Selfcode Academy

www.selfcode.in

Practice Sheet (C and C++ Programming)
(Solution)

Section 1: Programming with C

Lecture 2: Introduction to C programming

Practical Question: Write a C program to display "Hello, World!" on the console.

```c
#include <stdio.h>

int main() {
    printf("Hello, World!");
    return 0;
}
```

Practical Question: Write a C program to take two numbers as input from the user and print their sum.

```c
#include <stdio.h>

int main() {
    int num1, num2, sum;

    printf("Enter the first number: ");
    scanf("%d", &num1);

    printf("Enter the second number: ");
    scanf("%d", &num2);

    sum = num1 + num2;

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

    return 0;
}
```

Theory Question: Explain the basic structure of a C program.

C program typically consists of the following basic structure:

```c
#include <stdio.h> // Include necessary header files

int main() { // The starting point of the program
    // Declarations and statements

    return 0; // Optional: Return statement to indicate successful execution
}
```

The #include <stdio.h> line includes the standard input/output library, which provides functions like printf() and scanf(). The main() function is the entry point of the program, and it contains the program's logic. Declarations and statements are written inside the main() function. The return 0; statement is optional and is used to indicate that the program has executed successfully.

Lecture 3: Variables and Datatypes: (Part 1)

Practical Question: Write a C program to declare and initialize variables of different data types (int, float, char) and display their values.

```c
#include <stdio.h>

int main() {
    int num = 10;
    float pi = 3.14;
    char letter = 'A';

    printf("Integer: %d\n", num);
    printf("Float: %.2f\n", pi);
    printf("Character: %c\n", letter);

    return 0;
}
```

Practical Question: Write a C program to swap the values of two variables without using a temporary variable.

```c
#include <stdio.h>

int main() {
    int num1, num2;

    printf("Enter the first number: ");
    scanf("%d", &num1);

    printf("Enter the second number: ");
    scanf("%d", &num2);

    printf("Before swapping: num1 = %d, num2 = %d\n", num1, num2);

    num1 = num1 + num2;
    num2 = num1 - num2;
    num1 = num1 - num2;

    printf("After swapping: num1 = %d, num2 = %d\n", num1, num2);

    return 0;
}
```

Theory Question: Explain the concept of variables and data types in C programming.

In C programming, a variable is a named storage location that holds a value. Variables are used to store data that can be manipulated and used within the program. Before using a variable, it must be declared with a specific data type, which determines the size and type of data that the variable can hold.

C supports various data types, including:

- int: Used to store integers (whole numbers).
- float: Used to store floating-point numbers (real numbers with a fractional part).
- char: Used to store single characters.
- double: Used to store double-precision floating-point numbers.
- short, long: Used to store integers with different ranges.
- unsigned: Used to store positive integers or zero.
- void: Represents the absence of type or value.

Variables can be initialized with an initial value at the time of declaration. For example, int num = 10; declares an integer variable named num and initializes it with the value 10.

The choice of data type depends on the nature of the data to be stored and the range of values it should hold. It is important to choose the appropriate data type to ensure efficient memory usage and correct calculations in the program.

Lecture 4: Variables and Datatypes: (Part 2)

Practical Question: Write a C program to calculate the area of a rectangle using user-input values for length and width.

```c
#include <stdio.h>

int main() {
    float length, width, area;

    printf("Enter the length of the rectangle: ");
    scanf("%f", &length);

    printf("Enter the width of the rectangle: ");
    scanf("%f", &width);

    area = length * width;

    printf("The area of the rectangle is: %.2f", area);

    return 0;
}
```

Practical Question: Write a C program to convert temperature from Celsius to Fahrenheit.

```c
#include <stdio.h>

int main() {
    float celsius, fahrenheit;

    printf("Enter the temperature in Celsius: ");
    scanf("%f", &celsius);

    fahrenheit = (celsius * 9 / 5) + 32;

    printf("The temperature in Fahrenheit is: %.2f", fahrenheit);

    return 0;
}
```

Theory Question: Differentiate between local variables and global variables in C.

In C programming, local variables and global variables are two types of variables with different scopes and lifetimes:

Local variables:

- Declared inside a block of code (such as a function or a compound statement).
- Limited to the block where they are declared and can only be accessed within that block.
- Have automatic storage duration, meaning they are created when the block is entered and destroyed when the block is exited.
- Each invocation of the block creates a new instance of the local variable.
- Local variables must be initialized before they are used.

- **Example:**

```
void myFunction() {
    int localVar = 10; // Local variable
    // Rest of the code
}
```

Global variables:

- Declared outside of any block, usually at the top of the program.
- Can be accessed and modified by any part of the program, including functions.
- Have static storage duration, meaning they exist throughout the execution of the program.
- Only one instance of a global variable exists, and it retains its value between function calls.
- Global variables are initialized by default if not explicitly initialized.

- **Example:**

```
int globalVar = 20; // Global variable

void myFunction() {
    // Access and modify globalVar here
    // Rest of the code
}
```

It is generally recommended to use local variables whenever possible to encapsulate data and prevent unintended modifications. Global variables should be used sparingly and only when necessary.

Lecture 5: Type Conversion

Practical Question: Write a C program to take an integer as input and convert it into its equivalent character.

```c
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);

    char ch = (char)num;

    printf("The equivalent character is: %c", ch);

    return 0;
}
```

Practical Question: Write a C program to perform arithmetic operations on mixed data types (int and float).

```c
#include <stdio.h>

int main() {
    int num1 = 10;
    float num2 = 3.5;

    float sum = num1 + num2;
    float difference = num1 - num2;
    float product = num1 * num2;
    float quotient = num1 / num2;

    printf("Sum: %.2f\n", sum);
    printf("Difference: %.2f\n", difference);
    printf("Product: %.2f\n", product);
    printf("Quotient: %.2f\n", quotient);

    return 0;
}
```

Theory Question: Explain the concept of type conversion in C programming.

Type conversion, also known as type casting, refers to the process of converting a value from one data type to another. In C programming, type conversion is used to ensure proper data handling and compatibility in expressions or assignments involving different data types.

There are two types of type conversion:

Implicit Type Conversion:

- Also known as automatic type conversion or coercion.
- Performed automatically by the compiler when compatible types are involved in an expression.
- The conversion is done to promote the operands to a common type, following a set of rules defined by the C language.
- Example: Assigning an integer value to a float variable.

Explicit Type Conversion:

- Also known as type casting.
- Performed manually by the programmer using casting operators.
- Used when there is a need to convert a value explicitly from one data type to another.
- It allows for more control over the conversion process but should be used with caution to avoid loss of data or precision.
- Example: Converting an integer to a character or vice versa using (char) or (int).

Type conversion is essential for performing operations involving mixed data types, ensuring that the operands are compatible and the correct result is obtained. However, it is important to be aware of the potential loss of data or precision when converting between different types.

Lecture 6: Constants

Practical Question: Write a C program to calculate the area of a circle using a constant value for pi.

```c
#include <stdio.h>

#define PI 3.14159

int main() {
    float radius, area;

    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);

    area = PI * radius * radius;

    printf("The area of the circle is: %.2f", area);

    return 0;
}
```

Practical Question: Write a C program to define and use symbolic constants for maximum and minimum values.

```c
#include <stdio.h>

#define MAX_VALUE 100
#define MIN_VALUE 0

int main() {
    int num;

    printf("Enter a number between %d and %d: ", MIN_VALUE, MAX_VALUE);
    scanf("%d", &num);

    if (num >= MIN_VALUE && num <= MAX_VALUE) {
        printf("Valid number entered: %d", num);
    } else {
        printf("Invalid number entered.");
    }

    return 0;
}
```

16

Theory Question: What is the difference between a constant and a variable in C?

In C programming, constants and variables are used to store and represent data, but they differ in their properties and behavior:

Constants:

- A constant is a value that does not change during program execution.
- Constants are defined using the const keyword or preprocessor directive #define.
- Constants are typically used for values that are known and fixed, such as mathematical constants or predefined limits.
- Constants can be of any data type, including integers, floating-point numbers, characters, or even user-defined types.
- The value of a constant cannot be modified once it is defined.
- Constants are often used to improve code readability and avoid magic numbers.

Variables:

- A variable is a named storage location that can hold a value that may change during program execution.
- Variables are declared with a specific data type, indicating the type of data they can store.
- Variables are used to store and manipulate data as the program runs.
- The value of a variable can be modified multiple times during the execution of the program.
- Variables can be assigned different values based on conditions, user input, or calculations.

In summary, constants represent fixed values that do not change, while variables store values that can be modified and updated during program execution.

Lecture 7: Operators and Expressions

Practical Question: Write a C program to perform arithmetic operations on two numbers and display the results.

```c
#include <stdio.h>

int main() {
    int num1, num2;

    printf("Enter the first number: ");
    scanf("%d", &num1);

    printf("Enter the second number: ");
    scanf("%d", &num2);

    int sum = num1 + num2;
    int difference = num1 - num2;
    int product = num1 * num2;
    int quotient = num1 / num2;
    int remainder = num1 % num2;

    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);

    return 0;
}
```

Practical Question: Write a C program to calculate the area of a triangle using the values of base and height entered by the user.

```c
#include <stdio.h>

int main() {
    float base, height, area;

    printf("Enter the base of the triangle: ");
    scanf("%f", &base);

    printf("Enter the height of the triangle: ");
    scanf("%f", &height);

    area = 0.5 * base * height;

    printf("The area of the triangle is: %.2f", area);

    return 0;
}
```

Theory Question: Explain the different types of operators in C programming.

In C programming, operators are symbols that perform specific operations on one or more operands. The different types of operators in C can be categorized as follows:

Arithmetic Operators:

- Used for basic arithmetic calculations.
- Examples: ‘+’ (addition), ‘ - ’ (subtraction), ’ * ‘ (multiplication), ’ / ‘ (division), ‘ % ’ (modulus).

Relational Operators:

- Used for comparison between operands.
- Results in a logical (Boolean) value of either 0 (false) or 1 (true).
- Examples: ’ == ’ (equality), ’ != ‘ (inequality), ’ > ’ (greater than), ’ < ‘ (less than), ‘ >= ‘ (greater than or equal to), ‘<=’ (less than or equal to).

Logical Operators:

- Used to combine or modify logical conditions.
- Examples: ‘ && ’ (logical AND), ‘ || ’ (logical OR), ‘ ! ’ (logical NOT).

Assignment Operators:

- Used to assign values to variables.
- Examples: ‘ = ‘ (simple assignment), ‘ += ’ (addition assignment), ‘ -= ‘ (subtraction assignment), ‘ *= ’ (multiplication assignment), ‘ /= ‘ (division assignment), ‘ %= ‘ (modulus assignment).

Increment and Decrement Operators:

- Used to increment or decrement the value of a variable.
- Examples: ‘ ++ ‘ (increment), ‘ – ‘ (decrement).

