C GROUP PROJECT & ASSIGNMENT

———ASSIGNMENT———

Batch - 2023-27

|Section – I |

BCA Hons. (Artificial Intelligence & Data Science)

Submitted by HIMANSHU JAIN

SUBMITTED TO RISHI KUMAR Pooja CHAHAR

QUESTIONS;

Q1. What are;

- Constants and Variables
- Types of Constants
- Keywords
- Rules for Identifiers
- Int, Char, Double, Float, Long, Void.

Answer:

- Constants are values that remain fixed and unchanged throughout the program. They are like pieces of information that don't change, such as Mathematical constants like Pi.

  Variables are used to store data that can be changed during the program's execution. They can hold different values at different times.

  Types of constants:
- Numeric Constants: Constants that represents numeric values. For example, 'const int MAX_VALUE = 100;'.
  Character Constants: Single characters enclosed in single quotes, like 'const char NEW_LINE = '\n' ; '.
  String Constants: Sequences of characters enclosed in double quotes, like 'const char* GREETING = "Hello, World!";'.
  Boolean Constants: Representing true or false values, like 'const bool IS_ACTIVE = true;'.
  Floating Constants: Decimal numbers, like 'const float PI = 3.14;'.

- Keywords in C are reserved words that have special meanings and are used to define the syntax and structure of a C program. These words cannot be used as identifiers because they serve specific roles in the programming language. Keywords are essential for creating meaningful and well-structured C code.
  Some examples of C keywords include:
  int, char, float, if, else, for, while, return, void, struct, default, etc.

- Identifiers in C are names given to various program elements such as variables, functions, arrays, etc. Here are the rules for creating identifiers in C:
  1. Character Set: can consist of letters (both uppercase and lowercase), digits and underscore.4

2. Length limitation: there is no strict limit on the length of an identifier, but the first 31 characters must be unique.
3. Case sensitivity: C is case sensitive, so ‘count’, and ‘Count’ would be treated differently.
4. Reserved Words: Identifiers cannot be the same as C keywords (int, if, else, etc.) as these are reserved for specific purposes.

- int: declares the integer data type.
  Float: declares a floating-point data type.
  Char: declares a character type.
  Double: declares a double-precision floating-point data type.
  Long: declares a long integer data type.
  Void: specifies that a function returns no value or that a pointer has no type.

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.

Answer:

Arithmetic Operators: The operators used for basic mathematical operations in C, allowing you to perform addition, subtraction, multiplication, and division as well as handle incrementing and decrementing values.

Increment and Decrement: Increment and decrement are unary operators in programming, often used to increase and decrease the value of a variable by 1, respectively.

Increment (++): it adds 1 to the current value of a variable.

Decrement (--): it subtracts 1 from the current value of a variable.

Relational Operators: these operators in C are used to compare two values and determine the relationship between them. These operators return a Boolean result, indicating whether the specified relation is true or false. For example, equal too (==), not equal to (!=), <,>, <=, >=.

Logical Operators: These operators are used to perform logical operations on Boolean values. They allow you to combine or manipulate Boolean expressions. Some common logical operators are: logical AND (&&), logical OR (||), logical NOT (!).

Bitwise Operators: these operators are used to perform operations on individual bits of integer values. These operators manipulate the binary representations of numbers at the bit level. Some common bitwise operators are: Bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise NOT, left shift (<<), right shift (>>).

Conditional Operators: these operators also known as ternary operators that provide a concise way to express conditional statements in programming languages. :, ;

Type Conversions: also known as type casting, refer to the process of converting a value from one data type to another in a programming language. These may be implicit or explicit type conversions.

Expressions: An Expression in programming is a combination of values, variables, operators and function calls that can be evaluated to produce a result.

Associativity of Operations: associativity in programming languages refers to the order in which operators of the same precedence are evaluated in an expression. Most operators have left-to-right associativity, meaning they are evaluated from left to right when they appear in a sequence with the same precedence.

