C PROGRAMMING ASSIGNMENT

GEU

BATCH

2023-2026

BCA(AI&DS)

SUBMITTED BY - **AADARSH CHAUDHARY**

SUBMITTED TO - **MR. RISHI KUMAR**

STUDENT ID - **23151260**

ROLL NO. - **1**

SECTION - **I**

ASSISTANT PROFESSOR - **MS. POOJA CHAHAR**

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

Ans.

Constants: *In C programming, constants are fixed values that do not change during the execution of a program. They can be of various types, including integer constants, floating-point constants, character constants, and string constants.*

Variables: *Variables in C are named storage locations that hold values, and these values can be changed during program execution. Variables must be declared with a specific data type.*

Types of Constants:

Integer Constants: *Like 1, -21, and 100.*

Floating point Constants: *Like 3.14, -0.5, and 2.0.*

Character Constants: *Enclosed in single quotes, e.g., 'A' or '1'.*

String Constants*: Enclosed in double quotes, e.g., "Hello".*

Keywords: *Keywords in C are reserved words that have special meanings. They cannot be used as identifiers. Examples include int, float, if, else, and void.*

Rules for Identifiers:

1. *Must begin with a letter (uppercase or lowercase) or an underscore _.*
2. *After the first letter, digits (0-9) may also be included.*
3. *No spaces or special characters are allowed, except for underscore _.*
4. *Keywords cannot be used as identifiers.*

Data Types:

int: *Represents integer data type.*

float: *Represents floating-point data type.*

char: *Represents character data type.*

double: *Represents double-precision floating-point data type.*

long: *Represents a long integer data type.*

void: *Used as a return type in functions where the function does not return any value. It can also indicate that a function takes no arguments.*

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.

Ans.

1. **Arithmetic Operators:**

Arithmetic operators perform basic mathematical operations on operands.

- **Addition (+):** *Adds two operands (a + b)*
- **Subtraction (-):** *Subtracts the right operand from the left operand (a – b)*
- **Multiplication (*):** *Multiplies two operands (a * b)*
- **Division (/):** *Divides the left operand by the right operand (a / b)*
- **Modulus (%):** *Returns the remainder when the left operand is divided by the right operand (a % b)*

2. **Increment and Decrement Operators:**

These operators are used to increase or decrease the value of a variable.

- **Increment (++):** *Increases the value of the operand by 1. Example: a++ or ++a.*
- **Decrement (--):** *Decreases the value of the operand by 1. Example: a-- or --a.*

3. Relational Operators:

Relational operators compare two values and return either true (1) or false (0).

- **Equal to (==):** *Checks if two operands are equal. Example: a == b.*
- **Not equal to (!=):** *Checks if two operands are not equal. Example: a != b.*
- **Greater than (>):** *Checks if the left operand is greater than the right operand. Example: a > b.*
- **Less than (<):** *Checks if the left operand is less than the right operand. Example: a < b.*
- **Greater than or equal to (>=):** *Checks if the left operand is greater than or equal to the right operand. Example: a >= b.*
- **Less than or equal to (<=):** *Checks if the left operand is less than or equal to the right operand. Example: a <= b.*

4.Logical Operators:

Logical operators perform logical operations.

- **Logical AND (&&):** *Returns true if both operands are true. Example: a && b.*
- **Logical OR (||):** *Returns true if at least one operand is true. Example: a || b.*
- **Logical NOT (!):** *Returns true if the operand is false, and vice versa. Example: !a.*

*5.*Bitwise Operators:

Bitwise operators perform operations on individual bits of integer operands.

