F.Y. B.Sc.(IT) : Sem. I

Imperative Programming

Time : 2½ Hrs.] Prelim Question Paper Solution **[Marks : 75**

Q.1 Attempt the following (any THREE) [15]

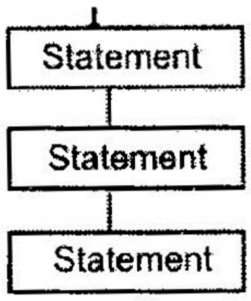
Q.1(a) List various techniques for the development of a program? Explain any one with suitable example. [5]

Ans.: Software designing is very anesthetic phase of software development cycle. The beauty of heart, skill of mind and practical thinking is mixed with system objective to implement design. Here are some approaches :

- Structural Programming
- Modular Designing
- Top Down Designing
- Bottom Up Designing
- Object Oriented Programming

1. Structural Programming : The program is divided into several basic structures. These structures are called building blocks. These are following :

(a) Sequence Structure : This module contains program statements one after another. This is a very simple module of Structured Programming.



(b) Selection or Conditional Structure : The Program has many conditions from which correct condition is selected to solve problems. These are (a) if-else (b) else-if, and (c) switch-case

(c) Repetition or loop Structure : The process of repetition or iteration repeats statements blocks several times when condition is matched, if condition is not matched, looping process is terminated.

2. Modular Programming: When we study educational philosophy, the concept of modulation can be clear without any ambiguity. Rene Descartes (1596-1650) of France has given concept to reconstruct our knowledge by piece by piece. The piece is nothing, but it is a module of modem programming context.

In modular approach, large program is divided into many small discrete components called Modules. Inprogramming language, different names are used for it.

Modules are debugged and tested separately and combined to build system. The top module is called root or boss modules which charges control over all sub-modules from top to bottom. The control flows from top to bottom, but not from bottom to top.

3. Top down Approach :

(a) The large program is divided into many small module or subprogram or function or procedure from top to bottom.

(b) At first supervisor program is identified to control other sub modules. Main modules are divided into sub modules, sub-modules into sub-sub-modules. The decomposition of modules is continuing whenever desired module level is not obtained.

(c) Top module is tested first, and then sub-modules are combined one by one and tested.

4. Bottom up Approach :

- In this approach designing is started from bottom and advanced stepwise to top. So, this approach is called Bottom up approach.
- At first bottom layer modules are designed and tested, second layer modules are designed and combined with bottom layer and combined modules are tested. In this way, designing and testing progressed from bottom to top.

- In software designing, only pure top down or Bottom up approach is not used. The hybrid type of approach is recommended by many designers in which top down and bottom up, both approaches are utilized.

5. Object oriented programming :
In the object-oriented programming, program is divided into a set of objects. The emphasis given on objects, not on procedures. All the programming activities revolve around objects. An object is a real world entity. It may be airplane, ship, car, house, horse, customer, bank Account, loan, petrol, fee, courses, and Registration number etc. Objects are tied with functions. Objects are not free for walk without leg of functions. One object talks with other through earphone of functions. Object is a boss but captive of functions.

Q.1(b) Define keywords and identifiers in C language? Also differentiate between keywords and identifiers. [5]

Ans.: **C Keywords :** Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, int is a keyword that indicates 'money' is a variable of type integer.

C Identifiers : Identifier refers to name given to entities such as variables, functions, structures etc. Identifier must be unique. They are created to give unique name to a entity to identify it during the execution of the program. For example:

```
int money;
double accountBalance;
```

Here, money and accountBalance are identifiers.

Also remember, identifier names must be different from keywords. You cannot use int as an identifier because int is a keyword.

Difference Between Keyword and Identifier. Every language has keywords and identifiers, which are only understood by its compiler. Keywords are predefined reserved words, which possess special meaning. An identifier is a unique name given to a particular variable, function or label of class in the program.

Q.1(c) Determine if the following constants are valid in C : [5]
(i) 27,882 (ii) 0.8E8 (iii) "Name" (iv) "1.3e12" (v) 0xBCFDAL

Ans.: (i) 27,882 is INVALID (ii) 0.8E8 is VALID
(iii) "Name" is INVALID (iv) "1.3e12" is INVALID
(v) 0xBCFDAL is INVALID

Q.1(d) What is the difference between machine level language and high level language? [5]

Ans.: Machine language, or machine code, is the only language that is directly understood by the computer, and it does not need to be translated. All instructions use binary notation and are written as a string of 1s and 0s. A program instruction in machine language may look something like this:

```
10010101100101001111101010011011100101
```

A high-level language is a programming language that uses English and mathematical symbols, like +, -, % and many others, in its instructions. When using the term 'programming languages,' most people are actually referring to high-level languages. High-level languages are the languages most often used by programmers to write programs. Examples of high-level languages are C++, Fortran, Java and Python.

To get a flavor of what a high-level language actually looks like, consider an ATM machine where someone wants to make a withdrawal of $100. This amount needs to be compared to the account balance to make sure there are enough funds. The instruction in a high-level computer language would look something like this:

```
1.  x = 100
2.  if balance x:
3.  print 'Insufficient balance'
4.  else:
5.  print 'Please take your money'
```

This is not exactly how real people communicate, but it is much easier to follow than a series of 1s and 0s in binary code.

There are a number of advantages to high-level languages.

- The first advantage is that high-level languages are much closer to the logic of a human language.
- The second advantage is that the code of most high-level languages is portable and the same code can run on different hardware.

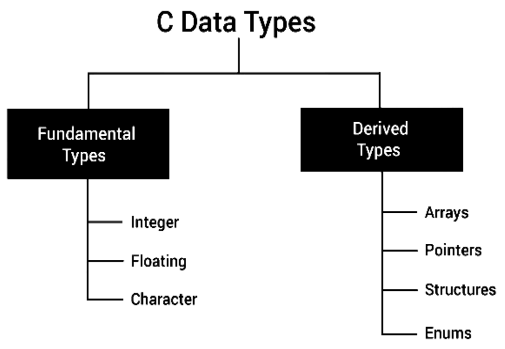
Q.1(e) What are the various data types in C? Explain them. [5]

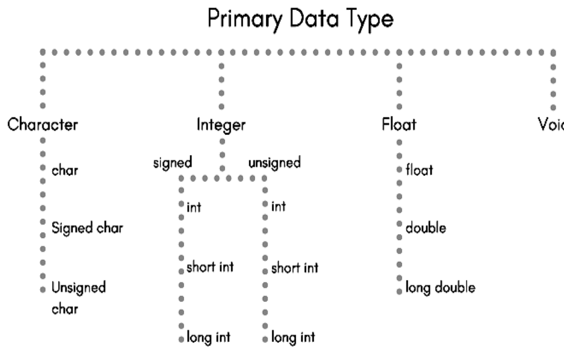
Ans.: Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types:

1. Primary data types : These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.
2. Derived data types : Derived data types are nothing but primary datatypes but a little twisted or grouped together like array, stucture, union and pointer. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.





Integer type : Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type : Floating types are used to store real numbers.
Size and range of Integer type on 16-bit machine :

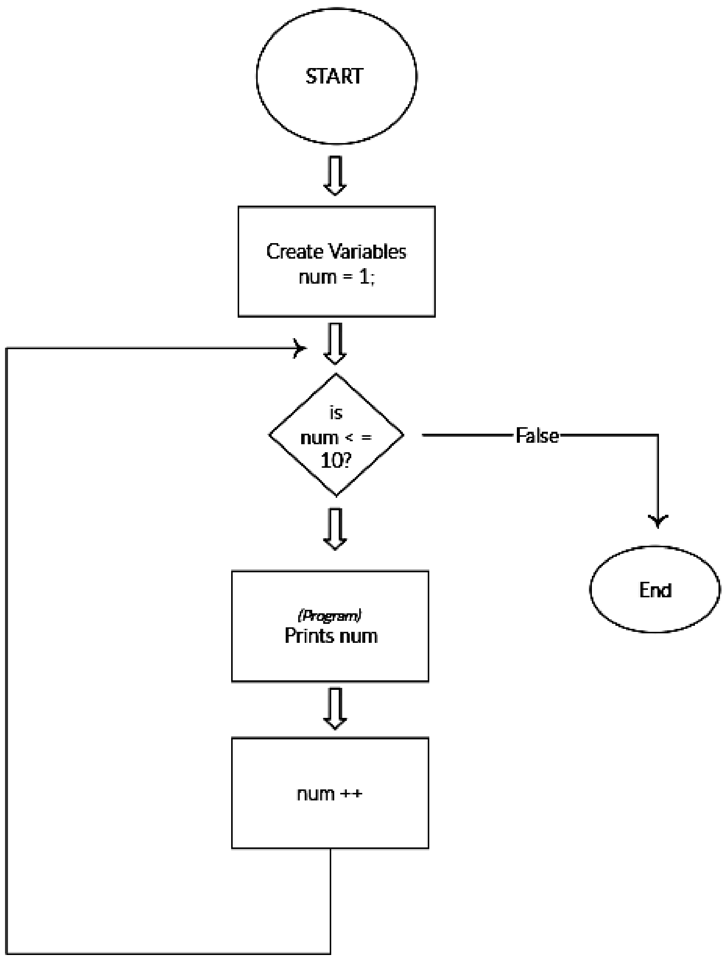
Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type : Character types are used to store characters value.
Size and range of Integer type on 16-bit machine :

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type : void type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

Q.1(f) Draw a flowchart to generate numbers from 1 to 10. **[5]**
Ans.:



Q.2 Attempt the following (any THREE) : **[15]**
Q.2(a) Explain the purpose and use of following operators with suitable examples : **[5]**
(i) == and = **(ii) Conditional Operator (?:)**
Ans.: (i) Assignment Operator (=) :
= is an Assignment Operator in C, C++ and other programming languages, It is Binary Operator which operates on two operands.
= assigns the value of right side expression's or variable's value to the left side variable.

Let's understand by example:

1. x=(a+b);
2. y=x;

Here, When first expression evaluates value of (a+b) will be assigned into x and in second expression y=x; value of variable xwill be assigned into y.

Equal To Operator (==) :

== is an Equal To Operator in C and C++ only, It is Binary Operator which operates on two operands.

== compares value of left and side expressions, return 1 if they are equal other will it will return 0.

Let's understand by example:

1. int x,y;
2. x=10;
3. y=10;
4. if(x==y)
5. printf("True");
6. else
7. printf("False");

When expression x==y evaluates, it will return 1 (it means condition is TRUE) and "TRUE" will print.

Conclusion : So it's cleared now, both are not same, = is an Assignment Operator it is used to assign the value of variable or expression, while ==is an Equal to Operator and it is a relation operator used for comparison (to compare value of both left and right side operands).

(ii) Conditional Operators [?:] : Ternary Operator Statement in C

1. They are also called as Ternary Operator.
2. They also called as ?: operator
3. Ternary Operators takes on 3 Arguments

   Syntax :

   expression 1 ? expression 2 : expression 3

   where, expression1 is Condition
   expression2 is Statement Followed if Condition is True
   expression3 is Statement Followed if Condition is False

   Example : Check whether Number is Odd or Even

```
#include<stdio.h>
int main()
{
    int num;
    printf("Enter the Number : ");
    scanf("%d",&num);
    (num%2==0)?printf("Even"):printf("Odd");
}
```

Q.2(b) C program contains the following variable declarations : **[5]**

float a = 2.5, b = 0.0005, c = 3000;

Show the output from following printf statements

(i) printf("%f %f %f", a, b, c); **(ii) printf("%3f %3f %3f", a, b, c);**

(iii) printf("%8f %8f %8f", a, b, c); **(iv) printf(8.4f f%8.4f %8.4f", a, b, c);**

(v) printf("%e %e %e", a, b, c);

Ans.: (i) 2.500000 0.000500 3000.000000
(ii) 2.500000 0.000500 3000.000000
(iii) 2.500000 0.000500 3000.000000
(iv) Error
(v) 2.500000e+000 5.000000e-004 3.000000e+003

Q.2(c) Write an interactive C program to find roots of a quadratic equation $ax^2 + bx + c = 0$ and roots are given by $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. **[5]**

Ans.:

```
#include<stdio.h>
#include<math.h>
void main()
{
float A,B,C,R1,R2,PART,EQN;
clrscr();
printf("ENTER THE VALUE OF A : ");
scanf("%f",&A);
printf("ENTER THE VALUE OF B : ");
scanf("%f",&B);
printf("ENTER THE VALUE OF C : ");
scanf("%f",&C);
EQN=pow(B,2)-(4*A*C);
if(EQN<0)
{
   PART=sqrt(abs(EQN));
   PART=-PART;
}
else
{
  PART=sqrt(EQN);
}
R1=(-B+PART)/(2*A);
R2=(-B-PART)/(2*A);
printf("THE CALCULATED ROOTS ARE : %.2f AND %.2f",R1,R2);
getch();
}
```

Q.2(d) Write a program in C to solve the following expression F = P(1 + i)n. **[5]**

Ans.:

```
#include<stdio.h>
#include<math.h>

void main()
{
    int i,v,p,n,a;
    float r;
    char ch;
    while(ch != 'n')
    {
        printf("\nEnter the value of p ::  ");
        scanf("%d",&p);

        printf("Enter the value of r ::  ");
        scanf("%f",&r);
        printf("Enter the value of n ::  ");
        scanf("%d",&n);

        a = p * (1 + r);
        v = pow(a,n);

        printf("Value of v is ::  %d",v);

        printf("\nDo you want to continue  ? ");

        ch = getchar();
    }
}
```

Q.2(e) Explain gets and printf statements used in C programming language. **[5]**

Ans.: gets()

Reads characters from the standard input (stdin) and stores them as a C string into str until a newline character or the end-of-file is reached.

- It is not safe to use because it does not check the array bound.
- It is used to read string from user until newline character not encountered.

Example : Suppose we have a character array of 15 characters and input is greater than 15 characters, gets() will read all these characters and store them into variable. Since, gets() do not check the maximum limit of input characters, so at any time compiler may return buffer overflow error.

```
// C program to illustrate
// gets()
#include <stdio.h>
#define MAX 15

int main()
{
    char buf[MAX];

    printf("Enter a string: ");
    gets(buf);
    printf("string is: %s\n", buf);

    return 0;
}
```

printf() function in C language :

- In C programming language, printf() function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen.
- We use printf() function with %d format specifier to display the value of an integer variable.
- Similarly %c is used to display character, %f for float variable, %s for string variable, %lf for double and %x for hexadecimal variable.
- To generate a newline,we use "\n" in C printf() statement.

Note :

- C language is case sensitive. For example, printf() and scanf() are different from Printf() and Scanf(). All characters in printf() and scanf() functions must be in lower case.

Example #1: C Output

```
#include <stdio.h>                    //This is needed to run printf() function.
int main()
{
    printf("C Programming");          //displays the content inside quotation
    return 0;
}
```

Output :

```
C Programming
```

Q.2(f) Explain the increment and decrement operator in C with example. [5]

Ans.: C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example #1: Increment and Decrement Operators

```
// C Program to demonstrate the working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Output :

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Q.3 Attempt the following (any THREE) : [15]

Q.3(a) Differentiate between while and do while loop with suitable examples. When to use which Loop? [5]

Ans.:

Basis For Comparison	**while**	**do-while**

General Form	while (condition) {
	statements; //body of loop
	}
		do{
		.
		statements; // body of loop.
		.
		} while(Condition);

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

While loop example:
```
#include<stdio.h>
int main()
{
    int i=1;
    while(i<=10)
    {
        printf("\n%d",i);
        i++;
    }
    return 0;
}
```

Do-while loop example:
```
#include<stdio.h>
int main()
{
    int i=1;
    do
    {
        printf("\n%d",i);
        i++;
    }while(i<=10);
    return 0;
}
```

When to use which Loop?
A for loop is used where you know at compile time how many times this loop will execute.
A while loop is normally used in a scenario where you don't know how many times a loop will actually execute at runtime.
A do-while loop is used where your loop should execute at least one time.

For example consider a program which writes some text in a file until file size becomes 2KB. You can use while loop in this scenario like this.

Q.3(b) Predict the output of following C codes. **[5]**

(i)
```
int i;
for(i=0;i< =2, i++)
{
    switch(i)
    {
        case 1: printf ("%d", i);
        case 2: printf ("%d", i);
        default: printf ("%d", i);
    }
}
```

(ii)
```
Void exchange (int, int);
void main()
{
    int x=20, y=10;
    exchange(x, y);
        printf("%d, %d", y,x);
}
Void exchange (int x, int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
```

Ans.: (i) Output: Error
Correction line is:
for(i=0;i<=2, i++)…this will become for(i=0;i<=2; i++)

Now the output is:

```
011122
```

(ii) ERROR. Correction is
Void exchange (int x, int y) will become void exchange (int x, int y).
Now the output will be:

```
10, 20
```

Q.3(c) What is recursion? Write a recursive function to calculate factorial of a number. [5]

Ans.: 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.

```
void recursion() {
    recursion(); /* function calls itself */
}

int main() {
    recursion();
}
```

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.

```
#include<stdio.h>

long factorial(int);

int main()
{
    int n;
    long f;

    printf("Enter an integer to find its factorial\n");
    scanf("%d", &n);

    if (n < 0)
        printf("Factorial of negative integers isn't defined.\n");
    else
    {
        f = factorial(n);
        printf("%d! = %ld\n", n, f);
    }

    return 0;
}

long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return(n * factorial(n-1));
}
```

Q.3(d) Write a function in C to swap two integer variable using call by value and call by reference. **[5]**

Ans.: Swapping numbers using Call by Value in C :

```
void swap(int a, int b)
{
    int temp;

    temp = b;
    b = a;
    a = temp;
    printf("Values of a and b is %d  %d\n",a,b);
}
```

Swapping numbers using Call by Reference in C :

```
void swap(int *a, int *b)
{
    int temp;

    temp = *b;
    *b = *a;
    *a = temp;
}
```

Q.3(e) Explain the switch… case statement in C with an example. **[5]**

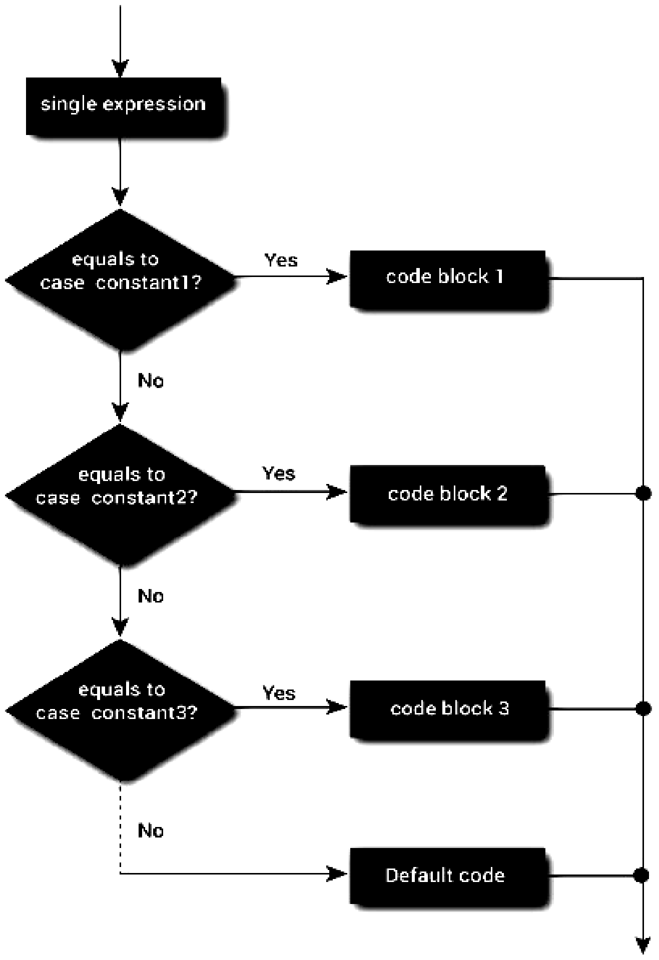
Ans.: A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax : The syntax for a switch statement in C programming language is as follows –

```
switch(expression) {
    case constant-expression  :
        statement(s);
        break; /* optional */

    case constant-expression  :
        statement(s);
        break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
    statement(s);
}
```

When a case constant is found that matches the switch expression, control of the program passes to the block of code associated with that case.

In the above pseudocode, suppose the value of n is equal to constant2. The compiler will execute the block of code associate with the case statement until the end of switch block, or until the break statement is encountered.
The break statement is used to prevent the code running into the next case.



Q.3(f) Write a program in C to generate the Fibonacci series (0, 1, 1, 2, 3, 5, 8,...) n terms using a while loop. **[5]**

Ans.:

```
#include <stdio.h>
int main()
{
    int i, n, t1 = 0, t2 = 1, nextTerm;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci Series: ");
    for (i = 1; i <= n; ++i)
    {
        printf("%d, ", t1);
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
    return 0;
}
```

Q.4 Attempt the following (any THREE) : **[15]**

Q.4(a) What do you understand from storage classes? List various storage classes? Explain any two. **[5]**

Ans.: Storage classes in C

In C language, each variable has a storage class which decides the following things:

- scope i.e where the value of the variable would be available inside a program.

- default initial value i.e if we do not explicitly initialize that variable, what will be its default initial value.
- lifetime of that variable i.e for how long will that variable exist.

The following storage classes are most oftenly used in C programming,

1. Automatic variables	2. External variables
3. Static variables	4. Register variables

Note: Explain any of the above two in detail:

Automatic variables: auto

- Scope: Variable defined with auto storage class are local to the function block inside which they are defined.
- Default Initial Value: Any random value i.e garbage value.
- Lifetime: Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function's execution is completed. Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.

```
#include<stdio.h>
void main()
{
    int detail;
    // or
    auto int details;    //Both are same
}
```

External or Global variable

- Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.
- Default initial value: 0(zero).
- Lifetime: Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a Global Variable. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero). One important thing to remember about global variable is that their values can be changed by any function in the program.

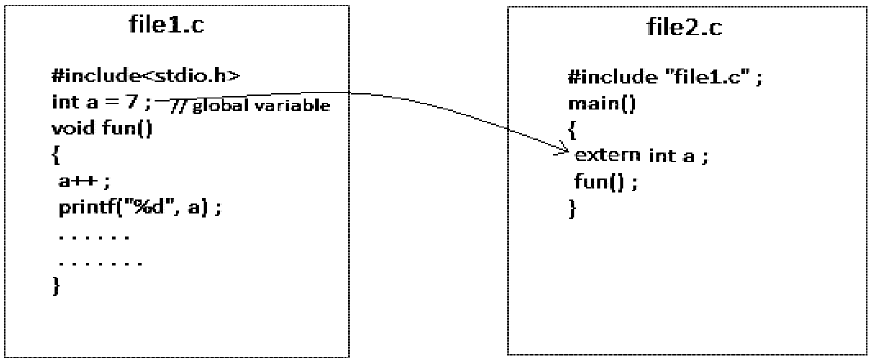
```
#include<stdio.h>
int number;    // global variable
void main()
{
    number = 10;
    printf("I am in main function. My value is %d\n", number);
    fun1();    //function calling, discussed in next topic
    fun2();    //function calling, discussed in next topic
}

/* This is function 1 */
fun1()
{
    number = 20;
    printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */
fun2()
{
    printf("\nI am in function fun2. My value is %d", number);
}
```

```
I am in function main. My value is 10
I am in function fun1. My value is 20
I am in function fun2. My value is 20
```

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

Note: Declaring the storage class as global or external for all the variables in a program can waste a lot of memory space because these variables have a lifetime till the end of the program. Thus, variables, which are not needed till the end of the program, will still occupy the memory and thus, memory will be wasted.

extern keyword :
The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.



file1.c

```
#include<stdio.h>
int a = 7 ; // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 . . . . . .
 . . . . . . .
}
```

file2.c

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

global variable from one file can be used in other using **extern** keyword.

Static variables :

- Scope: Local to the block in which the variable is defined
- Default initial value: 0(Zero).
- Lifetime: Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, staticvariable is initialized only once and remains into existence till the end of the program. A staticvariable can either be internal or external depending upon the place of declaration. Scope of internal static variable remains inside the function in which it is defined. External static variables remain restricted to scope of file in which they are declared.

They are assigned 0 (zero) as default value by the compiler.

```
#include<stdio.h>
void test();    //Function declaration (discussed in next topic)
int main()
{
    test();
    test();
    test();
}
void test()
{
    static int a = 0;      //a static variable
    a = a + 1;
    printf("%d\t",a);
}
```

```
1 2 3
```

Register variable :

- Scope: Local to the function in which it is declared.
- Default initial value: Any random value i.e garbage value
- Lifetime: Till the end of function/method block, in which the variable is defined.

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time.
Note: We can never get the address of such variables.

Syntax :

```
    register int number;
```

Note: Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU are limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

Q.4(b) Write a C program to find largest number out of given n numbers stored in an array using a function. [5]

Ans.:

```
#include <stdio.h>
int main()
{
    int i, n;
    float arr[100];

    printf("Enter total number of elements(1 to 100): ");
    scanf("%d", &n);
    printf("\n");
    // Stores number entered by the user
    for(i = 0; i < n; ++i)
    {
        printf("Enter Number %d: ", i+1);
        scanf("%f", &arr[i]);
    }

    // Loop to store largest number to arr[0]
    for(i = 1; i < n; ++i)
    {
        // Change < to > if you want to find the smallest element
```

```
            if(arr[0] < arr[i])
                arr[0] = arr[i];
        }
        printf("Largest element = %.2f", arr[0]);
        return 0;
    }
```

Q.4(c) What is a macro? Write a program in C to find the area of a rectangle and square using macros. **[5]**

Ans.: **Macros using #define**

You can define a macro in C using #define preprocessor directive.

A macro is a fragment of code that is given a name. You can use that fragment of code in your program by using the name. For example,

```
#define c 299792458  // speed of light
#include<stdio.h>
#include<conio.h>
#define AREA l*b
#define PERIMETER 2*(l+b)
main()
{
    int l,b;
    printf("Enter the length and bredth of the rectangle:\n");
    scanf("%d%d",&l,&b);
    printf("Area = %d",AREA);
    printf("Perimeter = %d",PERIMETER);
    printf("Press any key to continue...");
    getch();
}
```

Output :

```
Enter the length and breadth of the rectangle:
15
6
Area = 90
Perimeter = 42
Press any key to continue...
```

Q.4(d) Explain strlen, strcat, strcmp functions with example. **[5]**

Ans.: **strlen( )**

This function counts the number of characters present in a string. Its usage is illustrated in the following program :

```
main( )
{
char arr[ ] = "Bamboozled";
int len1, len2;
len1 = strlen(arr);
len2 = strlen("Humpty Dumpty");
printf("\nstring = %s length = %d", arr, len1);
printf("\nstring = %s length = %d", "Humpty Dumpty", len2);
}
```

The output would be …

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

While calculating the length it doesn't count '\0'. Even in the second call,
len2 = strlen("HUmpty Dumpty");
what gets passed to strlen( ) is the address of the string and not the string itself.

strcat( )

This function concatenates the source string at the end of the target string. For example, "Bombay" and "Nagpur" on concatenation would result into a string "BombayNagpur". Here is an example of strcat( ) at work.

```
main( )
{
char source[ ] = "Folks!";
char target[30] = "Hello";
strcat(target, source);
printf("\nsource string = %s", source);
printf("\ntarget string = %s", targate);
}
```

And here is the output ...

```
source string = Folks!
target string = HellowFolks!
```

Note that the target string has been made big enough to hold the final string.

strcmp( )

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, strcmp( ) returns a value zero. It they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts strcmp( ) in action.

```
main( )
{
char string1[ ] = "Jerry";
char string2[ ] = "Ferry";
int i, j, k;
i = strcmp (string1, "Jerry");        /1! = 0
j = strcmp (string1, string2);
k = strcmp (string1, "Jerry boy");
printf("\n%d %d %d", i, j, k);
}
```

And here is the output ...

```
0.4-32
```

The value returned is −32, which is the value of null character minus the ASCII value of space, i.e., '\0' minus ' ', which is equal to −32.

The exact value of mismatch will rarely concern us. All we usually want to know is whether or note the first string is alphabetically before the second string. If it is, a negative value is returned; if it isn't, a positive value is returned. Any non-zero means there is a mismatch.

Q.4(e) What are preprocessor directives in C? Explain #include and #define in C. [5]

Ans.: *C preprocessor directives :*

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

#include

```
#include <header name>
```

The include directive instructs the preprocessor to paste the text of the given file into the current file. Generally, it is necessary to tell the preprocessor where to look for header files if they are not placed in the current directory or a standard system directory. This can be done either at compile time or as part of your compiler's project file. This feature is implementation-specific, so see the compilers page for more information.
If an included file cannot be found, compilation will cease with an error.

#define
The #define directive takes two forms: defining a constant and creating a macro.
Defining a constant

```
#define token [value]
```

When defining a constant, you may optionally elect not to provide a value for that constant. In this case, the token will be replaced with blank text, but will be "defined" for the purposes of #ifdefand ifndef. If a value is provided, the given token will be replaced literally with the remainder of the text on the line. You should be careful when using #define in this way; see this article on the c preprocessor for a list of gotchas.

Defining a parameterized macro

```
#define token(<arg> [, <arg>s ... ]) statement
```

For instance, here is a standard way of writing a macro to return the max of two values.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Note that when writing macros there are a lot of small gotchas; you can read more about it here: the c preprocessor. To define a multiline macro, each line before the last should end with a \, which will result in a line continuation.

Q.4(f) What are two dimensional arrays in C? How can they be declared and initialized in C? [5]

Ans.: An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns. Before we discuss more about two Dimensional array lets have a look at the following C program.
In C programming, you can create an array of arrays known as multidimensional array. For example,

```
float x[3][4];
```

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	**Column 2**	**Column 3**	**Column 4**
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, The array y can hold 24 elements.
You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

How to initialize a multidimensional array?
There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array

```
// Different ways to initialize two dimensional array
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Q.5 Attempt the following (any THREE) : [15]

Q.5(a) What is a structure? How does a structure differ from an array? [5]

Ans.: Arrays allow to define type of variables that can hold several data items of the same kind. Similarly structure is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Structure :

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

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

Difference between Array and Structure :

Array and structure both are the container data type. The major difference between an array and structure is that an "array" contains all the elements of "same data type" and the size of an array is defined during its declaration, which is written in number within square brackets, preceded by the array name. A "structure" contains all the elements of "different data type", and its size is determined by the number of elements declared in a structure when it is defined.

Basis for comparison	Array	Structure
Basic	An array is a collection of variables of same data type.	A structure is a collection of variables of different data type.

Syntax	type array_name[size];	struct sruct_name{
		type element1;
		type element1;
		.
		.
		} variable1, variable2, . .;

Memory	Array elements are stored in contiguous memory location.	Structure elements may not be stored in a contiguous memory location.
Access	Array elements are accessed by their index number.	Structure elements are accessed by their names.
Operator	Array declaration and element accessing operator is "[ ]" (square bracket).	Structure element accessing operator is "." (Dot operator).
Pointer	Array name points to the first element in that array so, array name is a pointer.	Structure name does not point to the first element in that structure so, structure name is not a pointer.
Objects	Objects (instances) of an array can not be created.	Structure objects (instance or structure variable) can be created.
Size	Every element in array is of same size.	Every element in a structure is of different data type.
Bit filed	Bit filed can not be defined in an array.	Bit field can be defined in a structure.

Keyword	There is no keyword to declare an array.	"struct" is a keyword used to declare the structure.
User-defined	Arrays are not user-defined they are directly declared.	Structure is a user-defined datatype.
Accessing	Accessing array element requires less time.	Accessing a structure elements require comparatively more time.
Searching	Searching an array element takes less time.	Searching a structure element takes comparatively more time than an array element.

Q.5(b) Explain : (i) Pointer declaration **[5]**
(ii) '*' and "&" operators used with pointers

Ans.: **(i) Pointer declaration :**

- Pointers are the special type of data types which stores memory address (reference) of another variable. Here we will learn how to declare and initialize a pointer variable with the address of another variable?
- Pointer Declarations
- Pointer declaration is similar to other type of variable except asterisk (*) character before pointer variable name.
- Here is the syntax to declare a pointer
  data_type *poiter_name;

(ii) '*' and "&" operators used with pointers :

Address Operator (&) #

To find the address of a variable, C provides an operator called address operator (&). To find out the address of the variable marks we need to place & operator in front of it, like this:

&marks

The following program demonstrates how to use address operator (&).

```
// Program to demonstrate address(&) operator
#include<stdio.h>
int main()
{
    int i = 12;
    printf("Address of i = %u \n", &i);
    printf("Value of i = %d ", i);

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Address of i = 2293340
Value of i = 12
```

Dereferencing Pointer Variable *

Dereferencing a pointer variable simply means accessing data at the address stored in the pointer variable. Up until now, we have been using the name of the variable to access data inside it, but we can also access variable data indirectly using pointers. To make it happen, we will use a new operator called indirection operator (*). By placing indirection operator (*) before a pointer variable we can access data of the variable whose address is stored in the pointer variable.

```
int i = 100, *ip = &i;
```

Here ip stores address of variable i, if we place * before ip then we can access data stored in the variable i. It means following two statements does the same thing.

```
printf("%d\n", *ip); // prints 100
printf("%d\n", i); // prints 100
```

Q.5(c) Explain nested structure in C with example. [5]

Ans.: One structure can be nested within another structure. Using this facility complex data types can be created. The following program shows nested structures at work. main( )

```
{
    struct address
    {
        char phone[15] ;
        char city[25] ;
        int pin ;
    } ;
    struct emp
    {
        char name[25] ;
        struct address a ;
    } ;
    struct emp e = { "jeru", "531046", "nagpur", 10 };
    printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
    printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

And here is the output...

```
name = jeru phone = 531046
city = nagpur pin = 10
```

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

e.a.pin or e.a.city

Of course, the nesting process need not stop at this level. We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

maruti.engine.bolt.large.qty

This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a maruti car.

Q.5(d) Write a short note on pointer arithmetic in C. [5]

Ans.: A pointer is nothing more that a variable used to store a memory address. And if this is new to you go back and read Pointer Basics before continuing with this chapter. In this chapter, we will discuss arithmetic operations that can be performed on pointers.

We can't perform every type of arithmetic operations with pointers. Pointer arithmetic is slightly different from arithmetic we normally use in our day to day life. The only valid arithmetic operations applicable on pointers are:

1. Addition of integer to a pointer
2. Subtraction of integer to a pointer
3. Subtracting one pointer from another of the same type

The pointer arithmetic is performed relative to the base type of the pointer. For example: if we have an integer pointer ip which contains address 1000, then on incrementing it by 1, we will get 1004 (i.e 1000 + 1 * 4) instead of 1001 because the size of the int data type is 4 bytes. If we had been using a system where the size of int is 2 bytes then we would get 1002 (i.e 1000 + 1 * 2).

Similarly, on decrementing it we will get 996 (i.e 1000 - 1 * 4) instead of 999. So the expression ip + 4 will point to address 1016 (i.e 1000 + 4 * 4).

Pointer Arithmetic on Integers #

Pointer Expression	How it is evaluated?
ip = ip + 1	ip => ip + 1 => 1000 + 1*4 => 1004
ip++ or ++ip	ip++ => ip + 1 => 1004 + 1*4 => 1008
ip = ip + 5	ip => ip + 5 => 1008 + 5*4 => 1028
ip = ip - 2	ip => ip - 2 => 1028 - 2*4 => 1020
ip-- or --ip	ip => ip + 2 => 1020 + 2*4 => 1028

Pointer Arithmetic on Float #

Pointer Expression	How it is evaluated ?
dp + 1	dp = dp + 1 => 2000 + 1*8 => 2008
dp++ or ++dp	dp++ => dp+1 => 2008+1*8 => 2016
dp = dp + 5	dp => dp + 5 => 2016+5*8 => 2056
dp = dp - 2	dp => dp - 2 => 2056-2*8 => 2040
dp-- or --dp	dp => dp - 1=>2040-1*8=>2032`

Pointer Arithmetic on Characters #

Pointer Expression	How it is evaluated ?
cp + 1	cp = cp + 1 => 3000 + 1*1 => 3001
cp++ or ++cp	cp => cp + 1 => 3001 + 1*1 => 3002
cp = cp + 5	cp => cp + 5 => 3002 + 5*1 => 3007
cp = cp - 2	cp => cp + 5 => 3007 - 2*1 => 3005
cp-- or --cp	cp => cp + 2 => 3005 - 1*1 => 3004

Note: When we increment or decrement pointer variables using pointer arithmetic then, the address of variables i, d, ch are not affected in any way.

Q.5(e) Explain the terms "array of pointers" and "pointer to an array" in C. [5]

Ans.: Array of pointers :

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{
    char charArr[4];
    int i;

    for(i = 0; i < 4; ++i)
    {
        printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
    }
    return 0;
}
```

When you run the program, the output will be:

```
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47
```

Note: You may get different address of an array.

Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array charArr.
But, since pointers just point at the location of another variable, it can store any address.

Pointer to an array :
We can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.
Syntax:

```
    data_type (*var_name)[size_of_array];
```

Example:

```
    int (*ptr)[10];
```

Here ptr is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 10 integers'. Note : The pointer that points to the 0th element of array and the pointer that points to the whole array are totally different. The following program shows this:

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>

int main()
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];

    // Points to 0th element of the arr.
    p = arr;

    // Points to the whole array arr.
    ptr = &arr;
    printf("p = %p, ptr = %p\n", p, ptr);

    p++;
    ptr++;

    printf("p = %p, ptr = %p\n", p, ptr);

    return 0;
}
```

Run on IDE

Output:

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

Q.5(f) Explain how members of a structure are accessed by a variable and a pointer in C. [5]

Ans.: Accessing Structure Elements by a variable :

```
struct book
{
    char name[10] ;
    float price ;
    int pages ;
} ;
struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed. In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to pages of the structure defined in our sample program we have to use,

```
    b1.pages
```

Similarly, to refer to price we would use,

```
    b1.price
```

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

Accessing structure's member through pointer :

The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct. Such pointers are known as 'structure pointers'. Let us look at a program that demonstrates the usage of a structure pointer.

```
main( )
{
    struct book
    {
        char name[25] ;
        char author[25] ;
        int callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;
    struct book *ptr ;
    ptr = &b1 ;
    printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
    printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```

The first printf( ) is as usual. The second printf( ) however is peculiar. We can't use ptr.name or ptr.callno because ptr is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator ->, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the '.' structure operator, there must always be a structure variable, whereas on the left hand side of the '->' operator there must always be a pointer to a structure.

❑ ❑ ❑ ❑ ❑