Tutorial Sheet (Week 11)

ESC101 – Fundamentals of Computing

Announcement

1. There was no tutorial in week 11 of the course.
2. **Major quiz**: October 31st 12 – 1 PM, L20
3. **End-sem Lab Exam**: November 04th
    a. Morning exam (Mon, Tue batches) 10:30 AM - 2 PM
    b. Afternoon exam (Wed, Thu batches) 2:30 PM - 6 PM
    c. See lecture slides for room number assignment

The Six Golden Rules of Functions

1. **RULE 1**: When we give a variable as input, the value stored inside that variable gets passed as an argument. For pointer variables, the address stored inside gets passed as an argument.
2. **RULE 2** If we give an expression as input, the value generated by that expression gets passed as argument. If that value is an address (e.g. the expression may be &a) the address is passed.
3. **RULE 3** (the type-mismatch rule): In case of a mismatch b/w type of argument promised and type that is passed, typecasting will be attempted. However, this may cause errors. For example, promising a pointer to char and then passing a pointer to an int or a pointer to a pointer may not give any compilation errors.
4. **RULE 4** (the copy rule): All values passed to a function get copied and stored in a fresh variable inside that function. Modifying the copy does not modify the original variable.
5. **RULE 5** (the return rule): Values returned by a function can be used freely in any way values of that data-type could have been used. However, make sure that the value suits the operation you are performing.

    a. If you are indexing an array with an int returned by a function, verify that integer is not negative or out of bounds.
    b. If taking square root of a float returned by a function, verify that the float is not negative.
6. **RULE 6** (the address rule): Even though the clones may have their own variable names without interfering, they use the same memory address space. If one clone modifies an element at a certain memory location directly, all clones will see that change.

Returning multiple values from a function

1. **METHOD 1**: return an array from the function. Rule 5 of pointers. Array name is simply pointer to its first element. To return an array, return address of its first element.
    a. Advantage: return as many values you want
    b. Disadvantage: all values must be of same type
    c. Disadvantage: can only return one array
2. **METHOD 2**: Pass-by-reference trick – give the function the address of a variable and ask the function to modify the variable at that memory location. Since all clones share the same memory address space, any changes will get reflected.
    a. Advantage: return as many values you want and that too of different datatypes
    b. Advantage: can return multiple arrays as well. Array names are pointers anyway so nothing to be done. Just pass the array to the function and it can modify the array itself.
    c. Disadvantage: Be careful with pointers
    d. Disadvantage: can only return one array
3. **METHOD 3**: Return a structure
    a. Advantage: no hassle of pointers
    b. Disadvantage: have to define a structure
    c. Disadvantage: can only return one structure (unless we are returning an array of structures).

Passing 2D arrays as arguments to functions

1. **Case 1. The 2D array has a fixed number of columns**: suppose it is promised that the 2D array will always have 7 columns. In this case simply declare the function as `void foo(int mat[][7]){ … }`. Suppose we have a 3 x 7 integer 2D array `int arr[3][7]`. We can pass it to the function simply as `foo(arr)`.
   It does not matter if number of rows is known or unknown.
2. **Case 2. The 2D array is actually an array of arrays**: in this case it does not matter whether number of rows/columns is known or not. We can declare the function as `void foo(int **mat){ … }` and call it as `foo(arr)`. Note that foo must have been malloced.
3. **Case 3. The 2D array is neither an array or arrays nor does it have fixed number of columns**: in this case passing this 2D array directly is problematic since Mr C has no way of knowing how many elements are there in the first row, in order to access the second row (in arrays of arrays, there is a separate pointer to first element of every row so this problem does not arise).
   Solution: treat the 2D array as a 1D array and index it yourself. This works since a 2D array is stored internally as a 1D array. See code provided with lecture slides for an example.

Some pitfalls and recognizing compiler error messages

1. Do not statically declare an array inside a function and return it. These get destroyed when the function returns. If you want to declare an array inside a function and return it, you must malloc/calloc/realloc this array.
2. No matter whether we are passing a pointer, an address generated by an expression or a normal variable or value, **everything passed gets copied**. Modifying the copy inside the function does nothing to the original variable.

3. When we pass as argument, a normal variable like a char or a float to a function, or an expression generating a normal value like int or double, it is often called *pass-by-value*.
4. When we pass as argument, an expression that generates an address, it is often called *pass-by-reference*. Note that the reference rule of pointers applies here.
5. When we pass a pointer variable as an argument, it is often called *pass-by-pointer*.
6. If we pass an array to a function, the sizeof operator applied to that array inside that function will just give answer 8 and not the actual size of the array since when an array is passed, only a pointer to its first element is passed.