ISDM **(INDEPENDENT SKILL DEVELOPMENT MISSION)**

REVIEW OF C++ BASICS: DATA TYPES, OPERATORS, LOOPS, FUNCTIONS

CHAPTER 1: UNDERSTANDING DATA TYPES IN C++

Introduction to Data Types

C++ is a strongly typed language, meaning every variable must be declared with a specific data type. This ensures efficient memory allocation and prevents unintended operations. Data types define the kind of data that a variable can store and manipulate. The three major categories of data types in C++ are:

- **Primitive Data Types:** Includes int, float, double, char, and bool.
- **Derived Data Types:** Arrays, pointers, references.
- **User-Defined Data Types:** Structures, classes, unions.

Understanding these data types is crucial for writing optimized and error-free C++ programs.

Primitive Data Types

1. **Integer (int)** – Stores whole numbers.
2. int age = 25;

3. **Floating-point (float, double)** – Stores decimal numbers.
4. float price = 10.99;
5. **Character (char)** – Stores a single character.
6. char grade = 'A';
7. **Boolean (bool)** – Stores true or false.
8. bool isRaining = false;

Exercise

1. Declare variables of different data types and initialize them.
2. Write a C++ program that takes user input for different data types and displays the values.

Case Study: Data Types in a Banking System

A banking system requires different data types to store information like:

- **Account Number:** int
- **Account Balance:** double
- **Customer Name:** string
- **Account Status:** bool Using the right data types ensures smooth banking transactions and data accuracy.

CHAPTER 2: OPERATORS IN C++

Introduction to Operators

Operators in C++ allow manipulation of variables and values. They include:

- Arithmetic Operators (+, -, *, /, %)
- Relational Operators (==, !=, >, <, >=, <=)
- Logical Operators (&&, ||, !)
- Bitwise Operators (&, |, ^, <<, >>)

Types of Operators with Examples

1. **Arithmetic Operators:**
2. int a = 10, b = 3;
3. int sum = a + b; // 13
4. int remainder = a % b; // 1
5. **Relational Operators:**
6. if (a > b) {
7.   cout << "a is greater than b";
8. }
9. **Logical Operators:**
10.   bool isSunny = true;
11. bool isWarm = false;
12.   if (isSunny && isWarm) {
13.   cout << "Great weather!";
14.   }

Exercise

1. Implement a program that takes two numbers and performs arithmetic operations.
2. Create a program to check if a number is positive, negative, or zero using relational and logical operators.

Case Study: Operators in Payroll Calculation

A company calculates salaries using arithmetic operators. Relational and logical operators determine whether an employee is eligible for overtime pay.

CHAPTER 3: LOOPS IN C++

Introduction to Loops

Loops help execute a block of code multiple times, reducing redundancy. The three types of loops in C++ are:

- **For Loop** – Used when the number of iterations is known.
- **While Loop** – Used when the number of iterations is unknown.
- **Do-While Loop** – Ensures at least one execution before checking the condition.

Examples of Loops

1. **For Loop:**
2. for (int i = 1; i <= 5; i++) {
3.   cout << i << " ";
4. }

5. **While Loop:**
6. int i = 1;
7. while (i <= 5) {
8.   cout << i << " ";
9.   i++;
10.   }
11. **Do-While Loop:**
12.   int i = 1;
13. do {
14.   cout << i << " ";
15.   i++;
16.   } while (i <= 5);

Exercise

1. Write a program that prints numbers from 1 to 10 using all three loops.
2. Create a program that calculates the sum of the first N natural numbers using a while loop.

Case Study: Loops in Inventory Management

A warehouse uses loops to update stock levels. A while loop checks inventory levels, and a for loop updates prices of all products in bulk.

CHAPTER 4: FUNCTIONS IN C++

Introduction to Functions

Functions allow modular programming and code reusability. A function is a block of code that performs a specific task. The four types of functions in C++ are:

1. Built-in functions
2. User-defined functions
3. Recursive functions
4. Inline functions

Types of Functions

1. **User-Defined Function:**
2. void greet() {
3.   cout << "Hello, welcome to C++!";
4. }
5. **Function with Parameters:**
6. int add(int a, int b) {
7.   return a + b;
8. }
9. **Recursive Function:**
10.   int factorial(int n) {
11.   if (n == 0) return 1;
12.   return n * factorial(n - 1);
13. }

Exercise

1. Implement a function that calculates the area of a circle given its radius.
2. Write a program that uses recursion to calculate the factorial of a number.

