# 第1章

## 1.5 类简介

The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define class types that behave as naturally as the built-in types.

To use a class we need to know three things:

• What is its name?

• Where is it defined?

• What operations does it support?

### 1.5.1. The Sales\_item Class

The purpose of the Sales\_item class is to represent the total revenue, number of copies sold, and average sales price for a book.

Sales\_item item;

In addition to being able to define variables of type Sales\_item, we can:

• Call a function named isbn to fetch the from a Sales\_item object.

• Use the input (>>) and output (<<) operators to read and write objects of type Sales\_item.

• Use the assignment operator (=) to assign one Sales\_item object to another.

• Use the addition operator (+) to add two Sales\_item objects. The two objects must refer to the same ISBN . The result is a new Sales\_item object whose ISBN is that of its operands and whose number sold and revenue are the sum of the corresponding values in its operands.

• Use the compound assignment operator (+=) to add one Sales\_item object

into another.

**Using File Redirection**

It can be tedious to repeatedly type these transactions as input to the

programs you are testing. Most operating systems support file redirection,

which lets us associate a named file with the standard input and the standard output:

$ addItems <infile >outfile

Assuming $ is the system prompt and our addition program has been compiled into an executable file named addItems.exe (or addItems on UNIX systems), this command will read transactions from a file named infile and write its output to a file named outfile in the current directory.

### 1.5.2 初始成员函数

# 第3章 字符串、向量和数组

## 3.1 命名空间的using声明

These names use the scope operator (::) (§ 1.2, p. 8), which says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. Thus, std::cin says that we want to use the name cin from the namespace std.

Referring to library names with this notation can be cumbersome. Fortunately, there are easier ways to use namespace members. The safest way is a using declaration.§ 18.2.2 (p. 793) covers another way to use names from a namespace.

#include <iostream>

// using declaration; when we use the name cin, we get the one from the //namespace std

using std::cin;

int main()

{

int i;

cin >> i; // ok: cin is a synonym for std::cin

cout << i; // error: no using declaration; we must use the full name

std::cout << i; // ok: explicitly use cout from namepsace std

return 0;

}

**每个名字都需要独立的using 声明**

#include <iostream>

// using declarations for names from the standard library

using std::cin;

using std::cout; using std::endl;

int main()

{

cout << "Enter two numbers:" << endl;

int v1, v2;

cin >> v1 >> v2;

cout << "The sum of " << v1 << " and " << v2

<< " is " << v1 + v2 << endl;

return 0;

}

Recall that C++ programs are free-form, so we can put each using declaration on its own line or combine several onto a single line. The

important part is that there must be a using declaration for each name we use, and each declaration must end in a semicolon.

**头文件不应包含using声明**

The reason is that the contents of a header are copied into the including program’s text. If a header has a using declaration, then every program that includes that header gets that same using declaration. As a result, a program that didn’t intend to use the specified library name might encounter unexpected name conflicts.

## 3.2 标准库类型string

A string is a variable-length sequence of characters. To use the string type, we must include the string header. Because it is part of the library, string is defined in the std namespace. Our examples assume the following code:

#include <string>

using std::string;

Note

In addition to specifying the operations that the library types provide, the standard also imposes efficiency requirements on implementors. As a result, library types are efficient enough for general use.

### 3.2.1 定义和初始化字符串

**Table 3.1. Ways to Initialize a string**

string s1 Default initialization; s1 is the empty string.

string s2(s1) s2 is a copy of s1.

string s2 = s1 Equivalent to s2(s1),s2 is a copy of s1.

string s3(“value”) s3 is a copy of the string literal,not including the null.

string s3 = “value” Equivalent to s3(“value”),s3 is a copy of the string literal.

string s4(n,’c’) Initialize with n copies of the character ‘c’.

**直接初始化和拷贝初始化**

When we initialize a variable using =, we are asking the compiler to copy initialize the object by copying the initializer on the right-hand side into the object being created. Otherwise, when we omit the =, we use direct initialization.