Precedence: In C and many other languages, operator precedence determines the order in which operators are evaluated in an expression. When an expression involves multiple operators, precedence rules define the grouping of the operators and their operands.

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

Answer:

1. **IF STATEMENT**

The ‘if’ statement is a conditional statement in C that allows you to execute a block of code based on a specified condition. The syntax of the ‘if’ statement is; *if (condition) {*

}

//if the code inside the braces is true it will be executed and if not, it will be skipped.

EXAMPLE:

```
#include <stdio.h>

int main() {

        int x = 10;  if (x>5) {            printf(“x

is greater than 5\n”);

        }

printf(“this statement is always executed\n”);
return 0;
}
```

2. **IF ELSE STATEMENT**

The ‘if else’ statement in C allows you to specify two blocks of code: one to be executed if a condition is true (if block) and another to be executed if the condition is false (else block).

EXAMPLE:

```
#include <stdio.h>

int main() {

int num = 10;
```

```
   // Check if num is even or odd
if (num % 2 == 0) {        printf("%d
is even.\n", num);
   } else {
      printf("%d is odd.\n", num);
   }
   return 0;
}
```

3. NESTED IF ELSE

A nested if else statement is an ‘if else’ statement that is placed inside another ‘if’ or ‘else’ block. This allows multiple levels of condition checking and execution of different code blocks based on various conditions.

EXAMPLE:

```
#include <stdio.h>

int main() {
   int x = 10;
int y = 5;

   if (x > 5) {
      printf("x is greater than 5.\n");

      if (y > 3) {
         printf("y is greater than 3.\n");
      } else {
         printf("y is not greater than 3.\n");
```

```
        }

    } else {
        printf("x is not greater than 5.\n");
    }

    return 0;
}
```

Q4. Explain Switch Case statement with example.

Answer:

The SWITCH statement in C provides a way to select one of many code blocks to be executed based on the value of an expression. It is a more concise alternative to a series of nested 'if else' statements.
EXAMPLE:

```
#include <stdio.h>

int main() {     char
grade = 'B';

   switch (grade) {
case 'A':
printf("Excellent!\n");
break;
```

```
      case 'B':
printf("Good job!\n");
break;

      case 'C':
printf("Passed.\n");
break;

      default:
         printf("Invalid grade.\n");
   }

   return 0;
}
```

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

Answer:

Loops in programming are structures that allow a certain block of code to be executed repeatedly, either for a specified number of times or until a certain condition is met. Loops are essential for automating repetitive tasks and are a fundamental concept in programming. In C, there are mainly three types of loops; FOR, WHILE, DO-WHILE.

1. **FOR LOOP**

The FOR loop is commonly used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and iteration.

Syntax:

For (condition) {

}

EXAMPLE:

```
#include <stdio.h>


int main() {
   for (int i = 1; i <= 5; i++) {
printf("%d ", i);
   }
   return 0;
}
```

2. WHILE LOOP

The WHILE loop is used when the number of iterations is not known beforehand, and it continues executing the block of code as long as the specified condition is true.

Syntax:

while (condition) {

}

EXAMPLE:

```
#include <stdio.h>
```

```
int main () {
        int i=1;
while (i<= 5) {
                printf(“%d”, i);
                i++;
        }
        return 0;
}
```

3. DO-WHILE:

The Do-while loop is similar to the ‘while’ loop, but it guarantees that the block of code is executed at least once, even if the condition is initially false. Syntax; *do {*

} while (condition)

EXAMPLE:

```
#include <stdio.h>


int main() {
   int i = 1;     do {
printf("%d ", i);
i++;
   } while (i <= 5);


   return 0;
}
```

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

Answer:

- Importance of Debugging:
  Debugging is crucial in software development for several reasons;

  1. Error Detection: identifies and fixes errors in codes.
  2. Correcting Logic Errors: Help find and resolve logical mistakes.
  3. Improving Code quality: encourages clean, maintainable codes.
  4. Optimizing performance: identifies and addresses performance issues.
  5. Enhancing user experience: Ensures a smoother, bug-free product.

