

Chapter 6: Functions

6.2.2 Passing Arguments by Reference

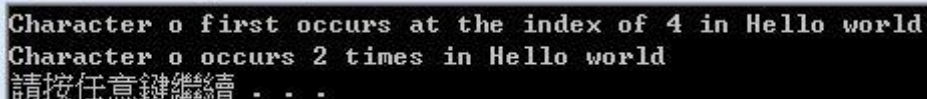
Example: write a program to return the index of the first occurrence of a char in a string and to count how many times this char occurs in a string

charOccur.cpp

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

// 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
}

int main() {
    string s = "Hello world";
    string::size_type cnt;
    auto index = find_char(s, 'o', cnt);
    cout << "Character o first occurs at the index of "
        << index << " in " << s << endl;
    cout << "Character o occurs " << cnt << " times"
        << " in " << s << endl;
    return 0;
}
```



```
Character o first occurs at the index of 4 in Hello world
Character o occurs 2 times in Hello world
請按任意鍵繼續 . . .
```

Exercise 6.1 In-class Coding Exercise

Ex61.cpp

Write a function `stringToLower` to change a given string to all lowercase.

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

( YOUR FUNCTION HERE )

int main()
{
    string s ;
    cout << "Enter a string: ";
    getline(cin, s);
    cout << "The input string in a lowercase is: "
         << stringToLower(s) << endl;
    return 0;
}
```

```
Enter a string: Hello WORLD!!
The input string in a lowercase is: hello world!!
```

Answer:

Summary: In C++ choosing a parameter-passing mechanism is an easily overlooked chore of the programmer that can affect correctness and efficiency. The rules are actually relatively simple:

- Call by reference is required for any object that may be altered by the function.
- Call by value is appropriate for small objects that should not be altered by the function. This generally includes primitive types and also function objects.
- Call by constant reference is appropriate for large objects that should not be altered by the function. This generally includes library containers such as vector, general class types, and even string.

6.2.4 Array Parameters

(see ppt)

A function can have a parameter for an entire array so that when the function is called, the

argument that is plugged in for this formal parameter is an entire array. However, a parameter for an entire array is neither a call-by-value parameter nor call-by-reference parameter. It is a new kind of formal parameter referred to as an **array parameter**.

```
| void fillUp(int a[], size_t size)
| {
|     // fillUp the array ...
| }
```

The parameter `int a[]` is an array parameter and `size_t` is a machine-specific unsigned type that is large enough to hold the size of any object in memory. In `int a[]`, the empty square brackets with no index expressed inside, are what C++ uses to indicate an array parameter.

An array parameter is not quite a call-by-reference parameter but for most practical purposes, it behaves very much like a call-by-reference parameter. When passing the array as an argument, you simply pass the name of array (the memory address).

```
| ...
| int a[20];
| fillUp(a, 20);
| ...
```

When an array argument is plugged in for an array parameter, all that is given to the function is the address in memory of the first element (the one indexed by zero). In C++, when we use **the name of an array** in an expression, that name is automatically converted into **the memory address of the first element of the array**.

Because arrays are converted to pointers, when we pass an array to a function, we are actually passing a pointer to the array's first element. Thus,

```
| void fillUp(int a[], size_t size);
|
| void fillUp(int* a, size_t size);
```

are equivalent.

Exercise 6.2 In-class Coding Exercise

Ex62.cpp

Write a `sum` function that sums up all elements in an `int` array and return the sum.

```
| #include <iostream>
```

```
using namespace std;
```

```
(YOUR FUNCTION HERE)
```

```
int main()
{
    int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ib[3] = {1, 3, 8};
    cout << "The sum of ia is: " << sum(ia, 10) << endl;
    cout << "The sum of ib is: " << sum(ib, 3) << endl;
    return 0;
}
```

```
The sum of ia is: 55
The sum of ib is: 12
```

Answer:

Because arrays are passed as pointers, functions ordinarily **DO NOT** know the size of the array they are given. They must rely on additional information provided by the caller. We can explicitly pass a size parameter as we did in the previous exercise. It can also be achieved by `begin` and `end` iterators:

Exercise 6.3 In-class Coding Exercise

Alternative solution using `begin` and `end` iterators

Ex63.cpp

```
#include <iostream>
#include <iterator>

using namespace std;

(YOUR FUNCTION HERE)
```

```

int main()
{
    int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ib[3] = {1, 3, 8};
    cout << "The sum of ia is: " << sum(begin(ia), end(ia)) <<
endl;
    cout << "The sum of ib is: " << sum(begin(ib), end(ib)) <<
endl;
    return 0;
}

```

Answer:

From A to A+ : some twists on the **range-for loop** for the built-in array

Recall we can use range for loop to access the element in an array as we did for string and vector.

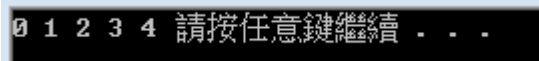
arrAutoSimple.cpp

```

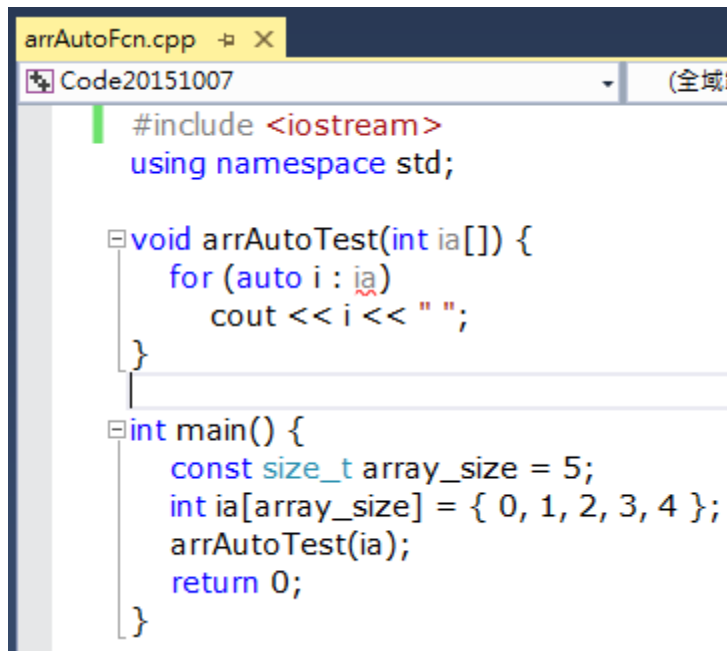
#include <iostream>
using namespace std;

int main(){
    const size_t array_size = 5;
    int ia[array_size] = {0, 1, 2, 3, 4};
    for (auto i : ia)
        cout << i << " ";
    return 0;
}

```



However, when we pass the array into a function, the range-for **WILL NOT** work since we lose an important information: the size of an array!

arrAutoFcn.cpp

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

void arrAutoTest(int ia[]) {
    for (auto i : ia)
        cout << i << " ";
}

int main() {
    const size_t array_size = 5;
    int ia[array_size] = { 0, 1, 2, 3, 4 };
    arrAutoTest(ia);
    return 0;
}
```

Remark: for modern C++ implementation, you should **almost always use vectors and iterators in preference to the lower-level arrays and pointers**. Well-designed programs use arrays and pointers **only in** the internals of class implementations where speed is essential.

In-class Exercise: rewrite the arrAutoFcn.cpp program using `vector`.

vecAutoFcn.cpp**Answer:**

Beyond A+: one way to work around the problem is to pass an array reference instead an array.

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

void arrAutoTest(int (&ia)[5])
{ for (auto i : ia)
    cout << i << " ";
}

int main() {
    const size_t array_size = 5;
    int ia[array_size] = { 0, 1, 2, 3, 4 };
    arrAutoTest(ia);
    return 0;
}
```

```
0 1 2 3 4 請按任意鍵繼續 . . .
```

