Introduction to Computing

Recursion, Basics of Strings

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

Pointers (recap)

- <type> *<name>; ⇒ *declaration syntax of pointer variable*
- Pointer variable value can be accessed using <name>
- Access the value at the stored address using *<name> ⇒ *treat the value at the stored location as the declared <type>*
- Access the memory address of the pointer variable using &<name>

```
int a=10;        int *ptr;

printf (“%d”, a);       ⇒ 10
printf (“%p”, &a);      ⇒ address of a

ptr = &a;

printf (“%p”, ptr);     ⇒ ?
printf (“%d”, *ptr);    ⇒ ?
printf (“%p”, &ptr);    ⇒ ?
```

Functions and Pointers (refresher)

- 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 ot that location

So, Let’s 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;
}
```

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

Functions

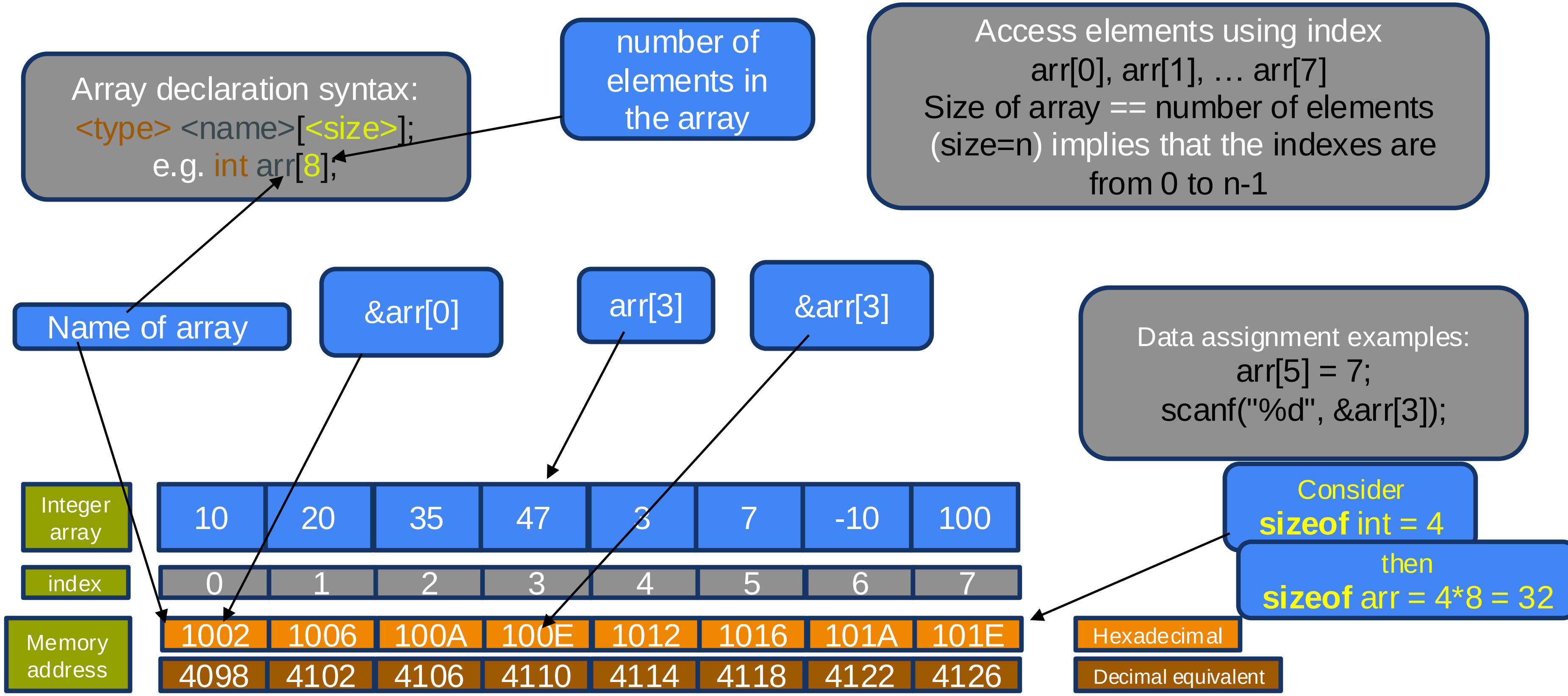
<ret_type> <name> (param1, param2, …)

Write a function for the following function definitions

- $f(x) = x^2 + 10$
- $g(x, y) = (x + y)^2$
- factorial(n) = n!
- permutations(n,r) = nPr
- combinations(n,r) = nCr

- A function that returns the mean of all elements in an array of integers

Array (contd.)



Array declaration syntax:
<type> <name>[<size>];
e.g. int arr[8];

number of elements in the array

Access elements using index
arr[0], arr[1], … arr[7]
Size of array == number of elements
(size=n) implies that the indexes are from 0 to n-1

Data assignment examples:
arr[5] = 7;
scanf("%d", &arr[3]);

Consider
sizeof int = 4
then
sizeof arr = 4*8 = 32

Integer array	10	20	35	47	3	7	-10	100
index	0	1	2	3	4	5	6	7
Memory address	1002	1006	100A	100E	1012	1016	101A	101E
	4098	4102	4106	4110	4114	4118	4122	4126

Hexadecimal

Decimal equivalent

Passing Array to Functions

- Array is a memory block
- Array variable is basically the first address of the entire 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)    ⇒ 10 * sizeof (int)
  Call func (A)
```