- Tools for Debugging:

  1. **Integrated Development Environments (IDEs)**: IDEs like Visual studio, or eclipse provide built-in debugging tools. They offer features like breakpoints, variable inspection, and step-by-step execution.
  2. **Print Statements**: Printing variable values at different points in your code can help you understand the flow and identify the state of variables.
  3. **Assertions**: Adding assertions in your code to check specific conditions can help catch issues early during development.
  4. **Online Debugging Tools**: Platforms like JSFiddle (for javascript), Repl.it (for various languages) or online IDEs often provide debugging features for quick testing and debugging.

• Common Errors in Programming:

1. **Syntax Errors**: Errors related to the structure of the code, such as missing semicolons, parentheses, or mismatched brackets.
2. **Logical Errors**: Errors that occur when there's a flaw in the algorithm or logic of the code, leading to unexpected behavior.
3. **Runtime Errors**: Errors that occur during the execution of the program, such as dividing by zero or accessing an array out of the bounds.

- SYNTAX for DEBUGGING:

In C, debugging is often done using a debugger tool. One of the commonly used debugger tools is **gdb** (GNU Debugger). Here are the basic steps and commands for using **gdb** for debugging in C:

- Compile with Debug Information:

```
gcc -g -o my_program my_program.c.
```

Debugging is an iterative process, and it requires patience and systematic analysis. The goal is to understand the code's behavior, identify the root cause of the issue, and implement a reliable fix.

- Testing of C programs:

Testing a C program after debugging is a crucial step to ensures that your code works correctly and produces the expected results. The steps are, UNIT testing, INTEGRATION testing, TEST with boundary values, POSITIVE and NEGATIVE testing, AUTOMATED testing, TEST INPUT/OUTPUT streams, MEMORY management, and STRESS testing.

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

Answer:

In C programming, both user-defined functions and predefined (or standard) functions play important roles in the development of programs.

1. User-Defined Functions:

These are functions that you define yourself in your C program.

User-Defined functions allow you to break down your program into smaller, more manageable pieces. Each function performs a specific task, making your code modular and easier to understand.
We declare a user-defined function by providing its prototype before the main function, and then you define its behavior later in the code.

```
int addNumbers(int a, int b);
```

2. Predefined (standard) Functions:

These are the functions provided by the C standard library.

Their main purpose is to offer common operations and functionalities. It's included through header files ("#include <stdio.h>", etc)

Examples: printf, scanf, sqrt, etc.

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 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 Arguments to a Function:

- User defined functions:

Definition: when calling a user-defined function, you pass values(arguments) to the function.
Syntax: define the function to accept parameters, and when calling it, provide values for those parameters.

- Pre-defined functions:
  Definition: these are the function provided by the C standard library.
  Purpose: offer common operations and functionalities.
  Usage: included through header files (#include <stdio.h>, etc)
  Examples: printf, scanf. Sqrt, etc.
  In both cases, passing arguments and returning values enhance the flexibility and usefulness of functions in a C program. The values passed and returned allow for dynamic and efficient code execution.

2) Storage Classes, Automatic, Static, Register, External.

In C programming, storage classes are used to define the scope(visibility) and lifetime(duration) of variables. There are four main storage classes: automatic, static, register, and external.

1. Automatic Storage Class:

- Keyword: 'auto'
- Scope: Limited to the block or function where it is declared.
- Lifetime: created when the block is entered and destroyed when the block is exited.
- Example:

```
void exampleFunction() {
auto int localVar = 10;
}
```

2. Static Storage Class:

- Keyword: 'static'
- Scope: limited to the block or function where it is declared.
- Lifetime: persists throughout the program's execution.