Case Study: Functions in Online Shopping Platforms

E-commerce platforms use functions for checkout, payment processing, and order tracking. These functions enhance maintainability and scalability.

OBJECT-ORIENTED PROGRAMMING (OOP) PRINCIPLES IN C++

CHAPTER 1: INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

Understanding OOP

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. It provides a structured way of organizing and managing code by encapsulating data and behavior within objects. Unlike procedural programming, which focuses on functions and procedures, OOP emphasizes real-world modeling using objects and classes.

OOP is widely used in software development due to its modularity, reusability, and scalability. C++ is one of the most popular OOP languages, enabling developers to create complex, efficient, and maintainable software applications.

Key Features of OOP

1. **Encapsulation:** Bundling data and methods into a single unit.
2. **Abstraction:** Hiding complex implementation details and exposing only the necessary functionalities.
3. **Inheritance:** Reusing existing class properties in new classes to reduce redundancy.
4. **Polymorphism:** Allowing objects to be treated as instances of their parent class, enabling flexibility in method usage.

Example: Basic OOP Structure in C++

```
#include <iostream>
```

```
using namespace std;


class Car {
public:
   string brand;
   int speed;


   void displayInfo() {
      cout << "Car Brand: " << brand << ", Speed: " << speed << " km/h"
<< endl;
   }
};


int main() {
   Car car1;
   car1.brand = "Toyota";
   car1.speed = 180;
   car1.displayInfo();


   return 0;
}
```

Exercise

1. Define a class Person with attributes name and age, and a method to display the details.
2. Create multiple objects from the class and call the method.

Case Study: OOP in Library Management System

A library management system uses OOP to create classes like Book, Member, and Librarian. Each class has attributes and methods to manage book inventory, issue books, and track members.

CHAPTER 2: ENCAPSULATION IN C++

What is Encapsulation?

Encapsulation is the practice of **restricting direct access** to an object's data and only allowing manipulation through well-defined methods. It ensures **data security** and **prevents unintended modifications**.

How to Implement Encapsulation in C++

- Use **private** and **protected** access specifiers to restrict direct data access.
- Provide **public methods (getters and setters)** to access and modify data safely.

Example: Encapsulation in C++

```
#include <iostream>

using namespace std;
```

```
class BankAccount {
private:
   double balance;

public:
   void setBalance(double amount) {
      if (amount >= 0)
         balance = amount;
      else
         cout << "Invalid amount!" << endl;
   }

   double getBalance() {
      return balance;
   }
};

int main() {
   BankAccount account;
   account.setBalance(1000);
   cout << "Account Balance: $" << account.getBalance() << endl;
```

```
    return 0;
}
```

Exercise

1. Implement a class Student with private attributes name and marks. Use setter and getter methods to assign and retrieve values.
2. Try accessing private attributes directly and observe the error.

Case Study: Encapsulation in Online Banking Systems

Online banking applications use encapsulation to **protect sensitive customer data** such as account balance and transaction history, ensuring only authorized access.

CHAPTER 3: INHERITANCE IN C++

What is Inheritance?

Inheritance allows a **child class (derived class)** to acquire properties and behaviors of a **parent class (base class)**. It promotes **code reusability** and maintains a hierarchical class structure.

Types of Inheritance

1. **Single Inheritance:** One class derives from another.
2. **Multiple Inheritance:** A class inherits from more than one base class.
3. **Multilevel Inheritance:** A class is derived from another derived class.

4. **Hierarchical Inheritance:** Multiple classes inherit from a single base class.
5. **Hybrid Inheritance:** Combination of two or more types of inheritance.

Example: Single Inheritance in C++

```
#include <iostream>

using namespace std;


class Animal {

public:

   void eat() {

      cout << "This animal eats food." << endl;

   }

};


class Dog : public Animal {

public:

   void bark() {

      cout << "The dog barks." << endl;

   }

};
```

```
int main() {
  Dog myDog;
  myDog.eat();
  myDog.bark();


  return 0;
}
```

Exercise

1. Create a class Vehicle with an attribute maxSpeed and method showSpeed(). Derive a class Car that inherits from Vehicle and adds a new method showBrand().
2. Implement **multilevel inheritance** using Animal -> Mammal -> Human.