Bitwise Operators:

- Used for bit-level operations on binary values.
- Examples: ‘ & ‘ (bitwise AND), ‘ | ‘ (bitwise OR), ’ ^ ‘ (bitwise XOR), ‘ ~ ‘ (bitwise NOT), ‘ << ‘ (left shift), ‘ >> ‘ (right shift).

These are some of the commonly used operators in C programming. They allow for various operations and manipulations of data, enabling complex computations and decision-making in programs.

Lecture 8: Control Structures

Practical Question: Write a C program to check whether a number entered by the user is positive, negative, or zero.

```c
#include <stdio.h>

int main() {
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num > 0) {
        printf("The number is positive.");
    } else if (num < 0) {
        printf("The number is negative.");
    } else {
        printf("The number is zero.");
    }

    return 0;
}
```

Practical Question: Write a C program to determine whether a character entered by the user is a vowel or consonant.

```c
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a character: ");
    scanf(" %c", &ch);

    if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' ||
        ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {
        printf("The character is a vowel.");
    } else {
        printf("The character is a consonant.");
    }

    return 0;
}
```

Theory Question: Explain the concept of control structures in C programming.

Control structures in C programming are used to control the flow of program execution based on certain conditions or criteria. They determine the order in which statements are executed and allow for making decisions or repeating actions.

The three main types of control structures in C are:

Decision-making Control Structures:

- Allows the program to make decisions based on conditions.
- Example: if statement, if-else statement, switch statement.
- Controls the flow of execution based on whether a condition is true or false.

Loop Control Structures:

- Repeats a block of code multiple times as long as a condition is true.
- Example: for loop, while loop, do-while loop.
- Controls the repetition of statements or actions.

Branching Control Structures:

- Alters the normal flow of program execution by transferring control to a different part of the program.
- Example: break statement, continue statement, goto statement.
- Allows jumping to a specific section of code or terminating a loop prematurely.

Control structures provide flexibility and enable programmers to create programs that respond dynamically to different conditions. They play a crucial role in implementing logic, making decisions, and performing repetitive tasks in C programming.

Lecture 9: Switch Statements

Practical Question: Write a C program to display the name of a day of the week based on the user-inputted number (1-7).

```c
#include <stdio.h>

int main() {
    int day;

    printf("Enter a number (1-7): ");
    scanf("%d", &day);

    switch (day) {
        case 1:
            printf("Monday");
            break;
        case 2:
            printf("Tuesday");
            break;
        case 3:
            printf("Wednesday");
            break;
```

```
        case 4:
            printf("Thursday");
            break;
        case 5:
            printf("Friday");
            break;
        case 6:
            printf("Saturday");
            break;
        case 7:
            printf("Sunday");
            break;
        default:
            printf("Invalid input");
    }

    return 0;
}
```

Practical Question: Write a C program to calculate the grade of a student based on their percentage using switch statements.

```c
#include <stdio.h>

int main() {
    float percentage;

    printf("Enter the percentage: ");
    scanf("%f", &percentage);

    char grade;

    if (percentage >= 90) {
        grade = 'A';
    } else if (percentage >= 80) {
        grade = 'B';
    } else if (percentage >= 70) {
        grade = 'C';
    } else if (percentage >= 60) {
        grade = 'D';
    } else if (percentage >= 50) {
        grade = 'E';
    } else {
        grade = 'F';
    }
```

```
    switch (grade) {
        case 'A':
            printf("Excellent");
            break;
        case 'B':
            printf("Very Good");
            break;
        case 'C':
            printf("Good");
            break;
        case 'D':
            printf("Average");
            break;
        case 'E':
            printf("Pass");
            break;
        case 'F':
            printf("Fail");
            break;
        default:
            printf("Invalid percentage");
    }

    return 0;
}
```

28

Theory Question: What is the purpose of using switch statements in C? Provide an example.

Example:

```c
#include <stdio.h>

int main() {
    int choice;

    printf("Select an option:\n");
    printf("1. Start\n");
    printf("2. Pause\n");
    printf("3. Stop\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Starting the program...");
            break;
        case 2:
            printf("Pausing the program...");
            break;
        case 3:
            printf("Stopping the program...");
            break;
        default:
            printf("Invalid choice");
    }

    return 0;
}
```

In this example, the user is prompted to select an option, and the program uses a switch statement to perform different actions based on the chosen option. The switch statement provides a structured and efficient way to handle multiple cases and execute the appropriate code block based on the selected option.

The purpose of using switch statements in C is to provide a convenient way to make decisions based on multiple possible values of a single variable. Switch statements offer an alternative to using multiple if-else if statements, making the code more readable and concise.

Lecture 10: For Loops

Practical Question: Write a C program to print all even numbers between 1 and 50 using a for loop.

```c
#include <stdio.h>

int main() {
    printf("Even numbers between 1 and 50:\n");

    for (int i = 2; i <= 50; i += 2) {
        printf("%d ", i);
    }

    return 0;
}
```

Practical Question: Write a C program to calculate the factorial of a number using a for loop.

```c
#include <stdio.h>

int main() {
    int number;
    unsigned long long factorial = 1;

    printf("Enter a number: ");
    scanf("%d", &number);

    for (int i = 1; i <= number; i++) {
        factorial *= i;
    }

    printf("Factorial of %d = %llu", number, factorial);

    return 0;
}
```

Theory Question: Explain the structure and working of a for loop in C programming.

The structure of a for loop in C programming is as follows:

```
for (initialization; condition; increment/decrement) {
    // code to be executed
}
```

- The initialization step is executed only once at the beginning of the loop and is used to initialize the loop control variable.
- The condition is evaluated before each iteration of the loop. If the condition is true, the loop body is executed. If the condition is false, the loop terminates.
- The increment/decrement step is executed at the end of each iteration and is used to modify the loop control variable.

The working of a for loop can be summarized as follows:

- The initialization step is performed.
- The condition is evaluated. If the condition is false, the loop is terminated, and the program continues with the next statement after the loop.
- If the condition is true, the loop body is executed.
- After the execution of the loop body, the increment/decrement step is performed.
- The condition is evaluated again, and the process continues until the condition becomes false.

The loop control variable is typically used to control the number of iterations or to keep track of the loop progress. The for loop provides a concise and structured way to execute a block of code repeatedly based on a given condition.

Lecture 11: Nested for loop, While loop, and Do-while loop

Practical Question: Write a C program to display a pyramid pattern using nested for loops.

```c
#include <stdio.h>

int main() {
    int rows;

    printf("Enter the number of rows: ");
    scanf("%d", &rows);

    for (int i = 1; i <= rows; i++) {
        for (int j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }

    return 0;
}
```

Practical Question: Write a C program to find the sum of digits of a number using a while loop.

```c
#include <stdio.h>

int main() {
    int number, sum = 0, digit;

    printf("Enter a number: ");
    scanf("%d", &number);

    while (number > 0) {
        digit = number % 10;
        sum += digit;
        number /= 10;
    }

    printf("Sum of digits = %d", sum);

    return 0;
}
```

Theory Question: Compare and contrast the while loop and do-while loop in C.

While loop:

- The while loop is an entry-controlled loop, which means that the condition is evaluated before the loop body is executed.
- If the condition is false initially, the loop body is not executed at all.
- The syntax of a while loop is as follows:

```
while (condition) {
    // code to be executed
}
```

Do-while loop:

- The do-while loop is an exit-controlled loop, which means that the condition is evaluated after the loop body is executed.
- The loop body is guaranteed to be executed at least once, even if the condition is false initially.
- The syntax of a do-while loop is as follows:

```
do {
    // code to be executed
} while (condition);
```

Comparison:

- The primary difference between the while loop and the do-while loop is the evaluation of the condition. In the while loop, the condition is evaluated before the loop body, while in the do-while loop, the condition is evaluated after the loop body.
- If the condition is false initially, the while loop will not execute the loop body at all, while the do-while loop will execute the loop body once before checking the condition.
- In both loops, the condition determines whether the loop will continue or terminate.
- The choice between the while loop and the do-while loop depends on the specific requirements of the program. If you want the loop body to execute at least once, use a do-while loop. If you want to check the condition before the first iteration, use a while loop.

Lecture 12: Arrays

Practical Question: Write a C program to find the largest element in an array.

```c
#include <stdio.h>

#define SIZE 5

int main() {
    int arr[SIZE];

    printf("Enter %d elements:\n", SIZE);

    for (int i = 0; i < SIZE; i++) {
        scanf("%d", &arr[i]);
    }

    int max = arr[0];

    for (int i = 1; i < SIZE; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    printf("Largest element: %d", max);

    return 0;
}
```

Practical Question: Write a C program to calculate the average of elements in an array.

```c
#include <stdio.h>

#define SIZE 5

int main() {
    int arr[SIZE];
    int sum = 0;
    float average;

    printf("Enter %d elements:\n", SIZE);

    for (int i = 0; i < SIZE; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i];
    }

    average = (float)sum / SIZE;

    printf("Average: %.2f", average);

    return 0;
}
```

Theory Question: Explain the concept of arrays and their use in C programming.

An array is a collection of elements of the same data type stored in contiguous memory locations. It provides a way to store multiple values of the same type under a single variable name. Each element in the array is accessed using an index, which represents its position within the array.

Arrays are useful in programming because they allow us to work with a group of related values efficiently. Some key points about arrays in C programming are:

- Declaration: Arrays are declared using the syntax 'datatype arrayName[arraySize];'. For example, 'int numbers[5];' declares an integer array named 'numbers' with a size of 5.
- Indexing: The elements of an array are accessed using their index, starting from 0. For example, 'numbers[0]' represents the first element of the 'numbers' array.
- Initialization: Arrays can be initialized at the time of declaration using a comma-separated list of values enclosed in curly braces. For example, 'int numbers[5] = {1, 2, 3, 4, 5};' .
- Accessing Array Elements: Array elements can be accessed and modified using their index. For example, 'numbers[2] = 10; ' assigns the value 10 to the third element of the 'numbers' array.
- Iterating Over Arrays: Loops, such as 'for' or 'while' , are often used to iterate over arrays and perform operations on each element.

Arrays are widely used in various programming tasks, such as storing a list of numbers, characters, or objects, implementing data structures like stacks and queues, and processing large amounts of data efficiently.

Lecture 13: Strings

Practical Question: Write a C program to concatenate two strings entered by the user.

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

#define MAX_SIZE 100

int main() {
    char str1[MAX_SIZE], str2[MAX_SIZE];

    printf("Enter the first string: ");
    gets(str1);

    printf("Enter the second string: ");
    gets(str2);

    strcat(str1, str2);

    printf("Concatenated string: %s", str1);

    return 0;
}
```

Practical Question: Write a C program to reverse a string using a loop.

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

#define MAX_SIZE 100

int main() {
    char str[MAX_SIZE];

    printf("Enter a string: ");
    gets(str);

    int len = strlen(str);

    for (int i = len - 1; i >= 0; i--) {
        printf("%c", str[i]);
    }

    return 0;
}
```

Theory Question: What is a string in C? Explain the functions used to manipulate strings.

