Chapter 5: Functions and Parameter Passing

In this chapter, we examine the difference between function calls in C and C++ and the resulting difference in the way functions are defined in the two languages. Finally, we illustrate the semantics of the three methods of passing a parameter in C++ and the three kinds of function return values. You should use the demo program and its output as reference material when you are uncertain about how to use the various parameter and return types.

> **Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it.**
> — Samuel Johnson, English, author of the Dictionary of the English Language.

5.1 Function Calls

The implied argument. In C++, every call to a class function has an "implied argument", which is the object whose name is written before the function name. The rest of the arguments are written inside the parentheses that follow the function name. Additional explicit arguments are written inside parentheses, as in C. Compare this C++ code fragment to the C code fragment below:

```
if ( in.eof() )...
stk->push(curtok);
```

The implied argument is called `this`. In contrast, in C, all arguments are written inside the parentheses. In the stack.c example, the end-of-file test and the call on `push()` are written like this:

```
if (feof( in ))...
push(&stk, curtok);
```

Number of parameters. Function calls in C++ tend to be shorter and easier to write than in C. In C, functions often have several parameters, and functions with no parameters are rare. In the `stack.c` example, note that the call on `push()` has two arguments in C and one in C++. The difference is the stack object itself, which must be passed by address in C and is the *implied argument* in C++.

Functions with empty or very short parameter lists are common in C++. First, the implied argument itself is not writtten as part of the argument list. Also, classes are used to group together related data, and that data is available for use by class functions. We often eliminate parameters by using class data members to store information that is created by one class function and used by others.

Code organization. When writing in a procedural language, many programmers think sequentially: do this first, this second, and so on. Function definitions often reflect this pattern of thought Thus, in C, functions tend to be organized by the sequence in which things should be done.

In an object-oriented language, the code is not arranged sequentially. Thus, in C++, functions are part of the class of the objects they work on. Control passes constantly into and out of classes; consecutive actions are likely to be in separate functions, in separate classes with a part-whole relationship. One object passes control to the class of one of its components, which does part of the required action and passes on the rest of the responsibillity to one of its own components.

The number and length of functions. There are many more functions in a C++ program than in a C program and C++ functions tend to be very short. Each C++ function operates on a class object. The actions it performs directly should all be related to that object and to the purpose of that class within the application. Actions related to class components are *delegated* (passed on) to the components to execute. Thus, long functions with nested control structures are unusual, while one-line functions and one-line loops are common. Inline expansion is used to improve the efficiency of this scheme.

Non-class functions. Functions can be defined inside or outside classes in C++ . For those defined outside a class, the syntax for definitions and calls is the same as in C. Non-class functions in C++ are used primarily to modularize the main program, to extend the input and output operators, and to permit use of certain C library functions.

5.2 Parameter Passing

C supports two ways to pass a parameter to a function: call-by-value and call-by-pointer. C++ supports a third parameter-passing mechanism: call-by-reference. The purpose of this section is to demonstrate how the three parameter-passing mechanisms work and to help you understand which to use, when, and why.

5.2.1 Three Odd Functions

These three functions do not compute anything sensible or useful; their purpose is to illustrate how the three parameter-passing mechanisms are similar and/or different. Each function computes the average of its two parameters, then prints the parameter values and the average. Finally, both parameters are incremented. The computation works the same way in all versions. However, the increment operator does not work uniformly: sometimes a pointer is incremented, sometimes an integer; sometimes the change is local, sometimes not.

Call by value. The argument values are copied into the parameter storage locations in the function's stack frame. There is no way for the function to change values in main's stack frame.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd1( int aa, int bb ){                 // Make copies of the argument values.
    int ansr = (aa + bb) / 2;                // See diagram 1.
    cout <<"\nIn odd1, average of " <<aa <<", " <<bb <<" = " <<ansr <<endl;
    ++aa;  ++bb;                             // Increment the two local integers.
}
```

Call by pointer. The arguments must be addresses of integer variables in the caller's stack frame. These addresses are stored in the parameter locations in the function's stack frame. They permit the function to modify its caller's memory locations.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd2( int* aa, int* bb ){               // Call by address or pointer.
    int ansr = (*aa + *bb) / 2;              // See diagram 2.
    cout << "\nIn odd2, average of " <<*aa <<", " <<*bb <<" = " <<ansr <<endl;
    ++aa;                                    // increment the local pointer
    ++*bb;                                   // increment main's integer indirectly
}
```

