char;

* **Enumerations**

**enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet}**; This statement does two things:

1. It makes **spectrum the name of a new type**; spectrum is termed an enumeration, much as a struct variable is called a structure;
2. It establishes red, orange, yellow, and so on, **as symbolic constants for the integer values 0–7**. These constants are called enumerators;

You can use an enumeration name to declare a variable of the enumeration type:

**spectrum** **band**; // band a variable of type spectrum

band = blue; **// valid, blue is an enumerator**

band = 2000; **// invalid, 2000 not an enumerator**

**Only the assignment operator is defined for enumerations**. In particular, **arithmetic operations are NOT defined**:

band = orange; // valid

++band; **// not valid, ++ discussed in Chapter 5**

band = orange + red; **// not valid, but a little tricky**

By default, enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second enumerator, and so forth.

**Enumerators are of integer type and can be promoted to type int**, but int types are not converted automatically to the enumeration type:

int color = blue; // valid, spectrum type promoted to int

band = 3; // invalid, int not converted to spectrum

color = 3 + red; // valid, red converted to int

band = orange + red; // not valid, but a little tricky

It is true that the **+ operator is not defined for enumerators**. But **it is also true that enumerators are converted to integers when used in arithmetic expressions**, so the expression **orange + red gets converted to 1 + 0, which is a valid expression**. But it is of type int and hence cannot be assigned to the type spectrum variable band.

You can assign an int value to an enum, provided that the value is valid and that you use an explicit type cast:

band = spectrum(3); // typecast 3 to type spectrum, band is actually green

band = spectrum(40003); // undefined

* **Setting Enumerator Values**

You can set enumerator values explicitly by using the assignment operator:

enum bits{one = 1, two = 2, four = 4, eight = 8};

**The assigned values must be integers.** You also can define just some of the enumerators explicitly:

enum bigstep{first, second = 100, third};

In this case, first is 0 by default. Subsequent uninitialized enumerators are larger by one than their predecessors. **So, third would have the value 101.**

Finally, you can create more than one enumerator with the same value:

enum {zero, null = 0, one, numero\_uno = 1};

Here, **both zero and null are 0**, and **both one and numero\_uno are 1**. In earlier versions of C++, you could assign only int values (or values that promote to int) to enumerators, but that restriction has been removed so that you can use type long or even long long values.

* **Value Ranges for Enumerations**

Each enumeration has a range, and **you can assign any integer value in the range**, **even if it’s not an enumerator value**, by using a **type cast** to an enumeration variable. For example, suppose that bits and myflag are defined this way:

enum bits{one = 1, two = 2, four = 4, eight = 8};

bits myflag;

In this case, the following is valid:

myflag = bits(6); **// valid, because 6 is in bits range, and bits(6) is a type cast.**

* **Pointers and the Free Store**

Pointers are variables that store addresses of values rather than the values themselves.

cout << " donuts address = " << &donuts << endl; // **donuts address = 0x0065fd40 直接16进制**

Object-oriented programming differs from traditional procedural programming in that **OOP emphasizes making decisions during runtime instead of during compile time**. Runtime means while a program is running, and compile time means when the compiler is putting a program together.

比如说定义一个array, the traditional way is to declare an array. To declare an array in C++, you have to commit yourself to a particular array size. Thus, the array size is set when the program is compiled; **it is a compile-time decision**. While with OOP, you would like to make the array size a runtime decision.

With pointers**, the name of the pointer represents the location**. **Applying the \* operator, called the *indirect value* or the *dereferencing operator*, yields the value at the location**.

