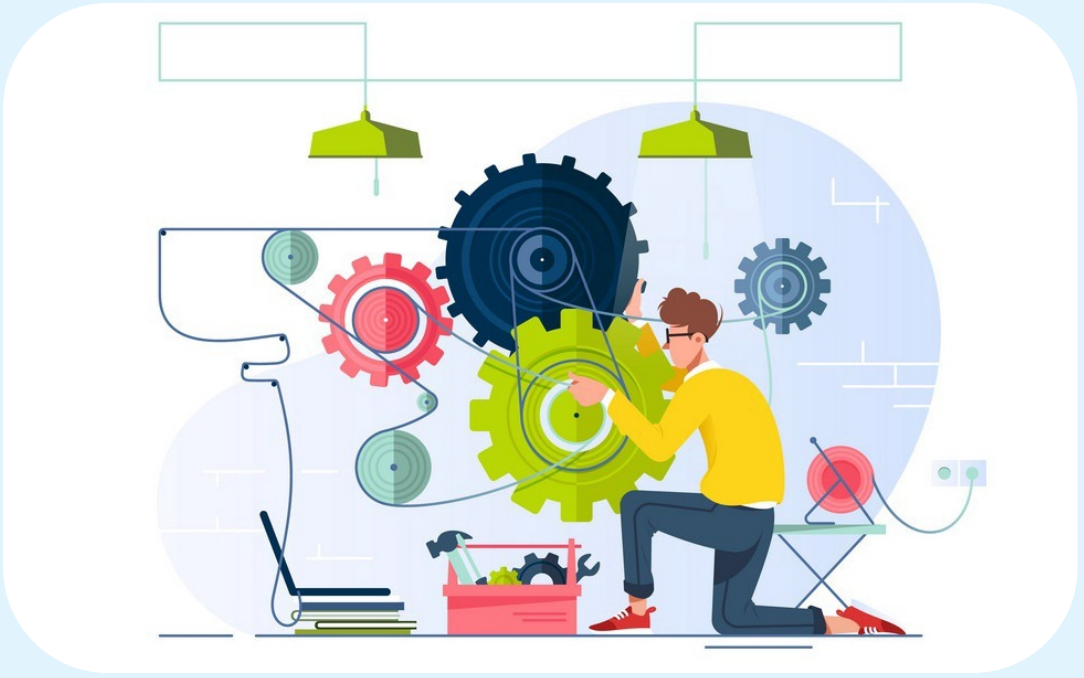
MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR OPEN & DISTANCE EDUCATION

Object Oriented Programing Concepts

Bachelor of Computer Applications (BCA)
Semester - 2



SELF LEARNING MATERIAL

MATS UNIVERSITY
www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

Bachelor of Computer Applications
ODL BCA DSC 04
Object Oriented Programing Concepts

Course Introduction	**1**
Module 1 **Object-oriented concepts**	**3**
Unit 1: Principles of OOP, Classes and Objects	**7**
Unit 2: Member Functions	**42**
Unit 3: Array, Memory Management	**49**
Module 2 **Functions, constructors, and destructors**	**78**
Unit 4: Memory Allocation of Objects, Friend Function	**79**
Unit 5: Constructors	**105**
Unit 6: Destructors	**116**
Module 3 **Operator overloading and inheritance**	**124**
Unit 7: Operator Overloading Basics	**125**
Unit 8: Types of Inheritance, Inheritance Implementation	**136**
Unit 9: Constructors in Derived Classes and Member Classes	**147**
Module 4 **Pointer, virtual function, and polymorphism**	**162**
Unit 10: Pointers in C++	**164**
Unit 11: Virtual Functions	**171**
Unit 12: Overloading and overriding	**176**
Module 5 **Console i/o operations and file handling**	**190**
Unit 13: Console-Based I/O Operations	**193**

Unit 14: File Handling in C++	**201**
References	**228**

COURSE DEVELOPMENT EXPERT COMMITTEE

Prof. (Dr.) K. P. Yadav, Vice Chancellor, MATSUniversity, Raipur, Chhattisgarh

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Sanjay Kumar, Professor and Dean, Pt. Ravishankar Shukla University, Raipur, Chhattisgarh

Prof. (Dr.) Jatinder kumar R. Saini, Professor and Director, Symbiosis Institute of Computer Studies and Research, Pune

Dr. Ronak Panchal, Senior Data Scientist, Cognizant, Mumbai

Mr. Saurabh Chandrakar, Senior Software Engineer, Oracle Corporation, Hyderabad

COURSECOORDINATOR

Prof. (Dr.) Bhavna Narain, Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Prof. (Dr.) Bhavna Narain, Professor and Mrs. Hemlata Patel, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-93-49916-99-9

@MATS Centre for Distance and Online Education, MATSUniversity, Village-Gullu, Aarang, Raipur-(Chhattisgarh)

Printed & published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. MeghanadhuduKatabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Disclaimer-Publisher of this printing material is not responsible for any error or dispute from contents of this course material, this completely depends onAUTHOR'SMANUSCRIPT.