- **Bitwise AND (&):** *Performs a bitwise AND operation. Example: a & b.*
- **Bitwise OR (|):** *Performs a bitwise OR operation. Example: a | b.*
- **Bitwise XOR (^):** *Performs a bitwise exclusive OR operation. Example: a ^ b.*
- **Bitwise NOT (~):** *Flips the bits of the operand. Example: ~a.*
- **Left Shift (<<):** *Shifts the bits of the left operand to the left by the number of positions specified by the right operand. Example: a << b.*
- **Right Shift (>>):** *Shifts the bits of the left operand to the right by the number of positions specified by the right operand. Example: a >> b.*

6. Conditional Operator:

The conditional operator (? :) is a ternary operator that evaluates a condition and returns one of two values based on whether the condition is true or false. ***Copy code*** *result = (condition) ? value_if_true : value_if_false;*

Type Conversions and Expressions:

C supports automatic type conversion (coercion) in expressions, but explicit type casting can also be done.

Implicit Type Conversion: *Happens automatically by the compiler.*

Explicit Type Conversion: *Involves using type casting operators like (int), (float), etc.*

Precedence and Associativity of Operators:

Operators in C have different levels of precedence, and when multiple operators appear in an expression, the one with higher precedence is evaluated first. Associativity determines the order in which operators of equal precedence are evaluated.

For example, in the expression a + b * c, the multiplication has higher precedence, so b * c is evaluated first.

Q3. Explain with Example conditional statements if, ifelse, else if, nested if else.

Ans.

Conditional statements are used to control the flow of the program based on certain conditions. Here are examples of different types of conditional statements: if, if-else, else if (often written as elseif), and nested if-else.

1. **if statement:** *The basic if statement checks a condition and executes a block of code if the condition is true.*

2. **if-else statement:** *The if-else statement allows you to specify two blocks of code—one to be executed if the condition is true, and another to be executed if the condition is false.*

3. **else if (or elseif) statement:** *The else if statement is used when you have multiple conditions to check. It allows*

you to check additional conditions if the preceding ones are false.

4. **Nested if-else statement:** *Nested if-else statements occur when you have an if-else statement inside another if or else block.*

Q4. Explain Switch Case statement with example.

Ans.

The switch statement in C is a control flow statement that allows you to select one case from a list of possible cases based on the value of an expression. It's a concise way to handle multiple conditions.

- *The user is prompted to enter a number (choice).*
- *The switch statement is used to evaluate the value of choice.*
- *Cases are defined using the case keyword, followed by the constant value to be compared with the switch expression.*
- *The break statement is used to exit the switch block once a case is matched. If break is omitted, execution will "fall through" to the next case.*
- *The default case is optional and is executed when none of the cases match the switch expression.*

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

Ans.

In C programming, loops are used to repeat a block of code until a certain condition is met. There are three main types of loops: for, while, and do-while

1. **for loop:** *The for loop is used when you know in advance how many times you want to execute the block of code.*

2. **while loop:** *The while loop is used when you want to repeat a block of code as long as a certain condition is true.*

3. **do-while loop:** *The do-while loop is similar to the while loop, but it guarantees that the block of code is executed at least once, as the condition is checked after the execution.*

   The loop control variable (i in these examples) is incremented inside the loop to eventually satisfy the loop condition and terminate the loop.
   Loops are fundamental for repetitive tasks and are widely used in programming to iterate over data, perform calculations, or execute a block of code until a specific condition is met.

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

Ans.

Debugging is crucial in programming to identify and rectify errors in code.

1. Syntax Errors:

- *<u>Importance</u>: Syntax errors prevent code compilation, making it crucial to address them before execution.*
- *<u>Example:</u>*

```
#include <stdio.h> int
main() {
   printf("Hello, World!")
return 0;
}
```

Debugging: *The compiler will highlight the line with the syntax error during compilation.*

2. Logic Errors:

- *Importance: Logic errors lead to incorrect program output and require careful code review.*
- *Example:*

```
#include <stdio.h>
int main() {     int x
= 5;
   printf("The square of x is: %d", x * x);
return 0;
}
```

Debugging: *Review the code to identify the incorrect logic; in this case, it should be x * x for the square.*