#include <iostream>

int main()

{

using namespace std;

int updates = 6;

int \* p\_updates;

**p\_updates = &updates;**

cout << "Values: updates = " << updates;

cout << ", \*p\_updates = " << \*p\_updates << endl;

cout << "Addresses: &updates = " << &updates;

cout << ", **p\_updates** = " << **p\_updates** << endl;

\*p\_updates = \*p\_updates + 1;

cout << "Now updates = " << updates << endl;

return 0;

}

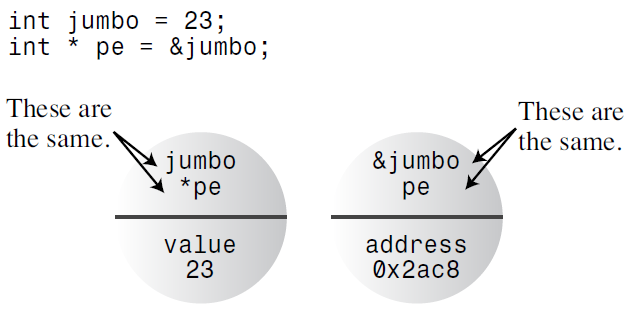
书上示例的结果：

Values: updates = 6, \*p\_updates = 6

Addresses: &updates = 0x0065fd48, p\_updates = 0x0065fd48

Now updates = 7

p\_updates is a pointer (an address), and **\*p\_updates is an int** and not a pointer.



* **Declaring and Initializing Pointers**

Be aware, however, that the following declaration creates **one pointer (p1) and one ordinary int (p2)**:

int\* p1, p2;

You need an \* for each pointer variable name.

You can use a declaration statement to initialize a pointer. In that case, **the pointer, not the pointed-to value, is initialized**. That is, **the following statements set pt and not \*pt to the value &higgens**:

int higgens = 5;

int \* pt = &higgens;

Danger awaits those who incautiously use pointers. One extremely important point is that **when you create a pointer in C++, the computer allocates memory to hold an address, but it does not allocate memory to hold the data to which the address points.**

long \* fellow; // create a pointer-to-long

**\*fellow = 223323**; **// Disaster! place a value in never-never land**

Sure, fellow is a pointer**. But where does it point? The code failed to assign an address to fellow**. So **where is the value 223323 placed? We can’t say**. Because fellow wasn’t initialized, it could have any value. **Whatever that value is, the program interprets it as the address at which to store 223323.** If fellow happens to have the value 1200, then the computer attempts to place the data at address 1200, even if that happens to be an address in the middle of your program code. Chances are that wherever fellow points, that is not where you want to put the number 223323.

意思是说，fellow才是pointer，但这个程序没有给fellow赋一个地址，那么系统就随机的给fellow一个值，然后这个值当作一个地址，并在这个地址处存上223323.

**Pointer Golden Rule**: Always initialize a pointer to a definite and appropriate **address** before you apply the dereferencing operator (\*) to it.

* **Pointers and Numbers**

**Pointers are not integer types, even though computers typically handle addresses as integers**. Conceptually, pointers are distinct types from integers.

int \* pt;

pt = 0xB8000000; **// type mismatch**

Here, **the left side is a pointer to int**, so you can assign it an address, but the **right side is just an integer**. **You might know that 0xB8000000 is the combined segment-offset address** of video memory on your aging computer system, but **nothing in the statement tells the program that this number is an address**.

int \* pt;

pt = **(int \*)** 0xB8000000; **// types now match**

* **Allocating Memory with new**

int \* pn = new int;

#include <iostream>

int main()

{

using namespace std;

int nights = 1001;

cout << "nights value = ";

cout << nights << ": &nights = " << &nights << endl;

int \* pt = new int;

\*pt = 1001;

cout << "\* pt = " << \*pt << ": pt = " << pt << ": &pt = " << &pt << endl;

double \* pd = new double;

\*pd = 10000001.0;

cout << "\* pd = " << \*pd << ": pd = " << pd << ": &pd = " << &pd << endl;

cout << "size of pt = " << sizeof(pt);

cout << ": size of \*pt = " << sizeof(\*pt) << endl;

cout << "size of pd = " << sizeof pd;

cout << ": size of \*pd = " << sizeof(\*pd) << endl;

return 0;

}

自己运行的结果：

momo@HMI:~/C++PrimerPlus/Chapter4$ ./List4-17

nights value = 1001: &nights = 0x7fff0bd95c1c

\* pt = 1001: pt = 0x80a010: &pt = 0x7fff0bd95c20

\* pd = 1e+07: pd = 0x80a030: &pd = 0x7fff0bd95c28

size of pt = 8: size of \*pt = 4

size of pd = 8: size of \*pd = 8

momo@HMI:~/C++PrimerPlus/Chapter4$ vim List4-17.cpp

所以

int \* pt = new int;

\*pt = 1001;

的意思是：**先在heap中分配一个存int的内存单元，这个单元的地址是0x80a010，这是一个很低的地址，所以符合memory对的layout; 然后int \* pt又分配了一个内存单元去存0x80a010这个地址，这个内存单元的地址是0x7fff0bd95c20, 是个很高的地址, 应该在stack中**.

