C Assignment

BATCH – 2023-24

BCA 1ST YEAR

NAME – SAHIL KUMAR

STUDENT ID – 2316912023 SECTION - I

SUBMITTED TO – MR. RISHI KUMAR

MS. POOJA CHAHAR

ASSIST. PROFESSOR, CSIT, GEU, DEHRADUN

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

Answer. **Constants are values that remains unchanged during the execution of the program, while variables are placeholders for storing and manipulating data.**

Types of Constants: -

1. **Integer constant**
2. **Floating-point constant**
3. **Character constant**

- **Keywords are reserved words in a programming language with specific meanings and cannot be used as identifiers.**
- **Rules of identifiers include starting with a letter or underscore, followed by letters, digits, or underscores. They cannot be keywords and should not contain spaces.**
- **Int, float, char, double, long, and void are data types in programming language. They define the type of data a variable can hold. For example, int is used for integers, float for floating-point numbers, char for characters, double for double-precision floating-point numbers, long for long integers, and void indicates that a function returns no value.**

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.

Answer. Arithmetic operators are symbols used to perform mathematical operations on numerical values. These operators are fundamental in performing basic mathematical calculations in programming and mathematics.

Here are some examples of common arithmetic operator:

1. Addition (+)- 5+3=8

2. Subtraction (-) – 8-5=3

3. Multiplication (*) – 8*5=40

4. Division (/) – 8/4=2

5. Modulus (%) – Returns the remainder after division

Ex – 5%2=1

6. Exponentiation (or^) – Raises the number to the power of another.**

Ex – 23=8**

- **Increment (++) and Decrement (--) operators are used to increase or decrease the value of a variable by 1, respectively. The ++ operator adds 1 to the variable and – operator subtracts 1 from the variable. These operators can be used with variables of numeric types like int, float and double.**
  Example (++)
  int x=5;
  x++; // increment x by 1
  output = 6
  Example (--)
  int x=5;
  x--; // decrement x by 1
  output = 4

- **Relational operators are used to compare values in programming. These operators are commonly used in conditional statements and loops to make decisions on the relationship between different values.**
  Here are some common relational operators:
  1. Equal to (==):
  Ex – 5==5 evaluates to true, while 5==3 evaluates false.
  2. Not equal to (! =) :
  Ex – 5! =3 evaluates to true, while 5! =5 evaluates false.
  3. Greater than (>):
  Ex – 5>3 evaluates to true, while 3>5 evaluates false.
  4. Less than (<): Ex – 3<5 evaluates to true, while 5<3 evaluates false.
  5. Greater than or equal to (>=): Ex – 5>=5 evaluates to true, while 4>=5 evaluates false.

6. Less than or equal to (<=): Ex – 5<=6 evaluates to true, while 5<=4 evaluates false.

- **Logical operators are used to perform logical operations (true or false). These operators are fundamental in logic and are widely used in programming to make decisions based on certain conditions. Here are common logical operators: -**
  1. AND (&&):
  Ex – A && B is true only if both A and B are true.
  If x=true
  y=false
  result= x and y // result is false
  2. OR (||):
  Ex – A || B is true if at least one of A or B is true
  P=true;
  Q=false;
  Result=p||q; // result is true
  3. NOT (!):
  Ex- (!) A is true if A is false, and vice versa.
  flag=true;
  negated Flag = (!)flag; // negated Flag is false

- **Bitwise operators are used to perform operations on individual bits of binary numbers. These operators are often used in low-level programming, for tasks like optimizing code or dealing with hardware-level operations. Here are some common bitwise operators: -**
  1. AND (&): Returns 1 for each bit position where both operands have 1.

Ex- 1010
&1100 = 1000

2. OR (|): Returns 1 for each bit position where only one operand has 1.
Ex – 1010
| 1100 = 1110
3. XOR (^): Returns1 for each bit position where only one operand has 1.
Ex – 1010
^ 1100= 0110
4. NOT (~): Flips the bits, changing 1s to 0s and vice versa.
Ex - ~1010 = 0101
5. Left Shift (<<): Shifts the bits to the left by a specified number of positions, filling with zeros on the right.
Ex – 1010<<2 = 101000
6. Right Shift (>>): Shifts the bits to the right by a specified number of positions. The leftmost bits are filled based on the sign bit for signed integers.
Ex – 1010>>1 = 1101

- **In C, type conversions and expressions play a crucial role in manipulating data of different types. It is essential for writing efficient and correct C programs. Here are some examples: -**
  1. Implicit Type conversion: This happens automatically by the compiler.
  Ex – int a=5;
  float b=2.5;

float result= a + b; // Implicit conversion of a to float before addition.