3. Runtime Errors:

- *Importance: These errors occur during execution and can cause program crashes or unexpected behavior.*
- *Example:*

```
#include <stdio.h> int
main() {     int arr[3] =
{1, 2, 3};
   printf("Value at index 5: %d", arr[5]);
return 0;
```

}

Debugging: *Use tools like gdb to trace and identify the runtime error, in this case, an array index out of bounds.*

✞ *Debugging Tools in C:*

- *gdb (GNU Debugger): Allows step-by-step execution, setting breakpoints, and inspecting variables.*
- *Valgrind: Detects memory leaks, memory corruption, and other memory-related issues.*
- *strace: Traces system calls and signals, aiding in debugging system-level issues.*

✞ Testing C Programs:

- *Unit Testing: #include <assert.h> int square(int x) {*

  *return x * x;*

}

int main() {

assert(square(2) == 4);

return 0;

}

- *Integration Testing: Test interactions between functions or modules.*
- *Regression Testing: Ensure that new changes don't break existing functionality.*
- *Boundary Testing: #include <assert.h> int divide(int a, int b) { assert(b != 0); return a / b;*

}

A combination of careful code review, debugging tools, and a comprehensive testing strategy is essential to ensure code correctness and reliability.

Q7. What is the user defined and pre-defined functions.

Explain with example call by value and call by reference.

Ans.

In C programming, functions are divided into user-defined functions and pre-defined (standard library) functions.

- **User-Defined Functions:**

Definition: *Functions created by the programmer to perform specific tasks.*

Example:

```
#include <stdio.h> //
User-defined function
int add(int a, int b) {
return a + b;
}

int main() {
   int result = add(3, 5); // Calling the user-defined function
printf("The sum is: %d", result);     return 0;
}
```

In this example, the add function is user-defined to add two integers.

- **Pre-defined Functions:**

Definition: **Functions provided by the C standard library.**

Example:

```
#include  <stdio.h>
#include <math.h>
int main() {
```

```
   double squareRoot = sqrt(25); // Using the pre-defined sqrt function
   printf("The square root is: %f", squareRoot);
return 0;
}
```

Here, the sqrt function is pre-defined and calculates the square root.

- **Call by Value:**

Definition: *In call by value, function parameters receive copies of the actual arguments, and modifications inside the function do not affect the original values.*

Example:

```
#include <stdio.h> void
increment(int x) {     x =
x + 1;
}

int main() {
int num = 5;
   increment(num); // Passing the value of num
printf("Original value: %d", num);     return 0;
}
```

The increment function modifies its local copy of x, leaving the original num unchanged.

- **Call by Reference:**

Definition: *In call by reference, function parameters receive the memory address of the actual arguments. Changes made inside the function affect the original values.*

Example:

```
#include <stdio.h>
void incrementByReference(int *x) {
   (*x) = (*x) + 1;
}

int main() {
   int num = 5;
   incrementByReference(&num); // Passing the address of num
printf("Updated value: %d", num);     return 0;
}
```

The incrementByReference function modifies the value at the memory address pointed to by x, thereby changing the original num.

✞ User-defined functions are created by programmers, while pre-defined functions are part of the C standard library. Call by value involves passing copies of values, and call by reference involves passing memory addresses to modify 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 output of S1 and S2 b) Count the number of characters and empty spaces in S1and S2.

Ans.

In C programming, passing arguments to a function involves providing values or variables as input when calling the function,

while returning values involves using the return statement to send a result back to the calling code.

- **Passing Arguments to a Function:**

When defining a function, you declare its parameters inside parentheses. These parameters act as placeholders for the values or variables you pass when calling the function.

- **Returning Values from a Function:**

Functions can return values using the return statement. The return type is declared in the function signature. Passing arguments allows you to provide input to a function, and returning values allows a function to communicate its result back to the part of the program that called it.

