C’s Assignment

Name-Krishna Pant

Student Id-231601093

Class roll no.-49

Submitted to-Rishi Kumar

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

Ans .Constants: Constants are values that remain unchanged during the execution of a program. They are also called literals. Examples include numeric constants like 1, 3.14, and character constants like 'a'.

Variables: Variables are memory locations that hold values. These values can change during the execution of a program. For example, int x; declares an integer variable named x.

Types of Constants:

Integer Constants: Whole numbers without any decimal point (e.g., 5, -3, 0).

Real or Floating-point Constants: Numbers with a decimal point or in exponential form (e.g., 3.14, -0.001, 2.5e3).

Character Constants: Single characters enclosed in single quotes (e.g., 'a', '5', '$').

String Constants: A sequence of characters enclosed in double quotes (e.g., "Hello, World!").

Symbolic Constants: Represented by #define in C/C++. These are user-defined constants.

Keywords:

Keywords are reserved words in a programming language that have special meanings. Examples in C include int, float, if, for, etc. You cannot use these words as identifiers (names for variables, functions, etc.) in your program.

Rules for Identifiers:

Must begin with a letter (uppercase or lowercase) or an underscore (_).

The following characters can be letters, digits, or underscores.

No special characters like @, $, and % are allowed.

Keywords are not allowed as identifiers.

Data Types:

int: Integer data type, used for whole numbers (e.g., 5, -10).

float: Floating-point data type, used for numbers with a decimal point (e.g., 3.14, -0.001).

char: Character data type, used for single characters (e.g., 'a', '5').

double: Double-precision floating-point data type, allows for more decimal places than float.

long: Used to declare long integers.

void: Represents the absence of a type or the lack of a value. It's often used as the return type of functions that do not return any value.

These data types help in defining the type of data a variable can store, and they determine the size and format of the stored data.

Q2. Explain with examples Arithmetic Operators, Increment and Decrement Operators,

Relational Operators, Logical Operators, Bitwise Operators, Conditional Operators, Type

Ans

Arithematic operators;

Arithmetic operators perform mathematical operations on operands.

```
int a = 10, b = 5;

int sum = a + b; // Addition

int difference = a - b; // Subtraction

int product = a * b; // Multiplication

int quotient = a / b; // Division

int remainder = a % b; // Modulus (remainder after division)
```

Increment and Decrement Operators:

Increment and decrement operators are used to increase or decrease the value of a variable by 1

. int x = 5;

x++; // Increment x by 1

int y = 10;

y--; // Decrement y by 1

Reltional operator

Relational operators are used to compare two values.

int p = 10, q = 20;

printf("%d\n", p == q); // Equal to

printf("%d\n", p != q); // Not equal to

printf("%d\n", p < q); // Less than

printf("%d\n", p > q); // Greater than

printf("%d\n", p <= q); // Less than or equal to

printf("%d\n", p >= q); // Greater than or equal to

Logical Operator

Logical operators perform logical operations on boolean values (0 or 1).

int x = 1, y = 0;

printf("%d\n", x && y); // Logical AND

printf("%d\n", x || y); // Logical OR

printf("%d\n", !x); // Logical NOT

Bitwise Operator

Bitwise operators perform operations on individual bits of integers.

```
int a = 5, b = 3;

printf("%d\n", a & b); // Bitwise AND

printf("%d\n", a | b); // Bitwise OR

printf("%d\n", a ^ b); // Bitwise XOR

printf("%d\n", ~a); // Bitwise NOT

printf("%d\n", a << 1); // Left shift by 1

printf("%d\n", a >> 1); // Right shift by 1
```

Conditionary Operator

The cint x = 10, y = 20;

int max = (x > y) ? x : y; // If x > y, assign x to max; otherwise, assign yonditional operator is a shorthand way of writing an if-else statement.

Types ,Conversion and Expression

Type conversions can be implicit (automatic) or explicit (casting).

```
int a = 5;

float b = 2.5;

float result = a + b; // Implicit type conversion (promotion)

int integerResult = (int)(a + b); // Explicit type conversion (casting)
```

Precedence and Associativity Operator

Operator precedence determines the order in which operators are evaluated.

```
int result = 5 * 3 + 2; // Multiplication has higher precedence
```

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

1. if Statement:

The if statement allows you to execute a block of code only if a specified condition is true.

```
#include <stdio.h>



int main() {

int x = 10;



if (x > 5) {

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

}



return 0;

}
```

2. if-else Statement:

The if-else statement allows you to execute one block of code if a condition is true and another block if it is false.

```
#include <stdio.h>



int main() {

int x = 3;
```

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


return 0;
}
```

2. else if (or elseif) Statement:

The else if statement allows you to check multiple conditions in sequence.

