**ASSIGNMENT**

**QUESTION 1 :-** WHAT ARE CONSTANTS AND VARIABLES, TYPES OF CONSTANTS, KEYWORDS, RULES FOR IDENTIFIERS, INT, FLOAT, CHAR, DOUBLE, LONG, VOID.

**ANSWER:-**

Constants:

Constants are fixed values that do not change during the execution of a program. They are used to represent unchanging data, such as numerical values or strings. Constants can be categorized into two main types: literals and symbolic constants.

Types of Constants:

Literal Constants: These are specific values directly written in the source code. Examples include:

Integer literals (e.g., 42, -7)

Floating-point literals (e.g., 3.14, -0.001)

Character literals (e.g., 'A', '5')

String literals (e.g., "Hello, World!")

Boolean literals (e.g., true, false)

Symbolic Constants:-

These are defined names representing fixed values, often using the const keyword or preprocessor macros (in languages like C and C++).

Variables:-

Variables are used to store and manipulate data that can change during the execution of a program. They have a name, a data type, and a value. Variables allow you to work with dynamic or changing data within your program.

Keywords:-

Keywords are reserved words in a programming language that have specific meanings and cannot be used as identifiers (variable names or function names).

Examples of keywords in C and C++ include if, else, for, while, int, float, and return.

Rules for Identifiers:-

Identifiers are names given to variables, functions, classes, and other user-defined entities in a program.Rules for valid identifiers vary between programming languages but often include:

Starting with a letter or underscore.

Consisting of letters, digits, and underscores.

Not being a keyword.

Being case-sensitive (var Name and var name are different identifiers in most languages).

Data Types:-

Data types define the kind of data that a variable can hold and the operations that can be performed on it.

Common data types in programming include:

int: Represents integers (whole numbers).

float: Represents floating-point numbers with decimal places.

char: Represents a single character.

double: Represents double-precision floating-point numbers.

long: Typically represents a larger range of integers.

void: Used as a return type in functions to indicate that they don't return a value.

**QUESTION 2 :-** EXPLAIN WITH EXAMPLES ARITHMETIC OPERATORS, INCREMENT AND DECREMENT OPERATORS, RATIONAL OPERATORS, LOGICAL OPERATORS, BITWISE OPERATORS, CONDITIONAL OPERATORS, TYPE CONVERSION, AND EXPRESSION, PRECEDENCE, AND ASSOCIATIVITY OF OPERATORS.

**ANSWER:-**

1 ) Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations on operands. Common arithmetic operators include addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%).

Example:

int a = 10, b = 3;

int sum = a + b;// sum = 13

int difference = a - b;// difference = 7

int product = a \* b; // product = 30

int quotient = a / b; // quotient = 3

int remainder = a % b;// remainder = 1

2 ) Increment and Decrement Operators:

Increment (++) and decrement (--) operators are used to increase or decrease the value of a variable by 1.

Example:

int x = 5;

x++; // x is now 6

x--; // x is now 5 again

3 ) Relational Operators:

Relational operators are used to compare two values and return a Boolean result (true or false). Common relational operators include equality (==), inequality (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

Example:

int p = 5, q = 7;

bool is Equal = (p == q); // is Equal is false

bool is Not Equal = (p != q); // is Not Equal is true

bool is Greater = (p > q); // is Greater is false

bool is Less = (p < q); // is Less is true

4 ) Logical Operators:

Logical operators are used to perform logical operations on Boolean values. Common logical operators include AND (&&), OR (||), and NOT (!).

Example:

bool a = true, b = false;

bool result1 = a && b; // result1 is false

bool result2 = a || b; // result2 is true

bool result3 = !a; // result3 is false

5 ) Bitwise Operators:

Bitwise operators are used to manipulate individual bits of integer values. Common bitwise operators include bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise NOT (~), and left shift (<<) and right shift (>>).

Example:

int num1 = 5; // Binary: 0101

int num2 = 3; // Binary: 0011

int resultAND = num1 & num2; // resultAND is 1 (Binary: 0001)

int resultOR = num1 | num2; // resultOR is 7 (Binary: 0111)

6 ) Conditional (Ternary) Operator:

The conditional operator (?:) is used to create a conditional expression with three operands. It is a shorthand way of writing an if-else statement.

Example:

int age = 18;

bool canVote = (age >= 18) ? true : false;

7 ) Type Conversion and Casting:

Type conversion is the process of changing the data type of an expression or a variable. Casting allows you to explicitly change the data type of a value.

