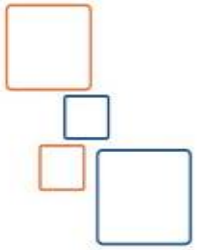
C Programming Language

Introduction to Programming Using C



Invest In Yourself,
Develop Your Career

Course Duration and Evaluation

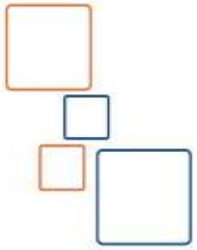
- Duration: 42 hours
  - 7 Lectures (42 hours)
  - 7 Labs (42 hours)

- Evaluation Criteria:
  - 40% on labs activities and assignments
  - 60% on written exam after 7 days of the last lecture.

Course Content

- D1: Introduction to Programming
- D2: Operators and Control Statements
- D3: Arrays and Strings
- D4: Structures and array of structures
- D5: Modularity In C: Functions
- D6: Pointer Datatype
- D7: Dynamic Allocation and Miscellaneous topics in C

Introduction to Programming



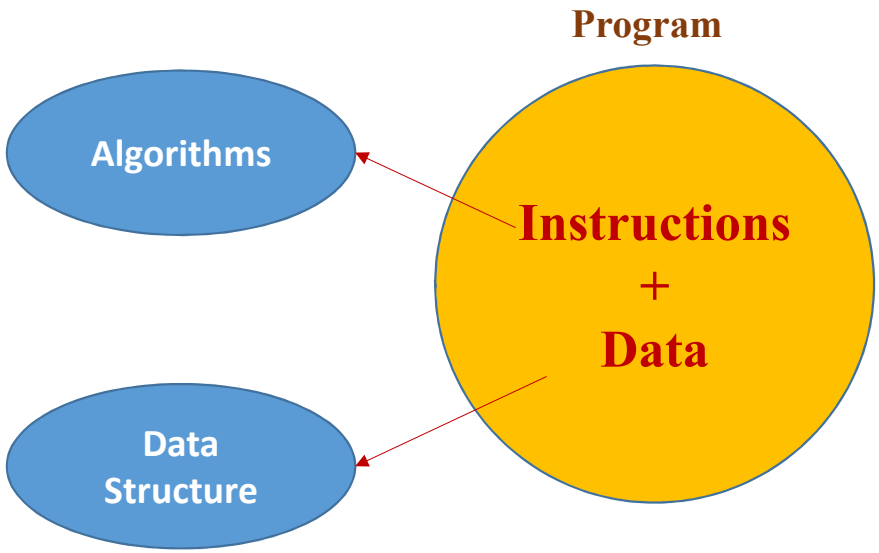
Invest In Yourself,
Develop Your Career

Content

- Introduction
- Programming Languages Levels
- Programming Techniques History
- How Do Programming Languages Work?
- C Program Structure
- Some of basics in C Program

1.1. Introduction

- Why we need a computer program?
  - To perform a desired task; Problem Solving
- What is a computer program includes?
  - Instructions
  - Data



1.2. Programming Languages Levels

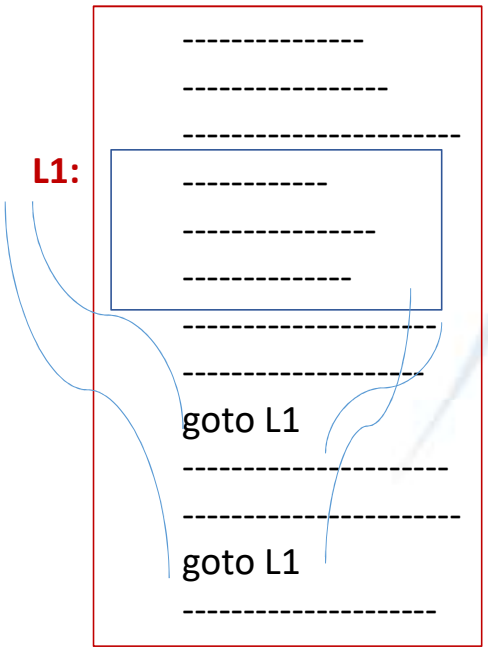
- Low-Level Languages
  - Fundamentals language for computer processors,
  - 0's and 1's that represent high and low electrical voltage (Machine Language),
  - Modified to use symbolic operation code to represent the machine operation code (Assembly Language)
- High-Level Languages
  - Use English like statements, executed by operating system (in most cases)
  - C, C++, Pascal, Java, ….
- Very High-Level Languages
  - Usually *domain-specific* languages, limited to a very specific application, purpose, or type of task, and they are often scripting languages
  - PLAN, Prolog, LISP

1.3. Programming Techniques History

- Linear Programming Languages
  - BASIC
- Structured Programming Languages
  - C, Pascal
- Object-Oriented Programming Languages
  - C++, Java, C#

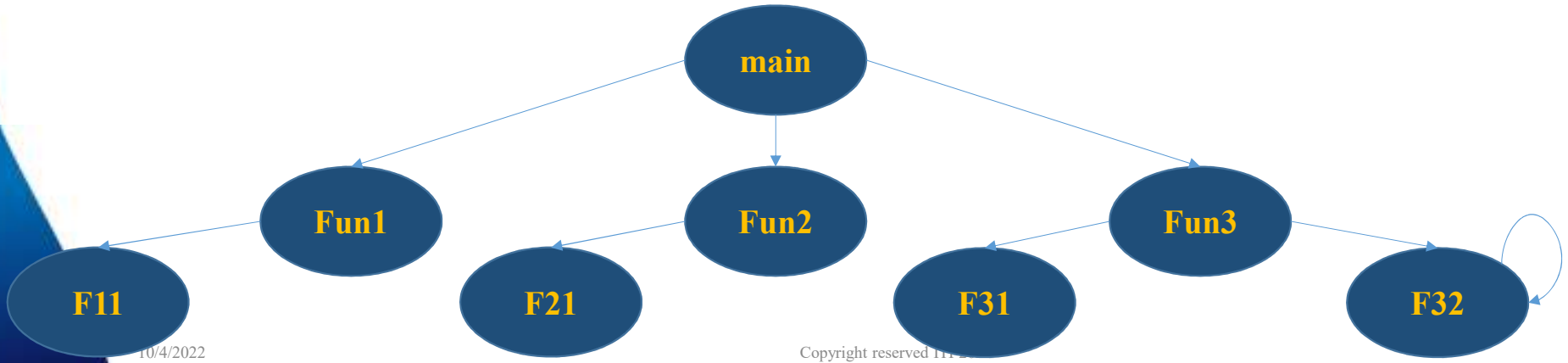
1.3.1 Linear Programming Languages

- A program is executed in a sequential manner.
- The program size is exponential increasing by increasing the functionality of the program.
- GOTO statement is used …!!!



1.3.2 Structured Programming Languages

- Some times called Function-Oriented Programming
- Similar like subroutine in assembly language
- Subroutine equivalent Function equivalent Procedure ; Sub-program
- Any program consists of functions, at least “main” function
- The main function is the entry point of the program



1.3.3 Object-Oriented Programming Languages

- Programming model that organizes software design around data, or objects, rather than functions.
- An object can be defined as a data field that has unique attributes and behavior.
- A Class is the template of objects that describe the same data but each one has different values of attributes.
- Reusability of the classes is one of the advantages of this technique.
- Details will be clear in Object-Oriented Programming using C++ course …

1.4. How Do Programming Languages Work?

- **Interpreted Languages**
  - It depends on an interpreter program that reads the source code and translates it on the fly into computations and system calls- line by line.
  - The source has to be re-interpreted (and the interpreter present) each time the code is executed. **BASIC** and most of scripting languages (Ex: **HTML**)
- **Compiled Languages**
  - Compiled languages get translated into executable file, no need to recompile again –if there is no changes – run the executable directly. **C, C++, Pascal**
- **Compiled & Interpreted Languages**
  - Compiler translate to intermediate code and need the interpreter to run that intermediate code. **Java**

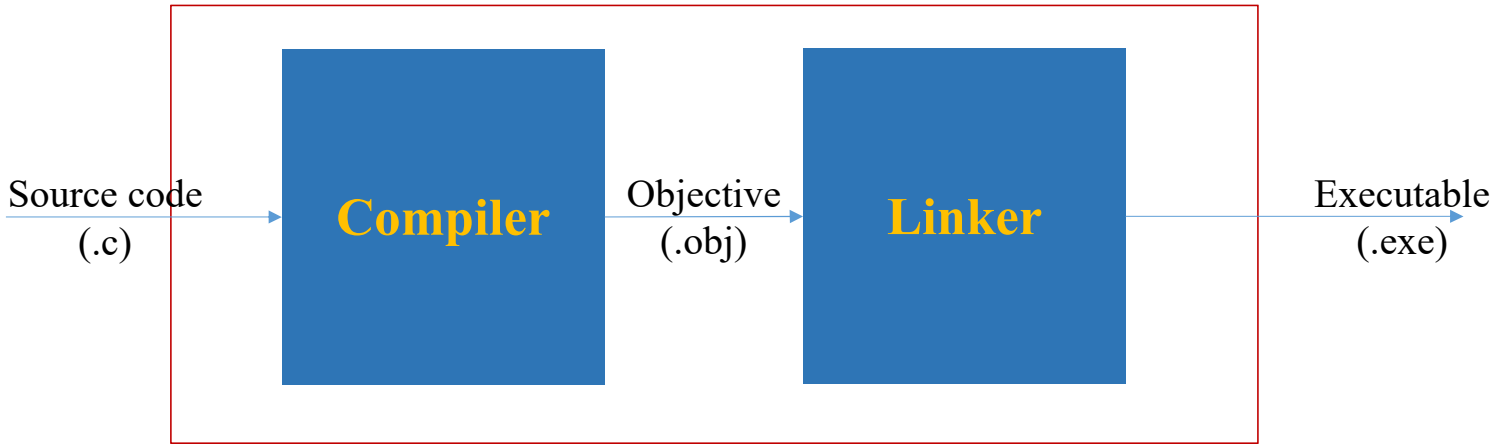
1.5. C Program Structure

- **Part I: Preprocessing Part**
  - Include Libraries
  - Define and Macro (Text Replacement)
  - Structures definitions
  - Global Variables
  - User-defined Functions Prototypes
- **Part II: The Main Entry Point : main Function**
- **Part III: User-defined Functions**

1.5.1.1 Include Libraries

- **#include**
- For using a C function in any program you have to include its library before using. Why it is in the preprocessing part?
- The compiler needs to check if you call the function in a right way or not: check on the name and the input parameters – numbers and types.
- Each library consists of C (.c)file and Header File (.h), it is prefer to include the header file not the C file. Why?
- The compiler check and translate the source code (.c) to executable instructions – system calls- except the predefined functions in libraries and generate intermediate file called Objective file (.obj).
- The linker program – as a part of compiler environment – use the libraries –which included in the program - to complete the Objective file (.obj) to be an executable file. *How?*

1.5.1.1 Include Libraries



- You can include just one library using one include statement
- Must be the first of the file
- Has two forms:
  - #include <stdio.h>
  - #include “d:\\myNewLib\\extern.h”

1.5.1.2 Define and Macro

- **#define**
- It is used to define a replacement text with another after the statement directly till the end of the file before starting the compilation.
- Ex:
  - #define PI 3.14
  - ----
  - printf (“%f”, PI); // the replacement done before start compilation the o/p is “3.14”
- Ex:
  - #define one 1
  - #define two one+one
  - #define four two*two

  - printf (“%d”, four); // what will the output?

1.5.1.2 Define and Macro

- Macro is another type of text replacement but with using the function operator ().
- Ex:
  - #define SUM(X , Y) X+Y
  - -------
  - -------
  - printf("%d", SUM(3, 8)); // o/p will be 11
  - printf("%d", SUM(-4, 3)); // o/p will be -1

1.5.1.3 Structures definitions

- Structure mean record with number of fields which they are non-homogeneous in most cases
- It will be clear in the 4th lecture.

1.5.1.4 Global Variables

- The scope of them is global; these variables are accessible by all the functions of the program.
- It is not preferable to use it without strong reasons; use it if and only if you have to use it. Why?
- They are saved in a part of the memory sections allocated to the program, this part is called: ***Heap Memory***

1.5.1.5 User-defined Functions Prototypes

- It is the header only of the user-defined function in the program.
- The compiler use it to check if you call these function in a right way using its prototype – the same concept of using the header files of the libraries-
- **Note**: you may not use this part but you have to rearrange the sequence of user-defined function and so the main function; the called function must be written before the caller function.

1.5.2 Part II: The Main Entry Point: main function

- At the first use the following main function header:

```
void main (void)        // The function header
{                       // Start of the function body


}                        // End of the function body
```

- First void indicate there is no return data type for the caller of the main function. Whom?
- The void between braces indicate there is no input parameters will be sent to the main function when it is called.

1.5.3 Part III: User-defined Functions

- It is the part to write the structure of functions you designed it to your program
- It is the full functions; Header and body for each one

1.6. Some of basics in C Program

- Case sensitive.
- Braces, and blocks: {}
- Delimiters after each statements – except include and define: ;
- Basic input and output functions:
  - printf
  - scanf
  - getch or getchar
  - clrscr or system("cls") ---------> compiler dependent

1.6. Some of basics in C Program

- Primitive Data Types in C:

• char	size	1	byte	from	0	to	$2^8 - 1$	unsigned
• int	size	2	bytes	from	-2^{15}	to	$2^{15}-1$	signed
• long	size	4	bytes	from	-2^{31}	to	$2^{31}-1$	signed
• float*	size	4	bytes	from	$-1.2*10^{38}$	to	$3.4*10^{38}$	signed
• double*	size	8	bytes	from	$-2.3*10^{308}$	to	$1.7*10^{308}$	signed

- Complex Data Types in C:
  - Arrays
  - Structure

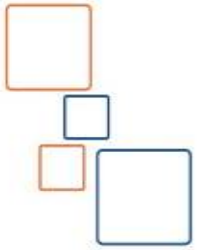
* The ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic ("the IEEE standard" or "IEEE 754“)

1.7. Welcome To ITI World Program

• First program using C:

```
#include <stdio.h>
#include <conio.h>
void main (void)
{
    clrscr();  // or use system("cls");(compiler dependent)
    printf("Welcome to ITI World");
    getch();
}
```

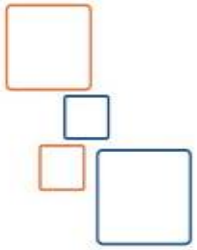
Lab Exercise



Assignments

- Install the C environment
- Write a C program to test the different format specifiers with “printf “
- Write a C program to read a character from the user and print it and its ASCII code.
- Write a C program to display the octal and the hexadecimal representation of an integer number.

Operators and Control Statements



Invest In Yourself,
Develop Your Career

Content

- Operators in C
- Control Statements in C
  - Branching Statements
  - Looping Statements
- Break Statement
- Continue Statement
- Comments in C
- Introduction to Magic Box Assignment

2.1 Operators in C

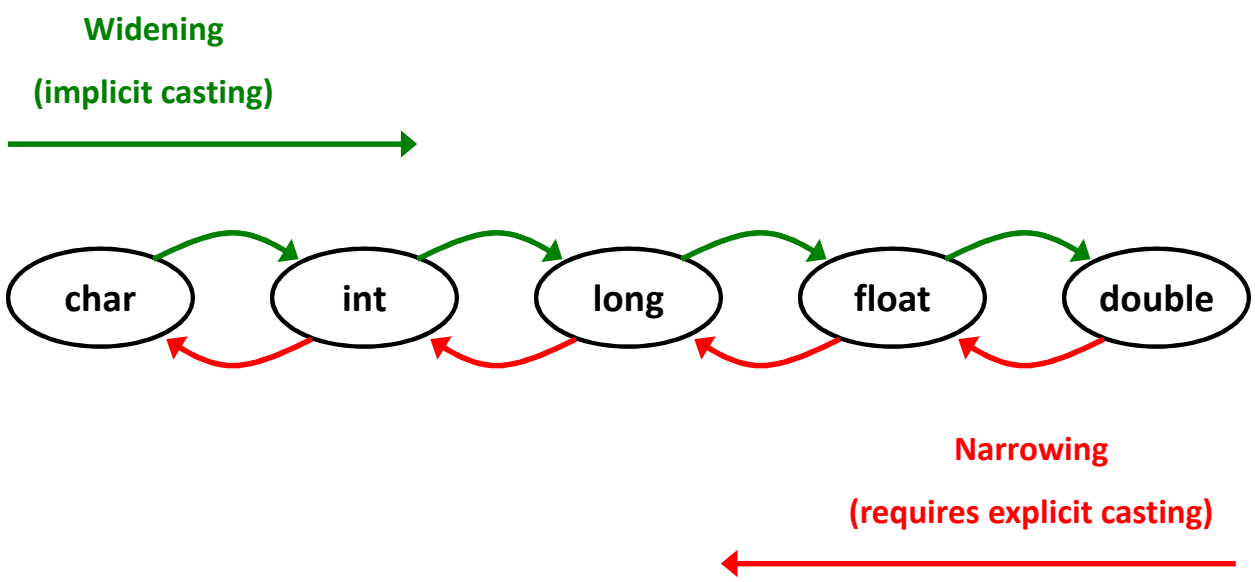
- An operator is a symbol that tells the compiler to perform a specific mathematical or logical operation.
- C language is rich in built-in operators and provides the following types of operators:
  - Unary Operators
  - Arithmetic Operators
  - Relational (Comparison) Operators
  - Bitwise (Logical) Operators
  - Short-Circuit Operators
  - Shift Operators
  - Assignment Operators
  - Misc (*miscellaneous*) Operators

2.1.1 Unary Operators

- It is operators work on just one operand, as follow:
  - + : positive sign
  - - : negative sign
  - ++ : increment operator to increase the value of the operand by 1 unit
  - -- : decrement operator to decrease the value of the operand by 1 unit
  - ! : Boolean inversion operator; !false=true, !true=false ; !0=1, !1=0 →(in C)
  - ~ : one's complement operator; convert to the one's complement of the operand; 0's → 1's and 1's → 0's
  - () : casting operator; to change the value of the operand to put it in different data type

Ex:
```
int x=5;   int y;
y = x++;          //  y = 5    and    x = 6
y = ++x;          //  y = 6    and    x = 6
```
Ex:
```
int x=3, y=5; float f;
f = y / x;        // f = 1.0;
f = (float)y/x;   // f = 1.66666
```

Implicitly (Auto Conversion) and Explicitly casting



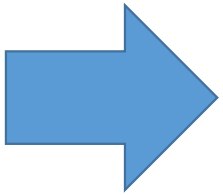
2.1.2 Arithmetic Operators

- The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
–	Subtracts second operand from the first.	A – B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

2.1.2 Arithmetic Operators

5 % 2 =	1
5 % -2 =	1
-5 % 2 =	-1
-5 % -2 =	-1



```
int x=30600;
int y=15236;
int z=x*y/x;
```

Unexpected results

2.1.3 Relational (Comparison) Operators

- The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

- Note: There is no boolean data type in c; 0 means false and non-zero means true.

2.1.4 Bitwise (Logical) Operators

- Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for **&**, **|**, and **^** is as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

2.1.4 Bitwise (Logical) Operators

- The following table lists the bitwise operators supported by C. Assume variable 'A' holds 7 and variable 'B' holds 5, then

Operator	Description	Example
&	**Binary AND Operator** copies a bit to the result if it exists in both operands.	(A & B) = 5, i.e., 0000 0101
|	**Binary OR Operator** copies a bit if it exists in either operand.	(A | B) = 7, i.e., 0000 0111
^	**Binary XOR Operator** copies the bit if it is set in one operand but not both.	(A ^ B) = 2, i.e., 0000 0010

2.1.5 Short-Circuit Operators

- Following table shows all the logical Short-Circuit operators supported by C language. Assume variable A holds 1 and variable B holds 0, then

Operator	Description	Example
&&	Short-Circuit AND Operator. If left operand equal false the result will be false without check on the right operand.	(A && B) is false= 0.
||	Short-Circuit OR Operator. If left operand equal true the result will be true without check on the right operand.	(A || B) is true= 1.

2.1.6 Shift Operators

- The following table lists the bitwise Shift operators supported by C. Assume variable 'A' holds 60, then:

Operator	Description	Example
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

2.1.7 Assignment Operators

- The following table lists the assignment operators supported by the C language:

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

2.1.7 Assignment Operators

- The following table lists the assignment operators supported by the C language:

Operator	Description	Example
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

2.1.8 Misc Operators

- There are a few other important operators including **sizeof** and ***ternary*** supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is float, will return 4.
&	Returns the address of a variable.	&a; returns the logical address of the variable a.
*****	Pointer to a variable.	*a; where a is a pointer variable
? :	Ternary Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Condition(s) ? True statement : false statement;

Ex: z=(x>y) ? 10 : 0;

Ex: (x>y) ? Z=10 : z=0;

2.1.9 Operators Precedence in C

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right

2.1.9 Operators Precedence in C

Category	Operator	Associativity
Logical Short-Circuit AND	&&	Left to right
Logical Short-Circuit OR	||	Left to right
Ternary	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

2.2 Control Statements

- There are some statements in all of programming languages to control the flow of the program execution; like conditional statements or repeating statements.
- In C there are two categories of control statements:
  - Branching Statements: which has two statements:
    - **if** statement
    - **switch** statement
  - Looping Statements: which has three statements:
    - **for** statement
    - **while** statement
    - **do .. while** statement

2.2.1 Branching Statements

- IF statement form is:

```
if( Condition(s) )
{
        …
        …     //true statements
        …
}
[else]
{
        …
        …     //false statements
        …
}
```

Example:

```
int grade = 48;

if(grade > 60)
      printf("Pass");
else
{
      printf("Fail");
}
```

2.2.1 Branching Statements

- Switch statement form is:

```
switch(myVariable){
      case value1:
        …
        …
        break;
      case value2:
        …
        …
        break;
      default:
        …
}
```

- **int**
- char
- enum

2.2.1 Branching Statements

- Switch example:

```
int type;
scanf("%d",&type);
switch(type)
{    case 10:
         printf("Perfect");
         break;
     case 5:
     case 4:
         printf("below avarage");
         break;
     default:
         printf("Not accepted");
}
```

2.2.2 Looping Statements (Iteration)

- **For statement:** The for loop is used when the number of iterations is predetermined.
- Its syntax:

```
for (initial statement(s) ; continuous condition(s) ; repeated step(s))
{
        …
        …
        …
}
```

```
for (i=0 ; i<10 ; i++)
{
      printf("%d\t", i);
}
```

2.2.2 Looping Statements (Iteration)

- **While Statement:** The while loop is used when the termination condition occurs unexpectedly and is checked at the beginning.
- Its syntax:

```
while (condition(s))
{
      …
      …
      …
}
```

```
int x = 0;

while (x<10) {
    printf("%d\n", x);
    x++;
}
```

2.2.2 Looping Statements (Iteration)

- **Do .. While Statement:** The do..while loop is used when the termination condition occurs unexpectedly and is checked at the end.
- Its syntax:

```
do
{
      …
      …
      …

}
while(condition(s));
```

```
int x = 0;

do{
     printf("%d\n", x);
     x++;
} while (x<10);
```

2.3 Break Statement

- The break statement can be used in loops or switch.
- It transfers control to the first statement after the loop body or switch body.

```
.......
while(age <= 65)
{
      ......
      balance = payment * 1;
      if (balance >= 25000)
            break;
}
.......
```

2.4 Continue Statement

- The continue statement can be used Only in loops.
- Abandons the current loop iteration and jumps to the next loop iteration.

```
.....
for( year=2000; year<= 2099; year++){
     if (year % 4 == 0)
          continue;
     printf("Y = %d", year)
}
.....
```

2.5 Comments in C

- To comment a single line:

```
// write a comment here
```

- To comment multiple lines:

```
/*  comment line 1
    comment line 2
     comment line 3 */
```

2.6 Introduction to Magic Box assignment

- You have the following box and need to put the numbers from 1 to 9 in each cell without repeating and with another constraint: the summation of each row equals to 15 and the summation of each column equals to 15 and the summation of each diagonal equals to 15.

	0	1	2
0	6	1	8
1	7	5	3
2	2	9	4

2.6 Introduction to Magic Box assignment - Algorithm

- The main constraints:
  - The number of rows equal to the number of the columns.
  - The order of the box (N) must be odd number; 3X3 or 5X5 or 31X31 and so on.
  - The numbers to put in the box start from 1 to NXN
- The algorithm steps:
  - Put the number “1” in the middle of the first row
  - Repeat the following test for each number to decide the place of the next number starting from number “1” till number “NXN -1”:

```
If (CurrentNumber % N !=0 )
      { decrement the current row: with constraint to circulate if necessary
        decrement the current column: with constraint to circulate if necessary
      }
else
      { increment the current row: with constraint to circulate if necessary
        use the same column
      }
 go to the right place and put the next number
```

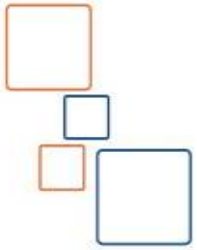
2.6.1 How to move the cursor in C?

- Some compilers has a function in ***conio.h*** library called ***gotoxy(),*** and some other has not.
- If your compiler has not this function you may make it in your program as follow:

```
void gotoxy1(int x, int y)
{
     COORD coord;
     coord.X = x;
     coord.Y = y;
     SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
```

- You just include Windows.h library and write the function before the main function for now.
- **Note:** you will learn how to write a function later.

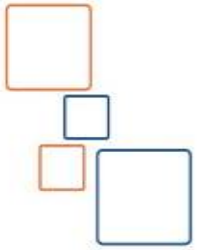
Lab Exercise



Assignments

- Write a C program to implement the algorithm of the Magic Box puzzle.
- Write a C program to receive numbers from the user, and exit when the sum exceeds 100.
- Write a C program to print a simple menu with 3 choices, when select one choice print the choice word and exit.
- Write a C program to print the multiplication table in ascending order from table 1 to table 10 sequentially and separated by group of “ *’s ”.
- Rewrite the previous program to print them in descending order.

Arrays and Strings



Invest In Yourself,
Develop Your Career

Content

- Meaning of Arrays and why we need it: Application Level
- Array Characteristics
- How to declare an array in C: Abstract Level
  - How to access a certain element and how to store or retrieve data
- How the array is implemented in the memory: Implementation Level
- Multi-Dimensional Arrays
- String as a one-dimensional array of character
- String Manipulation
- Normal and Extended Keys

3.1 Meaning of the array and why do we need it – Application Level

- An array is a collection of items stored at ***contiguous*** memory locations and elements can be ***accessed randomly*** using ***indices*** of an array. They are used to ***store similar type of elements*** as in the data type must be the same for all elements. They can be used to store collection of primitive data types such as int, float, double, char, or long. And so, an array can store complex or derived data types such as the structures, another arrays; but from the same types and size.
- **Why do we need arrays?**
  - The idea of an array is to represent many instances of data in one variable while they have a logical relation among them while this relation can be mapped to indicator to any element of these instances.
  - For example: list of the grads in a subject for group of students in a class: number of student is an indicator, the grads are the data stored in the array elements

3.2 Array Characteristics

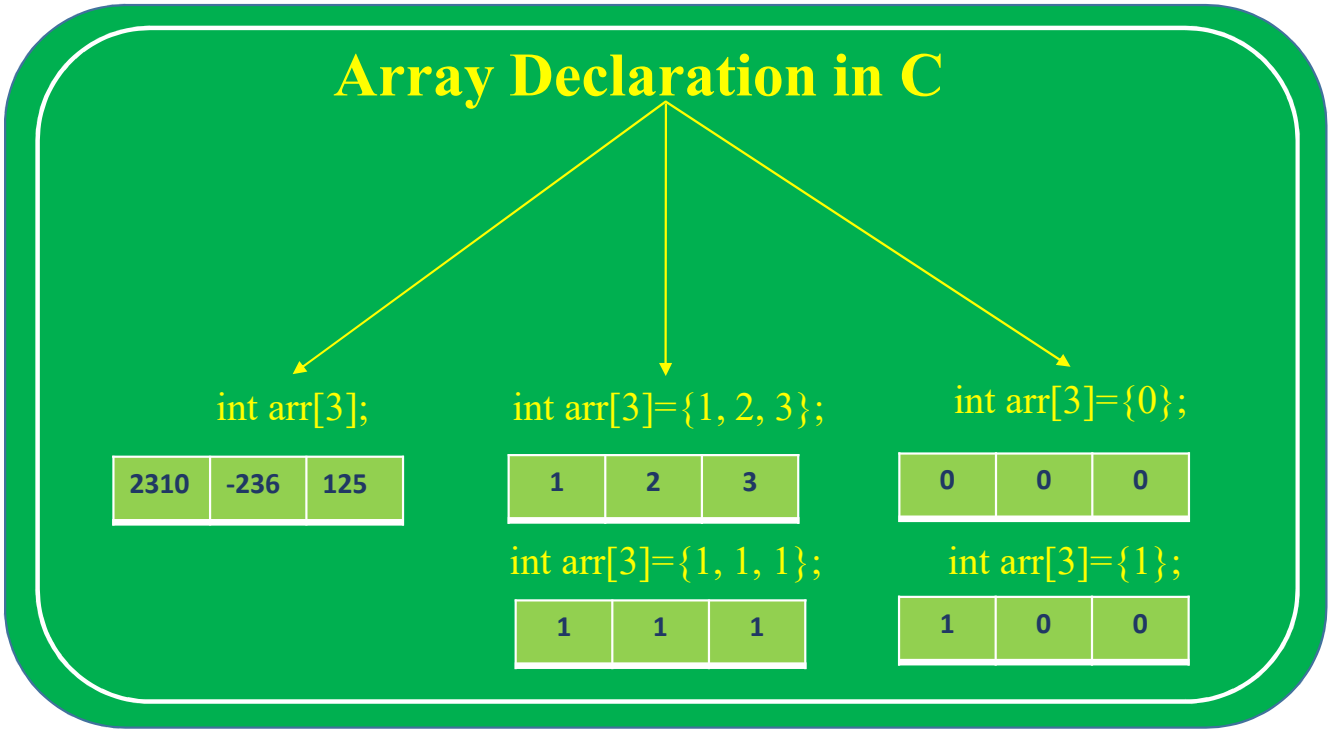
• There are seven characteristics for any array:

1. **Homogeneous**: All the elements data type must be the same
2. **Fixed size**: The size if the array can not changed at run-time (In C the size must be known in compilation-time).
3. **Contiguous**: All the instances of data for any array are allocated in contiguous memory locations.
4. **Indexed**: It uses the indexing technique to access any element (In C the index starts from 0 to (size -1))
5. **Ordered**: The places of the elements of any array are ordered.
6. **Finite (Limited)**: Any array has a first place and a last place. (Not circulated)
7. **Random or Direct Access**: The time consumed to reach to any element of an array is constant regardless its place

3.3 How to declare an array and access it in C: Abstract Level

- **Array declaration in C:**
  - There are various ways in which we can declare an array. It can be done by:
    - specifying its type and size,
    - by initializing it,
    - or both.
  - Array declaration by specifying size:
    - Data_Type Array_Name [Array_Size]; //some times we may use: Data_Type [Array_Size]Array_Name;
    - Ex: int arr [10];
  - Array declaration by initializing elements
    - Data_Type Array_Name [ ] = {val1, val2, val3, ….};
    - Ex: int arr [ ] = { 2, 4, -5, 88, -120}; // Compiler creates an array of size 5.
  - Array declaration by specifying size and initializing elements
    - Data_Type Array_Name [Array_Size] = {val1, val2, val3, ….};
    - Ex: int arr [10] = { 2, 4, -5, 88}; // Compiler creates an array of size 10, initializes first 4 elements as // specified by programmer and the rest 6 elements as 0.

3.3 How to declare an array and access it in C: Abstract Level



3.3 How to declare an array and access it in C: Abstract Level

- **Accessing Array Elements in C:**
  - Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1
  - Ex:

```
void main(void)
{       int arr[5];
        arr[0] = 5;
        arr[2] = -10;
        arr[3 / 2] = 2;          // this is the same as arr[1] = 2
        arr[3] = arr[0];
        printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);  // Output:  5   2   -10  5
}
```

- Note: There is no index out of bounds checking in C. ***Why?***

3.4 How the array is implemented in the memory: Implementation Level

- At first we need to clarify how the compiler declare and store the variables, by generate a ***Variables Vector***. It may look like the following table:
- Ex: int x; float f

Var Name	Type	Size	Address	…	…	…
x	int	2	2120			
f	float	4	3240			

- And so the compiler generate a vector for arrays, it may be look like the following table:
- Ex: int arr[10];

Array Name	Type	Size	Element Size	Base Address	…	…
arr	Int	10	2	1000		

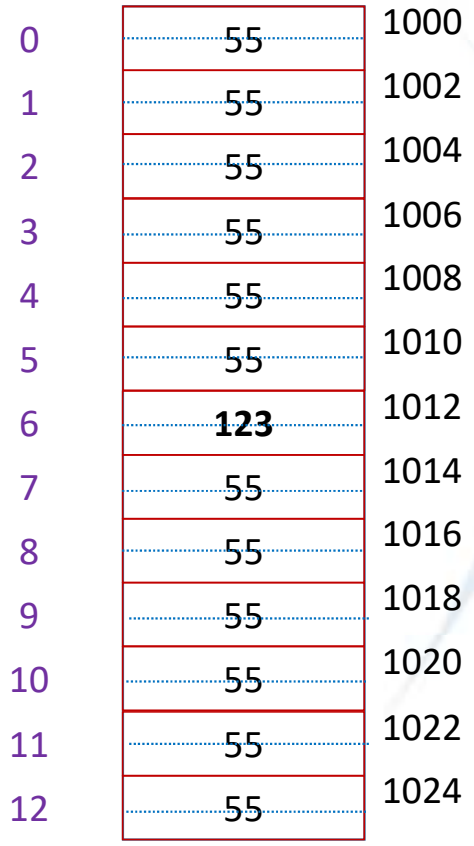
3.4 How the array is implemented in the memory: Implementation Level

- The allocation of the array will be in memory starting from base address –as the address of first byte allocated to the array- with long of bytes equal to number of elements X element size. For the above example the base address is 1000 and the size of array in bytes = 10X2 =20 bytes; so the allocated locations for the array in memory from 1000 till 1019 – contiguous -
- The compiler generate an equation to access any element of the array using the index of the element. It may be look like:
  - *the address of element with index I = base address + I X element size*
- For the above example: the equation will be:
  - The address for element indexed with I = 1000 + I X 2
  - Ex the address of the fourth element in the array (Index = 3) arr[3] is 1006

    address of arr[3] = 1000 + 3 X 2 = 1006
- Note: for the way of array implementation there is no index out of bounds checking in C.

3.4 How the array is implemented in the memory: Implementation Level

```
int arr[13],x;
arr[6] = 123;
X = arr[12];   // x = 55;

The address computing:
     base + index * element size;
Case arr[6]:
     1000+6*2 = 1012
Case arr[12]
     1000+12*2= 1024
```



3.5 Dealing with arrays by looping statements

- Ex:

```
int i , arr[10];
for(i  = 0 ; i < 10 ; i ++)
  scanf("%d", & arr[i]);
for (i = 9; i >=0 ; i --)
  printf("%d", arr[i]);
```

3.6 Multi-Dimensional Arrays

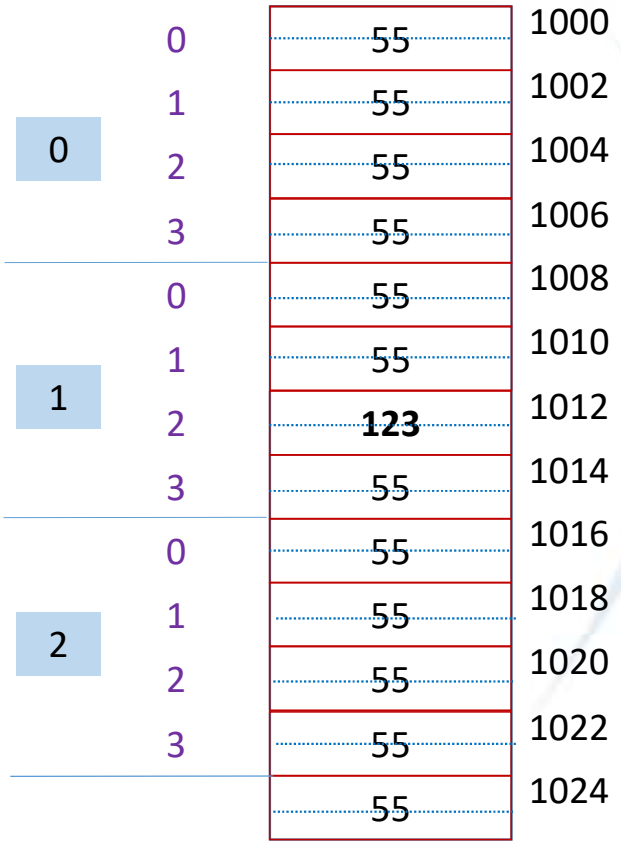
- Array may be array of arrays, that is mean, to access an array element you have to use multi-indices: each level of index represent a dimensional level.
- Declaration and accessibility: as two dimensional
  - Data_Type Array_Name [Dim1_Size][Dim2_Size];
  - Ex: int arr2d [4] [5];
  - arr2d[0][0] = 22; // first element
  - arr2d [3][4] = 99; // last element
- The Equation for the two-dimensional array:
  - *the address of element with index A , B = base address + A X 2nd Dim_array_size + B X data_element_size*

  Ex: address of arr2d[1][2] = base address + 1 X (5 X 2) + 2 X 2

- Declaration of N dimension:
  - Data_Type Array_Name [Dim1_Size] [Dim2_Size] [Dim3_Size] …. [DimN_Size];

3.6 Multi-Dimensional Arrays

```
int arr2d[3][4],x ;
arr2d[1][2] = 123;
x = arr2d[2][3];

The address computing:
   base address + 1st index * 2nd dim array size
                + 2nd index * element size
Case arr2d[1][2]: 1000 + 1*(4*2) + 2*2 = 1012
Case arr2d[2][3]: 1000 + 2*(4*2) + 3*2 = 1022
```



3.7 String

- There is no string data type in C language, the dealing with a string in C is represented by using a one dimensional array of character.
- In C programming, a string is a sequence of characters terminated with a null character **'\0'**
- There are two ways to declare a string in c language:
  - By char array
  - By string literal
- Let's see the example of declaring string by char array in C language:
  - char str[10]={'C', '-', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
- As we know, array index starts from 0, so it will be represented as in the figure given below:

0	1	2	3	4	5	6	7	8	9
C	-	P	r	o	g	r	a	m	\0

3.7 String

- While declaring string, size is not mandatory. So we can write the above code as given below:
  - char str1[]={'C', '-', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
- We can also define the string by the string literal in C language. For example:
  - char str2[]="C-Program";
- In such case, '\0' will be appended at the end of the string by the compiler.
- There are two main differences between char array and literal:
  - We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
  - The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

3.7 String

- **Notes:**
- char str1[] = {‘A’, ‘S’, ‘A’, ‘D’}; // is this a String or not?
- char str2 [5] = {‘A’,’L’,’Y’,’\0’}; // is this a String or not?
- char str3 [6] = “Information”; // is this a String or not?
- str3 = “Technology”; // Can we assign a string to array of char after declaration? **NO**

3.8 String Manipulation

- **Read a string from a user:**
  - By using **scanf("%s", str);** // problem with spaces
  - By using other functions for read lines like: **gets(str);** // accept the spaces
- **Print a string on screen:**
  - By using **printf("%s", str);**
  - By using other functions like: **puts(str);**
- **Functions in string.h:** there are many functions like:
  - **strlen(str)** : return the number of characters in the string – before the terminator
  - **strcat(str1,str2)** : concatenation the second string to the first string
  - **strcpy(destination, source)** : copy the source to the destination
  - **strcmp(str1, str2)** : compare between the two strings and return 0, or +ve value or –ve value !!
    - 0 : if str1 equals to str2
    - +ve: if str1> str2 : if the ASCII value of the first unmatched character is greater than the second and return the subtraction.
    - -ve if str1< str2 : if the ASCII value of the first unmatched character is less than the second and return the subtraction.

3.8 String Manipulation

- **Functions in string.h:** there are another functions compiler dependent like:
  - **strlwr()** : converts string to lowercase, found in *Borlandc* or *Turboc*
  - **strupr()** : converts string to uppercase, found in *Borlandc* or *Turboc*
  - **strrev(str)** : reverse characters of a string, found in *Borlandc* or *Turboc*
  - **strcmpi(str1, str2)** : compare between the two strings –case-insensitive- and return 0, or +ve value or –ve value !!, found in *Borlandc* or *Turboc*
    - 0 : if str1 equals to str2
    - +ve: if str1> str2 : if the ASCII value of the first unmatched character is greater than the second and return the subtraction.
    - -ve if str1< str2 : if the ASCII value of the first unmatched character is less than the second and return the subtraction.
  - **strstr(s1, s2)** : Find the first occurrence of a substring(s2) in another string (s1), and return the address of the occurrence if success or return NULL, found in *Borlandc* or *Turboc*.

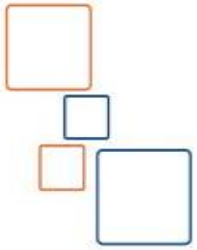
3.9 Normal Keys and Extended Keys

- Each key in the keyboard has an ASCII code, which is limited by one byte.
- At the first of computer generation the keyboard was limited not like today; there was arrows or page up or down of function keys, …. etc.
- So, the new added keys needed ASCII code to be manipulated, so the Extended Keys are appeared.
- Its ASCII encapsulated in two bytes, the lowest byte equals null and the second byte has a code.
- When you pressed on an extended key there was 2 bytes are stored in the keyboard local buffer.
- The following program clarify how to now any key if it is normal or extended and print its code.

3.9 Normal and Extended Keys

```
void main (void)
{
     char ch;
      flushall(); // or fflush(stdin); (compiler dependent)
     printf("\n Press the key u want to know its type and code");
     ch=getch();
     if(ch!=NULL) // Some compiler not accept, replace with -32 or 224
          printf("\nIt is a normal key with code = %d",ch);
     else{
          ch=getch();
          printf("\nIt is a Extended key with 2nd byte code = %d",ch);
     }
     getch();
}
```

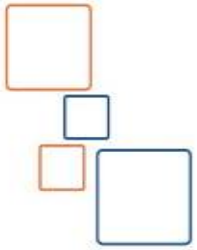
Lab Exercise



Assignments

- Write a program to read an array and print it using 2 for loops?
- Write a program to find the maximum and minimum values of a set of numbers using a single dimension array.
- If you have a matrix of dimension 3*4. Write a program to read it from the user and find the sum of each row & the average of each column

Structures and array of structures



Invest In Yourself,
Develop Your Career

Content

- Structure Meaning
- Structure definition (Abstract level)
- Structure declaration (Abstract level)
- Accessing Structure Members (Abstract level)
- Structure implementation in the memory (Implementation level)
- Array of Structures

4.1 Structure Meaning

- Structure is a user-defined datatype in C language which allows you to combine data of different types together. Structure helps to construct a complex data type which is more meaningful.
- **For example:** If I have to write a program to store Employee information, which will have Employee's name, age, address, phone, salary etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.
- In structure, data is stored in form of records.

4.2 Structure definition (Abstract level)

- ***struct*** keyword is used to define a structure. ***struct*** defines a new data type which is a collection of primary and derived datatypes.

```
struct structure_Name
{
    type member1;
    type member2;
    type member3;
     ...
/* declare as many members as desired, but the entire
structure size must be known to the compiler. */
}[structure_variables];
```

4.2 Structure definition (Abstract level)

- Such a struct declaration may also appear in the context of a *typedef* declaration of a type alias or the declaration or definition of a variable:

```
typedef struct tag_name {
    type member1;
    type member2;
    type member3;
       ...
} struct_alias;
```

4.2 Structure definition (Abstract level)

- Example of Employee

```
struct Employee
{
        int ID;
        float Age;
        float salary
        float deduct;
        float bonus;
        char Name[51];
};
```

- To make the definition of a structure available to all functions in you program, you have to define it as part of preprocessing part in a C program.

4.3 Declaration a variable from a structure

- Structure variable declaration is similar to the declaration of any other datatype. Structure variables can be declared in following two ways:
  - Declaring structure variable separately,
  - Declaring Structure variables with structure definition
- **Declaring structure variable separately**

```
struct Employee e1;
```

- **Declaring Structure variables with structure definition**

```
struct Employee
{
        int ID;
        float Age;
        float salary
        float deduct;
        float bonus;
        char Name[51];
} e2;
```

4.3 Declaration a variable from a structure

- **Initialization of a structure variable**

```
struct Employee e1 = {213, 46, 3562.12,
                      324.2, 2.5,
                       "Mohamed Aly"};
```

4.4 Accessing Structure Members

- Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable using a dot "." operator also called *period* or *member access operator.* For Example:

```
struct Employee e1;
e1.ID = 213;
scanf("%f",&e1.salary);
gets(e1.Name);
strcpy(e1.Name, "Hossam");
e1.Name[1] = 'M';
f = e1.salary + e1.bonus - e1.deduct;
```

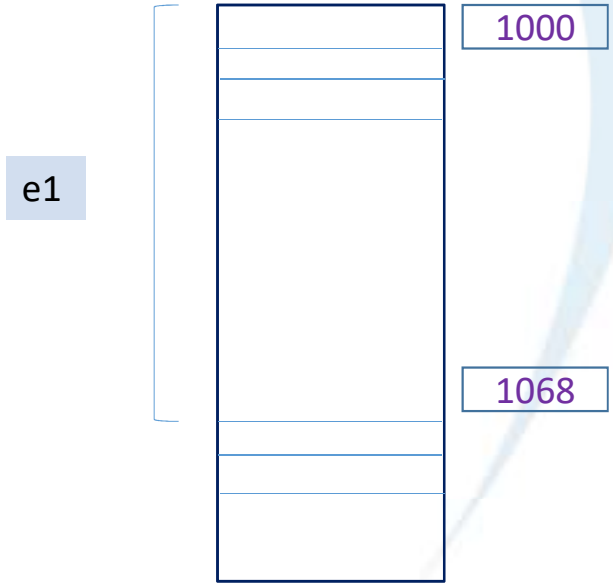
4.5 Structure implementation in the memory (Implementation level)

- When the compiler read the definition of a structure it will create a structure table for that structure datatype, it looks like:

Struct Employee Table

Field Name	Field Type	Field Size	offset
ID	int	2	0
Age	float	4	2
Salary	float	4	6
deduct	float	4	10
Bonus	float	4	14
Name	char[51]	51	18

69



4.5 Structure implementation in the memory (Implementation level)

• Example:

```
e1.Salary = 333.3;
```

The offset of the field ‘Salary’ is added to the address if variable e1to reach to the address of the field and store the data.

4.6 Array of Structure

- The form is:

  struct Struct-Name Array_Name [Array_size];

- Example:

```
struct Employee empArr[5];

empArr[2].salary = 2315.6;
gets(empArr[3].Name);
scanf("%f",&empArr[0].bonus);
empArr[4].Name[2] = 'T';
```

- Note: if we use ***typedef*** keyword when defining the structure we can use the *alias name* directly without need to use the ***struct*** keyword

4.6 Array of Structure

- The array of structures initialization can be done by assign the values for each structure of the array, for example:

```
struct Employee empArr[2]= {
        {10, 35.2, 4500, 234.7, 200, "Hassan Aly"},
        {20, 42.3, 7500, 456.3, 450, "Mohsen Ayman"} };
```

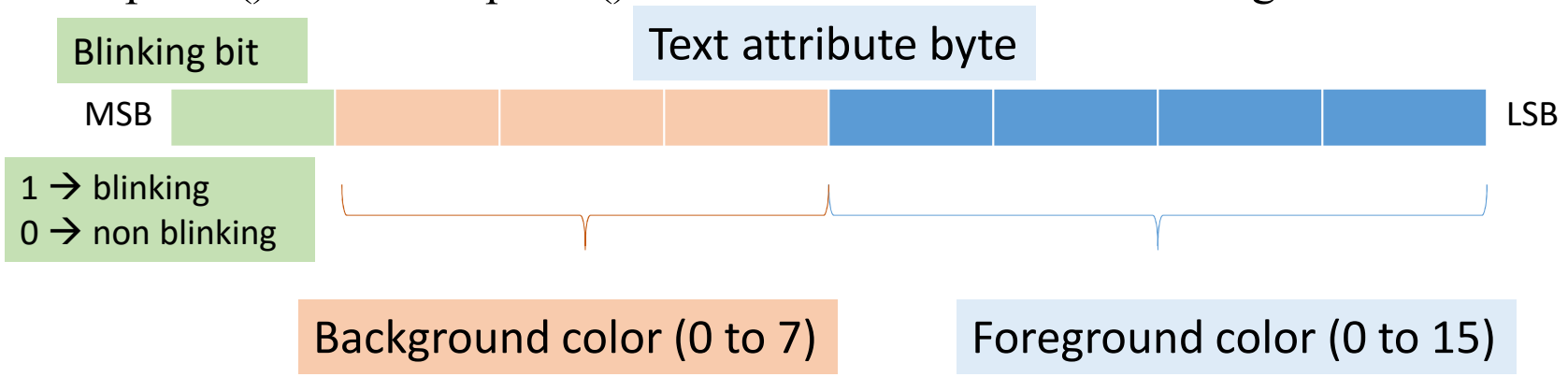
4.7 Introduction to Highlight Menu Assignment

> New
  Display
  Exit

• up arrow, down arrow, home, end, enter and escape.

Text attributes in Borland C

- To control the text attribute in Borland C by using the following functions:
  - textattr(integer value to represent the foreground color and background color)
  - textbackground(integer value to represent the background color and attribute)
  - textcolor(integer value to represent the foreground color)
  - cprintf() instead of printf() for affect the text attribute using above functions



Needed Data

1. defined some replacement texts:

```
#define UP              0x48      //Extended Key (72)
#define DOWN            0x50      //Extended Key (80)
#define HOME            0x47      //Extended Key (71)
#define END             0x4F      //Extended Key (79)
#define ENTER           0xD       //Normal Key   (13)
#define ESCAPE          0x1B      //Normal Key   (27)
#define NORMAL_ATTR     0x07      //work with BorlandC only
#define HILIGHT_ATTR    0x70      //work with BorlandC only
#define New             0
#define Display         1
#define Exit            2
```

Needed Data

2. Array of string to represent the menu items:

```
char Menu[3][10] = {" New      ", " Display ", " Exit    "};
```

3. Some primitive data types:

- **terminated**:int initial by 0 ; using as the flag termination
- **row**:int initial by the row position for the first item to be printed
- **col**:int initial by the col position for the items to be printed
- **pos**:int intial by 0; it is represent the position of selected item
- **i**:int as a needed counter
- **ch**:char to read the key pressed by the user

The algorithm

```
do{
    0- clear the user screen ( clrscr() | system("cls"); )
    1- draw the menu with selected current item
    2- read the pressed key from the user
    3- take the suitable action dependent on the pressed key

}while(!terminated);
```

1- draw the menu with highlighted current item

```
for (i=0; i<3; i++)
{
    1- go to the suitable place to print the current item
            (you may use the gotoxy() function in slide 57)
    2- if(i == pos)
        print the selection indicator character ">"
        (or change to highlighted attributes)

    3- print the current menu item
    (if you made changing highlighted attributes in step 2,
    you have to make changing to normal attribute here)
}
```

2- Read the pressed key by the user

```
ch = getch();
```

3- Take the suitable action

```
switch(ch)
{
     case NULL://Some compilers not accept, replace with (-32 or 224)
          - It is one of the extended keys : END, HOME, UP_ARROW, or
               DOWN_ARROW
          break;
     case ENTER:
          - Dependent on the value of pos it will perform a desired
            task
          break;
     case ESC:
          terminated = 1;  // to escape from the do .. While loop and
                           // terminate the program
}
```

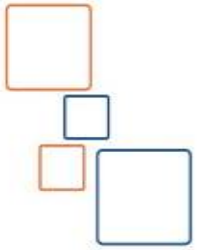
3- Take the suitable action

```
case Null:
     ch=getch(); // get the code of the 2nd byte for the normal key
     switch(ch)
     {    case UP_ARROW:
               decrement the pos value with constraint of circulation if
               necessary
               break;
          case DOWN_ARROW:
               increment the pos value with constraint of circulation if
               necessary
               break;
          case HOME:   pos = 0;     break;
          case END:    pos = 2 ;    break;
     }
```

3- Take the suitable action

```
case ENTER:
    switch(pos)
    {   case New:
            make the new action
            break;
        case Display:
            make the display action
            break;
        case Exit:
            terminated = 1;
            break;
    }
```

Lab Exercise



Assignments

- Write a program to display a menu to the user with the following options

  New

  Display

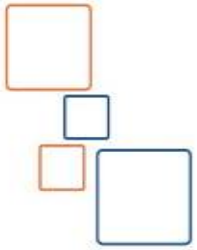
  Exit

The target of each option is to display for the user a specified message. The program terminates if the user choose ‘Exit’ from the menu or press ‘ESC’.

N.B. The keys available to the user is up arrow, down arrow, home, end, enter and escape.

- Write a program to receive one employee’s data display the code, name, and net salary.
- Write a program to receive data into an array of 5 employees, then display the code, name, and net salary for each.

Modularity In C Functions



Invest In Yourself,
Develop Your Career

Content

- Modularity
- Function Header and prototype
- Function Parameters and Return Data Type
- Function Body
- Functions Arguments
- Sequence Passing arguments in C
- Recursion and its Constraints

5.1 Meaning of Modularity

- Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- In C the function is the tool to perform the modularity. Each function has its objectives to execute, it may be divided into many smaller functions to perform it.
- Functions can call any other functions, either the main function itself – with constraints for avoiding the infinite loop case-
- Functions may call itself, called recursive function with constraints too.

Function Header, Body, and Prototype

- The function consists of:
  - Header,
  - Body
- Function header consists of 3 parts as follow:

  Return_Datatype Function_Name (List_OF_Input_Parameters)
- Function Body which contains the implementation of the algorithm to execute the function work which is written inside **{ }**
- We can write the function – header and body – before the function will call it, we can put it in any place in the file but after write its prototype in the preprocessing part before any functions.
- Function prototype is the copy of function header with "**;**" at the end of header

  Return_Datatype Function_Name (List_OF_Input_Parameters);

5.2 Example 1

```
int sum2Var(int A, int B)
{
    int sum;
    sum = A+ B;
    return sum;
}
```

- The reserved word "***return***" are used with functions to terminate its work and return to the caller function, if the called function is expected to return datatype, the using of return must added with the value – or variable- with the return datatype.
- How this function is called and how to write its prototype?

5.2 Example 1

```
int sum2Var(int A, int B);          // Function prototype
void main (void)
{
     int x=5, y=7, z;
     z = sum2Var(x, y);             // Call the function with vars as arguments
     z = sum2Var(-7, 12);           // Call the function with values as args
     printf ("%d", sum2Var(51, -31)); //We can call a function as an argument
}
int sum2Var(int A, int B)               // Function Header
{
     int sum;                           //
     sum = A+ B;                        //   Function Body
     return sum;                        //
}
```

5.3 The different types of functions

- Function may have no return datatype, and may have no list of input parameters; write ***void*** in both places.
- Function may have no return datatype, and may have list of input parameters; write ***void*** at the beginning of header and write the type and name for each input parameter.
- Function may have return datatype, and may have no list of input parameters; write the return data type in the header of function –either primitive or complex- and void in place of list of input parameters
- Function may have return datatype, and may have list of input parameters; write the return data type in the header of function –either primitive or complex- and write the type and name for each input parameter.

5.4 Parameters and Arguments

- The term parameter refers to any declaration within the parentheses following the function name in a function declaration or definition; the term argument refers to any expression within the parentheses of a function call.
- The following rules apply to parameters and arguments of C functions:
  - Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. Or no parameters nor arguments
  - The maximum number of arguments (and corresponding parameters) is 253 for a single function.
  - Arguments are separated by commas. However, the comma is not an operator in this context
  - Arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value
  - The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated.

5.5 The sequence of passing the arguments to the function

- We pass different arguments into some functions. Now one questions may come in our mind, that what the order of evaluation of the function parameters. Is it left to right, or right to left?
- To check the evaluation order we will use a simple program. Here some parameters are passing. From the output we can find how they are evaluated.

Example 2:

```
#include<stdio.h>
void test_function(int x, int y, int z) {
   printf("The value of x: %d\n", x);
   printf("The value of y: %d\n", y);
   printf("The value of z: %d\n", z);
}
main() {
   int a = 10;
   test_function(a++, a++, a++);
}
```

Example 2:

```
Output will be:
The value of x: 12
The value of y: 11
The value of z: 10
```

- From this output we can easily understand the evaluation sequence. At first the z is taken, so it is holding 10, then y is taken, so it is 11, and finally x is taken. So the value is 12.

5.6 The main return values

- The return value for main indicates how the program exited. Normal exit is represented by a 0 return value from main. Abnormal exit is signaled by a non-zero return.
- As old fashion in C programming the values 1, 2, or 3 represent abnormal termination and represent the following state for each value:
  - 1 : abnormal exit without saying the reason
  - 2 : abnormal exit for memory problem (could not allocate memory, …)
  - 3 : abnormal exit for I/O problem (could not change driver mode, …)
- In general the using values are:
  - Return 0 for normal exit : like calling function exit(0)
  - Return non-zero value for abnormal exit: like calling function exit(1)

5.7 Recursion

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.
- **What is the base condition in recursion?** In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

5.7 Recursion

- **How a particular problem is solved using recursion?** The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 1. We return 1 when n = 1.
- **Why Stack Overflow error occurs in recursion?** If the base case is not reached or not defined, then the stack overflow problem may arise.
- **How memory is allocated to different function calls in recursion?** When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

5.7.1 Recursion Example: Factorial

• Iteration version:

```
int Fact( int n)
{
    int result=1, i;
    for( i = n; i > 1; i--)
        result *= i;
    return result;
}
```

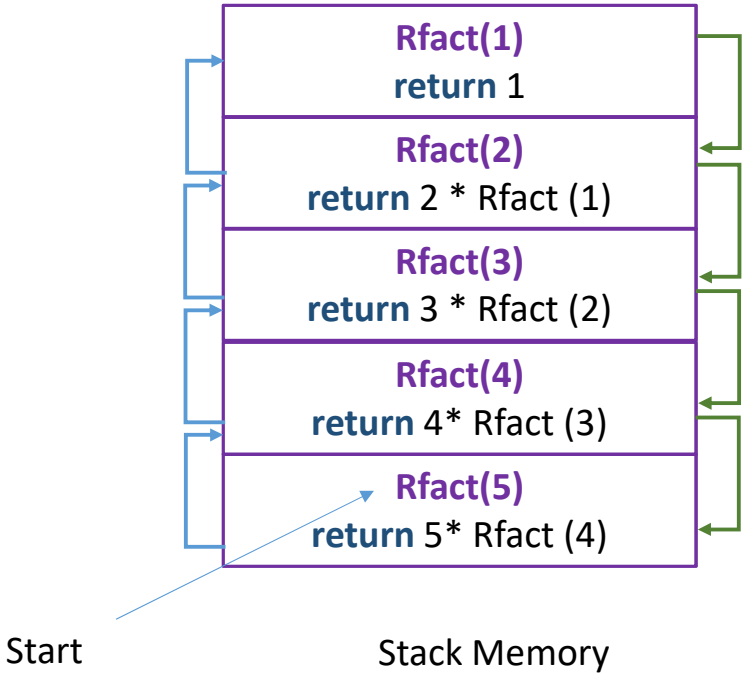
5.7.1 Recursion Example: Factorial

• Recursive version:

```
int RFact( int n)
{
    if (n == 1)
        return 1;
    return n * Rfact( n-1 );
}
```

5.7.2 How to trace a recursive function

```
RFact(5)= 5 *
     RFact(4) = 4 *
          Rfact(3) = 3 *
               RFact(2) = 2 *
                    RFact(1) = 1
```



5.7.3 Example 2: Fibonacci series

- The following example generates the Fibonacci series for a given number using a recursive function

```
#include <stdio.h>
int fib (int i) {

    if(i == 1 || i == 2) {
       return 1;
    }

    return fib (i-1) + fib (i-2);
}
```

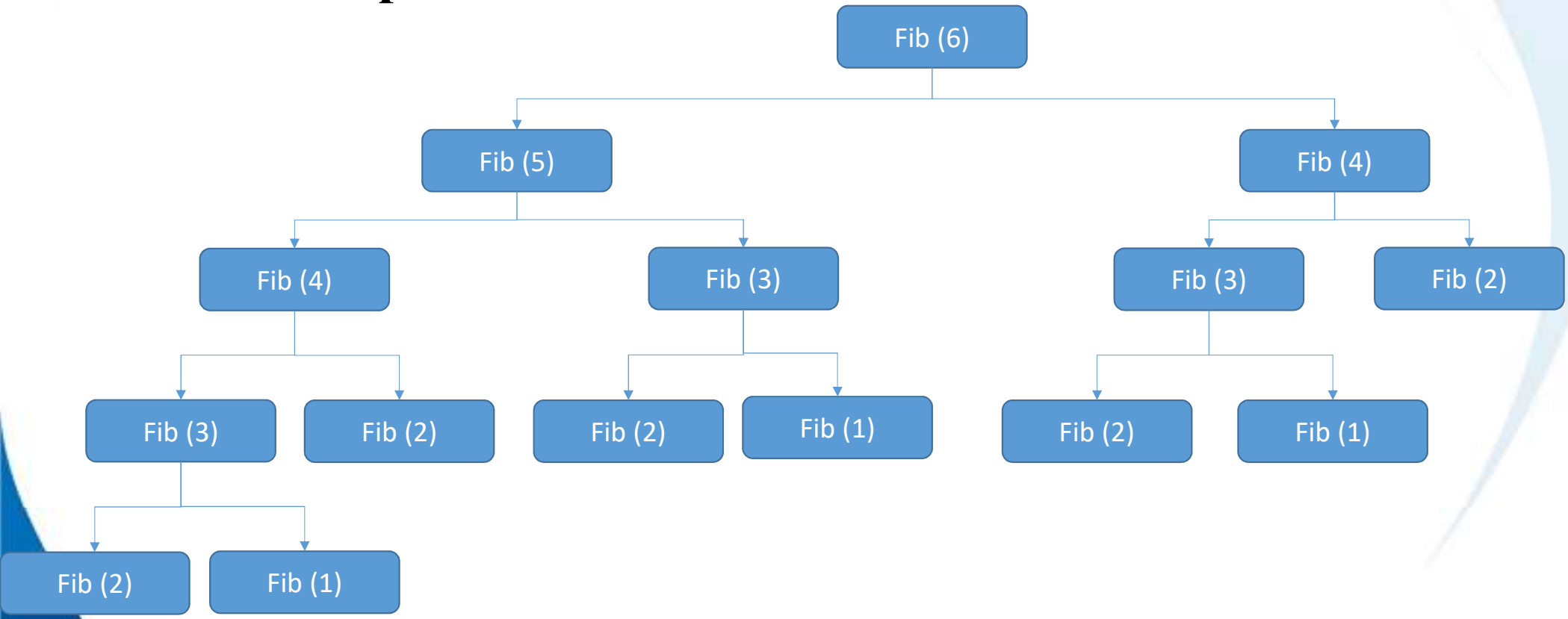
5.7.3 Example 2: Fibonacci series

```
int  main(void) {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fib(i));
   }

   return 0;
}
```

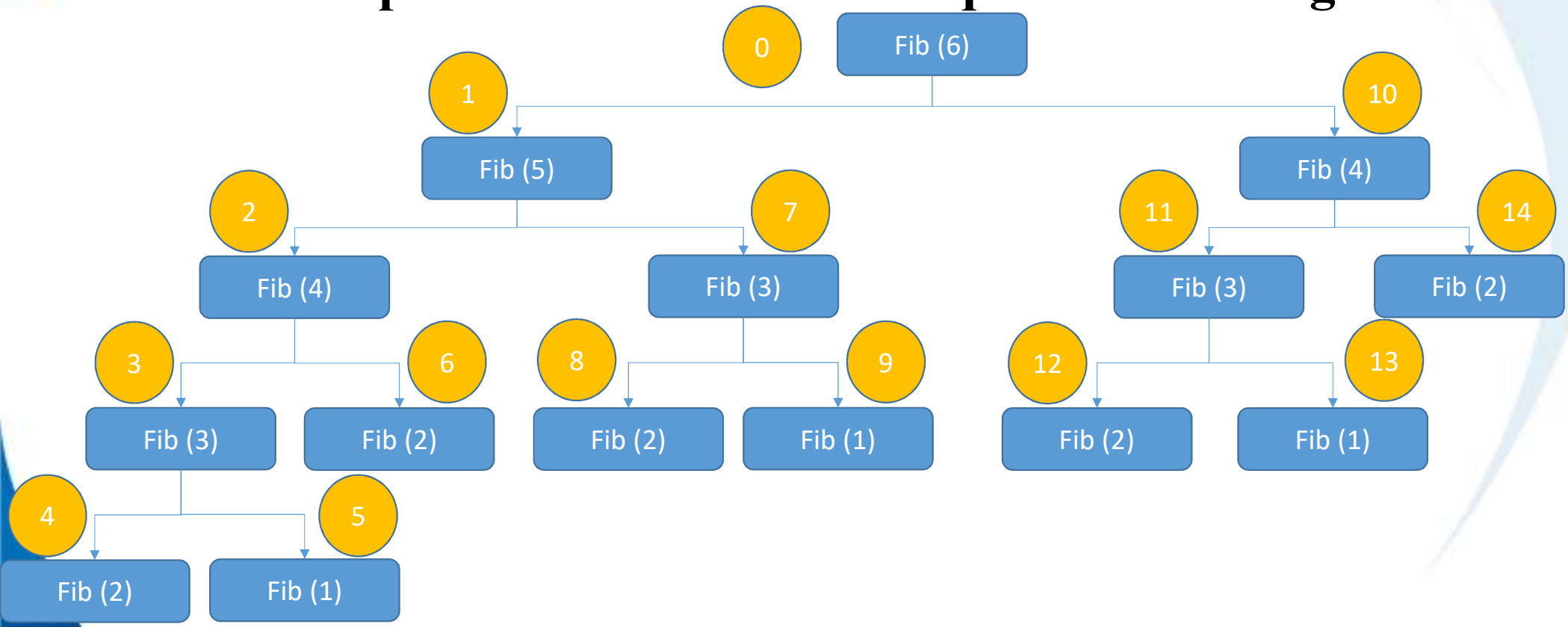
5.7.3 Example 2: Fibonacci series

```
The output as follow:
1
1
2
3
5
8
13
21
34
55
```

5.7.3 Example 2: Fibonacci series



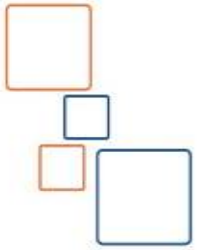
5.7.3 Example 2: Fibonacci series sequence of calling



5.7.4 Advantages and Disadvantages of Recursion

- Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.
- That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.
- There is an important technique used in algorithms dependents on recursion concept, ***Divide-and-conquer*** which is used in many types of problem solving such sorting and searching problems, such as Merge sort and binary search algorithms.

Lab Exercise

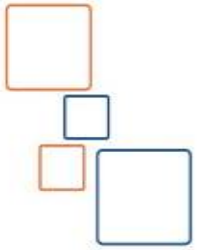


Invest In Yourself,
Develop Your Career

Assignments

- Write two functions to read an Employee's data and other to print its Name with the net salary.
- Write a recursive function to compute the power operation: X^Y

Pointer Datatype



Invest In Yourself,
Develop Your Career

Content

- Introduction to Pointers
- Pointers' Arithmetic
- Arrays and pointers
- Pointers Comparisons
- Using Pointers to call by address
- Using pointers to pass an array to function
- Introduction to dynamic allocation
- malloc(), realloc(), and free() functions
- Dynamic allocations of arrays
- Pointer to pointer and array of pointers
- The standard main header in c

6.1 Introduction to Pointers

- Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.
- **What are Pointers?** A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
Datatype * Pointer_Name;
```

Ex: `int * ptr;`

- ***Datatype*** is the pointer's base type; it must be a valid C datatype and ***Pointer_Name*** is the name of the pointer variable. The asterisk **'*'** used to declare a pointer

6.1 Introduction to Pointers

- The actual data type of the value of all pointers, whether pointer to integer, float, character, or otherwise, is the same, ***an hexadecimal number that represents a memory address (Logical address)***. The only difference between pointers of different datatypes is the datatype of the variable or constant that the pointer *points to*.
- The key to deal with pointer is to know at any time if you deal with its contents or the contents of what it points to
- As you know, every variable has memory location (s) and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined:

6.1 Introduction to Pointers

```
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

• The output:

Address of var1 variable: fff4

Address of var2 variable: ffea

6.1.1 How to Use Pointers?

- There are a few important operations, which we will do with the help of pointers very frequently.
  - We define a pointer variable,
  - Assign the address of a variable to a pointer and,
  - Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.
- The following example makes use of these operations:

6.1.1 How to Use Pointers?

```
#include <stdio.h>
void main (void) {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
                  /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
                  /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
}
```

6.1.1 How to Use Pointers?

- The output of the above example:

```
Address of  var  variable: fff4
Address stored in  ip  variable: fff4
Value of  *ip  variable: 20
```

6.2 Pointers Arithmetic:

- A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.
- There are four arithmetic operators that can be used on pointers: **++, --, +,** and **–**
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address **1000**. Assuming 16-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr ++;
```

- After the above operation, the **ptr** will point to the location **1004** because each time **ptr** is incremented, it will point to the next integer location which is 2 bytes next to the current location.
- This operation will move the pointer to the next memory location without impacting the actual value at the memory location.
- If **ptr** points to a character whose address is **1000**, then the above operation will point to the location **1001** because the next character will be available at **1001**.

6.2 Pointers Arithmetic: Example

- Consider the following piece of code:

```
int x = 5;          // consider address of x is 1000
int * ptr = &x; // the ptr is points to address 1000
```

- What will happen if we run each of the following statements in x an ptr?

```
y = * ptr ++;     // ptr=1002, x=5, y= 5 (compiler dependent)
y = * (ptr)++;    // ptr=1002, x=5, y= 5 (compiler dependent)
y = (*ptr)++;     // ptr=1000, x=6, y=5
```

6.2 Pointers Arithmetic: Example

```
int main(void) {
    char arr[] = "ITI World";
    char *ptr = &arr[0];
    char *ptr1 = &arr[0];
    char *ptr2 = &arr[0];
    ++*ptr; // increment the content of the position pointed by ptr
    printf("\n value of *ptr = %c \t %x",*ptr, ptr);
    *ptr1++;// get the content of the position after increment ptr1
    printf("\n value of *ptr1= %c \t %x", *ptr1, ptr1);
    *++ptr2;// get the content of the position after increment ptr2
    printf("\n value of *ptr2 = %c \t %x", *ptr2, ptr2);
    return 0;
}
```

6.2 Pointers Arithmetic: Example

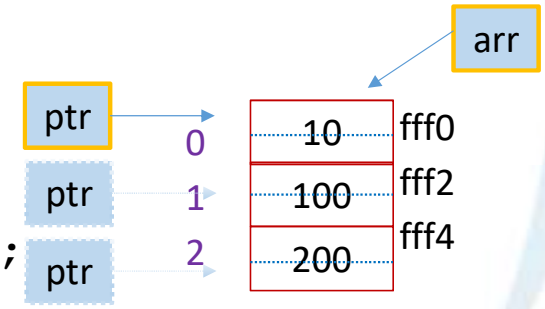
```
Output as follow:

value of *ptr = J  ffe8
value of *ptr1= T  ffe9
value of *ptr2= T  ffe9
```

6.3 Arrays and Pointers

- The array name is the base address; the first address dedicated to the array elements, so we can assign the array name to a pointer to the same datatype of array.
- The variable pointer can be incremented, unlike the array name which cannot be incremented because it is a ***constant pointer***.
- The following program increments the variable pointer to access each succeeding element of the array

6.3 Arrays and Pointers

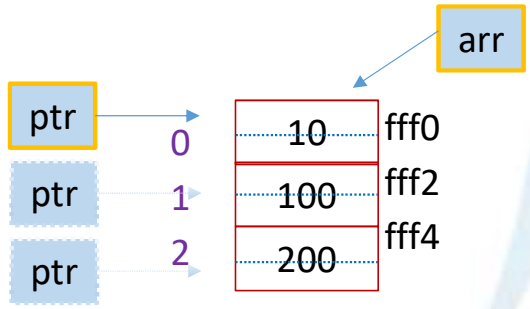
```
void main (void) {
    int  arr[] = {10, 100, 200};
    int  i, *ptr;
    /* let us have array address in pointer */
    ptr = arr;
    for ( i = 0; i < 3; i++) {
       printf("Address of arr[%d] = %x\n", i, ptr );
       printf("Value of arr[%d] = %d\n", i, *ptr );
       /* move to the next location */
       ptr++;
    }
}
```



6.3 Arrays and Pointers

• The output will be as follow:

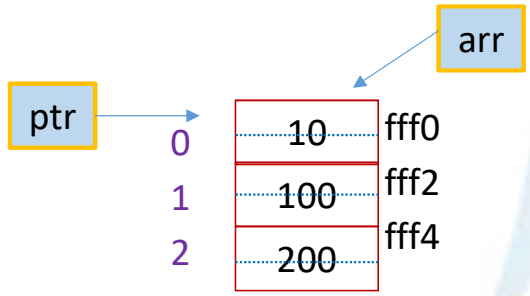
```
Address of arr[0] = fff0
Value of arr[0] = 10
Address of arr[1] = fff2
Value of arr[1] = 100
Address of arr[2] = fff4
Value of arr[2] = 200
```



6.3 Arrays and Pointers

- One of the advantages of using the pointers is: we can deal with locations in memory relatively to what pointer points to. For example:

```
void main (void) {
    int  arr[] = {10, 100, 200};
    int  i, *ptr;
    /* let us have array address in pointer */
    ptr = arr;
    for ( i = 0; i < 3; i++) {
       printf("Address of arr[%d] = %x\n", i, ptr+i );
       /* deal with the ith location after pointer points to*/
       printf("Value of arr[%d] = %d\n", i, *(ptr+i) );
    }
}
```



6.3 Arrays and Pointers

- We can deal with the name of the array as a pointer – but without trying to change it- i.e. we may use the pointer operator with array name and vise versa, we can use the array operator with the pointer. For example:

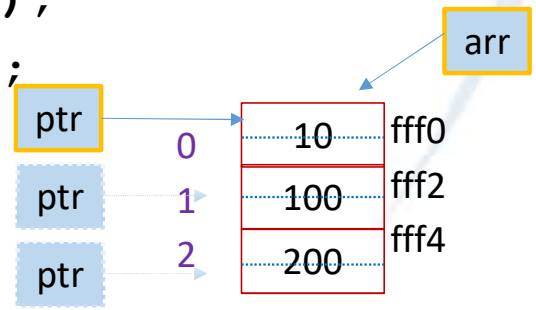
```
void main (void) {
    int  arr[10], i, *ptr;
    ptr = arr;
    for ( i = 0; i < 10; i++)
        scanf("%d", arr+i );
    for ( i = 0; i < 10; i++) {
        printf("Value of arr[%d] = %d\n", i, ptr[i] );
    }
}
```

6.4 Pointer Comparisons

- Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is *&arr[2]*

6.4 Pointer Comparisons

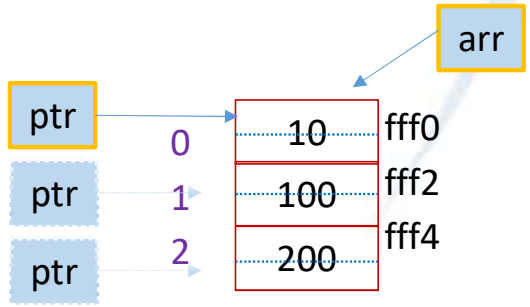
```
#include <stdio.h>
void main (void) {
    int  arr[] = {10, 100, 200, 400, 500};
    int  i=0, *ptr;
    /* let us have address of the first element in pointer */
    ptr = arr;
    while ( ptr <= &arr[2] ) {
        printf("Address of arr[%d] = %x\n", i, ptr );
        printf("Value of arr[%d] = %d\n", i, *ptr );
        /* point to the next location */
        ptr++;
        i++;
    }
```



6.4 Pointer Comparisons

• The output will be as follow:

```
Address of arr[0] = fff0
Value of arr[0] = 10
Address of arr[1] = fff2
Value of arr[1] = 100
Address of arr[2] = fff4
Value of arr[2] = 200
```



6.5 NULL Value in pointers

- If the pointer has the NULL value, that is mean it is not pointing to any location in the memory; i.e. it is not allowable to deal with what it points to.
- If you try to deal with what pointer points to with NULL value, the program is terminated immediately with run-time exception called ***"Null Pointer Exception"*** .
- Ex:

```
int * ptr = NULL;
* ptr = 5;  // causes Null Pointer Exception
```

- It is important to check on the value of the pointer content if it is not NULL before trying to deal with what it points

6.6 Using Pointers to call by address: Passing pointers to functions in C

- C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Lets try to make a function to swap the values between two integers, lets try to thing make it call by value and discuss.

```
void swap (int A, int B)
{   int temp;
    temp = A;
    A = B;
    B = temp;
}
```

- What will happen when we call it with two variables are passed to it, see the following program.

6.6 Using Pointers to call by address: Passing pointers to functions in C

```
void main (void)
{   int x=5, y=7;
    printf("\nValues: x=%d, y=%",x,y);
    /* call swab function */
    swap(x, y);
    printf("\n Values after swapping: x=%d, y=%",x,y);
}
```

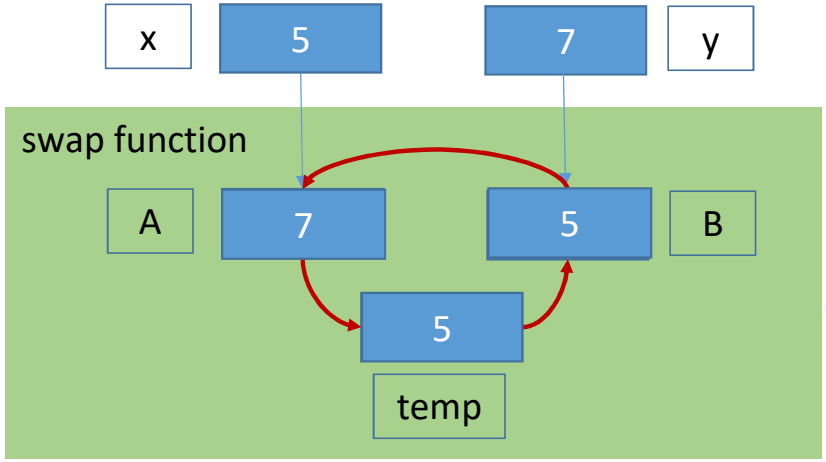
• The output will be:

```
Values: x=5, y=7
Values after swapping: x=5, y=7
```

6.6 Using Pointers to call by address: Passing pointers to functions in C

- Why this function not working? Actually it is working and the swapping is done, but between the copy of the variables not between the original variables, because we call the function *by value*, How?
- When we pass a variable as an argument to a function, a copy of the value of the variable is put in the input parameter; in different place as local variable for this function, so the working of the function execute on the local variable (a copy) not on the original variable

Function Call : `swap ( x , y)`



6.6 Using Pointers to call by address: Passing pointers to functions in C

- The solving of the above problem by using pointer to call the function and pass the address of the variables as arguments not the values of the variables; *Call by address*. As the following function: swap2

```
void swap2 (int * A, int * B)
{   int temp;
    temp = *A;
    *A = *B;
    *B = temp;
}
```

- The using of this function will be as follow:

6.6 Using Pointers to call by address: Passing pointers to functions in C

```
void main (void)
{   int x=5, y=7;
    printf("\nValues: x=%d, y=%",x,y);
    /* call swab function */
    swap2( &x, &y);
    printf("\n Values after swapping: x=%d, y=%",x,y);
}
```

• The output will be:

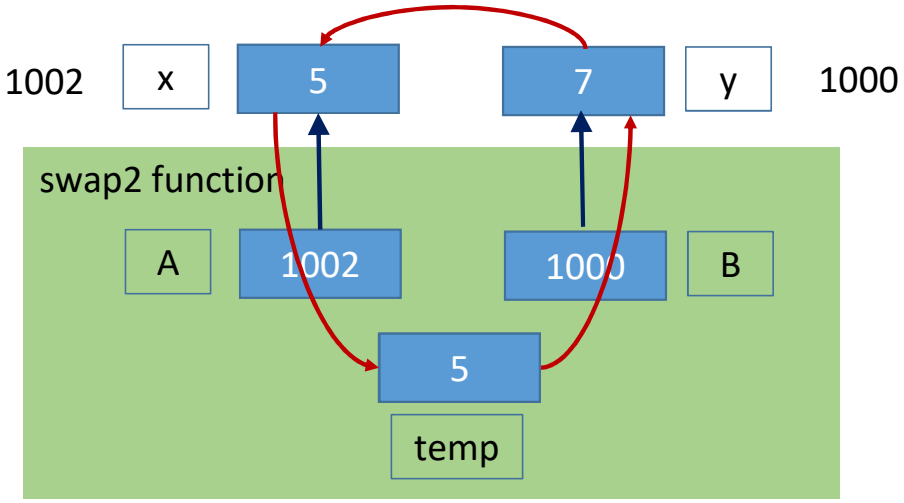
```
Values: x=5, y=7
Values after swapping: x=7, y=5   //Success result
```

• How it is working? Let's see ….

6.6 Using Pointers to call by address: Passing pointers to functions in C

- The following simulation will explain how it is working

```
void swap2 (int * A, int * B)
{    int temp;
     temp = *A;
     *A = *B;
     *B = temp;
}
```



Function Call : `swap2 ( &x , &y)`

6.7 Using pointers to pass an array to function

- As we show early in the lecture, the array name is a fixed or constant pointer to the first byte dedicated to the array, so we can pass the array name to a function which has an input parameter as pointer. For example we have the following function:

```
void printArray(int * ptr, int size)
{ int i;
     printf("\n The values of the array:");
    for (i=0; i<size; i++)
        printf("\n element No %d = %d",i+1, ptr[i]);
}
```

- You may call this function as follow:

```
    int arr[25]={2,4,6,8,10,12};
    printArr(arr, 25);  // or u may use less than size
```

6.7 Using pointers to pass an array to function

• Ex: Make a function to count the length of a string, like strlen().