* **Freeing Memory with delete**

int \* ps = new int; // allocate memory with new

. . . // use the memory

delete ps; // free memory with delete when done

This removes the memory to which ps points; **it doesn’t remove the pointer ps itself**.

You should not attempt to free a block of memory that you have previously freed. Also you cannot use delete to free memory created by declaring ordinary variables. **You should use delete only to free memory allocated with new**. However, it is safe to apply delete to a null pointer.

int \* ps = new int; // ok

delete ps; // ok

delete ps; // not ok now

int jugs = 5; // ok

int \* pi = &jugs; // ok

delete pi; // not allowed, memory not allocated by new

* **Using new to Create Dynamic Arrays**

If you create an array by declaring it, **the space is allocated when the program is compiled**. Whether or not the program finally uses the array, the array is there, using up memory. **Allocating the array during compile time is called static binding**, meaning that the array is built in to the program at compile-time.

1. **Creating a Dynamic Array with new**

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

The **new operator returns the address of the first element of the block**. In this example, that value is assigned to the pointer psome.

delete **[]** psome; // free a dynamic array

**You can’t use the sizeof operator, for example, to find the number of bytes in a dynamically allocated array.**

The presence of the brackets tells the program that it should free the whole array, not just the element pointed to by the pointer. Note that the brackets are between delete and the pointer. If you use new without brackets, you should use delete without brackets. If you use new with brackets, you should use delete with brackets.

int \* pt = new int;

short \* ps = new short [500];

delete [] pt; **// effect is undefined, don't do it**

delete ps;  **// effect is undefined, don't do it**

1. **Using a Dynamic Array**

在上面的例子中，psome points to the first element of the array, \*psome is the value of the first element. You can **use psome[0] instead of \*psome for the first element, psome[1] for the second element**, and so on. It turns out to be very simple to use a pointer to access a dynamic array, even if it may not immediately be obvious why the method works. The reason you can do this is that C and C++ handle arrays internally by using pointers anyway.

#include <iostream>

int main()

{

using namespace std;

double \* p3 = new double [3];

p3[0] = 0.2; // treat p3 like an array name

p3[1] = 0.5;

p3[2] = 0.8;

cout << "p3[1] is " << **p3[1]** << ".\n"; **// p3[1], not \*p3[1]**

p3 = p3 + 1; // increment the pointer, **p3 is the base address**

cout << "Now p3[0] is " << p3[0] << " and "; **// p3[0] is 0 offset based on p3**

cout << "p3[1] is " << p3[1] << ".\n";

p3 = p3 - 1; // point back to beginning

delete [] p3; // free the memory

return 0;

}

书上的示例结果：

p3[1] is 0.5.

Now p3[0] is 0.5 and p3[1] is 0.8.

**The fundamental difference between an array name and a pointer appears in the following line**:

p3 = p3 + 1; // okay for pointers, wrong for array names

**You can’t change the value of an array name. But a pointer is a variable, hence you can change its value.** Note the effect of adding 1 to p3.The expression p3[0] now refers to the former second element of the array.

* **Pointers, Arrays, and Pointer Arithmetic**

Adding 1 to a pointer variable increases its value by the number of bytes of the type to which it points.