✞ **In C programming, storage classes determine the scope, lifetime, and visibility of variables. There are four primary storage classes: automatic, static, register, and external.**

- *Automatic (auto):*

Variables declared as automatic have a local scope, meaning they are only accessible within the block in which they are defined.

Memory is automatically allocated when the block is entered and released when the block is exited.
The auto keyword is rarely used explicitly, as it is the default storage class for local variables.

- *Static:*

Variables declared as static persist throughout the program's execution.

They retain their values between function calls, making them useful for maintaining state across multiple invocations.

By default, static variables have internal linkage within the file. Adding static explicitly makes it clear and restricts the scope.

- *Register:*

The register keyword suggests to the compiler that the variable should be stored in a register for faster access.

It's a hint to the compiler, and it may or may not actually use a register depending on the architecture and other factors.

- *External (extern):*

Variables declared with extern keyword are used to declare a global variable that is defined in another file. It allows the linkage of variables across multiple files.

storage classes in C help define how variables behave in terms of scope, lifetime, and linkage, providing flexibility in managing memory and variable access in different contexts.

✞ ***Program for 2 strings S1 and S2.***

```
#include   <stdio.h>
#include <string.h>
int main() {
   char s1[100], s2[100];
printf("Enter string s1: ");
scanf("%s", s1);     printf("Enter
string s2: ");     scanf("%s", s2);
if (strcmp(s1, s2) == 0) {
printf("Strings are equal.\n");
   } else {
      printf("Strings are not equal.\n");
   }
   return 0;
}
```

This program uses the strcmp function from the string.h library to compare the two strings. It takes input for s1 and s2 and then prints whether the strings are equal or not.

✞ Program to display a concatenated outout of S1 and S2

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

int main() {
   char s1[100], s2[100], result[200];

   printf("Enter string s1: ");
scanf("%s", s1);

   printf("Enter string s2: ");
scanf("%s", s2);

   // Concatenate s1 and s2
strcpy(result, s1);     strcat(result,
s2);

   // Display the concatenated string
printf("Concatenated string: %s\n", result);

   return 0;
}
```

This program uses the strcpy function to copy the contents of s1 into result and then uses strcat to concatenate s2 to result. Finally, it prints the concatenated string.

✞ Program to count the number of characters and empty spaces in S1 and S2

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

int main() {
   char s1[100], s2[100];

   printf("Enter string s1: ");
gets(s1);

   printf("Enter string s2: ");
gets(s2);

   int len_s1 = strlen(s1);
int len_s2 = strlen(s2);

   int charCount_s1 = 0, spaceCount_s1 = 0;
   int charCount_s2 = 0, spaceCount_s2 = 0;

   for (int i = 0; i < len_s1; i++) {
if (s1[i] == ' ') {
         spaceCount_s1++;
      } else {
         charCount_s1++;
```

```
        }
    }

    for (int i = 0; i < len_s2; i++) {
if (s2[i] == ' ') {
            spaceCount_s2++;
} else {
            charCount_s2++;
        }
    }

    printf("s1 - Character count: %d, Space count: %d\n",
charCount_s1, spaceCount_s1);
    printf("s2 - Character count: %d, Space count: %d\n",
charCount_s2, spaceCount_s2);

    return 0;
}
```

This program uses the strlen function to find the length of each string and then iterates through each character to count the number of characters and spaces. Finally, it prints the counts for both s1 and s2. Note that gets function is used to input strings, but keep in mind that it has been deprecated in newer versions of C.

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.

Ans.

- **1D array**

```
#include <stdio.h> int
main() {
   // 1D array
   int one_dimensional_array[] = {1, 2, 3, 4, 5};
// Accessing elements in the 1D array     for
(int i = 0; i < 5; ++i) {
      printf("%d ", one_dimensional_array[i]);
   }
   return 0;
}
```

In this example, one_dimensional_array is a 1D array of integers, and the elements are accessed using indices.

