© ALL RIGHTS RESERVED WITH THE PUBLISHERS

(All rights resolved, no part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher. Any violation thereof shall lead to legal recourse without any further notice.)

SYLLABUS

UNIT-I

C basics: C character set, Identifiers and keywords, Data types, constants, variab and arrays, declarations, expressions statements, symbolic constants, compou statements, arithmetic operators, unary operators, relational and logical operato assignment operators, conditional operators, bit operators.

C constructs: If statement, if...else statement, if....else if...else statement, wh statement, do...while statement, for statement, switch statement, nested control stateme break operator, continue operator, comma operator, goto statement. **[No. of Hrs: 1**

UNIT-II

C Functions: Functions: declaration, definition & scope, recursion, call by value, call reference.

Storage Classes: automatic, external (global), static & registers.

Arrays: Arrays, pointers, array & pointer relationship, pointer arithmetic, dynam memory allocation, pointer to arrays, array of pointers, pointers to functions, array of pointe to functions, Preprocessor directives: #include, #define, macro's with arguments, the operato # and ##, conditional compilations. **[No. of Hrs: 1**

UNIT-III

Structures: Structures, unions, passing structure to functions, bit fields, file handli [text (ASCII), binary]. **[No. of Hrs: 1**

UNIT-IV

String manipulation functions and other standard library functions from stdio, stdlib.h, conio.h, ctype.h, math.h, string.h, process.h. **[No. of Hrs: 1**

END TERM EXAMINATION
FIRST SEMESTER [BCA-105]
INTRODUCTION TO PROGRAMMING USING C
[DECEMBER-2013]

Time : 3 Hours **M.M. : 75**

Note: ***Attempt five questions including Q.no. 1 which is compulsory. Select one question from each unit.***

Q.1.(a) Write in brief about datatypes in C. **(2.5×10=25)**

Sol. A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement.

C language data types can be broadly classified as

- Primary data type
- Derived data type
- User defined data type

All C Compilers accept the following fundamental data types:

1. Integer	int
2. Character	Char
3. Floating Point	float
4. Double precision floating point	Double
5. Void	Void

Q.1.(b) Explain the use of ternary operator.

Sol. The conditional operator consists of 2 symbols the question mark (?) and the colon (:).

The syntax for a ternary operator is as follows:

exp1 ? exp2 : exp3

The ternary operator works as follows :

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

```
a = 10;
b = 15;
x = (a > b) ? a : b
```

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/* Example : to find the maximum value using conditional operator)
#include <stdio.h>
void main()                    //start of the program
{
int i,j,larger;                //declaration of variables
printf ("Input 2 integers : ");  //ask the user to input 2 numbers
scanf("%d %d",&i, &j);         //take the number from standard input and store it
larger = i > j ? i : j;        //evaluation using ternary operator
```

Scanned by CamScanner

```
printf("The largest of two numbers is %d \n", larger);   // print the largest number
}                    // end of the program
```

Q.1.(c) What are Literals in C?

Sol. The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**. Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal.*

Integer literals: An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

Floating-point literals: A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159        /* Legal */
314159E-5L     /* Legal */
510E           /* Illegal: incomplete exponent */
210f           /* Illegal: no decimal or exponent */
.e55           /* Illegal: missing integer or fraction */
```

Character literals: Character literals are enclosed in single quotes, e.g., **'x'** and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

String literals: String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
```


Q.1.(d) Write rules for naming identifiers.

Sol. Rules for Constructing Variable Names

(*a*) A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

(*b*) The first character in the variable name must be an alphabet or underscore.

(*c*) No commas or blanks are allowed within a variable name.

(*d*) No special symbol other than an underscore (as in **gross_sal**) can be used in a variable name.

Ex.: si_int
m_hra
pop_e_89

C compiler is able to distinguish between the variable names by making it compulsory for you to declare the type of any variable name that you wish to use in a program.

Q.1.(e) Explain use of relational operators.

Sol. Relational operators checks relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0. For example:

a>b

Here, > is a relational operator. If *a* is greater than *b*, *a>b* returns 1 if not then, it returns 0.

Relational operators are used in decision making and loops in C programming.

Operator	Meaning of Operator	Example
==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

A simple relational expression contains only one relational operator and takes the following form:

exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

```
6.5 <= 25        TRUE
-65 > 0          FALSE
10 < 7 + 5       TRUE
```

Q.1.(f) Write in brief about Recursion.

Sol. A recursive function is a function that calls itself. Some problems can be easily solved by using recursion, such as when you are dividing a problem into sub- problems with similar natures. Note that recursion is a time-consuming solution that decreases the speed of execution because of stack overheads. In recursion, there is a function call and the number of such calls is large. In each call, data is pushed into the stack and when the call is over, the data is popped from the stack. These push and pop operations are time-consuming operations. If you have the choice of iteration or recursion, it is

Scanned by CamScanner

better to choose iteration because it does not involve stack overheads. You can use recursion only for programming convenience.

Content of a Recursive Method:-

- Base case(s).
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls.
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Benefits of Recursion:

1. Although at most of the times a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system cannot be solved without recursion.
2. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
3. Using recursion, the length of the program can be reduced.

Program to find factorial of a number using recursion:-

```
#include <stdio.h>
int fact(int n);
main()
{
    int k = 4,i;
    i =fact(k);
    printf("The value of i is %d\n",i);
}
int fact(int n)
{
 if(n<=0)
    return(1);
 else
    return(n*fact(n-1));
}
```

Explanation

You can express factorials by using recursion as shown:

fact (5) = 5 * fact (4)

In general,

fact (N) = N * fact (N-1)

fact 5 is calculated as follows:

fact (5) = 5 * fact (4) i.e. there is call to fact (4)

fact (4) = 4 * fact (3)

fact (3) = 3 * fact (2)

fact (2) = 2 * fact (1)

fact (1) = 1 * fact (0)

fact (0) = 1

fact (1) = 1 * 1, that is the value of the fact(0) is substituted in 1.

fact (2) = 2 * 1 = 2

fact (3) = 3 * 2 = 6

fact (4) = 4 * 6 = 24

fact (5) = 5 * 24 = 120

Q.1.(*g*) Write & explain any 2 functions of process in header file.

Sol. 1. exit() :- exit terminates the program. exit terminates the calling process. Before termination, exit does the following:-

- closes all files.
- writes buffered output (waiting to be output).
- calls any registered "exit functions" (posted with at exit).

Typically a value of 0 indicates a normal exit, and a non-zero value indicates some error.

2. abort():- Abnormally terminates a process. abort() writes a termination message on stderr ("Abnormal program termination"), then aborts the program by a call to exit().

Q.1.(*h*) Write short note on enum.

Sol. Use of enumerated data type:

```
#include <stdio.h>
enum colours {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
    NUMBER_OF_COLOURS};
typedef enum colours colour_t;
int main(void)
{
    colour_t sky, forest;
    printf("There are %d colours in the enum\n", NUMBER_OF_COLOURS);
    sky = BLUE;
    forest = GREEN;
    printf("sky = %d\n", (int)sky);
    printf("forest = %d\n", (int)forest);
    return 0;
}
```

The output of this program is:

There are 7 colours in the enum

sky = 4

forest = 3

Structure:

- Structure is a method of packing the data of different types.
- When we require using a collection of different data items of different data types in that situation we can use a structure.
- A structure is used as a method of handling a group of related data items of different data types.

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types.

Scanned by CamScanner

Nested Structure: When a structure is a member of another structure, it is called a *nested structure*. A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date
{
    int day;
    int month;
    int year;
};
struct student
{
    int id_no;
    char name[20];
    char address[20];
    char combination[3];
    int age;
    structure date def;
    structure date doa;
}oldstudent, newstudent;
```

Q.1.(*i*) Write important features of C.

Sol. There are several reasons why many computer professionals feel that C is at the top of the list:-

1. FLEXIBLILITY: C is a powerful and flexible language. C is used for projects as diverse as operating system, word processor, graphics, spreadsheets and even compilers for other language. C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.

2. PORTABILITY: C is a portable language. Portable means that a C program written for one computer system can be run on another system with little or no modification. Portability is enhanced by the ANSI standard by C, the set of rules for C compilers.

3. COMPACTNESS: C is a language of few words, containing only a handful terms, called keywords, which serve as the base on which the language's functionality is built. You might think that a language with more keyword would be more powerful. This isn't true. As you will find that it can be programmed to do any task.

4. REUSABILITY: C is modular, C code can and should be written in routine called functions. These functions can be reused in other applications or programs. By passing pieces of information to the function, you can create useful, reusable code.

Q.1. (*j*) Explain use of "extern" keyword.

Ans. extern - Storage Class:- extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to decalre a global variable or function in another files.

File 1: main.c

```
int count=5;
main()
{
write_extern();
}
```

File 2: write.c

```
void write_extern(void);
extern int count;
void write_extern(void)
{
printf("count is %i\n", count);
}
```

Here extern keyword is being used to declare count in another file.

UNIT I

Q.2.(*a*) Write a program to check whether a number is prime or not. (6.5)

Ans.

```
#include<stdio.h>
#include<conio.h>
void main()
{
inti,n;
clrscr();
printf("Enter the number\n");
scanf("%d", &n);
for(i=2;i<=n-1;i++)
    {
        if(n%i==0)
        {
            break;
        }
}
if(i==n)
{
printf(" Number is prime number");
}
else
{
printf(" Number is not a prime number");
}
getch();
}
```

Q.2.(*b*) Explain the purpose of various loop control structures in C. (6)

Ans. Basically , a loop involves repeating some portion of the program either a specified number of times ,or until a particular condition is being satisfied . In C, this repetitive method is achieved through a **while** or a **do- while loop** or **for loop**

1.while: The *while* loop is used to execute a block of code as long as some condition is true. If the condition is false from the start the block of code is not executed at all. The while loop tests the condition before it's executed so sometimes the loop may never be

Scanned by CamScanner

executed if initially the condition is not met. *while* loop is also known as **entry-controlled loop**. Its syntax is as follows:

```
while (tested condition is satisfied)
{
block of code
}
```

In all constructs, curly braces should only be used if the construct is to execute more than one line of code. Here is a simple example of the use of the while loop. This program counts from 1 to 100.

```
#include <stdio.h>
int main(void)
{
int count = 1;
while (count <= 100)
{
printf("%d\n",count);
count += 1;                    // Notice this statement
}
return 0;
}
```

2. do ....while

The do loop also executes a block of code as long as a condition is satisfied.

The "do" loop tests its condition after it's been executed at least once. Since the condition is tested at the bottom of a do loop, its block of code is always executed at least once.

The "do ....while" loops syntax is as follows

```
do{
block of code
} while (condition is satisfied);
```

Note that a semi-colon (;) must be used at the end of the do ...while loop. This semi-colon is needed because it instructs whether the while (condition) statement is the beginning of a while loop or the end of a do ...while loop.

Here is an example of the use of a do loop.

```
include<stdio.h>
int main(void)
{
int value, r_digit; printf("Enter a number to be reversed.\n");
scanf("%d", &value);
    do
    {
    r_digit = value % 10;
    printf("%d", r_digit);
    value = value / 10;
    } while (value != 0);
printf("\n"); return 0;
}
```


3. for loop: The for loop is probably the most popular looping instruction. In for loop the initialization, testing, and incrementation of loop counter done in a separate steps.

Condition is tested after executing body of the loop.

Here is an example of the use of a for loop.

```
# include<stdio.h>
int main()
{
int i ;
for(i=1;i<11;i++)
        printf("%d\n",i);
return 0;
}
```

Or

Q.3.(*a*) Differentiate between "break" and "continue" using example program. **(6)**

Ans. Break statement: When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. A **break** is usually associated with an **if**.

Example:

```
void main( )
{
    int num, i ;
    printf ( "Enter a number " ) ;
    scanf ( "%d", &num ) ;
    i = 2 ;
    while ( i <= num - 1 )
    {
        if ( num % i == 0 )
        {
        printf ( "Not a prime number" ) ;
        break ;
    }
    i++ ;
    }
  if ( i == num )
      printf ( "Prime number" ) ;
}
```

The keyword**break**, breaks the control only from the **while**in which it is placed.

Continue statement: continue statement is a jump statement. The continue statement can be used only inside for loop, while loop and do-while loop. Execution of these statement does not cause an exit from the loop but it suspend the execution of the loop for that iteration and transfer control back to the loop for the next iteration.

Example:

```
void main( )
{
int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
```

Scanned by CamScanner

```
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                continue ;
            printf ( "\n%d`%d\n", i, j ) ;
        }
    }
}
```

Q.3.(b) Write a program to generate following pattern: **(6.5)**

Ans.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, j;
int c=1;
clrscr();
for(i=1;i<=4;i++)
{
    for(j=1;j<=i;j++)
    {
    printf("%d",c++);
    }
printf("\n");
}
getch();
}
```

UNIT II

Q.4.(a) Write a program that prints transpose of a 3 × 3 matrix entered by the user. **6**

Sol.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[50][50],b[50][50],i,j,m,n;
clrscr();
printf("Enter the order of the matrix\n");
scanf("%d%d",&m,&n);
printf("Enter the matrix elements:\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&a[i][j]);
b[j][i]=a[i][j];
}
```



```
}
printf("Entered matrix was\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
printf("Transpose of the entered matrix:\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
printf("%d\t",b[i][j]);
}
printf("\n");
}
getch();
}
```

Q.4.(b) Write down benefits using functions in C. Also write a function to calculate factorial of a number. **(6.5)**

Sol. BENEFITS OF USER-DEFINED FUNCTIONS:

1. Modular Programming It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.

2. Reduction of source code The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. Easier Debugging It is easy to locate and isolate a faulty function for further investigation.

4. Code Reusability a program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.

5. Function sharing Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.

Function to calculate Factorial of a number:-

```
#include<stdio.h>
#include<conio.h>
long int factorial(int n)
{
 int i;
 long int fact=1;
 for(i=n;i>=1;i—) ,
 {
```

Scanned by CamScanner

```
fact=fact*i;
}
return fact;
}
void main()
{
long int result;
int n;
clrscr();
printf("Enter a no.:\n");
scanf("%d",&n);
result=factorial(n);
printf("Factorial of %d = %ld",n,result);
getch();
}
```

Or

Q.5.(a) Compare Automatic Storage class with static storage class in detail. (6)

Sol. A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class:- auto is the default storage class for all local variables.

```
{
int Count;
auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables. By default all local variables are of auto storage class.

register - Storage Class:- register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it might be stored in a register - depending on hardware and implimentation restrictions.

static - Storage Class:- static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
```



```
{
printf("%d\n", Road);
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initalised at run time but is not reinitalized when the function is called. This inside a function static variable retains its value during vairous calls.

```
void func(void);
static count=10; /* Global variable - static is the default */
main()
{
while (count—)
{
func();
}
}
void func( void )
{
static i = 5;
i++;
printf("i is %d and count is %d\n", i, count);
}
```

This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

Q.5.(b) Write a C program that returns reversal of a number passed to a function. Function is to be implemented using call by reference parameter passing technique. (6.5)

Sol.

```
#include<stdio.h>
#include<conio.h>
int reverse(int *n)
{
int rev=0;
while(*n!=0)
{
rev=rev*10+(*n)%10;
(*n)=(*n)/10;
}
```

Scanned by CamScanner

```
return rev;
}
void main()
{
int m,result;
clrscr();
printf("enter the no\n");
scanf("%d", &m);
result=reverse(&m);
printf("reverse =%d", result);
getch();
}
```

UNIT III

Q.6.(a) Explain the concept of structure within a structure using complete program. **(6.5)**

Sol. Consider the student information name, roll no, DOB and avg. The DOB consists of day, month, and year. It can be defined as shown below,

```
struct data
{
    int day;
    int month;
    int year;
};
struct student
{
    char name [10];
    int roll_number;
    struct data dob;
    int marks [3];
    float avg;
} s1;
```

Note that the above structure consists of a member identified by dob whose type is struct data.

The members contained in the inner structure namely day, month and year can be referred to as

s1.dob.day
s1.dob.month
s1.dob.year

An inner-most member in a structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most).

Below program illustrates nesting of structures:-

```
#include <stdio.h>
struct student
{
  char name [10];
  int rno;
  float avg;
```



```
  struct data
  {
    int day;
    int month;
    int year;
  } dob;
};
int main ( )
{
  struct student s1;
printf ("/n                    Enter details of student");
print f ("/n Enter student Name : ");
gets (s1.name);
printf ("Marks and average");
scanf ("%d%f", &sl.rno, &sl.avg);
printf ("Enter date of birth (dd/mm/yyyy)");
scanf ("%d%d"%d&s1.dob.day,&sl.dob.month,&sl.dob.year);
printf("/n        The details of student ");
printf ("/n Student Name : ");
puts (sl. name);
printf ("Marks=%d /n Average=% f ", sl.rno, sl. avg);
prinf ("/n The date of birth (dd/mm/yy)");
printf ("%d/%d/%d", sl.dob.day, sl.dob.month,sl.dob.year);
return ();
}
```

Q.6.(b) Write short note on each: **(3+3)**

(i) Array of pointers

Sol. ARRAY OF POINTERS:

- A pointer variable always contains an address; an array of pointers would be nothing but a collection of addresses.
- The addresses present in the array of pointers can be addresses of variables or addresses of array elements or any other addresses.
- The major application of this is in handling data tables efficiently and table of strings.
- All rules that apply to an ordinary array apply to the array of pointes as well.
- The Syntax for declaration of array of pointers as follows,

data type *arr_ptr [size];

- This declaration tells the compiler arr_ptr is an array of addresses, pointing to the values of data type.
- Then initialization can be done same as array element initialization. Example **arr_ptr [3] =&var**, will initialize 3rd element of the array with the address of var.
- The dereferencing operator is used as follows

***(arr_ptr [index])** —> will give the value at particular address.

Example

```
#include<stdio.h>
void main ()
{
```

Scanned by CamScanner

```
    int i=1, j=2, k=3, l=4, x;
    int *arr [4];//declaration of array of pointers
    arr [0] =&i;//initializing 0th element with address of i
    arr [1] =&j;//initializing 1st element with address of j
    arr [2] =&k;//initializing 2nd element with address of k
    arr [3] =&l;//initializing 3rd element with address of l
    for (x=0; x<4; x++)
        printf ("\n %d",*(arr[x]));
}
```

(ii) Array of structures

Sol. Array of Structures: As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of class of students, consisting of name, roll_number and marks, A better approach would be to use an array of structures.

Array of structures can be declared as follows,

struct tag_name arrayofstructure[size];

After having the above declaration a list of variables of type struct tag_name.

Let's take an example, to store the information of 3 students, we can have the following structure definition and declaration,

```
struct student
{
    char name[10];
    int rno;
    float avg;
};
struct student s[3];
```

Defines array called **s**, which contains three elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows,

struct student s[3]={{"ABC",1,56.7},{"xyz",2,65.8},{"pqr",3,82.4}};

Or

Q.7.(a) Write complete program to concatenate 2 strings using your own function. **(6.5)**

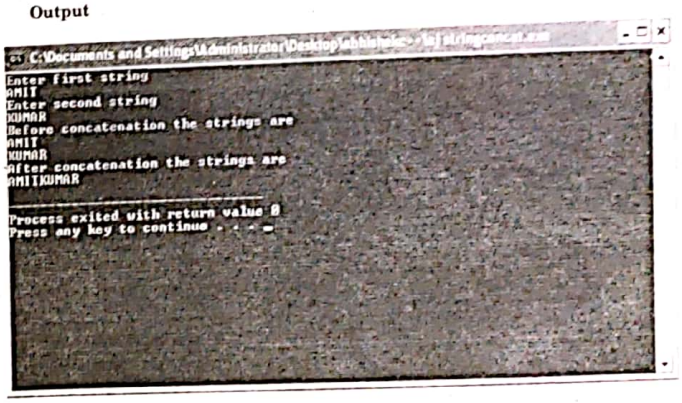
Sol.

```
#include<stdio.h>
int main()
{
 int i=0,j=0;
 char str1[20],str2[20];
 puts("Enter first string");
 gets(str1);
 puts("Enter second string");
 gets(str2);
 printf("Before concatenation the strings are\n");
 puts(str1);
 puts(str2);
 while(str1[i]!='\0')
```



```
 {
 i++;
 }
 while(str2[j]!='\0')
 {
 str1[i++]=str2[j++];
 }
 str1[i]='\0';
 printf("After concatenation the string is\n");
 puts(str1);
 return 0;
}
```

Output



Q.7.(b) Write short note on each:

(i) What are macros?

Sol. An *macro* is a simple identifier which will be replaced by a code fragment. It is called object-like macro because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.

You create macros with the '#define' directive. '#define' is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named BUFFER_SIZE as an abbreviation for the token 1024.

By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros.

The macro's body ends at the end of the '#define' line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

Scanned by CamScanner

```
#define NUMBERS 1, \
    2, \
    3
int x[] = { NUMBERS };
 ==> int x[] = { 1, 2, 3 };
```

The most common visible consequence of this is surprising line numbers in error messages.

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

(ii) How to pass structure to a function?

Sol. The second method involves passing a copy of the entire structure to the called function. Any changes to structure members within the function are not reflected in the original structure. It is therefore, necessary for the function to return the entire structure back to the calling function.

The general format of sending a copy of a structure to the called function is:

return_type function_name (structure_variable_name);

The called function takes the following form:

```
data_type function_name(struct_type tag_name)
{
    ----------
    ----------
    return(expression);
}
```

The called function must be declared for its type, appropriate to the data type it is expected to return.

The structure variable used as the actual argument and the corresponding formal argument in the called function must of the same struct type.

The return statement is necessary only when the function is returning some data back to the calling function. The expression may be any simple variable or structure variable or an expression using simple variables.

When a function returns a structure, it must be assigned to a structure of identical type in the calling function. The called functions must be declared in the calling function appropriately.

Example:

```
#include<stdio.h>
struct student{
    char name [10];
    int rno;
    float avg;
};
void fun (struct student);
int main ( )
{
    struct student s1 = {"SASIDHAR", 101, 78.9};
```



```
    fun (s1);
    printf("/n The Average of the studnet=%.2f",s1.avg);
    return ( );
}
void fun (struct student s)
{
    s.avg=s.avg+10;
    print("/n The Average of the student=%.2f",s.avg);
}
```

UNIT-IV

Q.8.(a) Write a complete program to write 5 student records to a file "Student dat". **(6.5)**

Sol.

```
#include<stdio.h>
struct student
{
int stud_no;
char stud_name[20];
char stud_addr[20];
float stud_percentage;
};

void main()
{
FILE *fp;
char *fname,ch='y';
int n=0;
clrscr();
printf("\n Enter source file name:\t");
gets(fname);
fp=fopen(fname,"w");
if(fp==NULL)
{
printf("\n unable to open");
exit(0);
}
while(n < 5)
{
printf("\n Enter student number");
scanf("%d", &s.stud_no);
printf("\n Enter student name:\t");
scanf("%s", s.stud_name);
printf("\n Enter stud address\n");
scanf("%s", s.stud_addr);
printf("\n Enter percentage");
```

Scanned by CamScanner

```
scanf("%f", &s.stud_percentage);
fprintf(fp,"%d%s%s%f",s.stud_no,s.stud_name,s.stud_addr,s.stud_percentage);
n++;
fflush(stdin);
}
printf("\n %d record are stored in %s file", n, fname);
fclose(fp);getch();}
```

Q.8.(*b*) Write short notes on: **(3+3)**

(*i*) Typedef

TYPEDEF: C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. Its purpose is to redefine the name of an existing variable type.

The general syntax of the typedef is as follows,

typedef data_type IDENTIFIER;

Where,

- typedef is the keyword tells the compiler about the type definition.
- Data type refers to an existing data type.
- IDENTIFIER refers the "new" name given to the data type.
- Usually, uppercase letters are used to make it clear that we are dealing with a renamed data type.

Note that using typedef, we are not creating new data types. Instead we are creating only new name for the existing data type. These new data type names are called user-defined data types.

Suppose we want to store marks scored in various subjects in variables subl, sub2 and sub3. These variables can be declared as follows,

intsubl, sub2, sub3;

Using the user-defined data types, the variables can be declared as shown below,

typedefint MARKS;

MARKS subl, sub2, sub3;

Advantages

- Provides a meaningful way of declaring the variables.
- Increase the readability of the program.
- A complex declaration can be reduced to short and meaningful declaration.
- typedef is widely used while dealing with structures.

(*ii*) Bitfields

Ans. Bitfields

If in a program a variable is to take only two values 1 and O, we really need only a single bit to store it. If a variable is to take values from O to 3, then two bits are sufficient to store these values. So, we cannot waste entire integer to store such data.

There are several variables whose max values are small enough to pack into a single memory location. We can use 'bit fields' to store several values in a single integer.

A bit field is a set of adjacent bits whose size can be from 1 to 16 bits length. A word can therefore be divided in to a number of bit fields.

The name and size of bit fields are defined using structure. Since bit fields are defined within a structure, the various bits can be accessed in a way we access individual members of a structure.

The general for or syntax for bit field definition as follows,

structtag_Name

Unsigned intnamel: bit length;

Unsigned int name2: bit length;

Unsigned intnamen: bit length;

Where,

- unsignedint is always used to define bit fields.
- bit length is the number of bits used for the specified name.
- field name is followed by semicolon.

Remember that a signed bit field should have at least 2bits (1 bit for sign). The bit length is decided by the range of value stored. The largest value that can be stored is 2^{n-1}, where n is bit length.

The internal representation of bit fields is machine dependent. Some machines store bits from left to right and others from right to left. The below Figure illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.

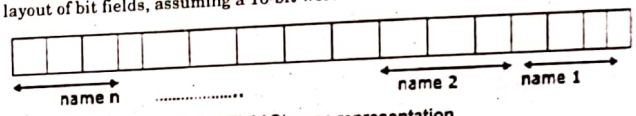


Fig. Bit Field Storage representation

There are several specific points to observe:

- The first field always starts with first bit of the word
- A bit can not overlap integer boundaries. This is, the sum of lengths of all fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
- There can be unnamed field declared with size. Example, unsigned : bit length; Such fields provide within a word.
- There can be unused bits in a word.
- We cannot take the address of a bit field variable.
- Bit fields cannot be arrayed.
- Bit fields should be assigned values that are the range of their Size.

Or

Q.9.(*a*) Write a program to copy contents of one file to another file. **(6.5)**

Ans.

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
void main(int argc, char *argv[])
{
 FILE *fs,*ft;
char ch;
clrscr();
if(argc!=3)
{
        puts("Invalid number of arguments.");
```

Scanned by CamScanner

```
        exit(0);
    }
fs = fopen(argv[1],"r");
if(fs==NULL)
    {
        puts("Source file cannot be opened.");
        exit(0);
}
ft = fopen(argv[2],"w");
if (ft==NULL) // check the condition if the file pointer is NULL or not
{
    puts("Target file cannot be opened.");
    fclose(fs);
    exit(0);
}
while(1)
{
    ch=fgetc(fs);
    if (ch==EOF) // check the condition if the file is end or not
        break;
    else
        fputc(ch,ft);
}
fclose(fs);
fclose(ft);
getch();
}
```

Q.9.(b) Write down format of 3 functions each from string h and c type h. **(6)**

Ans. String are the combination of number of characters these are used to store any word in any variable of constant. A string is an array of character. It is internally represented in system by using ASCII value. Every single character can have its own ASCII value in the system. A character string is stored in one array of character type.

e.g. "Ram" contains ASCII value per location, when we are using strings and then these strings are always terminated by character '\0'. We use conversion specifies %s to set any string we can have any string as follows:-

char nm [25].

When we store any value in nm variable then it can hold only 24 character because at the end of the string one character is consumed automatically by '\0'.

#include<string.h>

There are some common inbuilt functions to manipulation on string in string.h file. Some of them are as follows:-

1. strlen( ) - String Length function

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

```
void main( )
{
char arr[ ] = "NewDelhi" ;
```



```
int len1, len2 ;
len1 = strlen ( arr ) ;
len2 = strlen ( "Capital" ) ;
printf ( "\nstring = %s length = %d", arr, len1 ) ;
printf ( "\nstring = %s length = %d", " Capital ", len2 ) ;
}
```

The output would be...

string = NewDelhi length = 8

string = Capital length = 7

2. strcpy() - String Copy function

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function.

```
void main( )
{
char source[ ] = "Country";
char target[20] ;
strcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

source string = Country

target string = Country

3. strcmp() - String Compare function

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. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is an example:-

```
void main( )
{
char string1[ ] = "Merry" ;
char string2[ ] = "Herry" ;
int i, j, k ;
i = strcmp ( string1, "Jerry" ) ;
j = strcmp ( string1, string2 ) ;
k = strcmp ( string1, "Jerry boy" ) ;
printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

0 5 -32

4. strcat() - String Concatenate function

This function concatenates the source string at the end of the target string. For example, "New" and "Delhi" on concatenation would result into a string "NewDelhi". Here is an example of **strcat( )**:-

Scanned by CamScanner

```
void main( )
{
char source[ ] = "New" ;
char target[30] = "York" ;
strcat ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

source string = New

target string = NewYork

ctype.h

1. isalpha( )

The **isalpha( )** function returns nonzero if *ch* is a letter of the alphabet; otherwise zero is returned. What constitutes a letter of the alphabet may vary from language to language. For English, these are the upper- and lowercase letters A through Z.

Syntax:- int isalpha(int *ch*);

2. isalnum( )

The **isalnum( )** function returns nonzero if its argument is either a letter of the alphabet or a digit. If the character is not alphanumeric, zero is returned.

Syntax:- int isalnum(int *ch*);

3. isdigit( )

The **isdigit( )** function returns nonzero if *ch* is a digit, that is, 0 through 9. Otherwise zero is returned.

Syntax: int isdigit(int *ch*);

Example

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
char ch;
for(;;)
{
ch = getchar();
if(ch == '.') break;
if(isalpha(ch)) printf("%c is a letter\n", ch);
if(isalnum(ch)) printf("%c is alphanumeric\n", ch);
if(isdigit(ch)) printf("%c is a digit\n", ch);
}
return 0;
}
```

FIRST SEMESTER [BCA]
END TERM EXAMINATION [2014]
INTRODUCTION TO PROGRAMMING LANGUAGE USING C [BCA-105]

M. M. 75

Time: 3 hrs.

Note: *Attempt five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. (*a*) What are logical and relational operations?

(*b*) Differentiate between pre-increment and post-increment operator.

(*c*) What is ternary operator? Explain.

(*d*) What is an array of pointers?

(*e*) Differntiate between structute and union.

(*f*) What are pre-processor directives?

(*g*) What are static variables?

(*h*) What is enum datatype?

(*i*) Write is any two function of conio.h.

(*j*) Explain bitwise operators. **(2.5×10=25)**

Sol. 1(*a*): Relational Operators: it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator come into picture. C supports the following relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to

It is required to compare the marks of 2 students, salary of 2 persons, we can compare them using relational operators.

A simple relational expression contains only one relational operator and takes the following form exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25	TRUE
–65 > 0	FALSE
10 < 7 + 5	TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

Logical Operators: C has the following logical operators, they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Scanned by CamScanner

Logical AND (&&)

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Example:- a > b && x = = 10

The expression to the left is a > b and that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (II)

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example:- a < m II a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

Example:- ! (x >= y) the NOT expression evaluates to true only if the value of x is neither greater than or equal to y.

Sol. 1(*b*): The increment operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below:

1. ++ variable name
2. variable name++

The increment operator ++ adds the value 1 to the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following .

```
m = 5;
y = ++m;          (pre-increment)
```

In this case the value of y and m would be 6

Suppose if we rewrite the above statement as

```
m = 5;
y = m++;          (post-increment)
```

Then the value of y will be 5 and that of m will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

Sol. 1(*c*):- The conditional operator consists of 2 symbols the question mark (?) and the colon (:).

The syntax for a ternary operator is as follows:

exp1 ? exp2 : exp3

The ternary operator works as follows :

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

a = 10;

b = 15;

x = (a > b) ? a : b

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/* Example : to find the maximum value using conditional operator)
#include <stdio.h>
void main()                    //start of the program
{
int i,j,larger;                //declaration of variables
printf ("Input 2 integers : ");  //ask the user to input 2 numbers
scanf("%d %d",&i, &j);         //take the number from standard input and store it
larger = i > j ? i : j;        //evaluation using ternary operator
printf("The largest of two numbers is %d \n", larger);  // print the largest number
}              // end of the program
```

Sol. 1(*d*):- ARRAY OF POINTERS:

- A pointer variable always contains an address; an array of pointers would be nothing but a collection of addresses.
- The addresses present in the array of pointers can be addresses of variables or addresses of array elements or any other addresses.
- The major application of this is in handling data tables efficiently and table of strings.
- All rules that apply to an ordinary array apply to the array of pointes as well.
- The Syntax for declaration of array of pointers as follows,

data type *arr_ptr [size];

- This declaration tells the compiler arr_ptr is an array of addresses, pointing to the values of data type.
- Then initialization can be done same as array element initialization. Example **arr_ptr [3] =&var**, will initialize 3rd element of the array with the address of var.
- The dereferencing operator is used as follows

***(arr_ptr [index])** —> will give the value at particular address.

Example

```
#include<stdio.h>
void main ()
{
    int i=1, j=2, k=3, l=4, x;
    int *arr [4];//declaration of array of pointers
    arr [0] =&i;//initializing 0th element with address of i
    arr [1] =&j;//initializing 1st element with address of j
    arr [2] =&k;//initializing 2nd element with address of k
    arr [3] =&l;//initializing 3rd element with address of l
    for (x=0; x<4; x++)
        printf ("\n %d",*(arr[x]));
}
```

Scanned by CamScanner

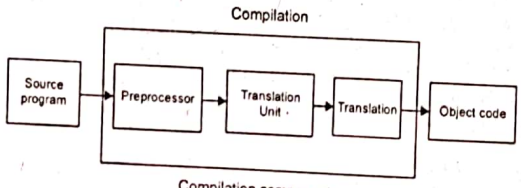
Sol. 1(*e*):

	Structures	Unions
Keyword	struct	union
Definition	A structure is a collection of logically related elements, possibly of different types, having a single name.	A union is a collection of logically related elements, possibly of different types, having a single name shares single memory locati

Declaration	struct tag _ name
	{
	type 1 member 1 ;
	type 1 member 2 ;
	.....
	.....
	};
	struct tag _ name var;
		union tag_ name
		{
		type 1 member 1 ;
		type 1 member 2 ;
		.....
		.....
		};
		union tag _ name var;

Memory Allocation	Each member of the structure occupies unique location, stored in contigenous locations.	Memory is allocated by considering the size of largest member. All the members share the common location
Size	Size of the structure depends on the type of members, adding size of all members. sizeof (st_var);	Size is given by the size of largest member storage. size of (–unvariable);
Access	All members can be accessed at a time	Only one member can be accessed at a time.

Sol. 1(*f*): The C compiler is made of two functional parts: a preprocessor and a translator. The preprocessor is a program which processes the source code before it passes through the compiler. The translator converts the C statements into machine code that in places in an object module.



Compilation components

The preprocessor is a collection of special statements called commands or preprocessor directives. Each directive is examined by the preprocessor before the source program passes through the compiler.

- Statements beginning with # are considered preprocessor directives.
- Preprocessor directives indicate certain things to be done to the program code prior to compilation.

- It includes certain other files when compiling, replace some code by other code.
- Only white space characters before directives on a line.
- Preprocessor commands can be placed anywhere in the program.

There are three major tasks of a preprocessor directive

1. Inclusion of other files (file inclusion)
2. Definition of symbolic constants and macros(macro definition)
3. Conditional compilation of program code/Conditional execution of preprocessor rectives

Sol 1(g): Static Variables

- A static variable that is declared in a block scope, **static** defines the extent of the ariable.
- The computer allocates storage only once. The linkage is internal, and not visible other blocks.
- Also Know as static Local Variable.
- A variable with an **static block** specification has the following storage haracteristic:

Scope: block Extent: static Linkage: internal

Declaration

Static type variable_Name;

Initialization

A static variable name can be initialized where it is defined, or it can be left uninitialized. If it is initialized, it is initialized only once. If not initialized, its value will be initialized to zero.

```
// C Program illustrating static storage class in block
# include <stdio.h>
void fun 1 () ;
int main ()
{
  int i = 1;
  printf ("\n OUTPUT\n");
  for (i = 1; i < = 5; i ++)
          fun 1 () ;
  getch ();
  return 0;
}
void fun 1 ()
{
    static int  x = 0;
    printf ( "/n x = %d", x ++);
}
```

Output

```
x=0
x=1
x=2
x=3
x=5
```

Sol. 1(*h*): Refer Q.1(*h*) from EndTerm Solution December 2013.

Sol. 1(*i*):- conio.h header contains functions for console input/output. Some of the most commonly used functions of conio.h are clrscr, getch, getche, kbhit etc. Functions of conio.h can be used to clear screen, change color of text and background, move text, check if a key is pressed or not and many more. conio.h file is provided by borland turbo c compiler and gcc compiler doesn't support it.

Scanned by CamScanner

clrscr()

clrscr function clears the screen and move the cursor to upper left hand corne screen.

Syntax:- clrscr();

getch()

getch in c language: getch function prompts the user to press a character and t character is not printed on screen.

Syntax:- getch(); fig.

Sol. 1(*j*):- C has a distinction of supporting special operators known as bitw operators for manipulation data at bit level. A bitwise operator operates on each bi data. Those operators are used for testing, complementing or shifting bits to the ri on left. Bitwise operators may not be applied to a float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
~	Bitwise complement
^	Bitwise Exclusive OR
<<	Shift left
>>	Shift right

UNIT I

Q.2 (*a*) Write a program to generate the following output loop structure:- (6.5

```
*
**
***
**
*
```

Sol.

```
# include<stdio.h>
void main()
{
int i,j;
for (i=1; i<=3; i++)
 {
    for(j=1; j<=i; j++)
 {
    printf("* ");
    }
printf("\n");
}
for(i=2; i>=1; i--)
{
```



```
    for (j=1; j<=i; j++)
 {
    printf("* ");
 }
printf("\n");
 }
getch();
}
```

Q.2.(*b*) Differentiate between getchar 0, getch 0 and getche 0. (6)

Sol.

getchar()

The **getchar( )** function returns the next character from **stdin.** The character is read as an **unsigned char** that is converted to an integer. If the end of the file is reached, **getchar( )** returns **EOF**. If **getchar( )** encounters an error, **EOF** is also returned. The **getchar( )** function is often implemented as a macro.

Example

```
#include <stdio.h>
int main(void)
{
char s[256], *p;
p = s;
while((*p++ = getchar())!= '\n') ;
*p = '\0'; /* add null terminator */
printf(s);
return 0;
}
```

getch()

getch in c language: getch function prompts the user to press a character and that character is not printed on screen.

Syntax:- getch();

getche()

getche function prompts the user to press a character and that character is printed on screen.

Syntax:- getche();

Q.3(*a*) write a program to input and compare three integers and print the maximum and minimum of the three. (6.5)

(*b*) Write a program to generate Fibonacci series upto first 20 numbers. (6)

Sol 3(a):-

```
# include<stdio.h>
void main()
{
    int a,b,c;
    printf("Enter any 3 numbers:\n");
    scanf("%d%d%d", &a, &b, &c);
    printf("Minimum: %d\nMaximum: %d", a<b?a:b<c?b:c, a>b?a:b>c?b:c);
```

Scanned by CamScanner

```
    getch();
}
```

Sol 3(b):-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=0,b=1,i,c,n;
clrscr();
printf("ENTER HOW MANY TERMS OF FIBONACCI SERIES IS TO BE PRINTED:");
scanf("%d",&n);
printf("%d %d",a,b);
for(i=1;i<=n-2;i++)
{
c=a+b;
a=b;
b=c;
printf(" %d",c);
}
getch();
}
```

Output:

```
ENTER HOW MANY TERMS IS TO BE PRINTED : 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181_
```

UNIT II

Q.4(a) Explain the difference between call by value and call by refrence with an example. **(6.5)**

Ans. Call by value: In this type, the value of actual arguments is passed to the formal arguments and operation is done on the formal arguments. Any change in the formal argument made does not affect the actual arguments because formal arguments are the photocopy of the actual argument. Hence, when a function is called by the call by value method, it does not affect the actual contents of the arguments. Changes made in the formal arguments are local to the block of the called function. Once control returns back to the calling function, changes made vanish. The example given below illustrates the use of call by value.

In call-by-reference, the address of the parameter is passed by the calling routine to the called function so that it can be altered there. Execution of a function using call-by-reference can take longer than one using call-by-value, since in call-by-reference indirect mode instructions are needed for any calculations involving the variables passed.

Example1:- Call-by-value

```
void main( )
{
int a = 10, b = 20 ;
swapv ( a, b ) ;
printf ( "\na = %d b = %d", a, b ) ;
```



```
}
swapv ( int x, int y )
{
int t ;
t = x ;
x = y ;
y = t ;
printf ( "\nx = %d y = %d", x, y ) ;
}
```

The output of the above program would be:

x = 20 y = 10

a = 10 b = 20

Example 2:- Call-by-reference

```
void main( )
{
int a = 10, b = 20 ;
swapr ( &a, &b ) ;
printf ( "\na = %d b = %d", a, b ) ;
}
swapr( int *x, int *y )
{
int t ;
t = *x ;
*x = *y ;
*y = t ;
}
```

The output of the above program would be:

a = 20 b = 10

Call by value	Call by Reference
Formal parameter is a new local variable that exists within the scope of the procedure/function.	Formal parameter is an alias for the actual parameter.
Value of actual parameter is used to initialize the formal parameter.	Formal parameter refers to the same memory cell for the actual parameter.
Changes made to the formal parameter do not get transmitted back to the caller.	Each formal parameter is a reference to the actual parameter and modifying the formal will modify the actual.

Q.4.(b) Write a program using array to read the integers and find the maximum and minimum of them. **(6)**

Ans.

```
# include<stdio.h>
void main()
{
```

Scanned by CamScanner

```
int a[10],n,min,max,i;
printf("Enter the size of array: ");
scanf("%d",&n);
printf("Enter the array:\n");
for (i=0; i<n; i++)
{
scanf("%d",&a[i]);
}
min=a[0]; max=a[0];
for (i=1; i<n; i++)
{
if(a[i]<min)
min=a[i];
else if(a[i]>max)
max=a[i];
}
printf("Minimum= %d\nMaximum= %d",min,max);
}
```

Output:

```
Enter the size of array: 8
Enter the array:
2
6
8
–4
–38
30
–5
–40
Minimum = –40
Maximum = 30
```

Q.5.(a) What are storage classes in C? Explain. (E

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class:- auto is the default storage class for all local variables.

```
{
int Count;
auto int Month;
}
```


The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables. By default all local variables are of auto storage class.

register - Storage Class:- register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implimentation restrictions.

static - Storage Class:- static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
{
printf("%d\n", Road);
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initalised at run time but is not reinitalized when the function is called. This inside a function static variable retains its value during vairous calls.

```
void func(void);
static count=10; /* Global variable - static is the default */
main()
{
while (count—)
{
func();
}
}
void func( void )
{
static i = 5;
i++;
printf("i is %d and count is %d\n", i, count);
}
```

This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
```

Scanned by CamScanner

i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0

extern - Storage Class:- extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to decalre a global variable or function in another files.

File 1: main.c

```
int count=5;
main()
{
write_extern();
}
```

File 2: write.c

```
void write_extern(void);
extern int count;
void write_extern(void)
{
printf("count is %i\n", count);
}
```

Here extern keyword is being used to declare count in another file.

Q.5.(*b*) Write a program to multiply two matrices. **(6)**

Ans.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int ch,i,j,m,n,p,q,k,r1,c1,a[10][10],b[10][10],c[10][10];
clrscr();
printf("\nMULTIPLICATION OF TWO MATRICES\n\n");
printf("Input rows and columns of A matrix:");
scanf("%d%d",&m,&n);
printf("Input rows and columns of B matrix:");
scanf("%d%d",&p,&q);
if(n==p)
{
 printf("matrices can be multiplied\n");
 printf("resultant matrix is %d*%d\n",m,q);
```



```
printf("Input A matrix\n");
/*Function call to read the matrix*/
read_matrix(a,m,n);
printf("Input B matrix\n");
/*Function call to read the matrix*/
    read_matrix(b,p,q);
/*Function for Multiplication of two matrices*/
printf("\n *******Matrix Multiplication*******\n");
for(i=0;i<m;++i)
for(j=0;j<q;++j)
{
c[i][j]=0;
for(k=0;k<n;++k)
     c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
printf("Resultant of two matrices:\n");
    write_matrix(c,m,q);
}
/*end if*/
else
{
printf("Matrices cannot be multiplied.");
}
getch();
}
/*Function read matrix*/
void read_matrix(int a[10][10],int m,int n)
{
int i,j;
for(i=0;i<m;i++)
     for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
}
/*Function to write the matrix*/
void write_matrix(int a[10][10],int m,int n)
{
int i,j;
for(i=0;i<m;i++)
{
     for(j=0;j<n;j++)
              printf("%5d",a[i][j]);
     printf("\n");
}
}
```

Scanned by CamScanner

UNIT III

Q.6.(a) Write a program to declare any enumeration data type and display its integer values on screen. (6)

Ans.
```
#include <stdio.h>
enum colours {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
NUMBER_OF_COLOURS};
typedef enum colours colour_t;
int main(void)
{
 colour_t sky, forest;
 printf("There are %d colours in the enum\n", NUMBER_OF_COLOURS);
 sky = BLUE;
 forest = GREEN;
 printf("sky = %d\n", (int)sky);
 printf("forest = %d\n", (int)forest);
 return 0;
}
```

The output of this program is:

```
There are 7 colours in the enum
sky = 4
forest = 3
```

Q.6. (b) Write a 'C' program to demonstrate the declaration of structures within structures and arrays within structures. (6)

Ans. Structure within a structure:

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
#include <stdio.h>
#include <conio.h>
struct stud_Res
{
        int rno;
        char std[10];
        struct stud_Marks
        {
                char subj_nm[30];
                int subj_mark;
        }marks;
}result;
void main()
{
        clrscr();
        printf("\n\t Enter Roll Number : ");
```



```
        scanf("%d",&result.rno);
        printf("\n\t Enter Standard : ");
        scanf("%s",result.std);
        printf("\n\t Enter Subject Code : ");
        scanf("%s",result.marks.subj_nm);
        printf("\n\t Enter Marks : ");
        scanf("%d",&result.marks.subj_mark);
        printf("\n\n\t Roll Number : %d",result.rno);
        printf("\n\n\t Standard : %s",result.std);
        printf("\nSubject Code : %s",result.marks.subj_nm);
        printf("\n\n\t Marks : %d",result.marks.subj_mark);
        getch();
}
```

Arrays within structures:- using arrays within structure.

```
#include <stdio.h>
#include <conio.h>
struct result
{
        int rno;
        int mrks[5];                    //Array within structure concept.
        char nm;
}res;
void main()
{
        int i,total;
        clrscr();
        total = 0;
        printf("\n\t Enter Roll Number : ");
        scanf("%d",&res.rno);
        printf("\n\t Enter Marks of 3 Subjects : ");
        for(i=0;i<3;i++)
        {
                scanf("%d",&res.mrks[i]);
                total = total + res.mrks[i];
        }
        printf("\n\n\t Roll Number : %d",res.rno);
        printf("\n\n\t Marks are :");
        for(i=0;i<3;i++)
        {
                printf(" %d",res.mrks[i]);
        }
        printf("\n\n\t Total is : %d",total);
        getch();
}
```

Scanned by CamScanner

Q.7 (a) Write a 'C' program to demonstrate the declaration of structure function parameters.

Ans. Structure as a function parameters. In this concept we need to structure variables as function parameters in the function as explained in the follo example:-

```
#include <stdio.h>
struct dob_st1
{
    int year;
    int month;
    int day;
};
typedef struct dob_st1 dob_st;
void input (dob_st *);
int main ()
{
dob_st date;
dob_st *p;
p=&date;
input(p);
printf("%02i.",p->day);
printf("%02i.",p->month);
printf("%i.",p->year);
return 0;
}
void input(dob_st *p)
{
printf ("Date of birth:\nDay?\n");
scanf ("%i",&(p->day));
printf ("Month?\n");
scanf ("%i",&(p->month));
printf ("Year?\n");
scanf ("%i",&(p->year));
}
```

Q.7.(b) Write a 'C' program that reads a line of text from a data file characte by character and display the text on the screen. (5

Ans.

```
/* Display.c: Display files specified as command line parameters */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
int i = 1 ;
int c ;
```



```
int num_args = 0 ;
FILE *fp;
if ( argc == 1 )
{
fprintf(stderr, "No input files\nUsage: % Display file..\n");
exit(1);
}
if ( argc > 1 )
printf("%d files to be displayed\n", argc-1);
num_args = argc - 1;
while( num_args > 0)
{
printf("[Displaying file %s]\n", argv[i]);
num_args—;
fp = fopen( argv[i], "r" ) ;
if ( fp == NULL )
{
fprintf(stderr,"Cannot display %s \n", argv[i]);
continue;                    /* Goto next file in list */
}
c = getc(fp) ;
while ( c != EOF )
{
putchar( c );
c = getc( fp );
}
fclose( fp );
printf("\n[End of %s]\n————————\n\n", argv[i]);
i++;
}
}
```

Note: The continue statement causes the current iteration of the loop to stop and control to return to the loop test.

Alternate Solution:-

```
#include "stdio.h"
main( )
{
FILE *fs, *ft ;
char ch ;
fs = fopen ( "pr1.c", "r" ) ;
if ( fs == NULL )
{
```

Scanned by CamScanner

```
puts ( "Cannot open source file" ) ;
exit( ) ;
}
ft = fopen ( "pr2.c", "w" ) ;

if ( ft == NULL )
{
puts ( "Cannot open target file" ) ;
fclose ( fs ) ;
exit( ) ;
}

while ( 1 )
{
ch = fgetc ( fs ) ;
if ( ch == EOF )
break ;
else
fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

Q.7. (c) Differentiate between binary and text files. (2.5)

Ans. Text File: A text file is the one where data is stored as a stream of characters that can be processed sequentially. In the text format, data are organized into lines terminated by newline character. The text files are in human readable form and they can be created and read using any text editor. Since text files process only characters, they can read or write only one character at a time. A new line may be converted to carriage return and line feed character. Figure shows the data transfer in text file.

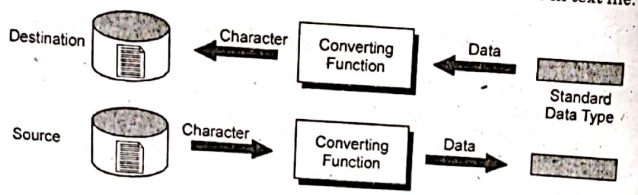


Fig. Reading and writing text files

Because of these translations, the number of characters written / read may not be the same as the number of characters written / read on the external device. Therefore, there may not be a one–to–one relationship between the characters written / read into the file and those stored on the external devices.

For example, the data 2345 requires 4 bytes with each character occupying exactly one byte. A text file cannot contain integers, floating – point numbers etc. converted to their character – equivalent formats.

Binary File: A binary file is the one where data is stored on the disk in the same way as it is represented in the computer memory. The binary files are not in human readable form and they can be created and read only by specific programs written for them. The binary data stored in the file cannot be read using any of the text editors.

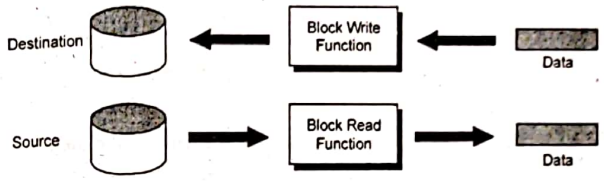


Figure:- Block Input and Output

For example, the data 2345 takes 2 bytes of memory and is stored as 0 x 0929 i.e., 0 x 09 is stored as one byte and 0 x 29 is stored as other byte. The number of characters written / read is same as the number of characters written / read on the external device. Therefore, there is one to – one relationship between the characters written / read into the file and those stored on the external devices.

Differences between Text File and Binary File: The differences between text file and binary file are as shown in Table. The following Figure illustrates the storage difference between text file and binary file.

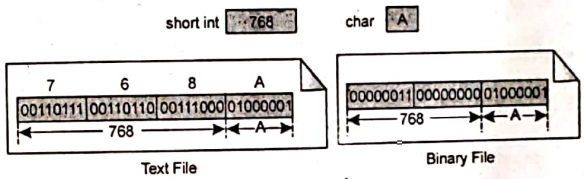


Fig. Binary and Text Files

Table: Differences between Text File and Binary File

Text File	Binary File
1. Human readable format.	1. Not in human readable format.
2. Data is stored as line of characters with each line terminated by/n which may be translated into carriage return +line feed.	2. Data is stored on the disk in the same way as it is represented in the computer memory.
3. Translation of new line character to carriage return + line feed.	3. No translation of any type.
4. Data can be read using any of the text editiors.	4. Data can be read only by specific programs written for them.

Scanned by CamScanner

5. The number of characters written/read may not be the same as the number of characters written/read on the external device such as disk.	5. The number of character written/read same as the number of character written/read on the external device as disk.
6. There may not be a one-to-one relationship between the characters written/read into the file and those stored on the external devices.	6. There is one-to-one relationship between the characters written/read into the file and those stored on the external devices.
7. The data 2345 requires 4 bytes with each character requiring one byte. So, it is stored as 0x 32, 0x 33, and 0 × 34, 0 × 35 (ASCII values) thus requiring 4 bytes.	7. The data 2345 requires 2 bytes and stored as 0x0929. The data 2345 is requiring 2 bytes.

UNIT-IV

Q.8(*a*) Write a program in C that writes records of five employees to employees. (6.5)

Ans.

```
#include<stdio.h>
#include<conio.h>
struct emp
{ int id;
  char name [30];
  char add [30];
};
void main ()
{ struct emp e;
  File * fp;
  int i;
  clrscr ();
  fp = fopen ("employee. dat", "wb");
  for (i=0; i<5;i++)
  {
      printf ("Enter the empid:");
      scan f ("%d", &e.id);
      printf ("Enter the ename:");
      gets (e.name); f
      printf ("Enter the eadd:");
      gets (e.add);
      fwrite (&e, sizeof (e), 1, fp);
      fflush (stdin);
  }
  fclose (fp);
  fp = fopen ("employee.dat", "rb");
  printf ("The employee records are:");
  for (i = 0;i<5;i++)
```



```
  {
      fread (&e, size of (e) ,1, fp);
      printf ("empid:%d\n ename: %s/n eadd: %s\n\n", e.id,e.name,e.add);
  }
  fclose (fp);
  getch ();
```

Enter the empid, ename, eadd : 1 atul kumar singh
vasant kunj
Enter the empid, ename, eadd : 2 royal k sunny
masoodpur
Enter the empid, ename, eadd : 3 lokes thakur
pachim puri
Enter the empid, ename, eadd : 4 abhishek bist
lakshmi nagar
Enter the empid, ename, eadd : 5 raj anand
masoodpur

Q.8.(*b*) Write 'C' program that uses at least two function from given standard libraries: string.h and stdio. h. (6.5)

Ans.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
void main( )
{
int nov=0, l, i, noc=0;
char ch, s[100];
clrscr( );
printf("Enter text:- ");
gets(s);
l=strlen(s);
for(i=0;i<l; i++)
{
if(isalpha(ch))
noc++;
switch(s[i])
{
case 'a':
case 'e':
case 'i':
```

Scanned by CamScanner

```
case 'o'
case 'u'
case 'A'
case 'E'
case 'I'
case 'O'
case 'U'
nov++;
break;
}
}
printf("Total no of vowels are:- %d", nov);
printf("Total no of characters are:- %d", noc);
getch( );
}
```

Q.9. Write formate of the following library functions with example: (12.
fabs (), getch (), labs(), sqrt(), ceil(), pow(), calloc (), getchar(), scanf
rand()

Ans. fabs():-Returns the absolute value of x (a negative value becomes positive
positive value is unchanged). There is no range limit on the argument. The return val
is always positive. This function is declared in math.h header file.

Example:-

```
#include <math.h>
#include <stdio.h>
int main(void)
{
printf("%1.1f %1.1f", fabs(1.0), fabs(-1.0));
return 0;
}
```

Output:- **1.0 1.0**

getch()

getch in c language: getch function prompts the user to press a character and that character is not printed on screen.

Syntax:- getch();

labs():-The **labs( )** function returns the absolute value of *num*.

Example

This function converts the number entered at the keyboard into its absolute value:

```
long int get_labs()
{
```



```
char num[80];
gets(num);
return labs(atol(num));
}
```

sqrt()

sqrt function returns square root of a number.

Declaration :- double sqrt(double);

ceil()

Ceil function is used to round up a number. Declaration: double ceil(double);

pow(): The C library function double pow(double x, double y) returns x raised to the power of y i.e. x^y.

Following is the declaration for pow() function.

```
double pow(double x, double y)
```

Example

The following example shows the usage of pow() function.

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

int main()
{
printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));

printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

return(0);
}
```

Let us compile and run the above program that will produce the following result "

```
Value 8.0 ^ 3 = 512.000000
Value 3.05 ^ 1.98 = 9.097324
```

calloc( ):-

The **calloc( )** function allocates memory the size of which is equal to ***num* * *size***. That is, **calloc()**

allocates sufficient memory for an array of *num* objects of size *size*. All bits in the allocated memory are initially set to zero.

The **calloc( )** function returns a pointer to the first byte of the allocated region. If there is not enough memory to satisfy the request, a null pointer is returned. It is always important to verify that the return value is not null before attempting to use it.

calloc() takes two arguments: total number of data and the size of each data.

eg:- p = calloc(100, sizeof(float));

getchar()

The **getchar( )** function returns the next character from **stdin.** The character is read as an **unsigned char** that is converted to an integer. If the end of the file is reached, **getchar( )** returns **EOF.** If **getchar( )** encounters an error, **EOF** is also returned. The **getchar( )** function is often implemented as a macro.

Scanned by CamScanner

Example

```
#include <stdio.h>
int main(void)
{
char s[256], *p;
p = s;
while((*p++ = getchar())!= '\n') ;
*p = '\0'; /* add null terminator */
printf(s);
return 0;
}
```

scanf()

The **scanf( )** function is a general-purpose input routine that reads the stre **stdin** and stores the information in the variables pointed to in its argument list. It c read all the built-in data types and automatically converts them into the proper intern format. The control string pointed to by *format* consists of three classifications characters:

Format specifiers

White-space characters

Non-white-space characters

The input format specifiers begin with a % sign and tell **scanf( )** what type of da is to be read next. For example, **%s** reads a string, while **%d** reads an integer. The form string is read left to right, and the format specifiers are matched, in order, with t arguments that make up the argument list.

rand():- The C library function **int rand(void)** returns a pseudo-random numb in the range of 0 to *RAND_MAX*.

Following is the declaration for rand() function.

```
int rand(void);
```

END TERM EXAMINATION [DEC. 2015]
FIRST SEMESTER [BCA]
INTRODUCTION TO PROGRAMMING LANGUAGE USING C [BCA–105]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.No.1 which is compulsory.*

Q.1. (a) What do you mean by term 'Data Type'? Explain any four data types with examples. **(2.5×10=25)**

Ans. A 'C' language programmer has to tell the system before-hand, the type of numbers or characters he/she is using in his/her program. These are known as data types. There are many data types in C language. A C programmer has to use appropriate data type as per his/her requirement.

C language data types can be broadly classified as

- Primary data type
- Derived data type
- User defined data type

Integer data types: Integers are whole numbers that can have both positive and negative values, but no decimal values. Example: 0, -5, 10

In C programming, keyword *int* is used for declaring integer variable. For example:

```
int id;
```

Here, *id* is a variable of type integer.

You can declare multiple variable at once in C programming. For example:

```
int id, age;
```

The size of int is either 2 bytes (In older PC's) or 4 bytes.

Floating types: Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using either float or double keyword. For example:

```
float accountBalance;
double bookPrice;
```

Here, both *accountBalance* and *bookPrice* are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

```
float normalizationFactor = 22.442e2;
```

Character types:-

Keyword *char* is used for declaring character type variables. For example:

```
char test = 'h'
```

Here, *test* is a character variable. The value of *test* is 'h'. The size of character variable is 1 byte.

Void Type: The void type has no values therefore we cannot declare it as variable as we did in case of integer and float. The void data type is usually used with function to specify its return type. As well as we can declare a pointer variable of type void.

Q.1. (b) Differentiate between 'Break' and 'Exit' statements.

Ans. Differentiate between break and exit()

Scanned by CamScanner

Break	exit()
break is a keyword in C.	exit() is a standard library function.
break causes an immediate exit from the if, switch statement or loop (for, while or do).	exit() terminates program execution when it is called.
break is a reserved word in C; therefore it can't be used as a variable name.	exit() can be used as a variable name.
No header files needs to be included in order to use break statement in a C program.	stdlib.h/process.h needs to be included in order to use exit().
break transfers the control to the statement follows the if, switch or loop (for, while or do) in which break is executed.	exit() returns the control to the operating system or another program that uses this one as a sub-process.
break is a program control statement which is used to alter the flow of control upon a specified conditions.program,	exit() is a library function, which causes immediate termination of the entire program forcing a return to the operating system.

Q.1. (c) Differentiate between 'Call by Value' and 'Call by Reference'.

Ans. Refer Q.4. (a) End Term Examination December 2014.

Q.1. (d) Explain the concept of 'Pointer' with the help of suitable example'

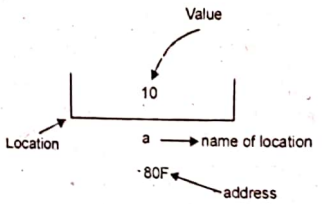
Ans. Pointers are the variables that hold address of another variable of same data type. Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

Benefit of using pointers

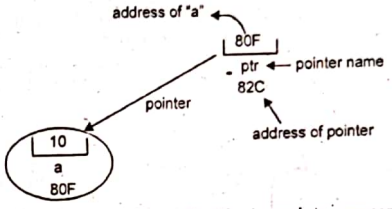
- A pointer enables us to access a variable that is defined outside the function.
- Pointers are more efficient in handling the data tables.
- Pointers reduce the length and complexity of a program.
- The use of a pointer array to character strings save data storage space in memory.
- It allows C to support dynamic memory management.

When a **variable** is declared, system allocates a location to that variable in the memory, to hold value. This location will have its own address number. Let us assume that system has allocated memory location 80F for a variable **a**.

int a = 10 ;



We can access the value 10 by either using the variable name a or the address 80F. Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory address are called pointer variables. A pointer variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of pointer variable will be stored in another memory location.



Example: To access the value of variable 'a' using pointer, we can use the following code:

```
#include<stdio.h>
void main()
{
int a=10;     /* Declaring a variable of type int */
int *ptr;     /* Declaring a pointer variable to an int */
ptr = &a;     /* Storing address of 'a' into pointer variable, after this statement
ptr will point to 'a' */
printf("Value of a =%d \n" , a);
printf("Value of a =%d \n" , *ptr);
printf("Address of a =%d \n" , &a);
printf("Address of a =%d \n" , ptr);
}
```

Note: The *unary* operator & gives the "address of a variable". The *indirection* or dereference operator * gives the "value of an object *pointed to* by a pointer".

Q.1. (e) Write a program to swap two numbers without using third variable.

Ans.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a, b;
printf("Enter two numbers to swap\n");
scanf("%d%d", &a, &b);
printf("Before swapping:- \n a = %d \t b = %d\n", a,b);
a = a + b;
b = a – b;
a = a – b;
```

Scanned by CamScanner

```
printf("After swapping:- \n a = %d \t b = %d\n", a,b);
getch();
}
```

Q.1. (f): Explain the use of 'Continue' statements.?

Ans. Continue statement is a jump statement. The continue statement can be used only inside for loop, while loop and do-while loop. Execution of these statement does not cause an exit from the loop but it suspend the execution of the loop for that iteration and transfer control back to the loop for the next iteration.

Example:-

```
void main( )
{
int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                continue ;
            printf ( "\n%d %d\n", i, j ) ;
        }
    }
}
```

The output of the above program would be...

1 2

2 1

Q.1. (g) What do you mean by 'Recursive Functions?'

Ans. A recursive function is a function that calls itself. Some problems can be easily solved by using recursion, such as when you are dividing a problem into sub- problems with similar natures. Note that recursion is a time-consuming solution that decreases the speed of execution because of stack overheads. In recursion, there is a function call and the number of such calls is large. In each call, data is pushed into the stack and when the call is over, the data is popped from the stack. These push and pop operations are time-consuming operations.

Contents of a Recursive Method:

- Base case(s).
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls.
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Example:-

```
int fact(int n)
```



```
{
    if(n<=0)
            return(1);
else
            return(n*fact(n-1));
}
```

Q.1. (h) Differentiate between const char *p and char const p.

Ans. In const char* p, the character pointed by 'p' is constant, so you can't change the value of character pointed by p but you can make 'p' refer to some other location. In char const* p, the pointer 'p' is constant, but not the character referenced by it, so you can't make 'p' point to any other location but you can change the value of the char pointed by 'p'.

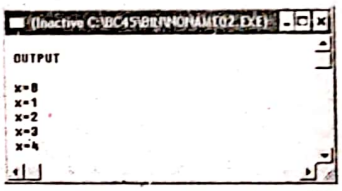
Q.1. (i) Explain the concept of 'Static Variable' with the help of suitable example.

Ans. A static variable tells the compiler to persist the variable until the end of execution of a program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static is initialized only once and remains into existence till the end of the program. A static variable 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 whereas External static variables remain restricted to scope of file in each they are declared.

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

Example:-

```
#include<stdio.h>
Void fun 1();
int main ()
{
    int i=1;
    printf ("/n OUTPUT/n");
    for (i=1; < = 5; ++)
            fun1();
    getch ();
    return 0;
}
void fun1()
{
    static int x = 0;
    printf ("\n x = % d", x++);
}
```



Q.1. (j) What is "Enumerated Data Type"? Explain with suitable example.

Ans. *enum* is the abbreviation used for enumerated data type, and we can use this keyword to declare and initialize a sequence of integer constants. Here is an example:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

Scanned by CamScanner

In the above example constant names are in uppercase, but one can name them whichever way they want.

Here, *colors* is the name given to the set of constants - the name is optional. Now, if one do not assign a value to a constant, the default value for the first one in the list, *RED* in our case, has the value *0*. The rest of the undefined constants have a value 1 more than the one before, so in our case, *YELLOW* is *1*, *GREEN* is 2 and *BLUE* is *3*.

But you can assign values if you want to:

```
enum colors {RED=1, YELLOW, GREEN=6, BLUE };
```

Now RED=1, YELLOW=2, GREEN=6 and BLUE=7.

The main advantage of *enum* is that if we don't initialize your constants, each one would have a unique value. The first would be zero and the rest would then count upwards. We can name constants in any order.

Example:-

```
#include <stdio.h>
int main() {
enum {RED=5, YELLOW, GREEN=4, BLUE};
printf("RED = %d\n", RED);
printf("YELLOW = %d\n", YELLOW);
printf("GREEN = %d\n", GREEN);
printf("BLUE = %d\n", BLUE);
return 0;
}
```

This will produce following output:

```
RED = 5
YELLOW = 6
GREEN = 4
BLUE = 5
```

Q.2. What are the different 'Control Structures' used in C language? Explain all of them with suitable examples. (12.5)

Ans. Control constructs are those which controls the flow of execution. There are mainly two categories of control constructs:- 1. Selection(if, switch) 2. Iteration(loops like while and for).

Selection Control structures: The selection constructs allow us to follow different paths in different situations. We may also think of them as enabling us to express decisions. In 'C', we have the following selection control structures:

if- Statement:

It is the basic form where the if statement evaluate a test condition and direct program execution depending on the result of that evaluation.

Syntax:

```
if (expression)
{
Statement 1;
Statement 2;
}
```


Where a statement may consist of a single statement, a compound statement or nothing as an empty statement. The Expression also referred so as test condition must be enclosed in parenthesis, which cause the expression to be evaluated first, if it evaluate to true (a non zero value), then the statement associated with it will be executed otherwise ignored and the control will pass to the next statement.

Example:

```
if (a>b)
{
printf ("a is larger than b");
}
```

if-else Statement: An if statement may also optionally contain a second statement, the "else clause", which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
average = sum / n;
else
{
printf("can't compute average\n");
average = 0;
}
```

Nested if statement:- It is always legal in C programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Example:-

```
#include <stdio.h>
int main ()
{
int a = 100;  /* local variable definition */
int b = 200;
if( a = = 100 )          /* check the boolean condition */
{
    if( b = = 200 )      /* if condition is true then check this condition*/
    {
    printf("Value of a is 100 and b is 200\n" ); /* if condition is true then print the following */
    }
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
}
```

else-if ladder

- An if statement can be followed by an optional else if...else-statement, which is very useful to test various conditions using single if...else if statement.
- When using if, else if, else statements there are few points to keep in mind.

Scanned by CamScanner

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Example:-

```
#include <stdio.h>
int main ()
{
int a = 100;
if( a == 10 )
{
printf("Value of a is 10\n" );
}
else if( a == 20 )
    {
    printf("Value of a is 20\n" );
    }
else if( a == 30 )
    {
    printf("Value of a is 30\n" );
    }
else
    {
    printf("None of the values is matching\n" );
    }
printf("Exact value of a is: %d\n", a );
return 0;
}
```

Switch Case: This is another form of the multi way decision. It is well structured but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).

Each possible value of the variable can control a single branch. A final, catch all default branch may optionally be used to trap all unspecified cases.

Example:

```
void main( )
{
int i = 2 ;
    switch ( i )
    {
    case 1 :
            printf ( "I am in case 1 \n") ;
            break ;
    case 2 :
            printf ( "I am in case 2 \n" ) ;
            break ;
    case 3 :
            printf ( "I am in case 3 \n" ) ;
            break ;
    default :
            printf ( "I am in default \n" ) ;
    }
}
```


The output of this program would be:

I am in case 2

Iteration or Looping Constructs: Looping is a way by which we can execute any some set of statements more than one times continuously. In C there are mainly three types of loops are used:

- while Loop
- do while Loop
- For Loop

While Loop: Loops generally consist of two parts: one or more *control expressions* which control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The general syntax of a while loop is

```
Initialization;
while( expression )
{
    Statement1;
    Statement2;
    Statement3;
}
```

The most basic *loop* in C is the while loop. A while loop has one control expression, and executes as long as that expression is true.

Example:

```
#include <stdio.h>
void main ()
{
int a = 10;
while( a < 20 )
{
    printf("value of a: %d\n", a);
    a++;
}
getch();
}
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

Scanned by CamScanner

for Loop:- Our second loop, is the for loop. The general syntax of a for loop is

```
for( Initialization; expression; Increments/decrements )
{
    Statement1;
    Statement2;
    Statement3;
}
```

Example:-

```
#include <stdio.h>
void main ()
{
for( int a = 10; a < 20; a = a + 1 )
{
    printf("value of a: %d\n", a);
}
getch();
}
```

do_while Loop:- This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

Example1:-

```
do
{
    printf("Enter 1 for yes, 0 for no :");
    scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0);
```

Example2:-

```
#include <stdio.h>
void main ()
{
int a = 10;
do
{
    printf("value of a: %d\n", a);
    a = a + 1;
}while( a < 20 );
getch();
}
```

Note: Apart from the above two types of control constructs, there are also some jump statements in 'C' like break, continue and goto statements.

Q.3. (a) Write a program to display first n prime numbers. **(6)**

Ans.

```
#include<stdio.h>
int main()
{
int n, i = 3, count, c;
printf("Enter the number of prime numbers required\n");
scanf("%d", &n);
if ( n >= 1 )
{
    printf("First %d prime numbers are :\n", n);
    printf("2\n");
}
for ( count = 2 ; count <= n ; )
{
    for ( c = 2 ; c <= i - 1 ; c++ )
    {
        if ( i%c == 0 )
            break;
}
if ( c == i )
{
    printf("%d\n",i);
    count++;
}
i++;
}
return 0;
}
```

Output:-

```
Enter the number of prime numbers required
10
First 10 prime numbers are:
2
3
5
7
11
13
17
```

Scanned by CamScanner

19

23

29

Q.3. (b) Program to calculate factorial of a number

```
#include<stdio.h>
int main()
{
int i,n, fact = 1;
printf(Enter a number to calculate it's factorial/n");
scan("/od",&n);
for (i = 1;i <n;i++)
    fact = fact * i;
printf ("Factorial of %d = d/n", n, fact);
return 0;
}
```

Output:

Enter a number to calculate it's factorial

6

Factorial of 6 = 720

Q.4. (a) Explain the concept of 'Arrays'. How to declare & initialize 2-Dimensional array? How 2-Dimension arrays are stored in memory? Explain with example. **(6)**

Ans. Definition: An array can be defined as a collection of homogeneous data elements (i.e. data elements of the same data types) stored in the continuous memory locations which is described by a single name (i.e. name of the array). The elements in an array share the same name and are distinguished from one another by their numbers or subscripts. The subscripts are always 0,1,.., size-1.

Array Characteristics

•An array represents a group of related data.

•An array has two distinct characteristics:

An array is ordered: data is grouped sequentially. In other words, here elements are referred as element 0, element 1, etc.

An array is homogenous: every value within the array must share the same data type. In other words, an int array can only hold int values, not doubles.

How to Create an Array

•Before we create an array, we need to decide on two properties:

Array type: what kind of data will your array hold? int, double, char? Remember, we can only choose one type.

Array size: how many elements will your array contain?

When we initially write our program, we must pick an array size. An array will stay this size throughout the execution of the program. In other words, we can change the size of an array at compile time, but cannot change it at run-time.

Declaration Of Two Dimensional(2-D) Array

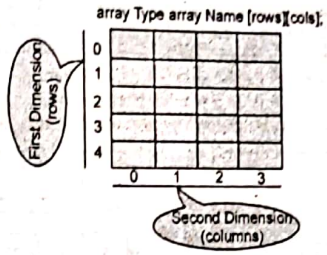


Figure: Two-dimensional Array representation

The above figure shows the representation of elements in the two-dimensional array

Initialization of Two Dimensional arrays

- An array may be initialized at the time of declaration:

```
int num [3][4] = {
                  {1, 2, 3, 4},
                  {5, 6, 7, 8},
                  {9, 10, 11, 12}
                };
```

- An integer array may be initialized to all zeros as follows

  int nums [3][4] = {0};

- In the declaration of two dimensional array the column size should be specified, to that it can arrange the elements in the form of rows and columns.
- Two-dimensional arrays in C are stored in "row-major format": the array is laid out contiguously, one row at a time:

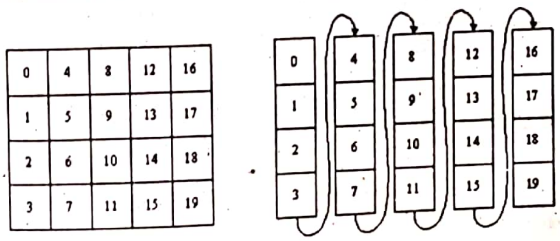
Memory Representation of 2-D Array:

A 2D array's elements are stored in continuous memory locations. It can be represented in memory using any of the following two ways:

1. Column-Major Order
2. Row-Major Order

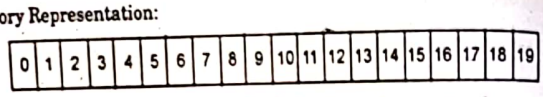
1. Column-Major Order: In this method, the elements are stored column wise, i.e. m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on. Columns are stored in consecutive addresses in the memory

Graphical Representation of column major order:

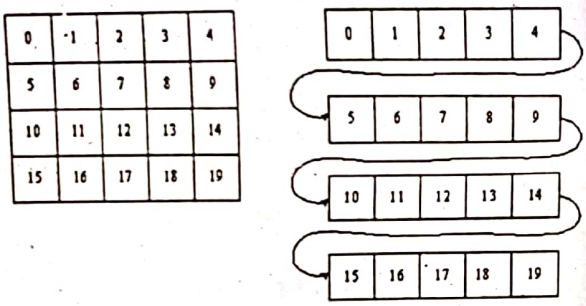


Scanned by CamScanner

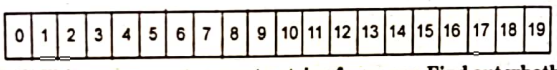
Memory Representation:



2. Row-Major Order: In this method the elements are stored row wise, i.e. elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on. Data stored by rows in consecutive addresses in the memory.



Representation: Memory



Q.4. (b) Write a program to accept a string from user. Find out whether the string is palindrome or not with the help of user defined function. (6.5)

Ans.

```
#include <stdio.h>
#include <string.h>
int str_palindrome(char string1[])
{
int i, flag = 0;
int length = strlen(string1);
for(i=0;i < length ;i++)
{
    if(string1[i] != string1[length-i-1])
    {
            flag = 1;
            break;
    }
}
return flag;
}
void main()
{
char string1[20];
```



```
int result;
printf("Enter a string:");
scanf("%s", string1);
result=str_palindrome(string1);
if (result)
{
printf("%s is not a palindrome", string1);
}
else
{
printf("%s is a palindrome", string1);
}
getch();
}
```

Q.5. What are various 'Storage Classes' in C. Explain their concept and usage with the help of user defined function. (12.5)

Ans. A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class: auto is the default storage class for all local variables.

```
{
 int Count;
 auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables. By default all local variables are of auto storage class.

Example:-

```
void main()
{
 auto mum = 20 ;
 {
 auto num = 60 ;
 printf("1nNum : %d",num);
 }
 printf("1nNum : %d",num);
}
```

register - Storage Class:- register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size

Scanned by CamScanner

equal to the register size (usually one word) and can not have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

Example: /*shows uses of register storage class*/

```
#include<stdio.h>
int main()
{
register int i;
for(i=10; i>=0; i=i-2)
  printf("%d ",i);
return 0;
}
```

Output:-

10 8 6 4 2 0

static - Storage Class:- static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
{
printf("%d\n", Road);
}
```

static variables can be 'seen' within all functions in this source file. At link time, static variables defined here will not be seen by the object modules that are brought. *static* can also be defined within a function. If this is done the variable is initialized at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls.

```
void func(void);
static count=10;        /* Global variable - static is the default */
void main()
{
while (count—)
{
func();
}
}
void func( void )
```



```
{
static i = 5;
i++;
printf("i is %d and count is %d\n", i, count);
}
```

This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

extern - Storage Class:- extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another files.

File 1: main.c

```
int count=5;
main()
{
write_extern();
}
```

File 2: write.c

```
void write_extern(void);
extern int count;
void write_extern(void)
{
printf("count is %i\n", count);
}
```

Here extern keyword is being used to declare count in another file.

Q.6. (a) What is 'File pointer'? Explain any five 'File handling Functions'. Write a program to copy contents of one text file to another text file. (6.5)

Ans. C communicates with files using a new datatype called a **file pointer**. This type is defined within stdio.h, and written as **FILE ***. A **file pointer** called output_file is declared in a statement like. **FILE** *output_file; Opening a **file pointer** using fopen.

Scanned by CamScanner

File Handling Functions:

Function	Syntax	Example

fopen() - Open file	FILE * fopen(char * filename, char * mode)
	Filename – string filename with path
	Mode – mode of opening
	r read only
	w write only
	a append file
	+ read and write
	(r+/w+a+)
	Return value – File pointer on
	sucess
	NULL on failure
		Filename – string filename with infile = fopen ("a. txt.","t");
		
		FILE * outfile = fopen ("b.txt","w");
		if(infile ==NULL)
		outfile == NULL)
		{
		
		printf("error");
		}

fclose() – close the open file	fclose (FILE * fptr) fptr-FILE* of the open file Returns void	fclose (infile) fclose (outfile)

fgetc() – read one char from file	int fgetc (FILE*fptr)
	
	fptr - FILE* of the open file
	returns - character read
	-1 on eof or error
		while (ch=fgetc())!=-1)
		putchar(ch);
		// display content of file
		//on screen

fputc() – write one character to the file	int fputc (int ch.FILE*fptr ch – character to write fptr - open file pointer returns character written EOF if there is error	for (ch='A; ch<='Z';ch++) fbutc(ch, fbtr);
ftell() – get the position of file pointer	int ftell (FILE* ptr) ptr – FILE pointer Returns – position of file pointer -1 on error	FILE* ptr = fopen (b[1], "r"); fseek (ptr, 0L, SEEK_END); int size _of _ file = ftell(ptr);

Program to copy contents of one text file to another text file.

```
#include <stdio.h>
void main()
{
    FILE *fp1, *fp2;
    char c ;
    char fname1[40], fname2[40] ;
    printf("Enter source file:") ;
    gets(fname1);
    printf("Enter destination file:");
    gets(fname2);
    fp1 = fopen( fname1, "r" );          /* open for reading */
    fp2 = fopen( fname2, "w" );          /* open for writing */
```



```
    if ( fp1 == NULL )                   /* check does file exist etc */
    {
        printf("Cannot open %s for reading \n", fname1 );
        exit(1);                         /* terminate program */
    }
    else if ( fp2 == NULL )
    {
        printf("Cannot open %s for writing \n", fname2 );
        exit(1);                         /* terminate program */
    }
    else                                 /* both files O.K. */
    {
        c = getc(fp1) ;                  /* read from source */
        while ( c != EOF)
        {
            putc( c, fp2);               /* copy to destination */
            c = getc( fp1 ) ;
        }
    fclose ( fp1 );                      /* Now close files */
    fclose ( fp2 );
    printf("Files successfully copied \n");
    }
}
```

Q.6. (*b*) Explain the concept of 'Pointer to array' with the help of suitable example. **(6)**

Ans. An array name is a constant pointer to the first element of the array. Therefore, in the declaration -

```
int arr[50];
```

arr is a pointer to &arr[0], which is the address of the first element of the array arr. Thus, the following program fragment assigns p as the address of the first element of arr -

```
int *p;
int arr[10];
p = arr;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(arr + 4) is a legitimate way of accessing the data at arr[4].

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on. Given below is the example to show all the concepts discussed above –

```
#include <stdio.h>
void main ()
{
 /* an array with 5 elements */
```

Scanned by CamScanner

```
int arr[5] = {1000, 2, 3, 17, 50};
int *p;
int i;
p = arr;

/* output each array element's value */
printf( "Array values using pointer\n");
for ( i = 0; i < 5; i++ )
{
    printf("*(p + %d) : %d\n", i, *(p + i) );
}
printf( "Array values using arr as address\n");
for ( i = 0; i < 5; i++ )
{
    printf("*(arr + %d) : %d\n", i, *(arr + i) );
}
getch();
}
```

In the above example, p is a pointer to int, which means it can store the address of a variable of int type. Once we have the address in p, *p will give us the value available at the address stored in p.

Q.7. (a) Explain the concept of 'Preprocessor Directive'? Differentiate the simple macro, macro with argument and control directives with examples? (8.5)

Ans. The preprocessor is a collection of special statements called commands or preprocessor directives. Each directive is examined by the preprocessor before the source program passes through the compiler.

- Statements beginning with # are considered preprocessor directives.
- Preprocessor directives indicate certain things to be done to the program code prior to compilation.
- It includes certain other files when compiling, replace some code by other code.
- Only white space characters before directives on a line.
- Preprocessor commands can be placed anywhere in the program.

There are three major tasks of a preprocessor directive

1. Inclusion of other files (file inclusion)
2. Definition of symbolic constants and macros(macro definition)
3. Conditional compilation of program code/Conditional execution of preprocessor directives

Simple Macros(Macros without arguments):

Macro definition without arguments is referred as a constant. The body of the macro definition can be any constant value including integer, float, double, character, or string. However, character constants must be enclosed in single quotes and string constants in double quotes.

Example

```
#define PI 3.14159
```

Here "PI" replaces with "3.14159"

Macro with arguments:

The following syntax is used to define a macro:

```
#define macro_name(<arg_list>) macro_body
```

#define is a define directive, macro_name is a valid C identifier. macro_body is used to specify how the name is replaced in the program before it is compiled.

- Example

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

would cause

```
area = CIRCLE_AREA( 4 );
```

to become

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

- Use parenthesis for coding macro_body,Without them the macro

```
#define CIRCLE_AREA( x ) PI * ( x ) * ( x )
```

would cause

```
area = CIRCLE_AREA( c + 2 );
```

to become

```
area = 3.14159 * c + 2 * c + 2;
```

- We can also supply Multiple arguments like

```
#define RECTANGLE_AREA( x, y ) ( ( x ) * ( y ) )
```

would cause

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

to become

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

Control Directives:

- It allows us to control the compilation process by including or excluding statements.
- Cast expressions, size of, enumeration constants cannot be evaluated in preprocessor directives.
- Its structure similar to if statement.
- Syntax

```
#if expression1
        code to be included for true
#elif expression2
        code to be included for true
#else
        code to be included false
#endif
```

- Example,

```
#if !defined( NULL )
```

Scanned by CamScanner

```
#define NULL 0
#endif
```

•Determines if symbolic constant NULL has been defined If NULL is defin defined(NULL) evaluates to 1,If NULL is not defined, and this function defines NU to be 0.

• Every #if must end with #endif.

Q.7. (*b*) What is the difference between 'Structure'& 'Union'? Explain w example.

Ans. Refer Q.1. (e) from EndTerm Solution December 2014.

Example:

```
struct first
{
char c;
long l;
char *p;
};
union second
{
char c;
long l;
char *p;
};
```

A struct first contains all of the elements c, l, and p. Each element is separate an distinct.

A union second contains only one of the elements c, l, and p at any given time. Eac element is stored in the same memory location (well, they all start at the same memor location), and you can only refer to the element which was last stored. (ie: afte "secondptr->c = 2;" you cannot reference any of the other elements, such as "secondpt >p" without invoking undefined

Q.8. (*a*) Write a program to count the number of vowels in a given string? (6.5

Ans.

```
#include <stdio.h>
#include<conio.h>
void main()
{
 char string[80];
 int i, vowels = 0;
 clrscr();
 printf("Enter a string \n");
 gets(string);
 for (i = 0; string[i] != '\0'; i++)
 {
```



```
 if ((string[i] == 'a' || string[i] == 'e' || string[i] == 'i' || string[i] == 'o' || string[i]
 == 'u') || (string[i] == 'A' || string[i] == 'E' || string[i] == 'I' || string[i] == 'O' ||
 string[i] == 'U'))
 {
 vowels = vowels + 1;
 }
 }
 printf("No. of vowels in %s = %d\n", string, vowels);
 getch();
 }
```

Q.8. (*b*) Write a program using array to read ten integers and find out total umber of positive and negative numbers. (6)

Ans.

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int countp=0, countn=0, arr[10], i;
printf("Enter 10 numbers : ");
for(i=0; i<10; i++)
{
scanf("%d", &arr[i]);
}
for(i=0; i<10; i++)
{
    if(arr[i]<0)
    {
            countn++;
    }
    else if(arr[i]>0)
    {
            countp++;
    }
}
printf("Positive Numbers = %d\n", countp);
printf("Negative Numbers = %d\n", countn);
getch();
}
```

Output:-

```
Enter 10 numbers : 12
23
– 56
```

Scanned by CamScanner

0
–34
23
0
0
56
–76
Positive Numbers = 4
Negative Numbers = 3

END TERM EXAMINATION [DEC. 2016]
FIRST SEMESTER [BCA]
INTRODUCTION TO PROGRAMMING LANGUAGE USING C [BCA–105]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q. no. 1 which is compulsory. Select one question from each unit.*

Q. 1.(*a*) What is ternary operator? Write a program to find smallest among 3 numbers using ternary operator. **(2.5)**

Ans. Ternary operator consists of 2 symbols the question mark (?) and the colon (:)

The syntax for a ternary operator is as follows:-

exp1 ? exp2 : exp3

The ternary operator works as follows :-

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

Program:- Smallest of three numbers using ternary operator.

```
# include <stdio.h>
# include <conio.h>
void main( )
{
int a, b, c, small;
clrscr( );
printf("Enter value a, b and c");
scanf("%d%d%d", &a, &b, &c);
small=(a<b)?((a<c)?a:c):((b<c)?b:c);
printf("Smallest of the given  numbers is %d", small);
getch();
}
```

Q.1. (*b*) What are Bitwise operators? **(2.5)**

Ans. Refer Q.1. (*j*) from End Term Examination December 2015.

Q.1. (*c*) What are the basic data types and their associated range? **(2.5)**

Ans. Datatypes and their range

Datatype	Range
Char or signed char	–128 to 127
unsigned char	0 to 255
int or signed int	–32768 to 32767
unsigned int	0 to 65535
short int or signed short int	–128 to 127
unsigned short int	0 to 255

Scanned by CamScanner

long int or signed long int	–2147483648 to 2147483647
unsigned long int	0 to 4294967295
float	3.4 e – 38 to 3.4 e + 38
double	1.7e – 308 to 1.7e + 308
long double	3.4 e – 4932 to 3.4 e + 4932

Q.1. (*d*) Differentiate between exit control loop and entry control loop.

Ans.

Entry Controlled Loop	Exit Controlled Loop
Test condition is checked first, and then body of the loop will be executed. If Test condition is false, body of the loop will not be executed.	Body of the loop will be executed first, and then condition is checked. Body of the loop will be executed once even if Test condition is false.
for loop and while loop are the examples of Entry Controlled Loop.	do_while loop is the example of Exit controlled loop.
Entry Controlled Loops are used when checking of test condition is necessary before executing loop body.	Exit Controlled Loop is used when checking of test condition is necessary after executing the loop body.

Example:
.....
sum = 0;
n=1;
 	
while(n<=10)
{
sum=sum+n;
n=n+1;
}
.....
	Example:
	.....
	do
	{
	{printf("Enter a number");
	scanf("%d", &n);
	
	} while(n > 0);
	.....

Q.1. (*e*) Differentiate between break and continue. (2.

Ans. Break statement: When **break** is encountered inside any loop, cont automatically passes to the first statement after the loop. A **break** is usually associat with an **if.**

Example:

```
void main()
{
int num, i;
printf ( "Enter a number") ;
scanf("%d", &num ) ;
i = 2;
while ( i <= num - 1 )
```



```
{
if(num%i ==0)
{
printf ( "Not a prime number" ) ;
break ;
}
i++;
}
if(i==num)
printf ( "Prime number" ) ;
}
```

The keywordbreak, breaks the control only from the **while** in which it is placed.

Continue statement: continue statement is a jump statement. The continue statement can be used only inside for loop, while loop and do-while loop. Execution of these statement does not cause an exit from the loop but it suspend "the execution of the loop for that iteration and transfer control back to the loop for the next iteration.

Example:

```
void main( )
{
int i, j ;
  for (i = 1; i <= 2 ; i++)
{
        for (j = 1; j < = 2; j++)
{
if (i==j)
        continue:
        printf ("\n% d/n", i,j);
      }
    }
  }
```

Q.1. (*f*) What do you mean by variable? What are the various rules for naming a variable in C? **(2.5)**

Ans. Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working. It may help to think of variables as a placeholder for a value. You can think of a variable as being equivalent to its assigned value. A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of particular type

Rules for Constructing Variable Names

(*a*) A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

(*b*) The first character in the variable name must be an alphabet or underscore.

(*c*) No commas or blanks are allowed within a variable name.

(*d*) No special symbol other than an underscore can be used in a variable name.

Scanned by CamScanner

Ex.: si_int

m_hra

pop_e_89

gross_sal

Q.1. (*g*) Define operators. What are the different types of operators? (2

Ans. An operator is a symbol which helps the user to command the computer to a certain mathematical or logical manipulations. In other words, one can say opera tell about the operation to be performed. Operators are used in C language program operate on data and variables. C has a rich set of operators which can be classified

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increments and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators (comma operator, sizeof() operator)

Q.1. (*h*) What is modulus operator? What are the restrictions of modulu operator? (2.

Ans. Take a simple arithmetic problem: what's left over when you divide 11 by The answer is easy to compute: divide 11 by 3 and take the remainder: 2. But how wou you compute this in a programming language like C? C language provides a built- mechanism, the **modulus operator** ('%'), that computes the remainder that results fro performing integer division. The modulus operator is useful in a variety of circumstance It is commonly used to take a randomly generated number and reduce that number a random number on a smaller range, and it can also quickly tell you if one number is factor of another. There is one restriction for the modulus operator that it can be use with integers.

Example: When a = 9 is divided by b = 4, the remainder is 1.

Q.1. (*i*) Differentiate memory allocation for structure and array? (2.

Ans. Memory allocation for structure

Always, contiguous(adjacent) memory locations are used to store structure membe in memory. Consider below example to understand how memory is allocated f structures. Let us take an example

```
struct student
{
        char name [10];
        int roll_number;
        float avg_marks;
};                          // no memory is allocated for the structure definitio
```

Note the following points:

• name, roll_number and avg_marks are members of a structure and are themselve are not variables. They do not occupy any memory.

• Memory is not reserved for the structure definition since no variables are associate with the structure definition. The members of the structure do not occupy any memor until they are associated with the structure variables.

• For the above example the variable declaration as follows,

struct student s1; memory is allocated for the variable

now a variable of type

struct student (derived type) is created, the memory is allocated for the variable s1. The following figure shows the memory organization for the above example.

s1

← Name →	← Roll_Number →	← Avg_Marks →
< 10 Bytes >	< 2 Bytes>	< 4 Bytes >

Fig. Memory map for structure variable

Memory allocation for Array

Each array element occupies memory, stored in continuous memory locations.

For example:

int temp[5];

Creates an array of 5 integer elements and memory for these 5 integer elements will be allocated continuously at the compile time like:

If the size of integer element is 2 bytes, then 10 bytes will be allocated for the variable temp(array name will hold the base address *i.e.* the address of the first element).

temp[0]	temp[1]	temp[2]	temp[3]	temp [4]	
0	0	0	0	0	
1000	1002	1004	1006	1008	← Address

Fig. Memory Representation of array

Q.1. (*j*) Why does goto statement is not preferred in C-programming. (2.5)

Ans. The goto statement exists within the C language. The statement can make programs incredibly difficult to understand and maintain. It is error prone. There is widespread agreement in the programming community that goto should be avoided for general control, and many argue to avoid it in all cases.

Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them. As a general style rule, it should be avoided.

UNIT - I

Q.2. (*a*) List the different types of loop control statements and explain them with suitable examples. (6)

Ans. Refer Loop Control Structures from Q. 2. from End Term Examination Dec. 2015.

Q.2. (*b*) Write a program to display the following pattern as the output. (6.5)

a

a b

a b c

a b c d

a b c d e

Scanned by CamScanner

Ans.
```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, j, c;
printf("*****Pyramid**** \n\n\n");
for(i=1;i<=5;i++)
{
c=97;                    //Ascii value of c
for(j=1;j<=i;j++)
{
printf("%c",c);
c++;
}
printf("\n");
}
getch();
}
```

Output

```
**** Pyramid****
a
ab
abc
abcd
abcde
```

Q.3. (a) Write a program to reverse the given integer number. (Hint. Input: 1234, the reverse is 4321). **(6.5)**

Ans. :- Program to reverse the given integer number

```
# include <stdio.h>
# include <conio.h>
int reverse( int n );
void main( )
{
int a, b, n;
clrscr( );
printf("Enter a number:")
scanf("%d", &n);
a = reverse ( n );
printf("Reverse of %d is %d", n, a);
getch( );
}
int reverse( int n )
{
int r, rev=0, s;
```



```
while(n>0)
{
r = n%10;
rev = rev*10 + r ;
n = n/10;
}
return rev;
}
```

Q.3. (b) What is the difference between pass by reference and pass by value? Explain with suitable example. **(6)**

Ans. Pass by value: In this, the value of actual arguments is passed to the formal arguments and operation is done on the formal arguments. Any change in the formal argument made does not affect the actual arguments because formal arguments are the photocopy of the actual argument. Hence, when a function is called by the call by value method, it does not affect the actual contents of the arguments. Changes made in the formal arguments are local to the block of the called function. Once control returns back to the calling function, changes made vanish.

Pass-by-reference: In this, the address of the actual parameter is passed by the calling routine to the called function so that it can be altered there. Execution of a function using call-by-reference can take longer than one using call-by-value, since in call-by-reference indirect mode instructions are needed for any calculations involving the variables passed.

Example:

```
void main( )
{
int a = 10, b = 20 ;
swapv ( a, b ) ;                    //Call by value
printf ( "\na = %d b = %d", a, b ) ;
swapr ( &a, &b ) ;                  //Call by reference
printf ( "\na = %d b = %d", a, b ) ;
}
void swapv ( int x, int y )         //Pass by value
{
int t ;
t = x ;
x = y ;
y = t ;
printf ( "\nx = %d y = %d", x, y ) ;
}
void swapr( int *x, int *y )        //Pass by reference
{
int t ;
t = *x ;
*x = *y ;
*y = t ;
}
```

Scanned by CamScanner

Pass by value	Pass by Reference
Formal parameter is a new local variable that exists within the scope of the procedure/function.	Formal parameter is an alias for the actual parameter.
Value of actual parameter is used to initialize the formal parameter. Changes made to the formal parameter do not get transmitted back to the caller.	Formal parameter refers to the same memory cell for the actual parameter. Each formal parameter is a reference to the actual parameter and modifying the formal will modify the actual.

UNIT-II

Q.4. (a) Explain the concept of static variable with the help of suitable example. (4

Ans. Refer Q.1.(i) from End Term Examination December 2015.

Q.4. (b) Differentiate between getc (), getch(), getche(), gets(). (2

Ans. getc() : The getc functions read the next character from the file stream, which can be a user-defined stream or stdin, and converts it in to an integer(typically ASCII value of read character) on success. It returns EOF on failure).

Syntax:

getc(stdin) or getc(file pointer)

getch() : getch function prompts the user to press a character and that character is not printed on screen.

Syntax:- getch();

getche() : getche function prompts the user to press a character and that character is printed on screen.

Syntax:- getche();

gets():- gets reads a whole line of input into a string until a new line or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines. When all input is finished, NULL is returned.

Syntax:- gets(line);

Q.4. (c) Write a program to copy one string into another string without using inbuilt function. (6.5

Ans. Program to copy one string to another string

```
#include <stdio.h>
int main()
{
   char str1[100], str2[100], i;
   printf("Enter string1:");
   scanf("%s",str1);
   for(i=0; str1[i]!='\0'; ++i)
   {
      str2[i]=str1[i];
   }
```



```
   str2[i]= '\0';
   printf("String2 : %s", str2);
   return 0;
}
```

Q.5. (a) Explain all storage classes in C with suitable example. (6)

Ans. Refer Q. 5. from End Term Examination Dec. 2015.

Q.5. (b) Differentiate between pointer to any array and array of pointers with suitable example. (6.5)

Ans. When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element is also allocated by the compiler.

Suppose we declare an array arr,

int arr[5] = { 1, 2, 3, 4, 5 };

Assuming that the base address of arr is 1000 and each integer requires two byte, the five element will be stored as follows

element	arr [0]	arr [1]	arr [2]	arr [3]	arr [4]
Address	1000	1002	1004	1006	1008

Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000.

arr is equal to &arr[0] // by default

We can declare a pointer of type int to point to the array arr

Example:

```
# include<stdio.h>
int main ()
{
int num [5] = {1, 2, 3, 4, 5};
int i, * ptr;
ptr = num;
for (i=0; i<5;i++)
{
printf C"\na [%d] = %d", i, *ptr,* (num+i));
ptr++;
}
return O;
}
```

Note: ptr++ is used to move to the next element of the array num.

Array of pointers: The way there can be an array of **ints** or an array of **floats**, similarly there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply to the array of pointers as well.

Scanned by CamScanner

Example:-

```
void main( )
{
int *arr[4] ;                    /* array of integer pointers */
int i = 31, j = 5, k = 19, l = 71, m ;
arr[0] = &i ;
arr[1] = &j ;
arr[2] = &k ;
arr[3] = &l ;
for ( m = 0 ; m <= 3 ; m++ )
printf( "%d ", * ( arr[m] ) ) ;
```

UNIT-III

Q.6. (a) What is 'recursion? What are various conditions associated with recursion? Write a program to calculate XY using recursion where values for X and Y are entered by user. (Don't Use power). **(6)**

Ans. Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A recursive function is a function that calls itself. Some problems can be easily solved by using recursion, such as when you are dividing a problem into sub- problems with similar natures. Note that recursion is a time-consuming solution that decreases the speed of execution because of stack overheads. In recursion, there is a function call and the number of such calls is large. In each call, data is pushed into the stack and when the call is over, the data is popped from the stack. These push and pop operations are time-consuming operations.

Condition of a Recursive Method:-

• Base case(s).

–Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).

–Every possible chain of recursive calls must eventually reach a base case.

• Recursive calls.

– Calls to the current method.

– Each recursive call should be defined so that it makes progress towards a base case.

Program to calculate X^Y using recursion

```
#include <stdio.h>
int power(int n, int p);
int main()
{
int num, pow;
printf("Enter a number and power you wish to raise it to: ");
scanf("%d%d", &num, &pow);
printf("%d raise to power %d : %d", num, pow, power(x, n));
return 0;
}
```



```
int power(int n, int p)
{                                   // anything raise to power 0 is 1.
if (p == 0)
    return 1;
else
    return n * power(n, p - 1);
}
```

Q.6. (b) What is pointer arithmetic? Explain with suitable examples. **(6.5)**

Ans. Pointer arithmetic is very important to understand, if you want to have complete knowledge of pointer. Pointer arithmetic is related to performing arithmetic operations on pointers. Some of the examples of pointer arithmetic are as below:

The following operations can be performed on a pointer:

• Addition of a number to a pointer. Pointer can be incremented to point to the next locations.

Example:

```
int i=4 , pi=&i;                //(assume address of i=1000)
float j,*pj=&j;                 // (assume address of j=2000)
pi = pi + 1;// here pi incremented by (1*data type times)
pi = pi + 9;                    // pi = 1000 + (9*2) → 1018 address
pj = pj + 3;                    // pj=1018+(3*4) → 1030 address
```

• Subtraction of a number from a pointer. Pointer can be decremented to point to the earlier locations.

Example:

```
int i=4,*pi=&i;                 //assume address of i =1000)
char c, *pc=&c;                 // assume address of c = 2000
double d, *pd = &d;             // assume address of d=3000
pi = pi-2;/* pi = 1000-(2*2)=996 address */
pc = pc-5;                      /* pc = 2000-(5*1)=1985 address
pd = pd-6;                      /* pd = 3000-(6*8)=2952 address */
```

•Pointer variables may be subtracted from one another. This is helpful while finding array boundaries. Be careful while performing subtraction of two pointers.

•Pointer variables can be used in comparisons, but usually only in a comparison to NULL.

• We can also use increment/decrement operators with pointers this is performed same as adding/subtraction of integer to/from pointer.

The following operations cannot be performed on pointers.

• Addition of two pointers.

• Multiplication of a pointer with a constant, two pointers.

• Division of a pointer with a constant, two pointers.

Q.7. Explain the following with suitable examples: **(12.5)**

(a) Nested Structure

Ans. Nested Structures: A structure which includes another structure is called nested structure i.e a structure can be used as a member of another structure.

Scanned by CamScanner

Good candidates for nested structures:

Student: dob -> day, month, year

→ Employee: allowance -> da, ta, hra

Player: tests, one days, T20

Example:

```
#include <stdio.h>
struct Employee
{
  char ename[20];
  int ssn;
  float salary;
  struct date  //Nesting of structures
    {
    int date;
    int month;
    int year;
    }doj;
}emp = {"Amit",1001,45000.50,{18,4,2004}};
int main( )
{
printf("\nEmployee Name   : %s", emp.ename);
printf("\nEmployee SSN    : %d", emp.ssn);
printf("\nEmployee Salary : %f", emp.salary);
printf("\nEmployee DOJ   : %d/%d/%d", emp.doj.date, emp.doj.month, emp.doj.year);
return 0;
}
```

Accessing Nested Elements :

- Structure members are accessed using dot operator.
- 'date' structure is nested within Employee Structure.
- Members of the 'date' can be accessed using 'employee'
- emp and doj are two structure names (Variables)

Q.7. (*b*) Arrays within Structure

Ans. Arrays within structure : It is also possible to declare an array as a member of structure, like declaring ordinary variables. For example to store marks of a student in three subjects the we can have the following definition of a structure.

```
struct student
{
        char name [5];
      int roll_number;
     int marks [3];
     float avg;
};
```


Example:

```
#include<stdio.h>
struct student
{
  char name [10]
  char rollno [10];
  char gender;
  int marks[3]
  float avg;
};
int main ()
{
int i, sum = 0;
struct student S1;
printf ("\n\t\t\t Enter the students details : ");
printf ("\n\n Enter the name of the student : ");
fflush (stdin);
gets (s1. name);
printf (" Enter the gender (Male/Female) : ") ;
sum = 0;
for (i = 0; i < 3; i ++)
scanf ("%d", &s1. marks [i]);
sum=sum+s1. marks [i];
}
s1. avg=sum/6.0;
printf("\n\t\t\t The details of the student are ");
print f("\n\n Name of the student : %s" s1. name);
print f("\n Roll\no of the student:%s", S1. Roll no);
print f("\n Gender of the student : %C", s1. gender);
printf ("\n Average marks of the student : %f", s1. avg);
return 0;
}
```

Q.7. (*c*) Structure as function parameter

Ans. Structure as function parameter: A structure variable can be passed to the function as an argument/parameter as a normal variable. If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable. If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

Example:

```
#include <stdio.h>
struct student
{
  char sname[30];
```

Scanned by CamScanner

```
  int rollno;
  float avg;
};
void display(struct student);
        // function prototype should be below to the structure declaration
        otherwise compiler shows error
void main()
{
  struct student s1;
  printf("Enter student's name:");
  scanf("%s", s1.sname);
  printf("Enter roll number:");
  scanf("%d", &s1.rollno);
  printf("Enter average:");
  scanf("%f", &s1.avg);
  display(s1);          // passing structure variable s1 as an parameter
  getch();
}
void display(struct student s)
{
  printf("Output: \nName: %s", s.sname);
  printf("\nRoll No: %d", s.rollno);
  printf("\nAverage: %f", s.avg);
}
```

UNIT-IV

Q.8. (a) What are header files? Give the format of any three library functions for each of these files: ctype. h, stdio. h, math. h

Ans. Header Files: In C programming, a source code file that contains common definitions and data structures that all programmers may use as required. Also called "include files," because the #include statement is used to incorporate them with the program, header files use a .h extension.

Library functions from ctype.h

isalpha( ) : The **isalpha( )** function returns nonzero if *ch* is a letter of the alphabet; otherwise zero is returned. What constitutes a letter of the alphabet may vary from language to language. For English, these are the upper- and lowercase letters A through Z.

Syntax:- int isalpha(int *ch*);

isalnum( ) : The **isalnum( )** function returns nonzero if its argument is either a letter of the alphabet or a digit. If the character is not alphanumeric, zero is returned.

Syntax:- int isalnum(int *ch*);

isdigit( ) : The **isdigit( )** function returns nonzero if *ch* is a digit, that is, 0 through 9. Otherwise zero is returned.

Syntax:- int isdigit(int *ch*);

Library functions from stdio.h

printf(): The **printf( )** function writes to **stdout** the arguments that make up the argument list as specified by the string pointed to by *format*. The string pointed to by

format consists of two types of items. The first type is made up of characters that will be printed on the screen. The second type contains format specifiers that define the way the arguments are displayed. A format specifier begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order. For example, the following **printf( )** call displays **"Hi c 10 there!"**

printf("Hi %c %d %s", 'c', 10, "there!");

scanf(): The **scanf( )** function is a general-purpose input routine that reads the stream **stdin** and stores the information in the variables pointed to in its argument list. It can read all the built-in data types and automatically converts them into the proper internal format. The control string pointed to by *format* consists of three classifications of characters:

Format specifiers
White-space characters
Non-white-space characters

The input format specifiers begin with a % sign and tell **scanf( )** what type of data is to be read next. For example, **%s** reads a string, while **%d** reads an integer. The format string is read left to right, and the format specifiers are matched, in order, with the arguments that make up the argument list.

getchar(): The **getchar( )** function returns the next character from **stdin.** The character is read as an **unsigned char** that is converted to an integer. If the end of the file is reached, **getchar( )** returns **EOF**. If **getchar( )** encounters an error, **EOF** is also returned. The **getchar( )** function is often implemented as a macro.

Example:

```
#include <stdio.h>
int main(void)
{
char s[256], *p;
p = s;
while((*p++ = getchar())!= '\n') ;
*p = '\0';                          /* add null terminator */
printf("%s", s);
return 0;
}
```

Library functions from math.h: math.h contains functions to perform mathematical operations often required in c programs such as calculating absolute value of a number, calculating logarithms and using trigonometric functions to calculate sine, cosine of an angle.

ceil() : Ceil function is used to round up a number.

Declaration: double ceil(double);

floor(): Floor function returns the greatest integer not greater than x. For example if the input is 2.25 then output will be 2.00.

Declaration :- double floor(double x);

sqrt() : sqrt function returns square root of a number.

Declaration :- double sqrt(double);

Scanned by CamScanner

Q.8. (b) What is preprocessor directive? Explain different preprocess directives with examples. (6)

Ans. Refer Q.7. (a) from End Term Examination Dec. 2015.

Q.9. What is file pointer? Write a program in C that write 10 integers in da txt file and copy into the duplicate. txt. file. (12)

Ans. A file pointer is a pointer to a structure, which contains information about t file, including its name, current position of the file, whether the file is being read written, and whether errors or end of the file have occurred. The user does not need Know the details, because the definitions obtained from stdio.h include a structu declaration called FILE. The only declaration needed for a file pointer is symbolized

FILE *fp;

This says that fp is the file pointer that points to a FILE structure.

Program to write 10 integers in data.txt file and copy into the duplicate.t file

```
#include <stdio.h>
#include <conio.h>
int main()
{
 int i, num[10];
 FILE *fptr;
 FILE *fptr1;
 clrscr();
 fptr = fopen("data.txt", "w");
 if(fptr == NULL)
 {
   printf("Error!");
   exit(1);
 }
 for(i = 0; i < 10; i++)
 {
  printf("Enter number%d: ",i+1);
  scanf("%d", &num[i]);
  fprintf(fptr,"%d \n", num[i]);
 }
 fclose(fptr);
 fptr = fopen("data.txt", "r");
 fptr1 = fopen("duplicate.txt", "w");
 for(i = 0; i < 10; i++)
 {
  fscanf(fptr,"%d \n", num[i]);
  fprintf(fptr1,"%d \n", num[i]);
 }
 fclose(fptr);
 fclose(fptr1);
 return 0;
}
```

END TERM EXAMINATION [DEC. 2017]
FIRST SEMESTER [BCA]
INTRODUCTION TO PROGRAMMING LANGUAGE USING C [BCA–105]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.no.1 which is compulsory, Select one question from each unit.*

Q.1. (a) Whal are the keywords in C? Give examples. (2.5)

Ans. A keyword is a reserved identifier used by the language to describe a special feature. It is used in declarations to describe the basic type of an object, or in a function body to describe the statements executed. A keyword name cannot be used as an object name. All C keywords are lowercase names, so because lowercase and uppercase letters are different, the uppercase version of a keyword is available as an identifier although this is probably not a good programming practice.

The C keywords are:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Q. 1. (b) Explain various bitwise operators available in C. (2.5)

Ans. One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various Bitwise Operators available in C are shown

Operator	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (Exclusive OR)

These operators can operate upon **ints** and **chars** but not on **floats** and **doubles**.

One's Complement Operator: One's complement operator is represented by the symbol ~. One's complement of 65 means one's complement of 0000 0000 0100 0001, which is binary equivalent of 65 would be, 1111 1111 1011 1110.

Right Shift Operator : The right shift operator is represented by >>. It needs two operands. It shifts each bit in its left operand to the right. The number of places the bits are shifted depends on the number following the operator (i.e. its right operand). Note

Scanned by CamScanner

that as the bits are shifted to the right, blanks are created on the left, these bla
should be filled with zeros.

Left Shift Operator: This is similar to the right shift operator, the only differe
being that the bits are shifted to the left, and for each bit shifted, a 0 is added to
right of the number.

Bitwise AND Operator: This operator is represented as **&**. Remember it is differ
than **&&**, the logical AND operator. The **&** operator operates on two operands. Wh
operating upon these two operands they are compared on a bit-by-bit basis. Hence b
the operands must be of the same type (either **char** or **int**). The second operand is o
called an AND mask. The **&** operator operates on a pair of bits to yield a resultant b

Bitwise OR Operator: Another important bitwise operator is the OR operator wh
is represented as **|**. The rules that govern the value of the resulting bit obtained o
ORing of two bits is shown in the truth table below.

|	0	1
0	0	1
1	1	1

Bitwise XOR Operator: The XOR operator is represented as **^** and is also cal
an Exclusive OR Operator. The OR operator returns 1, when any one of the two bit
both the bits are 1, whereas XOR returns 1 only if one of the two bits is 1. The tr
table for the XOR operator is given below.

^	0	1
0	0	1
1	1	0

Q. 1. (c) What is the function of break and continue statement in C? (2.

Ans. Refer Q.1.(e) from EndTerm Examination December 2016.

Q. 1. (d) Briefly explain the concept of recursion.

Ans. A recursive function is a function that calls itself. Some problems can be ea
solved by using recursion, such as when you are dividing a problem into sub- proble
with similar natures. Recursion is a time-consuming solution that decreases the sp
of execution because of stack overheads. In recursion, there is a function call and t
number of such calls is large. In each call, data is pushed onto the stack and when t
call is over, the data is popped from the stack. These push and pop operations are ti
consuming operations.

Benefits of Recursion:-

1. Although at most of the times a problem can be solved without recursion, bu
some situations in programming, it is a must to use recursion. For example, a prog
to display a list of all files of the system cannot be solved without recursion.

2. The recursion is very flexible in data structure like stacks, queues, linked
and quick sort.

3. Using recursion, the length of the program can be reduced.

Q. 1. (e) What are macros in C? (2

Ans. Macro is a process where an identifier in a program is replaced by a predefi
string or value. #define is a preprocessor statement and is used to define macros.

Syntax 1: Macros without arguments

#define identifier predefined_string or value

Example :

#define pf printf

#define sf scanf

Syntax 2: Macros with arguments

#define identifier(arg-1,arg-2,.....arg-n) definition

Example :

#define square(x) x*x

Q. 1. (f) Differentiate between function declaration and function definition. **(2.5)**

Ans. Function Declaration: A function declaration tells the compiler about a function's name, return type, and parameters. It must end with a semicolon (;).

Example:

```
int add(int , int);
int add(int a, int b);
```

Function Definition: A function **definition** provides the actual body of the function.

Example:

```
int add(int a, int b){
  return a+b;
}
```

In short:

- A function declaration declares the existence of a function so you can call it.
- A function definition defines the function body, and defines what happens when the function is called. A function definition also acts as a function declaration.

Q. 1. (g) Name functions of standard library ctype.h. **(2.5)**

Ans. Functions of standard library ctype.h

S.No.	Function & Description
1	**isalpha( ):** This function returns nonzero if *ch* is a letter of the alphabet; otherwise zero is returned.
2	**isalnum( ):** This function returns nonzero if its argument is either a letter of the alphabet or a digit. If the character is not alphanumeric, zero is returned.
3	**isdigit( ):** This function returns nonzero if *ch* is a digit, that is, 0 through 9. Otherwise zero is returned.
4	**islower( )** This function check whether the passed character is lowercase letter.
5	**isprint( )** This function check whether the passed character is printable.
6	**ispunct( )** This function check whether the passed character is punctuation character.
7	**isspace( )** This function check whether the passed character is white-space.
8	**isupper( )** This function check whether the passed character is uppercase letter.

Q. 1. (h) What are nested control statements? Give examples. **(2.5)**

Ans. A nested control statement is a control statement that is contained within another control statement. You can do this to many levels, building up elaborate

Scanned by CamScanner

composites of various control statements. The compiler interprets these hierarchically, treating a composite control statement as a single statement for the purpose of determining what is affected by the next higher level of control.

Example: Consider this C fragment containing a for statement nested inside another.

```
for ( i=0; i<3; i++)
        for(j=0; j<4; j++)
                printf("%d %d\n", i, j);
```

Similarly, we can use nested statements for all the other control structures like if, switch, while, do_while etc.

Q. 1. (i) What is the need of pre-processor directives in C? (2.5)

Ans. The preprocessor is a program which processes the source code before it passes through the compiler. Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

The C preprocessor modifies a source file before handing it over to the compiler, allowing conditional compilation with #ifdef, defining constants with #define, including header files with #include, and using built in macros such as __FILE__.

Q. 1. (j) Differentiate between post increment and pre-increment operator with example.

Ans. Pre-increment operator(++x): value of x(any variable) is incremented before assigning it to the variable x(any variable) means first it will increment the value of x(any variable) and then it will assign the incremented value to x(any variable).

Consider the following .

x = 5;

x = ++x; (pre-increment)

In this final value of x would be 6.

m = 5;

y = ++m; (pre-increment)

In this case the value of y and m would be 6.

Post-increment operator(x++): value of x(any variable) is incremented after assigning it to the variable x means first it will assign the value of x without increment to the variable x and then it will increment the value of x.

Consider the following .

x = 5;

x = x++; (post-increment)

In this final value of x would be 6.

m = 5;

y = m++; (post-increment)

In this case the value of y and m would be 5 and 6 respectively.

UNIT-I

Q. 2. (a) Explain loop statements in C. (6.5)

Ans. Refer Q2(a) from EndTerm Examination December 2016.

Q. 2. (b) What are different types of operators in C? Briefly explain. (6)

Ans. Types of Operators:

Arithmetic Operators: These are the operators used to perform arithmetic/mathematical operations on operands. Examples: (+, -, *, /, %,++,–).

Arithmetic operator are of two types:

1. **Unary Operators**: Operators that operates or works with a single operand are unary operators.

For example: (++ , –)

2. **Binary Operators**: Operators that operates or works with two operands are binary operators. For example: (+ , – , * , /)

Relational Operators: Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc. Some of the relational operators are (==, <, >, != , <=, >=).

Logical Operators: Logical Operators are used to combine two or more conditions/constraints(relational expressions) or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false(0 or 1). Examples: Logical OR(| |), Logical AND(&&), Logical NOT(!)

Assignment Operators(=): Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.

In addition, C has a set of shorthand assignment operators/compound assignment operator of the form.

var *oper* = exp;

Here var is a variable, exp is an expression and *oper* is a C binary arithmetic operator. The operator oper = is known as shorthand or compound assignment operator

Example

x + = 1 is same as x = x + 1

Statement with simple assignment operator	Statement with shorthand operator
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n+1)	a *= (n+1)
a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

Increment and Decrement Operators: The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below:-

1. ++ variable name

Scanned by CamScanner

2. variable name++
3. – –variable name
4. variable name– –

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator – – subtracts the value 1 from the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following .

m = 5;
y = ++m; (pre-increment)

In this case the value of y and m would be 6.

Suppose if we rewrite the above statement as

m = 5;
y = m++; (post-increment)

Then the value of y will be 5 and that of m will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

Conditional or Ternary Operator: The conditional operator consists of 2 symbols the question mark (?) and the colon (:)

The syntax for a ternary operator is as follows:-

exp1 ? exp2 : exp3

The ternary operator works as follows :-

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

a = 10;
b = 15;
x = (a > b) ? a : b

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

Bitwise Operators: The Bitwise operators is used to perform bit-level operations on the operands. The operators are first converted to bit-level and then calculation is performed on the operands. The mathematical operations such as addition , subtraction , multiplication etc. can be performed at bit-level for faster processing.

The various Bitwise Operators available in C are shown

Operator	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR(Exclusive OR)

Special Operators: C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->).

The Comma Operator: The **comma operator** (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, and then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

<u>sizeof() Operator</u>

- The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.
- Examples:- m = sizeof (sum);

n = sizeof (long int);
k = sizeof (235L);

- The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

Q. 3. (a) What is an array? Write a program to print average of all elements of an array. **(6.5)**

Ans. An array can be defined as a collection of homogeneous data elements (i.e. data elements of the same data types) stored in the continuous memory locations which is described by a single name (i.e. name of the array). The elements in an array share the same name and are distinguished from one another by their numbers or subscripts. The subscripts are always 0,1,.., size-1.

Program to print average of all elements of an array

```
void main()
{
int a[100], i, n, sum=0;
clrscr();
printf("Enter size of array\n")
scanf("%d", &n);
printf("Enter elements in array\n")
for(i=0; i < n; i++)
{
scanf("%d", &a[i]);
sum = sum + a[i];
}
printf("Average of all elements of array is %d", sum / n);
getch();
}
```

Q. 3. (b) Differentiate between while and do-while loop constructs with example. **(6)**

Scanned by CamScanner

Ans.

while Loop	do_while Loop
In while loop, test condition is checked first, and then body of the loop will be executed.	In do_while loop, body of the loop will be executed first, and then condition is checked.
If test condition is false, body of the loop will not be executed.	Body of the loop will be executed once even if test condition is false.
while loop is Entry Controlled Loop.	do_while loop is Exit controlled loop.
It is used when checking of test condition is necessary before executing loop body.	It is used when checking of test condition is necessary after executing the loop body.

Example:
void main()
{
int n=1, sum=0;
while (n<=10)
{
sum=sum+n;
n=n+1;
}
printf("Sum of first 10 natural numbers: %d",sum);
}
	Example:
	void main()
	{
	int n=1, sum=0;
	do
	{
	sum=sum+n;
	n=n+1;
	} while(n<=10);
	printf("Sum of first 10 natural numbers: %d", sum);
	}

UNIT–II

Q. 4. (a) With the help of a program. Explain the concept of call by value and call by reference. **(6.5)**

Ans. Refer Q3(b) from EndTerm Examination December 2016.

Q. 4. (b) (i) Define the following terms: **(6)**

(i) Static Variable

Ans. Refer Q1(i) from EndTerm Examination December 2015.

Q. 4. (b) (ii) Pointers

Ans. Refer Q1(d) from EndTerm Examination December 2015.

Q. 4. (b) (iii) Dynamic Memory Allocation

Ans. The process of allocating memory at runtime is known as dynamic memory allocation. Library routines known as memory management functions are used for allocating and freeing memory during execution of a program. These functions are defined in stdlib.h header file.

Function	Description
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Q. 5. (a) Explain the concept of recursion by writing a program to find factorial of any given number. **(6.5)**

Ans. Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A recursive function is a function that calls itself.

```
//program to find factorial using recursion
#include<stdio.h>
#include<conio.h>
long int factorial(int num)
{
if(num==0 || num==1)
        return 1;
else
        return num*factorial(num-1);
}
void main()
{
int n;
long int fact;
clrscr();
printf("\nEnter any number :");
scanf("%d", &n);
fact = factorial(n);
printf("\n\nFactorial : %ld", fact);
getch();
}
```

Explanation:

You can express factorials by using recursion as shown:

fact (5) = 5 * fact (4)

In general,

fact (N) = N * fact (N-1)

fact 5 is calculated as follows:

fact (5) = 5 * fact (4) i.e. there is call to fact (4)

fact (4) = 4 * fact (3)

fact (3) = 3 * fact (2)

fact (2) = 2 * fact (1)

fact (1) = 1 * fact (0)

fact (0) = 1

fact (1) = 1 * 1, that is the value of the fact(0) is substituted in 1.

fact (2) = 2 * 1 = 2

fact (3) = 3 * 2 = 6

fact (4) = 4 * 6 = 24

fact (5) = 5 * 24 = 120

Scanned by CamScanner

Q. 5. (b) How array of pointer are passed as arguments in a function. Give example. **(8)**

Ans. Example to illustrate array of pointers passed as arguments in a function:

```
#include<stdio.h>
void display(int *p[])
{
 int i;
 for(i=0;i<4;++i)
 printf("%d ",*p[i]);
}
int main ()
{
 int a=6,b=3,c=4,d=50;
 int *p[4];
 clrscr ();
 p[0]=&a;
 p[1]=&b;
 p[2]=&c;
 p[3]=&d;
display(p);
 getch ();
 return 0;
}
```

UNIT–III

Q. 6. (a) Example the concept and need of structures in C. How is union different from structure? **(6.5)**

Ans. Need of structures: We have seen that arrays can be used to represent a group of data items that belong to the same type. However in real world, we often deal with entities that are collection of dissimilar data items. Using array variables we cannot store different items under one name. C supports a derived data type known as structure, which is a method for storing data of different types.

Definition: A structure is a collection of logically related elements, possibly of different types, having a single name. A structure definition forms a template that may be used to crate structure objects. The variables that make up the structure are called members of the structure.

Example of structure:

```
#include <stdio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
```



```
int main()
{
        struct student record = {0};    //Initializing to null
        record.id=1;
        strcpy(record.name, "Amit");
        record.percentage = 86.5;
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

Structure vs Union

Structure	Union
Struct keyword is used to define a structure.	Union keyword is used to define a union.
Members do not share memory in a structure.	Members share the memory space in a union.
Any member can be retrieved at any time in a structure.	Only one member can be accessed at a time in a union.
Several members of a structure can be initialized at once.	Only the first member can be initialized.
Size of the structure is equal to the sum of size of the each member.	Size of the union is equal to the size of the largest member.
Altering value of one member will not affect the value of another.	Change in value of one member will affect other member values.
Stores different values for all the members.	Stores same value for all the members.

Q. 6. (b) What is file handling? What are various functions used in file handling Briefly explain. **(6)**

Ans. A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file. File handling is the concept of working with file in order to store and retrieve data from the files.

File Handling Functions:

Function	Syntax	Example

fopen - Open file	FILE * fopen(char * filename, char * mode)Filename – string filename with pathMode – mode of opening
	
	r read only
	w write only
	a append file
	+ read and write (r+/w+/a+)
	Return value – **FILE pointer** on success **NULL** on failure
		FILE * infile ;
		Infile = fopen("a.txt", "r");
		FILE *outfile = fopen ("b.txt","w");
		if(infile==NULL
		|| outfile==NULL)
		{
		printf("error");
		}

Scanned by CamScanner

fclose – close the open file	fclose(FILE *fptr) fptr – FILE * of the open file Returns void	fclose(infile);

fgetc – read one char from file	int fgetc(FILE *fptr)
	fptr – FILE * of the open file
	returns – character read
	-1 on eof or error
		while((ch=fgetc())!=-1)
		putchar(ch);//display content of file
		//on screen

fputc – write one character to the file	int fputc(int ch,FILE *fptr) ch – character to writefptr – open file pointerreturns character written EOF if there is error	for(ch= 'A';ch<='Z';ch++) fputc(ch,fptr);
fscanf – Formatted read from file	int fscanf(FILE *ptr,char * fmt,..) ptr – FILE pointerfmt – format specifier for variables to be read Returns – number of values read successfully	fscanf(infile, "%d %d",&n1, &n2);
fprintf – formatted write to file	int fprintf(FILE *ptr,char * fmt,..) ptr – FILE pointer fmt – format specifier for variables to be readReturns – number of characters printed or negative number on error	fprintf(outfile,"Value is %d\n",n1);
fread – read from binary file	int fread(void *buffer, size_t size, size_t num, FILE *ptr)buffer – stores the value readsize – size of the buffernum – number of blocks to be readptr – file pointer Returns – number of values read	Char *str = malloc (50);fread(str,50,1,infile);

fwrite – write to a binary file	int fwrite(void *buffer, size_t size, size_t num, FILE *ptr)
	buffer –stores the value read
	size – size of the buffernum – number of blocks to be readptr – file pointer
	Returns – number of values written
		for(i=0;i<10;i++)
		write(&,sizeof(i),1,1,outfile);
		char a[]="end";
		fwrite (a,strlen(a)+1,1, outfile);

fseek – move the file pointer by given offset	int fseek(FILE*ptr,long offset,int whence)ptr – file pointeroffset – offset in bytes from third parameter
	whence – SEEK_SET – from beginning of file
	SEEK_CUR – from current position
	SEEK_END – from end of file
	Returns – zero on success
	Non-zero on failure
		if(fseek(infile,4L,SEEK_SET) ==0)
		{
		char ch=fgetc(infile);
		printf("The fourth char of the file is %c\n",ch);
		}

ftell – get the position of file pointer	int ftell(FILE *ptr)ptr – FILE pointerReturns – position of file pointer-1 on error	FILE *ptr = fopen(b[1],"r"); fseek(ptr,0L,SEEK_END); int size_of_file = ftell(ptr);
rewind – moves the file pointer to the beginning of the file	void rewind(FILE *ptr)ptr – FILE pointer	rewind(infile); int n = fscanf(infile,"%d",&n);

Q. 7. What is file pointer? Write a program in C to create a file, write string data into it and count number of characters in that file. **(12.5)**

Ans. A file pointer is a pointer to a structure, which contains information about the file, including its name, current position of the file, whether the file is being read or written, and whether errors or end of the file have occurred. The user does not need to Know the details, because the definitions obtained from stdio.h include a structure declaration called FILE. The only declaration needed for a file pointer is symbolized by

FILE *fp;

This says that fp is the file pointer that points to a FILE structure.

Program to create a file and count number of characters in the file

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char str[50],ch;
int ch_count=0,i;
clrscr();
fp=fopen("INPUT.txt","w");
printf("Enter text to be stored in file Input.txt\n");
gets(str);
fprintf(fp, "%s", str);
fclose(fp);
fp=fopen("INPUT.txt","r");
printf("\n\nContents of the file Input.txt\n");
while(ch!=EOF)
{
ch=fgetc(fp);//Reading a character from file.
putchar(ch);  //Writing character to the screen.
if((ch>=65 && ch<=90) || (ch>=97 && ch<=122))
{
        ch_count=ch_count+1;
}
}
printf("\n\nNumber of Characters in File = %d\n",ch_count);
fclose(fp);
getch();
}
```

UNIT-IV

Q. 8. (a) What is array of strings? Describe various string manipulation functions with example.

Ans. Array of Strings: we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters or array of strings. C provides a more compact way to declare this:

```
char ch_arr[3][10] = {
        "Amit",
        "Hemant",
        "Ashish"
};
```

Scanned by CamScanner

The first subscript of the array i.e 3 denotes the number of strings in the array and the second subscript denotes the maximum length of the string.

String manipulation functions: The C Library provides a rich set of string handling functions that are placed under the header file <string.h>.

S.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strlwr(s1);** Converts string to lowercase
6	**strupr(s1);** Converts string to uppercase
7	**strrev(s1);** returns reverse of the string

Example:

The following example uses some of the above-mentioned functions:

```
#include <stdio.h>
#include <string.h>
int main ()
{
  char str1[12] = "Hello";
  char str2[12] = "World";
  char str3[12];
  int  len ;
  /* copy str1 into str3 */
  strcpy(str3, str1);
  printf("strcpy( str3, str1) :  %s\n", str3 );
  /* concatenates str1 and str2 */
  strcat( str1, str2);
  printf("strcat( str1, str2):   %s\n", str1 );
  /* total length of str1 after concatenation */
  len = strlen(str1);
  printf("strlen(str1) :  %d\n", len );
  /* Reverse of string */
  printf("Reverse of str1 :  %s\n", strrev(str1) );
  /* String in upper case*/
  printf("String in upper case :  %s\n", strupr(str1) );
  /* String in lower case*/
  printf("String in lower case :  %s\n", strlwr(str1) );
  return 0;
}
```

Q. 8. (b) What are standard libraries in C? Give example of functions in libraries stdio. h, math,h, and string.h. (6)

Ans. The C standard library is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages C does not include built-in keywords for these tasks, so nearly all C programs rely on the standard library to operate.

The C programming language previously did not provide any elementary functions, such as I/O operations. Over time, user communities of C shared ideas and implementations to provide those functions. These ideas became common, and were eventually incorporated into the definition of the standardized C programming language in 1989. These are now called the C standard libraries.

1. math.h

math.h contains functions to perform mathematical operations often required in c programs such as calculating absolute value of a number, calculating logarithms and using trigonometric functions to calculate sine, cosine of an angle.

ceil()

Ceil function is used to round up a number.

Declaration: double ceil(double);

floor()

Floor function returns the greatest integer not greater than x. For example if the input is 2.25 then output will be 2.00.

Declaration :- double floor(double x);

sqrt()

sqrt function returns square root of a number.

Declaration :- double sqrt(double);

2. stdio.h

printf()

The **printf( )** function writes to **stdout** the arguments that make up the argument list as specified by the string pointed to by *format*. The string pointed to by *format* consists of two types of items. The first type is made up of characters that will be printed on the screen. The second type contains format specifiers that define the way the arguments are displayed. A format specifier begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order. For example, the following **printf( )** call displays **"Hi c 10 there!"**

```
printf("Hi %c %d %s", 'c', 10, "there!");
```

scanf()

The **scanf( )** function is a general-purpose input routine that reads the stream **stdin** and stores the information in the variables pointed to in its argument list. It can read all the built-in data types and automatically converts them into the proper internal format. The control string pointed to by *format* consists of three classifications of characters:

Format specifiers

White-space characters

Non-white-space characters

The input format specifiers begin with a % sign and tell **scanf( )** what type of data is to be read next. For example, **%s** reads a string, while **%d** reads an integer. The format string is read left to right, and the format specifiers are matched, in order, with the arguments that make up the argument list.

getchar()

The **getchar( )** function returns the next character from **stdin.** The character is read as an **unsigned char** that is converted to an integer. If the end of the file is reached, **getchar( )** returns **EOF.** If **getchar( )** encounters an error, **EOF** is also returned. The **getchar( )** function is often implemented as a macro.

Scanned by CamScanner

3. string.h

(a) strlen( ) – String Length function

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

Syntax:- int length = strlen ("String") ;

(b) strcpy() - String Copy function

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function.

Syntax:- strcpy (target, source) ;

(c) strcmp() - String Compare function

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. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is an example:-

Syntax:- int j = strcmp (string1, string2) ;

Q. 9. Write a program to first compare and then concatenate two strings entered by user. **(12.5)**

Ans. Program to compare and concatenate two strings entered by the user:

```
#include <stdio.h>
int main()
{
   char s1[100], s2[100], i, j;
   printf("Enter first string: ");
   scanf("%s",s1);
   printf("Enter second string: ");
   scanf("%s",s2);
if (compare_strings(s1, s2) == 0)
    printf("Equal strings.\n");
  else
    printf("Unequal strings.\n");
/*Logic for concatenation of two strings */
   for(i=0; s1[i]!='\0'; ++i);            /* i contains length of string s1. */
   for(j=0; s2[j]!='\0'; ++j, ++i)
   {
      s1[i]=s2[j];
   }
   s1[i]= '\0';
   printf("After concatenation: %s",s1);
   return 0;
}
int compare_strings(char a[], char b[])
{
  int c = 0;
  while (a[c] == b[c]) {
    if (a[c] == '\0' || b[c] == '\0')
      break;
    c++;
  }
  if (a[c] == '\0' && b[c] == '\0')
    return 0;
  else
    return -1;
}
```

END TERM EXAMINATION [NOV. DEC. 2018]
FIRST SEMESTER [BCA]
INTRODUCTION TO PROGRAMMING LANGUAGE USING C [BCA–105]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions in all including Q. no. 1 which is compulsory. Select one question from each unit.*

Q.1. Write short notes of the following (Any Five):

(a) What you mean by pre increment and post-increment operator? Explain with an example. **(5)**

Ans. Refer Q.1.(j) from End Term Examination December 2017.

Q. 1. (b) Differentiate between structure and union. Explain the use of "extern" keyword. **(5)**

Ans.

	Structures	Unions
Keyword	struct	union
Definition	A structure is a collection of logically related elements, possibly of different types, having a single name.	A union is a collection of logically related elements, possibly of different types, having a single name shares single memory location.

Declaration	struct tag _ name
	{
	type 1 member 1 ;
	type 1 member 2 ;
	.....
	.....
	};
	struct tag _ name var;
		union tag_ name
		{
		type 1 member 1 ;
		type 1 member 2 ;
		.....
		.....
		};
		union tag _ name var;

Memory Allocation	Each member of the structure occupies unique location, stored in contigenous locations.	Memory is allocated by considering the size of largest member. All the members share the common location
Size	Size of the structure depends on the type of members, adding size of all members. sizeof (st_var);	Size is given by the size of largest member storage. size of (–unvariable);
Access	All members can be accessed at a time	Only one member can be accessed at a time.

Example:

```
struct first
{
char c;
long l;
```

Scanned by CamScanner

```
char *p;
};
union second
{
char c;
long l;
char *p;
};
```

A struct first contains all of the elements c, l, and p. Each element is separate and distinct.

A union second contains only one of the elements c, l, and p at any given time. Each element is stored in the same memory location (well, they all start at the same memory location), and you can only refer to the element which was last stored. (ie: after "secondptr->c = 2;" you cannot reference any of the other elements, such as "secondptr->p" without invoking undefined

Q. 1. (c) Explain the concept of pointer? Define the dynamic memory allocation and its various functions. **(5)**

Ans. A pointer is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds an integer value, however an integer pointer holds the address of a integer variable.

Declaring a pointer variable: In C, every variable must be declared before they are used. Since the pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them.

The syntax for declaring a pointer variable is as follows,

data type *ptr_name;

This tells the compiler three things about the variable ptr_name.

1.The asterisk(*) tells that the variable ptr_name is a pointer variable.

2.ptr_name needs a memory location.

3.ptr_name points to a variable of type data type.

For example,

int *pi;

declares the variable p as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed by pi.

Initializing Pointers

- Once a pointer variable has been declared, it can be made to point to a variable using statement such as

ptr_name=&var;

- Which cause ptr_name to point to var.Now ptr_name contains the address of var. This is known as pointer initialization.
- Before a pointer is initialized it should not be used.

Access the value of a variable using pointer variable

Once a pointer variable has been assigned the address of a variable, we can access the value of a variable using the pointer. This is done by using the indirection operator(*).

*ptr_name

Function	Working
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

Q. 1. (d) Differentiate between static and register storage class in C language. **(5)**

Ans. Register Variable:

- Register tells the compiler that the variables should be stored in high-speed memory registers if possible. This is done for efficiency.
- Only a few such registers are available, and can be assigned to frequently-accessed variables if execution time is a problem.
- If no register space is free, variables become auto instead. Keep registers in small local blocks which are reallocated quickly.
- A register variable address is not available to the user .this means we can't use the address operator and the indirection operator (pointer) with a register.
- A variable with register specification has the following storage characteristic:

Scope: block Extent: automatic/Register Linkage: internal

Declaration: register type variable_Name;

Initialization: A register variable can be initialized where it is defined or left uninitialized. When auto variables are not initialized its value is garbage value.

Static Variables :

- A static variable that is declared in a block scope, static defines the extent of the variable.
- The computer allocates storage only once. The linkage is internal, and not visible to other blocks.
- Also Known as static Local Variable.
- A variable with an static block specification has the following storage characteristic:

Scope: block Extent: static Linkage: internal

Declaration: static type variable_Name;

Initialization: A static variable name can be initialized where it is defined, or it can be left uninitialized. If it is initialized, it is initialized only once. If not initialized, its value will be initialized to zero.

Q. 1. (e) What is a string? Write program for the concatenation of two strings. **(5)**

Ans. Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

Declaration of strings: Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

char str[size];

Scanned by CamScanner

Program to concatenate two strings:

```
#include <stdio.h>
int main()
{
  char s1[100], s2[100], i, j;
  printf("Enter first string: ");
  scanf("%s",s1);
  printf("Enter second string: ");
  scanf("%s",s2);
  for(i=0; s1[i]!='\0'; ++i); /* i contains length of string s1. */
  for(j=0; s2[j]!='\0'; ++j, ++i)
  {
    s1[i]=s2[j];
  }
  s1[i]='\0';
  printf("After concatenation: %s",s1);
  return 0;
}
```

Q. 1. (f) What are preprocessor directives? From which symbol do they start? (5)

Ans. The preprocessor is a collection of special statements called commands or preprocessor directives. Each directive is examined by the preprocessor before the source program passes through the compiler.

- Statements beginning with # are considered preprocessor directives.
- Preprocessor directives indicate certain things to be done to the program code prior to compilation.
- It includes certain other files when compiling, replace some code by other code.
- Only white space characters before directives on a line.
- Preprocessor commands can be placed anywhere in the program.

There are three major tasks of a preprocessor directive

1. Inclusion of other files (file inclusion)
2. Definition of symbolic constants and macros(macro definition)
3. Conditional compilation of program code/Conditional execution of preprocessor directives

UNIT-I

Q. 2. (a) Explain while loop and do- while loop with an example. Elaborate Difference between them. (4.5)

Ans. Refer Q.3.(b) from End Term Examination December 2017.

Q. 2. (b) Write a program to find the largest number among the three numbers. Write a program to generate Fibonacci series upto first 20 numbers. (8)

Ans.

```
# include<stdio.h>
void main()
{
    int a,b,c;
    printf("Enter any 3 numbers:\n");
    scanf("%d%d%d", &a, &b, &c);
    printf("Maximum: %d", a>b?a:b>c?b:c);
    getch();
```



```
}
#include<stdio.h>
#include<conio.h>
void main()
{
int a=0,b=1,i,c,n;
clrscr();
printf("ENTER HOW MANY TERMS OF FIBONACCI SERIES IS TO BE PRINTED:");
scanf("%d",&n);
printf("%d %d",a,b);
for(i=1;i<=n-2;i++)
{
c=a+b;
a=b;
b=c;
printf(" %d",c);
}
getch();
}
```

Output:

```
ENTER HOW MANY TERMS IS TO BE PRINTED : 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181_
```

Q. 3. (a) Explain the Switch statement, Break statement, and Continue statement with an example. (6)

Ans. Switch statement: 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);
}
```

Rules apply to a switch statement

- The expression used in a switch statement must have an integral type (i.e. either int or char datatype only).
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

Scanned by CamScanner

- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:-

```
#include <stdio.h>
int main ()
{
char grade = 'B';
        switch(grade)
        {
        case 'A':
                printf("Excellent!\n" );
                break;
        case 'B':
        case 'C':
                printf("Well done\n" );
                break;
        case 'D':
                printf("You passed\n" );
                break;
        case 'F':
                printf("Better try again\n" );
                break;
        default :
                printf("Invalid grade\n" );
        }
printf("Your grade is %c\n", grade );
return 0;
}
```

Break statement:-

When break is encountered inside any loop, control automatically passes to the first statement after the loop. A break is usually associated with an if. The break statement in C programming language has following two usage:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

2. It can be used to terminate a case in the switch statement.

If you are using nested loops (i.e. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a break statement in C is as follows:

```
break ;
```


Example:-

```
void main( )
{
        int num, i ;
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        i = 2 ;
        while ( i <= num - 1 )
        {
                if ( num % i == 0 )
                {
                printf ( "Not a prime number" ) ;
                break ;
        }
        i++ ;
        }
if ( i == num )
        printf ( "Prime number" ) ;
}
```

The keyword break, breaks the control only from the while in which it is placed.

