Bryan Kline

CS326

Homework 6

LaTex (TeXstudio)

11/22/2016

1. Can you write a macro in C that "returns" the factorial of an integer argument (without calling a subroutine)? Why or why not?

This cannot be done if the portion of code that calculates the factorial is recursive as each call cannot be expanded indefinitely, but it is possible to do it if it's iterative instead. As macros are simple code substitution in C, an iterative implementation of a factorial calculation can be made a macro containing variables that will be known to hold the value to calculate. The following demonstrates this:

```
...
#define x while(y > 1){ result *= y; y--; }

...
int y = 5;
x
...
```

The while loop will be substituted into and the value of y will be calculated to be 120.

2. Consider the following (erroneous) program in C:

```
void p()
{
    int y;
    printf("%d ", y);
    y = 2;
}
void main()
{
    p();
    p();
}
```

Although the local variable y is not initialized before being used, the program prints two values – the first value typically is garbage (or possibly 0, if you are executing inside a debugger or other controlled environment), but the second value might be 2 (try this on Unix!).

(a) Explain this behavior. Why does the local variable y appear to retain its value from one call to the next?

The reason for the fact that y will retain its value is that the runtime stack, when the function p is entered for the first time, has a new frame pushed onto it, space for y is made, it's printed, and then set to 2, and the frame is popped off the stack. This then happens a second time and because it's the same function it will create the same space for y which will be the same memory location so that when it's printed it will retain the previous value, 2.

(b) Explain in what circumstances (without modifying function p) the local variable y will not retain its value between calls, and show an example.

A way to prevent y from retaining it's value without modifying p is by simply adding another function and calling it in between the two calls to p so that a different frame is pushed onto the stack and so a different value is written into the memory location that p will use for y:

```
void p()
```

```
{
    int y;
    printf("%d", y);
    y = 2;
}
void q()
{
    int y;
    y = 9;
}
void main()
{
    p();
    q();
    p();
}
```

3. Consider a subroutine swap that takes two parameters and simply swaps their values. For example, after calling swap(X,Y), X should have the original value of Y and Y the original value of X. Assume that variables to be swapped can be simple or subscripted (elements of an array), and they have the same type (integer). Show that it is impossible to write such a general-purpose swap subroutine in a language with:

(a) parameter passing by value.

Passing by value is obviously a case wherein the values of X and Y will not change with the swap subroutine because pass by value is where a copy of the variable that's local to the subroutine is created and then swapped and so while inside the subroutine the values are swapped, after that scope is exited the copies and destroyed and the actual values are not swapped.

(b) parameter passing by name.

Passing by name won't always swap the values of X and Y because it's basically textual substitution, whatever is passed into the subroutine is evaluated exactly as it passed in and so in the case of mutually dependent parameters, i.e. an array which is indexed with a variable that is itself also passed in, for example:

```
...
swap(index, array[index]);
...
```

Here, instead of first evaluating `array[index]` and then passing that element into the subroutine, the actual text itself is passed in and then evaluated inside the subroutine. This prevents the swap subroutine from working because as `index` itself is also passed in it will change, and then `array[index]` will be evaluated with the changed value `index`, which will not be the element in the array that was originally passed in.

4. Consider the following program, written in no particular language. Show what the program prints in the case of parameter passing by (a) value, (b) reference, (c) value-result, and (d) name. Justify your answer. When analyzing the case of passing by value-result, you may have noticed that there are two potentially ambiguous issues – what are they?

```
int i;
int a[2];

p(int x, int y)
{
    x++;
    i++;
    y++;
}

main()
{
    a[0] = 1;
    a[1] = 1;
    i = 0;
    p(a[i],a[i]);
    print(a[0]);
    print(a[1]);
}
```

(a)

The program outputs: `1 1`

This is because it's passed by value and so the value of the actual variable, `a[0]`, is not changed in the function and it keeps its original value.

(b)

The program outputs: `3 1`

This is because it's passed by reference and so the value of the actual variable, `a[0]`, is the thing that's passed in and it is changed in the function and is incremented twice.

(c)

The may output: `1 2`
Or the program may output: `2 1`

This is because `a[0]` is passed in, two copies are made of it, one copy is incremented, then `i` is incremented, the other copy is incremented, then upon exiting the function one of those copies is saved into the actual value of `a[i]`, but `i` has been incremented so it's `a[1]` that gets the result, in the first case. However, there is some ambiguity based on whether the language uses shallow or deep binding, the first case being shallow, the referencing environment, the value of `i`, taking the most recent binding. The second case is if the language has deep binding, in which case `i` in `a[i]` upon exiting the function will keep the original value, and so `a[0]` gets the result. There is also ambiguity because as both arguments to the function are the same element of `a` and there are two copies made of it inside the function, it's not clear which copy will get stored in `a[i]` upon exiting.

(d)

The program outputs: `2 2`

This is because it's passed by name and so `a[0]` is passed in, it's incremented,

then `i` is incremented so it's now `a[1]` that's incremented, so both get incremented. They are the actual elements of the array because `a[i]` is passed in, like an expression, that is evaluated once it enters the function, so they retain the change.