Passing Arguments

A comparison among programming languages

Curtis Bright

April 20, 2011

Abstract

This report describes and compares the argument passing styles used in contemporary programming languages, such as *call-by-value*, *call-by-reference*, and *call-by-name*.

1 Introduction

The concept of passing arguments to functions with parameters is ubiquitous among programming languages. During the function definition, a list of *formal parameters* may be specified and appear in the function's body as placeholders for the actual arguments supplied during a function call. However, the specific manner in which the arguments are bound to the function's parameters varies from one language to another. These specifics are important to keep in mind, as they can have a large effect on program execution.

The most common argument evaluation strategies that have been applied in programming languages are known by *call-by-value*, *call-by-reference*, and *call-by-name*. As a simple example to distinguish between them, we will consider the following pseduocode:

```
swap(a, b)
{   define temp;
    temp = a;
    a = b;
    b = temp;
}
```

Superficially, this point of this function is to switch the values of the two parameters it accepts. Whether it works as intended will depend on the evaluation strategy.

1.1 Call-by-value

In the call-by-value evaluation strategy, when a function is called the arguments passed to it are all evaluated beforehand, and copies of the values are placed

into newly allocated memory. During execution of the function body, the formal parameters now refer to their associated copied values in this new memory. At the function's conclusion this memory may simply be deallocated.

The advantages of call-by-value include its simplicity and the fact that the caller's local variables are protected from modification during the function execution. On the other hand, it may be the case that we do in fact want such changes to be seen by the caller, as in the `swap` function above. However, it is often convenient to modify the parameters during the function execution as if they were local variables, and call-by-value allows us to do this with impunity.

For disadvantages, call-by-value requires extra time and memory to initialize the argument copies; this could be significant overhead if the argument types include complex data structures. Additionally, arguments which include expressions will always be evaluated when the function is called, regardless of whether the values are actually ever used during the function. This behaviour is known as *eager evaluation* and can be needlessly inefficient.

As for our `swap` function, as it stands now it will be useless if used with call-by-value evaluation, since the switching of the parameters internally will not propagate up to the calling arguments.

1.2 Call-by-reference

In the call-by-reference evaluation strategy, when a function is called it receives the memory locations of the arguments passed to it. Therefore, during execution of the function, its formal parameters may refer to and modify local variables of the caller of the function. In this case, no extra memory need be allocated: the formal parameters of the function simply use the same memory used by the passed arguments.

Notice call-by-reference is only interesting when the argument is an *lvalue*[1], for example a variable name; an arbitrary expression does not even necessarily have a memory location that can be referred to. Thus when using call-by-reference we must either forbid general expressions or ensure that all expressions are evaluated and stored in some memory location which can then be bound to the formal parameters. In the latter case, the end result is executing as if call-by-value was used.

Thus, call-by-reference has the advantage that no extra memory is required, but only when the actual parameter is an lvalue. In addition, multiple return values can be simulated with call-by-reference by including extra input parameters and storing in them the additional return values. On the other hand, a consequence of this freedom is that extra caution must be applied when handling variables before and during a function call.

The passing method of choice for the `swap` function is call-by-reference, which gets the job done correctly with minimal overhead.

[1]The 'lvalue' term refers to something which may occur on the left of an assignment statement [6].

1.3 Call-by-name

In the call-by-name evaluation strategy, the argument expressions are directly substituted in place of the formal parameters in the function body. Of course, the scope of all variables which appear in the argument expressions must be extended to include the function body, if necessary. Additionally, any variables locally defined within the function body must be renamed if the names happen to clash with any of the variables in the argument expressions, or the intended meaning of the expression will be lost.

For example, calling `swap` on the variables `a` and `temp` leads to the following statements:

```
define temp;
temp = a;
a = temp;
temp = temp;
```

Assuming the local definition of `temp` takes precedence within the `swap` function, there is now no way to refer to the variable `temp` which we want to swap with the variable `a`, and in this case the `swap` function has no effect. To overcome this problem requires the renaming of the local variable, leading to the statements:

```
define temp';
temp' = a;
a = temp;
temp = temp';
```

And here the `swap` function still works as intended. However, in general when using call-by-name the `swap` function will not work. For example, consider calling it with `i` and `A[i]`, leading to the statements:

```
define temp;
temp = i;
i = A[i];
A[i] = temp;
```

Which leads to `A[A[i]]` being set to `i`, which is not correct when `A[i]` $\neq$ `i`. The problem in this case is that changing the value of `i` changes the memory location that `A[i]` refers to. In fact, it has been shown [4] that there can be no swap procedure utilizing call-by-name which works for all inputs.

Complications such as these seem to have limited the use of call-by-name in practice, although the strategy may be aesthetically pleasing. It can be considered a generalization of call-by-reference in that the behavior of both strategies is identical when the arguments consist only of single variables.

