**Chapter 7 Functions**

**The return value of C++ functions cannot be an array**. Everything else is possible—integers, floating-point numbers, pointers, and even structures and objects! (Interestingly, even though a C++ function can’t return an array directly**, it can return an array that’s part of a structure or object**.)

ANSI C, to preserve compatibility with classic C, made prototyping optional, whereas **C++ makes prototyping(函数声明) mandatory**. For example, consider the following function declaration:

void say\_hi();

**In C++, leaving the parentheses empty is the same as using the keyword *void*** within the parentheses. It means the function has no arguments. In ANSI C, leaving the parentheses empty means that you are declining to state what the arguments are. That is, it means you’re forgoing prototyping the argument list.

double cube(double x);

void cheers(int);

**cheers(cube(2));**

In C, this could create weird errors. For example, if a function expects a type int value (assume that’s 16 bits) and you pass a double (assume that’s 64 bits), the function looks at just the first 16 bits of the 64 and tries to interpret them as an int value. However, **C++ automatically converts the value you pass to the type specified in the prototype, provided that both are arithmetic types**.

Then cube() returns a type double value (8.0) to be used as an argument to cheers().Again, the compiler checks the prototypes and notes that cheers() requires an int. It converts the return value to the integer 8.

**Prototyping** takes place during **compile time** and is termed static type checking.

* **Function Arguments and Passing by Value**

double cube(double x); // Prototyping

double side = 5;

double volume = cube(side);

When this function is called, it creates a new type double variable called x and initializes it with the value 5.This insulates data in main() from actions that take place in cube() because **cube() works with a copy of side rather than with the original data**.

Variables, including parameters, declared within a function are private to the function. When a function is called, the computer allocates the memory needed for these variables. When the function terminates, the computer frees the memory that was used for those variables.

* **Functions and Arrays**

参数中有数组的函数的声明如下：

int sum\_arr(int arr[], int n); // prototype

C++ interprets an array name as the address of its first element:

cookies == &cookies[0] // array name is address of first element

int cookies[ArSize] = {1,2,4,8,16,32,64,128};

**// Applying sizeof to an array name yields the size of the whole array, in bytes;**

**// Spplying the address operator & to an array name returns the address of the whole array;**

**// for example, &cookies would be the address of a 32-byte block of memory if int is 4 bytes**

cookies is the name of an array, hence by C++ rules cookies is the address of the array’s first element. The function passes an address. **Because the array has type int elements, cookies must be type pointer-to-int, or int \***.

所以一个函数如果用cookies这个数组作为参数，它的声明**也可以**是：

int sum\_arr(**int \*** arr, int n) // arr = array name, n = size

C++ the notations ***int \*arr*** and ***int arr[]*** **have the identical meaning when (and only when) used in a function header or function prototype**. Remember that the notations ***int \*arr*** and ***int arr[]*** are not synonymous in any other context. For example, you can’t use the notation int tip[] to declare a pointer in the body of a function.

arr[i] == \*(arr + i) // 二者等价

&arr[i] == arr + i // 二者等价

**int sum\_arr(int \* arr, int n)**; // arr = array name, n = size

This function uses the original array. If you pass an ordinary variable, the function works with a copy. **But if you pass an array, the function works with the original**.

**Is the correspondence between array names and pointers a good thing?**

Indeed, it is. The design decision to use array addresses as arguments **saves the time and memory needed to copy an entire array**. The overhead for using copies can be prohibitive if you’re working with large arrays.

On the other hand, working with the original data raises the possibility of inadvertent data corruption. That’s a real problem in classic C, but ANSI C and C++’s ***const*** modifier provides a remedy.

#include <iostream>

const int ArSize = 8;

int sum\_arr(int arr[], int n);

int main()

{

int cookies[ArSize] = {1,2,4,8,16,32,64,128};

std::cout << **&cookies** << "= &cookies\n";

std::cout << **&cookies + 1** << "= &cookies + 1\n";

std::cout << **cookies** << " = array address, ";

std::cout << **sizeof cookies** << " = sizeof cookies\n";

int sum = sum\_arr(cookies, ArSize);

std::cout << "Total cookies eaten: " << sum << std::endl;

sum = sum\_arr(cookies, 3);

std::cout << "First three eaters ate " << sum << " cookies.\n";

sum = sum\_arr(cookies + 4, 4);

std::cout << "Last four eaters ate " << sum << " cookies.\n";

return 0;

}

int sum\_arr(int arr[], int n)

{

int total = 0;

std::cout << arr << " = arr, ";

std::cout << **sizeof arr** << " = sizeof arr\n";

for (int i = 0; i < n; i++)

total = total + arr[i];

return total;

}

在自己电脑上测试结果为：

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-6

**0x7fffa8db8090= &cookies**

**0x7fffa8db80b0= &cookies + 1 // &cookies + 1实际上加了32个bytes；**

**0x7fffa8db8090 = array address, 32 = sizeof cookies // &cookies == cookies**

0x7fffa8db8090 = arr, **8 = sizeof arr**

Total cookies eaten: 255

0x7fffa8db8090 = arr, 8 = sizeof arr

First three eaters ate 7 cookies.

