C assignment

**INTRODUCTION TO C**

**PROGRAMMING**



**BATCH: - 2023 -2024**

**BCA 1st YEAR**

SUBMITTED BY: -

ASHISH CHAUHAN

SUBMITTED to: - Mr. rishi Kumar.

**Assignment**

Q1.What are Constants and variables, Types of Constants, Keywords, Rules for identifiers, int, float, char, double, long, void.

Q2. Explain with examples Arithmetic Operators, Increment and Decrement Operators, Relational Operators, Logical Operators, Bitwise Operators, Conditional Operators, Type Conversions, and expressions, Precedence, and associativity of operators.

Q3. Explain with Example conditional statements if, if-else, elseif, nested if else.

Q4. Explain Switch Case statement with example.

Q5. Explain Loops, for loop, while loop, do while loop with examples.

Q6. Explain with examples debugging importance, tools common errors: syntax, logic, and runtime errors, debugging, and Testing C Programs.

Q7. What is the user defined and pre-defined functions. Explain with example call by value and call by reference.

Q8. 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.

Q9. 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.

Q10. 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.

Q1: - What are Constants and variables, Types of Constants, Keywords, Rules for identifiers, int, float, char, double, long, void.

explanation of each:

1. **Constants and Variables**:
   * [A **constant** is a value that cannot change during the program’s execution](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/). [Examples include literal constants like 21, 'A', "Hello world!", and named constants like PI = 3.14159](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [A **variable** is an identifier with a value that can change during normal execution](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
2. **Types of Constants**:
   * [**Integer Constants**: An integer quantity which contains a sequence of digits with no decimal points](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**Floating point Constants**: These types of constants contain a decimal point or an exponent](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**Character Constants**: A single character enclosed in single quotes](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
3. [**Keywords**: Keywords are predefined, reserved words used in programming that have special meanings to the compiler](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/). [They are part of the syntax and they cannot be used as an identifier](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
4. **Rules for Identifiers**:
   * [A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [The first letter of an identifier should be either a letter or an underscore](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * You cannot use keywords like int, while etc. [as identifiers](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
5. **Data Types**:
   * [**int**: It is used to store integer numbers (any number including positive, negative and zero without decimal part)](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**float**: It is used to store single-precision floating point numbers](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**chart**: It is used to store single-bit characters](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**double**: It is used to store double precision floating point values](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**long**: It is used to represent whole numbers, much like ‘int’, but often with a greater capacity to hold larger values](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).
   * [**void**: The void data type, similar to the Nothing data type described earlier, is the data type for the result of a function that returns normally, but does not provide a result value to its caller](https://press.rebus.community/programmingfundamentals/chapter/constants-and-variables/).

Q2. Explain with examples Arithmetic Operators, Increment and Decrement Operators, Relational Operators, Logical Operators, Bitwise Operators, Conditional Operators, Type Conversions, and expressions, Precedence, and associativity of operators.

the explanations and examples for each of the operators mentioned:

1. **Arithmetic Operators**: These are used to perform mathematical operations like addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%). For example, if a = 10 and b = 20, then a + b = 30, a - b = -10, a \* b = 200, b / a = 2, and b % a = 0.
2. **Increment and Decrement Operators**: These are used to increase or decrease the value of a variable by 1. The increment operator (++) increases the value by 1, and the decrement operator (–) decreases the value by 1. For example, if a = 10, then a++ would make a = 11, and a-- would make a = 9.
3. **Relational Operators**: These are used to compare two values. They include: equal to (==), not equal to (! =), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). For example, if a = 10 and b = 20, then a == b is false, a != b is true, a > b is false, a < b is true, a >= b is false, and a <= b is true.
4. **Logical Operators**: These are used to combine two or more conditions. They include: logical AND (&&), logical OR (||), and logical NOT (!). For example, if a = true and b = false, then a && b is false, a || b is true, and! a is false.
5. **Bitwise Operators**: These are used to perform operations on bits. They include: bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise NOT (~), left shift (<<), and right shift (>>). For example, if a = 60 (binary: 0011 1100) and b = 13 (binary: 0000 1101), then a & b = 12 (binary: 0000 1100), a | b = 61 (binary: 0011 1101), a ^ b = 49 (binary: 0011 0001), ~a = -61 (binary: 1100 0011), a << 2 = 240 (binary: 1111 0000), and a >> 2 = 15 (binary: 0000 1111).
6. **Conditional Operators**: These are used to perform different computations depending on a condition. The conditional operator is also known as the ternary operator and is represented as? : For example, a = 10, b = 20, max = (a > b) ? a : b would result in max = 20.
7. **Type Conversions**: These are used to convert a variable from one data type to another. For example, if a = "123", then int(a) would convert a to an integer with a value of 123.
8. **Expressions, Precedence, and Associativity of Operators**: An expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Operator precedence determines the grouping of terms in an expression, which affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. For example, x = 7 + 3 \* 2; here, x is 13, not 20 because the multiplication operator (\*) has a higher precedence than the addition operator (+). Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Most of the operators have left-to-right associativity. For example, x = 100 / 10 / 5; here, x is 2, not 50.

Q3. Explain with Example conditional statements if, if-else, elseif, nested if else.

the explanations and examples for each of the conditional statements mentioned:

1. **if statement**: The if statement is used to specify a block of code to be executed if a condition is true. For example:

int a = 10;

if (a > 5) {

printf ("a is greater than 5");

}

In this example, the message “a is greater than 5” will be printed because the condition a > 5 is true.

1. **if-else statement**: The if-else statement is used to specify a block of code to be executed if a condition is true, and another block of code to be executed if the condition is false. For example:

int a = 10;

if (a > 20) {

printf ("a is greater than 20");

} else {

Printf ("a is not greater than 20");

}

In this example, the message “a is not greater than 20” will be printed because the condition a > 20 is false.

1. **else if statement**: The else if statement is used to specify a new condition if the first condition is false. For example:

int a = 10;

if (a > 20) {

printf ("a is greater than 20");

} else if (a > 5) {

Printf ("a is greater than 5");

} else {

Printf ("a is not greater than 5");

}

In this example, the message “a is greater than 5” will be printed because the first condition a > 20 is false, but the second condition a > 5 is true.

1. **nested if-else statement**: A nested if-else statement is an if-else statement with another if-else statement as its if block or else block. For example:

int a = 10;

if (a > 5) {

if (a > 20) {

printf ("a is greater than 20");

} else {

printf ("a is greater than 5 but not greater than 20");

}

} else {

Printf ("a is not greater than 5");

}

In this example, the message “a is greater than 5 but not greater than 20” will be printed because the first condition a > 5 is true, but the nested condition a > 20 is false.

Q4. Explain Switch Case statement with example.

the switch statement in C is a control statement that is used to test several values for equality against a single expression. It is often used as an alternative to if-else if-else chains.

Here’s an example of how a switch statement works:

#include <stdio.h>

int main () {

int day = 3;

switch (day) {

case 1:

printf("Monday");

break;

case 2:

printf("Tuesday");

break;

case 3:

printf("Wednesday");

break;

case 4:

printf("Thursday");

break;

case 5:

printf("Friday");

break;

case 6:

printf("Saturday");

break;

case 7:

printf("Sunday");

break;

default:

printf ("Invalid day");

}

return 0;

}

In this example, the switch statement checks the value of the variable day. Depending on the value, it prints the name of the corresponding day of the week. If day has a value not covered by any of the case statements, the default statement is executed, and “Invalid day” is printed.

The break keyword is used to exit the switch statement. Without it, the program would continue to the next case statement, even if a match has been found.

Q5. Explain Loops, for loop, while loop, do while loop with examples.

the explanations and examples for each of the loops mentioned:

1. **Loops**: In programming, a loop is used for executing a block of code repeatedly until a particular condition is met. There are three types of loops in C: for, while, and do-while.
2. **for loop**: The for loop is used when we know in advance how many times the loop needs to be executed. Here’s an example:

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

printf ("%d ", i);

}

This will print the numbers 0 to 4 on the console.