Case Study: Inheritance in Ride-Sharing Apps

Ride-sharing apps like Uber use inheritance to define a **base class (Vehicle)** and derived classes like Car, Bike, and Truck. This enables code reuse and maintains a structured hierarchy.

Chapter 4: Polymorphism in C++

What is Polymorphism?

Polymorphism allows **one interface, multiple implementations**. It enables different classes to **override** the same method with distinct behavior.

Types of Polymorphism

1. **Compile-Time (Method Overloading):** Multiple functions with the same name but different parameters.
2. **Run-Time (Method Overriding):** A derived class redefines a function from the base class.

Example: Method Overloading in C++

```
#include <iostream>

using namespace std;


class MathOperations {

public:

   int add(int a, int b) {

      return a + b;

   }


   double add(double a, double b) {

      return a + b;

   }

};


int main() {

   MathOperations obj;
```

```
    cout << obj.add(10, 20) << endl;

    cout << obj.add(5.5, 2.3) << endl;


    return 0;

}
```

Example: Method Overriding in C++

```
#include <iostream>

using namespace std;


class Animal {

public:

    virtual void makeSound() {

        cout << "Animal makes a sound." << endl;

    }

};


class Dog : public Animal {

public:

    void makeSound() override {

        cout << "Dog barks." << endl;

    }
};


int main() {
   Animal* myPet = new Dog();
   myPet->makeSound();


   delete myPet;
   return 0;
}
```

Exercise

1. Implement method overloading in a class Calculator for performing addition of integers and floating-point numbers.
2. Override a method displayInfo() in a class Person, which is further inherited by a class Student.

Case Study: Polymorphism in Gaming Engines

Game development engines like **Unity and Unreal Engine** use polymorphism to define **common base classes for game objects**, allowing customization of character movements and behaviors.

CONCLUSION

Object-Oriented Programming (OOP) in C++ provides a **modular, reusable, and scalable** approach to software development. Mastering **Encapsulation, Inheritance, and Polymorphism** helps in building efficient real-world applications like **banking systems, ride-sharing apps, gaming engines, and e-commerce platforms**.

By practicing **exercises and real-world case studies**, you will gain hands-on experience in **writing optimized C++ programs using OOP principles**.

ADVANCED FUNCTIONS: FUNCTION OVERLOADING, DEFAULT ARGUMENTS

FUNCTION OVERLOADING

Function overloading is a feature in object-oriented programming languages like C++, Java, and Python that allows a programmer to define multiple functions with the same name but with different signatures. This concept allows the same function to be used for different purposes, improving code readability and reusability. Overloading in functions typically depends on the number of parameters or the types of parameters used. Function overloading is not about changing the function’s name, but rather altering its behavior based on its input arguments.

In many programming languages, the function name remains the same, but the number or type of parameters defines the specific version of the function that is called. This eliminates the need to come up with entirely different names for similar functions, making the code cleaner and more intuitive. For example, a function for calculating the area of different geometric shapes could be overloaded to handle different shapes like circles, squares, and rectangles, each requiring different input parameters.

Example: Function Overloading in C++

```
#include <iostream>

using namespace std;


// Overloaded functions to calculate the area of different shapes
```

```
double area(double radius) {

   return 3.14 * radius * radius;  // Area of a circle

}


int area(int side) {

   return side * side;  // Area of a square

}


int area(int length, int width) {

   return length * width;  // Area of a rectangle

}


int main() {

   double circleArea = area(5.0);  // Circle area

   int squareArea = area(4);       // Square area

   int rectangleArea = area(4, 6); // Rectangle area


   cout << "Circle Area: " << circleArea << endl;

   cout << "Square Area: " << squareArea << endl;

   cout << "Rectangle Area: " << rectangleArea << endl;

   return 0;
}
```

In this example, the area() function is overloaded three times, each with different parameters: one for a circle (using a double), one for a square (using an int), and one for a rectangle (using two int parameters). The correct version of the function is called based on the arguments passed.

CASE STUDY: FUNCTION OVERLOADING IN REAL-WORLD APPLICATIONS

In real-world applications, function overloading is commonly used in mathematical software, graphical programs, and even game development. For instance, a graphics rendering engine could overload functions to handle different shapes. One function could accept parameters for drawing a circle, while another could accept parameters for drawing a rectangle or a polygon, depending on the complexity of the object being rendered.

