**ASSIGNMENT FILE**

**FOUNDATION OF C PROGRAMMING**



**BATCH 2023-2027**

**BCA(Hons.) AI & DS**

**SUBMITTED BY: SUBMITTED TO:**

**HIMANSHU MAWDI MR. RISHI KUMAR**

**23011782**

ASSIGNMENT

**Question 1:- What are constants and Variables, Types of Constants, Keywords, Rules of identifiers, int, float, char, double, long, void.**

**CONSTANTS:** These are fixed values that do not change throughout program execution.

**VARIABLES:** Variables are storage places with symbolic names that hold some known and unknown quantity of information known as a value.

**KEYWORDS:** In a computer language, they are reserved words with particular significance. They are not usable as identifiers.

**CONSTANT TYPES:** - Integer constants, floating-point constants, character constants, and string constants are all included.

**RULES OF IDENTIFIERS:** It must begin with a letter, be made up of letters, numerals, and underscores, and avoid reserved terms.

Datatypes include int, float, char, double, long, and void.

Integers (whole numbers) are represented by int.

float:- Represents floating-point values (decimal numbers).

Characters (single letters or symbols) are represented by char.

double: Represents floating-point numbers with double precision.

long: Long integers are represented by this type.

The lack of a data type or a function that does not return a value is represented by void.

**QUESTION 2:-Explain with example Arithmetic operators, increment and decrement operators, relational operators, logical operators, bitwise operators, conditional operators, type conversions, and expressions, precedence and associativity of operators.**

**ARITHMETIC OPERATOR:-**Arithmetic operators carry out fundamental mathematical operations. The most prevalent are

+ addition,

- subtraction, and

\* multiplication,

/ division, and

% modulus are all examples of operations.#include<stdio.h>

int main()

{

int a = 10, b = 3;

int addition = a + b; // 13

int subtraction = a - b; // 7

int multiplication = a \* b; // 30

int division = a / b; // 3

int modulus = a % b; // 1

return 0;

}

**INCREMENT/DECREMENT OPERATORS:-**

In C programmes, increment operators are used to increase the value of a variable by one and decrement operators are used to reduce the value of a variable by one.

->Pre-increment operator: ++var\_name:

Increment operator

post-increment operator var\_name++ ->Operator for incrementing:

--var\_name: operator for pre-decrement

post-decrement operator var\_nameEXAMPLE:-

#include<stdio.h>

int main()

{

int i=1;

while(i<10);

{

printf("%d ",i);

i++;

}

}

Output:1 2 3 4 5 6 7 8 9

**RELATIONAL OPERATORS:-**

The relational operator (= =) is used to determine if two values are equal or not.

If the values are equal, the result is "values are equal". Otherwise, "values are not equal" is printed as output.

To compare two values, use the double equal symbol (==). We should not use just one equal symbol (=). Example/Description

> x > y (x is greater than y)

< x < y (x is less than y)

>= x >= y (x is greater than or equal to y)

<= x <= y (x is less than or equal to y)

= = x == y (x is equal to y)

!= x != y (x is not equal to y

EXAMPLE:-

#include<stdio.h>

intmain()

{

Int m=40,n=20;

if (m == n)

{

printf("m and n are equal");

}

else

{

printf("m and n are not equal");

}

}

**LOGICAL OPERATOR**:-

These operators are used to conduct logical operations on the expressions provided to them.

In C, there are three logical operators. There are three of them: logical AND (&&), logical OR (||), and logical NOT (!).

|  |  |
| --- | --- |
| OPERATORS | EXAMPLE/DESCRIPTION |
| &&  (logical AND) | (x>5)&&(y<5)  It returns true when both conditions are true |
| ||  (logical OR) | (x>=10)||(y>=10)  It returns true when at-least one of the condition is true |
| !  (logical NOT) | !((x>5)&&(y<5))  It reverses the state of the operand “((x>5) &&  (y<5))”  If “((x>5) && (y<5))” is true, logical NOT operator make it false. |

EXAMPLE:-

#include<stdio.h>

int main()

{

int m=40,n=20;

int o=20,p=30;

if (m>n && m !=0)

{ printf("&& Operator : Both conditions are true\n");

}

if (o>p || p!=20)

{ printf("|| Operator : Only one condition is true\n");

}

if (!(m>n && m !=0)) { printf("! Operator : Both conditions are true\n");

}

else

{ printf("! Operator : Both conditions are true. " \ "But, status is inverted as false\n"); }

}

**BITWISE OPERATOR:-** These operators carry out bit operations.

Decimal numbers are transformed to binary values, which are a series of bits, and bitwise operators operate on these bits.