1. **while loop**: The while loop is used when we want the loop to be executed as long as the condition is true. Here’s an example:

int i = 0;

while (i < 5) {

printf ("%d ", i);

i++;

}

This will also print the numbers 0 to 4 on the console.

1. **do-while loop**: The do-while loop is similar to the while loop, but the difference is that the do-while loop will execute the block of code once before checking the condition. Here’s an example:

int i = 0;

do {

printf ("%d ", i);

i++;

} while (i < 5);

This will also print the numbers 0 to 4 on the console.

In all these examples, the loop will continue to print the next number and increment i by 1 as long as i is less than 5. Once i is no longer less than 5, the loop will stop.

Q6. Explain with examples debugging importance, tools common errors: syntax, logic, and runtime errors, debugging, and Testing C Programs.

**Debugging** is a crucial part of programming and involves identifying, isolating, and fixing problems (bugs) in code. It ensures that the program works as expected and helps improve its efficiency and performance. Here are some common types of errors and debugging tools used in C programming:

1. **Syntax Errors**: These are errors where the code doesn’t conform to the grammar rules of the C programming language. They are detected by the compiler and need to be fixed before the program can be run. For example, forgetting a semicolon at the end of a statement.

int main () {

int a = 5 // Syntax error: missing semicolon

return 0;

}

1. **Logic Errors**: These errors occur when the program doesn’t perform as intended despite having correct syntax. They are harder to detect as the compiler doesn’t flag them. For example, using the wrong variable in a calculation.

int main () {

int a = 5, b = 10, c = 20;

int sum = a + a; // Logic error: should be a + b + c

return 0;

}

1. **Runtime Errors**: These errors occur during the execution of the program. They can be caused by illegal operations such as division by zero or accessing an out-of-bounds array index.

int main () {

int a = 5, b = 0;

int c = a / b; // Runtime error: division by zero

return 0;

}

**Debugging Tools**: There are several tools available for debugging C programs. One of the most popular is **GDB (GNU Debugger)**. It allows you to step through your code line by line, inspect variables, and understand the flow of execution.

**Testing C Programs**: Testing is another important aspect of programming. It involves running your program with various inputs to ensure it behaves as expected. Unit tests (testing individual components of a program) and integration tests (testing how components interact) are common testing methodologies.

Remember, debugging and testing are not one-time activities but are part of the ongoing software development process. They help in maintaining the quality of the software and in reducing the cost of development in the long run by catching and fixing errors early.

Q7. What is the user defined and pre-defined functions. Explain with example call by value and call by reference.

In programming, there are two types of functions: **user-defined functions** and **pre-defined functions**.

**User-defined functions** are those that are defined by the programmer to perform a specific task. These functions are created according to the needs of the program or the logic supplied by the programmer.

Here’s an example of a user-defined function in Python:

def add numbers (a, b):

return a + b

result = add numbers (5, 10)

print(result) # Output: 15

In this example, add numbers is a user-defined function that takes two parameters a and b, and returns their sum.

**Pre-defined functions**, on the other hand, are those that are already defined and provided by the programming language. These functions perform common tasks and are built into the language.

Here’s an example of a pre-defined function in Python:

numbers = [1, 2, 3, 4, 5]

length = Len(numbers)

print(length) # Output: 5

In this example, Len is a pre-defined function in Python that returns the number of items in a list.

Now, let’s discuss **call by value** and **call by reference**.

**Call by value** means that the function creates a new copy of the variable’s value and works with it, so any changes made inside the function do not affect the original variable.

Here’s an example of call by value in Python:

def change value(num):

num = num + 10

return num

number = 5

new number = change value(number)

print(number) # Output: 5

print(new number) # Output: 15

In this example, the function change value does not modify the original variable number.