This improves the modularity of the system and makes it easier to maintain, as the same function name is reused across different contexts, ensuring a simpler interface for the developer.

Default Arguments

Default arguments are another powerful feature in many programming languages that allow functions to be called with fewer arguments than they are defined to accept. Default arguments are provided in the function definition, enabling the function to be invoked with fewer arguments. If a function is called without a specific argument, the default value is used instead. This feature simplifies function calls and makes code more flexible.

In languages like C++ and Python, the default value for a function parameter can be specified in the function definition. This eliminates the need for overloading functions for every possible combination of parameters. Default arguments allow developers to create more flexible and concise functions without requiring them to define multiple versions of the same function.

Example: Default Arguments in Python

```
def greet(name, message="Hello"):

    print(f"{message}, {name}!")



# Function calls with and without the second argument

greet("Alice")

greet("Bob", "Good Morning")
```

In this Python example, the function greet() is designed to print a greeting message. The message parameter has a default value of "Hello". When the function is called with only one argument (as in greet("Alice")), it defaults to using "Hello". However, if the second argument is provided (as in greet("Bob", "Good Morning")), it uses the provided message.

CASE STUDY: DEFAULT ARGUMENTS IN WEB DEVELOPMENT

Consider a web application where a user is sending emails. The function that sends an email could be designed to accept multiple parameters, such as the recipient's email address, subject, message, and priority. If some of these parameters are optional, default arguments can be set. For instance, if the priority is not specified, the system could automatically default it to "normal". This reduces the complexity for users of the function and ensures that fewer parameters need to be specified in common scenarios, improving the efficiency of the code and the user experience.

CONCLUSION

Both function overloading and default arguments play an essential role in making code more efficient, flexible, and easier to maintain. Function overloading helps by allowing multiple functions with the same name to handle different types or numbers of parameters, while default arguments provide flexibility by enabling functions to be called with fewer parameters. By using these techniques, programmers can create cleaner, more understandable, and more reusable code that can handle a wide range of use cases with minimal effort. These concepts are widely used in modern programming languages and are critical for building scalable and maintainable software systems.

Exercises

1. **Function Overloading:**
    - Create a program that calculates the perimeter of different shapes (circle, square, and rectangle). Implement function overloading to handle these different shapes with varying numbers of parameters.
2. **Default Arguments:**
    - Write a function that calculates the total price of an item after applying a discount. The discount should be optional and default to 10% if not provided. Write a

program that calls this function with and without the discount parameter.

NAMESPACES & PREPROCESSORS IN C++

NAMESPACES IN C++

Namespaces in C++ are a way of organizing code into logical groups to avoid name conflicts. In larger projects, it is common to have multiple functions, classes, and variables with the same names, especially when integrating third-party libraries. To resolve such conflicts, C++ introduces the concept of namespaces. A namespace allows developers to define a scope to group related classes, functions, and variables under a name, preventing name clashes across different parts of the program.

Namespaces help avoid conflicts when the same name is used for different identifiers in different libraries or in different parts of a program. They also enhance the modularity and clarity of the code, making it easier for developers to understand and maintain. The std namespace is one of the most commonly used namespaces in C++ as it includes the Standard Library components such as input/output operations, containers, and algorithms.

Example: Basic Namespace Usage

```
#include <iostream>

using namespace std;


namespace MathOperations {

  int add(int a, int b) {

    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }
}

int main() {
  int sum = MathOperations::add(5, 3);
  int difference = MathOperations::subtract(5, 3);

  cout << "Sum: " << sum << endl;
  cout << "Difference: " << difference << endl;
  return 0;
}
```

In this example, a namespace MathOperations is created to group the add and subtract functions. In the main function, these functions are accessed using the scope resolution operator :: followed by the namespace name.

CASE STUDY: USE OF NAMESPACES IN LARGE PROJECTS

In large software systems, namespaces become indispensable. Consider a large-scale software application where multiple teams work on different modules such as user authentication, database access, and data processing. Each module might have classes and functions with similar names. Without namespaces, there could be a significant risk of name conflicts. By encapsulating each module into its own namespace, you ensure that all the classes, variables, and functions remain logically separated and can be used independently. This improves code readability and maintainability, as it is clear which module each part of the code belongs to.