Example:

int x = 10;

double y = (double)x; // Explicit casting to double

8 ) Expression, Precedence, and Associativity of Operators:

An expression is a combination of values, variables, and operators that can be evaluated to produce a result. Precedence determines the order in which operators are evaluated. Higher precedence operators are evaluated first. Associativity defines the order in which operators with the same precedence are evaluated (left-to-right or right-to-left).

Example:

int result = 5 + 3 \* 2; // Here, multiplication (\*) has higher precedence, so it's evaluated

**QUESTION 3 :-** EXPLAIN WITH EXAMPLES CONDITIONAL STATEMENTS IF, IF-ELSE, ELSE IF, NESTED IF ELSE.

**ANSWER:-** Here are some commonly used conditional statements:

1 ) if Statement:

The if statement is used to execute a block of code if a specified condition is true. If the condition is false, the code inside the if block is skipped.

Syntax:

if (condition)

{

// Code to be executed if the condition is true

}

Example:

int age = 25;

if (age >= 18)

{

printf("You are an adult.\n");

}

2 ) if-else Statement:

The if-else statement is used to execute one block of code if a condition is true and another block if the condition is false.

Syntax:

if (condition)

{

// Code to be executed if the condition is true

} else

{

// Code to be executed if the condition is false

}

Example:

int temperature = 30;

if (temperature > 25)

{

printf("It's hot outside.\n");

} else

{

printf("It's not too hot outside.\n");

}

3 ) else if Statement:

The else if statement is used when you have multiple conditions to check. It allows you to test multiple conditions one by one.

Syntax:

if (condition1)

{

// Code to be executed if condition1 is true

} else if (condition2)

{

// Code to be executed if condition2 is true

} else

{

// Code to be executed if none of the conditions are true

}

Example:

int score = 85;

if (score >= 90)

{

printf("You got an A.\n");

} else if (score >= 80)

{

printf("You got a B.\n");

} else if (score >= 70)

{

printf("You got a C.\n");

} else

{

printf("You need to improve.\n");

}

4 ) Nested if-else Statement:

Nested if-else statements are used when you want to have an if-else statement inside another if or else block.

Example:

int num = 10;

if (num > 0)

{

if (num % 2 == 0)

{

printf("The number is positive and even.\n");

} else

{

printf("The number is positive and odd.\n");

}

} else

{

printf("The number is not positive.\n");

}

**QUESTION 4 :-** EXPLAIN SWITCH CASE STATEMENT WITH EXAMPLE .

**ANSWER:-**

The switch statement is a control structure in many programming languages, including C, C++, Java, and others, used for making decisions based on the value of an expression. It allows you to choose from a set of options (cases) based on the value of the expression. Here's how the switch statement works and an example:

Syntax of the switch statement:

switch (expression)

{

case value1:

// Code to be executed if expression matches value1

break; // Optional, used to exit the switch block

case value2:

// Code to be executed if expression matches value2

break;

// Add more cases as needed

default:

// Code to be executed if none of the cases match the expression

}

The switch statement evaluates the expression and then compares it to the values associated with each case. If a case value matches the expression, the code block following that case is executed.

The break statement is used to exit the switch block. If a break is not used, execution will continue into the next case. This can be useful for handling multiple cases with the same code. If none of the case values match the expression, the default case (if present) is executed.

Example of a switch statement in C:

#include <stdio.h>

int main()

{

int choice;

printf("Select a number between 1 and 3: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("You chose 1.\n");

break;

case 2:

printf("You chose 2.\n");

break;

case 3:

printf("You chose 3.\n");

break;

default:

printf("Invalid choice.\n");

}

return 0;

}

**QUESTION 5 :-** EXPLAINS LOOPS, FOR LOOP, WHILE LOOP, DO WHILE, LOOP WITH EXAMPLES .

**ANSWER :-**

Loops are control structures in programming that allow you to execute a block of code repeatedly, as long as a certain condition is met. Loops are essential for automating repetitive tasks and iterating over data structures. There are several types of loops in most programming languages, including the for loop, while loop, and do-while loop. Let's explain these loop types with examples:

1 ) for Loop:

The for loop is used when you know in advance how many times you want to execute a block of code. It typically consists of three parts: initialization, condition, and iteration.

Syntax:

for (initialization; condition; iteration)

{

// Code to be executed repeatedly

}

Example:

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

{

printf("Iteration %d\n", i);

}

In this example, the loop runs five times, and the loop variable i is incremented by 1 in each iteration.