```
#include <stdio.h>


int main() {
int x = 3;


if (x > 5) {
printf("x is greater than 5.\n");
} else if (x == 5) {
printf("x is equal to 5.\n");
} else {
printf("x is less than 5.\n");
```

```
}


return 0;
}
```

4. Nested if-else Statements:

You can nest if-else statements inside each other to handle more complex conditions.

```
#include <stdio.h>


int main() {
int x = 3, y = 7;


if (x > 0) {
if (y > 0) {
printf("Both x and y are positive.\n");
} else {
printf("x is positive, but y is not.\n");
}
} else {
printf("x is not positive.\n");
}


return 0;
```

}

Q4. Explain Switch Case statement with example.

ans The switch statement is a control flow statement in many programming languages that allows you to select one of many code blocks to be executed. It is particularly useful when you have a variable or an expression that can take multiple values, and you want to perform different actions based on the value of that variable or expression.

Here's the basic syntax of the switch statement:

```
switch (expression) {

case value1:

// code to be executed if expression == value1

break;

case value2:

// code to be executed if expression == value2

break;

// additional cases as needed

default:

// code to be executed if none of the cases match

}
```

And here's an example to illustrate how the switch statement works:

```
#include <stdio.h>

int main() {
int day = 3;

switch (day) {
case 1:
printf("Monday\n");
break;
case 2:
printf("Tuesday\n");
break;
case 3:
printf("Wednesday\n");
break;
case 4:
printf("Thursday\n");
break;
case 5:
printf("Friday\n");
break;
```

```
case 6:

printf("Saturday\n");

break;

case 7:

printf("Sunday\n");

break;

default:

printf("Invalid day\n");

}



return 0;

}
```

In this example, the switch statement evaluates the value of the variable day. Depending on the value of day, the corresponding block of code is executed. If none of the cases match, the default block is executed. In this case, if day is 3, it will print "Wednesday."

It's important to note that each case block should end with a break statement. The break statement is used to exit the switch statement once a match is found. If you forget to include a break, execution will continue into the next case block, leading to unexpected behavior.

Loops are control flow structures that allow you to repeatedly execute a block of code. They are essential for automating repetitive tasks and iterating over data structures. In C, there are three types of loops: for, while, and do-while.

1. for Loop:

The for loop is used when the number of iterations is known before entering the loop.

c

Copy code

```
#include <stdio.h>



int main() {

// Print numbers 1 to 5 using a for loop

for (int i = 1; i <= 5; i++) {

printf("%d ", i);

}



return 0;

}
```

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

Ans This example prints numbers 1 to 5 using a for loop. The loop has three parts: initialization (int i = 1), condition (i <= 5), and update (i++). The loop will continue executing as long as the condition is true.

2. while Loop:

The while loop is used when the number of iterations is not known beforehand, and the loop continues as long as a specified condition is true.

```
#include <stdio.h>


int main() {

// Print numbers 1 to 5 using a while loop

int i = 1;

while (i <= 5) {

printf("%d ", i);

i++;

}


return 0;

}
```

This example achieves the same result as the for loop but using a while loop. The loop will continue as long as the condition i <= 5 is true.

3. do-while Loop:

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

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

```
int main() {

// Print numbers 1 to 5 using a do-while loop

int i = 1;

do {

printf("%d ", i);

i++;

} while (i <= 5);



return 0;

}
```

This example also prints numbers 1 to 5, and even if the condition i <= 5 is false initially, the loop body will execute at least once.

In summary:

Use a for loop when you know the number of iterations beforehand.

Use a while loop when the number of iterations is not known beforehand.

Use a do-while loop when you want to ensure the loop body executes at least once.

Q6. Explain with examples debugging importance, tools common errors: syntax, logic, and

runtime errors, debugging, and Testing C Programs

ans The switch statement is a control flow statement in many programming languages that allows you to select one of many code blocks to be executed. It is particularly useful when you have a variable or an expression that can take multiple values, and you want to perform different actions based on the value of that variable or expression.

Here's the basic syntax of the switch statement:

```
switch (expression) {
case value1:
// code to be executed if expression == value1
break;
case value2:
// code to be executed if expression == value2
break;
// additional cases as needed
default:
// code to be executed if none of the cases match
}
```

And here's an example to illustrate how the switch statement works:

```
#include <stdio.h>

int main() {
int day = 3;

switch (day) {
```

```
case 1:
printf("Monday\n");
break;
case 2:
printf("Tuesday\n");
break;
case 3:
printf("Wednesday\n");
break;
case 4:
printf("Thursday\n");
break;
case 5:
printf("Friday\n");
break;
case 6:
printf("Saturday\n");
break;
case 7:
printf("Sunday\n");
break;
default:
printf("Invalid day\n");
}
```

```
return 0;

}
```

In this example, the switch statement evaluates the value of the variable day. Depending on the value of day, the corresponding block of code is executed. If none of the cases match, the default block is executed. In this case, if day is 3, it will print "Wednesday."

It's important to note that each case block should end with a break statement. The break statement is used to exit the switch statement once a match is found. If you forget to include a break, execution will continue into the next case block, leading to unexpected behavior.

The default case is optional but can be useful for handling values not covered by the specific case blocks.

User

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

ChatGPT

Loops are control flow structures that allow you to repeatedly execute a block of code. They are essential for automating repetitive tasks and iterating over data structures. In C, there are three types of loops: for, while, and do-while.

1. for Loop:

The for loop is used when the number of iterations is known before entering the loop.

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

```
int main() {

// Print numbers 1 to 5 using a for loop

for (int i = 1; i <= 5; i++) {

printf("%d ", i);

}



return 0;

}
```

This example prints numbers 1 to 5 using a for loop. The loop has three parts: initialization (int i = 1), condition (i <= 5), and update (i++). The loop will continue executing as long as the condition is true.

2. while Loop:

The while loop is used when the number of iterations is not known beforehand, and the loop continues as long as a specified condition is true.

```
#include <stdio.h>



int main() {

// Print numbers 1 to 5 using a while loop

int i = 1;
```

```
while (i <= 5) {

printf("%d ", i);

i++;

}



return 0;

}
```

This example achieves the same result as the for loop but using a while loop. The loop will continue as long as the condition i <= 5 is true.

3. do-while Loop:

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

```
#include <stdio.h>

int main() {

// Print numbers 1 to 5 using a do-while loop

int i = 1;

do {

printf("%d ", i);

i++;

} while (i <= 5);
```

```
return 0;

}
```

This example also prints numbers 1 to 5, and even if the condition i <= 5 is false initially, the loop body will execute at least once.

In summary:

Use a for loop when you know the number of iterations beforehand.

Use a while loop when the number of iterations is not known beforehand.

Use a do-while loop when you want to ensure the loop body executes at least once.

User

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

Importance of Debugging:

Debugging is a crucial part of the software development process. It involves identifying and fixing errors or bugs in a program. Here are some reasons why debugging is important:

Correctness: Debugging ensures that a program behaves as intended and produces the correct output.

Efficiency: Debugging helps optimize code by identifying and removing unnecessary or inefficient code.

Quality Assurance: Debugging improves the overall quality of a program, making it more reliable and robust.

User Experience: Debugging contributes to a positive user experience by eliminating unexpected behavior and crashes.

Common Types of Errors:

Syntax Errors:

Occur when the code violates the rules of the programming language.

Examples: missing semicolons, mismatched parentheses, undefined variables.

Logic Errors:

Occur when the code is syntactically correct but does not produce the intended result.

Examples: incorrect algorithms, wrong conditions in if statements.

Runtime Errors:

Occur during the execution of a program.

Examples: division by zero, accessing an array out of bounds.

Debugging Tools:

Print Statements:

Inserting print statements in the code to output variable values at specific points.

Debugger:

Integrated development environments (IDEs) often provide debugging tools that allow you to set breakpoints, inspect variables, and step through code.

Static Analysis Tools:

Tools that analyze the code without executing it to find potential issues. Examples include linters and code analyzers.

Profiling Tools:

Tools that analyze the program's runtime behavior, helping to identify performance bottlenecks.

Debugging C Programs:

Example 1: Syntax Error