PREPROCESSORS IN C++

The C++ Preprocessor is a powerful tool that allows you to manipulate code before it is compiled. Preprocessors are directives that provide instructions to the compiler to preprocess the source code. These preprocessing operations occur before the actual compilation process starts. Preprocessor directives in C++ begin with a # symbol and include operations like including files, defining constants, and conditional compilation.

The most commonly used preprocessor directives are #include, #define, #ifdef, #ifndef, and #endif. These directives help in including header files, defining macros, and controlling the inclusion of certain parts of code based on conditional checks. The preprocessor is a powerful feature because it allows you to modify and control the code before it is compiled, enabling more efficient and flexible programming.

Example: Preprocessor Directives in C++

```
#include <iostream>

#define PI 3.14
```

```
int main() {
   float radius = 5.0;
   float area = PI * radius * radius; // PI is replaced by 3.14 by the
preprocessor

   std::cout << "Area of the circle: " << area << std::endl;
   return 0;
}
```

In this example, the #define directive is used to define a constant PI with the value 3.14. The preprocessor replaces all instances of PI in the code with the defined value before compiling the program.

CONDITIONAL COMPILATION WITH PREPROCESSORS

Preprocessors are also used for conditional compilation, which allows certain sections of code to be included or excluded based on specific conditions. This is especially useful when writing cross-platform code or debugging code that only needs to run in certain environments.

```
#include <iostream>
#define DEBUG

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

```
    #ifdef DEBUG

    std::cout << "Debugging: x = " << x << ", y = " << y << std::endl;

    #endif


    return 0;

}
```

In this example, the #ifdef directive checks if the DEBUG macro is defined. If it is, the debugging code will be included in the compilation; otherwise, it will be ignored. This allows developers to include or exclude specific sections of code without modifying the entire codebase, which is useful during development and testing.

Case Study: Preprocessors in Cross-Platform Development

In cross-platform software development, preprocessors are crucial for writing platform-specific code. For example, when developing a program that runs both on Windows and Linux, you might need to include different headers or use different APIs based on the operating system. By using preprocessor directives like #ifdef WIN32 and #ifdef LINUX, you can include operating system-specific code only when necessary, ensuring that the program compiles and runs correctly on different platforms.

Conclusion

Both namespaces and preprocessors are fundamental concepts in C++ that contribute significantly to the organization and flexibility of code. Namespaces allow developers to organize their code logically, preventing naming conflicts in large projects. They improve modularity and ensure that components are easy to maintain and understand. Preprocessors, on the other hand, provide a way to manipulate code before it is compiled, allowing for efficient debugging, conditional compilation, and code optimization. These features are essential tools for C++ developers, enabling them to write flexible, efficient, and maintainable code.

Exercises

1. **Namespaces Exercise:**
   - Create a program that includes two different namespaces: one for basic arithmetic operations (addition, subtraction) and another for advanced operations (square root, power). Implement functions for both sets of operations and call them in the main function by using the namespace scope resolution operator.
2. **Preprocessor Exercise:**
   - Write a program that defines a constant for the maximum number of users (MAX_USERS). Use conditional compilation to include different code blocks depending on whether MAX_USERS is above or below a certain value (e.g., 100). Display different messages depending on the value of MAX_USERS.

Input & Output Operations (File Handling Basics)

Input and Output Operations in C++

In C++, input and output operations are fundamental for interacting with the user and other external data sources, such as files and databases. These operations allow programs to receive data from the user (input) and send information to the user or external systems (output). The standard input and output streams in C++ are handled by the cin and cout objects, which are part of the standard C++ library (iostream header). The cin object is used for taking input, while cout is used for displaying output to the user.

The standard input/output operations in C++ are synchronous, meaning that the program execution pauses until the user provides input or until the output is displayed. This makes it easy to interact with users in a sequential flow, allowing them to enter data that is processed in real-time.

In addition to basic operations like reading from the keyboard and writing to the console, C++ also provides more advanced features such as file handling. File handling allows programs to read from and write to files, making it possible to store data persistently and retrieve it later. File handling is essential for applications like text editors, loggers, and databases.

Example: Basic Input/Output Operations