**0x7fffa8db80a0** = arr, 8 = sizeof arr **// cookies + 4加了16个bytes;**

Last four eaters ate 240 cookies.

momo@HMI:~/C++PrimerPlus/Chapter7$

**sizeof cookies is 32**, whereas **sizeof arr is only 4**. That’s because ***sizeof cookies*** is the size of the whole array, whereas ***sizeof arr*** is the size of the pointer variable.

**By the way, this is why *you have to explicitly pass the size of the array rather than use sizeof arr* in sum\_arr(); the pointer by itself doesn’t reveal the size of the array.**

**[注] 这里你也没法用strlen()：**

int sum\_arr(int arr[], int n)

{

int total = 0;

std::cout << strlen(arr) << "= strlen(arr)\n";

编译会报错：

List7-6.cpp:26:26: error: cannot convert ‘int\*’ to ‘const char\*’ for argument ‘1’ to ‘size\_t strlen(const char\*)’

std::cout << strlen(arr) << "= strlen(arr)\n";

strlen()是针对char []或者char \*，或者string的；

**还有，int cookies[ArSize] = {1,2,4,8,16,32,64,128} 本身后面也没有 ‘\0’，那同样是针对字符串的，连字符数组都没有；**

[Note] To indicate the kind of array and the number of elements to an array-processing function, you pass the information as two separate arguments:

void fillArray(int arr[], int size); **// prototype**

Don’t try to pass the array size by using brackets notation:

void fillArray(int arr[size]); **// NO -- bad prototype**

下面是自己测试的几个int []的例子：

#include <iostream>

int main()

{

int Size = 5;

**int Array[Size] = {1,2,3,4,5};**

int sum = 0;

int index;

for (index = 0; **index < 6**; index++)

{

std::cout << Array[index] << std::endl;

sum = sum + Array[index];

}

std::cout << "sum is " << sum << std::endl;

return 0;

}

编译会报错：

momo@HMI:~/C++PrimerPlus/Chapter7$ g++ TestIntArray.cpp -o TestIntArray

TestIntArray.cpp: In function ‘int main()’:

TestIntArray.cpp:6:30: error: **variable-sized object ‘Array’** may not be initialized

int Array[Size] = {1,2,3,4,5};

^

momo@HMI:~/C++PrimerPlus/Chapter7$

**这里的原因是Array的长度在定义时用的是变量Size**; 可以如下改正：

int Array[5] = {1,2,3,4,5};

或者

int Size = 5放在main()之外；

或者

Const int Size = 5，仍然在main()之中定义；

改正以后在自己电脑上运行运行：

momo@HMI:~/C++PrimerPlus/Chapter7$ ./TestIntArray

1

2

3

4

5

**32767**

**sum is 32782**

momo@HMI:~/C++PrimerPlus/Chapter7$

**无论哪一种改正办法都是这个结果，说明即使你在定义时明确了数组的长度，仍然可以显式的超范围读取数据而没有任何问题，编译连warning都没有；**

**More Array Function Examples**

下面的是一个从键盘输入一个double数组的例子：

* **Filling the Array**

int fill\_array(double ar[], int limit)

{

using namespace std;

**double temp**;

int i;

for (i = 0; i < limit; i++)

{

cout << "Enter value #" << (i + 1) << ": ";

**cin >> temp**;

if (**!cin**) **// bad input**

{

**cin.clear();**

while (cin.get() != '\n')

continue;

cout << "Bad input; input process terminated.\n";

break;

}

else if (temp < 0) // signal to terminate

break;

ar[i] = temp;

}

return i;

}

* **Showing the Array and Protecting It with *const***

To keep a function from accidentally altering the contents of an array argument, you can use the keyword ***const.***

void show\_array(**const** double ar[], int n);

The declaration states that the **pointer ar points to *constant data***. This means that you can’t use ar to change the data. **Note that this doesn’t mean that the original array needs be constant**; it just means that you can’t use ar in the show\_array() function to change the data. Thus, show\_array() treats the array as read-only data.