2. Explicit Type Conversion (Casting):

This involves the explicit use of casting operators.

Ex –

```
int x=10;
float y=3.14;
int sum= x+(int)y; // Explicitly casting y to int before addition.
```

3. Expressions: These are combinations of values and operators that can be evaluated to produce a result.

Ex –

```
int a=5, b=3, c=2;
int result = a*b + c; // Expression involving multiplication and addition
```

4. Mixing Data Types: C allows mixing data types in expressions, but be cautious about potential loss of precision.

Ex –

```
int a=10;
double pi=3.14159;
double result=num*pi; // Mixing int and double in an expression
```

5. Type Promotion: When different data types are involved in an operation, C automatically promotes them to a common type.

Ex-

```
int a=5;
double b=2.5;
double result= a + b; // a is promoted to double before addition
```

- **Precedence refers to the priority of operators, determining which one gets executed first. For example, in the expression 2+3*4, multiplication has a higher precedence than addition, so 3*4 is evaluated first, resulting in 2+12.**

- **Associativity is relevant when there are multiple operators with the same precedence. It defines the order in which operators of equal precedence are processed. Commonly, arithmetic operators have left-to-right associativity. For example, in the expression 5-3+2, the subtraction and addition have the same precedence, and because of left-to-right associativity, it's evaluated as (5-3) +2.**

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

Answer. Conditional statements are used to execute a set of statements on some conditions.

1. IF Statement: - If the expression is true, then if block statement are executed and if false than passed to the next statement.

Syntax: if *(Logical Expression) Statement*

Example –

```
int x=10;
if(x>5) {
 print f ("x is greater than 5\n");
}
```

In this example, the if statement checks if the value of x is greater than 5. If the condition is true, it executes the code inside the curly brackets.

2. If-else: - Used to execute the code if condition is true or false.

Syntax: -

if *(Logical Expr)*

statement A

else

statement B

Example: -

```
int y=3;
if(y>5) {
print f (“y is greater than 5\n”);
} else {
print f (“y is lesser than 5\n”):
}
```

In this example, if the condition (y>5) is false, the code inside the else block will be executed.

3. Else-if: - Used to execute one code from multiple conditions.

Syntax: -

```
if (condition 1)
   statement 1;
else if (condition 2)
   statement 2;
else
    statement 4;
next statement;
```

Example: -

```
int marks=100;
if (marks>80) {
   print f (“Section A\n”);
} else if (marks>60) {
```

```
print f (“Section B\n”);
} else {
print f (“Section C\n”);
}
```

In this example, if the condition (marks>80) is false than else if will executed again if that condition is also false than else block will be executed.

4. Nested if-else: - It allows you to check for multiple test expression and execute different codes for more than 2 conditions.

Example –

```
int a, b, c;
print f (“Enter three numbers\n”);
scan f (“%d %d %d”, &a, &b, &c);
if (a>b) {
If (a>c)
print f (“a is greatest\n”);
else
print f (“c is greatest\n”);
} else {
if (b>c)
print f (“b is greatest\n”);
else
print f (“c is greatest\n”);
}
```

This example shows how to use multiple conditions with nested if-else.

Q4. Explain switch case statement with example.

Answer. A switch statement is used to make decisions based on the value of a variable or expression. It provides a more readable and efficient way to write code for multiple conditional branches.

Example:

```
int choice;
print f (“Enter a number (1-3): \n”);
scan f (“%d”, &choice);
switch (choice) {
case 1: print f (“You choose 1.\n”); break;
case 2: print f (“You choose 2.\n”); break;
case 3: print f (“You choose 3.\n”); break;
default: print f (“Invalid choice.\n”);
}
```

In this example, the user inputs a number, and the switch statement checks the value of choice. Depending on the value, it executes the corresponding case. The break statement is essential to exit the switch block after a case is executed. The default case is optional and provides a default action if none of the specified cases match the value of the expression.