Continue statement:- continue statement is a jump statement. The continue statement can be used only inside for loop, while loop and do-while loop. Execution of these statement does not cause an exit from the loop but it suspend the execution of the loop for that iteration and transfer control back to the loop for the next iteration.

Example:-

```
void main( )
{
int i, j ;
        for ( i = 1 ; i <= 2 ; i++ )
        {
                for ( j = 1 ; j <= 2 ; j++ )
                {
                        if ( i == j )
                                continue ;
                        printf ( "\n%d %d\n", i, j ) ;
                }
        }
}
```

Q. 3. (b) Write a program to find the factorial of a number using recursion. **(6.5)**

Ans. Refer Q. 5. (a) from EndTerm Examination December 2017.

UNIT-II

Q. 4. (a) What do you mean by pointer? How pointer variable are initialized? Write a note on pointer arithmetic with a proper example in C language. **(6.5)**

Scanned by CamScanner

Ans. A pointer is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds an integer value, however an integer pointer holds the address of a integer variable.

Declaring a pointer variable: In C, every variable must be declared before they are used. Since the pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them.

The syntax for declaring a pointer variable is as follows,

data type *ptr_name;

This tells the compiler three things about the variable ptr_name.

4. The asterisk(*) tells that the variable ptr_name is a pointer variable.
5. ptr_name needs a memory location.
6. ptr_name points to a variable of type data type.

For example,

int *pi;

declares the variable p as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed by pi.

Initializing Pointers

- Once a pointer variable has been declared, it can be made to point to a variable using statement such as

ptr_name = &var;

- Which cause ptr_name to point to var.Now ptr_name contains the address of var. This is known as pointer initialization.
- Before a pointer is initialized it should not be used.

Access the value of a variable using pointer variable

Once a pointer variable has been assigned the address of a variable, we can access the value of a variable using the pointer. This is done by using the indirection operator(*).

***ptr_name**

POINTER ARITHMETIC: The following operations can be performed on a pointer:

- Addition of a number to a pointer. Pointer can be incremented to point to the next locations.

Example:

```
int i=4, pi=&i;          //(assume address of i=1000)
float j,*pj=&j;          // (assume address of j=2000)
pi = pi + 1;             // here pi incremented by (1*data type times)
pi = pi + 9;             // pi = 1000 + (9*2) à 1018 address
pj = pj + 3;             // pj=2000+(3*4)à2012 address
```

- Subtraction of a number from a pointer. Pointer can be decremented to point to the earlier locations.

Example:

```
int i=4,*pi=&i;          //assume address of i =1000)
char c, *pc=&c;          // assume address of c = 2000
double d, *pd=&d;        // assume address of d=3000
pi = pi-2;               /* pi=1000-(2*2)=996 address */
```



```
pc = pc-5;               /* pc=2000-(5*1)=1985 address
pd = pd-6;               /* pd=3000-(6*8)=2952 address */
```

- Pointer variables may be subtracted from one another. This is helpful while finding array boundaries. Be careful while performing subtraction of two pointers.
- Pointer variables can be used in comparisons, but usually only in a comparison to **NULL**.
- We can also use increment/decrement operators with pointers this is performed same as adding/subtraction of integer to/from pointer.

The following operations cannot be performed on pointers.

- Addition of two pointers.
- Multiplication of a pointer with a constant, two pointers.
- Division of a pointer with a constant, two pointers.

Example:

```
#include <stdio.h>
int main()
{
  int arr[5] = {10, 20, 30, 40, 50};
  int *ptr;
  int count;
  ptr = &arr[0];                    // ptr points to arr[0]
  count = 0;
  printf("Accessing array elements using pointer \n");
  while(count < SIZE)
  {
    printf("arr[%d] = %d \n", count, *(ptr + count));
    count++;
  }
  return 0;
}
```

Q. 4. (b) Write a program to find the out maximum number in array using pointer. **(6)**

Ans. Program to find maximum number in an array using pointers:

```
#include <stdio.h>
int main()
{
  int a [100], *maximum, n, i, location = 1;
  printf("Enter the number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d elements\n", n);
  for ( i = 0 ; i < n ; i++ )
    scanf("%ld", &a[i]);
  maximum = a;
```

Scanned by CamScanner

```
*maximum = *a;
for (i = 1; i < n; i++)
{
 if (*(a+i) > *maximum)
 {
   *maximum = *(a+i);
   location = i+1;
 }
}
printf("Maximum element is found at location %d and it's value is %d.\n", location, *maximum);
return 0;
}
```

Q. 5. (a) Explain the array of pointers to functions with suitable example. (4.5)

Ans. An array of function pointers can be created for the functions of same return type and taking same type and same number of arguments. For example, if you have four function definitions for performing basic arithmetic operations (addition, subtraction, multiplication, and division) on two integer quantities, all these functions will take two integer arguments and will return an integer as a result.

Example:

```
#include <stdio.h>
int sum(int, int);
int difference(int, int);
int product(int, int);
int division(int, int);
int main()
{
 typedef int (*functionPtr)(int, int);   // type-def: 'functionPtr' now can be used as type
 int a = 30, b = 20;
 // assign the function's address
 functionPtr arrayFp[4] = {sum, difference, product, division};
 printf ("Sum of %d and %d : %d\n", a, b, arrayFp[0](a, b));
 printf ("Difference of %d and %d : %d\n", a, b, arrayFp[1](a, b));
 printf ("Product of %d and %d : %d\n", a, b, arrayFp[2](a, b));
 printf ("Division of %d and %d : %d\n", a, b, arrayFp[3](a, b));
printf ("\nProduct of sum and difference of %d and %d is : %d\n", a, b, arrayFp[2](arrayFp[0](a, b), arrayFp[1](a, b)));
}
int sum(int x, int y)
{
 return x + y;
```



```
}
int difference(int x, int y)
{
 return x - y;
}
int product(int x, int y)
{
 return x * y;
}
int division(int x, int y)
{
 return x / y;
}
```

Q. 5. (b) Write the difference between call by value and call by reference with suitable example. (8)

Ans. Call by value: In this, the value of actual arguments is passed to the formal arguments and operation is done on the formal arguments. Any change in the formal argument made does not affect the actual arguments because formal arguments are the photocopy of the actual argument. Hence, when a function is called by the call by value method, it does not affect the actual contents of the arguments. Changes made in the formal arguments are local to the block of the called function. Once control returns back to the calling function, changes made vanish.

Call-by-reference: In this, the address of the actual parameter is passed by the calling routine to the called function so that it can be altered there. Execution of a function using call-by-reference can take longer than one using call-by-value, since in call-by-reference indirect mode instructions are needed for any calculations involving the variables passed.

Example:

```
void main( )
{
int a = 10, b = 20 ;
swapv ( a, b ) ;                    //Call by value
printf ( "\na = %d b = %d", a, b ) ;
swapr ( &a, &b ) ;                  //Call by reference
printf ( "\na = %d b = %d", a, b ) ;
}
void swapv ( int x, int y )         //Pass by value
{
int t ;
t = x ;
x = y ;
y = t ;
printf ( "\nx = %d y = %d", x, y ) ;
}
void swapr( int *x, int *y )        //Pass by reference
```

Scanned by CamScanner

```
{
int t;
t = *x;
*x = *y;
*y = t;
}
```

Call by value	Call by Reference
Formal parameter is a new local variable that exists within the scope of the procedure/function.	Formal parameter is an alias for the actual parameter.
Value of actual parameter is used to initialize the formal parameter. Changes made to the formal parameter do not get transmitted back to the caller.	Formal parameter refers to the same memory cell for the actual parameter. Each formal parameter is a reference to the actual parameter and modifying the formal will modify the actual.

UNIT-III

Q.6. Explain the concept of structure. Write a C program to stores the information (name, roll, Branch, dept., and marks of 10 students using structures. Find who has topped in the class? **(12.5)**

Ans. A structure is a collection of logically related elements, possibly of different types, having a single name.

Advantages:-

- Structures help to organize complex data in a more meaningful way.
- Being able to manipulate several variables as a single group makes your programs easier to manage.
- Business data processing uses the concepts of structures in almost every program. Example, student database and employee database management.

Good candidates for structures:-

Time: year, month, day, hour, minutes, second

Complex number: real part, imaginary part

Point in a 2-dimensional plane: x axis, y axis

Definition:

As variables are defined before they use in the program, structures are also defined and declared before they are used.

A structure definition forms a template that may be used to crate structure objects. The variables that make up the structure are called members of the structure.

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

Syntax:

```
struct [structure_tag]
{
  //member variable 1
  //member variable 2
```



```
  //member variable 3
  ...
} structure_variables;
```

As shown in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Program:

```
#include <stdio.h>
struct student
{
char name[80];
int rollno;
char branch[80];
char department[80];
int marks;
};
void accept(struct student[], int);
void display(struct student[], int);
int findMax(struct student[], int);
void topper(struct student[], int);
void accept(struct student list[60], int s)
{
int i;
for (i = 0; i < s; i++)
{
printf("\nEnter data of student #%d", i + 1);
printf("Enter name : ");
gets(list[i].name);
printf("\nEnter rollno : ");
scanf("%d", &list[i].rollno);
printf("Enter branch : ");
gets(list[i].branch);
printf("Enter department : ");
gets(list[i].department);
printf("Enter marks : ");
scanf("%d", &list[i].marks);
}
}
void display(struct student list[60], int s)
{
```

Scanned by CamScanner

```
int i;
printf("\n\nName\t Rollno\t Branch\t Department\t Marks\n");
for (i = 0; i < s; i++)
{
printf("%s\t%d\t%s\t%s\t%d\n", list[i].name, list[i].rollno, list[i].branch,
list[i].department list[i].marks);
}
}
int findMax(struct student list[], int s)
{
int i, max;
max = list[0].marks;
for (i = 1; i < s; i++)
{
if (list[i].marks > max)
{
max = list[i].marks;
}
}
return max;
}
void topper (struct student list[], int s)
{
int i;
for (i = 0; i < s; i++)
{
if (list[i].marks == findMax(list, s))
{
printf("%s\n", list[i].name);
}
}
}
int main()
{
struct student records[60];
int n, choice, rollno;
printf("Number of records you want to enter? : ");
scanf("%d", &n);
accept(records, n);
do
{
printf("\nResult Menu :\n");
```



```
printf("Press 1 to display all records.\n");
printf("Press 2 to display topper name.\n");
printf("Press 0 to exit\n");
printf("\nEnter choice(0-2) : ");
scanf("%d", &choice);
switch (choice)
{
case 1: display(records, n);
 break;
case 2: topper(records, n);
}
} while (choice != 0);

return 0;
}
```

Q. 7. Write a short note on each. **(12.5)**

(i) How to pass a structure to a function?

Ans. Structure as function parameter: A structure variable can be passed to the function as an argument/parameter as a normal variable. If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable. If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

Example:

```
#include <stdio.h>
struct student
{
  char sname[30];
  int rollno;
  float avg;
};
void display(struct student);
          // function prototype should be below to the structure declaration
          otherwise compiler shows error
void main()
{
  struct student s1;
  printf("Enter student's name:");
  scanf("%s", s1.sname);
  printf("Enter roll number:");
  scanf("%d", &s1.rollno);
  printf("Enter average:");
  scanf("%f", &s1.avg);
  display(s1);          // passing structure variable s1 as an parameter
```

Scanned by CamScanner

```
  getch();
}
void display(struct student s)
{
  printf("Output: \nName: %s", s.sname);
  printf("\nRoll No: %d", s.rollno);
  printf("\nAverage: %f", s.avg);
}
```

Q. 7. (ii) Bit fields:

Ans. Bit Fields: If in a program a variable is to take only two values 1 and 0, we really need only a single bit to store it. If a variable is to take values from 0 to 3, then two bits are sufficient to store these values. So, we cannot waste entire integer to store such data.

There are several variables whose max values are small enough to pack into a single memory location. We can use 'bit fields' to store several values in a single integer.

A bit field is a set of adjacent bits whose size can be from 1 to 16 bits length. A word can therefore be divided in to a number of bit fields. The name and size of bit fields are defined using structure. Since bit fields are defined within a structure, the various bits can be accessed in a way we access individual members of a structure.

The general for or syntax for bit field definition as follows,

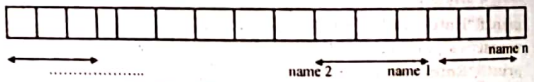
```
struct tag_Name
{
        Unsigned int name1: bit_length;
        Unsigned int name2: bit_length;
        ...
        ...
        Unsigned int namen: bit_length;
};
```

Where,

- unsigned int is always used to define bit fields.
- bit_length is the number of bits used for the specified name.
- field name is followed by colon.

Remember that a signed bit field should have at least 2bits (1bit for sign). The bit_length is decided by the range of value stored. The largest value that can be stored is 2^{n-1}, where n is bit_length.

The internal representation of bit fields is machine dependent. Some machines store bits from left to right and others from right to left. The below Figure illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



Bit Field Storage Representation

Q. 7. (iii) Automatic Storage Classes

Ans. Automatic Storage Class:

- A variable with an **auto** specification has the following storage characteristic:

Scope: block Extent: automatic Linkage: internal

- The default and the most commonly used storage class is auto.
- Memory for automatic variables is allocated when a block or function is entered. They are defined and are "local" to the block.
- When the block is exited, the system releases the memory that was allocated to the auto variables, and their values are lost.
- These are also referred with local variables

Declaration:

auto type variable_name;

There is no point in using auto, as it's implicitly there anyway. By default all variables are created as automatic variables.

Initialization

An auto variable can be initialized where it is defined or left uninitialized. When auto variables are not initialized its value is garbage value.

// C Program to illustrate the auto storage class

```
#include <stdio.h>
int main( )
{
 auto int j = 1;
 {
  auto int j= 2;
  {
   auto int j = 3;
   printf ( " %d ", j);
  }
  printf ( "\t %d " j);
 }
 printf( "%d\n", j);
}
```

OUTPUT:

3 2 1

Q. 7. (iv) File handing

Ans. File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

There are many functions in the C library to open, read, write, search and close the file. Some of the file functions are given below:

Scanned by CamScanner

S. No.	Function	Description
1.	fopen()	opens new or existing file
2.	fprintf()	write data into the file
3.	fscanf()	reads data from the file
4.	fputc()	writes a character into the file
5.	fgetc()	reads a character from file
6.	fclose()	closes the file
7.	fseek()	sets the file pointer to given position
8.	fputw()	writes an integer to file
9.	fgetw()	reads an integer from file
10.	ftell()	returns current position
11.	rewind()	sets the file pointer to the beginning of the file

UNIT-IV

Q. 8. (a) Write a program to compare two strings whether they are identical or not. **(4.5)**

Ans. Program to compare two strings:

```
#include<stdio.h>
void main()
{
    char str1[20],str2[20];
    int i, flag = 0;
    printf("Enter the first string: ");
    gets(str1);
    printf("Enter the second string: ");
    gets(str2);
    for(i=0; str1[i]!='\0'; i++)
    {
        if(str1[i] == str2[i])
            flag = 1;
        else
            flag = 0;
    }
    if(flag == 1)
    printf("Both strings are identical.");
    else
    printf("Both string not identical.");
getch();
}
```

Q. 8. (b) Explain the following inbuilt functions. **(8)**

(i) strlen() **(ii) strcat()** **(iii) strlwr()** **(iv) strupr()**

Ans. strlen () function:-

This function counts and returns the number of characters in a string. It takes the form

n=strlen(string);

Where n is an integer variable, which receives the value of the length of the string. The counting ends at the first null character.

Example:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char name[100]="Amit Arora";
    printf ("%d", strlen (name));
    getch();
}
```

Output: 10

(ii) strcat () function:

The strcat function joins two strings together. It takes of the following form:

strcat(string1,string2);

String1 and string2 are character arrays. When the function strcat is executed, string2 is appended to string1.It does so by removing the null character at the end of string1 and placing string2 from there.

strcat function may also append a string constant to a string variable. The following is valid.

strcat(part1,"Good");

C permits nesting of strcat functions. Example:

strcat(strcat(string1,string2),string3);

Example

```
#include <stdio.h>
#include <string.h>
void main ()
{
   char country[255] = "The capital of India is...";
   char capital[255] = "New Delhi";
   strcat (country, capital);
   printf ("%s", country);
   getchar();
}
```

(iii) strlwr(): strlwr function converts a string to lower case

Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
  char string[50];
  printf("Input a string to convert to lower case\n");
```

Scanned by CamScanner

```
  gets(string);
  printf("The string in lower case: %s\n", strlwr(string));
  return 0;
}
```

(iv) strupr(): strupr function converts a string to upper case

Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
  char string[50];
  printf("Input a string to convert to upper case\n");
  gets(string);
  printf("The string in upper case: %s\n", strupr(string));
  return 0;
}
```

Q. 9. (a) Write a program to find the largest and smallest word in a string. (4.5)

Ans. Program to find smallest and largest word in a string:

```
#include<stdio.h>
#include<string.h>
void main()
{
  int i=0, j=0, k=0, a, minS=0,maxS=0,max=0,min=0, len;
  char str[100], substr[100][100],c;
  printf("Enter a line\n");
  gets(str);
  while(str[k]!='\0')     //for splitting line into words
  {
    j=0;
    while(str[k]!=' '&&str[k]!='\0')
    {
      substr[i][j]=str[k];
      k++;
      j++;
    }
    substr[i][j]='\0';
    i++;
    if(str[k]!='\0')
    {
      k++;
    }
```



```
  }
  len=i;
  max=strlen(substr[0]);
  min=strlen(substr[0]);
/* After splitting, getting length of string and finding its subscript value having max
length and subscript value having min length */
  for(i=0; i<len; i++)
  {
    a=strlen(substr[i]);
    if(a>max)
    {
      max=a;
      maxS=i;
    }
    if(a<min)
    {
      min=a;
      minS=i;
    }
  }
  printf("Largest Word is %s \nSmallest word is %s\n", substr[maxS],substr[minS]);
}
```

Q. 9. (b) Explain the following header files.

(i) stdlib.h (ii) ctype.h (iii) math.h (iv) process.h (8)

Ans. (i) Header file <stdlib.h> contains general utility library functions like: string searching, conversion, sorting, dynamic memory allocation etc. Some of the functions to perform mathematical operations are defined under <stdlib.h> with most of the functions defined under <math.h> header file. Some of the functions which are defined in this header file are as below:

Function	Working
abort()	Terminates program abnormally
abs()	Calculates the absolute value
atoi()	Converts string to int and returns it
atol()	Converts string to long and returns it
exit()	Terminates program normally
labs()	Calculates the absolute value

(ii) Header file <ctype.h> includes a number of standard library functions to handle characters. There are various library functions defined under <ctype.h> for character handling. Some of the functions which are defined in this header file are as below:

Function	Working	Function	Working
isalnum()	checks alphanumeric character		
isalpha()	checks whether a character is an alphabet or not		
isdigit()	checks numeric character		

Scanned by CamScanner

islower()	checks lowercase alphabet
isprint()	checks printable character
isspace()	check white-space character
isupper()	checks uppercase alphabet
tolower()	converts alphabet to lowercase
toupper ()	converts to lowercase alphabet

(iii) math.h: There are various standard library functions and a macro defined under <math.h> to perform mathematical operations in C programming. Some of the functions which are defined in this header file are as below:

Function	Working
ceil()	computes the nearest integer greater than argument
cos()	computes the cosine of an argument.
sin()	computes the sine of an argument.
exp()	computes the exponential raised to the argument
fabs()	computes absolute value
floor()	calculates the nearest integer less than argument
log()	computes natural logarithm of an argument.
pow()	Computes power of a number
sqrt()	computes square root of a number
tan ()	computes tangent

(iv) process.h is a C header file which is contains function declarations and macros used in working with threads and processes. Some of the functions which are defined in this header file are as below:

1. exit() :- exit terminates the program. exit terminates the calling process. Before termination, exit does the following:-

- closes all files.
- writes buffered output (waiting to be output).
- calls any registered "exit functions" (posted with at exit).

Typically a value of 0 indicates a normal exit, and a non-zero value indicates some error.

2. abort():- Abnormally terminates a process. abort() writes a termination message on stderr ("Abnormal program termination"), then aborts the program by a call to exit().

3. getpid():- Gets the process ID of the program. getpid is a macro that gets the process ID of the program. A process ID uniquely identifies a program. This concept is borrowed from multitasking operating systems like UNIX, where each process is associated with a unique process number.

Example:

```
#include <stdio.h>
#include <process.h>
int main()
{
  printf("This program's process identification number (PID) number is %X\n",
getpid());
  printf("Note: under DOS it is the PSP segment\n");
  return 0;
}
```

Scanned by CamScanner