University of Nevada, Reno

CS 326 — Programming Languages

Assignment #6

Joshua Gleason

Instructor:
Dr. Mircea Nicolescu

November 9, 2010

1. No, because macros in C are textural replacements done **once** before the code is compiled. This means that something like the following...

```
#define fact(a) (a <= 1 ?  1 :  a*fact(a-1))
```

   will not work because if `fact(a)` is used somewhere in the code, it is just replaced with the ternary expression which includes `fact(a-1)`. Because the substitution is only done once, this "new" `fact` will be undefined.

2. (a) The local variable `y` appears the retain its value because although the memory freed on the stack, that same memory is then re-allocated when the function is called again. Because the stack is not modified between calls, the bit values remain in tact.

   (b) If some variable makes use of the memory where `y` was allocated, then when `p()` is called again, the first value of `y` is lost. For example...

```
#include <stdio.h>

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

void q()
{
  float z = 1.0;
}

void main()
{
  p();
  q();
  p();
}
```

The output for this code is...

```
0 1065353216
```

3.
```
procedure swap(a,b)
begin
  temp := a
  a := b
  b := a
end
```

(a) If the parameter passing is by value, then swap(a,b) swaps the temporary variables inside the swap function, but these values are discarded when the function returns. So after this function executes, `a` and `b` have not changed. In fact it is impossible to change `a` and `b` during a function call, making a swap function impossible.

(b) If the parameter passing is by name, then swap(a,b) will work for non-array values. However suppose that swap(a,x[a]) is called. In this case the actual names can be replaced inside the swap function in order to simulate evaluation.

```
temp := a
a := x[a]
x[a] := temp
```

because `a` has changed, `x[a]` has also changed, so now `x[a]` could be out of bounds, or could be changing some other index. Although special cases can be written to fix this particular example. Infinitely many more examples like this one exist with nested subscripts. Therefore it is impossible to write a swap function when parameters are passed by name.

4. (a) `11`

Because pass by value does not allow parameters to be changed, nothing happens to `a[0]` and `a[1]` after during the function call to `p()`. Therefore the original value that were assigned still hold.

(b) `31`

When `p()` is called with a[0] as both parameters, a reference to that location is held by both `x` and `y`. When `x++` and `y++` are executed, both increase the value of `a[0]`. `a[1]` is not changed in this function.

(c) When `a[i]` is restored, is `a[1]` now set to a value because i was increased? In that case the output is `12` because `x` and `y` are both `2` so whichever one is returned doesn't matter because the value will be `2`. The other ambiguity is whether `a[0]` gets "restored" or `a[1]`. If `a[0]` is restored then the output is `21`.

(d) `22`

When `p()` is called here, `x` and `y` both reference the name `a[i]`. When `x++` is called i

5. When using optional parameters, if the programmer specifies the values as literals, then the function call will be the same. However if the optional parameters are filled with variables, an extra operation must be executed in order to obtain the value that the variable is holding. In the latter case, not specifying the values for the optional parameters is a bit faster.