# Homework 6

Keizou Wang

April 22, 2025

**1.** Consider the following C program:

```
void foo()
{
    int i;
    printf("%d ", i++);
}

void main()
{
    int j;
    for (j = 1; j <= 10; j++)
        foo();
}
```

Local variable i in subroutine foo is never initialized. On many systems, however, variable i appears to "remember" its value between the calls to foo, and the program will print 0 1 2 3 4 5 6 7 8 9.

($a$) Suggest an explanation for this behavior

i is declared without a default value every time so the value is determined by what's already at the address. In this scenario, it seems whenever foo is ran, i takes the same address and the same value is incremented.

($b$) Change the code above (without modifying function foo) to alter this behavior.

```
void main()
{
    int j, i;
    for (j = 1; j <= 10; j++){
        printf("%d ", i);
        foo();
    }
}
```

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

It would be impossible because recursion is impossible with just textual substitution and a loop solution can't directly "return" a value.

**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:

($a$) Parameters passing by value.

It's impossible when passing by value because the programmer only has access to the values of X and Y and not the addresses of the variables. Therefore, the programmer can't set the value at the address of X to the value of Y and vice versa.

($b$) Parameters passing by name.

Consider the case `swap(i, a[i])`. After changing the value of `i`, `a[i]` references a different address so the behavior becomes unpredictable.

**4.** Consider the following program, written in no particular language.

```
procedure f (x, y, z)
        x := x + 1
        y := z
        z := z + 1

// main
i := 1;
a[1] := 10;
a[2] := 11
f (i, a[i], i);
print (i);
print (a[1]);
print (a[2]);
```

Show what the program prints in the case of parameter passing by

($a$) value

```
1 10 11
```

($b$) reference

```
3 2 11
```

($c$) value-result

If the value of `i` is updated to the result before `a[i]` is updated:

```
2 10 1
```

If the value of `i` is updated to the result after `a[i]` is updated:

```
2 1 11
```

Another source of ambiguity with value-result, though not present here, is if two parameter values evaulate to different results. For example, if x and z ended up at different values, which result would i take?

($d$) name

```
3 10 2
```

**5.** Does a program run faster when the programmer does not specify values for the optional parameters in a subroutine call?

No because the default value is set to the parameter in runtime so it's equivalent to setting the value within the function normally.