In C programming, a string is a sequence of characters stored in an array. The string is terminated by a null character ('\0'), which marks the end of the string. Strings in C are represented as character arrays.

The standard library provides various functions for manipulating strings. Some commonly used string functions are:

- strlen(): Returns the length of a string by counting the number of characters until the null character is encountered.
- strcpy(): Copies one string into another. It takes two arguments, the destination string and the source string.
- strcat(): Concatenates (appends) one string at the end of another. It takes two arguments, the destination string and the source string.
- strcmp(): Compares two strings and returns an integer value based on the comparison. It returns 0 if the strings are equal, a negative value if the first string is less than the second string, and a positive value if the first string is greater than the second string.
- strchr(): Searches for a specific character in a string and returns a pointer to the first occurrence of the character.
- strstr(): Searches for a specific substring in a string and returns a pointer to the first occurrence of the substring.

These functions, along with others, provide a convenient way to perform various operations on strings, such as finding the length, copying, concatenating, comparing, and searching. They are part of the ‘<string.h>’ header file in C.

Lecture 14: Functions: (Part 1)

Practical Question: Write a C program to find the factorial of a number using a user-defined function.

```c
#include <stdio.h>

unsigned long long factorial(int num);

int main() {
    int num;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    if (num < 0) {
        printf("Error: Factorial is not defined for negative numbers.");
    } else {
        unsigned long long fact = factorial(num);
        printf("Factorial of %d = %llu", num, fact);
    }

    return 0;
}
```

```c
unsigned long long factorial(int num) {
    unsigned long long fact = 1;

    for (int i = 1; i <= num; i++) {
        fact *= i;
    }

    return fact;
}
```

Practical Question: Write a C program to check whether a number is prime or not, using a user-defined function.

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

bool isPrime(int num);

int main() {
    int num;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    if (num < 2) {
        printf("Error: Prime numbers are greater than or equal to 2.");
    } else {
        if (isPrime(num)) {
            printf("%d is a prime number.", num);
        } else {
            printf("%d is not a prime number.", num);
        }
    }

    return 0;
}
```

```
bool isPrime(int num) {
    for (int i = 2; i <= num / 2; i++) {
        if (num % i == 0) {
            return false;
        }
    }

    return true;
}
```

Theory Question: Explain the concept of functions in C programming and their advantages.

In C programming, a function is a block of code that performs a specific task. Functions provide a way to break down a program into smaller, modular units, making it easier to read, understand, and maintain. Some key points about functions in C are:

- Function Declaration: A function declaration specifies the name of the function, the return type, and the types of parameters (if any). For example, ‘ int sum(int a, int b); ‘ declares a function named ‘ sum ‘ that takes two integers as parameters and returns an integer.
- Function Definition: A function definition includes the implementation of the function. It consists of the function header (same as the declaration) followed by the function body enclosed in curly braces. For example,

```
int sum(int a, int b) {
    return a + b;
}
```

- Function Call: To use a function, you need to call it. A function call consists of the function name followed by parentheses containing the arguments (if any). For example, int result = sum(2, 3); calls the sum function with arguments 2 and 3 and assigns the result to the variable result.

Advantages of using functions in C programming:

Modularity: Functions promote code modularity by allowing you to break down a complex program into smaller, manageable parts. Each function performs a specific task, making the code easier to read, understand, and maintain.

Reusability: Functions can be reused in different parts of a program or in multiple programs. Once a function is defined, it can be called whenever needed, reducing code duplication and improving code efficiency.

Abstraction: Functions provide a level of abstraction by hiding the implementation details. The calling code only needs to know the function name, its purpose, and the expected inputs and outputs, without knowing the internal workings of the function.

Code Organization: Functions help in organizing code by separating different tasks into separate functions. This promotes code clarity, reduces complexity, and improves code readability.

Debugging and Testing: Functions make it easier to debug and test code. Since functions are modular, you can test each function individually, making it easier to identify and fix issues.

Functions are an essential part of C programming and play a crucial role in creating efficient, readable, and maintainable code.

Lecture 15: Functions: (Part 2)

Practical Question: Write a C program to calculate the power of a number using a recursive function.

```c
#include <stdio.h>

double power(double base, int exponent);

int main() {
    double base;
    int exponent;

    printf("Enter the base number: ");
    scanf("%lf", &base);

    printf("Enter the exponent: ");
    scanf("%d", &exponent);

    double result = power(base, exponent);

    printf("%.2lf ^ %d = %.2lf", base, exponent, result);

    return 0;
}
```

```
double power(double base, int exponent) {
    if (exponent == 0) {
        return 1.0;
    } else if (exponent > 0) {
        return base * power(base, exponent - 1);
    } else {
        return 1.0 / (base * power(base, -exponent - 1));
    }
}
```

Practical Question: Write a C program to find the maximum of three numbers using a user-defined function.

```c
#include <stdio.h>

int maximum(int a, int b, int c);

int main() {
    int num1, num2, num3;

    printf("Enter three numbers: ");
    scanf("%d %d %d", &num1, &num2, &num3);

    int max = maximum(num1, num2, num3);

    printf("Maximum number: %d", max);

    return 0;
}
```

```c
int maximum(int a, int b, int c) {
    int max = a;

    if (b > max) {
        max = b;
    }

    if (c > max) {
        max = c;
    }

    return max;
}
```

Theory Question: Discuss the difference between a function declaration and a function definition in C.

In C programming, a function declaration and a function definition serve different purposes:

- Function Declaration: A function declaration specifies the name of the function, the return type, and the types of parameters (if any). It tells the compiler about the existence and interface of the function without providing the implementation. A function declaration typically appears before its first use in the program, usually in a header file or at the beginning of the source file. For example, int sum(int a, int b); is a function declaration that informs the compiler about a function named sum that takes two integers as parameters and returns an integer.
- Function Definition: A function definition includes the implementation of the function. It consists of the function header (same as the declaration) followed by the function body enclosed in curly braces. The function definition provides the actual code that is executed when the function is called. **For example,**

```
int sum(int a, int b) {
    return a + b;
}
```

The function definition includes the complete implementation of the sum function, specifying how the addition operation is performed.

In summary, a function declaration is used to inform the compiler about the existence and interface of a function, while a function definition provides the implementation of the function.

Lecture 16: Pointers: (Part 1)

Practical Question: Write a C program to swap the values of two variables using pointers.

```c
#include <stdio.h>

void swap(int* a, int* b);

int main() {
    int num1, num2;

    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    printf("Before swapping: num1 = %d, num2 = %d\n", num1, num2);

    swap(&num1, &num2);

    printf("After swapping: num1 = %d, num2 = %d\n", num1, num2);

    return 0;
}

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Practical Question: Write a C program to find the length of a string using a pointer.

```c
#include <stdio.h>

int stringLength(const char* str);

int main() {
    char str[100];

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    int length = stringLength(str);

    printf("Length of the string: %d\n", length);

    return 0;
}

int stringLength(const char* str) {
    int length = 0;

    while (*str != '\0') {
        length++;
        str++;
    }

    return length;
}
```

Theory Question: Explain the concept of pointers in C and their importance in programming.

In C programming, a pointer is a variable that stores the memory address of another variable. Pointers provide a way to indirectly access and manipulate data in memory. Some key points about pointers in C are:

- Declaration: Pointers are declared using the ‘ * ‘ symbol. For example, ‘ int* ptr; ‘ declares a pointer variable named ‘ ptr ‘ that can store the memory address of an integer.
- Initialization: Pointers can be initialized with the address of a variable using the ‘ & ‘ operator. For example, ‘ int num = 10; int* ptr = # ‘ initializes the pointer ‘ ptr ‘ with the address of the ‘ num ‘ variable.
- Dereferencing: Dereferencing a pointer means accessing the value stored at the memory address pointed to by the pointer. The * operator is used for dereferencing. For example, ‘ int value = *ptr; ‘ assigns the value stored at the memory address pointed to by ‘ ptr ‘ to the variable ‘ value ‘.
- Pointer Arithmetic: Pointers can be incremented or decremented to navigate through memory. Pointer arithmetic depends on the data type the pointer is pointing to. For example, ‘ ptr++; ‘ increments the pointer ‘ ptr ‘ by the size of the data type it points to.

Importance: Pointers are important in programming for various reasons:

Dynamic Memory Allocation: Pointers are essential for dynamically allocating memory during runtime using functions like ‘ malloc() ‘ and ‘ free() ‘. This allows for efficient memory management and flexibility.

Passing Parameters: Pointers are used to pass parameters by reference, allowing functions to modify the original variables passed to them.

Data Structures: Pointers are used to implement complex data structures like linked lists, trees, and graphs, where nodes are connected using pointers.

Efficient Manipulation: Pointers provide a way to directly access and manipulate data in memory, enabling efficient algorithms and data processing.

Understanding pointers is crucial for advanced programming in C and is a fundamental concept in memory management and data manipulation.

Lecture 17: Pointers: (Part 2)

Practical Question: Write a C program to pass an array to a function and display its elements using pointers.

```c
#include <stdio.h>

void displayArray(int* arr, int size);

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    displayArray(arr, size);

    return 0;
}

void displayArray(int* arr, int size) {
    printf("Array elements: ");

    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
    }

    printf("\n");
}
```

Practical Question: Write a C program to allocate memory dynamically for an integer and store its value using a pointer.

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

int main() {
    int* ptr;
    ptr = (int*)malloc(sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    int value;

    printf("Enter an integer: ");
    scanf("%d", &value);

    *ptr = value;

    printf("Value stored at the dynamically allocated memory: %d\n", *ptr);

    free(ptr);

    return 0;
}
```

Theory Question: What is the difference between pass-by-value and pass-by-reference in C? Explain with examples.

- Pass-by-Value: In pass-by-value, the function receives a copy of the argument passed to it. Any changes made to the function's parameter do not affect the original argument.

  Example:

```c
#include <stdio.h>

void increment(int num) {
    num++;
}

int main() {
    int num = 5;
    increment(num);
    printf("num: %d\n", num); // Output: num: 5
    return 0;
}
```

In this example, the increment function receives a copy of num, increments the copy, and the original num remains unchanged.

- Pass-by-Reference: In pass-by-reference, the function receives the memory address of the argument passed to it. Any changes made to the function's parameter directly affect the original argument.

  Example:

```c
#include <stdio.h>

void increment(int* numPtr) {
    (*numPtr)++;
}

int main() {
    int num = 5;
    increment(&num);
    printf("num: %d\n", num); // Output: num: 6
    return 0;
}
```

In this example, the ‘ increment ‘ function receives the memory address of ‘ num ‘, dereferences the pointer to access the original ‘ num ‘ , increments it, and the change is reflected in the original ‘ num ‘ variable.

Pass-by-reference is achieved in C by using pointers, where the address of the variable is passed to the function.

Lecture 18: Files

Practical Question: Write a C program to read data from a text file and display it on the console.

