Introduction to Computing

MCS1101B

Lecture 7-8

By
Soumadip Biswas
Associate Professor, IEM

Recap

- Array
  - Declaration
  - Initialization
  - Assignment
  - Accessing elements of array
- Pointers
  - Another type of variable
  - Can hold memory address of some variable
  - The scanf case

- Some example codes using array
  - Print all elements of array
  - Scan elements into array
  - Find the minimum from array
  - Search for a key element in an array

Array and Functions - Refresher

- Array
  - int arr[8] = {12, 14, 1, -2, 6, 91, 200, 10}
  - Print all elements of an array in reverse order
  - Print elements of an array within a given range e.g. 2-6
  - Print all elements of an array that are positive
  - Print all elements of an array that are even

- Function
  - <ret_type> <name> (param1, param2, …)
  - Write a function for $f(x) = x^2 + 10$
  - Write a function for $f(x, y) = (x + y)^2$
  - Write a function for f(n) = n!
  - A function that returns the mean of all elements in an array of integers

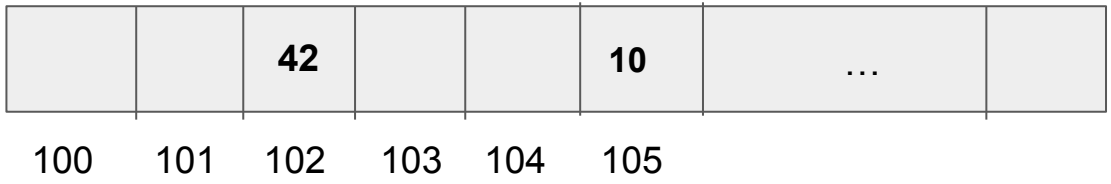
Pointers (recap)

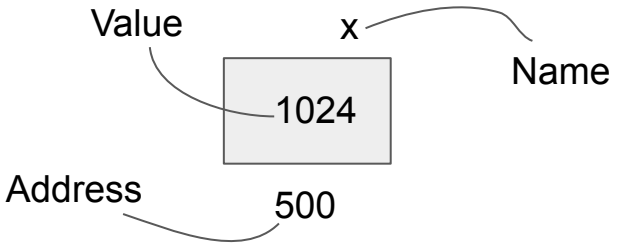
- Pointers are a special variables that can store memory locations
- Declaration of a pointer variable
  - <type> *<name>;
  - Variable value can be accessed using <name>
- Access the value at the stored address
  - *<name>
  - It will treat the value at the stored location as the declared <type>

- int a=10; int *ptr; //this is an integer type pointer
- printf (“%d”, a); ⇒ 10
- printf (“%d”, ptr); ⇒ <some garbage value>
- printf (“%p”, ptr); ⇒ <the same garbage value in the form of an memory address>
- printf (“%p”, &ptr); ⇒ the address of the variable ptr
- printf (“%p”, &a); ⇒ memory address of the variable a
- ptr = &a; //stores the address of a on ptr
- printf (“%p”, ptr); ⇒ the address of the variable a
- printf (“%d”, *ptr); ⇒ value of the integer at the location of the variable a
- printf (“%p”, &ptr); ⇒ the address of the variable ptr; remains the same

Pointers

- Each memory cell (byte) has an unique address
- Each memory cell can hold a value
- Contiguous memory cells have sequential addresses





Pointers: Size and Arithmetic

- Size of a pointer variable
  - It depends on the maximum possible number value for address in a machine
  - A 64-bit processor allows the machine to have 64 bit address - so it needs 8 bytes to store that address
  - sizeof(int*) ⇒ ?
  - sizeof(char*) ⇒ ?

Never *add floating points to pointers*, **Never** *add two pointer variables,* ***Never*** *assign* a direct value to a pointer variable – these are meaningless

In general, *don't use pointer arithmetic in coding unless there is no other way*

- So what does it mean to add some value to a pointer value?
  - You can't directly add value since address is constant
  - You can however, add to a pointer variable
  - ***<type>* ptr + <int_val>*** is equivalent to ***ptr + <int_val> * sizeof(<type>)***

Example:
int* **p**, **x**;
p = **&x**; // say, address of x is 100 and sizeof(int) = 4
p+2 ⇒ 100 + 2 * **sizeof(int)** = 108
p+1 = ? **p**+10 = ? **p**-3 = ? **p**-30 = ?