- **Multidimensional array**

```
#include <stdio.h>
int main() {    // 2D
array
   int two_dimensional_array[3][3] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };
   // Accessing elements in the 2D array
for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 3; ++j) {
         printf("%d ", two_dimensional_array[i][j]);
      }
      printf("\n");
   }
   return 0;
```

```
    }
```

In the 2D array example, two_dimensional_array is a 3x3 array, and elements are accessed using two indices for rows and columns. The nested loops are used to iterate through the elements.

✞ **C program that implements matrix multiplication for two matrices of size m x n. The program defines functions for matrix multiplication and for displaying matrices.**

```
#include <stdio.h>
#define MAX_SIZE 10
// Function to multiply two matrices
void multiplyMatrices(int firstMatrix[MAX_SIZE][MAX_SIZE],
int secondMatrix[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] += firstMatrix[i][k] * secondMatrix[k][j];
        }
    }
  }
}
// Function to display a matrix
void displayMatrix(int matrix[MAX_SIZE][MAX_SIZE], int
rows, int cols) {
  for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
}
```

```
// Function to display a matrix
void displayMatrix(int matrix[MAX_SIZE][MAX_SIZE], int
rows, int cols) {
   for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}
int main() {    int m, n,
p;    // Input matrix
sizes
   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);
   if (m > MAX_SIZE || n > MAX_SIZE || p > MAX_SIZE) {
printf("Matrix size exceeds the maximum limit of %d.\n",
MAX_SIZE);
      return 1;
   }
   int firstMatrix[MAX_SIZE][MAX_SIZE],
secondMatrix[MAX_SIZE][MAX_SIZE],
result[MAX_SIZE][MAX_SIZE];
   // Input elements of the first matrix
printf("Enter elements of the first matrix:\n");
   for (int i = 0; i < m; ++i) {
for (int j = 0; j < n; ++j) {
         scanf("%d", &firstMatrix[i][j]);
      }
   }
```

```
   // Input elements of the second matrix
printf("Enter elements of the second matrix:\n");
for (int i = 0; i < n; ++i) {
      for (int j = 0; j < p; ++j) {
         scanf("%d", &secondMatrix[i][j]);
      }
   }
   // Perform matrix multiplication
   multiplyMatrices(firstMatrix, secondMatrix, result, m, n, p);
   // Display the result
   printf("Resultant matrix after multiplication:\n");
displayMatrix(result, m, p);
   return 0;
}
```

✞ **Here are three functions in C programming for reading matrix elements, matrix multiplication, and printing matrix elements:**

```
   #include <stdio.h>
   #define MAX_SIZE 10
   // Function to read matrix elements
   void readMatrix(int matrix[MAX_SIZE][MAX_SIZE], int
   rows, int cols) {
      printf("Enter matrix elements:\n");
   for (int i = 0; i < rows; ++i) {       for
   (int j = 0; j < cols; ++j) {
            scanf("%d", &matrix[i][j]);
         }
      }
   }
   // Function to perform matrix multiplication
   void                    multiplyMatrices(int
   firstMatrix[MAX_SIZE][MAX_SIZE],     int
   secondMatrix[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] += firstMatrix[i][k] * secondMatrix[k][j];
        }
      }
   }
}
// Function to print matrix elements
void printMatrix(int matrix[MAX_SIZE][MAX_SIZE], int
rows, int cols) { printf("Matrix elements:\n");    for
(int i = 0; i < rows; ++i) {       for (int j = 0; j < cols; ++j)
{
        printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}
int main() {
int m, n, p;
   // Input matrix sizes
   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);
if (m > MAX_SIZE || n > MAX_SIZE || p > MAX_SIZE) {
printf("Matrix size exceeds the maximum limit of %d.\n",
MAX_SIZE);
    return 1;
  }
```

```
    int firstMatrix[MAX_SIZE][MAX_SIZE],
secondMatrix[MAX_SIZE][MAX_SIZE],
result[MAX_SIZE][MAX_SIZE];

    // Read elements of the first matrix

readMatrix(firstMatrix, m, n);     // Read

elements of the second matrix

readMatrix(secondMatrix, n, p);     //

Perform matrix multiplication

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

    // Print the elements of the result matrix
    printMatrix(result, m, p);

    return 0;

}
```