When we have a single initializer, we can use either the direct or copy form of initialization. When we initialize a variable from more than one value, such as in the initialization of s4 above, we must use the direct form of initialization:

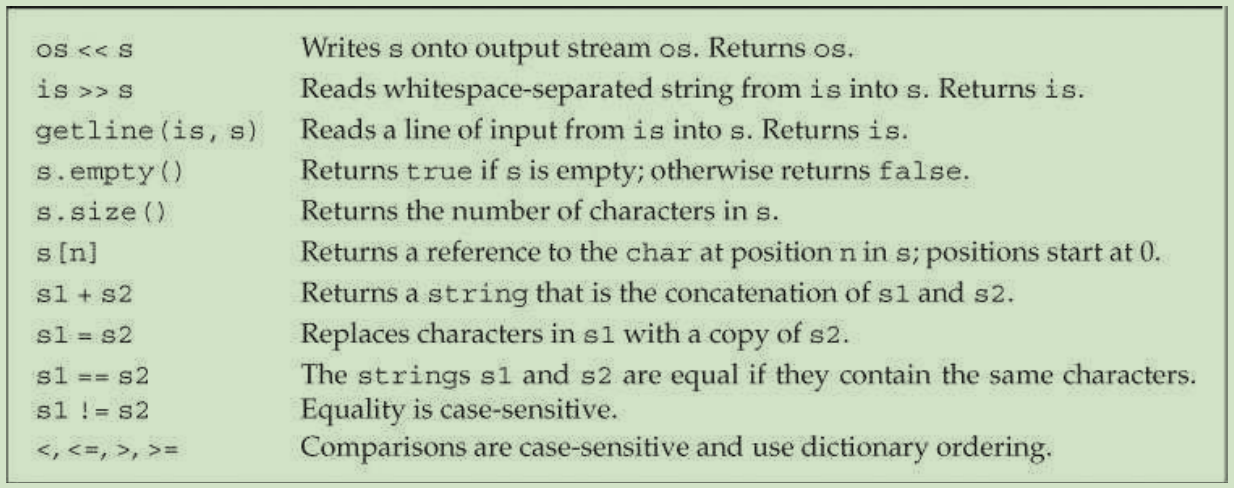
string s5 = "hiya"; // copy initialization

string s6("hiya"); // direct initialization

string s7(10, 'c'); // direct initialization; s7 is cccccccccc

### 3.2.2 string 对象上的操作

**Table 3.2. string Operations**



**读写string对象**

int main()

{

string s; // empty string

cin >> s; // read a whitespace-separated string into s

cout << s << endl; // write s to the output

return 0;

}

So, if the input to this program is Hello World! (note leading and trailing spaces),then the output will be Hello with no extra spaces.

**读取未知数量的string对象**

int main()

{

string word;

while (cin >> word) // read until end-of-file

cout << word << endl; // write each word followed by a new line

return 0;

}

If the stream is valid—it hasn’t hit end-of-file or encountered an invalid input—then the body of the while is executed.

**使用getline读取一整行**

Sometimes we do not want to ignore the whitespace in our input. In such cases, we can use the getline function instead of the >> operator. The getline function takes an input stream and a string. This function reads the given stream up to and including the first newline and stores what it read— not including the newline—in its string argument. After getline sees a newline, even if it is the first character in the input, it stops reading and returns. If the first character in the input is a newline, then the resulting string is the empty string.

Like the input operator, getline returns its istream argument. As a result, we can use getline as a condition just as we can use the input operator as a condition (§ 1.4.3, p. 14). For example, we can rewrite the previous program that wrote one word per line to write a line at a time instead:

int main()

{

string line;

// read input a line at a time until end-of-file

while (getline(cin, line))

cout << line << endl;

return 0;

}

Because line does not contain a newline, we must write our own. As usual, we use endl to end the current line and flush the buffer.

Note

The newline that causes getline to return is discarded; the newline is not

stored in the string.

**string 的empty和size操作**

// read input a line at a time and discard blank lines

while (getline(cin, line))

if (!line.empty())

cout << line << endl;

string line;

// read input a line at a time and print lines that are longer than 80 //characters

while (getline(cin, line))

if (line.size() > 80)

cout << line << endl;

**string::size\_type类型**

The string class—and most other library types—defines several companion types. These companion types make it possible to use the library types in a machine-independent manner. The type size\_type is one of these companion types. To use the size\_type defined by string, we use the scope operator to say that the name size\_type is defined in the string class.

Although we don’t know the precise type of string::size\_type, we do know

that it is an unsigned type (§ 2.1.1, p. 32) big enough to hold the size of any string. Any variable used to store the result from the string size operation should be of type string::size\_type.

Under the new standard, we can ask the compiler to provide the appropriate type by using auto or decltype (§ 2.5.2, p. 68):

auto len = line.size(); // len has type string::size\_type

For example, if n is an int that holds a negative value, then s.size() < n will almost surely evaluate as true. It yields true because the negative value in n will convert to a large unsigned value.