In the function func

```
  void func (int arr[])
       {
              sizeof (arr)    ⇒  sizeof (int*)
       }
```

Passing Array to Functions – Two ways

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

```
int A [10];
sizeof (A) ⇒ 4*10 = 40
```

- Call func1(A)
- Call func2(A)

One way

- void func1 (int arr[])

```
{
    sizeof (arr) ⇒ sizeof (int*)=8
}
```

Another way

- void func2 (int *arr)

```
{
    sizeof (arr) ⇒ Also sizeof (int*)=8
}
```

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 shortly*

Functions Calling Functions (type 1)

- int f1() {...}
- int f2()

  {...

        f1();

  …}
- int f3() {... f2(); …}

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

```
int factorial (n)
{
    int i, result = 1;
    for (i=1; i<=n; i++)
        result *= i;
    return result;
}

permutations(n,r) = nPr  // Can be written as follows:
        ⇒  factorial(n)/factorial(n-r)

combinations(n,r) = nCr
          ⇒ ?
```

Functions Calling Functions (type 2)

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

int f8() {... f8(); …}
```

These are basically never ending calls to one another

→ can this happen?

Factorial definition (from math)

```
f(n) = n*f(n-1)   //recursion
f(0)=1            //base case
```

```
int factorial (n)
{
    if (n==0)        //base case
        return 1;
    else
        return n* factorial(n-1);   //recursion
}
```

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

- Examples

  power(n, a) = n*power(n,a-1)
  power(0)=1

  f(n) = f(n-1) + f(n-2)
  f(0)=0, f(1)=1
  → what function is this?

  f(x) = x * g(x)
  g(x) = 2 + f(x-1)

  ⇒ f(x) = x * 2 + x * f(x-1)

Recursion (contd.)

Recursive solution template

- You need to first define the base cases (exit condition) for your function
- Then you write the recursive 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

- Requires careful coding
- Needs to make sure that your program terminates
- DIY Exercise using recursion:
  - Implement the GCD function
  - Implement the power function
  - Implement sum of an integer array
  - Search an element in an array
  - Count the number of vowels in a character array/string

Characters and ASCII codes

- Recall computer can only store numbers
- Characters are interpreted as integers numbers called ASCII code
- These codes are stored in place of each character
    - a-z, A-Z, 0-9,special characters (!, @, #, $, ...), \n, \b, \r, \t, etc.
    - The standard ASCII code ranges from 0 to 127 (7 bits long)
    - The extended ASCII code ranges from 128 to 255 (8 bits long)

```
// use for loop to print the capital letter from A to Z
   for (int  code = 65; code< 91; code++)
   {
      printf (" \n The ASCII value of %c is %d ", code, code);
   }
```

Outputs:

```
The ASCII value of A is 65
 The ASCII value of B is 66
 The ASCII value of C is 67
 The ASCII value of D is 68
 ...
The ASCII value of Z is 90
```

```
Doing the same for small letters, another way
for (int letter = 'a'; letter<= 'z'; letter++)
   {
      printf (" \n The ASCII value of %c is %d ", letter, letter);
   }
```

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 ("%[^\n^]%*c", ch_arr); ⇐ %s is equivalent to this,   is a blank space
        - ^This tells scanf to read characters as long as a newline (\n) or a space ( ) is not encountered
    - Similarly, scanf ("%[^\n]%*c", ch_arr); ⇐ reads a string with spaces until a newline(\n)

Next Week…

- More on strings
- User defined datatypes