In this program, the readMatrix function reads the elements of a matrix, the multiplyMatrices function performs matrix multiplication, and the printMatrix function prints the elements of a matrix. These functions are then used in the main function to read two matrices, multiply them, and print the result.

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.

Ans.

- **Structure Declaration and Initialization:**

```
#include <stdio.h>
// Structure Declaration
struct Point {
   int x;
int y;
};
int main() {
   // Structure Initialization
struct Point p1 = {3, 5};
   // Accessing structure members
printf("Coordinates: (%d, %d)\n", p1.x, p1.y);     return
0;
}
```

In this example, we've declared a structure Point with two members (x and y). We then initialize a variable p1 of type Point with values 3 and 5 for its members.

- ***Array Structures:***

```
#include <stdio.h> struct
Point {
   int x;
int y;
};
int main() {
   // Array of Structures     struct Point points[3] =
{{1, 2}, {3, 4}, {5, 6}};     // Accessing elements of
the array of structures     for (int i = 0; i < 3; ++i) {
      printf("Point %d: (%d, %d)\n", i+1, points[i].x, points[i].y);
```

```
    }
    return 0;
}
```

Here, we've created an array of structures (points) and initialized each structure with specific values.

- ***Use of Typedef:***

```
#include <stdio.h>
// Typedef for simplifying structure declaration typedef
struct {
    int x;
int y; }
Point; int
main() {
    // Using the typedef
    Point p1 = {3, 5};
    // Accessing structure members
printf("Coordinates: (%d, %d)\n", p1.x, p1.y);     return
0;
}
```

Typedef is used to simplify the declaration of structures. In this example, we use typedef to create an alias Point for the structure.

- ***Passing Structure to Functions:***

```
#include <stdio.h> struct
Point {
    int x;
int y;
};
```

```
// Function taking a structure as a parameter void
printPoint(struct Point p) {
   printf("Coordinates: (%d, %d)\n", p.x, p.y);
}
int main() {
   struct Point p1 = {3, 5};
   // Passing structure to a function
printPoint(p1);
   return 0;
}
```

Here, we define a function printPoint that takes a struct Point as a parameter and prints its coordinates. In the main function, we create a structure and pass it to the function.

✞ *Union Declaration and Initialization:*

```
#include <stdio.h> //
Union Declaration
union Shape {
   int sides;
   float radius;
};
int main() {
   // Union Initialization
union Shape circle;
   circle.radius = 5.0;
   // Accessing union members
   printf("Radius: %.2f\n", circle.radius);
// Changing the union member
   circle.sides = 3;
   // Accessing union members after changing
   printf("Sides: %d\n", circle.sides);
   return 0;
```

```
}
```

In this example, we've declared a union Shape with two members (sides and radius). The memory allocated for a union is large enough to hold its largest member. We initialize the union variable circle with a radius value and then change its member to represent the number of sides.

- ***Passing Unions and Structures to Functions:***

```
#include <stdio.h> //
Structure Declaration
struct Point {
   int x;
int y;
};
// Union Declaration union
Geometry {
   struct Point point;
   float area;
};
// Function taking a structure as a parameter
void printPoint(struct Point p) {
   printf("Coordinates: (%d, %d)\n", p.x, p.y);
}
// Function taking a union as a parameter void
printGeometry(union Geometry shape) {
   printf("Area: %.2f\n", shape.area);
}

int main() {
   // Structure Initialization
   struct Point p1 = {3, 5};

   // Union Initialization
union Geometry geometry;
geometry.point = p1;
```

```
    // Passing structure to a function
    printPoint(p1);

    // Passing union to a function
    printGeometry(geometry);

    return 0;
}
```

In this example, we have a structure Point and a union Geometry that can represent either a point or an area. The printPoint function takes a structure as a parameter and prints its coordinates. The printGeometry function takes a union as a parameter and prints its area. In the main function, we initialize a point and a union, then pass them to their respective functions.