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

Answer. A loop is a sequence of instructions that is continually repeated until a certain condition is reached.

1. for Loop: - A for loop is a loop that repeats a specified number of times.

Syntax: - for (initialization; condition; update) {

```
        statement(s);
}
```

Example –

```
for (int i=0; i<5; i++) {
print f (“%d”, i);
} Output: - 0, 1, 2, 3, 4
```

2. while Loop: - A WHILE loop is a loop that repeats while some condition is satisfied.

Syntax: - while (condition)

```
        statement;
```

Example –

```
int i=0;
while (i<5) {
print f (“%d”, i);
 i++;
}
```

3. do-while Loop: - Unlike a while loop, a do-loop tests its condition at the end of the loop.

Syntax: - do

```
statement
while (expression);
```

Example –

```
int i=0;

do {

 print f (“%d”, i); i++;

} while (i<5);
```

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

Answer. Debugging is crucial in programming to identify and fix errors in code. Debugging tools and practices play a vital role in identifying and fixing errors at various stages of program development, from catching syntax issues during compilation to addressing logic and runtime errors during execution. Additionally, incorporating testing methodologies, such as unit testing, contributes to creating robust and reliable C programs. Here are some common errors and tools for debugging in C: -

1. Syntax Errors: -

Example: Missing a semicolon at the end of a statement.

Tool: The compiler will usually catch syntax errors and point out the line where the issue occurs.

2. Logic Errors: -

Example: Incorrect mathematical operation leading to unexpected results.

Tool: Debugging tools like gdb (GNU Debugger) allow programmers to set breakpoints, inspect variables, and step through code to identify logic errors.

3. Runtime Errors: -

Example: Division by zero during program execution.

Tool: Tools like Valgrind can help detect memory-related runtime errors, while gdb can be used for runtime debugging.

4. Debugging: -

Example: Using print f statements to trace the flow of variables.

Tool: Debuggers like gdb provide a more systematic way to step through code, set breakpoints, and examine the program's state at different points.

5. Testing C Programs: -

Example: Writing test cases to ensure functions return expected results.

Tool: Unit testing frameworks like Check or C Unit help automate the testing process, ensuring that different parts of the program work as intended.

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

Answer. User-defined functions are functions created by the user to perform specific tasks. They are defined using a programming language's syntax and can be called within the program to execute the code encapsulated within them.

Pre-defined functions, on the other hand, are functions provided by the programming language or its libraries. These functions are built-in and can be directly used without the need for the user to define them.

- **Call by value and call by reference refer to the ways parameters are passed to functions.**

- **Call by Value: The value of the actual parameter is passed to the function. Any changes made to the parameter within the function do not affect the original value. It's commonly seen in languages like C.**
- **Example: -**

```
void square (int num) {
    num = num * num;
}
    int x = 5;
    square(x);
```

- **Call by Reference:** The memory address of the actual parameter is passed to the function. Changes made to the parameter within the function affect the original value. This is more common in languages like C++.

```
void square (int &num) {
    num = num * num;
}

int x = 5;
square(x);
```

In this example, the square function modifies the value at the memory address of x, so the change is reflected outside the function.

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 S1 and S2.

Answer. 1) Passing and Returning Arguments in Functions:

- **Passing Arguments:** Functions in C can receive input values, called arguments, by specifying them in the function's parameter list. For example, void myFunction(int a, float b) takes an integer a and a float b as arguments.

- **Returning Values: Functions can also return values using the return statement. For instance, int add (int x, int y) {**

return x + y;

} returns the sum of x and y.