In C, bitwise operators are: & (bitwise AND), | (bitwise OR), (bitwise NOT), (XOR), (left shift), and >> (right shift).

|  |  |
| --- | --- |
| OPERATIONS | EXAMPLE/DESCRIPTION |
| & | BITWISE AND |
| | | BIWSE OR |
| ~ | BITWISE NOT |
| ^ | XOR |
| << | LEFT SHIFT |
| >> | RIGHT SHIFT |

#include<stdio.h>

int main()

{

int m = 40, n = 80, AND\_opr, OR\_opr, XOR\_opr, NOT\_opr ;

AND\_opr = (m&n);

OR\_opr = (m|n); NOT\_opr = (~m);

XOR\_opr = (m^n);

printf("AND\_opr value = %d\n", AND\_opr );

printf("OR\_opr value = %d\n", OR\_opr );

printf("NOT\_opr value = %d\n", NOT\_opr ); printf("XOR\_opr value = %d\n", XOR\_opr ); printf("left\_shift value = %d\n", m << 1); printf("right\_shift value = %d\n", m >> 1);

}

OUTPUT:-

AND\_oprvalue = 0

OR\_oprvalue = 120

NOT\_oprvalue = -41

XOR\_oprvalue = 120

left\_shiftvalue = 80

right\_shiftvalue = 20

**CONDITIONAL OPERATOR:-**

Conditional operators return one value when the condition is true and another value when the condition is false.

The ternary operator is another name for this operator.

Syntax : (Condition? true\_value : false\_value);

Example : (A > 100 ? 0 : 1);

In above example, if A is greater than 100, 0 is returned else 1 is returned.

This is equal to if else conditional statements.

EXAMPLE:-

#include<stdio.h>

int main()

{

int x=1, y; y = ( x ==1 ? 2 : 0 ) ;

printf("x value is %d\n", x);

printf("y value is %d", y);

}

OUTPUT:

x value is 1 y value is 2

**Operator Precedence and Associativity Rules for evaluation of expression**

1. From left to right, the first parenthesized subexpressions are evaluated.

2. When there are many parentheses, the evaluation starts with the innermost subexpression.

3. The precedence rule is used to determine the order in which operators are applied while evaluating subexpressions.

4. When two or more operators of the same precedence level exist in the sub expression, the associability rule is applied.

5. Arithmetic expressions are evaluated from left to right using precedence rules.

6. When using parentheses, the expressions within the parenthesis take precedence.

**QUESTION 3:- EXPLAIN WITH EXAMPLE CONDITIONAL STATEMENT IF, IF-ELSE, ELSE-IF, NESTED IF ELSE.**

**ANSWER:-**

**IF STATEMENT**

If the expression evaluates to true, the if block statement(s) are performed.

If the expression is evaluated as false, control is passed to the sentence that follows it.

Syntax: if (LogicalExpr) Statement

Program context:

statement1;

if (LogicalExpr) statement2;

statement3;

Example:

#include<stdio.h>

void main()

{

int a;

printf(“\n enter any number”);

scanf(“%d”, & a);

if(a%2==0)

{

printf(“even”);

}

getch();

}

**IF-ELSE STATEMENT**

If the condition is true or false, the code is executed. It's also known as a two-way selection statement.

**Syntax:**

if (LogicalExpr)

Statement A

else

Statement B

**Example:**

#include<stdio.h>

void main()

{

int a;

printf(“\n enter any number”);

scanf(“%d”, & a);

if(a%2==0) {

printf(“even number”);}

else

{ printf(“Odd number”);

}

getch();

}

**IF-ELSE-IF STATEMENT**

Used to run a single code from many circumstances.

It's also known as a multipath decision statement.

The following is an example of an if-else-if statement (3 levels).

**Example-**

#include<stdio.h>

int main()

{

int marks=80;

if(marks>75) {

printf("First class");

}

else if(marks>65)

{

printf("Second class");

}

else if(marks>55)

{

printf("Third class");

}

else { printf(“fourth class”);

}

return 0;

**NESTED IF-ELSE STATEMENT**

You may use the nested if-else statement to check for several test expressions and execute separate code for more than two situations.