```
#include <stdio.h>

int main() {
int x = 5
printf("Value of x: %d\n", x);
return 0;
}
```

Error: Missing semicolon after int x = 5.

Example 2: Logic Error

```
#include <stdio.h>


int main() {
int x = 5;
int y = 0;
int result = x / y;
printf("Result: %d\n", result);
return 0;
}
```

Error: Division by zero.

Example 3: Runtime Error

```
#include <stdio.h>


int main() {
int arr[3] = {1, 2, 3};
printf("%d\n", arr[5]);
return 0;
}
```

Error: Accessing an array out of bounds.

Testing C Programs:

1 Unit Testing:

Testing individual functions or modules in isolation to ensure they produce the correct output.

2 Integration Testing

Testing the interactions between different modules or components of a program.

3 System Testing:

Testing the entire system to ensure it meets the specified requirements.

4 Regression Testing:

Testing after making changes to the code to ensure that existing functionality still works.

5Automated Testing:

Using testing frameworks and tools to automate the testing process.

6 Boundary Value Testing:

Testing with inputs at the boundaries of valid and invalid ranges.

7 Stress Testing:

Testing the program's performance under extreme conditions.

Debugging and testing go hand in hand to ensure the reliability and correctness of a C program. Proper testing practices help catch errors before deployment, while debugging tools aid in identifying and fixing issues during development.

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

and call by reference

ans

User-Defined Functions:

User-defined functions are functions that are created by the programmer to perform a specific task within a program. These functions allow for better code organization, reusability, and abstraction of complex tasks. Here's an example of a user-defined function:

```
#include <stdio.h>


// User-defined function to calculate the square of a number
int square(int x) {
return x * x;
}


int main() {
int num = 5;
int result = square(num);
printf("Square of %d is %d\n", num, result);
return 0;
}
```

In this example, square is a user-defined function that takes an integer x as a parameter and returns the square of x. The main function then calls this function with the value 5, and the result is printed.

Pre-defined Functions:

Pre-defined functions, also known as built-in functions, are functions that are provided by the programming language or its libraries. These functions perform common tasks and are readily available for use. In C, functions like printf, scanf, strlen, and sqrt are examples of pre-defined functions. Here's an example using printf:

```
#include <stdio.h>


int main() {

int num = 10;

// Using the pre-defined printf function

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

return 0;

}
```

Call by Value and Call by Reference:

Call by Value:

In call by value, the actual value of the argument is passed to the function. Any modifications made to the parameter inside the function do not affect the original argument.

```
#include <stdio.h>


void increment(int x) {

x++; // Incrementing the local copy of x

printf("Inside function: %d\n", x);

}
```

```
int main() {

int num = 5;

increment(num);

printf("Outside function: %d\n", num);

return 0;

}
```

In this example, the value of num inside the increment function is modified, but this modification does not affect the original num in the main function.

Call by Reference:

In call by reference, the memory address (reference) of the argument is passed to the function. This allows the function to directly modify the original value.

```
#include <stdio.h>


void incrementByReference(int *x) {

(*x)++; // Incrementing the value at the memory address pointed by x

printf("Inside function: %d\n", *x);

}


int main() {

int num = 5;
```

```
incrementByReference(&num); // Passing the address of num

printf("Outside function: %d\n", num);

return 0;

}
```

In this example, the value of num inside the main function is modified because the function incrementByReference receives the memory address of num. The *x notation is used to access the value at that memory address.

Understanding the difference between call by value and call by reference is important when designing functions and working with parameters in C. It affects how changes to parameters inside a function impact the original values.

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

Ans

1) Passing and Returning Arguments to and from Function:

Passing Arguments to a Function:

Arguments can be passed to a function in C using parameters. There are two ways to pass arguments: pass by value and pass by reference.

Pass by Value

```
#include <stdio.h>


void add(int a, int b) {
int sum = a + b;
printf("Sum: %d\n", sum);
}


int main() {
int x = 5, y = 7;
add(x, y);
return 0;
}
```

In this example, values of x and y are passed to the add function by value.

Pass by Reference:

```
#include <stdio.h>


void incrementByReference(int *x) {
(*x)++;
printf("Incremented value: %d\n", *x);
}
```

```
int main() {

int num = 10;

incrementByReference(&num);

return 0;

}
```

In this example, the address of num is passed to the incrementByReference function, allowing it to modify the original value.

Returning Values from a Function:

A function can return a value using the return statement.