Declaration ***const double ar[]*** also means ***const double \*ar***. Thus, the declaration really says that ar points to a constant value.

void show\_array(const double ar[], int n)

{

using namespace std;

for (int i = 0; i < n; i++)

{

cout << "Property #" << (i + 1) << ": $";

cout << ar[i] << endl;

}

}

* **Modifying the Array**

void revalue(double r, double ar[], int n)

{

for (int i = 0; i < n; i++)

ar[i] \*= r;

}

Because this function is supposed to alter the array values, you don’t use const when you declare ar.

* **Putting the Pieces Together**

**<List7-7.cpp>**

#include <iostream>

const int Max = 5;

**// function prototypes**

int fill\_array(double ar[], int limit);

void show\_array(**const** double ar[], int n); // don't change data

void revalue(double r, double ar[], int n);

int main()

{

using namespace std;

double properties[Max];

**int size = fill\_array(properties, Max);**

**show\_array(properties, size);**

if (size > 0)

{

cout << "**Enter revaluation factor:** ";

double factor;

while (!(cin >> factor)) // bad input

{

cin.clear();

while(cin.get() != '\n')

continue;

cout << "Bad input; Please enter a number: ";

}

**revalue(factor, properties, size);**

**show\_array(properties, size);**

}

cout << "Done.\n";

**cin.get();**

**cin.get();**

return 0;

}

**int fill\_array(double ar[], int limit)**

{

using namespace std;

double temp;

int i;

**for (i = 0; i < limit; i++)**

{

cout << "Enter value #" << (i + 1) << ": ";

cin >> temp;

if (!cin) **// 输入了字符等非数字的，程序直接terminate**

{

cout << “In Function fill\_array : cin failed” << endl;

cin.clear();

while (cin.get() != '\n')

{

cout << “In while loop of function fill\_array” << endl;

**continue;**

}

cout << "Bad input; input process terminated.\n";

**break; // Jump out of the for loop, not if**

}

else if (temp < 0) **// 输入是个数，但却是负数, 程序直接terminate**

**break; // Jump out of the for loop, not if**

ar[i] = temp; **// Correct input, assign to array**

**}** // End of for loop

return i;

}

**void show\_array(*const* double ar[], int n)**

{

using namespace std;

for (int i = 0; i < n; i++)

{

cout << "Property #" << (i + 1) << ": $";

cout << ar[i] << endl;

}

}

**void revalue(double r, double ar[], int n)**

{

for (int i = 0; i < n; i++)

ar[i] \*= r;

}

注意, **break是跳出当前Loop**, 在函数fill\_array**中指的是跳出for loop, 而不是只跳出if判断, 因为if判断不是个Loop**.

仔细看下面的输入函数：

**int fill\_array(double ar[], int limit)**

{

using namespace std;

double temp;

int i;

**for (i = 0; i < limit; i++)**

{

cout << "Enter value #" << (i + 1) << ": ";

cin >> temp;

if (!cin) **// 输入了字符等非数字的，程序直接terminate**

{

cout << “In Function fill\_array : cin failed” << endl;

cin.clear();

while (cin.get() != '\n')

{

cout << “In while loop of function fill\_array” << endl;

**continue;**

}

cout << "Bad input; input process terminated.\n";

**break; // Jump out of the for loop, not if**

}

else if (temp < 0) **// 输入是个数，但却是负数, 程序直接terminate**

**break; // Jump out of the for loop, not if**

ar[i] = temp; **// Correct input, assign to array**

**}** // End of for loop

return i;

}

首先，cin >> temp; cin自己会把input queue里的空格和tab给去掉, 检查剩下的数据是否是double型, 如果是, cin successful, 否则fail, 以newline符(回车键)作为一次输入结束；

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: (space)1.0(space)(回车)

Enter value #2: (Tab) 1.0(回车)

Enter value #3: 1.0(space)(回车)

Enter value #4: 1.0(space)(space)(回车)

Enter value #5: 1.0(回车)

Property #1: $1

Property #2: $1

Property #3: $1

Property #4: $1

Property #5: $1

所以上面这几个输入cin全部正确判断出来了；

**当输入的是非数字时：**

1. 由于cin >> temp的temp是double型的，不符合cin的要求，cin会fail；
2. cin会fail后会锁死input，所以先用cin.clear()恢复；
3. 后面的那个while循环的目的是把错误的输入从input queue里全提出去, 最后的回车符还保留在input queue, 没关系, cin本身会跳过回车符newline；
4. 然后break到for循环以外，进入main函数体；

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: **a #$%^**

In Function fill\_array : cin failed

In while loop of function fill\_array **// throw ‘a’**

In while loop of function fill\_array **// throw ‘ ‘**

In while loop of function fill\_array **// throw ‘#’**

In while loop of function fill\_array **// throw ‘$’**

In while loop of function fill\_array **// throw ‘%’**

In while loop of function fill\_array **// throw ‘^’**

Bad input; input process terminated.