Printedat: TheDigital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

This course provides a comprehensive understanding of Object-Oriented programming (OOP), covering fundamental concepts, functions handling, operator overloading, inheritance, polymorphism, and file handling. It aims to build a strong foundation for designing and implementing efficient object- oriented applications.

Module 1: Object Oriented Concepts

This Module introduces the fundamental principles of Object- Oriented Programming including encapsulation, abstraction, inheritance, and polymorphism.

Module 2: Functions, Constructors, and Destructors

A Function play a crucial role in structuring code efficiently. This Module covers differently types of functions, function overloading, constructors, and destructors.it emphasizes their importance in managing object lifecycle and resource allocation.

Module 3: Operator Overloading and Inheritance

Operator Overloading enhances the functionality of operators to work with user-defined datatypes. This Module explores operator overloading techniques and their applications. Additionally, it delves into inheritance, covering types of inheritance and their impact on code reuse and hierarchy management.

Module 4: Pointers, Virtual Functions, and Polymorphism

Data Pointers are essential for dynamic memory management in OOP. This Module covers pointer concepts, virtual functions, and polymorphism, demonstrating how runtime behaviour can be modified dynamically though method overriding and function pointers.

Module 5: Console I/O Operations and File Handling

The Efficient data input and output are crucial in programming. This Module focuses on consolebased input/output operations and file handling techniques, including reading from and writing to files, handling Streams, and file manipulation for storage.

By This course equips learners with the necessary skills to effectively implement Object Oriented programming principles in real words applications.

MODULE 1
OBJECT-ORIENTED CONCEPTS

LEARNING OUTCOMES

- Understand the features and structure of a C++ program.
- Learn the fundamentals of Object-Oriented Programming (OOP) and its advantages.
- Understand the concepts of objects and classes in C++.
- Learn about member functions in a class.
- Understand the use of arrays within a class in C++.

Unit 1: Principles of OOP, Classes and Objects

1.1 Features and Structure of C++ Program

Extending the powers of C, the strong, flexible programming language C++ brings object-oriented elements. Originally created by Bjarne Stroustrup in the early 1980s, C++ has become among the most widely used programming languages nowadays. Let's examine its salient characteristics and organizational framework.

Core Features of C++

Multi-Paradigm Support

C++ supports multiple programming paradigms, making it exceptionally flexible:

- **Procedural programming**: Like C, it allows structured, top-down code organization
- **Object-oriented programming**: Supports classes, inheritance, polymorphism, and encapsulation
- **Generic programming**: Through templates, enabling type-independent code
- **Functional programming**: With lambda expressions and higher-order functions

This versatility allows developers to choose the most appropriate approach for each situation rather than forcing a single paradigm.

Performance and Efficiency

- **Low-level memory manipulation**: Direct access to memory through pointers
- **Compile-time polymorphism**: Using templates and function overloading
- **Zero-overhead principle**: You don't pay for features you don't use
- **Efficient memory management**: Control over allocation and deallocation
- **Inline functions**: Reducing function call overhead
- **RAII (Resource Acquisition Is Initialization)**: Efficient resource management

These features make C++ suitable for performance-critical applications including game development, real-time systems, and high-frequency trading.

Rich Standard Library

The C++ Standard Library provides a comprehensive collection of classes and functions:

- **Containers**: vector, list, map, set, etc.
- **Algorithms**: sort, find, transform, etc.
- **Iterators**: For traversing container elements
- **Strings**: Sophisticated string handling capabilities
- **Streams**: Input/output operations
- **Smart pointers**: For safer memory management (shared_ptr, unique_ptr)
- **Utilities**: pair, tuple, optional, any, variant
- **Thread support**: For concurrent programming
- **Regular expressions**: Pattern matching functionality

Major component of the C++ Standard Library, the Standard Template Library (STL) uses generic programming approaches to accomplish many of these capabilities.