Tip

You can avoid problems due to conversion between unsigned and int by not using ints in expressions that use size().

**比较string 对象**

These operators use the same strategy as a (case-sensitive) dictionary:

1. If two strings have different lengths and if every character in the shorter string is equal to the corresponding character of the longer string, then the shorter string is less than the longer one.

2. If any characters at corresponding positions in the two strings differ, then the result of the string comparison is the result of comparing the first character at which the strings differ.

As an example, consider the following strings:

string str = "Hello";

string phrase = "Hello World";

string slang = "Hiya";

Using rule 1, we see that str is less than phrase. By applying rule 2, we see that slang is greater than both str and phrase.

**对string对象赋值**

In general, the library types strive to make it as easy to use a library type as it is to use a built-in type. To this end, most of the library types support assignment. In the case of strings, we can assign one string object to another:

string st1(10, 'c'), st2; // st1 is cccccccccc; st2 is an empty string

st1 = st2; // assignment: replace contents of st1 with a copy of st2

// both st1 and st2 are now the empty string

**两个string对象相加**

string s1 = "hello, ", s2 = "world\n";

string s3 = s1 + s2; // s3 is hello, world\n

s1 += s2; // equivalent to s1 = s1 + s2

**字面值和string对象相加**

When we mix strings and string or character literals, at least one operand to each + operator must be of string type:

string s4 = s1 + ", "; // ok: adding a string and a literal

string s5 = "hello" + ", "; // error: no string operand

string s6 = s1 + ", " + "world"; // ok: each + has a string operand

string s7 = "hello" + ", " + s2; // error: can't add string literals

The initializations of s4 and s5 involve only a single operation each, so it is easy to see whether the initialization is legal. The initialization of s6 may appear surprising, but it works in much the same way as when we chain together input or output expressions (§ 1.2, p. 7). This initialization groups as

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

The subexpression s1 + ", " returns a string, which forms the left-hand operand of the second + operator. It is as if we had written

string tmp = s1 + ", "; // ok: + has a string operand

s6 = tmp + "world"; // ok: + has a string operand

On the other hand, the initialization of s7 is illegal, which we can see if we parenthesize the expression:

string s7 = ("hello" + ", ") + s2; // error: can't add string literals

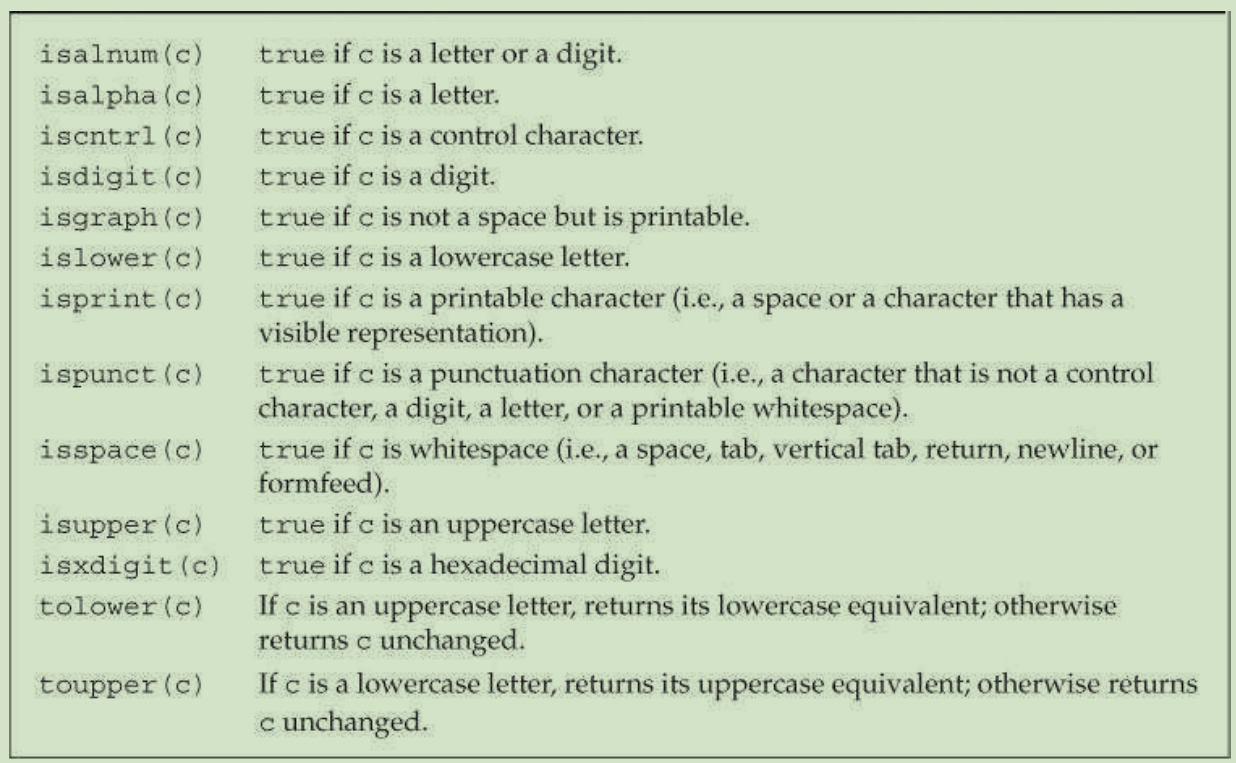
Now it should be easy to see that the first subexpression adds two string literals. There is no way to do so, and so the statement is in error.