- ***In C programming, structures and unions are both used to group different data types together, but they differ in how they allocate memory for their members.***

Structure:

- *Memory Allocation:*

Each member of a structure has its own memory location.

Memory is allocated for each member individually.

- *Memory Usage:*

Structures are useful when you want to store and access multiple pieces of data simultaneously.

It's common to use structures when you need to represent a record with different attributes.

- *Size Calculation:*

The size of a structure is the sum of the sizes of its individual members.

- *Accessing Members:*

Members are accessed using the dot (.) operator.

Example:

```
struct Point {
   int x;
int y;
};
```

Union:

- *Memory Allocation:*

All members of a union share the same memory location.

Only one member of the union can be active at a time.

- *Memory Usage:*

Unions are useful when you want to represent a single piece of data that can be of different types.

Useful for saving memory when only one member is needed at a time.

- *Size Calculation:*

The size of a union is the size of its largest member.

- *Accessing Members:*

Members are accessed using the dot (.) operator, similar to structures.

However, only the currently active member should be accessed.

Example:

```
union Value {     int
intValue;     float
floatValue;
```

};

Structures are more commonly used because they provide a convenient way to group related data, while unions are used in specific cases where memory efficiency is a critical concern.

✞ Program in C that represents a bank account with the specified fields

```
#include <stdio.h>
#include <string.h>
// Structure to represent a bank account
struct BankAccount {     int
accountNumber;     char
accountHolderName[50];
   float balance;
};
int main() {
   // Creating an instance of the BankAccount structure
struct BankAccount myAccount;     // Input values for the
account     printf("Enter Account Number: ");
scanf("%d", &myAccount.accountNumber);
printf("Enter Account Holder's Name: ");     scanf("%s",
myAccount.accountHolderName);  // Note: Using %s for
string input     printf("Enter Balance: ");     scanf("%f",
&myAccount.balance);
   // Displaying the details
printf("\nAccount Details:\n");
printf("Account Number: %d\n",
myAccount.accountNumber);
printf("Account Holder's Name: %s\n",
myAccount.accountHolderName);
   printf("Balance: $%.2f\n", myAccount.balance);

   return 0;
```

```
}
```

This program defines a structure Bank Account with the specified fields. It then creates an instance of this structure, takes input for the account details, and finally displays the entered information. Note that this is a basic example, and in a real-world scenario, you might want to include more features and error handling.

✞ C program that reads details for 10 people's bank accounts and displays the record with the highest bank balance

```
#include <stdio.h>
#include <string.h>
// Structure to represent a bank account
struct BankAccount {     int
accountNumber;     char
accountHolderName[50];
   float balance;
};
int main() {
   // Array to store details of 10 people
struct BankAccount accounts[10];     //
Input details for 10 people     for (int i
= 0; i < 10; ++i) {
       printf("Enter details for person %d:\n", i + 1);
printf("Enter Account Number: ");        scanf("%d",
&accounts[i].accountNumber);        printf("Enter
Account Holder's Name: ");        scanf("%s",
accounts[i].accountHolderName);
       printf("Enter Balance: ");
scanf("%f", &accounts[i].balance);
   }
```

```
   // Find the record with the highest 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;
      }
   }
   // Display the record with the highest balance
printf("\nDetails of the person with the highest bank
balance:\n");
   printf("Account Number: %d\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 uses an array of Bank Account structures to store details for 10 people. It then iterates through the array to find the record with the highest balance and displays the details of that person.