# 第6章函数

## 6.1 函数基础

**调用函数**

A function call does two things: It initializes the function’s parameters from the corresponding arguments, and it transfers control to that function. Execution of the calling function is suspended and execution of the called function begins.

Execution of a function ends when a return statement is encountered. Like a

function call, the return statement does two things: It returns the value (if any) in the return, and it transfers control out of the called function back to the calling function.

**函数的形参列表**

void f1()// implicit void parameter list

{

/\* ... \*/

}

void f2(void) // explicit void parameter list

{

/\* ... \*/

}

**函数返回类型**

Most types can be used as the return type of a function. In particular, the return type can be void, which means that the function does not return a value. However, the return type may not be an array type (§ 3.5, p. 113) or a function type. However, a function may return a pointer to an array or a function. We’ll see how to define functions that return pointers (or references) to arrays in § 6.3.3 (p. 228) and how to return pointers to functions in § 6.7 (p. 247).

### 6.1.1 局部对象

In C++, names have scope (§ 2.2.4, p. 48), and objects have lifetimes. It is important to understand both of these concepts.

**自动对象**

Objects that exist only while a block is executing are known as **automatic objects**. After execution exits a block, the values of the automatic objects created in that block are undefined.

**局部静态对象**

It can be useful to have a local variable whose lifetime continues across calls to the function. We obtain such objects by defining a local variable as static. Each **local static object** is initialized before the first time execution passes through the object’s definition. Local statics are not destroyed when a function ends; they are destroyed when the program terminates.

If a local static has no explicit initializer, it is value initialized (§ 3.3.1, p. 98), meaning that local statics of built-in type are initialized to zero.

### 6.1.2 函数声明

A function declaration is just like a function definition except that a declaration has no function body. In a declaration, a semicolon replaces the function body.

// parameter names chosen to indicate that the iterators denote a range of // values to print

void print(vector<int>::const\_iterator beg,

vector<int>::const\_iterator end);

These three elements—the return type, function name, and parameter types—describe the function’s interface. They specify all the information we need to call the function. Function declarations are also known as the function prototype.

**在头文件中进行函数声明**

Recall that variables are declared in header files (§ 2.6.3, p. 76) and defined in source files. For the same reasons, functions should be declared in header files and defined in source files.

### 6.1.3 分离式编译

To allow programs to be written in logical parts, C++ supports what is commonly known as separate compilation. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

## 6.2 参数传递

As we’ve seen, each time we call a function, its parameters are created and initialized by the arguments passed in the call.

Note

Parameter initialization works the same way as variable initialization.

As with any other variable, the type of a parameter determines the interaction between the parameter and its argument. If the parameter is a reference (§ 2.3.1, p.50), then the parameter is bound to its argument. Otherwise, the argument’s value is copied.

When a parameter is a reference, we say that its corresponding argument is

“passed by reference” or that the function is “called by reference.”

As with any other reference, a reference parameter is an alias for the object to which it is bound; that is, the parameter is an alias for its corresponding argument.

When the argument value is copied, the parameter and argument are independent objects. We say such arguments are “passed by value” or alternatively that the function is “called by value.”

### 6.2.1 传值参数

When we initialize a nonreference type variable, the value of the initializer is copied. Changes made to the variable have no effect on the initializer:

int n = 0; // ordinary variable of type int

int i = n; // i is a copy of the value in n

i = 42; // value in i is changed; n is unchanged

Passing an argument by value works exactly the same way; nothing the function does to the parameter can affect the argument.

**指针形参**

Pointers (§ 2.3.2, p. 52) behave like any other nonreference type. When we copy a pointer, the value of the pointer is copied. After the copy, the two pointers are distinct. However, a pointer also gives us indirect access to the object to which that pointer points. We can change the value of that object by assigning through the pointer (§ 2.3.2, p. 55):

int n = 0, i = 42;

int \*p = &n, \*q = &i; // p points to n; q points to i

\*p = 42; // value in n is changed; p is unchanged

p = q; // p now points to i; values in i and n are unchanged

The same behavior applies to pointer parameters:

// function that takes a pointer and sets the pointed-to value to zero

void reset(int \*ip)

{

\*ip = 0; // changes the value of the object to which ip points

ip = 0; // changes only the local copy of ip; the argument is unchanged

}

After a call to reset, the object to which the argument points will be 0, but the pointer argument itself is unchanged:

int i = 42;

reset(&i); // changes i but not the address of i

cout << "i = " << i << endl; // prints i = 0

Best Practices

Programmers accustomed to programming in C often use pointer parameters

to access objects outside a function. In C++, programmers generally use

reference parameters instead.