Warning

For historical reasons, and for compatibility with C, string literals are not standard library strings. It is important to remember that these types differ when you use string literals and library strings.

### 3.2.3 处理string对象中的字符

**Table 3.3. cctype Functions**



<https://www.programiz.com/cpp-programming/library-function/cctype/ispunct>

Advice: Use the C++ Versions of C Library Header

Ordinarily, C++ programs should use the c name versions of headers and

not the name .h versions. That way names from the standard library are

consistently found in the std namespace. Using the .h headers puts the

burden on the programmer to remember which library names are inherited

from C and which are unique to C++.

## 3.3 标准库类型vector

## 3.4 迭代器介绍

## 3.5 数组

## 3.6 多维数组

## 小结

# 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

## 6.3 返回类型和返回语句

A return statement terminates the function that is currently executing and returns control to the point from which the function was called. There are two forms of return statements:

return;

return expression;

### 6.3.1 无返回值函数

In a void function, an implicit return takes place after the function’s last statement.

Typically, void functions use a return to exit the function at an intermediate point. This use of return is analogous to the use of a break statement (§ 5.5.1, p.190) to exit a loop.

void swap(int &v1, int &v2)

{

// if the values are already the same, no need to swap, just return

if (v1 == v2)

return;

// if we're here, there's work to do

int tmp = v2;

v2 = v1;

v1 = tmp;

// no explicit return necessary

}

### 6.3.2 有返回值类型

// incorrect return values, this code will not compile

bool str\_subrange(const string &str1, const string &str2)

{

// same sizes: return normal equality test

if (str1.size() == str2.size())

return str1 == str2; // ok: == returns bool

// find the size of the smaller string;

auto size = (str1.size() < str2.size())? str1.size() : str2.size();

// look at each element up to the size of the smaller string

for (decltype(size) i = 0; i != size; ++i)

{

if (str1[i] != str2[i])

return;

// error #1: no return value; compiler should detect this error

}

// error #2: control might flow off the end of the function without a return the compiler might not detect this error

}

The second error occurs because the function fails to provide a return after the loop. If we call this function with one string that is a subset of the other, execution would fall out of the for. There should be a return to handle this case. The compiler may or may not detect this error. If it does not detect the error, what happens at run time is undefined.

Warning

Failing to provide a return after a loop that contains a return is an error.However, many compilers will not detect such errors.

**值是如何被返回的**

Values are returned in exactly the same way as variables and parameters are

initialized: The return value is used to initialize a temporary at the call site, and that temporary is the result of the function call.

As with any other reference, when a function returns a reference, that reference is just another name for the object to which it refers. As an example, consider a function that returns a reference to the shorter of its two string parameters:

// return a reference to the shorter of two strings

const string &shorterString(const string &s1, const string&s2)

{

return s1.size() <= s2.size() ? s1 : s2;

}

The parameters and return type are references to const string. The strings are not copied when the function is called or when the result is returned.