- Example: void exampleFunction() {   static int staticVar = 5;
  }

3. Register Storage Class:

- Keyword: ‘register’
- Scope: limited to the block or function where it is declared.
- Lifetime: similar to automatic variables but stored in CPU registers for faster access.
- Example: void exampleFunction() { register int regVar = 3;
  }

4. External Storage class:

- Keyword: ‘extern’
- Scope: global scope, visible to all functions in all files.
- Lifetime: persists throughout the program’s execution.
- Example: int globalVar = 100; extern int globalVar;

these storage classes help control the visibility, accessibility, and lifetime of variables in a C program providing flexibility in managing memory and program structure.

3) #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);


   printf("\nConcatenated Output of S1 and S2: %s%s\n", s1, s2);



   int lenS1 = strlen(s1);
int spacesS1 = 0;
```

```
    for (int i = 0; i < lenS1; i++) {
if (s1[i] == ' ') {
spacesS1++;
        }
    }



    int lenS2 = strlen(s2);
int spacesS2 = 0;


    for (int i = 0; i < lenS2; i++) {
if (s2[i] == ' ') {
spacesS2++;
        }
    }

    printf("\nNumber of characters in S1: %d\n", lenS1);
printf("Number of empty spaces in S1: %d\n", spacesS1);

    printf("\nNumber of characters in S2: %d\n", lenS2);
    printf("Number of empty spaces in S2: %d\n", spacesS2);

    return 0;
}
```

Q9. Explain with example ID 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:

ID ARRAY:

An ID array is typically used to store identification numbers.

Example:

```
#include <stdio.h> int
main() {
   // Example of an ID array     int
idArray[5] = {101, 102, 103, 104, 105};
for (int i = 0; i < 5; i++) {        printf("ID[%d]:
%d\n", i, idArray[i]);
    }
    return 0;
}
```

MULTIDIMENSIONAL ARRAY:

A multidimensional array is useful for representing matrices.

```
#include <stdio.h>


#define MAX 10

void readMatrix(int m, int n, int matrix[MAX][MAX])
{     printf("Enter the matrix elements:\n");     for (int i
= 0; i < m; i++) {         for (int j = 0; j < n; j++) {
printf("matrix[%d][%d]: ", i, j);             scanf("%d",
&matrix[i][j]);
        }
    }
}


void matrixMultiply(int m1, int n1, int matrix1[MAX][MAX], int m2, int n2, int
matrix2[MAX][MAX], int result[MAX][MAX]) {
    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];
            }
        }
    }
}
```

```
void printMatrix(int m, int n, int matrix[MAX][MAX]) {
printf("Matrix elements:\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;     int matrix1[MAX][MAX],
matrix2[MAX][MAX], result[MAX][MAX];

    printf("Enter the dimensions of matrix1 (m1 n1): ");
scanf("%d %d", &m1, &n1);     readMatrix(m1, n1,
matrix1);

    printf("Enter the dimensions of matrix2 (m2 n2): ");
scanf("%d %d", &m2, &n2);     readMatrix(m2, n2,
matrix2);



    if (n1 != m2) {         printf("Matrix multiplication is
not possible.\n");         return 1;
    }
    matrixMultiply(m1, n1, matrix1, m2, n2, matrix2, result);

    printf("Result of matrix multiplication:\n");
printMatrix(m1, n2, result);
```

```
    return 0;
}
```

This program includes three functions:

1. ‘readmatrix’: reads matrix elements from the user.
2. ‘matrixmultiply’: performs matrix multiplication.
3. ‘printmatrix’: prints matrix elements.
   The ‘main’ function takes input takes matrices from the user, performs matrix multiplication, and prints the result. The matrices are defined using a multidimensional array.

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.

1. Structure declaration and initialization:
   A structure is a user-defined data type in C that allows you to group variables of different data types under a single name.