```c
#include <stdio.h>

int main() {
    FILE* file;
    char ch;

    file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }

    printf("Data in the file:\n");
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }

    fclose(file);

    return 0;
}
```

Note: Make sure to create a file named "data.txt" in the same directory as your program and populate it with some text data.

Practical Question: Write a C program to write data to a binary file.

```c
#include <stdio.h>

typedef struct {
    int id;
    char name[20];
    float salary;
} Employee;

int main() {
    FILE* file;
    Employee employee;

    file = fopen("employees.bin", "wb");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }
```

```
    printf("Enter employee details:\n");
    printf("ID: ");
    scanf("%d", &employee.id);
    printf("Name: ");
    scanf("%s", employee.name);
    printf("Salary: ");
    scanf("%f", &employee.salary);

    fwrite(&employee, sizeof(Employee), 1, file);

    fclose(file);

    printf("Data written to the file successfully.\n");

    return 0;
}
```

Note: The above program writes a single employee record to a binary file named "employees.bin".

Theory Question: How are files handled in C? Explain the different file modes used in file handling.

In C, files are handled using a FILE pointer and a set of file handling functions from the <stdio.h> library.

To work with a file, the following steps are typically involved:

Declare a FILE pointer variable to hold the reference to the file.

Open the file using the fopen() function, which takes two parameters: the file name and the file mode.

Perform the desired file operations, such as reading from or writing to the file, using appropriate functions like fread(), fwrite(), fscanf(), fprintf(), etc.

Close the file using the fclose() function to release system resources and ensure data integrity.

The file mode specifies the intended operations to be performed on the file. Some commonly used file modes are:

- "r": Read mode. Opens the file for reading. The file must exist.
- "w": Write mode. Opens the file for writing. If the file already exists, its contents are truncated. If the file does not exist, a new file is created.
- "a": Append mode. Opens the file for writing at the end. If the file does not exist, a new file is created.
- "rb", "wb", "ab": Binary read, write, and append modes, respectively. These modes are used for binary file handling.
- "r+", "w+", "a+": Read and write modes. Opens the file for both reading and writing. The file must exist in "r+" mode, while "w+" and "a+" create a new file if it does not exist.

These are just a few examples of file modes available in C. The appropriate file mode depends on the specific requirements of the program.

Lecture 19: Hands-on Practice: Questions: Part 1

Practical Question: Write a C program to find the sum of all even numbers between 1 and 100 using a loop.

```c
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 2; i <= 100; i += 2) {
        sum += i;
    }

    printf("Sum of even numbers between 1 and 100: %d\n", sum);

    return 0;
}
```

Practical Question: Write a C program to reverse the digits of a number using a while loop.

```c
#include <stdio.h>

int main() {
    int number, reversedNumber = 0;

    printf("Enter a number: ");
    scanf("%d", &number);

    while (number != 0) {
        int remainder = number % 10;
        reversedNumber = reversedNumber * 10 + remainder;
        number /= 10;
    }

    printf("Reversed number: %d\n", reversedNumber);

    return 0;
}
```

Theory Question: Explain the concept of recursion and its advantages in programming.

Recursion is a programming technique in which a function calls itself directly or indirectly to solve a problem. It involves breaking down a complex problem into smaller, simpler instances of the same problem. Each recursive call solves a smaller part of the problem until a base case is reached, which terminates the recursion.

Advantages of recursion in programming include:

Simplicity: Recursion allows complex problems to be expressed in a simpler and more intuitive manner. It follows the principle of "divide and conquer," where a problem is divided into smaller subproblems until they can be easily solved.

Readability: Recursive code often mirrors the problem-solving approach and can be easier to understand and read compared to iterative solutions. It can be more concise and closer to the problem's natural description.

Code Reusability: Recursive functions can be reused to solve similar problems. Once a recursive function is implemented, it can be called with different inputs to solve related problems.

Solving Recursive Problems: Recursive algorithms are often well-suited for solving problems that exhibit self-similar or recursive properties. Examples include searching and traversing tree-like data structures, solving puzzles, and generating permutations or combinations.

However, recursion also has some potential drawbacks, such as increased memory usage due to multiple function calls on the stack and the risk of infinite recursion if not properly implemented with base cases.

Lecture 20: Hands-on Practice: Questions: Part 2

Practical Question: Write a C program to check whether a string is a palindrome or not.

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

int isPalindrome(const char* str) {
    int len = strlen(str);
    int i = 0;
    int j = len - 1;

    while (i < j) {
        if (str[i] != str[j]) {
            return 0; // Not a palindrome
        }
        i++;
        j--;
    }

    return 1; // Palindrome
}
```

```c
int main() {
    char str[100];

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

    if (isPalindrome(str)) {
        printf("%s is a palindrome.\n", str);
    } else {
        printf("%s is not a palindrome.\n", str);
    }

    return 0;
}
```

Practical Question: Write a C program to sort an array of integers in ascending order using a user-defined function.

```c
#include <stdio.h>

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[100];
    int size;

    printf("Enter the number of elements: ");
    scanf("%d", &size);

    printf("Enter the elements: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    bubbleSort(arr, size);

    printf("Sorted array in ascending order: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Theory Question: What is the purpose of header files in C programming? Provide examples of commonly used header files.

Header files in C programming are used to declare the prototypes of functions, definitions of constants, and declarations of data structures that are defined in other source files. They provide the necessary information about various libraries, modules, and functionality that a program depends on.

Commonly used header files in C include:

- <stdio.h>: Contains input/output functions like printf() and scanf().
- <stdlib.h>: Provides functions for memory allocation, process control, and conversion functions like atoi() and malloc().
- <string.h>: Includes string manipulation functions like strlen(), strcpy(), and strcmp().
- <math.h>: Contains mathematical functions like sqrt(), sin(), and pow().
- <time.h>: Provides functions for working with dates and time.
- <ctype.h>: Includes functions for character handling, such as isalpha(), isdigit(), and toupper().

These are just a few examples of commonly used header files in C. There are many more header files available, each serving a specific purpose.

Section 2 : Programming with C++

Lecture 22: Introduction to C++

Practical Question: Write a C++ program to display "Hello, World!" on the console.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Practical Question: Write a C++ program to take two numbers as input from the user and print their sum.

```cpp
#include <iostream>

int main() {
    int num1, num2, sum;

    std::cout << "Enter the first number: ";
    std::cin >> num1;

    std::cout << "Enter the second number: ";
    std::cin >> num2;

    sum = num1 + num2;

    std::cout << "Sum: " << sum << std::endl;

    return 0;
}
```

Theory Question: Explain the key features and advantages of C++ programming.

C++ is a powerful and versatile programming language that offers several key features and advantages, including:

Object-Oriented Programming (OOP): C++ supports the OOP paradigm, allowing you to organize your code into objects that encapsulate data and behavior. This promotes modularity, reusability, and easier maintenance of code.

Efficiency and Performance: C++ is known for its efficiency and performance. It provides low-level control over system resources and allows direct memory manipulation, making it suitable for system-level programming and performance-critical applications.

Standard Template Library (STL): C++ provides a rich set of libraries, including the STL, which offers various data structures and algorithms. The STL containers (such as vectors, lists, and maps) and algorithms (such as sorting and searching) provide a convenient way to work with data.

Platform Independence: C++ code can be compiled and executed on different platforms, making it a portable language. It allows you to write code once and run it on multiple systems without significant modifications.

Strong Community and Wide Adoption: C++ has a large and active community of developers, which means there is extensive documentation, resources, and support available. It is widely used in industries such as game development, embedded systems, high-performance computing, and more.

Lecture 23: Variables

Practical Question: Write a C++ program to declare and initialize variables of different data types (int, float, char) and display their values.

```cpp
#include <iostream>

int main() {
    int num = 10;
    float pi = 3.14;
    char letter = 'A';

    std::cout << "Integer: " << num << std::endl;
    std::cout << "Float: " << pi << std::endl;
    std::cout << "Character: " << letter << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to swap the values of two variables without using a temporary variable.

```cpp
#include <iostream>

int main() {
    int a = 10, b = 20;

    std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;

    a = a + b;
    b = a - b;
    a = a - b;

    std::cout << "After swap: a = " << a << ", b = " << b << std::endl;

    return 0;
}
```

Theory Question: Discuss the scope and lifetime of variables in C++.

Scope refers to the region in a program where a variable is defined and can be accessed. In C++, variables can have different scopes:

Global Scope: Variables declared outside any function or block have global scope. They can be accessed from anywhere in the program.

Local Scope: Variables declared inside a function or block have local scope. They are only accessible within that function or block.

Function Parameters: Parameters of a function have scope within that function. They are local to the function and can be accessed only within the function's body.

The lifetime of a variable determines the period during which it exists in memory:

Automatic Storage Duration: Variables declared inside a function or block have automatic storage duration. They are created when the function or block is entered and destroyed when it is exited.

Static Storage Duration: Variables declared with the static keyword have static storage duration. They are created when the program starts and exist throughout the program's execution. They retain their values between function calls.

Dynamic Storage Duration: Variables allocated dynamically using the new keyword have dynamic storage duration. They are created and destroyed explicitly using new and delete operators. Their lifetime can be controlled manually.

Lecture 24: Data Types

Practical Question: Write a C++ program to calculate the area of a rectangle using user-input values for length and width.

```cpp
#include <iostream>

int main() {
    float length, width, area;

    std::cout << "Enter the length: ";
    std::cin >> length;

    std::cout << "Enter the width: ";
    std::cin >> width;

    area = length * width;

    std::cout << "Area: " << area << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to convert temperature from Celsius to Fahrenheit.

```cpp
#include <iostream>

int main() {
    float celsius, fahrenheit;

    std::cout << "Enter temperature in Celsius: ";
    std::cin >> celsius;

    fahrenheit = (celsius * 9 / 5) + 32;

    std::cout << "Temperature in Fahrenheit: " << fahrenheit << std::endl;

    return 0;
}
```

Theory Question: Explain the different data types available in C++.

C++ provides several built-in data types, including:

Fundamental Data Types:

- int: Used to represent integers (whole numbers).
- float: Used to represent floating-point numbers with single precision.
- double: Used to represent floating-point numbers with double precision.
- char: Used to represent single characters.
- bool: Used to represent Boolean values (true or false).

Derived Data Types:

- Array: A collection of elements of the same data type.
- Pointer: A variable that stores the memory address of another variable.
- Reference: An alias for an existing variable.
- Function: A named sequence of statements that can be called and executed.

User-Defined Data Types:

- Structure: A collection of variables of different data types grouped together under a single name.
- Class: An extension of structures that also includes member functions and data access control.

Enumeration:

A user-defined data type that consists of a set of named constants.

These data types allow programmers to choose the appropriate type for storing and manipulating different kinds of data in C++ programs.

Lecture 25: User-Input

Practical Question: Write a C++ program to take user-input for a student's name, roll number, and marks in three subjects, and display the details.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name;
    int rollNumber;
    float marks1, marks2, marks3;

    std::cout << "Enter student's name: ";
    std::getline(std::cin, name);

    std::cout << "Enter roll number: ";
    std::cin >> rollNumber;

    std::cout << "Enter marks in three subjects: ";
    std::cin >> marks1 >> marks2 >> marks3;

    std::cout << "Name: " << name << std::endl;
    std::cout << "Roll Number: " << rollNumber << std::endl;
    std::cout << "Marks: " << marks1 << ", " << marks2 << ", " << marks3 <<

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

```cpp
std::cout << "Name: " << name << std::endl;

std::cout << "Roll Number: " << rollNumber << std::endl;

std::cout << "Marks: " << marks1 << ", " << marks2 << ", " << marks3 << std::endl;
```

Practical Question: Write a C++ program to take user-input for two numbers and perform arithmetic operations on them.