**不要返回局部对象的引用和指针**

When a function completes, its storage is freed (§ 6.1.1, p. 204). After a function terminates, references to local objects refer to memory that is no longer valid:

// disaster: this function returns a reference to a local object

const string &manip()

{

string ret;

// transform ret in some way

if (!ret.empty())

return ret; // WRONG: returning a reference to a local object!

else

return "Empty"; // WRONG: "Empty" is a local temporary string

}

Both of these return statements return an undefined value—what happens if we try to use the value returned from manip is undefined. In the first return, it should be obvious that the function returns a reference to a local object. In the second case, the string literal is converted to a local temporary string object. That object, like the string named s, is local to manip. The storage in which the temporary resides is freed when the function ends.

Both returns refer to memory that is no longer available.

Tip

One good way to ensure that the return is safe is to ask:

To what preexisting object is the reference referring?

**返回类类型和调用操作符的函数**

Like any operator the call operator has associativity and precedence (§ 4.1.2, p. 136).The call operator has the same precedence as the dot and arrow operators (§ 4.6, p.150). Like those operators, the call operator is left associative. As a result, if a function returns a pointer, reference or object of class type, we can use the result of a call to call a member of the resulting object.

// call the size member of the string returned by shorterString

auto sz = shorterString(s1, s2).size();

Because these operators are left associative, the result of shorterString is the left-hand operand of the dot operator. That operator fetches the size member of that string. That member is the left-hand operand of the second call operator.

**引用返回左值**

Whether a function call is an lvalue (§ 4.1.1, p. 135) depends on the return type of the function. Calls to functions that return references are lvalues; other return types yield rvalues. A call to a function that returns a reference can be used in the same ways as any other lvalue. In particular, we can assign to the result of a function that returns a reference to nonconst:

char &get\_val(string &str, string::size\_type ix)

{

return str[ix]; // get\_val assumes the given index is valid

}

int main()

{

string s("a value");

cout << s << endl; // prints a value

get\_val(s, 0) = 'A'; // changes s[0] to A

cout << s << endl; // prints A value

return 0;

}

It may be surprising to see a function call on the left-hand side of an assignment. However, nothing special is involved. The return value is a reference, so the call is an lvalue. Like any other lvalue, it may appear as the left-hand operand of the assignment operator.

If the return type is a reference to const, then (as usual) we may not assign to the result of the call:

shorterString("hi", "bye") = "X"; // error: return value is const

# 第七章 类

The fundamental ideas behind **classes** are **data abstraction and encapsulation**. Data abstraction is a programming (and design) technique that relies on the separation of **interface** and **implementation**. The interface of a class consists of the operations that users of the class can execute. The implementation includes the class’ data members, the bodies of the functions that constitute the interface, and any functions needed to define the class that are not intended for general use.

Encapsulation enforces the separation of a class’ interface and implementation. A class that is encapsulated hides its implementation—users of the class can use the interface but have no access to the implementation.

A class that uses data abstraction and encapsulation defines **an abstract data type**. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. They can instead think abstractly about what the type does.

## 7.1 定义抽象数据类型

The Sales\_item class that we used in Chapter 1 is an abstract data type. We use a Sales\_item object by using its interface (i.e., the operations described in § 1.5.1 (p.20)). We have no access to the data members stored in a Sales\_item object. Indeed, we don’t even know what data members that class has.

Our Sales\_data class (§ 2.6.1, p. 72) is not an abstract data type. It lets users of the class access its data members and forces users to write their own operations. To make Sales\_data an abstract type, we need to define operations for users of Sales\_data to use. Once Sales\_data defines its own operations, we can encapsulate (that is, hide) its data members.

### 7.1.1 设计Sales\_data 类

Ultimately, we want Sales\_data to support the same set of operations as the

Sales\_item class.

Thus, the interface to Sales\_data consists of the following operations:

• An isbn member function to return the object’s ISBN

• A combine member function to add one Sales\_data object into another

• A function named add to add two Sales\_data objects

• A read function to read data from an istream into a Sales\_data object

• A print function to print the value of a Sales\_data object on an ostream

Sales\_data total; // variable to hold the running sum

if (read(cin, total)) { // read the first transaction

Sales\_data trans; // variable to hold data for the next transaction

while(read(cin, trans)) { // read the remaining transactions

if (total.isbn() == trans.isbn()) // check the isbns

total.combine(trans); // update the running total

else {

print(cout, total) << endl; // print the results

total = trans; // process the next book

}

}

print(cout, total) << endl; // print the last transaction

}

else { // there was no input

cerr << "No data?!" << endl; // notify the user

}

### 7.1.2 定义改进的Sales\_data 类

Nonmember functions that are part of the interface, such as add, read, and print, are declared and defined outside the class.