Also known as a multiple-choice statement.

**Example-**

#include<stdio.h>

void main()

{

int a,b,c;

printf(“\n enter any three number”); scanf(“%d %d %d”, & a, &b, &c);

if(a>b)

{

if(a>c)

printf(“a is greatest”);

else printf(“c is greatest”);

else{if(b>c)

printf(“ b is greatest”);

else

printf(“ c is greatest”);

}

getch();

}

**QUESTION 4:- Explain Switch Case Statement With Example.**

A switch statement has one or more case labels that are compared to the switch expression.

When the expression matches a case, the statements connected with that case are performed.

It serves as a replacement for a lengthy if-else-if ladder. Float variables are not permitted to be utilised as switch expressions or case values.

Example:-

#include<stdio.h>

void main ()

{

int a,b, ch;

printf(“\n enter two numbers”); scanf(“%d%d”, &a, &b);

printf(“\n enter choice between 1,2,3”); scanf(“%d”, & ch); switch(ch)

{

case 1: printf(“\n sum=%d”, a+b);

break;

case 2 : printf(“\n Subtraction =%d’, a-b); break;

case 3 : printf(“\n Multiplication=%d”, a\*b); break;

default : printf(“\n Invalid choice" );

}

printf(“\n end of switch statement”);

getch();

}

**Question 5:- Explain Loop For Loop While Loop Do While Loop With Example**.

**LOOPS:-**

A loop is a set of instructions that is repeatedly executed until a given condition is met.

Loops allow the same statement to be executed several times in a row.

Entry-controlled and exit-controlled.

while, do-while and for loop.

For and while loop is entry-controlled loops.

do-while is an exit-controlled loop.

**FOR LOOP**

A for loop is a loop that repeats itself a certain number of times.

The loop employs a counter to determine how many times the same sequence of activities should be repeated.

**Syntax:**

for ( init; condition; increment )

{

statement(s);

}

**Example:-**

for(i=1;i<=3;i++)

printf(“\n %d”, i); O/P: 1 2 3

**WHILE LOOP**

A WHILE loop is a loop that runs until a condition is met.

It checks its status at the start of each loop.

**syntax:**

while (condition) statement;

If the condition is met, the statement is performed; otherwise, the condition is re-evaluated.

If a while statement's condition is initially false, the statement is never executed. As a result, we say that a while statement runs zero or more times.

**Example:-**

int a=5;

while(a<5)

{

printf("%d\n",a);

printf(“\n end of loop”);

**do-while loops**

A do-while loop, unlike a while loop, evaluates its state at the conclusion of the loop.

This indicates that its activity sequence is always repeated at least once.

**Syntax:**

do

{

statement

}while (expression);

**Example:**

int a=5;

do

{

printf(“\n%d”, a);

}while(a<5);

printf(“\n end of loop”);

**QUESTION 6:-EXPLAIN WITH EXAMPLES DEBUGGING IMPORTANCE, TOOLS COMMON ERRORS: SYNTAX,LOGIC, AND RUNTIME ERRORS, DEBUGGING, AND TESTING C PROGRAMS**

**ERRORS:-**

Syntax errors arise when a programmer makes faults in accurately inputting the syntax of the code or produces types.

Syntax mistakes arise when a programmer does not adhere to the set of rules.

C language syntax rules are defined.

Because syntax problems are always identified by the compiler, they are also known as compilation errors.

The following are the most common syntax problems in C:

A semicolon (;) is missing.

omitted parenthesis ()

Giving a variable a value without declaring it.

Example:-

#include<stdio.h>

int main()

{

printf(“HIMANSHU MAWDI”) //semicolon missing

return 0;

}

**RUNTIME ERRORS:-**

RunTime Errors are faults that arise when a programme is being executed, or performed. These faults show up following a successful programme compilation.

Because the compiler is unable to recognise runtime mistakes, they might be a bit difficult to find. They are only identifiable once the programme has begun to run.

*Errors in the Code*: The programmer fails to include a break statement when running a while loop.

This will cause an infinite loop in the programme, which will lead to a run-time error.

*Memory Leaks*: A programme may begin to leak memory, resulting in a run-time error, if a programmer constructs an array in the heap but fails to remove the array's contents.