Backward Compatibility

C++ maintains strong compatibility with C, allowing:

- Compilation of most C code as C++
- Integration of legacy C code with new C++ code
- Use of C libraries within C++ programs

This compatibility has been instrumental in C++'s adoption and longevity.

Modern Features

Since C++11, many modern features have been added:

- **Auto type deduction**: Simplifying variable declarations
- **Range-based for loops**: Easier container iteration
- **Lambda expressions**: Anonymous functions
- **Move semantics**: Optimizing resource transfers
- **Smart pointers**: Automated memory management
- **Concepts** (C++20): Constraints on template parameters
- **Modules** (C++20): Better organization of code
- **Coroutines** (C++20): Simplified asynchronous programming

These features have significantly modernized C++, making it more expressive and safer while maintaining its performance characteristics.

Structure of a C++ Program

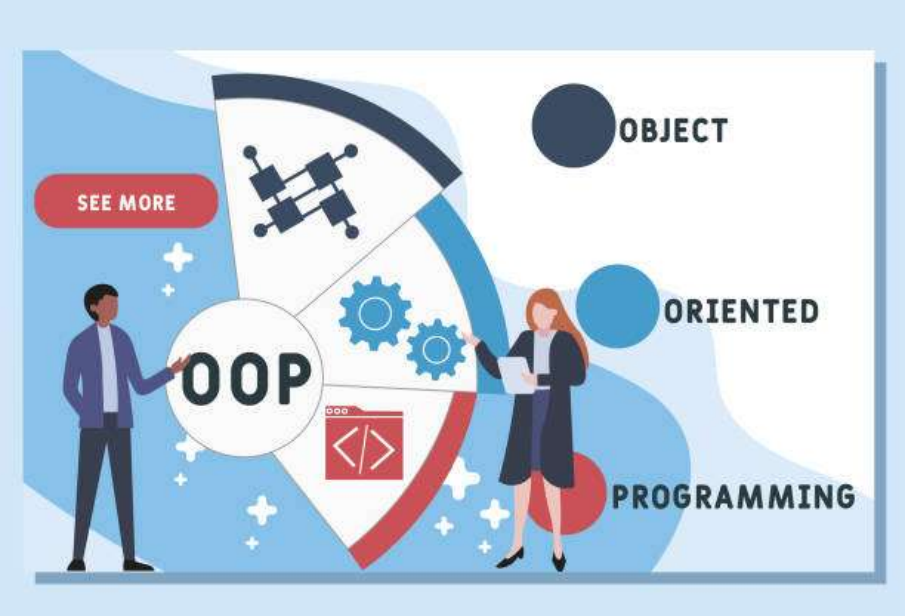


Figure 1 Structure of OOPS
[Source: https://www.istockphoto.com]

A typical C++ program consists of several components:

Header Files and Includes

```
#include <iostream>   // Standard library header
#include <vector>     // Container header
#include "myheader.h" // User-defined header
```

The #include directive brings in declarations from:

- Standard library headers (enclosed in <>)
- User-defined headers (enclosed in quotation marks)

These headers contain declarations of functions, classes, and variables that will be used in the program.

Namespaces

```
using namespace std;  // Using entire namespace (generally avoided in practice)

// Preferred approach:
using std::cout;
using std::vector;

// Or accessing with scope resolution operator:
std::string myString;
```

Namespaces prevent naming conflicts by grouping related declarations under a common name. The Standard Library components are found in std namespace.

Main Function

```
int main() {
   // Program execution begins here

   // Code statements

   return 0;  // Return value indicates execution status
}
```

Every C++ application has to include a main() function as the entrance point. Execution begins at the first statement in main() and the return value indicates the program's execution status (0 typically indicates successful execution).

Functions and Methods

```
// Function declaration
return_typefunction_name(parameter_list);

// Function definition
return_typefunction_name(parameter_list) {
   // Function body
   return value;  // Optional, depends on return_type
}
```

Functions encapsulate reusable code blocks. They might be either part of a class or stand-alone free functions.

(methods).

Classes and Objects

```
// Class declaration
class ClassName {
private:
   // Private members
   int privateData;

public:
   // Constructor
ClassName(int data) : privateData(data) {}
```

```
    // Methods
    void method1() {
        // Method implementation
    }

    int method2(double parameter);
};

// Method definition outside class
int ClassName::method2(double parameter) {
    // Implementation
    return privateData;
}

// Creating objects
ClassName object1(10);          // Stack allocation
ClassName* object2 = new ClassName(20);  // Heap allocation
```

Classes are the fundamental building blocks of object-oriented programming in C++. They encapsulate data (members) and behavior (methods) into a single Module .