```
#include <stdio.h>

struct BankAccount {     int
accountNumber;     char
accountHolderName[50];     float
balance;
};
```

```
int main() {
   // Structure initialization     struct BankAccount
account1 = {12345, "John Doe",
1000.0};

   printf("Account Number: %d\n",
account1.accountNumber);     printf("Account Holder
Name: %s\n", account1.accountHolderName);
printf("Balance: $%.2f\n", account1.balance);     return
0;
}
```

2. Array of structures:
   You can create an array of structures to store multiple records of the same type.

   Example: #include
```
<stdio.h> struct
BankAccount {
   int accountNumber;     char
accountHolderName[50];     float
balance;
};

int main() {
   // Array of structures
   struct BankAccount accounts[3] = {
      {12345, "John Doe", 1000.0},
      {67890, "Jane Smith", 500.0},
      {13579, "Bob Johnson", 1200.0}
   };

   for (int i = 0; i < 3; i++) {
      printf("Account Number: %d\n",
accounts[i].accountNumber);        printf("Account
Holder Name: %s\n",
accounts[i].accountHolderName);
printf("Balance: $%.2f\n", accounts[i].balance);
printf("\n");
    }
    return 0;
}
```

3.
Use of Typedef:

‘typedef’ is used to create aliases for data types, providing a way to simplify complex declarations.

Example:

```
#include <stdio.h>


typedef struct {     int
accountNumber;     char
accountHolderName[50];
   float balance;
} BankAccount;


int main() {
   // Using the typedef for structure
   BankAccount account = {12345, "John Doe", 1000.0};


   printf("Account Number: %d\n",
account.accountNumber);
   printf("Account Holder Name: %s\n",
account.accountHolderName);
   printf("Balance: $%.2f\n", account.balance);


   return 0;
}
```

4.

Passing Structures to functions:
You can pass structures to functions by value or by reference.
<u>Example</u>:

```
#include <stdio.h>

struct BankAccount {     int
accountNumber;     char
accountHolderName[50];     float
balance;
};

// Function to display account details void
displayAccount(struct BankAccount acc) {
printf("Account Number: %d\n",
acc.accountNumber);     printf("Account
Holder Name: %s\n",
acc.accountHolderName);
   printf("Balance: $%.2f\n", acc.balance);
}

int main() {
   // Structure initialization     struct BankAccount
account = {12345, "John Doe",
1000.0};

   // Passing structure to function
displayAccount(account);

   return 0;
}
```

5.

Union declaration and initialization:
A union is a smaller to be a structure, but it allows different variables to share the same memory space.

Example:
```
#include <stdio.h>

// Union declaration union
MyUnion {
   int intValue;     float
floatValue;     char
stringValue[20];
};

int main() {
   // Union initialization
   union MyUnion myUnion = {123};

   // Accessing union members
   printf("Int Value: %d\n", myUnion.intValue);

   // Modifying the union     myUnion.floatValue =
456.789;     printf("Float Value: %.3f\n",
myUnion.floatValue);

   return 0;
}
```

6. Differences between structure and union: Structure: Members have their own separate memory locations. Union: all members share the same memory location.

6.

7. Bank account program:
   Here's a program that reads details of 10 people's bank accounts and displays the record with the highest bank balance;

```
#include <stdio.h>

struct BankAccount {     int
accountNumber;     char
accountHolderName[50];     float
balance;
};

int main() {     struct BankAccount
accounts[10];

   for (int i = 0; i < 10; i++) {         printf("Enter details
for account %d:\n", i + 1);         printf("Account
Number: ");         scanf("%d",
&accounts[i].accountNumber);
printf("Account Holder Name: ");         scanf("%s",
accounts[i].accountHolderName);
printf("Balance: $");         scanf("%f",
&accounts[i].balance);         printf("\n");
   }
   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 Name: %s\n",
accounts[maxIndex].accountHolderName);
   printf("Balance: $%.2f\n",
accounts[maxIndex].balance);

   return 0;
}
```

This program uses an array of structures to store details for 10 people’s bank accounts. It then finds and displays the record with the highest bank balance.

END

HIMANSHU JAIN
23091073
BCA hons. With Artificial Intelligence and Data Science.