A potential advantage of call-by-name is that no effort is spent evaluating arguments which are never used in the function. On the other hand, it generally has to evaluate the same argument expression multiple times. This inefficiency is addressed by a variant of call-by-name known as *call-by-need*. In that optimization, each argument expression is only evaluated once, the first time it is used,

and the result is remembered for future instances. This is called *lazy evaluation* and in general this will lead to different results.

2 Language Comparison

In this section we give an overview of the evaluation strategies used in various real-world programming languages. Each subsection will examine a specific language, though we may also mention languages with similar evaluation strategies.

2.1 C

As described in [6], all function arguments are passed with call-by-value in C. In the case where `struct` objects are passed, a bit-for-bit copy is made of the entire object and this is what the called function sees; it has no way of interacting with the original object.

Then it would seem that the `swap` function is not possible to write directly in C. However, call-by-reference may be simulated in C by employing the concept of a *pointer*. This is done with the *dereference* operator `*` which 'follows' a pointer, for example if `a` is a pointer then `*a` is the actual memory location pointed to.

For simplicity, we will assume the `swap` function accepts integer arguments, though the same principle could be used for any type. Then the `swap` function in C looks like:

```
void swap(int* a, int* b)
{   int temp = *a;
    *a = *b;
    *b = temp;
}
```

This function accepts two pointers to the data to be swapped, so must not be called with the data itself but instead the *memory addresses* of the data. In C this done using the *address-of* operator `&`, so an example call would be `swap(&a, &b)`. The pointers themselves are passed by value, but this is irrelevant except that technically unnecessary copies are made of the pointers before the swapping occurs.

Finally, note that the passing of arrays in C is actually done by passing a pointer to the first element of the array, so arrays act as if passed by reference.

2.1.1 C preprocessor

It is also interesting to consider the behaviour of the function-like macros offered by the C preprocessor. They act like call-by-name except that they do not rename variables to ensure there are no clashes. That is, they perform simple textual substitution, so could perhaps be labeled *call-by-macro-expansion*.

This `swap` function looks like:

```
#define swap(a,b) {int temp = a; a = b; b = temp;}
```

With the previously mentioned caveats that it only correctly works in the case of single variables and does not work if one of the variables to be swapped happens to be named `temp`.

The typesetting system TEX is another programming language which also behaves in this manner, evaluating functions by macro expansion. In this system a `swap` macro can be written as:

```
\def\swap#1#2{\let\temp#1\let#1#2\let#2\temp}
```

2.2 C++

C++ was developed [11] to "be a better C" and address some issues with C that were seen to be deficiencies. Among them was the lack of a convenient way to pass arguments by reference; this is particularly important for passing large objects as is often done in C++.

Consequently, C++ allows one to declare *reference types*, for example `int& b = a;` defines an integer reference `b`, which simply acts as an alias for the integer `a`. Using references allows one to write a straightforward `swap` function:

```
void swap(int& a, int& b)
{   int temp = a;
    a = b;
    b = temp;
}
```

The result of the call `swap(a, b)` is the swapping of `a` and `b`.

Additionally, C++ offers a `const` keyword if one would like the performance benefits of call-by-reference but still wants assurance that the arguments are never modified as in call-by-value.

Finally, we note that several other languages allow the user to change between call-by-value and call-by-reference as required, including Pascal, PHP, and Visual Basic. In that order, we present an example `swap` function for each:

```
procedure swap(var a, b: integer);
    var
        temp : integer;
    begin
        temp := a;
        a := b;
        b := temp;
    end;

function swap(&$a, &$b)
{   $temp = $a;
    $a = $b;
    $b = $temp;
}
```

```
Function swap(a As Integer, b As Integer)
    Dim temp As Integer
    temp = a
    a = b
    b = temp
End Function
```

Visual Basic is unique in that it employs call-by-reference by default, and requires a keyword `ByVal` if call-by-value is desired. Also, Visual Basic is the only language mentioned in this section which does not throw an error when something other than a variable name is passed by reference.

2.3 Java

Java is fairly similar to C++, but does not support the pointer manipulation and reference types offered in C++ which we used in our implementation of the `swap` function. As stated in the Java specification [5], variables are passed by value.

However, consider the following function where the `IntegerWrapper` class contains a public integer variable `value`:

```
static void swap(IntegerWrapper a, IntegerWrapper b)
{   int temp = a.value;
    a.value = b.value;
    b.value = temp;
}
```

The swaps performed by this method *will* be visible to the caller, leading some to claim that Java passes objects by reference. However, this is not true as can be seen by attempting to swap objects using the straightforward implementation of `swap`.

The truth is that the definition `IntegerWrapper a` is really defining `a` to be a *pointer* to an object, but this is obscured by the Java syntax. Java does not offer a dereferencing operator like in C, but pointers are automatically dereferenced when necessary, for example the statement `a.value` in Java is equivalent to `(*a).value` when using C pointers.