**Call by reference** means that the function works directly with the original variable, so any changes made inside the function affect the original variable.

Here’s an example of call by reference in Python:

def change list (original list):

original list. append (10)

numbers = [1, 2, 3, 4, 5]

change list(numbers)

print(numbers) # Output: [1, 2, 3, 4, 5, 10]

In this example, the function change list modifies the original list numbers by appending a new element to it.

Please note that in Python, the behaviour of call by value or call by reference depends on whether the data type is mutable (like lists) or immutable (like integers and strings). For mutable types, Python’s behaviour is similar to call by reference, while for immutable types, it’s similar to call by value. However, Python’s official documentation refers to this as “call by object reference”.

Q8. 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.

1. **Passing and returning arguments to and from a function**:

In C, we can pass values to a function and a function can also return a value. Here is an example:

#include <stdio.h>

// Function declaration

int add (int a, int b);

int main () {

int sum;

// Passing values to the function

sum = add (5, 10);

printf ("The sum is %d", sum);

return 0;

}

// Function definition

int add (int a, int b) {

// Returning a value from the function

return a + b;

}

In this example, we’re passing the values 5 and 10 to the add function. The add function returns the sum of these values, which is then stored in the sum variable.

1. **Storage classes**:

Storage classes in C are used to define the scope (visibility) and life-time of variables and/or functions. There are four types of storage classes:

* **Automatic**: These are local variables that are declared inside a function or a block. They are created when the function starts execution and get destroyed after the function ends. Example:
* void func {
* auto int x = 1; // 'x' is an automatic variable
* }
* **Static**: These variables retain their values between different function calls. They are initialized only once. Their default value is 0. Example:
* void func () {
* static int x = 0; // 'x' is a static variable
* x++;
* Printf d ", x);
* }
* **Register**: These are similar to automatic variables but they are stored in the CPU register instead of RAM to have faster access. Example:
* void func () {
* register int x; // 'x' is a register variable
* }
* **External**: These are global variables that are defined outside any function. They are visible to all functions. Example:
* extern int x; // 'x' is an external variable

1. Here is a C program that concatenates two strings and counts the number of characters and spaces:

#include <stdio.h>

#include <string.h>

int main () {

char S1[100] = "Hello, World!";

char S2[100] = " How are you?";

char S3[200];

int i, count = 0, spaces = 0;

// Concatenating S1 and S2

Strcpy (S3, S1);

Strcat (S3, S2);

Printf ("Concatenated string: %s\n", S3);

// Counting characters and spaces

for (i = 0; S3[i]! = '\0'; i++) {

count++;

if (S3[i] == ' ') {

spaces++;

}

}

Printf ("Number of characters: %d\n", count);

Printf ("Number of spaces: %d\n", spaces);

return 0;

}

In this program, S1 and S2 are concatenated and stored in S3. The program then counts the number of characters and spaces in S3. The strcpy function is used to copy S1 to S3, and the strcat function is used to append S2 to S3. The for loop iterates over each character in S3, incrementing count for each character and spaces for each space. The results are then printed.

Q9. 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.

1. **One-Dimensional (1D) Array**:

A one-dimensional array is a type of linear array, which contains only one row for storing data. It is the simplest form of an array. Here is an example of a 1D array in C:

#include <stdio.h>

int main () {

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

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

printf ("%d ", array[i]);

}

return 0;

}

In this example, array is a one-dimensional array that can hold 5 integers.

1. **Multi-Dimensional Array**:

A multi-dimensional array is an array of arrays. This means that a multi-dimensional array can be used to store rows and columns. Here is an example of a two-dimensional array (which is the simplest form of a multi-dimensional array) in C:

#include <stdio.h>

int main () {

int array [2][3] = {{1, 2, 3}, {4, 5, 6}};

for (int i = 0; i < 2; i++) {

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

printf ("%d ", array[i][j]);

}

printf("\n");

}

return 0;

}

In this example, array is a two-dimensional array that can hold 2 rows and 3 columns.