2 ) while Loop:

The while loop is used when you want to repeatedly execute a block of code as long as a condition is true. The condition is checked before each iteration, and if it becomes false, the loop terminates.

Syntax:

while (condition)

{

// Code to be executed repeatedly

}

Example:

int count = 0;

while (count < 3)

{

printf("Count is %d\n", count);

count++;

}

This while loop will execute three times since the condition count < 3 becomes false when count reaches 3.

3 ) do-while Loop:

The do-while loop is similar to the while loop, but it guarantees that the block of code is executed at least once because the condition is checked at the end of the loop.

Syntax:

do

{

// Code to be executed repeatedly

} while (condition);

Example:

int num = 5;

do

{

printf("Number is %d\n", num);

num--;

} while (num > 0);

Even if num is initially 5, the loop will execute five times because the condition is checked at the end of each iteration.

**QUESTION 6 :-** Explain with examples debugging importance, tools common errors: syntax, logic, and runtime errors, debugging, and Testing C Programs .

**ANSWER :-**

Debugging is a critical and essential process in software development. It involves identifying and correcting errors or bugs in your code to ensure that your program behaves as expected. Debugging is important because it helps improve the reliability and quality of your software. There are various types of errors you may encounter, including syntax errors, logic errors, and runtime errors, each of which requires different approaches to debugging. Additionally, testing is an integral part of the debugging process to verify that your program functions correctly.

Importance of Debugging:

Debugging helps you find and fix errors in your code, preventing unexpected and incorrect behavior. It ensures that your program works as intended, which is crucial for delivering high-quality software. Debugging saves time and effort by addressing issues early in the development process.

Types of Common Errors:

1 ) Syntax Errors: These errors occur when the code violates the rules and structure of the programming language. They are detected by the compiler or interpreter and prevent the program from running.

Example:

int x = 10

// Missing semicolon at the end of the line is a syntax error.

2 ) Logic Errors: Logic errors result from flawed program logic or algorithms. The code may be syntactically correct, but it does not produce the desired output due to incorrect calculations or conditions.

Example:

// This code calculates the average incorrectly.

int sum = 10 + 20 + 30;

int average = sum / 3; // Incorrect, it should be sum / 3.0

3 ) Runtime Errors: These errors occur during program execution and can lead to program crashes, exceptions, or unexpected behavior. They are not always detected by the compiler but are revealed during runtime.

Example:

int denominator = 0;

int result = 10 / denominator; // Division by zero, a runtime error.

Debugging:

Debugging tools, integrated development environments (IDEs), and code editors provide features for debugging, such as breakpoints, step-by-step execution, and variable inspection. Techniques include running your program with different inputs, using print statements to trace program flow and variable values, and isolating problematic code.

Testing C Programs:

Testing is the process of systematically evaluating your program to ensure it meets the specified requirements and works correctly. Types of testing include unit testing, integration testing, system testing, and user acceptance testing. Use test cases with known inputs and expected outputs to verify the behavior of your program. Automation tools like testing frameworks can help streamline the testing process.

Example of Debugging and Testing in C:

#include <stdio.h>

int main()

{

int x = 10;

int y = 5;

int result = x + y;

// Debugging: Use print statements to check variable values and program flow.

printf("x = %d, y = %d\n", x, y);

printf("result = %d\n", result);

// Testing: Check if the result is as expected for various test cases.

if (result == 15)

{

printf("Test passed!\n");

} else

{

printf("Test failed!\n");

}

return 0;

}

**QUESTION 7 :-** What is the user defined and pre-defined functions. Explain with example call by value and call by reference.

**ANSWER :-**

User-defined functions and predefined (or built-in) functions are two fundamental types of functions in programming:

User-Defined Functions:

User-defined functions are functions that you create yourself within your program to perform specific tasks. These functions are defined by the programmer and can be customized to suit the requirements of the program. They help in organizing and modularizing code, making it more readable and maintainable.

Example of a user-defined function in C:

#include <stdio.h>

// Function to add two numbers

int add(int a, int b) {

return a + b;

}

int main() {

int result = add(5, 3);

printf("The sum is: %d\n", result);

return 0;

}

In this example, the add function is a user-defined function that takes two integer arguments, adds them, and returns the result.

Predefined (Built-In) Functions:

Predefined functions, also known as built-in or standard library functions, are functions that are provided by the programming language or external libraries. These functions are ready-made and serve common purposes, such as input/output, math operations, string manipulation, and more.

Example of a predefined function in C (using the standard library function printf):

#include <stdio.h>

int main()

{

int num = 42;

printf("The number is: %d\n", num);

return 0;

}

In this example, printf is a predefined function from the C standard library used to print output to the console.