```
#include <stdio.h>


int add(int a, int b) {

return a + b;

}


int main() {

int result = add(3, 4);

printf("Result: %d\n", result);

return 0;

}
```

Here, the add function returns the sum of two numbers, and the result is printed in the main function.

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

Automatic Storage Class:

Variables with automatic storage duration are created when a block or function is entered and destroyed when it is exited. They are declared using the auto keyword, but it is rarely used explicitly because, by default, local variables are automatic.

```
#include <stdio.h>


int main() {
auto int x = 10; // The 'auto' keyword is optional here
printf("%d\n", x);
return 0;
}
```

Static Storage Class:

Variables with static storage duration are created once and persist throughout the program's execution. They are declared using the static keyword.

```
#include <stdio.h>



void counter() {

static int count = 0;

count++;

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

}



int main() {

counter();

counter();

return 0;

}
```

In this example, the count variable retains its value between function calls due to the static storage class.

Register Storage Class:

Variables with the register storage class are stored in the CPU registers, allowing for faster access. However, the use of register is a suggestion to the compiler, and it may ignore the request.

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

```
int main() {

register int x = 5;

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

return 0;

}
```

External Storage Class:

Variables with external storage duration are shared across multiple files in a program. They are declared using the extern keyword.

c

Copy code

```
// File: file1.c

int globalVar = 10;
```

```
// File: file2.c

#include <stdio.h>


extern int globalVar;


int main() {

printf("%d\n", globalVar);
```

```
return 0;

}
```

In this example, globalVar is declared in one file and used in another file using the extern keyword.

3) Concatenation of Two Strings Program:

```
#include <stdio.h>

#include <string.h>


int main() {

char s1[50], s2[50];


// Input two strings

printf("Enter the first string: ");

scanf("%s", s1);


printf("Enter the second string: ");

scanf("%s", s2);


// Concatenate the strings

strcat(s1, s2);
```

```
// Display the concatenated string

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



return 0;

}
```

In this program, the user inputs two strings (s1 and s2), and the strcat function is used to concatenate them.

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.The result is then displayed.

Ans In this example, we'll have three functions: one for reading matrix elements, one for performing matrix multiplication, and one for printing matrix elements.

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

```
// Function to read matrix elements

void readMatrix(int matrix[][10], int rows, int columns) {

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

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

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

printf("Enter element at position [%d][%d]: ", i + 1, j + 1);

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

}

}

}



// Function to perform matrix multiplication

void multiplyMatrices(int firstMatrix[][10], int secondMatrix[][10], int resultMatrix[][10], int m, int n, int
p) {

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

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

resultMatrix[i][j] = 0;

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

resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];

}

}

}

}
```

```
// Function to print matrix elements
void printMatrix(int matrix[][10], int rows, int columns) {
printf("Matrix elements:\n");
for (int i = 0; i < rows; i++) {
for (int j = 0; j < columns; j++) {
printf("%d ", matrix[i][j]);
}
printf("\n");
}
}

int main() {
int m, n, p;

// Input the size of the matrices
printf("Enter the number of rows for the first matrix: ");
scanf("%d", &m);
printf("Enter the number of columns for the first matrix and rows for the second matrix: ");
scanf("%d", &n);
printf("Enter the number of columns for the second matrix: ");
scanf("%d", &p);

int firstMatrix[10][10], secondMatrix[10][10], resultMatrix[10][10];
```

```
// Read elements for the first matrix

readMatrix(firstMatrix, m, n);



// Read elements for the second matrix

readMatrix(secondMatrix, n, p);



// Perform matrix multiplication

multiplyMatrices(firstMatrix, secondMatrix, resultMatrix, m, n, p);



// Print the result matrix

printMatrix(resultMatrix, m, p);



return 0;

}
```

In this program, the readMatrix function is used to read elements for both matrices, the multiplyMatrices function performs matrix multiplication, and the printMatrix function prints the elements of the resulting matrix. The sizes of the matrices are taken as input from the user.