*Inaccurate Mathematical Operations*: There will also be a run-time error when dividing an integer by zero or finding the square root of -1.

*Undefined Variables*: The application will produce a run-time error if a programmer fails to declare a variable in the code.

**Example:**

#include<stdio.h>

void main() {

int var;

var = 20 / 0; // Error as we dividing a number by 0

printf("%d", var);

}

**LOGICAL ERRORS:-**

Sometimes, when a programme is compiled and run, the results we were hoping for are not produced.

The result that is produced differs from what was anticipated, despite the code appearing to be error-free. We refer to these kinds of mistakes as logical errors.

Logical errors are ones in which our code appears to be accurate, compiles without issue, and produces no errors when it runs, but the result we receive is not what we anticipated.

Example:

#include <stdio.h>

int main()

{

int i;

for(i = 0; i <= 5; i++);//

{

printf("HIMANSHU MAWDI");

}

return 0;

}

In above Example we think the output of a code is 5 times HIMANSHU MAWDI but due to semi-colon it will print it only 1 time.

**QUESTION 7:- WHAT IS THE USER DEFINED AND PREDEFINED FUNCTION EXPLAIN WITH EXAMPLE CALL BY VALUE AND CALL BY REFERENCE.**

**USER-DEFINED FUNCTION**:- In the C programming language, a user-defined function is a function type that the user defines for a particular purpose. It gives our programme modularity and code reusability. Unlike built-in functions, user-defined functions have user-specified functionality and don't require a header file to be used.

**PRE-DEFINED FUNCTION:-** The system libraries already include definitions for these functions.

To develop code that is error-free, programmers will reuse the code that is already available in system libraries.

However, the user has to understand the function's syntax in order to utilise the library functions.

**CALL BY VALUE:-** The value of the real parameters is replicated into the formal parameters when using the call by value technique. Put differently, we may state that the call by value technique uses the variable's value in the function call.

The formal parameter cannot be used to change the value of the actual parameter in the call by value method.

Since the value of the real parameter is transferred into the formal parameter in a call by value, separate memory is allotted for the actual and formal parameters.

The argument used in the function call is known as the actual parameter, whereas the argument used in the function specification is known as the formal parameter.

Example:-

#include<stdio.h>

void change(int num) {

    printf("Before adding value inside function num=%d \n",num);

    num=num+100;

    printf("After adding value inside function num=%d \n", num);

}

int main() {

    int x=100;

    printf("Before function call x=%d \n", x);

    change(x);//passing value in function

    printf("After function call x=%d \n", x);

return 0;

}

**CALL BY REFERENCE:-** When a function call is made via reference, the address of the variable is supplied as the real parameter.

Since the address of the real parameters is given, the value of the actual parameters can be changed by altering the formal parameters.

The memory allocation for both formal and real arguments in call by reference is the same. The value stored at the address of the real parameters is the source of all actions within the function, and the updated value is also kept there.

**Example:-**

#include<stdio.h>

void change(int \*num) {

    printf("Before adding value inside function num=%d \n",\*num);

    (\*num) += 100;

    printf("After adding value inside function num=%d \n", \*num);

}

int main() {

    int x=100;

    printf("Before function call x=%d \n", x);

    change(&x);//passing reference in function

    printf("After function call x=%d \n", x);

return 0;

}

**QUESTION 8**

* **EXPLAIN WITH PASSING AND RETURNING ARGUMENT TO AND FROM FUNCTION**

A function that accepts parameters returns values and gets any data from the calling function. These functions are void and have values as returns.

Example:-

#include <stdio.h>

#include <string.h>

int function(int, int[]);

int main()

{

int i, a = 20;

int arr[5] = { 10, 20, 30, 40, 50 };

a = function(a, &arr[0]);

printf("value of a is %d\n", a);

for (i = 0; i < 5; i++) {

printf("value of arr[%d] is %d\n", i, arr[i]);

}

return 0;

}

int function(int a, int\* arr)

{

int i;

a = a + 20;

arr[0] = arr[0] + 50;

arr[1] = arr[1] + 50;

arr[2] = arr[2] + 50;

arr[3] = arr[3] + 50;

arr[4] = arr[4] + 50;

return a;