2) Storage Classes in C:

- **Automatic Storage Class (auto): Variables declared inside a function without any storage class specifier have automatic storage. They are created when the function is called and destroyed when it exits.**
- **Static Storage Class (static): Variables declared as static inside a function retain their values between function calls. They have a longer lifetime than automatic variables.**
- **Register Storage Class (register): This suggests to the compiler that the variable is heavily used and should be stored in a register for faster access. However, the compiler may ignore this suggestion.**
- **External Storage Class (extern): Used to declare a variable that is defined in another file or at the top level of the current file. It allows sharing variables between different files.**

3) C Program for String Operations:

```
#include <stdio.h>

#include <string.h>


int main () {
```

```
    char S1[100], S2[100];
    // Input two strings
    print f ("Enter string S1: ");
    gets(S1);
    print f ("Enter string S2: ");
    gets(S2);
    // Concatenate and display
    print f ("Concatenated string: %s\n", strcat (S1, S2));
    // Count characters and spaces
    int char Count = strlen(S1) + strlen(S2);
    int space Count = 0;
    for (int i = 0; S1[i] != '\0'; i++) {
       if (S1[i] == ' ') {
          space Count++;
       }  }
    for (int i = 0; S2[i] != '\0'; i++) {
       if (S2[i] == ' ') {
          space Count++;
       } }
    print f ("Number of characters: %d\n", char Count);
    print f ("Number of spaces: %d\n", space Count);

    return 0;
}
```

This program takes two strings as input, concatenates them, and then calculates and displays the total number of characters and spaces in the combined string. Note that (gets) is used here for simplicity; in a real-world scenario, it's recommended to use (fgets) for safer string input.

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.

Answer. A 1D array is a linear collection of elements stored in contiguous memory location. It's like a list of items.

Example –

```
int main() {
    // Declaration and initialization of a 1D array
    int my Array[5] = {1, 2, 3, 4, 5};
    // Accessing elements of the 1D array
    for (int i = 0; i < 5; ++i) {
        print f("%d ", my Array[i]);
    }
    return 0;
}
```

- **A multidimensional array extends this concept to multiple dimensions. A common example is a 2D**

array, which is like a table with rows and columns. Here, each element in the 2D array has 2 indices: one for the row and one for the column. The value 5 is accessed as two dimensional array [1][1] (second row, second column).

Example –

```
int main() {
    // Declaration and initialization of a 2D array
    int my Matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    // Accessing elements of the 2D array
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            print f("%d ", my Matrix[i][j]);
        }
        print f("\n");
    }
    return 0;
}
```

Here, my matrix is a 2D array representing a 3*3 matrix. Elements are accessed using two indices (row and column), and the program prints the matrix.

- **Example of matrix multiplication in C programming using three functions for reading matrix elements, performing matrix multiplication, and printing matrix elements.**

```
#include <stdio.h>

#define MAX_SIZE 10

// Function to read matrix elements
void read Matrix(int matrix[MAX_SIZE][MAX_SIZE], int
rows, int cols) {
    print f("Enter matrix elements:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            print f ("Enter element at position (%d, %d): ", i + 1,
j + 1);
            scan f ("%d", &matrix[i][j]);
        } } }
// Function to perform matrix multiplication
void multiply Matrices(int first
Matrix[MAX_SIZE][MAX_SIZE], int second
Matrix[MAX_SIZE][MAX_SIZE], int
result[MAX_SIZE][MAX_SIZE], int m, int n, int p) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            result[i][j] = 0;
            for (int k = 0; k < n; k++) {
                result[i][j] += first Matrix[i][k] * second
Matrix[k][j];
```

```
        }}  }}
// Function to print matrix elements
void print Matrix(int matrix[MAX_SIZE][MAX_SIZE], int
rows, int cols) {
    print f ("Matrix elements:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            print f ("%d\t", matrix[i][j]);
        }
        print f ("\n");
    } }
int main () {
    int m, n, p;
    // Input the size of matrices
    print f ("Enter the size of the first matrix (m x n): ");
    scan f ("%d %d", &m, &n);
    print f ("Enter the size of the second matrix (n x p): ");
    scan f ("%d %d", &n, &p);
    int first Matrix [MAX_SIZE][MAX_SIZE], second Matrix
[MAX_SIZE][MAX_SIZE], result[MAX_SIZE][MAX_SIZE];
    // Read elements for the first matrix
    read Matrix (first Matrix, m, n);
    // Read elements for the second matrix
    Read Matrix (second Matrix, n, p);
```

```
    // Perform matrix multiplication
    multiply Matrices (first Matrix, second Matrix, result, m, n, p);
    // Print the result matrix
    print Matrix (result, m, p);
    return 0;
}
```

This C program prompts the user to input the size of two matrices, then reads the elements for each matrix, performs matrix multiplication, and finally prints the result matrix.

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.