#include <iostream>

int main()

{

using namespace std;

double wages[3] = {10000.0, 20000.0, 30000.0};

short stacks[3] = {3, 2, 1};

// Here are two ways to get the address of an array

**double \* pw = wages; // name of an array = address**

**short \* ps = &stacks[0]; // or use address operator**

cout << "pw = " << pw << ", \*pw = " << \*pw << endl;

pw = pw + 1;

cout << "add 1 to the pw pointer:\n";

cout << "pw = " << pw << ", \*pw = " << \*pw << "\n\n";

cout << "ps = " << ps << ", \*ps = " << \*ps << endl;

ps = ps + 1;

cout << "add 1 to the ps pointer:\n";

cout << "ps = " << ps << ", \*ps = " << \*ps << "\n\n";

cout << "access two elements with array notation\n";

cout << "stacks[0] = " << stacks[0]

<< ", stacks[1] = " << stacks[1] << endl;

cout << "access two elements with pointer notation\n";

cout << "\*stacks = " << \*stacks

<< ", \*(stacks + 1) = " << **\*(stacks + 1)** << endl;

cout << sizeof(wages) << " = size of wages array\n";

cout << sizeof(pw) << " = size of pw pointer\n";

return 0;

}

书上给的示例结果：

pw = 0x28ccf0, \*pw = 10000

add 1 to the pw pointer:

pw = 0x28ccf8, \*pw = 20000

ps = 0x28ccea, \*ps = 3

add 1 to the ps pointer:

ps = 0x28ccec, \*ps = 2

access two elements with array notation

stacks[0] = 3, stacks[1] = 2

access two elements with pointer notation

\*stacks = 3, \*(stacks + 1) = 2

24 = size of wages array

4 = size of pw pointer

**In general, wherever you use array notation, C++ makes the following conversion**:

**arrayname[i] becomes \*(arrayname + i)**

* **The Address of an Array**

The **name of the array** is interpreted **as the address of the first element** of an array, whereas applying the **address operator to the array name yields the address of the whole array**.

short tell[10]; // tell an array of 20 bytes

cout << tell << endl; // displays &tell[0]

cout << &tell << endl; // displays address of whole array

**&tell[0]**, and hence **tell**, is the address of a 2-byte block of memory, whereas **&tell** is the address of a 20- byte block of memory. So the expression **tell + 1** adds 2 to the address value, whereas **&tell + 1 adds 20 to the address value**.

书P170在此还有其他讲解；

* **Summarizing Pointer Points**

\*pc = 'S'; **// place 'S' into the memory location whose address is pc**

int \* pt = new int; // assigns an address to the pointer pt

\*pt = 5; // stores the value 5 at that address

C++ treats the name of an array as equivalent to the address of the first element of an array. One exception is when you use the name of an array with the sizeof operator. In that case, sizeof returns the size of the entire array, in bytes.

int size;

cin >> size;

int \* pz = new int [size]; // dynamic binding, size set at run time

...

delete [] pz; // free memory when finished

int \* pt = new int [10]; // pt points to block of 10 ints

**\*pt = 5;** **// set element number 0 to 5**

pt[0] = 6; // reset element number 0 to 6

pt[9] = 44; // set tenth element (element number 9) to 44

int coats[10];

\*(coats + 4) = 12; // set coats[4] to 12

char flower[10] = "rose";

cout << flower << "s are red\n";

**The cout object assumes that the address of a char is the address of a string**, so it prints the character at that address and then continues printing characters until it runs into the null character (\0). In short, if you give cout the address of a character, it prints everything from that character to the first null character that follows it.

**In general, wherever you use array notation, C++ makes the following conversion**:

arrayname[i] becomes \*(arrayname + i)

And it is, **for in C++ a quoted string, like an array name, serves as the address of its first element**. The preceding code doesn’t really send a whole string to cout; it just sends the string address. This means strings in an array, quoted string constants, and strings described by pointers are all handled equivalently.