```
int stringLength(char * str)
{
    int count=0;
    while(str[count]!='\0')
        count++;
    return count;
}
```

6.7 Using pointers to pass an array to function

• Ex: using the abov function

```
void main(void)
{
    char name[]={"Hassan Aly Mohamed"}; // 18 character
    printf("The lenth = %d" , stringLength(name));

}
```

6.8 Return a pointer from a function

- You may return a pointer datatype from a function if you want to return an address for a data but take in consideration this data – which you want to return its address – must be exist in memory after function return, i.e. if you return an address of a local data type it must be declared as ***static local variable*** (created at the first call to the function in heap section not in stack section and it will be exist until the program is terminated)
- Ex: A function to search a data in array and return its address:

6.8 Return a pointer from a function

```
int * searchAnInt(int * ptr, int size, int data)
{
    int i;
    for (i=0 ; i<size ; i++)
        if( ptr[i] == data )
            return ptr+i;

    return NULL;
}
```

6.8 Return a pointer from a function

```
void main (void)
{   int * add;
    int n, arr[10]={22, 3, 5, 88, -12, 6, 99, -3, 25, 169};
    printf("\n Enter a number to search it");
    scanf("%d",&n);
    add = searchAnInt(arr, 10, n);
    if(add==NULL) printf("\n The data u entered not exist");
    else printf("\n dat is exist with add: %x", add);
}
```

6.9 Pointer to a structure

- We have already learned that a pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
struct Employee e;
struct Employee * sptr;
sptr = &e;
```

6.9.1 Accessing structure members using Pointer

- There are two ways of accessing members of structure using pointer:
  - Using indirection (*) operator and dot (.) operator.
  - Using arrow (->) operator or membership operator.
- **Using Indirection (*) Operator and Dot (.) Operator**

```
(*sptr).ID =  23;
gets((*sptr).Name);
```

- **Using arrow operator (->)**

```
sptr -> ID = 23;
gets( sptr -> Name);
```

6.10 Dynamic Allocation

- One of the important using of pointers to make the dynamic allocation; allocate some of memory bytes at the runtime and deal with it. What is the dynamic allocation?
- As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as *dynamic memory allocation* in C programming.
- To allocate memory dynamically, library functions are *malloc()*, *calloc()*, *realloc()* and *free()* are used. These functions are defined in the <stdlib.h> header file and <alloc.h>.

6.10.1 malloc() Function

- The name "malloc" stands for memory allocation.
- The ***malloc()*** function reserves a block of memory of the specified number of bytes. And, it returns a ***pointer to void*** which can be ***casted*** into pointers of any form.
- Syntax of malloc()

```
ptr = (castType*) malloc(size in bytes);
```

- Ex:

```
ptr = (int*) malloc(100); // allocate 100 bytes
fptr = (float*) malloc(50 * sizeof(float));
```

- The second statement allocates 200 bytes of memory. It's because the size of float is 4 bytes. And, the pointer *fptr* holds the address of the first byte in the allocated memory.
- The expression results in a **NULL** pointer if the memory cannot be allocated.

6.10.2 calloc() Function

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to **zero**.
- Syntax of calloc():

```
ptr = (castType*)calloc(repetation_No, size);
```

- Ex:

```
fptr = (float*) calloc(25, sizeof(float));
```

- The above statement allocates contiguous space in memory for 25 elements of type float and all of it are initialized by 0.

6.10.3 free() Function

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.
- Syntax of free():

```
free(ptr);
```

- This statement frees the space allocated in the memory pointed by ptr.

6.10.4 realloc() Function

- The name “realloc" stands for reallocation
- If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.
- Syntax of realloc():

```
ptr = realloc(ptr, New_Size);
```

- Here, ptr is reallocated with a New_Size bytes.

6.10.5 Examples on Dynamic Memory Alocation

```
void main(void)
{
    int * ptr, N;
    scanf("%d",&N);
    ptr= (int *)malloc(N*sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

6.10.5 Examples on Dynamic Memory Alocation

```
void main(void)
{
    int * ptr, N;
    scanf("%d",&N);
    ptr= (int *)calloc(N, sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

6.10.5 Examples on Dynamic Memory Allocation

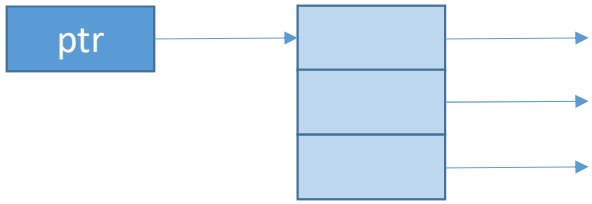
```
void main(void)
{   int * ptr, N, i;
    scanf("%d",&N);
    ptr= (int *)malloc(N*sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N ; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    ptr= (int *)realloc(ptr,(N+5)*sizeof(int));
    for(i=0; ptr!=NULL&&i<(N+5) ; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

6.11 Array of Pointers

- There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available.
- Following is the declaration of an array of pointers to an integer :

```
int * ptr[3];
```

- It declares ptr as an array of 3 integer pointers. Thus, each element in ptr, holds a pointer to an int value.

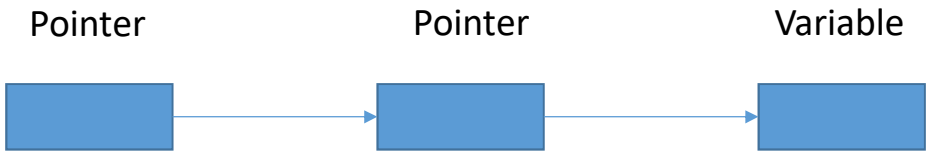


6.11.1 Example on array of pointers

```
void main(void)
{   char * strs[3];
    int i;
    for(i=0; i<3; i++)
        strs[i] = (char *)malloc(11 * sizeof(char));
    gets(strs[1]);
    strcpy(strs[0], "Hello");
    strcpy(strs[2], "In ITI");
    for(i=0; i<3; i++) printf("%s ",strs[i]);
}
```

6.12 Pointer to Pointer

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



6.12.1 Examples

```
void main (void) {
   int  var = 3000;
   int  *ptr;
   int  **pptr;
        /* take the address of var */
   ptr = &var;
        /* take the address of ptr using address of operator &
*/
   pptr = &ptr;
        /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
}
```

6.12.1 Examples

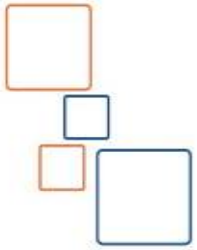
```
void main (void) {
   int i, j, N, M, **pptr;
   scanf("%d",&N);        scanf("%d",&M);
   pptr = (int **) malloc (N * sizeof(int *));
   for(i=0; i<N; i++)
        pptr[i] = (int *) malloc(M * sizeof(int));
   for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            scanf("%d", &pptr[i][j]);
 for(i=0; i<N; i++)
    for(j=0; j<M; j++)
        printf("Value of pptr[%d][%d] = %d\n",i,j, pptr[i][j]);
}
```

6.13 The standard header of the main function in C

```
int main (int argc, char ** argv)
```

- The main may return a value to the operating system to indicate the termination of the program done normally or there is an abnormal situation; 0 represent normal termination, other values means abnormal termination
- The operating system may pass arguments to the main, so the ***number of arguments*** is the first argument so it is an integer, and the ***arguments values*** is the second parameter as an array of strings – two dimensional array of character- so it is pointer to pointer to character.
- The first argument always is the name of the program you are running.

Lab Exercise

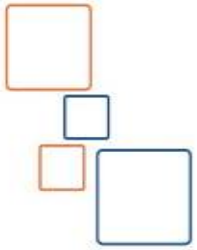


Invest In Yourself,
Develop Your Career

Assignments

- Write a swap function using call by address.
- Using a function to read an array from user, and another one to print it.
- Write a function to make a string copy from source string to destination string. (like strcpy())
- Write a program to make an array of Employees with size determined at run time from a user and read its data and print the ID and net salary for each employee.
- Write a program to make an array of strings which number of strings is read from the user and so the size of each string is read from the user, after that read all strings and print it.

Miscellaneous topics in C



Content

- Static local variables
- Modifiers and access specifiers
- Enumerate datatype
- What is void data type in C
- Union datatype

Static Local Variable

- It is a variable declared inside a function with modifier ***static***, which make the variable to be allocated in the heap section instead of stack section.
- The life time of the static local variable is start at the first call of the function after the program execute, and end with the termination of the program.
- The scope of it inside the owner function, i.e. it is not accessible outside the function, except if and only if you got its address and then you may access it outside its scope.
- After first call of a function includes a static variable, the static local variable is already exist in memory, so when you recall the function again, the static local variable will not be allocated again and it includes the last value from previous call.
- The following simple example illustrates a static local variable work as a counter for the number of calling the function and return that counting after each calling.

Static Local Variable

```
int doSomeThing(void)
{   static int count = 1;
    …
    return count;
}
void main(void){
    int i=0;
    for( ; i<10; i++)
        printf("\n%d", doSomeThing());
}
```

Modifiers and Qualifiers in C Language

- Modifiers are keywords in c which changes the meaning of basic data type in c.
- Modifier specifies the amount of memory space to be allocated for a variable.
- Modifiers are prefixed with basic data types to modify the memory allocated for a variable.
- Qualifiers are keywords which are used to modify the properties of a variable are called type qualifiers.

Modifiers and Qualifiers in C Language

- Types of modifiers and qualifiers can categorized as follow:
  - Signing: **signed** and **unsigned**; all primitive data types are signed by default except character, it is unsigned by default
  - Sizing: **short**, **long;** the size of integer datatype is dependent on the platform work on it (2 or 4 bytes) is the short size so the long will be 4 or 8 bytes.
  - Variable modifiers: **auto**, **extern**, **static**; auto means auto allocated and auto deallocated; local variables,

  extern means the variable (or the function) will find else where; i.e. in another file, and static used to make static local variable, if it is used with global variables, that means its scope within the file include it only.
  - Pointers Modifiers: **near**, **far**, **huge**; to represent the level of accessibility of the pointer: within the memory allocated to the program or out – like physical address- huge like far but more accurate.
  - Qualifiers: **volatile**, **const;** volatile variable may be accessed by background routine.
  - Interruption: **interrupt**; used to represent a function called as interrupt service routine or part of it.

Enumeration Datatype in C

- **What is *enum* data type in C**
- Enumeration Types are a way of creating your own Type in C.
- It is a user-defined data type consists of integral constants and each constant is given a name.
- The keyword used for an enumerated type is ***enum***.
- The enumerated types can be used like any other data type in a program.
- Here is the syntax of declaring an ***enum***

```
enum identifier{ value1, value2,...,valueN };
```

```
enum WeekDays{ Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

- Now any variable of enum days can take any one of the seven values.

```
enum WeekDays holiday = Friday;
```

- Here, *holiday* is a variable of data type ***enum WeekDays*** and is initialized with value Friday.

What is void data type in C?

- The **void data type** is an empty data type that refers to an object that does not have a value of any type. Here are the common uses of **void data type**. When it is used as a function return type.

```
void myFunction(int i);
```

  - Void return type specifies that the function does not return a value.
- When it is used as a function's parameter list:

```
int myFunction(void);
```

  - Void parameter specifies that the function takes no parameters.

- When it is used in the declaration of a pointer variable:

```
void *ptr;
```

  - It specifies that the pointer is "universal" and it can point to anything. When we want to access data pointed by a void pointer, first we have to type cast it.

What is Union Datatype in C?

- A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.
- **Defining a Union**
- To define a union, you must use the ***union*** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

```
union [union tag] {
    member definition;
        ...
    member definition;
} [one or more union variables];
```

- each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

The size of Union Datatype

- The following example illustrate that:

```
union Data {
    int i;
    float f;
    char str[20];
};
void main(void) {
    union Data data;
    printf("Memory size occupied by data:%d\n", sizeof(data));
}
```

- The output: `Memory size occupied by data : 20`

Accessing Union Members

- To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program using the above union defined.

Accessing Union Members

```
void main(void) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
}
```

- The ouput:

```
data.i : 19178
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Accessing Union Members

- Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.
- Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions

Accessing Union Members

```
void main(void) {
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i);
   data.f = 220.5;
   printf( "data.f : %f\n", data.f);
   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
}
```

- The output:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

With My Best Wishes

Ahmed Loutfy

Invest In Yourself,
Develop Your Career