}

* **EXPLAIN STORAGE, CLASSES, AUTOMATIC STATIC, REGISTER, EXTERNAL.**
* **AUTO:-** This is the storage type that is used by default for any variables that are defined inside of blocks or functions. Thus, while creating C programmes, the keyword auto is rarely utilised. The scope of auto variables is defined by the fact that they may only be accessed inside the block or function in which they have been declared. Naturally, they are accessible from within nested blocks that are contained within the parent function or block that defined the auto variable.
* **STATIC:-** Declaring static variables—which are frequently used while building C programs—uses this storage class. One advantage of static variables is that their value persists even after they are removed from their scope! Therefore, inside their scope, static variables retain the value of their most recent usage. Thus, we may conclude that they are initialised just once and remain till the programme terminates. They are not re-declared, therefore no additional RAM is allocated.
* **REGISTER:**- The register variables declared by this storage class work in the same way as the auto variables. The compiler's attempt to save these variables in the microprocessor's register, if a free register is available, is the sole distinction. Because of this, using register variables during programme execution is far faster than using variables that are kept in memory.
* **EXTERN:**- The variable is defined outside of the block where it is utilised, as the external storage class just informs us. In essence, it has a value set to it in a distinct block, which may also be overridden or modified in a different block. In order for a global variable to be used elsewhere, it must first be initialised with a valid value where it is declared. This is what an extern variable is. It is accessible from any function or block.
* **WRITE A PROGRAM FOR TWO STRING S1 AND S2. DEVELOP A C PROGRAM FOR THE FOLLOWING OPERATION.**
* **DISPLAY A CONCATENATED OUTPUT OF S1 AND S2.**
* **COUNT THE NUMBER OF CHARACTERS EMPTY SPACES IN S1 AND S2.**

**QUESTION 9:- Explain with example 1D array and multidimensional array. Consider two matrices of the size M and N implement matrix multiplication operation and display result using functions. Write three functions**

**ONE-DIMENSIONAL ARRAY:-**

It is a list of variables with related data kinds.

It permits arbitrary access, and each element's index facilitates access to all of them.

The array's size is fixed.

**Example:-**

int arr[5]; //an array with one row and five columns will be created.

{a , b , c , d , e}

**MULTI-DIMENSIONAL ARRAY:-**

It is a list of lists made up of the same type of variable.

Additionally, random access is permitted, and each element's index facilitates access to all of them.

It is also observable as an assemblage of 1D arrays. Another name for it is the Matrix.

Its dimensions may be expanded from two to three, four, and so on.

We refer to them together as a multi-dimension array.

A 2D array is the most typical type of multidimensional array.

**Example:-**

int arr[2][5]; //an array with two rows and five columns will be created.

a b c d e

f g h i j

* **DISPLAY MATRIX**

#include <stdio.h>

int main()

{

int i, j, m, n;

int matrix[10][20];

printf("Enter number of rows : ");

scanf("%d", &m);

printf("Enter number of columns : ");

scanf("%d", &n);

for (i = 0; i < m; i++)

{

for (j = 0; j < n; j++)

{

printf("Enter data in [%d][%d]: ", i, j);

scanf("%d", &matrix[i][j]);

}

}

for (i = 0; i < m; i++)

{

for (j = 0; j < n; j++)

{

printf("%d\t", matrix[i][j]);