```cpp
#include <iostream>

int main() {
    int num1, num2;

    std::cout << "Enter the first number: ";
    std::cin >> num1;

    std::cout << "Enter the second number: ";
    std::cin >> num2;

    int sum = num1 + num2;
    int difference = num1 - num2;
    int product = num1 * num2;
    int quotient = num1 / num2;

    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;
    std::cout << "Product: " << product << std::endl;
    std::cout << "Quotient: " << quotient << std::endl;

    return 0;
}
```

Theory Question: How can user input be obtained in C++? Explain with examples.

User input can be obtained in C++ using the ‘ std::cin ‘ object from the ‘ <iostream> ‘ library. Here are some examples of obtaining user input:

Single Value Input:

```cpp
#include <iostream>

int main() {
    int num;

    std::cout << "Enter a number: ";
    std::cin >> num;

    std::cout << "You entered: " << num << std::endl;

    return 0;
}
```

Multiple Value Input:

```cpp
#include <iostream>

int main() {
    int num1, num2;

    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;

    std::cout << "You entered: " << num1 << " and " << num2 << std::endl;

    return 0;
}
```

String Input:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name;

    std::cout << "Enter your name: ";
    std::getline(std::cin, name);

    std::cout << "Hello, " << name << "!" << std::endl;

    return 0;
}
```

Note: The std::getline() function is used to read a line of input including spaces.

Mixed Data Types Input:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name;
    int age;
    float height;

    std::cout << "Enter your name: ";
    std::getline(std::cin, name);

    std::cout << "Enter your age: ";
    std::cin >> age;

    std::cout << "Enter your height (in meters): ";
    std::cin >> height;

    std::cout << "Name: " << name << std::endl;
    std::cout << "Age: " << age << std::endl;
    std::cout << "Height: " << height << " meters" << std::endl;

    return 0;
}
```

In all these examples, the ‘ >> ‘ operator is used with ‘ std::cin ‘ to read input values into variables. The ‘ std::getline() ‘ function is used for reading strings with spaces.

Lecture 26: Operators

Practical Question: Write a C++ program to perform arithmetic operations on two numbers and display the results.

```cpp
#include <iostream>

int main() {
    int num1, num2;

    std::cout << "Enter the first number: ";
    std::cin >> num1;

    std::cout << "Enter the second number: ";
    std::cin >> num2;

    int sum = num1 + num2;
    int difference = num1 - num2;
    int product = num1 * num2;
    int quotient = num1 / num2;
    int remainder = num1 % num2;

    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;
    std::cout << "Product: " << product << std::endl;
    std::cout << "Quotient: " << quotient << std::endl;
    std::cout << "Remainder: " << remainder << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to calculate the area of a triangle using the values of base and height entered by the user.

```cpp
#include <iostream>

int main() {
    float base, height;

    std::cout << "Enter the base of the triangle: ";
    std::cin >> base;

    std::cout << "Enter the height of the triangle: ";
    std::cin >> height;

    float area = 0.5 * base * height;

    std::cout << "Area of the triangle: " << area << std::endl;

    return 0;
}
```

Theory Question: Discuss the different types of operators in C++.

C++ provides various types of operators that can be used to perform different operations. Here are the main types of operators in C++:

Arithmetic Operators: Arithmetic operators are used to perform mathematical operations. Examples include addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

Relational Operators: Relational operators are used to compare values. Examples include equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

Logical Operators: Logical operators are used to perform logical operations. Examples include logical AND (&&), logical OR (||), and logical NOT (!).

Assignment Operators: Assignment operators are used to assign values to variables. Examples include assignment (=), addition and assignment (+=), subtraction and assignment (-=), multiplication and assignment (*=), and division and assignment (/=).

Increment and Decrement Operators: Increment and decrement operators are used to increase or decrease the value of a variable by 1. Examples include increment (++) and decrement (--).

Bitwise Operators: Bitwise operators are used to perform bitwise operations on binary representations of numbers. Examples include bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise complement (~), left shift (<<), and right shift (>>).

Ternary Operator: The ternary operator (?:) is a conditional operator that allows you to write compact conditional expressions.

Member Access Operators: Member access operators are used to access members of a class or structure. Examples include dot operator (.) and arrow operator (->).

These operators provide a way to perform different operations on variables and values in C++ programming.

Lecture 27: Control Structure

Practical Question: Write a C++ program to check whether a number entered by the user is positive, negative, or zero.

```cpp
#include <iostream>

int main() {
    int number;

    std::cout << "Enter a number: ";
    std::cin >> number;

    if (number > 0) {
        std::cout << "The number is positive." << std::endl;
    } else if (number < 0) {
        std::cout << "The number is negative." << std::endl;
    } else {
        std::cout << "The number is zero." << std::endl;
    }

    return 0;
}
```

Practical Question: Write a C++ program to determine whether a character entered by the user is a vowel or consonant.

```cpp
#include <iostream>

int main() {
    char character;

    std::cout << "Enter a character: ";
    std::cin >> character;

    if (character == 'a' || character == 'e' || character == 'i' || characte
        character == 'A' || character == 'E' || character == 'I' || characte
        std::cout << "The character is a vowel." << std::endl;
    } else {
        std::cout << "The character is a consonant." << std::endl;
    }

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

```cpp
if (character == 'a' || character == 'e' || character == 'i' || character == 'o' || character == 'u' ||
    character == 'A' || character == 'E' || character == 'I' || character == 'O' || character == 'U') {
    std::cout << "The character is a vowel." << std::endl;
} else {
    std::cout << "The character is a consonant." << std::endl;
}
```

Theory Question: Explain the concept of control structures in C++.

Control structures in C++ are used to control the flow of program execution. They allow you to make decisions and repeat certain blocks of code based on specified conditions. The main control structures in C++ are:

If-else: The if-else statement allows you to execute a block of code if a certain condition is true, and another block of code if the condition is false.

Switch: The switch statement is used to select one of many code blocks to be executed based on the value of a variable or an expression.

Loops: Loops are used to repeat a block of code multiple times. There are three types of loops in C++:

- while loop: It repeats a block of code as long as a specified condition is true.
- do-while loop: It repeats a block of code at least once and then continues to repeat it as long as a specified condition is true.
- for loop: It repeats a block of code for a specified number of times, with a defined initialization, condition, and increment or decrement.

Break: The break statement is used to exit the current loop or switch statement and continue with the next statement after the loop or switch.

Continue: The continue statement is used to skip the current iteration of a loop and continue with the next iteration.

Control structures allow you to write programs with decision-making capabilities and the ability to repeat code, making your programs more flexible and efficient.

Lecture 28: Switch Statements

Practical Question: Write a C++ program to display the name of a day of the week based on the user-inputted number (1-7).

```
#include <iostream>

int main() {
    int dayNumber;

    std::cout << "Enter a number (1-7): ";
    std::cin >> dayNumber;

    switch (dayNumber) {
        case 1:
            std::cout << "Sunday" << std::endl;
            break;
        case 2:
            std::cout << "Monday" << std::endl;
            break;
        case 3:
            std::cout << "Tuesday" << std::endl;
            break;
        case 4:
            std::cout << "Wednesday" << std::endl;
            break;
```

```
        case 5:
            std::cout << "Thursday" << std::endl;
            break;
        case 6:
            std::cout << "Friday" << std::endl;
            break;
        case 7:
            std::cout << "Saturday" << std::endl;
            break;
        default:
            std::cout << "Invalid input. Please enter a number between 1 and
            break;
    }

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

```
default:
  std::cout << "Invalid input. Please enter a number between 1 and 7." << std::endl;
  break;
```

Practical Question: Write a C++ program to calculate the grade of a student based on their percentage using switch statements.

```cpp
#include <iostream>

int main() {
    float percentage;

    std::cout << "Enter the percentage: ";
    std::cin >> percentage;

    int grade;

    if (percentage >= 90) {
        grade = 1;
    } else if (percentage >= 80) {
        grade = 2;
    } else if (percentage >= 70) {
        grade = 3;
    } else if (percentage >= 60) {
        grade = 4;
    } else if (percentage >= 50) {
        grade = 5;
    } else {
        grade = 6;
    }
```

```
    switch (grade) {
        case 1:
            std::cout << "Grade: A" << std::endl;
            break;
        case 2:
            std::cout << "Grade: B" << std::endl;
            break;
        case 3:
            std::cout << "Grade: C" << std::endl;
            break;
        case 4:
            std::cout << "Grade: D" << std::endl;
            break;
        case 5:
            std::cout << "Grade: E" << std::endl;
            break;
        case 6:
            std::cout << "Grade: F" << std::endl;
            break;
    }

    return 0;
}
```

Theory Question: What is the purpose of using switch statements in C++? Provide an example.

The switch statement in C++ provides a way to select one of many code blocks to be executed based on the value of a variable or an expression. It simplifies the process of writing multiple if-else statements when you have a fixed set of values to compare.

For example, consider a scenario where you want to perform different actions based on the day of the week. Instead of writing multiple if-else statements, you can use a switch statement to achieve the same result in a more concise way:

```
#include <iostream>

int main() {
    int dayNumber;

    std::cout << "Enter a number (1-7): ";
    std::cin >> dayNumber;

    switch (dayNumber) {
        case 1:
            std::cout << "Sunday" << std::endl;
            break;
        case 2:
            std::cout << "Monday" << std::endl;
            break;
```

```
        case 3:
            std::cout << "Tuesday" << std::endl;
            break;
        case 4:
            std::cout << "Wednesday" << std::endl;
            break;
        case 5:
            std::cout << "Thursday" << std::endl;
            break;
        case 6:
            std::cout << "Friday" << std::endl;
            break;
        case 7:
            std::cout << "Saturday" << std::endl;
            break;
        default:
            std::cout << "Invalid input. Please enter a number between 1 and
            break;
    }

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

```
    default:
      std::cout << "Invalid input. Please enter a number between 1 and 7." << std::endl;
      break;
```

In this example, based on the value of ‘ dayNumber ‘, the corresponding day of the week is displayed. If the input is not within the range of 1-7, the default case is executed and an error message is displayed.

Switch statements provide a more organized and readable way to handle multiple cases compared to a series of if-else statements, especially when you have a fixed number of cases to consider.

Lecture 29: Loops

Practical Question: Write a C++ program to print all even numbers between 1 and 50 using a loop.