Call by reference. The arguments must be integer variables in the caller's stack frame. The addresses of these variables are stored in the parameter locations in the function's stack frame, permitting the function to modify its caller's memory locations.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd4( int& aa, int& bb ){               // Call by reference
    int ansr = (aa + bb) / 2;                // See diagram 3.
    cout << "\nIn odd4, average of " <<aa <<", " <<bb <<" = " <<ansr <<endl;
    ++aa;  ++bb;                             // increment two integers in main.
}
```

5.2.2 Calling The Odd Functions

This main program calls the three odd functions with arguments selected from the array named ar. It prints the whole array before and after each call so that you can see any changes that take place. Each call (and its results) is explained in the paragraphs that follow and stack diagrams are given to help you understand how the system works.

```
//-----------------------------------------------------------------------
// Calling the odd average functions.                      file: oddM.cpp
//-----------------------------------------------------------------------
#include "odds.hpp"                 // Contains the three odd functions.

void print( int* ap, int n ) {      //  Print the n values in an array.
    for( int k=0; k<n; ++k )  cout << "    [" << k << "] = " << ap[k] ;
    cout << endl;
}
//-----------------------------------------------------------------------
int main( void )
{
    int  ar[6] = {11, 22, 33, 44, 55, 66};
    int* ip = ar;       // Set a pointer to beginning of an array.
    int* iq = &ar[2];   // Set a pointer to 3rd item of array.

    cout << "\nInitially, ar is: ------------------------------------\n";
    print( ar, 6 );
    odd1( ar[0], *iq );     print( ar, 6 );     // See Figure 5.1.
    odd2( ip, iq );         print( ar, 6 );     // See Figure 5.2.
    odd2( &ar[1], &ar[4] ); print( ar, 6 );     // See Figure 5.3.
    odd4( ar[2], *(iq+2) ); print( ar, 6 );     // See Figure 5.4.
}
```

The output.

```
Initially, ar is: ------------------------------------
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55    [5] = 66

In odd1, average of 11, 33 = 22
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55    [5] = 66

In odd2, average of 11, 33 = 22
    [0] = 11    [1] = 22    [2] = 34    [3] = 44    [4] = 55    [5] = 66

In odd2, average of 22, 55 = 38
    [0] = 11    [1] = 22    [2] = 34    [3] = 44    [4] = 56    [5] = 66