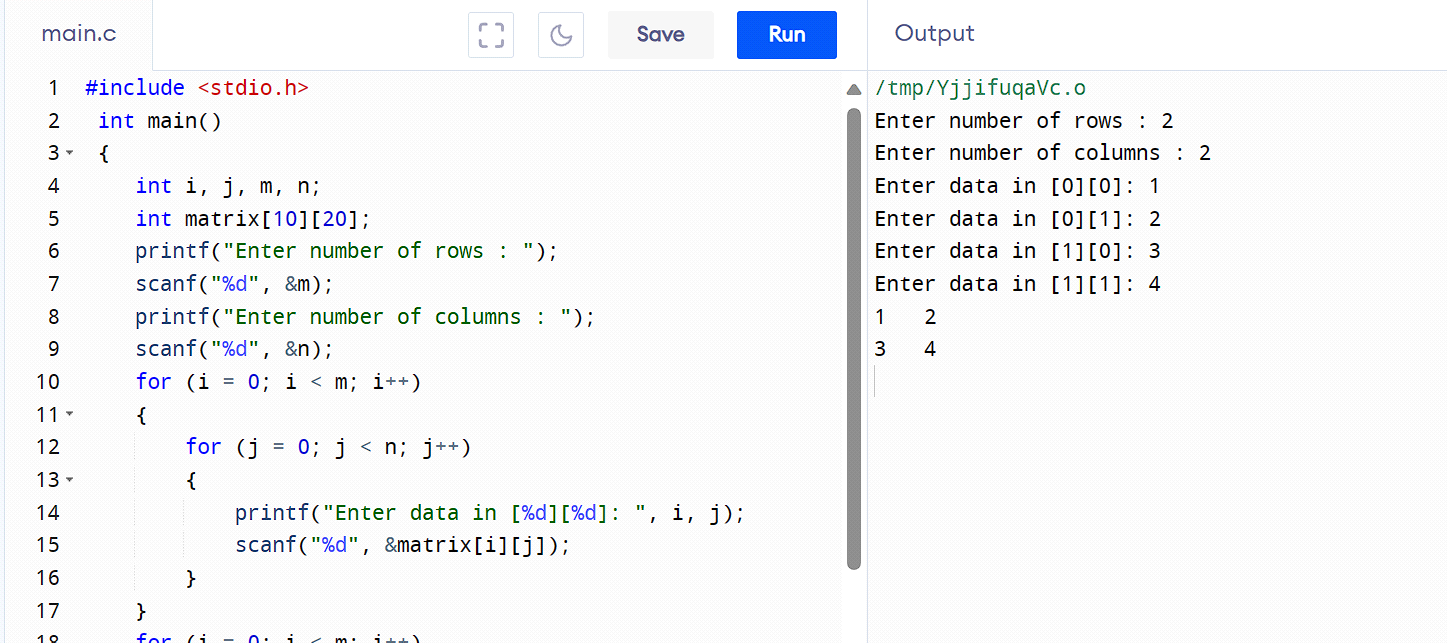
}

printf("\n");

}

return 0;

}



* **MATRIX MULTIPLICATION**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

printf("HIMANSHU MAWDI");

printf("\nenter the number of row=");

scanf("%d",&r);

printf("enter the number of column=");

scanf("%d",&c);

printf("enter the first matrix element=\n");

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

scanf("%d",&a[i][j]);

}

}

printf("enter these second matrix element=\n");

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

scanf("%d",&b[i][j]);

}

}

printf("multiply of the matrix=\n");

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

mul[i][j]=0;

for(k=0;k<c;k++)

{

mul[i][j]+=a[i][k]\*b[k][j];

}

}

}

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

printf("%d\t",mul[i][j]);