Done.

**(回车)**

**(回车)**

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: **asd 12.3**

In Function fill\_array : cin failed

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

Bad input; input process terminated.

Done.

**(回车)**

**(回车)**

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: **asd12.3**

In Function fill\_array : cin failed

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

Bad input; input process terminated.

Done.

**(回车)**

**(回车)**

momo@HMI:~/C++PrimerPlus/Chapter7$

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: **12.3 asd**

**Enter value #2:** In Function fill\_array : cin failed

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

Bad input; input process terminated.

Property #1: $12.3

Enter revaluation factor: **0.9dfg**

Property #1: $11.07

Done**.(这里直接结束，不需要你再按回车键)**

momo@HMI:~/C++PrimerPlus/Chapter7$

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: **12.3asd**

Enter value #2: In Function fill\_array : cin failed

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

Bad input; input process terminated.

Property #1: $12.3

Enter revaluation factor: **0.9asd**

Property #1: $11.07

Done.

momo@HMI:~/C++PrimerPlus/Chapter7$

momo@HMI:~/C++PrimerPlus/Chapter7$ ./List7-7

Enter value #1: 12.3 asd

Enter value #2: In Function fill\_array : cin failed

In while loop of function fill\_array

In while loop of function fill\_array

In while loop of function fill\_array

Bad input; input process terminated.

Property #1: $12.3

Enter revaluation factor: 0.9

Property #1: $11.07

Done.

**(回车) // 这里只要一次，因为有一次输入是完全正确的**

momo@HMI:~/C++PrimerPlus/Chapter7$

**所以，main中最后为什么要有两个cin.get()？**

这个是基于最坏的情况考虑的, 也就是在原始数据和乘积因子都输入了不合法格式时, 相应函数需要调用while把非法数据扔出input queue, 但最后的newline符却是留在input queue里的; 所以最后调用两次get()把他们扔掉；

所有输入都完全正确时，最后只需要按一次回车就可以退出程序；

* **The Usual Array Function Idiom**

If the function is intended to modify the array, the prototype might look like this:

void f\_modify(double ar[], int n);

If the function preserves values, the prototype might look like this:

void \_f\_no\_change(const double ar[], int n);

Of course, you can omit the variable names in the prototypes, and the return type might be something other than void.

As you have seen, the function loses some knowledge about the original array; for example, **it can’t use *sizeof* to get the size and relies on you to pass the correct number of elements**.

* **Pointers and const**

You can use the ***const*** keyword **two different ways with pointers**. The first way is to make a **pointer point to a constant object**, and that prevents you from using the pointer to change the pointed-to value. The second way is to make the pointer itself constant, and that prevents you from changing where the pointer points to.

int age = 39;

**const** int \* pt = &age;

This declaration states that pt points to a const int (39, in this case).Therefore, you can’t use pt to change that value. In other words, **the value \*pt is const** and cannot be modified:

\*pt += 1; // INVALID because pt points to a const int

cin >> \*pt; // INVALID for the same reason

\*pt = 20; // INVALID because pt points to a const int

age = 20; // **VALID** because age is not declared to be const

This declaration for pt doesn’t necessarily mean that the value it points to is really a constant; it just means the value is a constant insofar as pt is concerned. For example, pt points to age, and **age is not const**. ***You can change the value of age directly by using the age variable, but you can’t change the value indirectly via the pt pointer***.

上面这个例子是把一个普通变量的地址赋给一个const指针, that leaves two other possibilities: assigning the **address of a const variable to a pointer-to-const** and assigning the **address of a const to a regular pointer**. Are they both possible? **The first is, and the second isn’t**.

const float g\_earth = 9.80;

**const float \* pe = &g\_earth**; **// VALID**

const float g\_moon = 1.63;

**float \* pm = &g\_moon**; **// INVALID, otherwise you can modify the g\_moom**

**// through pointer pm**

C++ prohibits you from assigning the address of a const to a non-const pointer.

int age = 39; // age++ is a valid operation

int \* pd = &age; // \*pd = 41 is a valid operation

const int \* pt = pd; // \*pt = 42 is an invalid operation

You **can assign the *address*** of either const data or non-const data **to a pointer-to-const**, **provided that the data type is not itself a pointer**, but you **can assign the address of non-const data** only to a **non-const pointer**.