```
#include <iostream>

using namespace std;
```

```
int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;  // Taking input from the user
   cout << "Your age is: " << age << endl;  // Displaying output
   return 0;
}
```

In this example, the program prompts the user to enter their age, takes the input using cin, and then displays the age using cout.

File Handling Basics

File handling in C++ allows programs to interact with files stored on the disk. Files can be used to store data between program executions, making them useful for long-term data storage. The file handling capabilities in C++ are provided by the fstream library, which includes the classes ifstream (input file stream), ofstream (output file stream), and fstream (for both input and output).

When working with files, there are two primary types of operations: reading from a file and writing to a file. Reading from a file allows the program to access data that has been previously stored, while writing to a file allows the program to save data for later use. File handling operations can be performed in different modes, such as read, write, append, or binary mode.

Before performing any file operation, the file must be opened using the open() function, and once operations are completed, the file should be closed using the close() function to release system resources.

Example: Writing to a File

```
#include <iostream>

#include <fstream>

using namespace std;


int main() {

   ofstream outFile("example.txt");  // Open the file for writing

   if (outFile.is_open()) {

      outFile << "Hello, World!" << endl;  // Writing to the file

      outFile << "This is a C++ file handling example." << endl;

      outFile.close();  // Close the file

      cout << "Data written to file successfully." << endl;

   } else {

      cout << "Failed to open file." << endl;

   }

   return 0;

}
```

In this example, an ofstream object is used to open the file "example.txt" for writing. The program writes two lines of text to the file and then closes the file. If the file cannot be opened, an error message is displayed.

Example: Reading from a File

```
#include <iostream>

#include <fstream>

using namespace std;


int main() {

  ifstream inFile("example.txt");  // Open the file for reading

  string line;

  if (inFile.is_open()) {

    while (getline(inFile, line)) {  // Read the file line by line

      cout << line << endl;  // Display the content of the file

    }

    inFile.close();  // Close the file

  } else {

    cout << "Failed to open file." << endl;

  }

  return 0;

}
```

In this example, an ifstream object is used to open the file "example.txt" for reading. The getline() function reads each line from the file until the end of the file is reached, and the content is displayed on the console.

CASE STUDY: FILE HANDLING IN DATA LOGGING

Consider a scenario where a program needs to log sensor data from a real-time monitoring system. The data generated by the sensor (e.g., temperature readings) is written to a file periodically. The program also allows for the retrieval of previous logs for analysis. File handling becomes crucial here, as it enables the program to persist data over time, making it possible to retrieve historical sensor readings when needed.

For this system, the program can write data to a file in append mode, ensuring that new data is added to the end of the file without overwriting the previous entries. Additionally, the program can read the log file to display the historical data whenever required. File handling ensures that the program can manage large amounts of data efficiently and maintain records across multiple sessions.

Example: Data Logging Application

```
#include <iostream>

#include <fstream>

#include <ctime>

using namespace std;


int main() {

   ofstream logFile("sensor_data.txt", ios::app);  // Open the file in
append mode
```

```
  if (logFile.is_open()) {
    time_t currentTime = time(0);  // Get the current time
    char* timeStr = ctime(&currentTime);  // Convert time to string


    float temperature = 22.5;  // Example sensor data


    // Write data to the log file
    logFile << "Time: " << timeStr << " Temperature: " <<
temperature << "°C" << endl;
    logFile.close();  // Close the file
    cout << "Sensor data logged successfully." << endl;
  } else {
    cout << "Failed to open log file." << endl;
  }
  return 0;
}
```

In this example, sensor data is logged with a timestamp. The ios::app flag ensures that new data is appended to the file without overwriting previous entries. This allows for continuous data logging without losing past data.

CONCLUSION

File handling in C++ is a crucial aspect of programming, enabling developers to create applications that persist data across multiple runs and interact with external data sources. By understanding the basics of file operations and mastering input/output operations, programmers can enhance the functionality of their applications, making them more dynamic and flexible. C++ provides powerful tools like fstream for handling files, allowing programs to read from and write to files seamlessly. By incorporating file handling, C++ programs can store data for later use, ensuring that important information is retained and accessible when needed.

Exercises

1. **File Handling Exercise:**
   - Write a program that takes user input and saves it to a file. The program should ask the user for their name, age, and favorite color and then save this information to a file named user_info.txt.
2. **Log File Exercise:**
   - Create a program that simulates the logging of system events. The program should log messages to a file with timestamps. After logging several messages, read the file and display all the logged events on the screen.

ASSIGNMENT SOLUTION: IMPLEMENT A C++ PROGRAM THAT DEMONSTRATES FUNCTION OVERLOADING

In this assignment, we will demonstrate function overloading in C++ by creating a program that uses the same function name to perform different operations based on the number and type of arguments passed to the function.

Step-by-Step Guide to Solution

STEP 1: UNDERSTAND FUNCTION OVERLOADING

Function overloading in C++ allows you to define multiple functions with the same name, but with different parameter lists. The compiler determines which function to call based on the number and types of arguments provided when the function is called.

For example, you can have a function named add() that works differently depending on whether you are adding two integers or two floating-point numbers.

STEP 2: SET UP YOUR DEVELOPMENT ENVIRONMENT

1. Open your IDE (e.g., Code::Blocks, Visual Studio, or any text editor with a C++ compiler).
2. Create a new C++ file and name it function_overloading.cpp.

STEP 3: IMPLEMENT THE PROGRAM

We will write a simple program that demonstrates function overloading by creating multiple add() functions that work for different data types.