Comments

```
// Single-line comment

/*
   Multi-line
   comment
*/

/// Documentation comment for tools like Doxygen
```

Comments explain the code's purpose and functionality, making it more maintainable.

A Complete Example

Here's a complete C++ program demonstrating these structural elements:

```
// Include standard library headers
#include <iostream>
#include <string>
#include <vector>
```

```
// User-defined class
class Student {
private:
   std::string name;
   int id;
   std::vector<double> grades;

public:
   // Constructor
   Student(const std::string&studentName, int studentId)
      : name(studentName), id(studentId) {}

   // Methods
   void addGrade(double grade) {
grades.push_back(grade);
   }

   double getAverageGrade() const {
      if (grades.empty()) return 0.0;

      double sum = 0.0;
      for (double grade : grades) {
         sum += grade;
      }
      return sum / grades.size();
   }

   void displayInfo() const {
      std::cout<< "Student: " << name << " (ID: " << id << ")\n";
      std::cout<<  "Average  Grade:  "  <<getAverageGrade()  <<
std::endl;
   }
};

// Function declaration
void processStudents(const std::vector<Student>& students);
```

```
// Main function - program entry point
int main() {
    // Creating objects
    Student alice("Alice Smith", 12345);
    Student bob("Bob Johnson", 67890);

    // Using object methods
alice.addGrade(85.5);
alice.addGrade(92.0);
alice.addGrade(88.5);

bob.addGrade(77.0);
bob.addGrade(81.5);

    // Storing objects in a container
    std::vector<Student>studentList = {alice, bob};

    // Function call
processStudents(studentList);

    return 0;  // Indicate successful execution
}

// Function definition
void processStudents(const std::vector<Student>& students) {
    std::cout<< "Student Information:\n";
    std::cout<< "-------------------\n";

    for (const auto& student : students) {
student.displayInfo();
        std::cout<< "-------------------\n";
    }
}
```

This example demonstrates:

- Header inclusion
- Class definition with private data and public methods
- Function declaration and definition
- Container usage

- Object creation and manipulation
- Program flow through the main function

Compilation and Execution Process

Understanding how C++ programs are processed is essential:

1. **Preprocessing**: The preprocessor handles directives like #include and #define
2. **Compilation**: The compiler translates source code into object files
3. **Linking**: The linker combines object files and libraries into an executable
4. **Execution**: The operating system loads and runs the executable

This multi-stage process allows for separate compilation of program components, facilitating modular development of large applications.

1.1 Object-Oriented Programming Concepts and Advantages

Using "objects," or instances of classes, object-oriented programming (OOP) is a paradigm of design for computer programs and applications. One of the first languages to become really popular with OOP was Python. Let us investigate the main ideas and benefits of this method.

Figure 2 Concept of OOP's
[Source: https://www.shutterstock.com]

Core OOP Concepts in C++

Classes and Objects

Classes are user-defined data types that serve as blueprints for objects:

class Car {

```
private:
    std::string make;
    std::string model;
    int year;
    double fuelLevel;

public:
    Car(std::string mk, std::string mdl, int yr)
        : make(mk), model(mdl), year(yr), fuelLevel(100.0) {}

    void drive(double distance) {
fuelLevel -= distance * 0.05; // Simulate fuel consumption
    }

    void refuel() {
fuelLevel = 100.0;
    }

    double getFuelLevel() const {
        return fuelLevel;
    }
};
```

Objects are instances of classes:

```
Car myCar("Toyota", "Corolla", 2023);
Car yourCar("Honda", "Civic", 2022);

myCar.drive(50);  // Each object maintains its own state
yourCar.drive(25);
```

Classes define both:

- **Attributes** (data members): The state of an object
- **Methods** (member functions): The behavior of an object

Encapsulation

Encapsulation is the bundling of data and methods that operate on that data into a single Module (the class), while restricting direct access to some of the object's components:

```
class BankAccount {
private:
```

```
    double balance;      // Private data - hidden from outside
    std::string accountNumber;

public:
    // Public interface - controlled access to private data
BankAccount(std::string accNum, double initialDeposit)
        : accountNumber(accNum), balance(initialDeposit) {}

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;
    }

    double getBalance() const {
        return balance;
    }
};
```

Key elements of encapsulation:

- **Access specifiers**:
  - **private:** Accessible only within the class
  - **protected:** Accessible within the class and its derived classes
  - **public:** Accessible from anywhere
- **Benefits**:
  - Protects data from unintended access and modification
  - Enables data validation before changing state
  - Allows implementation changes without affecting client code

- Reduces system complexity by hiding implementation details

Inheritance

```
// Base class
class Vehicle {
protected:
   std::string make;
   std::string model;
   int year;

public:
   Vehicle(std::string mk, std::string mdl, int yr)
      : make(mk), model(mdl), year(yr) {}

   void displayInfo() const {
      std::cout<< year << " " << make << " " << model;
   }

   virtual void startEngine() {
      std::cout<< "Vehicle engine started\n";
   }
};

// Derived class
class ElectricVehicle : public Vehicle {
private:
   int batteryCapacity;

public:
ElectricVehicle(std::string mk, std::string mdl, int yr, int battery)
      : Vehicle(mk, mdl, yr), batteryCapacity(battery) {}

   void displayInfo() const {
      Vehicle::displayInfo();  // Call base class method
      std::cout<< " (Battery: " <<batteryCapacity<< " kWh)\n";
   }

   // Override base class method
```

```
    void startEngine() override {
        std::cout<< "Electric motor initialized\n";
    }

    void chargeBattery() {
        std::cout<< "Charging battery...\n";
    }
};
```

Types of inheritance in C++:

- **Single inheritance**: A class inherits from one base class
- **Multiple inheritance**: A class inherits from multiple base classes
- **Multilevel inheritance**: A class inherits from a derived class
- **Hierarchical inheritance**: Multiple classes inherit from a single base class
- **Hybrid inheritance**: Combination of multiple inheritance types

Benefits of inheritance:

- Code reusability
- Establishment of hierarchical relationships
- Creation of specialized classes from general ones
- Implementation of "is-a" relationships

Polymorphism

Polymorphism lets objects of many kinds be handled as objects of a shared base class, with varied behaviors depending on their real derived type:

```
// Base class with virtual function
class Shape {
public:
    virtual double area() const = 0;  // Pure virtual function
    virtual void draw() const {
        std::cout<< "Drawing a shape\n";
    }
    virtual ~Shape() {}  // Virtual destructor
};

// Derived classes
```

```
class Circle : public Shape {
private:
   double radius;

public:
   Circle(double r) : radius(r) {}

   double area() const override {
      return 3.14159 * radius * radius;
   }

   void draw() const override {
      std::cout<< "Drawing a circle\n";
   }
};

class Rectangle : public Shape {
private:
   double width, height;

public:
   Rectangle(double w, double h) : width(w), height(h) {}

   double area() const override {
      return width * height;
   }

   void draw() const override {
      std::cout<< "Drawing a rectangle\n";
   }
};

// Using polymorphism
void processShape(const Shape& shape) {
   std::cout<< "Area: " <<shape.area() << std::endl;
shape.draw();
}
```

```
int main() {
   Circle circle(5.0);
   Rectangle rectangle(4.0, 6.0);

processShape(circle);      // Uses Circle's implementations
processShape(rectangle);   // Uses Rectangle's implementations

   // Using polymorphism with pointers
   Shape* shapes[] = {
      new Circle(3.0),
      new Rectangle(2.0, 5.0)
   };

   for (Shape* shape : shapes) {
processShape(*shape);
      delete shape;  // Clean up dynamic memory
   }

   return 0;
}
```

C++ supports two main types of polymorphism:

1. **Compile-time (static) polymorphism**:
   - **Function overloading:** Multiple functions with the same name but different parameters
   - **Operator overloading:** Customizing operator behavior for user-defined types
   - **Template specialization:** Different implementations based on type
2. **Runtime (dynamic) polymorphism**:
   - **Virtual functions:** Methods that can be overridden in derived classes
   - **Pure virtual functions:** Abstract methods that must be implemented by derived classes
   - **Virtual destructors:** Ensure proper cleanup of derived objects

Benefits of polymorphism:

- Flexibility in designing and extending object hierarchies

- Uniform interface for different implementations
- Code that can work with new derived classes without modification
- Support for the "open-closed principle" (open for extension, closed for modification)