```cpp
#include <iostream>

int main() {
    for (int i = 2; i <= 50; i += 2) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to calculate the factorial of a number using a loop.

```cpp
#include <iostream>

int main() {
    int number;
    int factorial = 1;

    std::cout << "Enter a number: ";
    std::cin >> number;

    for (int i = 1; i <= number; i++) {
        factorial *= i;
    }

    std::cout << "Factorial of " << number << " is: " << factorial << std::en

    return 0;
}
```

Theory Question: Explain the structure and working of loops in C++.

Loops in C++ are used to repeatedly execute a block of code as long as a certain condition is met. They provide a way to automate repetitive tasks and iterate over a sequence of values.

There are three types of loops in C++:

while loop: It executes a block of code repeatedly as long as the specified condition is true. The structure of a while loop is as follows:

```
while (condition) {
    // Code to be executed
    // Iteration statement
}
```

The condition is evaluated before each iteration, and if it becomes false, the loop terminates. If the condition is true, the code block is executed, and then the iteration statement is executed to modify the loop control variable or condition.

do-while loop: It executes a block of code at least once, and then repeatedly as long as the specified condition is true. The structure of a do-while loop is as follows:

```
do {
    // Code to be executed
    // Iteration statement
} while (condition);
```

The code block is executed first, and then the condition is checked. If the condition is true, the loop continues to execute. If the condition is false, the loop terminates.

for loop: It is used to execute a block of code for a fixed number of times. The structure of a for loop is as follows:

```
for (initialization; condition; iteration) {
    // Code to be executed
}
```

The initialization statement is executed before the loop starts. The condition is checked before each iteration, and if it becomes false, the loop terminates. The code block is executed, and then the iteration statement is executed to modify the loop control variable or condition.

The working of loops involves the repetition of a code block until a certain condition is met. The loop control variable or condition is updated during each iteration to control the loop's execution. It is important to ensure that the loop condition will eventually become false to avoid infinite loops.

Loops are powerful constructs in programming that allow you to automate tasks and process data efficiently. They provide flexibility in handling repetitive operations and iterating over data structures.

Lecture 30: Questions on Loops

Practical Question: Write a C++ program to display a pyramid pattern using nested loops.

```cpp
#include <iostream>

int main() {
    int rows;

    std::cout << "Enter the number of rows: ";
    std::cin >> rows;

    for (int i = 1; i <= rows; i++) {
        // Print spaces
        for (int j = 1; j <= rows - i; j++) {
            std::cout << " ";
        }

        // Print asterisks
        for (int k = 1; k <= 2 * i - 1; k++) {
            std::cout << "*";
        }

        std::cout << std::endl;
    }

    return 0;
}
```

Practical Question: Write a C++ program to find the sum of digits of a number using a loop.

```cpp
#include <iostream>

int main() {
    int number;
    int sum = 0;

    std::cout << "Enter a number: ";
    std::cin >> number;

    int temp = number;

    while (temp != 0) {
        int digit = temp % 10;
        sum += digit;
        temp /= 10;
    }

    std::cout << "Sum of digits of " << number << " is: " << sum << std::endl;

    return 0;
}
```

Theory Question: Compare and contrast the different types of loops in C++.

In C++, there are three types of loops: while, do-while, and for. Here's a comparison of these loop types:

while loop:

- The condition is checked before each iteration.
- It may not execute the loop body if the condition is false from the start.
- It is suitable when the number of iterations is not known beforehand or when the loop may not execute at all.

do-while loop:

- The condition is checked after each iteration.
- It always executes the loop body at least once, even if the condition is false.
- It is useful when you need to execute the loop body at least once, such as taking user input validation.

for loop:

- It consists of an initialization, condition, and iteration statement.
- The initialization is executed only once at the start.
- The condition is checked before each iteration.
- The iteration statement is executed after each iteration.
- It is suitable when the number of iterations is known beforehand or when iterating over a specific range.

The choice of loop type depends on the specific requirements of the program. While and do-while loops are often used when the number of iterations is uncertain or depends on runtime conditions. For loops are commonly used when the number of iterations is known or when iterating over a specific range.

All three loop types provide ways to control the flow of execution based on certain conditions, allowing for repetition and automation of tasks. The appropriate loop type should be chosen based on the desired behavior and requirements of the program.

Lecture 31: Break and Continue

Practical Question: Write a C++ program to print the numbers from 1 to 10, skipping the number 7 using the continue statement.

```cpp
#include <iostream>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 7) {
            continue; // Skip number 7 and continue to the next iteration
        }
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to find the first prime number between 1 and 100 using the break statement.

```cpp
#include <iostream>

bool isPrime(int number) {
    if (number < 2) {
        return false;
    }
    for (int i = 2; i * i <= number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}

int main() {
    for (int i = 2; i <= 100; i++) {
        if (isPrime(i)) {
            std::cout << "The first prime number between 1 and 100 is: " <<
            break; // Stop the loop after finding the first prime number
        }
    }

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

std::cout << "The first prime number between 1 and 100 is: " << i << std::endl;

Theory Question: Explain the use of the break and continue statements in C++.

The **break** statement is used to immediately exit the innermost loop or switch statement. When encountered, the break statement terminates the loop execution and transfers control to the statement following the loop or switch. It is commonly used to exit a loop early based on certain conditions or to stop the execution of a switch statement.

The **continue** statement is used to skip the rest of the current iteration and move to the next iteration of a loop. When encountered, the continue statement immediately jumps to the loop's increment or update statement. It is used to bypass the remaining code within the loop body and proceed with the next iteration.

The break statement is useful when you want to exit a loop prematurely based on specific conditions. For example, you may want to stop a loop when a certain value is found or when an error condition occurs. It allows you to break out of the loop and resume execution from the next statement after the loop.

The continue statement, on the other hand, is used to skip certain iterations within a loop. It can be used when you want to exclude certain values or conditions from being processed within the loop body. The continue statement allows you to move directly to the next iteration, ignoring the remaining code in the current iteration.

Both break and continue statements provide control over the flow of execution within loops. They can be powerful tools to control the behaviour of loops based on specific conditions, allowing for more flexible and efficient code.

Lecture 32: Array

Practical Question: Write a C++ program to find the largest element in an array.

```cpp
#include <iostream>

int main() {
    const int size = 5;
    int arr[size];

    std::cout << "Enter " << size << " integers:" << std::endl;
    for (int i = 0; i < size; i++) {
        std::cin >> arr[i];
    }

    int largest = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }

    std::cout << "The largest element in the array is: " << largest << std::er

    return 0;
}
```

Practical Question: Write a C++ program to calculate the sum of elements in an array.

```cpp
#include <iostream>

int main() {
    const int size = 5;
    int arr[size];

    std::cout << "Enter " << size << " integers:" << std::endl;
    for (int i = 0; i < size; i++) {
        std::cin >> arr[i];
    }

    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }

    std::cout << "The sum of elements in the array is: " << sum << std::endl;

    return 0;
}
```

Theory Question: Discuss the concept of arrays in C++ and their importance in programming.

In C++, an array is a data structure that allows you to store a fixed-size sequence of elements of the same data type. Arrays provide a way to group related data together, making it easier to manage and manipulate collections of values.

Key points about arrays in C++:

Declaration: An array is declared by specifying the data type of its elements and its size. For example, int arr[5]; declares an integer array arr with a size of 5.

Indexing: Array elements are accessed using their index, which represents their position in the array. The index starts from 0 for the first element and goes up to size - 1 for the last element.

Fixed Size: Arrays have a fixed size, which is determined at the time of declaration and cannot be changed during runtime. The size is specified either explicitly or implicitly based on the number of elements.

Contiguous Memory: Array elements are stored in contiguous memory locations. This allows for efficient access to elements using pointer arithmetic.

Arrays are important in programming for several reasons:

Data Storage: Arrays provide a convenient way to store and organize large amounts of related data. They allow you to group similar data elements together, making the code more organized and readable.

Random Access: Arrays allow for random access to elements using their index. This means you can directly access any element in the array, regardless of its position, by specifying the index.

Efficient Iteration: Arrays facilitate efficient iteration over the elements using loops. You can easily perform operations on each element by iterating over the array sequentially.

Algorithmic Solutions: Arrays are widely used in algorithmic problem-solving. Many algorithms and data structures are built around arrays, such as sorting, searching, and dynamic programming.

Arrays are a fundamental concept in programming and are used extensively in various applications. They provide a basic building block for data storage and manipulation, forming the foundation for more complex data structures and algorithms.

Lecture 33: String

Practical Question: Write a C++ program to concatenate two strings entered by the user.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str1, str2;

    std::cout << "Enter the first string: ";
    std::getline(std::cin, str1);

    std::cout << "Enter the second string: ";
    std::getline(std::cin, str2);

    std::string concatenated = str1 + str2;

    std::cout << "Concatenated string: " << concatenated << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to reverse a string using a loop.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str;

    std::cout << "Enter a string: ";
    std::getline(std::cin, str);

    std::string reversed;

    for (int i = str.length() - 1; i >= 0; i--) {
        reversed += str[i];
    }

    std::cout << "Reversed string: " << reversed << std::endl;

    return 0;
}
```

Theory Question: What is a string in C++? Explain the functions used to manipulate strings.

In C++, a string is a sequence of characters. It is represented by the std::string class from the C++ Standard Library. The std::string class provides various member functions and overloaded operators to manipulate strings efficiently.

Some commonly used functions to manipulate strings in C++ include:

length(): Returns the length or number of characters in a string.

size(): Equivalent to length(), returns the length of the string.

append(): Concatenates or appends a string or character to the end of the current string.

substr(): Returns a substring of the string based on the specified position and length.

find(): Searches for a substring within the string and returns its position or index.

replace(): Replaces a portion of the string with another string or character.

compare(): Compares two strings and returns 0 if they are equal, a negative value if the first string is less, and a positive value if the first string is greater.

getline(): Reads a line of text from the input stream, including whitespace, and stores it as a string.

These functions, along with many others, provide powerful capabilities for string manipulation in C++. They allow you to perform operations such as concatenation, substring extraction, searching, and replacement. Additionally, the overloaded operators for + (concatenation), == (equality comparison), and [] (accessing individual characters) make string manipulation more convenient and intuitive.

The std::string class and its associated functions simplify string handling in C++ and provide a wide range of functionalities for working with textual data.

Lecture 34: Functions

Practical Question: Write a C++ program to find the factorial of a number using a user-defined function.