```
#include <iostream>

using namespace std;


// Function to add two integers

int add(int a, int b) {

   return a + b;

}


// Function to add two floating-point numbers

float add(float a, float b) {

   return a + b;

}


// Function to add three integers

int add(int a, int b, int c) {

   return a + b + c;

}


int main() {
   int x = 5, y = 10, z = 15;

   float p = 5.5, q = 3.5;


   // Calling the add function for two integers

   cout << "Sum of two integers: " << add(x, y) << endl;


   // Calling the add function for two floating-point numbers

   cout << "Sum of two floating-point numbers: " << add(p, q) <<
endl;


   // Calling the add function for three integers

   cout << "Sum of three integers: " << add(x, y, z) << endl;


   return 0;

}
```

STEP 4: EXPLANATION OF THE CODE

1. **Function Overloading:**
    - We define three versions of the add() function:
        - The first function takes two integers as arguments and returns their sum.

- The second function takes two floating-point numbers (float) as arguments and returns their sum.
- The third function takes three integers as arguments and returns their sum.

2. **main() Function:**
   - In the main() function, we create three integer variables (x, y, z) and two floating-point variables (p, q).
   - We call the add() function three times, each with different arguments (two integers, two floating-point numbers, and three integers).
   - The appropriate version of the add() function is called based on the type and number of arguments passed.

STEP 5: COMPILE AND RUN THE PROGRAM

1. **Compile the Program:**
   - After writing the code, save the file and compile it using your C++ compiler.
   - In a terminal, you can use the following command to compile the program if you're using g++:
   - g++ function_overloading.cpp -o function_overloading
2. **Run the Program:**
   - After compiling, you can run the program using the following command:
   - ./function_overloading
3. **Expected Output:**

4. Sum of two integers: 15
5. Sum of two floating-point numbers: 9
6. Sum of three integers: 30

STEP 6: TEST WITH DIFFERENT VALUES

Try modifying the values of x, y, z, p, and q to test different scenarios. Ensure that the correct overloaded function is called based on the arguments.

Concepts Covered in This Assignment:

1. **Function Overloading:**
   - Defining multiple functions with the same name but different parameter types or numbers of parameters.
2. **Compiler Decision:**
   - The compiler selects the correct function based on the number and type of arguments passed.
3. **Code Modularity:**
   - Function overloading helps keep code concise and readable, allowing similar operations to be performed with different data types or numbers of arguments.

CONCLUSION

In this assignment, we have implemented a C++ program that demonstrates function overloading. By creating multiple add() functions with different parameters, we have shown how function overloading works in C++. This technique helps in writing cleaner, more maintainable code and avoids the need to create separate function names for similar operations.

Assignment Solution: Write a Program to Read and Write Data from a File

In this assignment, we will write a C++ program that demonstrates reading from and writing to a file. We will use file handling functions to store user input in a file and then retrieve and display that data from the file.

Step-by-Step Guide to Solution

Step 1: Understand the File Handling Basics

In C++, file handling is done using the fstream library, which provides:

- ofstream (output file stream) for writing data to a file.
- ifstream (input file stream) for reading data from a file.
- fstream for both reading and writing.

When working with files, it's essential to open the file first using the open() function and close the file using the close() function once the file operations are done.

Step 2: Set Up Your Development Environment

1. Open your IDE (e.g., Code::Blocks, Visual Studio, or any text editor with a C++ compiler).
2. Create a new C++ file and name it file_read_write.cpp.

Step 3: Implement the Program

We will implement the program in two parts:

1. **Writing to a file**: First, we'll allow the user to input some data and store that in a file.
2. **Reading from a file**: Then, we'll read the data back from the file and display it on the screen.

