Patrick Austin  
CS 326

Homework # 6

11/19/2015

1. It is possible to return the factorial of an int argument using a macro in C, like so:

/\* Assumes ints n, total, and index have already been declared. The ‘total’ value will ‘return’ the resulting factorial. \*/

#define fact(n) \

for (total = 1, index = 1; index <= n; index++) \

{ \

total \*= index; \

}

While it is not possible to implement a recursive factorial function using a macro since a macro is only a simple textual substitution lacking the underlying stack structure that makes recursion work for functions, anything that can be done with recursion can be done with iteration. An iterative algorithm for the factorial operation can demonstrably be implemented with a macro, as above.

2.

a. The local variable appears to retain its value because the second call of the function p uses the same memory for its activation record as the first function call of p. That is to say, after the relevant memory on the stack is freed when leaving the scope of the first call of p, it is immediately used again for the second call of p, so the extant “garbage” value held in y before initialization will still be 2.

b. This behavior will not take place so long as the two calls of p are interrupted by some other function call or calls that would use the same memory. That is, the value will not be retained when the memory for activation records used to store the local variable y on the first call of p is written over by some other function in the meantime, as in this example code:

void p () // unchanged

{

int y;

printf ("%d ", y);

y = 2;

}

void foo () // new function, allocates an int

{

int x = 0;

}

void main ()

{

p();

foo(); //new call to foo spliced between the two calls of p

p();

}

The resulting output is now garbage 0 instead of garbage 2.

3.

a. It is impossible to implement such a swap function using call by value, due to definition of call by value. In call by value the parameters one are given to ‘work with’ within the scope of the function are local copies of the originals, so any changes you make to them will not persist when leaving the scope of the swap function- the values will always revert to the original ones that were passed and a swap cannot be carried out.

b. It is impossible to implement a “general purpose” swap function using pass by name because of cases such as this one:

swap( a, array[a] ) //assume pass by name, a is an int, array is an array of ints

{

int temp = a; //okay so far...

a = x[a]; //okay so far…

array[a] = temp; //Problem: a has now changed, so array[a] may not be the right

//location to swap from anymore, could be out of bounds, etc.

}

The “lazy” or “as-needed” textual substitution nature of pass by name causes an issue for this type of swap by potentially wreaking havoc on the array’s index. Although eg a swap function for two ints outside of an array would certainly work, and although one could write code to specifically get around the above problem for some particular values, the kind of situations described above render the creation of a single “general purpose” swap under pass by name rules impossible.

4.

a. Pass by value: prints 1 1

The manipulation of elements that happens within the p function is irrelevant to what is ultimately printed, since pass by value rules mean that the p function is manipulating local copies of the values that were passed, not the originals. The unchanged originals are what are used for the printing instruction.

b. Pass by reference: prints 3 1

The p function is passed the address of a[0] for both parameters, the value of which upon entering p is 1. The value at that address is then incremented twice, from 1 to 2 to 3. Since this is pass by reference, the changes “stick” upon leaving the local scope of p, since one was modifying at the actual address of a[0]. The print operation then prints a[0], which now contains 3, and a[1], which was never modified by p, and so still contains the value 1.

c. Pass by value-result: ambiguous - 1 2 or 2 1

The first ambiguous issue is whether the addresses that should be copied out to at the end of a function call are evaluated at the start of the function or at the end. If the addresses are evaluated at the start, then both x and y will be set to ‘update’ a[i] where i has the value 0. If the addresses are evaluated at the end, then both x and y will be set to ‘update’ a[i] where i has the value 1.

The other ambiguity arises about whether the value of x or the value of y should be used to update a[i] upon exiting p, since they both seen to have an equal claim to ‘becoming visible’ at the same location. In this case the ambiguity does not change the outcome, since in either case above both x and y end p equaling 2.

So: either a[0] is updated to 2 (though whether this was the value of x or the value of y we do not know) and a[1] is unchanged, producing output 2 1, or else a[1] is updated to 2 (though whether this was from the value of x or the value of y we do not know) and a[0] is unchanged, producing output 1 2.

d. Pass by name: prints 2 2

The p function first increments the value at x, a[i] where i is 0 from 1 to 2. The index i is then incremented. The p function then increments the value at y, a[i] where i is 1 from 1 to 2. The values, 2 and 2, are then printed. This transpires due to the “lazy” or “as-needed” evaluation of a[i] which is characteristic of pass by name.

5. In general, no. The call for a function where the optional parameters are omitted and the call where the optional parameters are provided at their default values would be the exact same call, so no speed difference of note should be expected.