#include <iostream>

#include <cstring>

int main()

{

using namespace std;

char animal[20] = "bear";

**const char \* bird = "wren";**

char \* ps;

cout << animal << " and ";

cout << bird << "\n";

cout << "Enter a kind of animal: ";

cin >> **animal;**

ps = animal;

cout << ps << "!\n";

cout << "Before using strcpy():\n";

cout << animal << " at " << **(int \*)** animal << endl;

cout << ps << " at " << **(int \*)** ps << endl;

**ps = new char[strlen(animal) + 1];**

**strcpy(ps, animal);**

cout << "After using strcpy():\n";

cout << animal << " at " << (int \*) animal << endl;

cout << ps << " at " << (int \*) ps << endl;

**delete [] ps;**

return 0;

}

Remember, **"wren" actually represents the address of the string**, so this statement assigns the address of "wren" to the bird pointer.

* [**解释bird前为什么要有const**] String literals are constants, which is why the code uses the **const keyword** in the declaration. Using const in this fashion means you can use bird to access the string **but not to change it**. **C++ doesn’t guarantee that string literals are stored uniquely**. That is, **if you use a string literal "wren" several times in a program, the compiler might store several copies of the string or just one copy**. **If it does the latter**, then setting bird to point to one "wren" makes it point to the only copy of that string. Reading a value into one string could affect what you thought was an independent string elsewhere. In any case, **because the bird pointer is declared as const, the compiler prevents any attempt to change the contents of the location pointed to by bird.**
* **[解释为什么要用array做输入而不是指针]** For input, the situation is a bit different. **It’s safe to use the array animal for input as long as the input** is short enough to fit into the array. **It would not be proper to use bird for input**. Worse yet is trying to read information into the location to which ps points. **Because ps is not initialized, you don’t know where the information will wind up**. It might even overwrite information that is already in memory. Fortunately, it’s easy to avoid these problems: You just use a sufficiently large char array to receive input and **don’t use** string constants to receive input or **uninitialized pointers to receive input**. (Or you can sidestep all these issues and use std::string objects instead of arrays.)

Notice what the following code accomplishes:

ps = animal; // set ps to point to string

...

cout << animal << " at " << **(int \*) animal** << endl;

cout << ps << " at " << **(int \*)** ps << endl;

It produces the following output:

fox at 0x0065fd30

fox at 0x0065fd30

**Normally**, if **you give cout a pointer, it prints an address**. P19的例子.

**But if the pointer is type char \*,** cout displays the **pointed-to string**. If you want to see the address of the string, you have to type cast the pointer to another pointer type, such as int \*, which this code does. So ps displays as the string "fox", but (int \*) ps displays as the address where the string is found.

**ps = new char[strlen(animal) + 1]; // add 1 to get the length, including the null character**

Often you encounter the need to place a string into an array. You use the = operator when you initialize an array; otherwise, you use strcpy() or strncpy(). You’ve seen the strcpy() function; it works like this:

char food[20] = "carrots"; // initialization

strcpy(food, "flan"); // otherwise

**strncpy**(food, "a picnic basket filled with many goodies", 19);

food[19] = '\0';

**Use strcpy() or strncpy(), not the assignment operator, to assign a string to an array.**

* **Using *new* to Create Dynamic Structures**

inflatable \* ps = new inflatable;

When you create a **dynamic structure**, **you can’t use the dot membership operator** with the structure name because the structure has no name. All you have is its address. **C++ provides an operator just for this situation: the arrow membership operator (->).** It does for pointers to structures what the dot operator does for structure names.

还有一个更恶心的办法：

A second, uglier approach to accessing structure members is to realize that if ps is a pointer to a structure, then **\*ps represents** the pointed-to value—**the structure itself**. Then, because \*ps is a structure, **(\*ps).price is the price member of the structure**. C++’s operator precedence rules require that you use parentheses in this construction.

* **An Example of Using new and delete**

Let’s look at an example that uses new and delete to manage storing string input from the keyboard.

#include <iostream>

#include <cstring> **// or string.h**

using namespace std;

**char \* getname(void);** **// function prototype**

int main()

{

char \* name;

**name = getname();**

cout << name << " at " << **(int \*) name** << "\n"; **// P25页中部解释了为什么加(int \*)**

delete [] name;

name = getname();

cout << name << " at " << **(int \*)** name << "\n";

delete [] name;

return 0;

}

char \* getname()

{

**char temp[80];**

cout << "Enter last name: ";

cin >> temp;

**char \* pn = new char[strlen(temp) + 1];**

strcpy(pn, temp);

**return pn;**

}

在getname()运行完后，它的函数stack里的一切都会被清空，包括temp数组；但我们已经在heap区申请了一段内存并将输入的数据copy进了这段heap，所以pn这个地址在getname()结束后仍然有效;

Note in this example that **getname()** allocates memory and main() frees it. It’s usually not a good idea to put new and delete in separate functions because that makes it easier to forget to use delete. But this example does separate new from delete just to show that it is possible.