In odd4, average of 34, 56 = 45
    [0] = 11    [1] = 22    [2] = 35    [3] = 44    [4] = 57    [5] = 66
```

5.2.3 Parameter Passing Mechanisms

In these diagrams, a round-cornered box represents a *stack frame* or *activation record* on the system's run-time stack. A stack frame is the data structure that is built by the run-time system to implement a function call. Within the box, the heavy dot-dashed line divides the local variables (above it) from the parameters (if any, below it). The leftmost column gives the name of the function and the return value, if any. The middle two columns supply the type, name, and address of each object allocated for the function. The rightmost column lists the current value, which is a literal number, a pointer (arrow with one stem), or a binding (arrow with a double stem).

Call by value is like making a Xerox copy. In C, there is only one way to pass a non-array argument to a function: call by value (Figure 5.1). In call by value, the value of the argument (in the caller) is shallowly copied into the parameter variable within the function's stack frame. This parameter passing method isolates the function from the caller; all references to this parameter are strictly local. . . the function can neither see nor change the caller's variables.

If the function increments the parameter variable or assigns a new value to it, the change is purely local and does not affect variables belonging to the caller. In this example, `aa` is a call-by-value parameter, so executing `++aa`, changes its value locally but does not affect `main's` variable `ar[0]`. After the function returns, `ar[0]` still contains `11`.

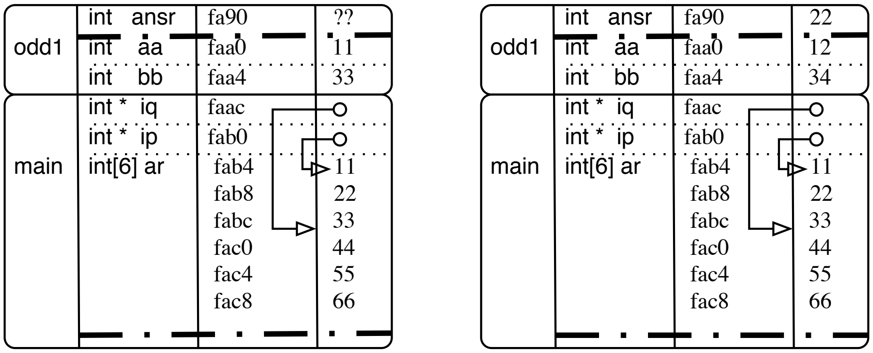

Figure 5.1: Execution of `odd1` after computing `ansr` (left) and after increments (right).

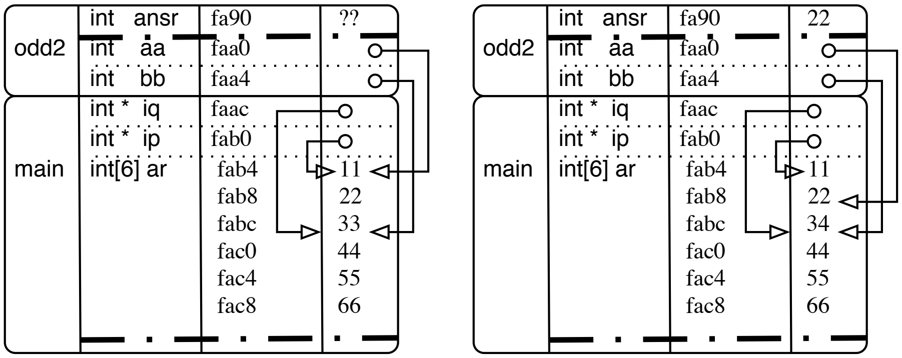

Figure 5.2: Execution of `odd2` after computing `ansr` (left) and after increments (right).

Call by pointer or address. Even in a call-by-value language, an argument that is an address or the value of a pointer can be used with a pointer parameter to give the function access to the underlying variable in the stack frame of the caller. The parameter can be used two ways within the function. By using the parameter name wiithout a `*`, the address stored locally in the parameter variable can be used or changed. By using the parameter name with a `*`, the value that is stored in the caller's variable can be used or changed.

Call by value with a pointer argument. The first call on `odd2` is diagrammed in Figure 5.2. The arguments are the values of two pointers. When these arguments are stored in the parameter locations, the parameters point at the same locations as the pointers in `main()`. Writing `++aa` increments the local value of the pointer `aa`, which changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing `++*bb` changes the integer in the caller's array slot.

Call by value with an `&` argument. The second call on `odd2` is shown in Figure 5.3. The effect is the same as the prior call; The arguments are the addresses of two slots in caller's array and are diagrammed as simple arrows. Writing `++aa` changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing `++*bb` changes the integer in the caller's array slot.

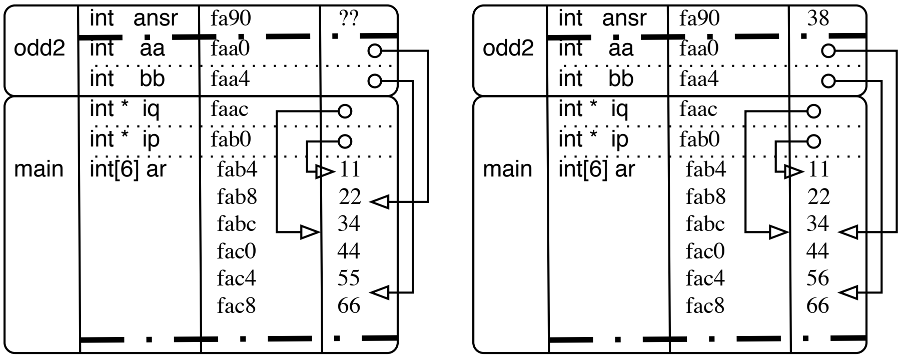

Figure 5.3: Execution of `odd2` after computing `ansr` (left) and after increments (right).

Call by reference (`&`). This form of parameter passing is normally used when the argument is or can be an object that is partly created by dynamic storage allocation.In call-by-reference (Figure 5.4), the parameter value is called a “binding” and is diagrammed as a shaded double arrow.

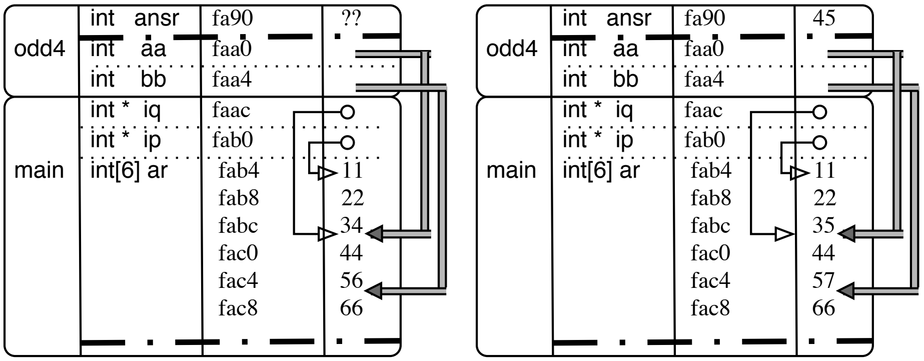

Figure 5.4: Execution of `odd4` after computing `ansr` (left) and after increments (right).

The value stored locally in the parameter cannot be changed because the parameter name becomes transparent: it is just an *alias*, or a synonym, for the caller’s variable. The program cannot read or change the address stored in the parameter. For example, line 29, using `++` changes the caller’s value, not the address stored locally in the parameter .

Call by reference vs. value. When call by value is used, the function cannot change values in the caller’s stack frame. Any reference to the parameter name is a reference to the local copy, and any assignment to the parameter changes only the local copy.

In contrast, when call by reference is used, the parameter name is an alias for the argument variable. Even though this is implemented by storing the address of the caller’s variable in the function’s stack frame, it does not function like a pointer. The dereference operator is not used because the dereference action is automatic. Any reference to or assignment to the parameter name IS a reference or assignment to the variable in the caller’s stack frame.

Call by reference vs. pointer. When call by pointer is used, the function can dereference the parameter and assign a new value to the argument variable in caller’s stack frame. Or, the function can change the address stored in the parameter and make it refer to a different variable.

In contrast, when call by reference is used, there is no choice; because the parameter name is an alias for the argument variable. It is not possible to change the address in the local stack frame. Also, syntactically, the parameter is not a pointer and cannot be dereferenced.

5.3 Function Returns

5.3.1 L-values and r-values.

An *l-value* is a value that can be used on the left side of an assignment operator; it is the location into which something will be stored. An *r-value* is a value that can only be used on the right side of an assignment operator; it is the value that will be stored. In the example below, the function `ret0()` returns an integer r-value, which we sometimes refer to as a “pure value”.

A reference (including one that was returned by a function) can be used directly as an l-value. When used as an r-value (on the right side of `=`) a reference is treated the same way that a variable name would be treated in that context: the value at that address is fetched from memory. In the example below, `ret3()` and `ret4()` return l-values.

A pointer is an r-value that can be dereferenced. If a function returns a pointer, the function’s result must be dereferenced before it can be used on the left side of `=`. In the example below, `ret1()` and `ret2()` return pointer r-values that can be dereferenced to get l-values. A dereferenced pointer can also be used as an r-value; when that is done, it will be automatically dereferenced again to get the underlying r-value.

5.3.2 Using L-values.

The following five functions all return a different flavor of int:

```
//---------------------------------------------------------------------------------
// Return by value, reference, and pointer, and how they differ.
// A. Fischer Apr 19, 1998                                        file: returns.cpp
//---------------------------------------------------------------------------------
int         ret0( int* ar ) { return  ar[0]; }    // Returns an integer R-value.
int*        ret1( int* ar ) { return &ar[1]; }    // A pointer R-value.
const int*  ret2( int* ar ) { return &ar[2]; }    // A read-only pointer R-value.
int&        ret3( int* ar ) { return  ar[3]; }    // An L-value (reference).
const int&  ret4( int* ar ) { return  ar[4]; }    // A read-only L-value reference.
```

Functions ret1 through ret4 all give the caller access to some variable selected by the function. Two of the return types provide read-only access; ret2 returns a const int* and ret4 returns a const int& . These return types are "safe", that is, they do not break encapsulation. They are often used in OO programming. In contrast, the other two return types, int* and int& can break encapsulation and give the caller a way to change a private variable belonging to the function's class. For this reason, returning non-const address and pointers is avoided whenever possible in C++.

In this example, functions ret1()...ret4() all return the address of one slot in the parameter array; the function simply selects which slot will be used. In such a situation, all these calls are "safe" and conform to OO principles, since the function is not giving the caller access to anything that is not already accessible to it. However, by using the same return mechanisms, a function could cause two serious kinds of trouble.

Violating privacy. Suppose the return value is the address of (or a pointer to) a private class variable. This breaks the protection of the class because it permits a caller to modify a class variable in an arbitrary way. Normally, non-class functions should not be permitted to change the value of class objects[1]. All changes to class members should be made by class functions that can ensure that the overall state of the class object is always valid and consistent. (The most important exception to this rule is extensions of the subscript operator.) For example, suppose we added these two functions to the stack class from Chapter 4:

```
int& how_full(){ return top; }
char* where_is_it(){ return s; }
```

A caller could destroy the integrity of a stack, St, by doing either of the following assignments:

```
St.how_full() = 0;
*St.where_is_it() = 'X';
```

Dangling pointers. A different problem would occur if a function returned the address of or a pointer to one of its local variables. At return time, the function's local variables will be deallocated, so the return-value will refer to an object that no longer exists. The resulting problems may occur immediately or may be delayed, but this is always a serious error.

5.3.3 Using Return Values in Assignment Statements

The following program calls each of the ret functions and assigns uses its return value to select one integer from ar array. This integer is saved in a variable and later printed. Four of the functions return the location of one slot in the ar array. Two of these slots are const, that is, read-only. The other two are non-const and we use them to modify the array.

```
//-----------------------------------------------------------------------
// Calling functions with the five kinds of function-return values.
// A. Fischer Apr 19, 1998                                 file: retM.cpp
//-----------------------------------------------------------------------
```

[1] There are a few exceptions, for example, when the subscript function is extended for new kinds of arrays, it returns a non-const reference because that is the entire purpose of subscript.

```
14  #include <iostream>
15  using namespace std;
16  #include "returns.cpp"
17
18  #define DUMPv(k) "\n" <<"    " #k " @ " <<&k <<"    value = " <<k
19  #define DUMPp(p) "\n" <<"    " #p " @ " <<&p <<"    value = " <<p \
20                        <<"    " #p " --> " << dec << *p
21  //-------------------------------------------------------------------------
22  void print5( int* ap ) {             // Print the five values in an array.
23      for( int k=0; k<5; ++k )  cout << "    [" << k << "] = " << ap[k] ;
24  }
25
26  //-------------------------------------------------------------------------
27  int main( void )
28  {
29      int ar[5] = {11, 22, 33, 44, 55};
30      int h, j, k, m, n;
31      int *p = ar;
32
33      cout << "Initially, variables are: ------------------------------\n";
34      print5( ar );
35      cout <<DUMPp(p);
36
37      h  =  ret0( ar );   // Answer should be 11
38      j  = *ret1( ar );   // Answer should be 22
39      k  = *ret2( ar );   // Answer should be 33
40      m  =  ret3( ar );   // Answer should be 44
41      n  =  ret4( ar );   // Answer should be 55
42      cout <<DUMPv(h) <<DUMPv(j) <<DUMPv(k) <<DUMPv(m) <<DUMPv(n) <<endl;
43
44      p = ret1( ar );     // Answer should be a pointer to ar[1].
45      *ret1( ar ) = 17;   // Storing through the pointer.
46      //*ret2( ar ) = 17; // Illegal: read-only location.
47      ret3( ar ) = -2;    // Assigning to the reference.
48      //ret4( ar ) = -2;  // Illegal: read-only location.
49
50      cout << "\nAfter return from functions variables are: -----------\n";
51      print5( ar );
52      cout <<DUMPp(p) <<endl;
53  }
```

The Dump macros. A macro lets us define a shorthand notation for any string of characters. We use macros when the same thing must be written several times, and that thing is long and error-prone to write. In this program, we want to be able to print out information that will be useful for understanding the program. We define one macro, `DUMPv` that will print the address and contents of its parameter, and a second, `DUMPp`, that prints the address of a pointer variable, the address stored in it, and the value of its referent.

A macro differs from an inline function in several ways. A very important difference is that macro parameters are not typed, and macro calls are not type checked. Thus, the macro defined here can be used to dump any object for which `<<` is defined.

The main function. We define an array, initialize it to easily-recognized values, and print it before and after calling the `ret` functions. By comparing the lines at the beginning and end of the output, you can verify that lines 45 and 46 changed the contents of the array. Calls on the `DUMP` macros in lines 35, 42, and 50 allow you to trace the effects of each function call.

The output is:

```
Initially, variables are: ------------------------------
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55
    p @ 0xbffffd40    value = 0xbffffd18    p --> 11
```

```
        h @ 0xbffffd2c    value = 11
        j @ 0xbffffd30    value = 22
        k @ 0xbffffd34    value = 33
        m @ 0xbffffd38    value = 44
        n @ 0xbffffd3c    value = 55

    After return from functions variables are: -----------
        [0] = 11    [1] = 17    [2] = 33    [3] = -2    [4] = 55
        p @ 0xbffffd40    value = 0xbffffd1c    p --> 17
```

Return by value. This is the commonest and safest kind of function return; the returned value is an r-value. (An r-value can be used on the right side of an assignment but not on the left.) Among the functions above, `ret0()` returns an integer by value. The sample program calls `ret0()` (line 37) and assigns the result to an integer variable.

Return by pointer. The return value of `ret1()` is a non-const pointer r-value (the address of ar[1]) that can be stored in a pointer variable or dereferenced to get an l-value. It is called three times:

- On line 38, the result is explicitly dereferenced to get an integer l-value, then implicitly dereferenced again to get an integer r-value, which is stored in the variable `j`.

  ```
   j = *ret1( ar );    // Copy the referent of the return value.
  ```

- On line 44, the result is implicitly dereferenced to get a pointer r-value, which is stored in the pointer variable `p`.

  ```
  p = ret1( ar );     // Store the return value in a pointer variable.
  ```

- On line 45, the result is on the left side of an assignment. It is explicitly dereferenced to get an integer l-value, which is used to store an integer constant.

  ```
  *ret1( ar ) = 17;   // Store through the pointer into its referent.
  ```

Return by const*. The return value is an address that provides read-only access to the underlying variable. The function result cannot be used on the left side of an assignment to change the underlying variable. In the sample program, `ret2()` is called once (line 39). Its result is dereferenced to get an integer l-value, which is automatically dereferenced again because it is on the right side of an assignment statement. The result is an `int` r-value, which is stored in `k`:

```
k = *ret2( ar );    // Copy referent of return value into k.
```

Return by reference. A reference can be used in the same ways that a variable name is used. The function `ret3()` returns an integer reference: an l-value that can be used (without *) on either side of an assignment operator. This function is called twice and used both as an r-value (line 40) and as an l-value (line 46):

```
m = ret3( ar );     // Coerce function result to r-value and store.
ret3( ar ) = -2;    // Store -2 in address returned by function.
```

Return by const reference. The function `ret4()` returns a const int& (a reference to a const int). This result can be only be used for read-access, because the underlying variable is a constant. `ret4()` is called once (line 41) and used with implicit dereference on the right side of an assignment statement:

```
n = ret4( ar );    // Copy referent of return value into n.
```

Illegal calls. The following lines won't compile. For both, my compiler gives the error comment *assignment of read-only location.*

```
*ret2( ar ) = -5;    // return by const *
ret4( ar ) = 25;     // return by const &
```