Array and Pointers

- Array elements are accessed using indexes
  - int arr[10];
    - Allocates a memory block equal to the size of 10 integers in total
    - Elements accessed as arr[0], arr[1], etc.
  - The **arr** is the address of the entire memory block; it is of type int* (read as *integer pointer*)
  - Therefore It can also be accessed similar to pointers variables
  - So *arr is arr[0]
    - How do you access the rest? → you use *pointer arithmetic*
  - Adding 1 to a pointer variable means increasing the value of the pointer by the size of the type of that pointer
    - adding 1 to an int* **variable** means adding sizeof(int) to the value of the **variable**
  - So, arr[1] == *(arr+1), arr[2] == *(arr+2), etc., i.e., arr[i] = *(arr+i)
  - Also, arr+i = &arr[i]

Functions and Pointers

- Since variables passed to the functions are basically a copy
- Pointers to the variables are used instead of a variable to pass the **reference** to a variable - only when required
  - Addresses of the variable is copied
  - Changes made by function are done to the memory address
  - So when function exits, it only forgets the memory location and not the changes made to that location

So, recall Swap

```
void swap (int a, int b)
{
        int tmp;
        tmp = a;
        a = b;
        b = tmp;
}
```

```
void swap (int *a, int *b)
{
        int tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
}
```

Passing Array to Functions

- Array is a memory block
- Array variable is basically the first address of the entire memory block
- Array type is how you access each element in the memory block
- The size of the block is known only to the function the array is defined in
- If you pass array to a function, *only the address of the memory block is copied, and nothing else*

Example:

- int A [10];
- sizeof (A) ⇒ sizeof (int)*10
- Call f(A)

A function

- void f (int arr[])
  {
  sizeof (arr) ⇒ sizeof (int*)
  }

Passing Array to Functions - Alternative way

Assume
sizeof(int) = 4 and *sizeof (int*) = 8*

- int A [10];
- sizeof (A) ⇒ 4*10 = 40
- Call f(A)
- Call f2(A)

A function

- void f (int arr[])
  {
  sizeof (arr) ⇒ sizeof (int*) = 8
  }

Another function

- void f2 (int *arr)
  {
  sizeof (arr) ⇒ ?
  }

So, to pass an array properly you need to pass the size (desired) of the array as well.

- void f (int arr[], int n)
- void f2 (int *arr, int n)

There is an exception to this rule for char array – we will discuss that later

Functions Calling Functions

- int f1() {...}
- int f2()
  {...
  f1();
  …}
- int f3() {... f2(); …}

- int f4() {... f3(); …}
- int f5() {... f2(); …}

- int f6() {... f7(); …}
- int f7() {... f6(); …}

- int f8() {... f8(); …}

These are basically never ending calls to one another

→can this happen?

Recursion

- A function calling itself
  - Directly call made to self
  - Indirectly call made to self via another function
  - Indirectly call made to self via a sequence of function calls
- This is known as recursion
  - Both in mathematics and in programming

- Example (math)
  - $f(n) = n*f(n-1), f(0)=1$
  - $f(n) = f(n-1) + f(n-2), f(0)=0, f(1)=1$
    $\rightarrow$ what function is this?
  - $f(x) = x * g(x), g(x) = 2 + f(x-1)$

    $\Rightarrow f(x) = 2 * x + 2 * f(x-1)$

Recursion (contd.)

- Requires careful coding
- Needs to make sure that your program terminates
- You need to first define the base cases (exit condition) for your function
- Then you write the logic of the rest of the function
- For breaking the call sequence of a recursive function
  - a **return** statement is generally used with some if condition
  - You can also use if-else
- Exercise:
  - Implement the factorial function using recursion
  - Implement the gcd function using recursion

Character Arrays or Strings

- Character arrays (aka Strings) are very useful in storing data
  - Even though they are basically integers underlying, but the range of the values are limited
  - This allows to have some additional functionalities (for convenience, of course)
- Strings are declared and defined the same way as any other array types
  - Since the values are in range of 0-127 (sometimes more, but still, limited), we have the convenience make some of the characters for special use such as:
    - newline(\n)
    - backspace (\b), etc.
  - In the case of character arrays we use a special character called the null character
    - Represented as ‘\0’ (backslash-zero)
    - Ascii value of this character is 0
    - It prints nothing on the computer screen

Strings - Initialization

- char ch = ‘a’;
- char ch_arr[10] = {‘S’, ‘o’, ‘u’, ‘m’, ‘a’, ‘d’, ‘i’, ‘p’, ‘\0’};
- char name[10] = “Soumadip”; //the above one is equivalent
  - This type of initialization makes sure that the null character is appended at the end
- String is basically short for “a string of characters”
  - A single character in C is written within single quotes e.g. ‘a’, ‘3’, ‘Z’, ‘%’, etc.
  - A string is written in C within double quotes e.g. “a_string”, “with spaces”, “and with $”, etc.
- Scanf also provides a shortcut for strings format **%s**
  - scanf (“%s”, ch_arr); ⇒ this allows you to read a string from user (without spaces)
  - scanf ("%[^]%*c", ch_arr); ⇐ %s is equivalent to this, is a blank space
    - This tells scanf to read characters as long as a space ( ) is not encountered
  - Similarly, scanf ("%[^\n]%*c", ch_arr); ⇐ reads a string with spaces until a newline(\n)
  - *Note that, a newline character will always end scanning for scanf irrespective of type*

Next Week…

- You will have Midsem :-)
- All the best
- Prepare well.