Or a generic version:

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

template <std::size_t arr_size>
void arrAutoTest(int(&ia)[arr_size])
{ for (auto i : ia)
    cout << i << " ";
}

int main() {
    const size_t array_size1 = 7;
    int ia[array_size1] = { 0, 1, 2, 3, 4, 5, 6 };
    arrAutoTest(ia);
    cout << endl;
    const size_t array_size2 = 4;
    int ib[array_size2] = { 0, 1, 2, 3 };
    arrAutoTest(ib);
    return 0;
}
```

```
0 1 2 3 4 5 6
0 1 2 3 請按任意鍵繼續 . . .
```

63 Return Types

(see ppt)

Return non-reference and reference types

```
// return plural version of word if ctr isn't 1
string make_plural(size_t ctr, const string& word,
    const string &ending)
```

```
{
    return (ctr == 1) ? word : word + ending;
}

// find longer of two strings
const string& shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

Now let's look at a **terribly incorrect** program; what's wrong?

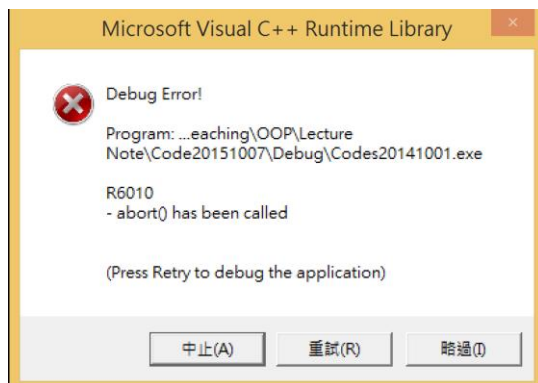
terribleCode.cpp

```
// this is a WRONG code

#include <string>
using namespace std;

string& manip(const string& s)
{
    string ret = s;
    ret += " is terrible";
    return ret;
}

int main(){
    string s = "The program ";
    string sTerrible = manip(s);
    return 0;
}
```



A:

6.4 Overloaded Functions

Function overloading can simplify the job of thinking up names for two or more functions that perform **different** versions of essentially the same task.

C++ compiler uses **the number of parameters** or **the types of parameters** to differentiate which version of the functions to be called.

Q: Given

```
| int gcd(int v1, int v2);
```

Are the following overloaded functions valid?

```
| int gcd(double v1, double v2);
```

```
| int gcd(int v1, int v2, int v3);
```

```
| int gcd(int v1, int v3);
```

```
| int gcd(double v1, int v3);
```

```
| double gcd(int v1, int v3);
```

A:

6.5.1 Functions with Default Arguments

(Ex) Given the following function declarations and calls, which, if any, of the calls are illegal? Which are legal but misleading?

```
| void init(int ht, int wd = 80, char bckgrnd = ' ');
|
| (a) init();
| (b) init(24,10);
| (c) init(14, '*');
```

(Ex) getline function

The C++ string class defines the **global function** `getline()` to read strings from an I/O stream. The `getline()` function reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then `getline()` will use *delimiter* to decide when to stop reading data.

Syntax:

```
istream& getline( istream& is, string& s, char delimiter = '\n' );
```

(Ex)

```
string s;  
getline( cin, s );  
cout << "You entered " << s << endl;
```

What does the above code fragment do?

A:

(Ex)

```
string s;  
getline( cin, s, '#' );  
cout << "You entered " << s << endl;
```

What does the above code fragment do?

A:

Exercise 6.4 In-class Coding Exercise: write a program to read some words and ignore those words after '!'. Below is a sample run:

```
Enter some words: To be or not to be is a BIG question! Or is it?  
Ignoring those words after !, you have entered:  
To be or not to be is a BIG question  
Press any key to continue . . .
```

Ex64.cpp

A: