



SNS COLLEGE OF TECHNOLOGY
(An Autonomous Institution)



Coimbatore – 641035

B.E / B. Tech – Internal Assessment Exam – III

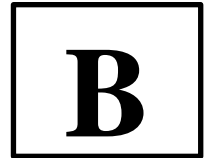
Academic Year 2022 – 2023 (ODD)

FIRST SEMESTER (REGULATION R2019)

19CST101 – PROGRAMMING FOR PROBLEM SOLVING

ANSWER KEY

PART A



1. Define Function. Differentiate between calling function and called function.

Function:

A function is a block of code which only **runs when it is called**. Functions are used to perform certain actions, and they are important for **reusing code**: Define the code once, and use it many times.

Calling function: A function that calls or invokes another function is called Calling function.

Called function: A function that is called or invoked by another function is called Called-function.

2. Build a working code for the implementation of recursive function.

```
#include<stdio.h>
int sum(int k);
int main() {
    int result = sum(10);
    printf("%d", result);
    return 0;
}
int sum(int k) {
    if (k > 0) {
        return k + sum(k - 1);
    } else {
        return 0; } }
```

3. Illustrate pointer and justify why pointer is a more efficient data type.

Pointers in C are used to store the address of variables or a memory location. This variable can be of any data type i.e, int, char, function, array, or any other pointer.

A pointer is a variable whose value is the address of another variable of the same type. The variable's value that the pointer points to is accessed by dereferencing using the * operator

```
data_type * pointer_variable_name;
```

```
int x = 45;
```

```
int *ptr;      //pointer variable declaration
ptr = &x;
```

Pointers are efficient because Pointers make possible to return more than one value from the function. Pointers increase the processing speed. In other words, Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location.

4. Define Structure and how a structure element can be accessed.

Structure in C is a User-Defined data type.

It is used to bind two or more similar or different data types or data structures together into a single type.

The structure is created using the struct keyword, and a structure variable is created using the struct keyword and the structure tag name.

Structure members are accessed using dot [.] operator.

5. Differentiate structure and union.

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

PART- B

6. (a). Illustrate and explain about the various Category Of Functions with example codes.

A **function** is a block of statements that can perform a particular task. As we all know, there is always at least one function in C, and that is main().

Example

In the example below, the function's name is `sum` and the data type is `int`. This task of this function is to produce the sum of two numbers:

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

Below, the function is declared in `main()`:

```
void main()
{
    int sum(int,int); //function declaration
    int x=5,y=6;
    total = sum(x,y);
}
```

Formal parameters and actual parameters

When we call a function in `main()` or anywhere else in the program, and the function we created needs parameters, we would pass parameters to it while calling the function. In the example above, we passed variables `x` and `y` to obtain the sum of `x` and `y`.

According to the example above, the *formal parameters* are `a` and `b`, and the *actual parameters* are `x` and `y`.

Essentially, the variables being passed in the function call are *actual parameters*, and the variables being initialized and used in the function are *formal parameters*.

Advantages of a function

1. A function can be used any number of times after it is defined once.
2. Functions make programs manageable and easy to understand.

Function categories

There are 4 types of functions:

1. Functions with arguments and return values

This function has arguments and returns a value:

Program code:

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

```

int sub(int,int); //function with return value and arguments
int x=10,y=7;
int res = sub(x,y);
printf("x-y = %d",res);
}
int sub(int a,int b) //function with return value and arguments
{
return(a-b); // return value
}

```

Output:

x -y = 3

2. Functions with arguments and without return values

This function has arguments, but it does not return a value:

```

#include <stdio.h>
int main()
{
void sum(float,float); //function with arguments and no return value
float x=10.56,y=7.22;
sum(x,y);
}
void sum(float a,float b) //function with arguments and no return value
{
float z = a+b;
printf("x + y = %f",z);
}

```

Output:

x + y = 17.780001

3. Functions without arguments and with return values

This function has no arguments, but it has a return value:

```
#include<stdio.h>
int main()
{
    int sum();
    int c = sum();
    printf("Sum = %d",c);
}
int sum() //function with no arguments and return data type
{
    int x=10,y=20,z=5;
    printf("x = %d ; y = %d ; z = %d \n",x,y,z);
    int sum = x+y+z;
    return(sum);
}
```

Output

x = 10 ; y = 20 ; z = 5 Sum = 35

4. Functions without arguments and without return values

This function has no arguments and no return value:

```
#include<stdio.h>
int main()
{
    void sum();
    sum();
}
void sum() //function with no arguments and return data type
{
    int x=15,y=35,z=5;
    printf("x = %d ; y = %d ; z = %d \n",x,y,z);
}
```

```
int sum = x+y+z;
printf("Sum = %d",sum);
}
```

Output:

x = 15 ; y = 35 ; z = 5 Sum = 55

(b) Construct a code for calculating the Factorial of a number using recursive function.

Recursion:

A recursive function is a function in code that refers to itself for execution. Recursive functions can be simple or elaborate. They allow for more efficient code writing, for instance, in the listing or compiling of sets of numbers, strings or other variables through a single reiterated process.

Factorial:

In Mathematics, factorial is a simple thing. Factorials are just products. An exclamation mark indicates the factorial. Factorial is a multiplication operation of natural numbers with all the natural numbers that are less than it

Program Code:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,i,f;

int recur(int);

printf("\n FACTORIAL OF A NUMBER USING RECURSION");

printf("Enter a number...");

scanf("%d",&n);

f=recur(n);
```

```
printf("Factorial is %d",f);
```

```
getch();
```

```
}
```

```
int recur(int x)
```

```
{
```

```
int fact;
```

```
if(x==1)
```

```
{
```

```
return(1);
```

```
}
```

```
else
```

```
{
```

```
fact=x*recur(x-1);
```

```
return(fact);
```

```
}
```

```
}
```

Output:

```
FACTORIAL OF A NUMBER USING RECURSION  
Enter a number...5
```

```
Factorial is 120: 
```

7. (a). Appraise and explain any four arithmetic operations on pointer with example for each.

Pointers variables are also known as address data types because they are used to store the address of another variable.

The address is the memory location that is assigned to the variable. It doesn't store any value.

We can perform arithmetic operations on the pointers like addition, subtraction, etc.

However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

Hence, there are only a few operations that are allowed to perform on Pointers in C language.

The operations are slightly different from the ones that we generally use for mathematical calculations.

Increment/Decrement of a Pointer

Rule for Increment: $\text{new_address} = \text{current_address} + i * \text{size_of}(\text{data type})$

Rule for Decrement: $\text{new_address} = \text{current_address} - i * \text{size_of}(\text{data type})$

Example:

```
#include <stdio.h>
int main()
{
int a = 22;
int *p = &a;
printf("p = %u\n", p); // p = 6422288
p++;
printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes
p--;
printf("p-- = %u\n", p); //p-- = 6422288 -4 // restored to original value
float b = 22.22;
float *q = &b;
printf("q = %u\n", q); //q = 6422284
q++;
printf("q++ = %u\n", q); //q++ = 6422288 +4 // 4 bytes
q--;
printf("q-- = %u\n", q); //q-- = 6422284 -4 // restored to original value
char c = 'a';
```



```

char *r = &c;
printf("r = %u\n", r); //r = 6422283
r++;
printf("r++ = %u\n", r); //r++ = 6422284 +1 // 1 byte
r--;
printf("r-- = %u\n", r); //r-- = 6422283 -1 // restored to original value
return 0;
}

```

Addition of integer to a pointer

Rule for Addition: $\text{new_address} = \text{current_address} + (\text{number} * \text{size_of}(\text{data type}))$

Example:

```

// C program to illustrate pointer Addition
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Addition: ");
    printf("%p \n", ptr2);

    // Addition of 3 to ptr2
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

    return 0;
}

```

Subtraction of integer to a pointer

Rule for Subtraction: $\text{new_address} = \text{current_address} - (\text{number} * \text{size_of}(\text{data type}))$

Example:

```

// C program to illustrate pointer Subtraction
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);

    return 0;
}

```

Subtracting two pointers of the same type

Rule for Subtraction of two pointer of same type:

Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

Example:

```

// C program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
    int x = 6; // Integer variable declaration
    int N = 4;

    // Pointer declaration
    int *ptr1, *ptr2;

    ptr1 = &N; // stores address of N
    ptr2 = &x; // stores address of x

    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
    // %p gives an hexa-decimal value,
    // We convert it into an unsigned int value by using %u

    // Subtraction of ptr2 and ptr1
    x = ptr1 - ptr2;

    // Print x to get the Increment
    // between ptr1 and ptr2
    printf("Subtraction of ptr1 "
           "%u ptr2 is %d\n",
           x);

    return 0;
}

```

Comparison of pointers of the same type

Rule for Comparing pointers: pointer 1 <comparison operator> pointer 2

Example:

```
#include <stdio.h>

int main() {

    // code
    int num1=5,num2=6,num3=5; //integer input
    int *p1=&num1; // addressing the integer input to pointer
    int *p2=&num2;
    int *p3=&num3;
    //comparing the pointer variables.
    if(*p1<*p2)
    {
        printf("\n%d less than %d",*p1,*p2);
    }
    if(*p2>*p1)
    {
        printf("\n%d greater than %d",*p2,*p1);
    }
    if(*p3==*p1)
    {
        printf("\nBoth the values are equal");
    }
    if(*p3!=*p2)
    {
        printf("\nBoth the values are not equal");
    }

    return 0;
}
```

(b) Generate a C code to swap 2 numbers entered by the user using call by reference and call by value.

Call By value:

In call by value method, the value of the actual parameters is copied into the formal parameters.

In other words, we can say that the value of the variable is used in the function call in the call by value method.

In call by value method, we cannot modify the value of the actual parameter by the formal parameter.

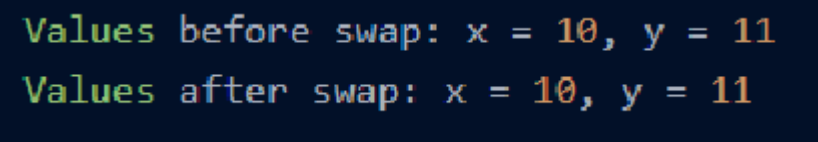
In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Example:

```
#include <stdio.h>
void swap(int x, int y)
{
int temp = x;
x = y;
y = temp;
}
int main()
{
int x = 10;
int y = 11;
printf("Values before swap: x = %d, y = %d\n", x,y);
swap(x,y);
printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output



```
Values before swap: x = 10, y = 11
Values after swap: x = 10, y = 11
```

Call by Reference:

Calling a function by reference will give function parameter the address of original parameter due to which they will point to same memory location and any changes made in the function parameter will also reflect in original parameters.

In call by reference, the address of the variable is passed into the function call as the actual parameter.

The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Example

```

#include <stdio.h>
void swap(int *x, int *y)
{
int temp = *x;
*x = *y;
*y = temp;
}
int main()
{
int x = 10; int y = 11;
printf("Values before swap: x = %d, y = %d\n", x,y);
swap(&x,&y);
printf("Values after swap: x = %d, y = %d", x,y);
}

```

Output

```

Values before swap: x = 10, y = 11
Values after swap: x = 11, y = 10

```

8. (a) A college conducts a semester exam and there are 6 subjects in that semester. Get all the required details of student and display the progress card with percentage of marks obtained in semester using Structures

```

#include <stdio.h>

struct Student
{
char name[50];

int rollno;

float marks[6];

};

int main()
{
struct Student s;

// Get the details of the student

```

```
printf("Enter the name of the student: ");
fgets(s.name, 50, stdin);
printf("Enter the roll number of the student: ");
scanf("%d", &s.rollno);
// Get the marks of the student in each subject
printf("Enter the marks of the student in each subject:\n");
for (int i = 0; i < 6; i++)
{
printf("Subject %d: ", i+1);
scanf("%f", &s.marks[i]);
}
// Calculate the total marks and percentage
float total = 0.0;
for (int i = 0; i < 6; i++)
{
total += s.marks[i];
}
float percentage = total / 6;
// Display the progress card
printf("\nProgress Card\n");
printf("Name: %s", s.name);
printf("Roll Number: %d\n", s.rollno);
printf("Marks:\n");
for (int i = 0; i < 6; i++) {
printf("Subject %d: %.2f\n", i+1, s.marks[i]);
}
printf("Total marks: %.2f\n", total);
```

```
printf("Percentage: %.2f%%\n", percentage);

return 0;

}
```

Output:

```
Enter the name of the student: HARSHA
Enter the roll number of the student: 33
Enter the marks of the student in each subject:
Subject 1: 100
Subject 2: 95
Subject 3: 100
Subject 4: 92
Subject 5: 97
Subject 6: 93

Progress Card
Name: HARSHA
Roll Number: 33
Marks:
Subject 1: 100.00
Subject 2: 95.00
Subject 3: 100.00
Subject 4: 92.00
Subject 5: 97.00
Subject 6: 93.00
Total marks: 577.00
Percentage: 96.17%
```

(b) Elaborate Union. Explain Union in detail with suitable program.

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

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

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

Here is the way you would define a union type named Data having three members i, f, and str

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

Accessing Union Members

To access any member of a union, we use the **member access operator** (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type.

Example:

```
#include<stdio.h>  
  
#include<conio.h>  
  
#include<math.h>  
  
typedef union  
{  
  
float aa;  
  
int bb;  
  
} nvals;  
  
typedef struct  
{  
  
float x;
```



```

char flag;

nvals exp;

}values;

void main()

{

values a;

int i;

float n,y;

clrscr();

printf("enter the value for x(ie,x power n)");

scanf("%f",&a.x);

printf("enter the value for n(ie,x power n)");

scanf("%f",&n);

i=(int)n;

a.flag=(i==n)?'i':'f';

if(a.flag=='i')

a.exp.bb=i;

else

a.exp.aa=n;

if(a.flag=='f'&& a.x<=0.0)

{

printf("can't rise a non positive number to a");

printf("\n floating point power");

```

```
}  
  
else  
  
{  
  
y=pow(a,x,n);  
  
printf("\ny=%.4f",y);  
  
}  
  
getch();  
  
}
```

OUTPUT:

```
Enter the value for x(ie,x power n)5  
Enter the value for n(ie,x power n)2  
y=25.0000> 
```