Call by Value and Call by Reference:

Call by Value:

In call by value, when a function is called, the values of the actual arguments (parameters) are copied into the formal parameters of the function. Any changes made to the formal parameters within the function do not affect the values of the actual parameters outside the function.

Example of Call by Value in C:

#include <stdio.h>

void modifyValue(int x) {

x = x \* 2;

}

int main() {

int num = 5;

modifyValue(num);

printf("The value of num is: %d\n", num);

return 0;

}

In this example, the modify Value function operates on a copy of the num variable, so the original value of num remains unchanged.

Call by Reference:

In call by reference, the memory address (reference) of the actual arguments is passed to the formal parameters. Any changes made to the formal parameters within the function directly affect the values of the actual parameters outside the function.

Example of Call by Reference in C (using pointers):

#include <stdio.h>

void modifyValue(int\* x)

{

\*x = (\*x) \* 2;

}

int main() {

int num = 5;

modifyValue(&num); // Pass the memory address of num

printf("The value of num is: %d\n", num);

return 0;

}

In this example, by passing a pointer to the modifyValue function, the original num variable is modified inside the function, and the change is reflected outside the function.

**QUESTION 8 :-** 1) Explain with Passing and returning arguments to and from Function. 2) Explain Storage classes, automatic, static, register, external. 3) Write a program for two strings S1 and S2. Develop a C Program for the following operations. a) Display a concatenated output of S1 and S2 b) Count the number of characters and empty spaces in S1and S2.

**ANSWER :-**

Passing and Returning Arguments to and from Functions:

Passing Arguments to a Function:

When you call a function in C or most programming languages, you can pass arguments to it. These arguments are provided in the function call. Function parameters (formal parameters) are placeholders for the values you pass. They are declared in the function's signature. Arguments can be passed by value or by reference (using pointers).

Example of passing arguments by value:

#include <stdio.h>

void displayValue(int x) {

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

}

int main() {

int num = 42;

displayValue(num);

return 0;

}

Returning Values from a Function:

Functions can also return values using the return statement. The type of the value returned by the function should match the function's return type, as declared in the function signature.

Example of returning a value from a function:

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

int main() {

int result = add(5, 3);

printf("The sum is: %d\n", result);

return 0;

}

Storage Classes (Automatic, Static, Register, External):

Storage classes in C determine the lifetime, scope, and storage location of variables.

1 ) Automatic: Variables declared as auto are by default automatic. They have local scope and are stored on the stack. They are implicitly initialized to garbage values.

2 ) Static: static variables have a longer lifetime than automatic variables. They retain their values between function calls and have file scope if declared outside of a function. Static variables are initialized only once.

3 ) Register: register variables are used to suggest to the compiler to store the variable in a CPU register for faster access. The storage location is implementation-dependent, and they have local scope.

4 ) External: extern variables are used for declaring a variable that is defined in another source file. They have global scope and storage allocated in a different translation unit.

C Program for String Operations:

a) Display a concatenated output of S1 and S2:

#include <stdio.h>

#include <string.h>

int main()

{

char s1[50], s2[50];

printf("Enter the first string (S1): ");

gets(s1);

printf("Enter the second string (S2): ");

gets(s2);

strcat(s1, s2); // Concatenate S2 to S1

printf("Concatenated string: %s\n", s1);

return 0;

}

b) Count the number of characters and empty spaces in S1 and S2:

#include <stdio.h>

#include <string.h>

int main() {

char s1[50], s2[50];

printf("Enter the first string (S1): ");

gets(s1);

printf("Enter the second string (S2): ");

gets(s2);

int len1 = strlen(s1); // Get the length of S1

int len2 = strlen(s2); // Get the length of S2

int spaceCount1 = 0, spaceCount2 = 0;

for (int i = 0; i < len1; i++) {

if (s1[i] == ' ') {

spaceCount1++;

}

}

for (int i = 0; i < len2; i++) {

if (s2[i] == ' ') {

spaceCount2++;

}

}

printf("S1: Characters = %d, Spaces = %d\n", len1, spaceCount1);

printf("S2: Characters = %d, Spaces = %d\n", len2, spaceCount2);

return 0;

}

**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 results using functions. Write three functions 1) Read matrix elements 2) Matrix Multiplication 3) Print matrix elements.