With this knowledge, we’re ready to write our revised version of Sales\_data:

struct Sales\_data {

// new members: operations on Sales\_data objects

std::string isbn() const { return bookNo; }

Sales\_data& combine(const Sales\_data&);

double avg\_price() const;

// data members are unchanged from § 2.6.1 (p. 72)

std::string bookNo;

unsigned units\_sold = 0;

double revenue = 0.0;

};

// nonmember Sales\_data interface functions

Sales\_data add(const Sales\_data&, const Sales\_data&);

std::ostream &print(std::ostream&, const Sales\_data&);

std::istream &read(std::istream&, Sales\_data&);

Note

Functions defined in the class are implicitly inline (§ 6.5.2, p. 238).

**定义成员函数**

Although every member must be declared inside its class, we can define a member function’s body either inside or outside of the class body. In Sales\_data, isbn is defined inside the class; combine and avg\_price will be defined elsewhere.

We’ll start by explaining the isbn function, which returns a string and has an empty parameter list:

std::string isbn() const { return bookNo; }

In this case, the block contains a single return statement that returns the bookNo data member of a Sales\_data object. The interesting thing about this function is how it gets the object from which to fetch the bookNo member.

**引入this**

For example, when we call

total.isbn()

the compiler passes the address of total to the implicit this parameter in isbn. It is as if the compiler rewrites this call as

// pseudo-code illustration of how a call to a member function is translated

Sales\_data::isbn(&total)

which calls the isbn member of Sales\_data passing the address of total.

Inside the body of a member function, we can use this. It would be legal, although unnecessary, to define isbn as

std::string isbn() const { return this->bookNo; }

Because this is intended to always refer to “this” object, this is a const pointer (§ 2.4.2, p. 62). We cannot change the address that this holds.

**引入const成员函数**

The other important part about the isbn function is the keyword const that follows the parameter list. The purpose of that const is to modify the type of the implicit this pointer.

By default, the type of this is a const pointer to the nonconst version of the class type.

**类作用域和成员函数**

It is worth noting that isbn can use bookNo even though bookNo is defined after isbn. As we’ll see in § 7.4.1 (p. 283), the compiler processes classes in two steps—the member declarations are compiled first, after which the member function bodies, if any, are processed. Thus, member function bodies may use other members of their class regardless of where in the class those members appear.

**在类的外部定义成员函数**

As with any other function, when we define a member function outside the class body, the member’s definition must match its declaration. That is, the return type, parameter list, and name must match the declaration in the class body. If the member was declared as a const member function, then the definition must also specify const after the parameter list. The name of a member defined outside the class must include the name of the class of which it is a member:

double Sales\_data::avg\_price() const {

if (units\_sold)

return revenue/units\_sold;

else

return 0;

}

The function name, Sales\_data::avg\_price, uses the scope operator (§ 1.2, p. 8) to say that we are defining the function named avg\_price that is declared in the scope of the Sales\_data class. Once the compiler sees the function name, the rest of the code is interpreted as being inside the scope of the class. Thus, when avg\_price refers to revenue and units\_sold, it is implicitly referring to the members of Sales\_data.

**定义一个返回this对象的函数**

The combine function is intended to act like the compound assignment operator, +=. The object on which this function is called represents the left-hand operand of the assignment. The right-hand operand is passed as an explicit argument:

Sales\_data& Sales\_data::combine(const Sales\_data &rhs)

{

units\_sold += rhs.units\_sold; // add the members of rhs into

revenue += rhs.revenue; // the members of ''this'' object

return \*this; // return the object on which the function was called

}

When our transaction-processing program calls

total.combine(trans); // update the running total

the address of total is bound to the implicit this parameter and rhs is bound to trans. Thus, when combine executes

units\_sold += rhs.units\_sold; // add the members of rhs into

the effect is to add total.units\_sold and trans.units\_sold, storing the result back into total.units\_sold.

our combine function must return a reference (§ 6.3.2, p. 226). Because the

left-hand operand is a Sales\_data object, the return type is Sales\_data&.

However, we do need to use this to access the object as a whole:

return \*this; // return the object on which the function was called

Here the return statement dereferences this to obtain the object on which the function is executing. That is, for the call above, we return a reference to total.

### 7.1.4 构造函数

Classes control object initialization by defining one or more special member functions known as **constructors**. The job of a constructor is to initialize the data members of a class object. A constructor is run whenever an object of a class type is created.

