CMR INSTITUTE OF TECHNOLOGY
INTERNAL ASSESSMENT 3 IMPORTANT QUESTIONS
C PROGRAMMING FOR PROBLEM SOLVING

CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test – III – Apr 2021

Sub:	C PROGRAMMING FOR PROBLEM SOLVING							Code:	18CPS13
Date:	7th Apr 2021	Duration:	90 mins	Max Marks:	50	Sem:	I	Section:	I, J, K, L, M, N, O

Answer **Any FIVE FULL** Questions

	***WRITE EXAMPLES TO ALL QUESTIONS WHEREVER ASKED. *WRITE OUTPUTS FOR ALL THE PROGRAMS WRITTEN.**	Marks	OBE	
			CO	RBT
1	Explain the different types of functions based on parameters and return types. Explain with syntax and examples (full programs).	[10]	CO4	L1
2	Define Function. Write the syntax of a function definition and label the parts. Write a C function isprime(num) that accepts an integer argument and return 1 if the argument is a prime or a 0 otherwise. Write a program that invokes this function to generate prime number between a given range.	[10]	CO4	L1, L3
3	Implement structures to read, write and compute average marks and display students scoring above average and students below average for a class of N students.	[10]	CO5	L3
4	Explain macro substitution and compiler control preprocessor directives in C with example of code snippets.	[10]	CO5	L2
5	Write the syntax and examples of declaration and initialization of pointer variable. Implement a C program to find sum, mean and standard of all elements in an array using pointers.	[10]	CO5	L1, L3
6	Define pointers. Explain call by value and call by reference with examples (full program)	[10]	CO5	L2
7	Explain recursive functions. Give 2 differences between iteration and recursion. Write a C program to generate the **n** Fibonacci numbers for a given value of n.	[10]	CO4	L2, L3
8	Write a C program to add two lengths (ft, inches) that accepts a structure as parameters to a function from a function call. (Make sure to use call by value and call by reference) Data structure needed: Structure length with two members – ft, inches Functions needed: entry() -> no parameters taken but returns structure variable sum() -> take 3 length structure as parameters (2 as call by value and 1 as call by reference) but doesn't return any value display() -> takes structure variable as parameter but no return values	[10]	CO4, CO5	L3

Q1. Explain the different types of functions based on parameters and return types. Explain with syntax and examples (full programs).

ANS. The category of functions based on parameters and return types are:

Category of functions

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and one return value.
4. Functions with no arguments but one return value.
5. Functions that return multiple values.

1. Functions with no arguments and no return values

The general syntax is:

```
main() ← calling function          void function1()
{                                   {            ← called
    ....                               ....        function
function call → function1();           ....
    ....                            }.
}
```

```
Eg:- main()              void add()
     {                   {
        add();              int a, b;
     }                      printf("Enter a and b");
                            scanf("%d %d", &a, &b);

                            printf("sum is %d", a+b);
                         }
```

② Functions with arguments and no return values.

The general syntax is:

```
main()                                   void function1(list of parameters)
{         actual parameters              {                formal parameters
   ----                                       ----
   function1(list of parameters);
   ....                                  }.
}
```

Eg:-

```
main()                                   void add(int a, int b)
{                                        {
   int a, b;                                printf("sum is %d", a+b);
   printf("Enter a and b:");             }
   scanf("%d%d", &a, &b);
   add(a, b);
}
```

③ Functions with arguments and one return value

The general syntax is :

```
main()
{
    ....          actual parameters
    function1 (list of parameters);
    ....
}
```

```
                              formal parameters
return_type function1 ( list of parameters)
{

    return (expression) value);
}
```

Eg:-

```
main()
{  int a, b;

   printf ("Enter a and b");
   scanf ("%d%d", &a, &b);
   int c= add (a, b);
   printf("%d is sum", c);
}
```

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

④ Functions with no arguments but one return value.

The general syntax is :

```
main()                          |  return-type function1()
{  ....                         |  {  ....
   function1();                 |     ....
   ....                         |     return (expression/value);
}                               |  }
```

Eg:-

```
main()                                  |  int add()
{                                       |  {  int a, b;
   int sum= add();                      |     printf("Enter a and b");
   printf("The sum is %d", sum);        |     scanf("%d%d", &a, &b);
}                                       |     return (a+b);
                                        |  }
```

(5) Functions that return multiple values.

Eg:-

```
main ()
{   int a= 50, b=20, sum, diff;
    mathop(a, b, &sum, &diff);
    printf(" sum = %d \n Diff = %d ", sum, diff);
}

void mathop (int x, int y, int *sum, int * diff)
{
    *sum = x+y;
    * diff = x-y;
}
```

Q2. Define Function. Write the syntax of a function definition and label the parts. Write a C function isprime(num) that accepts an integer argument and return 1 if the argument is a prime or a 0 otherwise. Write a program that invokes this function to generate prime number between a given range.

ANS:

A function is a self contained block of code that performs a particular task.

Need for user-defined functions

Altough it is possible to code any problem / program in main function, it leads to numerous problems.

* May become too large and complex as a result the task of debugging, testing and maintaining becomes difficult.
* If a program is divided into various parts, then each part may be independently coded and later combine into a single unit.

These independently coded programs are called subprograms that are much easier to understand, debug and test.

Such subprograms are referred to as functions.

```c
#include<stdio.h>

int isprime(int n)
{
    int i;

    for(i=2;i<=n/2;i++)
    {
        if(n%i == 0)
        {
            return 0;
        }
    }

    return 1;
}

int main()
{
    int beg, end, j;

    printf("\nEnter starting and ending number of range: ");
    scanf("%d%d", &beg, &end);

    for(j=beg;j<=end;j++)
    {
        if(isprime(j) == 1)
            printf("%d ", j);
    }

}
```

OUTPUT:

```
Enter starting and ending number of range: 2 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Q3. Implement structures to read, write and compute average marks and display students scoring above average and students below average for a class of N students.

```c
#include<stdio.h>
#define n 5

int main()
{
  struct student
  {
    int rno;
    char name[20];
    float marks[5], avg;
  };

  int i, j;
  float sum;

  struct student s[n];
```

```
printf("\nEnter student details: ");
for(i=0;i<n;i++)
{
  printf("\nEnter student%d details: ", i+1);

  printf("\nEnter roll no: ");
  scanf("%d", &s[i].rno);

  printf("\nEnter name: ");
  scanf("%s", s[i].name);

  printf("\nEnter 5 marks: ");
  sum = 0;
  for(j=0;j<5;j++)
  {
    scanf("%f", &s[i].marks[j]);
    sum = sum + s[i].marks[j];
  }
  s[i].avg = sum/5;
}
```

```c
    printf("\nStudents above average: ");
    printf("\nRoll No\tName\t\tAvg");
    for(i=0;i<n;i++)
    {
        if(s[i].avg>=40)
        {
            printf("\n%d\t\t%s\t\t\t%f", s[i].rno, s[i].name, s[i].avg);
        }
    }

    printf("\nStudents below average: ");
    printf("\nRoll No\tName\t\tAvg");
    for(i=0;i<n;i++)
    {
        if(s[i].avg<40)
        {
            printf("\n%d\t\t%s\t\t\t%f", s[i].rno, s[i].name, s[i].avg);
        }
    }
}
```

Q4. Explain macro substitution and compiler control preprocessor directives in C with example of code snippets.

ANS:

Macro substitution

→ Macro substitution is a process where an identifier in a program is replaced by a predefined string.

→ #define statement is used for this.

→ The general syntax is (a min one blank space is must)

#define identifier string

If this statement is included in the program at the beginning, then the preprocessor replaces every occurence of the identifier in the source with string.

→ There are 3 different forms of macro substitution.

1. Simple macro substitution.
2. Argumented " "
3. Nested " "

★ Simple Macro Substitution

→ Used to define constants.

→ Examples:

→ Example:

```
#include <stdio.h>
#define M 5.
int main ()
{
    int total;
    total = M * 100;
    printf("The value of M is %d", M);
    return (0);
}
```

→ A macro definition can also include expression.

Examples:

```
#define SIZE sizeof(int) * 4.
#define AREA 5 * 12.37
```

* Macros with Arguments

→ The preprocessor allows us to define more complex and more useful form of replacements.

→ The general syntax is:

```
#define identifier(f1, f2, ... fn) string.
```

no space should be there.

where f1, f2, ... fn are formal arguments.

→ Example:

```
#define CUBE(x) (x * x * x).
```

→ some of the commonly used definitions are:

```
#define MAX(a,b)     (((a)>(b)) ? (a) : (b))
#define MIN(a,b)     (((a)<(b)) ? (a) : (b))
#define ABS(x)       (((x) > 0) ? (x) : (-(x)))
```

★ Nesting of Macros

→ using one macro in the definition of another macro.

→ For examples:

```
#define M          5
#define N          M+1
#define SQUARE(x)  ((x) * (x))
#define CUBE(x)    (SQUARE(x) * (x))
#define SIXTH(x)   (CUBE(x) * CUBE(x))
```

Compiler Control Directives

→ While developing large programs, you may face one or more of the following situations:

★ You have included a file containing some macro definitions. It is not known whether a particular macro has been defined in that header file. However, you want to be certain that the macro is defined (or not defined).

★ Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.

★ You are developing a program for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.

① #ifdef, #else, #endif.

→ This directive checks if/whether particular macro is defined or not. If it is defined, "if" clause statements are included in source file.

→ Otherwise, "else" clause statements are included in source file for compilation and execution.

Syntax :

```
#ifdef MACRONAME
    Statement_block;
#endif
```

For example:

```
#include<stdio.h>
#define MAX 10

int main()
{
    #ifdef MAX
        printf("Max is defined!!");
    #else
        printf("Max is not defined!!");
    #endif
    return (0);
}
```

Output: Max is defined!!

② #ifndef, #endif, #else.

→ This exactly acts as reverse as #idef directive. If particular macro is not defined, "if" clause statements are included in source file.

→ Otherwise, "else" clause statements are included in source file for compilation and execution.

Syntax:

```
#ifndef MACRONAME
        statement_block;
#endif
```

For Example:

```
#include <stdio.h>
#define MAX 100

int main()
{
    #ifndef MIN
    {  -> optional
        printf("Min is not defined!! Define it now");
        #define MIN 16
    }
    #else
        printf("Min is already defined!!");
    #endif
    return(0);
}
```

Output: Min is not defined!! Define it now.

③ #if, #else, #endif

→ "if" clause statement is included in source file if given condition is true.

→ otherwise, "else" clause statement is included in source file for compilation and execution.

Syntax:

```
#if Expression.
    Statement 1
    Statement 2
    :
    Statement n
#endif
```

Note: Expressions should be only a constant

For Example:

```
#include<stdio.h>
#define X 50
int main()
{
    #if (X==50)
        printf("This line will be included!!");
    #else
        printf("This line will not be included");
    #endif
    return(0);
}
```

Output: This line will be included!!

④ #undef

→ This directive undefines existing user-defined macro in the program.

| Syntax : #undef MACRONAME |

For Example :

```
#include <stdio.h>
#define PI 3.14

int main()
{
    printf(" First defined value for pi = %f ", PI);
    #undef PI
    #define PI 3.1412
    printf(" After redefining value of pi = %f ", PI);
    return (0);
}
```

Output : First defined value for pi = 3.14.
After redefining value of pi = 3.1412

(5) #elif

→ #elif directive works like else if ladder.

→ The general syntax is:

```
#if Expression1
        statement-block 1;
#elif Expression2
        statement-block 2;
#elif Expression3
        statement-block 3;
#else
        statement-block 4;
#endif
```

For Example:

```
#include <stdio.h>
#define MARKS 90
int main()
{

    #if ((MARKS >= 80) && (MAKS <=100))
        printf("\n Distinction");
    #elif ((MARKS >= 60 && (MARKS <=79))
        printf("\n First Class");
    #elif ((MAKS >= 40) && (MARKS <= 59))
        printf("\n Second Class");
    #else
        printf("\n Fail");
    #endif
    return (0);
}
```

⑥ #error

→ This is used to produce diagnostic messages during debugging.

→ The general syntax is:

```
#error error_message.
```

For Example:

```
#include<stdio.h>
#define MAX 30
int main()
{
    #if (MAX)
        #error MAX is defined
    #endif
}
```

output: #error MAX is defined

⑦ Stringizing operator #

→ # called stringizing operator provided by ANSI C. to be used in the definition of macro functions

→ This operator allows a formal argument within a macro definition to be converted to a string.

For Example :

```
#include <stdio.h>
#define sum(xy)  printf(#xy " = %f\n", xy)
int main()
{
    sum(a+b);
}
```

The preprocessor will convert the line :

```
sum(a+b);
```

into

```
printf("a+b" " = %f\n", a+b);
```

which is equivalent to

```
printf("a+b = %f\n", a+b);
```

(8) Token Pasting Operator ##

→ ## called token pasting operator enables us to combine two tokens within a macro definition to form a single token.

For Example :

```
#include<stdio.h>
# define  t(p1, p2)    p1##p2

    int main()
    {
        int  var10 = 123;
        printf("%d", t(var, 10));
    }
```

The preprocessor will conver the line :

printf("%d", t(var,10));

into

printf("%d", var10);

∴ Output = 123

Q5. Write the syntax and examples of declaration and initialization of pointer variable. Implement a C program to find sum, mean and standard of all elements in an array using pointers.

ANS:

Declare a pointer variables

→ The general syntax ;

data-type * pointer_name ;

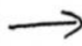
↑ tells that pointer-name is a pointer variable.

pointer-name points to a variable of type data-type.

→ eg:- int *p ; //Integer pointer

p is a pointer variable that points to a integer data type.

float *x ; // float pointer.

p 

→ ?

contains garbage

points to unknown location

Initialization of pointer variables

→ The process of assigning the address of a variable to pointer variable is called initialization.

→ Use assignment operator to initialise the variable.

Eg:- int *p; // declaration
int a;
p = &a; // Initialization

(Or)

int *p = &a; // Note a must be declared before initializing.

Note, this is initializing p and not *p.

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

int main()
{
  float A[10], *p, sd, var, mean, sum=0;
  int i, n;

  printf("\nEnter the number of elements: ");
  scanf("%d", &n);

  p=A;
  printf("\nEnter the array elements: ");
  for(i=0;i<n;i++)
  {
    scanf("%f", p);
    sum = sum + (*p);
    p++;
  }
```

```
    mean = sum/n;
    var = 0;

    p=A;

    for(i=0;i<n;i++)
    {
      var = var + pow((*p - mean), 2);
      p++;
    }

    var = var/n;
    sd = sqrt(var);

    printf("\nSum = %f\nMean = %f\nStandard Deviation = %f\n", sum, mean, sd);

}
```

OUTPUT:

```
Enter the number of elements: 5

Enter the array elements: 1
2
3
4
5

Sum = 15.000000
Mean = 3.000000
Standard Deviation = 1.414214
```

Q6: Define pointers. Explain call by value and call by reference with examples (full program)

ANS:

Pointers

→ Pointers are variables that stores addresses as their values.

→ Pointers are / can be used to manipulate data stored in the memory.

✱ Pass/call by value

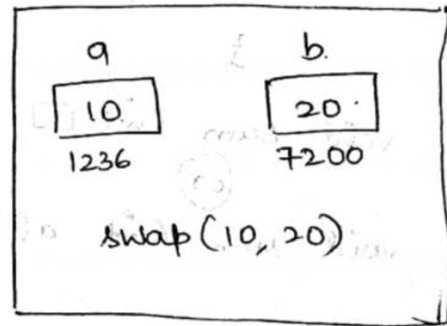
→ Values of actual parameters are copied to variables in the parameter list of the called function (formal parameters.

→ The called function works on the copy and not on the original values of the actual parameters.

→ ∴ The original data in the calling function doesn't change.

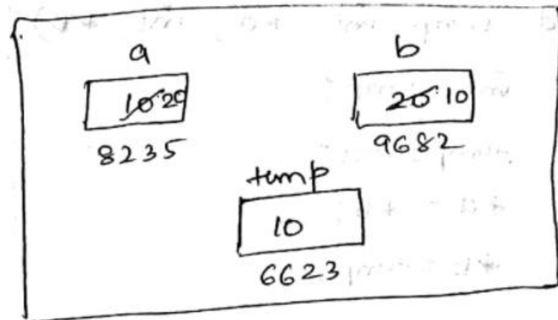
Eg:-

```
main()
{
    int a=10, b=20;
    printf("Before swapping:\n
    a=%d \n b=%d", a, b);
    swap(a, b);
    printf("After swapping:\n
    a=%d \n b=%d", a, b);
}
```



main()

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



swap()

∴ output : Before swapping a = 10 b = 20
After swapping a = 10 b = 20

* Pass/call by reference/pointers

→ The memory addresses of the variables rather than copies of values are sent to the called function.

→ The called function directly works on the data in the calling function.

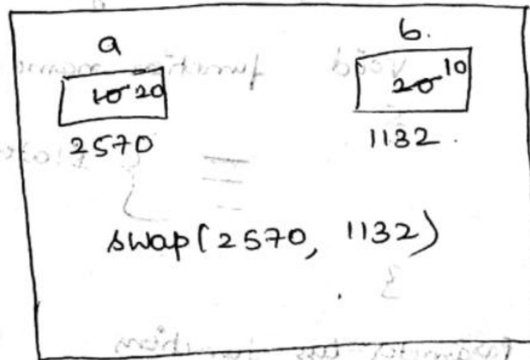
→ The changes are reflected in the calling function.

Eg:-

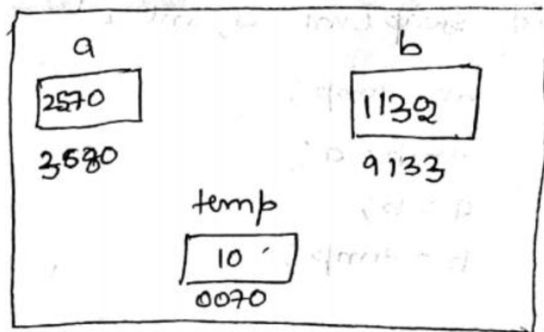
```
main()
{
    int a=10, b=20;
    printf("Before swapping:\n
    a=%d \n b=%d", a, b);

    swap(&a, &b);

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




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



∴ output: Before swapping a=10 b=20

After swapping a=20 b=10

Q7. Explain recursive functions. Give 2 differences between iteration and recursion. Write a C program to generate the n Fibonacci numbers for a given value of n.

ANS:

Recursion

→ A function calling itself is called as recursive function.

→ While using recursion, we have to be careful while defining the exit condition or terminating condition, otherwise will result in a infinite recursive call.

Recursion	Iteration
1. Recursion is the process in which a function has the ability to call itself until a certain condition is met. 2. Recursion needs to have function present in the program	1. Iteration is the process in which a set of statements has the ability to repeat itself until a certain condition is true. 2. Iteration does not need function compulsorily in the program

```c
#include<stdio.h>

int fib(int n)
{
    if((n==0) || (n==1))
        return n;

    else
        return fib(n-1) + fib(n-2);
}

int main()
{
    int n, i;

    printf("\nEnter the number of values: ");
    scanf("%d", &n);

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

OUTPUT:

```
Enter the number of values: 1
0 1 > ./a.out

Enter the number of values: 2
0 1 1 > ./a.out

Enter the number of values: 5
0 1 1 2 3 5 > ./a.out

Enter the number of values: 10
0 1 1 2 3 5 8 13 21 34 55 >
```

Q8. Write a C program to add two lengths (ft, inches) that accepts a structure as parameters to a function from a function call. (Make sure to use call by value and call by reference)

```c
#include<stdio.h>

struct length
{
  int ft, inches;
};

struct length entry()
{
  struct length s;

  printf("\nEnter values: ");
  scanf("%d%d", &s.ft, &s.inches);

  return s;
}

void sum(struct length a, struct length b, struct length *c)
{
  c->ft = a.ft + b.ft + (a.inches + b.inches) / 12;
  c->inches = (a.inches + b.inches) % 12;
}

void disp(struct length c)
{
  printf("\nSum = %d ft + %d inches", c.ft, c.inches);
}


int main()
{
  struct length a, b, c;

  //Call by value
  a = entry();
  b = entry();

  //call by reference
  sum(a, b, &c);

  disp(c);
}
```