**ANSWER :-**

1D Array:

A 1D array is a linear collection of elements stored in contiguous memory locations. It's often used for storing a list of values of the same data type.

int arr[5] = {1, 2, 3, 4, 5}; // Example of a 1D array

Multidimensional Array:

A multidimensional array is an array with more than one dimension. In the case of a 2D array, it's like a table with rows and columns.

int matrix[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

}; // Example of a 2D array (3x3 matrix)

Matrix Multiplication Using Functions:

Here's an example of a C program that performs matrix multiplication using functions:

#include <stdio.h>

// Function to read matrix elements

void readMatrix(int m, int n, int matrix[m][n]) {

printf("Enter matrix elements (%d x %d):\n", m, n);

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

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

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

}

}

}

// Function to perform matrix multiplication

void multiplyMatrix(int m1, int n1, int matrix1[m1][n1], int m2, int n2, int matrix2[m2][n2], int result[m1][n2]) {

if (n1 != m2) {

printf("Matrix multiplication not possible.\n");

return;

}

for (int i = 0; i < m1; i++) {

for (int j = 0; j < n2; j++) {

result[i][j] = 0;

for (int k = 0; k < n1; k++) {

result[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

}

// Function to print matrix elements

void printMatrix(int m, int n, int matrix[m][n]) {

printf("Matrix:\n");

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

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

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

}

printf("\n");

}

}

int main() {

int m1, n1, m2, n2;

printf("Enter the dimensions of the first matrix (m1 n1): ");

scanf("%d %d", &m1, &n1);

int matrix1[m1][n1];

readMatrix(m1, n1, matrix1);

printf("Enter the dimensions of the second matrix (m2 n2): ");

scanf("%d %d", &m2, &n2);

int matrix2[m2][n2];

readMatrix(m2, n2, matrix2);

if (n1 != m2) {

printf("Matrix multiplication not possible.\n");

return 0;

}

int result[m1][n2];

multiplyMatrix(m1, n1, matrix1, m2, n2, matrix2, result);

printMatrix(m1, n1, matrix1);

printf("\nMultiplied by\n\n");

printMatrix(m2, n2, matrix2);

printf("\nEquals\n\n");

printMatrix(m1, n2, result);

return 0;

}

In this program, three functions are defined:

1 ) readMatrix: Reads matrix elements from the user.

2 ) multiplyMatrix: Performs matrix multiplication.

3 ) printMatrix: Prints the elements of a matrix.

**QUESTION 10 :-** Explain with example with Structure, Declaration, and Initialization, Structure Variables, Array of Structures, and Use of typedef, Passing Structures to Functions. Define union declaration, and Initialization Passing structures 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, and balance. Write a program to read 10 people’s details and display the record with the highest bank balance.

**ANSWER :-**

Structure vs. Union:-

* Structure is a composite data type that allows you to group variables of different data types under a single name. Each member within a structure occupies its own memory location, and the size of a structure is the sum of the sizes of its members.
* Union is another composite data type, but it allows you to store data of different data types in the same memory location. A union uses the memory space of its largest member, and only one member can have a value at any given time.

Example of Structure:

#include <stdio.h>

struct BankAccount

{

int accountNumber;

char accountHolder[50];

float balance;

};

int main()

{

struct BankAccount account1;

account1.accountNumber = 12345;

strcpy(account1.accountHolder, "John Doe");

account1.balance = 1000.0;

printf("Account Number: %d\n", account1.accountNumber);

printf("Account Holder: %s\n", account1.accountHolder);

printf("Balance: %.2f\n", account1.balance);

return 0;

}

Example of Union:

#include <stdio.h>

union Data {

int i;

float f;

char c;

};

int main()

{

union Data data;

data.i = 42;

printf("Integer: %d\n", data.i);

data.f = 3.14;

printf("Float: %.2f\n", data.f);

data.c = 'A';

printf("Character: %c\n", data.c);

return 0;

}

Bank Account Program:

Here's a program that reads details of 10 bank accounts and finds the record with the highest bank balance using a structure.

#include <stdio.h>

#include <string.h>

struct BankAccount {

int accountNumber;

char accountHolder[50];

float balance;

};

int main() {

struct BankAccount accounts[10];

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

printf("Enter account number, account holder, and balance for account %d: ", i + 1);

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

getchar(); // Consume newline character

fgets(accounts[i].accountHolder, sizeof(accounts[i].accountHolder), stdin);

accounts[i].accountHolder[strcspn(accounts[i].accountHolder, "\n")] = '\0'; // Remove newline

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

}

int maxIndex = 0;

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

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

maxIndex = i;

}

}

printf("Account with the highest 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;

}