* **Automatic Storage, Static Storage, and Dynamic Storage**

C++ has **three ways** of managing memory for data, depending on the method used to allocate memory: ***automatic storage***, ***static storage***, and ***dynamic storage***, sometimes called the free store or heap. Data objects allocated in these three ways differ from each other in how long they remain in existence.

* Ordinary variables **defined inside a function use automatic storage** and are called automatic variables. These terms mean that the variables come into existence automatically when the function containing them is invoked, and they expire when the function terminates.

Actually, automatic values are local to the block that contains them. **A block is a section of code enclosed between braces**. If you define a variable inside one of those blocks, it exists only while the program is executing statements inside the block.

Automatic variables typically are stored on a **stack**.

* **Static storage** is storage that **exists throughout the execution of an entire program**. There are two ways to make a variable static. One is to define it externally, outside a function. The other is to use the keyword static when declaring a variable:

static double fee = 56.50;

* **Dynamic Storage**

This pool, a region of memory managed by new and delete, is separate from the memory used for static and automatic variables. As example above shows, **new and delete enable you to allocate memory in one function and free it in another**. Thus, the lifetime of the data is not tied arbitrarily to the life of the program or the life of a function.

* **Combinations of Types**

struct antarctica\_years\_end

{

int year;

};

antarctica\_years\_end s01, s02, s03;

antarctica\_years\_end \* pa = &s02;

antarctica\_years\_end trio[3];

// Access method

pa->year = 1999;

(\*pa).year = 1999;

trio[0].year = 1999; // trio is an array, but **trio[0] is a structure**

(trio+1)->year = 1999; // **an array name is a pointer**, we also can use the indirect membership

// operator

**const** antarctica\_years\_end \* arp[3] = {&s01, &s02, &s03};

arp[1]->year = 1999;

// We can create a pointer to such an array:

**const** antarctica\_years\_end \*\* ppa = arp;

Here, **arp is the name of an array**; hence, **it is the address of its first element**. But its first element is a pointer, so **ppa has to be a pointer to a pointer to const antarctica\_years\_end**, hence the \*\*. There are several ways you could mess up this declaration. For example, **you could omit the const**, forget an \* or two, transpose letters, or otherwise mangle the structure type.

How can you use ppa to access data? Because **ppa is a pointer to a pointer to a structure**, **\*ppa is a pointer to a structure**, so you can use it with the indirect membership operator:

std::cout << **(\*ppa)->year** << std::endl;

std::cout << (\*(ppa+1))->year << std::endl;

Because **ppa points to the first member of arp**, **\*ppa is the (address of) first member, which is &s01**. So (\*ppa)->year is the year member of s01. In the second statement, **ppa+1 points to the next element, arp[1],** which is &s02.The parentheses are needed to get the correct associations. For example, \*ppa->year would attempt to apply the \* operator to ppa->year, which fails because the year member is not a pointer.

#include <iostream>

struct antarctica\_years\_end

{

int year;

};

int main()

{

antarctica\_years\_end s01, s02, s03;

**s01.year = 1998;**

antarctica\_years\_end \* pa = &s02;

**pa->year = 1999;**

antarctica\_years\_end trio[3];

**trio[0].year = 2003;**

std::cout << **trio->year** << std::endl;

**const** antarctica\_years\_end **\* arp[3] = {&s01, &s02, &s03};**

std::cout << **arp[1]->year** << std::endl;

**const** antarctica\_years\_end **\*\* ppa = arp**;

auto ppb = arp; // C++11 automatic type deduction

// or else use const antarctica\_years\_end \*\* ppb = arp;

std::cout << **(\*ppa)->year** << std::endl;

std::cout << (\*(ppb+1))->year << std::endl;

return 0;

}

Here’s the output:

2003

1999

1998

1999

**再对const antarctica\_years\_end \*\* ppa做个解释：**

首先, 我想定义一个指针, 这个指针指向的数据是arp, 而arp是个（antarctica\_years\_end \*）型的, 所以我要定义的数据是（antarctica\_years\_end \*）\*, 也就是antarctica\_years\_end \*\*.