就是说, const型指针可以接受const data或者一般data的地址, 而非const型指针只能接受非const data的地址;

Suppose you have an array of const data:

const int months[12] = {31,28,31,30,31,30, 31, 31,30,31,30,31};

The prohibition against assigning the address of a constant array means that you cannot pass the array name as an argument to a function by using a non-constant formal argument:

int sum(int arr[], int n); // should have been const int arr[]

...

int j = sum(months, 12); // not allowed

***This function call attempts to assign a const pointer (months) to a non-const pointer (arr), and the compiler disallows the function call***.

**[再来看const的另一种用法]**

int age = 39;

const int \* pt = &age;

The const in the second declaration only prevents you from changing the value to which pt points, which is 39. It doesn’t prevent you from changing the value of pt itself. That is, you can assign a new address to pt:

int sage = 80;

**pt = &sage; // okay to point to another location**

But you still can’t use pt to change the value to which it points (now 80).

**The second way to use const** makes it **impossible** to change the value of the pointer itself:

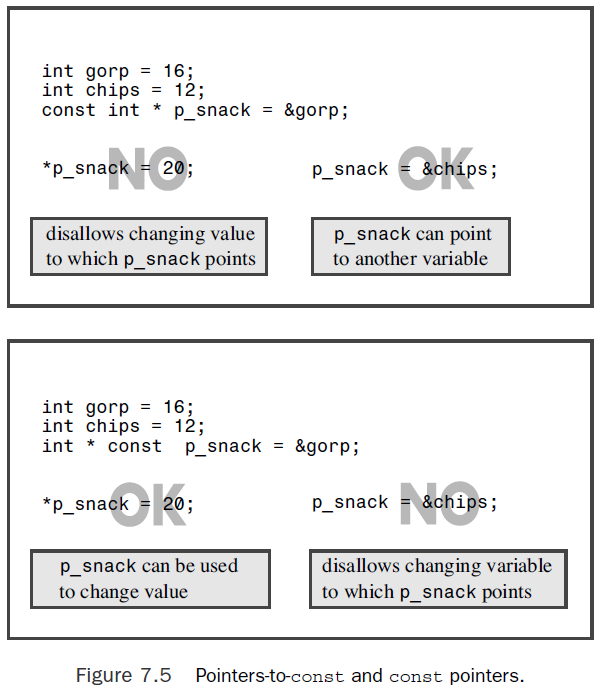
**int \* const** finger = &sloth; // a const pointer to int

**This form of declaration** **constrains *finger* to point only to sloth**. **However, it allows you to use finger to alter the value of sloth.**

const int \* ps = &sloth; // a pointer to const int

This form of declaration does **NOT allow you to use ps to alter the value of sloth**, but it permits you to have ps point to another location.

In short, **finger** and **\*ps** are both **const**, and **\*finger and ps are NOT const**.



If you like, you can declare a const pointer to a const object:

double trouble = 2.0E30;

const double \* const stick = &trouble;

Here stick can point only to trouble, and stick cannot be used to change the value of trouble. In short, both stick and \*stick are const.

* **Functions and Two-Dimensional Arrays**

Suppose, for example, that you start with this code:

**int data[3][4]** = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}};

int total = sum(data, 3);

What should the prototype for sum() look like? And why does the function pass the number of rows (3) as an argument and not also the number of columns (4)?

Well, ***data*** **is the name of an array with three elements**. The first element is, itself, an array of four int values. Thus, **the type of *data*** is **pointer-to-array-of-four-int**, so an appropriate prototype would be this:

int sum(int **(**\*ar2**)**[4], int size);

**int (\*ar2) [4]** declares a **single** pointer of pointer-to-array-of-four-int;

int \*ar2 [4] declare an array of four pointers of pointers-to-int;

Here’s an alternative format that means exactly the same thing as this first prototype, but, perhaps, is easier to read:

int sum(**int ar2[][4],** int size);

Either prototype states that ar2 is a pointer, not an array. **And a function parameter cannot be an array**.

Given that the parameter **ar2** is a **pointer to an array**, the simplest way is to use ar2 as if it were the name of a two-dimensional array:

int sum(int ar2[][4], int size)

{

int total = 0;

for (int r = 0; r < size; r++)

for (int c = 0; c < 4; c++)

total += ar2[r][c];

return total;

}

**Here’s why you can use array notation**. Because **ar2** points to the first element (element 0) **of an array whose elements are array-of-four-int**, the expression ar2 + r points to element number r. Therefore ar2[r] is element number r.