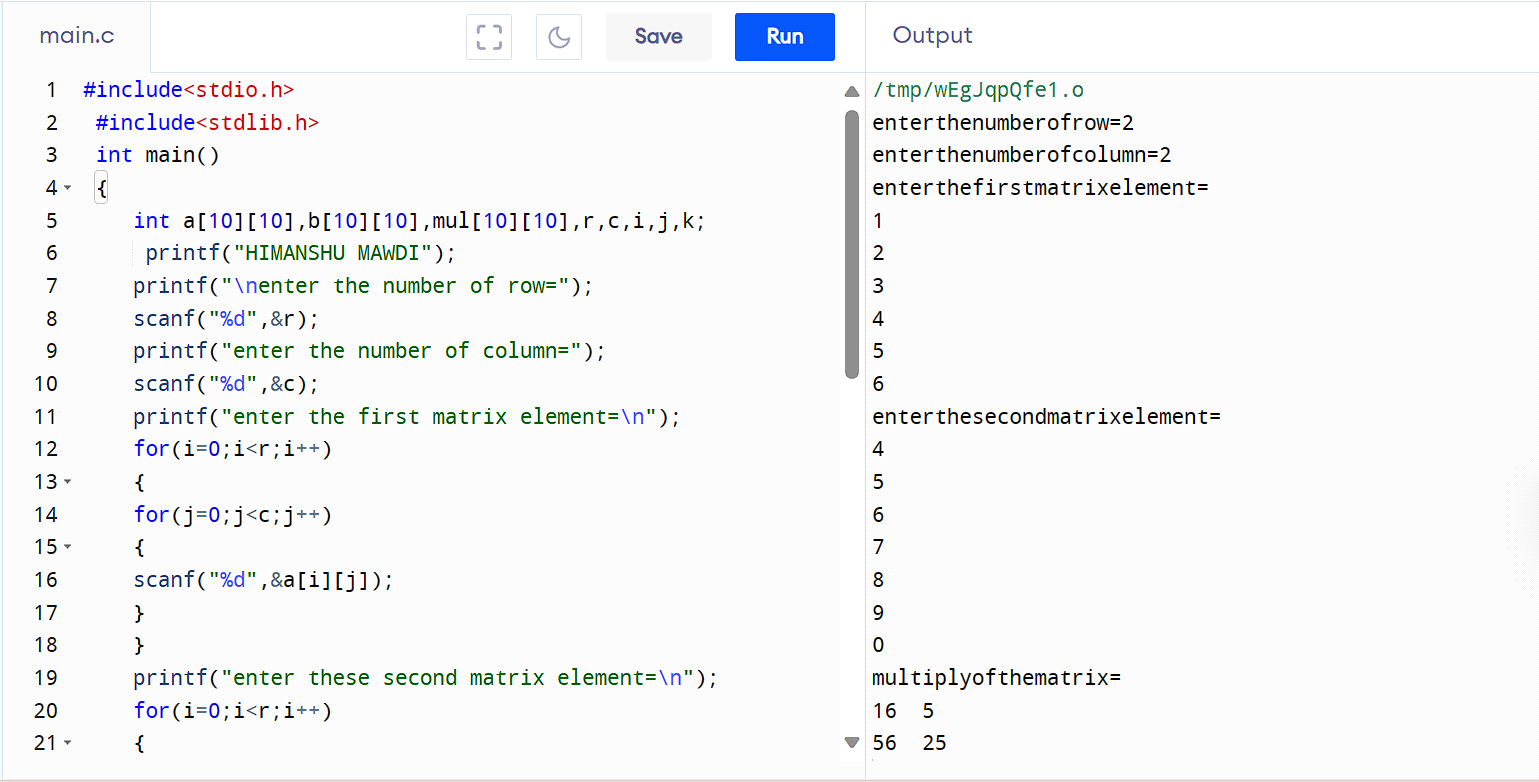
}

printf("\n");

}

return 0;

}



**Q10. Explain with example with structure,**

**declaration, and initialization, structure variables,array of structures, and use of typedef, passing structure to functions. Explain difference between structure and union. Write a program on details of a bank account with the fields account number, account holder’s name, balance. Write a program to read 10 people’s details and display the record with the highest bank balance.**

Answer. Structure:

• Declaration: This is where you specify the type and name of a variable without assigning a value to it.

• Initialization: This is the process of assigning an initial value to a variable.

Example –

struct BankAccount {

int accountNumber;

char accountHolder[50];

float balance;

};

int main() {

// Initializing a structure variable

struct BankAccount myAccount = {123456, "Farhan",

1000.50};

// Accessing structure members

print f("Account Number: %d\n",

myAccount.accountNumber);

print f ("Account Holder: %s\n",

myAccount.accountHolder);

printf("Balance: Rs%.2f\n", myAccount.balance);

return 0;

}

• Structure variables allow you to group different data

types under a single name. An array of structures is

an array where each element is a structure.

Example -

#include <stdio.h>

// Define a structure named 'Person'

struct Person {

char name[50];

int age;

float height;

};

int main() {

// Declare structure variables

struct Person person1, person2;

// Initialize structure variables

strcpy(person1.name, "Dubey”);

person1.age = 25;

person1.height = 1.75;

strcpy(person2.name, "Rancho");

person2.age = 22;

person2.height = 1.60;

// Print information using structure variables

printf("Person 1: Name=%s, Age=%d, Height=%.2f\n",

person1.name, person1.age, person1.height);

printf("Person 2: Name=%s, Age=%d, Height=%.2f\n",

person2.name, person2.age, person2.height);

// Declare and initialize an array of structures

struct Person peopleArray[3] = {

{"Virus", 30, 1.80},

{"Raju", 28, 1.65},

{"Chatur", 35, 1.70}

};

// Access and print information from the array of

structures

for (int i = 0; i < 3; ++i) {

printf("Person %d: Name=%s, Age=%d,

Height=%.2f\n", i + 1, peopleArray[i].name,

peopleArray[i].age, peopleArray[i].height);

}

return 0;

}

In this example, struct Person defines a structure with three members: a character array for the name, an integer for age, and a float for height. Two structure variables, person1 and person2, are declared and initialized.