Constructors have the same name as the class. **Unlike other functions**, constructors have no return type. **Like other functions**, constructors have a (possibly empty) parameter list and a (possibly empty) function body. A class can have multiple constructors. **Like any other overloaded function** (§ 6.4, p. 230), the constructors must differ from each other in the number or types of their parameters.

**Unlike other member functions**, constructors may not be declared as **const** (§

7.1.2, p. 258). When we create a const object of a class type, the object does not assume its “constness” until after the constructor completes the object’s initialization. Thus, constructors can write to const objects during their construction.

**合成的默认构造函数**

The compiler-generated constructor is known as the synthesized default

constructor. For most classes, this synthesized constructor initializes each datamember of the class as follows:

• If there is an in-class initializer (§ 2.6.1, p. 73), use it to initialize the member.

• Otherwise, default-initialize (§ 2.2.1, p. 43) the member.

**某些类不能依赖于合成的默认构造函数**

**定义Sales\_data的构造函数**

For our Sales\_data class we’ll define four constructors with the following

parameters:

• An **istream&** from which to read a transaction.

• A **const string&** representing an ISBN , an **unsigned** representing the count of how many books were sold, and a **double** representing the price at which the books sold.

• A **const string&** representing an ISBN . This constructor will use default

values for the other members.

• An empty parameter list (i.e., the default constructor) which as we’ve just seen we must define because we have defined other constructors.

struct Sales\_data {

// constructors added

Sales\_data() = default;

Sales\_data(const std::string &s): bookNo(s) { }

Sales\_data(const std::string &s, unsigned n, double p): bookNo(s), units\_sold(n), revenue(p\*n) { }

Sales\_data(std::istream &);

// other members as before

std::string isbn() const { return bookNo; }

Sales\_data& combine(const Sales\_data&);

double avg\_price() const;

std::string bookNo;

unsigned units\_sold = 0;

double revenue = 0.0;

};

**=default的含义**

First, note that this constructor defines the default constructor because it takes no arguments. We are defining this constructor only because we want to provide other constructors as well as the default constructor. We want this constructor to do exactly the same work as the synthesized version we had been using.

Under the new standard, if we want the default behavior, we can ask the compiler to generate the constructor for us by writing = default after the parameter list. The =default can appear with the declaration inside the class body or on the definition outside the class body. Like any other function, if the = default appears inside the class body, the default constructor will be inlined; if it appears on the definition outside the class, the member will not be inlined by default.

**Warning**

The default constructor works for **Sales\_data** only because we provide

initializers for the data members with built-in type. If your compiler does not support in-class initializers, your default constructor should use the

constructor initializer list (described immediately following) to initialize every member of the class.

**构造函数初始值列表**

Sales\_data(const std::string &s): bookNo(s) { }

Sales\_data(const std::string &s, unsigned n, double p): bookNo(s), units\_sold(n), revenue(p\*n) { }

The new parts in these definitions are the colon and the code between it and the curly braces that define the (empty) function bodies. This new part is a **constructor initializer list**, which specifies initial values for one or more data members of the object being created.

When a member is omitted from the constructor initializer list, it is implicitly initialized using the same process as is used by the synthesized default constructor. In this case, those members are initialized by the in-class initializers. Thus, the constructor that takes a string is equivalent to

// has the same behavior as the original constructor defined above

Sales\_data(const std::string &s): bookNo(s), units\_sold(0), revenue(0){ }

It is usually best for a constructor to use an in-class initializer if one exists and gives the member the correct value. On the other hand, if your compiler does not yet support in-class initializers, then every constructor should explicitly initialize every member of built-in type.

**在类的外部定义构造函数**

Sales\_data::Sales\_data(std::istream &is) {

read(is, \*this); // read will read a transaction from is into this

object

}

Thus, Sales\_data::Sales\_data says that we’re defining the Sales\_data member named Sales\_data. This member is a constructor because it has the same name as its class.

To understand the call to read, remember that read’s second parameter is a

reference to a Sales\_data object. In § 7.1.2 (p. 259), we noted that we use this to access the object as a whole, rather than a member of the object. In this case, we use \*this to pass “this” object as an argument to the read function.

## 7.2 访问控制与封装

## 7.3 类的其他特性

## 7.4 类的作用域

## 7.5 构造函数再探

## 7.6 类的静态成员