ar2[r][c] == \*(\*(ar2 + r) + c) // same thing

To understand this, you can work out the meaning of the sub-expressions from the inside out:

ar2 // pointer to first row of an array of 4 int

**ar2 + r** // **pointer to row r (an array of 4 int)**

**\*(ar2 + r)** // **row r** (an array of 4 int, hence the name of an array,

// **thus a pointer to the first element int in the row, i.e., ar2[r]**

\*(ar2 +r) + c // pointer int number c in row r, i.e., ar2[r] + c

\*(\*(ar2 + r) + c)// value of int number c in row r, i.e. ar2[r][c]

Incidentally, the code for **sum() doesn’t use const in declaring the parameter *ar2*** because **that technique is for pointers to fundamental types, and ar2 is a pointer to a pointer**.

const int \*\*pp2;

int \*p1;

const int n = 13;

pp2 = &p1; // **not allowed**, **but suppose it were**

**\*pp2** = &n; // valid, both const, but sets p1 to point at n

**\*p1** = 10; // valid, but changes const n

Here the code assigns a non-const address (&pl) to a const pointer (pp2), and that allows pl to be used to alter const data. **So the rule that you can assign a non-const address or pointer to a const pointer works only if there is just one level of indirection, for example, if the pointer points to a fundamental data type**.

* **Functions and C-Style Strings**

1. **Functions with C-Style String Arguments**

当拿一个string作为函数参数时，a string function prototype should use type ***char \**** as the type for the formal parameter representing a string.

**One important difference between a C-style string and a regular array is that the string has a built-in terminating character**. (Recall that **a char array containing characters but no null character** is just an array and not a string.) That means you don’t have to pass the size of the string as an argument.

unsigned int c\_in\_str(const char \* str, char ch);

unsigned int c\_in\_str(const char \* str, char ch)

{

unsigned int count = 0;

while (\*str) **// quit when \*str is '\0'**

{

if (\*str == ch)

count++;

str++;

}

return count;

}

1. **Functions That Return C-Style Strings**

Now suppose you want to write a function that returns a string.Well, a function can’t do that. But it can return the address of a string.

**char \*** buildstr(char c, int n);

**char \*** buildstr(char c, int n)

{

char \* pstr = new char[n + 1];

pstr[n] = '\0'; // terminate string

while (n-- > 0)

pstr[n] = c;

return pstr;

}

* **Functions and Structures**

You can pass structures by value, just as you do with ordinary variables. In that case, the function works with a copy of the original structure. **The name of a structure is simply the name of the structure**, and if you want its address, you have to use the & address operator.

函数对于struct的传递可以有以下几种办法：

1. Pass them as arguments and use them, if necessary, as return values.

However, there is one disadvantage to passing structures by value. If the structure is large, the space and effort involved in making a copy of a structure can increase memory requirements and slow down the system.

1. Passing the address of a structure and then using a pointer to access the structure contents.
2. C++ provides a third alternative, called passing by reference, that is discussed in Chapter 8.

**Passing and Returning Structures**

struct travel\_time

{

int hours;

int mins;

};

travel\_time sum(travel\_time t1, travel\_time t2);

**Passing Structure Addresses**

void show\_polar (const polar \* pda)

{

using namespace std;

const double Rad\_to\_deg = 57.29577951;

cout << "distance = " << pda->distance;

cout << ", angle = " << pda->angle \* Rad\_to\_deg;

cout << " degrees\n";

}

* **Functions and *string* Class Objects**

Although C-style strings and string class objects serve much the same purpose, a string class object is more closely related to a structure than to an array. If you need several strings, you can declare a one-dimensional array of string objects instead of a two-dimensional array of char.

#include <iostream>

#include <string>

using namespace std;

const int SIZE = 5;

void display(const string sa[], int n);

int main()

{

**string list[SIZE]**; // an array holding 5 string object

cout << "Enter your " << SIZE << " favorite astronomical sights:\n";

for (int i = 0; i < SIZE; i++)

{

cout << i + 1 << ": ";

**getline(cin,list[i]);**

}

cout << "Your list:\n";

display(list, SIZE);

return 0;

}

void display(const string sa[], int n)

{

for (int i = 0; i < n; i++)

**cout << i + 1 << ": " << sa[i] << endl;**

}

The main point to note in this example is that, aside from the getline() function, this program treats string just as it would treat any of the built-in types, such as int. If you want an array of string, you just use the usual array-declaration format:

string list[SIZE]; // an array holding 5 string object

Each element of the list array, then, is a string object and can be used as such:

getline(cin,list[i]);

Similarly, the formal argument sa is a pointer to a string object, so sa[i] is a string object and can be used accordingly:

cout << i + 1 << ": " << sa[i] << endl;

* **Functions and *array* Objects**

Suppose we have an array object intended to hold expense figures for each of the four seasons of the year:

std::array<double, 4> expenses;

**(Recall that using the array class requires the array header file and that the name array is part of the std namespace.)**

const std::array<std::string, Seasons> Snames =

{"Spring", "Summer", "Fall", "Winter"};

This adds a const array object containing four string objects representing the four seasons.