Now, let’s consider two matrices of size m x n and n x p. Here is a C program that performs matrix multiplication using functions to read matrix elements, perform the multiplication, and print the result:

#include <stdio.h>

void read matrix (int matrix [][10], int row, int col) {

int i, j;

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

for (j = 0; j < col; j++) {

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

}

}

}

void multiply matrix (int first [][10], int second [][10], int result [][10], int row, int col, int p) {

int i, j, k;

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

for (j = 0; j < p; j++) {

result[i][j] = 0;

for (k = 0; k < col; k++) {

result[i][j] += first[i][k] \* second[k][j];

}

}

}

}

void print\_matrix (int matrix [][10], int row, int col) {

int i, j;

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

for (j = 0; j < col; j++) {

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

}

printf("\n");

}

}

int main () {

int m, n, p;

int first [10][10], second [10][10], result [10][10];

printf ("Enter the number of rows and columns for the first matrix: ");

scanf ("%d %d", &m, &n);

printf ("Enter the elements of the first matrix:\n");

read matrix (first, m, n);

printf ("Enter the number of columns for the second matrix: ");

scanf ("%d", &p);

printf ("Enter the elements of the second matrix:\n");

read matrix (second, n, p);

multiply matrix (first, second, result, m, n, p);

printf("The result of matrix multiplication is:\n");

print matrix (result, m, p);

return 0;

}

In this program, read matrix reads the elements of a matrix from the user, multiply matrix performs the matrix multiplication, and print matrix prints the elements of a matrix. The main function gets the sizes of the matrices from the user, calls read matrix to read the matrices, calls multiply matrix to multiply them, and then calls print matrix to print the result. Note that the number of columns in the first matrix must be equal to the number of rows in the second matrix for matrix multiplication to be possible. In this program, that number is n. The resulting matrix will have m rows and p columns. Please replace 10 with the maximum expected size of m, n, or p in your actual use case.

Q10. 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.

Sure, let’s break it down:

**Structure in C:** A structure is a user-defined data type in C which allows you to combine data items of different kinds. Structures are used to represent a record.

struct Student {

char name [50];

int roll;

float marks;

};

**Declaration and Initialization of Structure Variables:** You can declare structure variables during the definition of the structure type or after defining the structure type.

struct Student s1 = {"John", 23, 78.5};

**Array of Structures:** An array of structures is similar to an array of fundamental types.

struct Student students[10];

**Use of typedef:** typedef is a keyword used in C language to assign alternative names to existing types.

typedef struct Student STUDENT;

STUDENT s1, s2;

**Passing Structures to Functions:** You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

void display (struct Student s) {

printf ("%s %d %.2f\n", s.name, s.roll, s.marks);

}

**Union in C:** A union is a special data type available in C that allows storing different types of data 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.

union Data {

int i;

float f;

char str[20];

};

**Difference between Structure and Union:** The key difference between structures and unions is that a structure uses all the memory of its members while a union uses the memory of the largest member.

Now, let’s write a program to manage bank account details and find the account with the highest balance.

#include <stdio.h>

#include <string.h>

struct BankAccount {

char accountNumber [20];

char accountHolderName [50];

float balance;

};

int main() {

struct BankAccount accounts [10];

int i;

float maxBalance = 0;

int maxIndex = 0;

// Assume that the accounts array is already filled with data

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

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

maxBalance = accounts[i].balance;

maxIndex = i;

}

}

Printf ("Account with the highest balance:\n");

Printf ("Account Number: %s\n", accounts[maxIndex].accountNumber);

Printf ("Account Holder's Name: %s\n", accounts[maxIndex].accountHolderName);

printf ("Balance: %.2f\n", accounts[maxIndex]. balance);

return 0;

}

This program assumes that the accounts array is already filled with data. It then finds the account with the highest balance and prints the details of that account. Please replace the comment with the actual data filling part in your real program.

TO: -

MR. RISHI KUMAR

Ass. Professor, CSIT, GEU, DEHRADUN