```cpp
#include <iostream>

// Function to calculate the factorial of a number
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;

    std::cout << "Enter a number: ";
    std::cin >> num;

    int fact = factorial(num);

    std::cout << "Factorial of " << num << " is: " << fact << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to check whether a number is prime or not using a user-defined function.

```cpp
#include <iostream>

// Function to check whether a number is prime
bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }

    for (int i = 2; i <= n / 2; i++) {
        if (n % i == 0) {
            return false;
        }
    }

    return true;
}
```

```cpp
int main() {
    int num;

    std::cout << "Enter a number: ";
    std::cin >> num;

    if (isPrime(num)) {
        std::cout << num << " is a prime number." << std::endl;
    } else {
        std::cout << num << " is not a prime number." << std::endl;
    }

    return 0;
}
```

Theory Question: Explain the concept of functions in C++ programming and their advantages.

In C++, a function is a self-contained block of code that performs a specific task. Functions provide a way to organize code into modular and reusable units, enhancing the readability, maintainability, and efficiency of programs.

Key points about functions in C++:

Function Declaration: A function is declared by specifying its return type, name, and parameter list (if any). For example, int add(int a, int b); declares a function named add that takes two integers as parameters and returns an integer.

Function Definition: The actual implementation of a function is provided in its definition. It includes the function body, which contains the statements executed when the function is called.

Function Call: To use a function, it must be called by its name followed by parentheses. Arguments (values) can be passed to the function if it expects parameters.

Return Statement: A function may return a value using the return statement. The return type in the function declaration specifies the type of value returned by the function.

Advantages of using functions in C++:

Modularity: Functions allow code to be organized into smaller, self-contained units. This promotes code reusability and makes programs easier to understand and maintain.

Code Reusability: Functions can be reused in multiple parts of a program or in different programs altogether. Once a function is defined, it can be called as many times as needed.

Abstraction: Functions abstract away complex operations by encapsulating them within a single unit. This simplifies the overall program structure and allows for a higher-level view of the code.

Code Readability: Functions improve code readability by dividing complex tasks into smaller, more manageable parts. Each function performs a specific task, making it easier to understand and debug the code.

Code Efficiency: Functions help eliminate code duplication by promoting the use of reusable code blocks. This reduces the overall size of the program and improves execution speed.

Ease of Testing: Functions can be tested individually, making it easier to identify and fix errors. Unit testing becomes simpler when functions are well-defined and isolated.

By utilizing functions effectively, programmers can create modular, reusable, and maintainable code, resulting in efficient and robust software systems.

Lecture 35: Pointers

Practical Question: Write a C++ program to swap the values of two variables using pointers.

```cpp
#include <iostream>

// Function to swap the values of two variables using pointers
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int num1, num2;

    std::cout << "Enter the first number: ";
    std::cin >> num1;

    std::cout << "Enter the second number: ";
    std::cin >> num2;

    std::cout << "Before swapping - num1: " << num1 << ", num2: " << num2 <<

    // Pass the addresses of the variables to the swap function
    swap(&num1, &num2);

    std::cout << "After swapping - num1: " << num1 << ", num2: " << num2 <<

    return 0;
}
```

"Some of the codes are cut off in the screenshot. They have been written below for your reference. Please take them into consideration."

```
  std::cout << "Before swapping - num1: " << num1 << ", num2: " << num2 << std::endl;

   // Pass the addresses of the variables to the swap function

   swap(&num1, &num2);

   std::cout << "After swapping - num1: " << num1 << ", num2: " << num2 << std::endl;

   return 0;

}
```

Practical Question: Write a C++ program to find the length of a string using a pointer.

```cpp
#include <iostream>

// Function to calculate the length of a string using a pointer
int stringLength(const char* str) {
    int length = 0;

    // Iterate through the string until the null character is encountered
    while (*str != '\0') {
        length++;
        str++;
    }

    return length;
}

int main() {
    const char* str = "Hello, World!";

    int length = stringLength(str);

    std::cout << "Length of the string: " << length << std::endl;

    return 0;
}
```

Theory Question: Explain the concept of pointers in C++ and their importance in programming.

In C++, a pointer is a variable that stores the memory address of another variable. Pointers play a crucial role in programming as they allow for dynamic memory allocation, efficient manipulation of data structures, and interaction with functions.

Key points about pointers in C++:

Declaration: Pointers are declared by specifying the data type they point to, followed by an asterisk (*). For example, int* ptr; declares a pointer to an integer.

Initialization: Pointers can be initialized with the memory address of a variable using the address-of operator (&). For example, int num = 10; int* ptr = # initializes ptr with the address of num.

Dereferencing: Dereferencing a pointer means accessing the value stored at the memory address it points to. It is done using the dereference operator (*). For example, int value = *ptr; retrieves the value stored at the memory address pointed to by ptr.

Null Pointers: Pointers can have a special value called nullptr, which represents a null or empty memory address. It is used to indicate that the pointer is not pointing to any valid memory location.

Dynamic Memory Allocation: Pointers are commonly used to allocate memory dynamically using the new operator. This allows the creation of variables and data structures at runtime, enabling flexibility in memory management.

Pointer Arithmetic: Pointers support arithmetic operations like addition and subtraction, which can be used to traverse arrays, manipulate data structures, and implement algorithms efficiently.

Passing Pointers to Functions: Pointers can be passed as function arguments to allow for indirect access and modification of variables. This enables functions to work with the original data, rather than making copies.

Pointers are powerful tools in C++ programming that provide direct memory access and manipulation capabilities. They allow for efficient memory management, enable interaction with low-level systems, and facilitate the implementation of advanced data structures and algorithms. However, they require careful handling to avoid common pitfalls such as null references and memory leaks.

Lecture 36: Structure & Unions

Practical Question: Write a C++ program to define a structure representing a student and display its details.

```cpp
#include <iostream>
#include <string>

// Define a structure representing a student
struct Student {
    std::string name;
    int rollNumber;
    int age;
    float averageScore;
};

int main() {
    // Create an instance of the Student structure
    Student student1;

    // Set the values for the student1 instance
    student1.name = "John Doe";
    student1.rollNumber = 101;
    student1.age = 18;
    student1.averageScore = 85.5;
```

```cpp
    // Display the details of student1
    std::cout << "Student Details" << std::endl;
    std::cout << "Name: " << student1.name << std::endl;
    std::cout << "Roll Number: " << student1.rollNumber << std::endl;
    std::cout << "Age: " << student1.age << std::endl;
    std::cout << "Average Score: " << student1.averageScore << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to define a union representing a person's name and display the name in different formats.

```cpp
#include <iostream>
#include <string>

// Define a union representing a person's name
union PersonName {
    std::string fullName;
    struct {
        std::string firstName;
        std::string lastName;
    } splitName;
};

int main() {
    // Create an instance of the PersonName union
    PersonName personName;

    // Set the value for the fullName member of the union
    personName.fullName = "John Doe";

    // Display the name in full format
    std::cout << "Full Name: " << personName.fullName << std::endl;

    // Display the name in split format
    std::cout << "First Name: " << personName.splitName.firstName << std::endl
    std::cout << "Last Name: " << personName.splitName.lastName << std::endl;

    return 0;
}
```

Theory Question: Discuss the use of structures and unions in C++.

Structures and unions are used in C++ to create custom data types that can hold multiple related variables.

Structures:

- A structure is a composite data type that allows you to group variables of different data types under a single name.
- It provides a way to represent a real-world entity or a collection of related data.
- Structure members can be accessed using the dot (.) operator.
- Structures can be passed as function arguments and returned from functions.
- They are often used to represent complex data structures, such as records, objects, or data packets.

Unions:

- A union is a special data type that enables you to store different data types in the same memory location.
- Unlike structures, all members of a union share the same memory space.
- Only one member of a union can be accessed at a time.
- The memory allocated for a union is equal to the size of its largest member.
- Unions are useful when you want to save memory by sharing memory space for different data types.
- They are commonly used in scenarios where only one member of a group needs to be active at a given time, such as in hardware programming or storing different data interpretations.

Both structures and unions provide a way to organize and manipulate complex data in C++ programs, allowing for better code organization, readability, and reusability.

Lecture 37: Files

Practical Question: Write a C++ program to read data from a text file and display it on the console.

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // Open the file
    std::ifstream file("data.txt");

    // Check if the file is successfully opened
    if (!file) {
        std::cerr << "Error opening the file." << std::endl;
        return 1;
    }

    std::string line;

    // Read and display each line from the file
    while (std::getline(file, line)) {
        std::cout << line << std::endl;
    }

    // Close the file
    file.close();

    return 0;
}
```

Practical Question: Write a C++ program to write data to a binary file.

```cpp
#include <iostream>
#include <fstream>

int main() {
    // Open the file in binary mode
    std::ofstream file("data.bin", std::ios::binary);

    // Check if the file is successfully opened
    if (!file) {
        std::cerr << "Error opening the file." << std::endl;
        return 1;
    }

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

    // Write the array to the file
    file.write(reinterpret_cast<char*>(numbers), sizeof(numbers));

    // Close the file
    file.close();

    return 0;
}
```

Theory Question: How are files handled in C++? Explain the different file modes used in file handling.

In C++, file handling is performed using the <fstream> library, which provides classes and functions for working with files. The primary classes used for file handling are std::ifstream for input (reading) from a file and std::ofstream for output (writing) to a file. These classes are derived from the base class std::fstream.

Different file modes are used to specify the purpose and behavior of file handling operations. The file modes are specified as flags passed to the constructor or through member functions.

The commonly used file modes in C++ are:

std::ios::in: Opens the file for input operations. Used with std::ifstream to read from a file.

std::ios::out: Opens the file for output operations. Used with std::ofstream to write to a file. If the file already exists, its previous contents are discarded.

std::ios::app: Appends the output to the end of the file instead of overwriting its contents. Used with std::ofstream.

std::ios::binary: Opens the file in binary mode, allowing input and output of non-text data. Used for binary file handling.

std::ios::ate: Sets the initial position at the end of the file. This is useful for both input and output operations.

std::ios::trunc: Truncates the file if it exists. Used with std::ofstream to discard the previous contents of the file.

These modes can be combined using the bitwise OR (|) operator to specify multiple behaviors.

File handling in C++ involves opening a file, performing read/write operations, and closing the file to release resources. The file stream objects (std::ifstream and std::ofstream) provide member functions like open(), close(), read(), write(), and getline() to perform file operations.

By utilizing file handling capabilities, C++ programs can read data from files, write data to files, and manipulate file content for various purposes such as data storage, data retrieval, and data processing.