#include <iostream>

#include <array>

#include <string>

const int Seasons = 4;

**const std::array<std::string, Seasons> Snames** =

{"Spring", "Summer", "Fall", "Winter"};

**void fill(std::array<double, Seasons> \* pa); // 定义了指针，类型和长度**

void show(std::array<double, Seasons> da);

int main()

{

**std::array<double, Seasons> expenses;**

fill(&expenses);

show(expenses);

return 0;

}

void fill(std::array<double, Seasons> \* pa)

{

using namespace std;

for (int i = 0; i < Seasons; i++)

{

cout << "Enter " << Snames[i] << " expenses: ";

**cin >> (\*pa)[i];**

}

}

void show(std::array<double, Seasons> da)

{

using namespace std;

double total = 0.0;

cout << "\nEXPENSES\n";

for (int i = 0; i < Seasons; i++)

{

cout << Snames[i] << ": $" << da[i] << endl;

total += da[i];

}

cout << "Total Expenses: $" << total << endl;

}

* **Recursion**

If a recursive function calls itself, then the newly called function calls itself, and so on, ad infinitum unless the code includes something to terminate the chain of calls. The usual method is to make the recursive call part of an if statement. For example, a type void recursive function called recurs() can have a form like this:

void recurs(argumentlist)

{

statements1

if (test)

recurs(arguments)

statements2

}

**As long as the if statement remains true**, **each call to recurs() executes statements1** and then invokes a new incarnation of recurs() **without reaching statements2**.

**When the if statement becomes false**, the current call then proceeds to statements2. Then when the current call terminates, program control returns to the previous version of recurs() that called it. Then, that version of recurs() completes executing its statements2 section and terminates, returning control to the prior call, and so on. Thus, if recurs() undergoes five recursive calls, first the **statements1** section is **executed five times in the order in which the functions were called**, and then the **statements2** section is **executed five times in the opposite order from the order in which the functions were called.**

#include <iostream>

void countdown(int n);

int main()

{

countdown(4); // call the recursive function

return 0;

}

**void countdown(int n)**

{

using namespace std;

cout << "Counting down ... " << n << endl;

if (n > 0)

**countdown(n-1);** // function calls itself

cout << n << ": Kaboom!\n";

}

Note that each recursive call creates its own set of variables, so by the time the program reaches the fifth call, it has five separate variables called n, each with a different value.

**Recursion with Multiple Recursive Calls**

Consider below approach to draw a ruler. Mark the two ends, locate the midpoint, and mark it. Then apply this same procedure to the left half of the ruler and then to the right half. If you want more subdivisions, apply the same procedure to each of the current subdivisions. This recursive approach is sometimes called the ***divide-and-conquer strategy***.

**<List7-17.cpp>**

#include <iostream>

**const int Len = 66;**

**const int Divs = 6;**

void subdivide(**char ar[]**, int low, int high, int level);

int main()

{

char ruler[Len];

int i;

for (i = 1; i < Len - 2; i++)

**ruler**[i] = ' ';

ruler[Len - 1] = '\0';

int max = Len - 2;

int min = 0;

ruler[min] = ruler[max] = '|';

std::cout << ruler << std::endl;

for (**i = 1**; **i <= Divs**; i++)

{

**subdivide(ruler, min, max, i);**

std::cout << ruler << std::endl;

for (int j = 1; j < Len - 2; j++)

ruler[j] = ' '; // reset to blank ruler

}

return 0;

}

void subdivide(char ar[], int low, int high, int level)

{

if (level == 0)

return;

int mid = (high + low) / 2;

ar[mid] = '|';

subdivide(ar, **low, mid**, level - 1);

subdivide(ar, **mid, high**, level - 1);

}

* **Pointers to Functions**

Functions, like data items, have addresses. A function’s address is the memory address at which the stored machine language code for the function begins.