Since Java is entirely call-by-value and does not offer the necessary pointer operations that could be used to simulate call-by-reference, writing a general `swap` function is not possible. For specific mutable objects, it is possible to write a function which swaps all data fields individually. For primitive types, it is possible to use an object wrapper as in the example.

The CLU Reference Manual [8] calls such behaviour *call-by-sharing*. Other languages which also employ call-by-sharing include JavaScript and Python.

2.4 Fortran

Unlike most programming languages, the default method of parameter passing in Fortran is call-by-reference. This makes the `swap` function easy to write:

```
SUBROUTINE SWAP(A, B)
    TEMP = A
    A = B
    B = TEMP
END
```

There was no option to pass by value until Fortran 2003 [9], when a `VALUE` attribute was added. However, call-by-value could sometimes be simulated by forcing the compiler make a spurious evaluation and pass a copy of the argument. For example, `CALL SWAP(A+0, (B))` does not affect the variables `A` and `B`.

2.5 ALGOL

ALGOL is an ancestor of many of the languages commonly used today. The last major specification [13] is known as ALGOL 68, and supports both call-by-value and call-by-reference. The `swap` procedure can be implemented as:

```
PROC swap = (REF INT a, b)VOID:
(   INT temp := a;
    a := b;
    b := temp
)
```

Interestingly, the ALGOL 60 specification [1] supported call-by-value and call-by-name, but the latter was dropped in favor of call-by-reference for ALGOL 68.

2.6 Haskell

Haskell employs the call-by-need variant of call-by-name, meaning that an argument expression is only evaluated at most once and not at all if it is not required. As mentioned, in general the optimization of storing and remembering the expression value will lead to different results. However, Haskell is a purely functional programming language, so generally functions do not have side effects and expressions do not change value over time, making call-by-need a safe optimization.

Another consequence of Haskell's purely functional nature is that it does not make sense to talk about implementing a `swap` function as we have been, as expressions to do not change in value.

2.7 Scheme

Scheme is another functional language, but one that allows side effects, and therefore allows an implementation of the `swap` function:

```
(define (swap a b)
    (begin (define temp a) (set! a b) (set! b temp))
)
```

In contrast to Haskell's lazy evaluation, Scheme evaluates expressions eagerly, i.e., using call-by-value. In a purely functional setting this is equivalent to call-by-reference, since there is no possibility of an argument being modified anyway. Therefore, if it is known during execution that no side effects are present the more efficient call-by-reference can be employed.

ML is another functional programming language which uses the same call-by-value behaviour as Scheme. Its `swap` function looks like:

```
fun swap(a, b) =
    let val temp = ref 0
    in  temp := !a;
        a := !b;
        b := !temp
    end;
```

Here `swap` should be called with reference types, for example:

```
val (a, b) = (ref 1, ref 2);
swap(a, b);
```

3 Summary Table

Finally, we present a summary of the languages we have considered and how they fit into the three general evaluation strategy families we have considered.

	call-by-value	call-by-reference	call-by-name
ALGOL 60	✓		✓
ALGOL 68	✓	✓	
C	✓		
C preprocessor			✓
C++	✓	✓	
Fortran 95		✓	
Fortran 2003	✓	✓	
Haskell			✓
Java	✓		
JavaScript	✓		
ML	✓		
Pascal	✓	✓	
PHP	✓	✓	
Python	✓		
Scheme	✓		
TEX			✓
Visual Basic	✓	✓	

References

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6:1–17, January 1963.

[2] A. Bloss and J. A. N. Lee. Parameter passing. In *Encyclopedia of Computer Science*, pages 1365–1367. John Wiley and Sons Ltd., Chichester, UK.

[3] R. K. Dybvig. *The Scheme Programming Language: ANSI Scheme*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996.

[4] A. C. Fleck. On the impossibility of content exchange through the by-name parameter transmission mechanism. *SIGPLAN Not.*, 11:38–41, November 1976.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[6] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[7] D. E. Knuth. *The TEXbook*. Addison-Wesley Professional, 1986.

[8] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. CLU reference manual. Technical report, Cambridge, MA, USA, 1979.

[9] J. Reid. The new features of fortran 2003. *SIGPLAN Fortran Forum*, 26:10–33, April 2007.

[10] H. Richards, Jr. Haskell: The craft of functional programming by simon thompson, addison-wesley, 1996. *J. Funct. Program.*, 8:633–637, November 1998.

[11] B. Stroustrup. An overview of C++. *SIGPLAN Not.*, 21:7–18, June 1986.

[12] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[13] A. van Wijngaarden. Revised report of the algorithmic language algol 68. *ALGOL Bull.*, pages 1–119, August 1981.