Additionally, an array of three structures, peopleArray, is declared and initialized. The program then prints information about individuals using both structure

variables and the array of structures.

• In C programming, typedef is used to create an alias

or a new name for existing data types, including

structures. This can make your code more readable

and maintainable.

Example –

#include <stdio.h>

// Define a structure without typedef

struct Point {

int x;

int y;

};

// Define a structure with typedef

typedef struct {

int x;

int y;

} Point2D;

int main() {

// Using the structure without typedef

struct Point p1;

p1.x = 10;

p1.y = 20;

// Using the structure with typedef

Point2D p2;

p2.x = 30;

p2.y = 40;

// Passing structures to functions

printPoint(p1);

printPoint2D(p2);

return 0;

}

// Function to print a Point structure

void printPoint(struct Point p) {

printf("Point: (%d, %d)\n", p.x, p.y);

}

// Function to print a Point2D structure

void printPoint2D(Point2D p) {

printf("Point2D: (%d, %d)\n", p.x, p.y);

}

In this example, typedef is used to create an alias Point2D for the structure without having to use the keyword struct each time you declare a variable of that type.

The printPoint and printPoint2D functions demonstrate passing structures to functions. The structures are passed by value, meaning a copy of the structure is passed to the function. Keep in mind that for large structures, passing by reference (using pointers) might be more efficient.

• In programming, a structure is a composite data type that groups together variables of different types under a single name. Each variable, or member, within the structure has its own memory location.

Structures are used to represent records where each member holds different information.

• On the other hand, a union is also a composite data type that allows storing variables of different types in the same memory location. However, unlike structures, unions share the same memory space for all their members. This means that only one member of a union can be active at a time, and accessing one member may overwrite the data in another.

• Program on details of a bank account with the fields account number, account holder’s name, balance.

#include <stdio.h>

// Structure definition for bank account

struct BankAccount {

int accountNumber;

char accountHolderName[50];

double balance;

};

int main() {

// Declare and initialize a bank account

struct BankAccount myAccount;

// Input details

printf("Enter Account Number: ");

scanf("%d", &myAccount.accountNumber);

printf("Enter Account Holder's Name: ");

scanf("%s", myAccount.accountHolderName); //

Assuming a single-word name for simplicity

printf("Enter Balance: ");

scanf("%lf", &myAccount.balance);

// Display account details

printf("\nAccount Details:\n");

printf("Account Number: %d\n",

myAccount.accountNumber);

printf("Account Holder's Name: %s\n",

myAccount.accountHolderName);

printf("Balance: $%.2lf\n", myAccount.balance);

return 0;

}

This program uses a structure (struct BankAccount) to group the account details together. It then declares an instance of this structure (myAccount)

and takes input for the account details. Finally, it displays the entered details. Keep in mind that this is a basic example, and in a real-world scenario, you might want to add more features and error handling.

• Program to read 10 people’s details and

display the record with the highest bank

balance

struct BankAccount {

int accountNumber;

char accountHolder[50];

float balance;

};

int main() {

// Array of structures for 10 people's details

struct BankAccount accounts[10];

// Reading 10 people's details

for (int i = 0; i < 10; i++) {

printf("Enter Account Number for person %d: ", i

+ 1);

scanf("%d", &accounts[i].accountNumber);

printf("Enter Account Holder Name for person

%d: ", i + 1);

scanf("%s", accounts[i].accountHolder);

printf("Enter Balance for person %d: $", i + 1);

scanf("%f", &accounts[i].balance);

}

// Finding and displaying the record with the

highest bank balance

float maxBalance = accounts[0].balance;

int maxIndex = 0;

for (int i = 1; i <= 10; i++) {

if (accounts[i].balance > maxBalance) {

maxBalance = accounts[i].balance;

maxIndex = i;

} }

printf("\nPerson with the highest bank balance:\n");

printf("Account Number: %d\n",

accounts[maxIndex].accountNumber);

printf("Account Holder: %s\n",

accounts[maxIndex].accountHolder);

printf("Balance: $%.2f\n",

accounts[maxIndex].balance);

return 0;

}

This program allows the user to input details for 10 people's bank accounts and then displays the record with the highest bank balance