**Obtaining the address of a Function**

Obtaining the address of a function is simple: **You just use the function name without trailing parentheses**. That is, if think() is a function, then think is the address of the function.

process(think); // passes address of think() to process()

thought(think()); // passes return value of think() to thought()

**Declaring a Pointer to a Function**

A pointer to a function has to specify to what type of function the pointer points. This means the declaration should identify the function’s return type and the function’s signature (its argument list). That is, the declaration should provide the same information about a function that a function prototype does.

double pam(int); // prototype

Here’s what a declaration of an appropriate pointer type looks like:

double (\*pf)(int); // pf points to a function that takes

// one int argument and that

// returns type double

In general, to declare a pointer to a particular kind of function, you can first write a prototype for a regular function of the desired kind and then ***replace the function name with an expression in the form (\*pf).*** In this case, pf is a pointer to a function of that type.

double (\*pf)(int); **// pf points to a function that returns double**

double \*pf(int); // pf() is a function that returns a pointer-to-double

After you declare pf properly, you can assign to it the address of a matching function:

double pam(int);

double (\*pf)(int);

pf = pam; // pf now points to the pam() function

Note that pam() has to match pf in both signature and return type. Function using pointer-to-function could have the following prototype:

void estimate(int lines, **double (\*pf)(int)**);

This declaration says the second argument is a pointer to a function that has an int argument and a double return value.

**Using a Pointer to Invoke a Function**

double pam(int);

double (\*pf)(int);

pf = pam; // pf now points to the pam() function

double x = pam(4); // call pam() using the function name

double **y = (\*pf)(5);** // call pam() using the pointer pf

Actually, C++ also allows you to use pf as if it were a function name:

double **y = pf(5);** // also call pam() using the pointer pf

***How can pf and (\*pf) be equivalent?***

One school of thought maintains that because **pf is a pointer to a function**, **\*pf is a function; hence, you should use (\*pf)() as a function** call. A second school maintains that because the name of a function is a pointer to that function, a pointer to that function should act like the name of a function; hence you should use pf() as a function call. C++ takes the compromise view that both forms are correct.

#include <iostream>

**double betsy(int);**

**double pam(int);**

**void estimate(int lines, double (\*pf)(int));**

int main()

{

using namespace std;

int code;

cout << "How many lines of code do you need? ";

cin >> code;

cout << "Here's Betsy's estimate:\n";

**estimate(code, betsy);**

cout << "Here's Pam's estimate:\n";

**estimate(code, pam);**

return 0;

}

double betsy(int lns)

{

return 0.05 \* lns;

}

double pam(int lns)

{

return 0.03 \* lns + 0.0004 \* lns \* lns;

}

**void estimate(int lines, double (\*pf)(int))**

{

using namespace std;

cout << lines << " lines will take ";

**cout << (\*pf)(lines) << " hour(s)\n";**

}

**Variations on the Theme of Function Pointers**

const double \* f1(const double ar[], int n);

const double \* f2(const double [], int);

const double \* f3(const double \*, int);

The signatures might look different, but they are the same. Next, suppose you wish to declare a pointer that can point to one of these three functions. The technique, you’ll recall, is if pa is the desired pointer, take the prototype for a target function and replace the function name with (\*pa):

const double \* (\*p1)(const double \*, int); // = f1

With the C++11 automatic type deduction feature, you can simplify this a bit:

auto p2 = f2; // C++11 automatic type deduction

Now consider the following statements:

cout << (\*p1)(av,3) << ": " << \*(\*p1)(av,3) << endl;

cout << p2(av,3) << ": " << \*p2(av,3) << endl;

Both (\*p1)(av,3) and p2(av,3), recall, represent calling the pointed-to functions (f1() and f2(), in this case) with av and 3 as arguments. Therefore, what should print are the return values of these two functions. The return values are type const double \* (that is, address of double values). So the first part of each cout expression should print the address of a double value. **To see the actual value stored at the addresses, we need to apply the \* operator to these addresses,** and that’s what the expressions \*(\*p1)(av,3) and \*p2(av,3) do.

const double \* (\*pa[3])(const double \*, int) = {f1,f2,f3};

With three functions to work with, it could be handy to have an array of function pointers. Then one can use a for loop to call each function, via its pointer, in turn.

const double \* (\*pa[3])(const double \*, int) = {f1,f2,f3};

**Operator precedence ranks [] higher than \*,** so \*pa[3] says pa is an array of three pointers. The rest of the declaration indicates what each pointer points to: a function with a signature of const double \*, int and a return type of const double \*. **Hence, pa is an array of three pointers, each of which is a pointer to a function that takes a const double \* and int as arguments and returns a const double \*.**

**后面还有一点点没有看完.**

* **Simplifying with typedef**

typedef const double \*(\***p\_fun**)(const double \*, int); // **p\_fun now a type name**

p\_fun p1 = f1; // p1 points to the f1() function

You then can use this type to build elaborations:

p\_fun pa[3] = {f1,f2,f3}; // pa an array of 3 function pointers

p\_fun (\*pd)[3] = &pa; // pd points to an array of 3 function pointers