```
#include <iostream>

#include <fstream>

#include <string>

using namespace std;


int main() {

  string name, city;

  int age;


  // Part 1: Writing data to a file

  ofstream outFile("user_data.txt");  // Open the file for writing

  if (outFile.is_open()) {

    cout << "Enter your name: ";

    getline(cin, name);  // Get the user's name

    cout << "Enter your age: ";
```

```
        cin >> age;  // Get the user's age
        cin.ignore();  // To ignore the newline character left in the buffer
        cout << "Enter your city: ";
        getline(cin, city);  // Get the user's city

        // Write the data to the file
        outFile << "Name: " << name << endl;
        outFile << "Age: " << age << endl;
        outFile << "City: " << city << endl;

        outFile.close();  // Close the file after writing
        cout << "Data has been written to the file successfully." << endl;
    } else {
        cout << "Failed to open file for writing." << endl;
        return 1;
    }

    // Part 2: Reading data from the file
    ifstream inFile("user_data.txt");  // Open the file for reading
    if (inFile.is_open()) {
        cout << "\nData read from file:\n";
        string line;

        // Read and display the content of the file

        while (getline(inFile, line)) {

          cout << line << endl;

        }


        inFile.close();  // Close the file after reading

    } else {

        cout << "Failed to open file for reading." << endl;

        return 1;

    }


    return 0;

  }
```

Step 4: Explanation of the Code

1. **Input and Output to File**:
   - The program begins by including necessary libraries: iostream for input/output operations, fstream for file handling, and string for string manipulation.

- We define three variables: name, age, and city to store the user's input.

2. **Writing to File**:
   - We open a file named user_data.txt using an ofstream object. If the file is successfully opened, we prompt the user for their name, age, and city. We use getline() to read the entire name and city (which may contain spaces) and cin to read the age.
   - After collecting the input, we write the data to the file using the << operator.
   - Finally, we close the file using outFile.close() to save the changes and release resources.

3. **Reading from File**:
   - We open the same file for reading using an ifstream object. If the file is successfully opened, we read its content line by line using getline() and display it on the console.
   - After reading the file, we close it using inFile.close() to release the resources.

Step 5: Compile and Run the Program

1. **Compile the Program**:
   - After writing the code, save the file and compile it using your C++ compiler.
   - In a terminal, you can use the following command to compile the program if you're using g++:

  - g++ file_read_write.cpp -o file_read_write

2. **Run the Program**:
   - After compiling, you can run the program using the following command:
   - ./file_read_write
3. **Expected Output**:
4. Enter your name: John Doe
5. Enter your age: 25
6. Enter your city: New York
7. Data has been written to the file successfully.
8.
9. Data read from file:
10. Name: John Doe
11. Age: 25
12. City: New York

STEP 6: TEST WITH DIFFERENT DATA

Try running the program multiple times with different inputs to verify that the data is correctly written to the file and read back.

1. Change the input values for name, age, and city and ensure they are written to the file correctly.

2. Verify that the program can read and display the data from the file.

Concepts Covered in This Assignment:

1. **File Handling in C++**:
   - Learn how to use ofstream for writing data to a file and ifstream for reading data from a file.
2. **Input/Output Stream Operations**:
   - Use standard I/O streams like cin, cout, and file streams (ifstream, ofstream) to perform file operations.
3. **File Opening and Closing**:
   - Learn how to open and close files to ensure proper resource management.

CONCLUSION

In this assignment, we have written a C++ program that demonstrates how to read and write data to a file. By using the fstream library, we were able to open a file, write user input to it, and then read the data back from the file. This functionality is essential for many real-world applications where data needs to be saved between program executions, such as in text editors, loggers, or data storage systems.

Understanding file handling allows you to create programs that can persistently store and retrieve information, improving the interactivity and functionality of your software.

Exercises

1. **Modify the Program to Append Data**:
   - Modify the program to append data to the file instead of overwriting it each time the program runs.
2. **Error Handling Exercise**:
   - Add error handling to check if the file already exists before attempting to write to it. If the file exists, display an appropriate message.
3. **Multiple Records Handling**:
   - Modify the program to allow the user to input multiple records (e.g., name, age, and city) and store them in the file. Then, read and display all records from the file.