Assignment 8

Parameter Passing

All Parts Due: 11:59 PM, Monday, April 4, 2016

In this assignment, you’ll learn about a number of parameter-passing conventions.

Setup

Use Subversion to copy the hw8 in the usual way. (If you have forgotten how, see the course Wiki for details.) As usual, you may work in pairs.

Parameter Passing

Read the provided handout, Chapter 18 of Adam Brooks Webber’s Modern Programming Languages: A Practical Introduction. Complete the reading answering any questions.

Review

From the reading, you’ll recall that if we have code

```c
void f(int x, int y)
{
    int z;
    ...
}
```

and a function call

```c
f(17, a[i]);
```

then the variables `x` and `y` in the function `f` are called formal parameters, while `17` and `a[i]` are the actual arguments or actual parameters of the function call.

The reading materials discuss seven different ways that programming languages can treat the actual arguments and formal parameters.

- Call-by-value: The actual argument is evaluated before the call; the formal parameter used by the function starts with a copy of that value.

- Call-by-reference: The formal parameters are aliases for the actual arguments, so when a function modifies its parameters the actual arguments immediately change.

- Call-by-result: No information is passed to the function; when the function returns the final value of the formal parameter is copied into the actual argument (which should be a variable, array access, or other valid l-value).

- Call-by-value/result: The values of arguments are copied into the formal parameters of the function (as in call-by-value). When the function returns, the final values of these formal parameters are copied back to the caller (as in call-by-result).
• Call-by-macro: The result of the function is the result of textually copying-and-pasting the actual arguments into the function body wherever the formal parameters are used (as in C-preprocessor macros), and then running the result.

• Call-by-name: The function reevaluates each argument each and every time the function uses the corresponding formal parameter. Equivalently, like call-by-macro except that we rename local variables to avoid capturing free variables during the substitution.

• Call-by-need: Like call-by-name, but after an argument has been evaluated the first time, we save the result and re-use it later; thus, actual arguments are evaluated either 0 or 1 times.

QUESTIONS

1. In your own words, carefully explain the problems with the following argument:

   Java passes simple types like integers by value (because integer values get copied) but passes object types by reference (because when an object is passed, implicitly what is happening is that a pointer is being copied, and call-by-reference is the convention that is implemented by passing implicit pointers).

   Provide your answer in plain text in the file Answers.txt.
   Your answer should be clear and grammatical, correctly spelled, etc. as should all your answers.

2. In checkConvention.c, write a function callingConvention() that when run, can figure out which of the seven calling conventions discussed is the default for integer parameters. That is, you should write a function which returns the string "by value" if the language uses call-by-value parameter passing, returns "by reference" if integers are passed by reference, etc. You may use helper functions, global variables, etc., but your helper functions should not take anything other than integers as parameters, and may not use recursion, explicit pointers, references, the address-of (&) operator, objects, or any other advanced language features.

   See the instructions in checkConvention.c for how to compile and run it with any of the above calling conventions; you can also compile and test against all of them using the script ./checkAll. As a bonus, we support two versions of call-by-macro (ByMacro and BySadMacro, the latter is prone to more issues than the former), and call-by-brokenness in which nothing is passed in or out at all.

   You may make the following simplifying assumptions:

   • Call-by-reference, value-result, and result normally need actual arguments to be "l-values," so that they can be changed by the call. You may assume that when these conventions are active, in a call like f(2, z, g(x)) the values of the arguments 2 and g(x) are copied into temporary memory locations
before the call occurs (this is how FORTRAN 77 worked, and how C++ works when passing by-constant-reference); of course, any changes to these locations will be lost once the call completes. Thus, you are always allowed to perform a write to the parameter variable for these schemes, it’s just that in some cases the written value will be lost once the function returns.

- When the convention is call-by-result, that integer arguments will begin “uninitialized” either with some arbitrarily chosen constant, or have a truly random value. In our system, the “arbitrarily constant” can be chosen somewhat adversarially. For example, if you try passing 42, on some runs it may decide that 42 is the arbitrary constant it always initializes the variable to. But when it is using an ‘arbitrarily constant” it will stay constant between calls.¹

3. Explain carefully why the code in the previous part is correct. What is it doing?

If your code is extremely well-commented, you may not need to say much here.

(This part is worth more points than the code, see Rubric.txt for exact details.)

4. Assume we have the code

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

Depending on the arguments and the method of parameter passing, applying swap to arguments may or may not have the intended effect.

For this problem, we are concerned with the first six calling conventions, but not call-by-need. (Haskell is implemented using call-by-need, but call-by-need in a language with unrestricted side-effects is just too weird.)

(a) Assume swap is applied to two integer variables, x and y. Under which of the six calling conventions would swap(x, y) cause x and y to exchange contents?

(b) The previous part told you that the swapped variables were two distinct variables with the specific names x and y; some of the techniques may work in that situation, but you may be able to contrive a “reasonable use” of swap where they fail. Which technique(s) and why?

(c) Assume we have an integer variable i and an array of integers a. Assuming that a has a length of at least i+1, under which of the conventions would swap(i, a[i]) work as expected? For the ones that would not work, explain why.

¹In my sample solution, I handle this issue using a probabilistic strategy; it’s technically possible for my code to give the wrong answer, but overwhelmingly unlikely in practice.