Lecture 38: Classes & Objects

Practical Question: Write a C++ program to create a class representing a rectangle and calculate its area.

```cpp
#include <iostream>

class Rectangle {
private:
    double length;
    double width;

public:
    // Constructor
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    // Function to calculate the area of the rectangle
    double calculateArea() {
        return length * width;
    }
};
```

```cpp
int main() {
    // Create an instance of the Rectangle class
    Rectangle rectangle(5.0, 3.0);

    // Calculate the area of the rectangle
    double area = rectangle.calculateArea();

    // Display the area
    std::cout << "Area of the rectangle: " << area << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to create a class representing a bank account and perform deposit and withdrawal operations.

```cpp
#include <iostream>

class BankAccount {
private:
    std::string accountNumber;
    double balance;

public:
    // Constructor
    BankAccount(std::string accNum, double initialBalance) {
        accountNumber = accNum;
        balance = initialBalance;
    }

    // Function to deposit money
    void deposit(double amount) {
        balance += amount;
    }

    // Function to withdraw money
    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            std::cout << "Insufficient balance." << std::endl;
        }
    }
```

```
    // Function to get the account balance
    double getBalance() {
        return balance;
    }
};

int main() {
    // Create an instance of the BankAccount class
    BankAccount account("123456789", 1000.0);

    // Deposit money into the account
    account.deposit(500.0);

    // Withdraw money from the account
    account.withdraw(200.0);

    // Get the account balance
    double balance = account.getBalance();

    // Display the account balance
    std::cout << "Account Balance: " << balance << std::endl;

    return 0;
}
```

Theory Question: Explain the concepts of classes and objects in C++.

In C++, a class is a blueprint or a template for creating objects. It defines the properties (data members) and behaviors (member functions) that objects of that class can have. The class specifies the structure and behavior of objects but does not allocate memory for them.

An object, on the other hand, is an instance of a class. It represents a specific entity or element created based on the class definition. Each object created from a class has its own set of data members and can perform operations defined by the class's member functions.

Key concepts of classes and objects in C++:

Encapsulation: Classes encapsulate data and functions into a single unit. Data members are usually declared as private to provide data hiding and ensure data integrity. Member functions are used to manipulate and access the data, providing controlled access to the object's internal state.

Abstraction: Classes abstract the complexities of implementation and provide a simplified interface for interacting with objects. Users of a class only need to know the public interface (public member functions) and don't need to be concerned with the internal details of the class.

Inheritance: Classes can inherit properties and behaviors from other classes. Inheritance allows for code reuse and supports the creation of class hierarchies.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. Polymorphic behavior is achieved through virtual functions and function overriding.

Constructor: Constructors are special member functions used to initialize objects of a class. They are automatically called when an object is created and can take arguments to initialize the object's data members.

Destructor: Destructors are special member functions used to clean up resources and perform necessary cleanup operations when an object is destroyed or goes out of scope.

Classes and objects provide a structured and modular approach to programming, allowing for code organization, reusability, and the implementation of real-world concepts and entities in the form of software components.

Lecture 39: Class Methods

Practical Question: Write a C++ program to create a class representing a calculator and perform arithmetic operations using class methods.

```cpp
#include <iostream>

class Calculator {
public:
    // Static method to add two numbers
    static int add(int a, int b) {
        return a + b;
    }

    // Static method to subtract two numbers
    static int subtract(int a, int b) {
        return a - b;
    }

    // Static method to multiply two numbers
    static int multiply(int a, int b) {
        return a * b;
    }
```

```
    // Static method to divide two numbers
    static double divide(int a, int b) {
        if (b != 0) {
            return static_cast<double>(a) / b;
        } else {
            std::cerr << "Error: Division by zero." << std::endl;
            return 0;
        }
    }
};

int main() {
    int num1, num2;
    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;

    // Perform arithmetic operations using class methods
    int sum = Calculator::add(num1, num2);
    int difference = Calculator::subtract(num1, num2);
    int product = Calculator::multiply(num1, num2);
    double quotient = Calculator::divide(num1, num2);
```

```
    // Display the results
    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;
    std::cout << "Product: " << product << std::endl;
    std::cout << "Quotient: " << quotient << std::endl;

    return 0;
}
```

Practical Question: Write a C++ program to create a class representing a book and display its details using class methods.

```cpp
#include <iostream>
#include <string>

class Book {
private:
    std::string title;
    std::string author;
    int year;

public:
    // Constructor
    Book(std::string t, std::string a, int y) {
        title = t;
        author = a;
        year = y;
    }

    // Method to display book details
    void displayDetails() {
        std::cout << "Title: " << title << std::endl;
        std::cout << "Author: " << author << std::endl;
        std::cout << "Year: " << year << std::endl;
    }
};
```

```
int main() {
    // Create an instance of the Book class
    Book book("The Great Gatsby", "F. Scott Fitzgerald", 1925);

    // Display the book details
    book.displayDetails();

    return 0;
}
```

Theory Question: Discuss the use of class methods in C++.

Class methods, also known as member functions, are functions defined within a class that operate on objects of that class. They are used to perform specific tasks and operations related to the class and its objects. Class methods have access to the class's data members and can manipulate them as needed.

Here are some key points regarding the use of class methods in C++:

Encapsulation: Class methods encapsulate operations and behavior within the class. They allow for better organization and modularization of code by keeping related functions together.

Access to Data Members: Class methods have direct access to the class's data members, including private members. This allows them to read and modify the state of objects.

Object-Specific Operations: Class methods can perform operations that are specific to objects of the class. They can access the object's state and perform calculations, transformations, or other actions based on that state.

Code Reusability: Class methods can be reused across multiple objects of the same class. Once a method is defined, it can be invoked on different objects, providing consistent behavior.

Code Organization: Class methods provide a logical and structured way to organize code related to a class. By grouping related functionality within the class, the code becomes more readable, maintainable, and easier to navigate.

Object-Oriented Principles: Class methods play a central role in implementing object-oriented principles such as encapsulation, abstraction, and polymorphism. They enable the interaction and manipulation of objects, allowing for the creation of complex software systems.

Overall, class methods contribute to the object-oriented programming paradigm by encapsulating behavior within classes, promoting code reusability, and providing a clear and structured way to work with objects.

Lecture 40: Object-oriented Programming

Practical Question: Write a C++ program to create a class representing a car and simulate its movement using object-oriented programming concepts.

```cpp
#include <iostream>
#include <string>

class Car {
private:
    std::string brand;
    std::string model;
    int year;
    int speed;

public:
    // Constructor
    Car(std::string b, std::string m, int y) {
        brand = b;
        model = m;
        year = y;
        speed = 0;
    }

    // Method to accelerate the car
    void accelerate() {
        speed += 10;
    }
```

```
    // Method to brake the car
    void brake() {
        if (speed >= 10) {
            speed -= 10;
        } else {
            speed = 0;
        }
    }

    // Method to get the current speed of the car
    int getSpeed() {
        return speed;
    }

    // Method to display car details
    void displayDetails() {
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Model: " << model << std::endl;
        std::cout << "Year: " << year << std::endl;
        std::cout << "Speed: " << speed << " km/h" << std::endl;
    }
};
```

```
int main() {
    // Create an instance of the Car class
    Car myCar("Tesla", "Model 3", 2021);

    // Display initial car details
    std::cout << "Initial Details:" << std::endl;
    myCar.displayDetails();

    // Accelerate the car
    myCar.accelerate();
    myCar.accelerate();

    // Display updated car details
    std::cout << "After Acceleration:" << std::endl;
    myCar.displayDetails();

    // Brake the car
    myCar.brake();

    // Display final car details
    std::cout << "After Braking:" << std::endl;
    myCar.displayDetails();

    return 0;
}
```

Practical Question: Write a C++ program to create a class representing a bank and perform account-related operations using object-oriented programming.

```cpp
#include <iostream>
#include <string>

class BankAccount {
private:
    std::string accountNumber;
    std::string accountHolder;
    double balance;

public:
    // Constructor
    BankAccount(std::string number, std::string holder, double initialBalance)
        accountNumber = number;
        accountHolder = holder;
        balance = initialBalance;
    }

    // Method to deposit money into the account
    void deposit(double amount) {
        balance += amount;
        std::cout << "Deposit of $" << amount << " successful." << std::endl;
    }
```

```
// Method to deposit money into the account
void deposit(double amount) {
    balance += amount;
    std::cout << "Deposit of $" << amount << " successful." << std::endl;
}

// Method to withdraw money from the account
void withdraw(double amount) {
    if (amount <= balance) {
        balance -= amount;
        std::cout << "Withdrawal of $" << amount << " successful." << std::en
    } else {
        std::cout << "Insufficient balance." << std::endl;
    }
}

// Method to display account details
void displayDetails() {
    std::cout << "Account Number: " << accountNumber << std::endl;
    std::cout << "Account Holder: " << accountHolder << std::endl;
    std::cout << "Balance: $" << balance << std::endl;
}
```

```
main() {
// Create an instance of the BankAccount class
BankAccount myAccount("123456789", "John Doe", 1000);

// Display initial account details
std::cout << "Initial Details:" << std::endl;
myAccount.displayDetails();

// Deposit money into the account
myAccount.deposit(500);

// Display updated account details
std::cout << "After Deposit:" << std::endl;
myAccount.displayDetails();

// Withdraw money from the account
myAccount.withdraw(200);

// Display final account details
std::cout << "After Withdrawal:" << std::endl;
myAccount.displayDetails();

return 0;
```

Theory Question: Explain the concept of object-oriented programming (OOP) and its advantages over procedural programming.

Object-oriented programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. It focuses on modeling real-world entities as objects that have data (attributes) and behavior (methods). OOP promotes code reusability, modularity, and allows for the implementation of various object-oriented principles, such as encapsulation, inheritance, and polymorphism.

Advantages of object-oriented programming over procedural programming include:

Modularity and Reusability: OOP allows code to be organized into self-contained objects, promoting modularity. Objects can be reused in different parts of a program or in different programs, enhancing code reusability.

Encapsulation: OOP enables encapsulation, which means bundling data and methods together within an object. Encapsulation provides data security by hiding implementation details and exposing only necessary interfaces, improving code maintainability and reducing dependencies.

Inheritance: Inheritance allows the creation of new classes (derived classes) based on existing classes (base classes). Derived classes inherit the properties and behaviors of their base classes, allowing code reuse and creating hierarchical relationships between classes.

Polymorphism: Polymorphism enables objects of different classes to be treated as objects of a common base class. It allows for flexibility and extensibility, as different objects can respond to the same method calls in different ways, based on their specific implementations.

Code Readability and Maintainability: OOP emphasizes clear and organized code structure, making programs more readable and understandable. OOP's modular nature simplifies maintenance tasks by isolating changes to specific objects or classes, without impacting the entire program.

Scalability and Flexibility: OOP provides a scalable approach to software development. New features can be added by creating new classes or extending existing ones, without modifying the entire codebase. This flexibility makes it easier to adapt and evolve software over time.

Overall, OOP facilitates the development of robust, scalable, and maintainable software systems by promoting code reuse, encapsulation, and modularity. It aligns well with real-world modeling, enabling developers to represent complex systems using intuitive object-oriented concepts.