### 6.2.2 传引用参数

Recall that operations on a reference are actually operations on the object to which the reference refers (§ 2.3.1, p. 50):

int n = 0, i = 42;

int &r = n; // r is bound to n (i.e., r is another name for n)

r = 42; // n is now 42

r = i; // n now has the same value as i

i = r; // i has the same value as n

As one example, we can rewrite our reset program from the previous section to take a reference instead of a pointer:

// function that takes a reference to an int and sets the given object to // zero

void reset(int &i) // i is just another name for the object passed to reset

{

i = 0; // changes the value of the object to which i refers

}

When we call this version of reset, we pass an object directly; there is no need to pass its address:

int j = 42;

reset(j); // j is passed by reference; the value in j is changed

cout << "j = " << j << endl; // prints j = 0

In this call, the parameter i is just another name for j. Any use of i inside reset is a use of j.

**使用引用避免拷贝**

// compare the length of two strings

bool isShorter(const string &s1, const string &s2)

{

return s1.size() < s2.size();

}

As we’ll see in § 6.2.3 (p. 213), functions should use references to const for reference parameters they do not need to change.

Best Practices

Reference parameters that are not changed inside a function should be

references to const.

**使用引用形参返回额外信息**

A function can return only a single value. However, sometimes a function has more than one value to return. Reference parameters let us effectively return multiple results.

As an example, we’ll define a function named find char that will return the

position of the first occurrence of a given character in a string. We’d also like the function to return a count of how many times that character occurs.

How can we define a function that returns a position and an occurrence count? We could define a new type that contains the position and the count. An easier solution is to pass an additional reference argument to hold the occurrence count:

// returns the index of the first occurrence of c in s

// the reference parameter occurs counts how often c occurs

string::size\_type find\_char(const string &s, char c, string::size\_type &occurs)

{

auto ret = s.size(); // position of the first occurrence, if any

occurs = 0; // set the occurrence count parameter

for (decltype(ret) i = 0; i != s.size(); ++i)

{

if (s[i] == c)

{

if (ret == s.size())

ret = i; // remember the first occurrence of c

++occurs; // increment the occurrence count

}

}

return ret; // count is returned implicitly in occurs

}

auto index = find\_char(s, 'o', ctr);

After the call, the value of ctr will be the number of times o occurs, and index will refer to the first occurrence if there is one. Otherwise, index will be equal to s.size() and ctr will be zero.

### 6.2.3 const 形参和实参

When we use parameters that are const, it is important to remember the discussion of top-level const from § 2.4.3 (p. 63).

As we saw in that section, a top-level const is one that applies to the object itself:

const int ci = 42; // we cannot change ci; const is top-level

int i = ci; // ok: when we copy ci, its top-level const is ignored

int \* const p = &i; // const is top-level; we can't assign to p

\*p = 0; // ok: changes through p are allowed; i is now 0

### 6.2.4 数组形参

**需求**

因为数组是以指针的形式传递给数组的，所以函数一开始不知道数组的确切尺寸，调用者应该提供一些额外的信息，管理指针形参有三种常用技术。

**技术一：使用标记指定数组长度**

void print(const char \*cp)

{

if (cp) // if cp is not a null pointer

while (\*cp) // so long as the character it points to is not a null

// character

cout << \*cp++; // print the character and advance the pointer

}

**缺点**

This convention works well for data where there is an obvious end-marker value (like the null character) that does not appear in ordinary data. It works less well with data, such as ints, where every value in the range is a legitimate value.

**技术二：使用标准库规范**

// function

void print(const int \*beg, const int \*end)

{

// print every element starting at beg up to but not including end

while (beg != end)

cout << \*beg++ << endl; // print the current element

// and advance the pointer

}

void main

{

int j[2] = {0, 1};

// j is converted to a pointer to the first element in j

// the second argument is a pointer to one past the end of j

print(begin(j), end(j));

}

**技术三：显示传递一个表示数组大小的形参**

// const int ia[] is equivalent to const int\* ia

// size is passed explicitly and used to control access to elements of ia

void print(const int ia[], size\_t size)

{

for (size\_t i = 0; i != size; ++i)

cout << ia[i] << endl;

}

void main

{

int j[] = { 0, 1 }; // int array of size 2

print(j, end(j) - begin(j));

}

**数组形参和const**

When a function does not need write access to the array elements, the array parameter should be a pointer to const (§ 2.4.2, p. 62). A parameter should be a plain pointer to a nonconst type only if the function needs to change element values.

**数组引用形参**

Note

The parentheses around &arr are necessary (§ 3.5.1, p. 114):

f(int &arr[10]) // error: declares arr as an array of references

f(int (&arr)[10]) // ok: arr is a reference to an array of ten ints

**传递多维数组**

void print(int matrix[][10], int rowSize)

{

/\* . . . \*/

}

// equivalent definition

// matrix points to the first element in an array whose elements are arrays // of ten ints

void print(int (\*matrix)[10], int rowSize)

{

/\* . . . \*/

}

declares matrix as a pointer to an array of ten ints.

Note

Again, the parentheses around \*matrix are necessary:

int \*matrix[10]; // array of ten pointers

int (\*matrix)[10]; // pointer to an array of ten ints