* **Array Alternatives**
* **The vector Template Class**

**The vector template** class is similar to the string class in that **it is a dynamic array**. You can set the size of a vector object **during runtime**, and you can append new data to the end or insert new data in the middle. Basically, it’s an alternative to using new to create a dynamic array. **Actually, the vector class does use new and delete to manage memory, but it does so automatically**.

**First**, to use a vector object, you need to include the vector header file. **Second**, the vector identifier is part of the std namespace, so you can use a using directive, a using declaration, or std::vector. **Third**, templates use a different syntax to indicate the type of data stored. **Fourth**, the vector class uses a different syntax to indicate the number of elements.

#include <vector>

using namespace std;

**vector<int> vi;** // create a **zero-size** array of int

int n;

cin >> n;

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

* **The array Template Class**

**The vector class** has more capabilities than the built-in array type, **but this comes at a cost of slightly less efficiency.** If all you need is a fixed-size array, it could be advantageous to use the built-in type. However, that has its own costs of lessened convenience and safety. C++11 responded to this situation by adding the array template class, which is part of the std namespace.

Like the built-in type, **an array object has a fixed size and uses the stack** (or else static memory allocation) instead of the free store, so it shares the efficiency of built-in arrays.

using namespace std;

array<int, 5> ai; // create array object of 5 ints

array<double, 4> ad = {1.2, 2.1, 3.43. 4.3};

#include <iostream>

#include <vector> // STL C++98

#include <array> // C++11

int main()

{

using namespace std;

**// C, original C++**

double a1[4] = {1.2, 2.4, 3.6, 4.8};

**// C++98 STL**

vector<double> **a2**(4); // create vector with 4 elements

// no simple way to initialize in C98

a2[0] = 1.0/3.0;

a2[1] = 1.0/5.0;

a2[2] = 1.0/7.0;

a2[3] = 1.0/9.0;

**// C++11 -- create and initialize array object**

array<double, 4> a3 = {3.14, 2.72, 1.62, 1.41};

array<double, 4> a4;

**a4 = a3;** // valid for array objects of same size

// use array notation

cout << "a1[2]: " << a1[2] << " at " << &a1[2] << endl;

cout << "a2[2]: " << a2[2] << " at " << &a2[2] << endl;

cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;

cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

// misdeed

a1[-2] = 20.2;

cout << "a1[-2]: " << a1[-2] <<" at " << &a1[-2] << endl;

cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;

cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

return 0;

}

书上示例的结果：

a1[2]: 3.6 at 0x28cce8

a2[2]: 0.142857 at 0xca0328

a3[2]: 1.62 at 0x28ccc8

a4[2]: 1.62 at 0x28cca8

a1[-2]: 20.2 at 0x28ccc8

a3[2]: 20.2 at 0x28ccc8

a4[2]: 1.62 at 0x28cca8

说明：

1. **a4 = a3;**

You can **assign an array object to another array object**. For built-in arrays, you have to copy the data element-by-element;

1. **a1[-2] 正好砸在a3头上了;**

Do the vector and array objects protect against this overflow behavior?

One is using the *at()* member function with objects of the vector or array type.

a2.at(1) = 2.3; // assign 2.3 to a2[1]

If you use at(), **an invalid index is caught during runtime** and the program, by default, **aborts**. This added checking **does come at the cost of increased run time**, which is why C++ gives you the option of using either notation. More than that, these classes offer ways of using objects that reduce the chances of accidental range errors. For example, the classes have *begin()* and *end()* member functions that let you delimit the range without accidentally exceeding the bounds.