Abstraction

Abstraction is the simplification of difficult systems by use of classifications based on fundamental characteristics and behaviors:

```
// Abstract class (contains at least one pure virtual function)
class DatabaseConnection {
protected:
   std::string connectionString;
   bool isConnected;

public:
DatabaseConnection(const std::string&connStr)
      : connectionString(connStr), isConnected(false) {}

   // Pure virtual functions - must be implemented by derived classes
   virtual bool connect() = 0;
   virtual bool disconnect() = 0;
   virtual bool executeQuery(const std::string& query) = 0;

   virtual ~DatabaseConnection() {
      if (isConnected) disconnect();
   }

   bool isActive() const {
      return isConnected;
   }
};

// Concrete implementation
class MySQLConnection : public DatabaseConnection {
public:
MySQLConnection(const std::string&connStr)
      : DatabaseConnection(connStr) {}
```

```
    bool connect() override {
        // MySQL-specific connection logic
        std::cout<< "Connecting to MySQL database...\n";
isConnected = true;
        return true;
    }

    bool disconnect() override {
        // MySQL-specific disconnection logic
        if (isConnected) {
            std::cout<< "Disconnecting from MySQL database...\n";
isConnected = false;
            return true;
        }
        return false;
    }

    bool executeQuery(const std::string& query) override {
        if (!isConnected) return false;

        std::cout<< "Executing MySQL query: " << query << std::endl;
        return true;
    }
};
```

Key aspects of abstraction:

- **Hiding implementation details**: Focus on what an object does, not how it does it
- **Abstract classes**: Classes containing at least one pure virtual function
- **Interfaces**: Pure abstract classes (all functions are pure virtual)
- **Implementation independence**: Client code depends on abstract interfaces, not concrete implementations

Advantages of Object-Oriented Programming

Modularity

OOP promotes modularity by encapsulating code and data into self-contained Module s:

- **Benefits**:
  - Easier maintenance and debugging
  - Independent development of modules
  - Code can be understood in smaller, manageable pieces
  - Facilitates team development
  - Simplifies testing

```
// Example of a self-contained module
class Logger {
private:
   std::string logFile;
   std::ofstreamfileStream;
LogLevelminLevel;

   void formatLogMessage(LogLevel level, const std::string& message);

public:
enumLogLevel { DEBUG, INFO, WARNING, ERROR, CRITICAL };

   Logger(const std::string& filename, LogLevel level = INFO);
   ~Logger();

   void setLogLevel(LogLevel level);
   void log(LogLevel level, const std::string& message);
```

```
    void debug(const std::string& message);
    void info(const std::string& message);
    void warning(const std::string& message);
    void error(const std::string& message);
    void critical(const std::string& message);
};
```

Reusability

OOP facilitates code reuse through inheritance, composition, and libraries:

- **Inheritance-based reuse**: Deriving new classes from existing ones
- **Composition-based reuse**: Building classes that contain instances of other classes
- **Class libraries**: Collections of reusable classes for common functionality

```
// Example of composition-based reuse
class Address {
public:
    std::string street;
    std::string city;
    std::string state;
    std::string zipCode;

    Address(std::string st, std::string c, std::string s, std::string z)
        : street(st), city(c), state(s), zipCode(z) {}
};

class Person {
protected:
    std::string name;
    Address homeAddress;  // Composition

public:
    Person(std::string n, Address addr)
        : name(n), homeAddress(addr) {}

    void displayInfo() const {
        std::cout<< "Name: " << name << "\n";
```

```
        std::cout<< "Address: " <<homeAddress.street<< ", "
<<homeAddress.city<< ", " <<homeAddress.state
<< " " <<homeAddress.zipCode<< std::endl;
    }
};

class Employee : public Person {  // Inheritance
private:
    int employeeId;
    double salary;
    Address workAddress;      // Another instance of Address
(composition)

public:
    Employee(std::string n, Address home, int id, double sal, Address
work)
        : Person(n, home), employeeId(id), salary(sal),
workAddress(work) {}

    void displayEmployeeInfo() const {
displayInfo();  // Reuse Person's method
        std::cout<< "Employee ID: " <<employeeId<< "\n";
        std::cout<< "Work Address: " <<workAddress.street<< ", "
<<workAddress.city<< ", " <<workAddress.state
<< " " <<workAddress.zipCode<< std::endl;
    }
};
```

Maintainability

OOP improves code maintainability through:

- **Encapsulation**: Changes to implementation don't affect client code
- **Single Responsibility Principle**: Classes have one reason to change
- **Loose coupling**: Limited dependencies between components
- **High cohesion**: Related functionality is grouped together

```
// Before refactoring (poor maintainability)
class UserManager {
public:
```

```
    void registerUser(std::string username, std::string password) {
        // Validate input
        if (username.empty() || password.empty()) return;

        // Hash password
        std::string hashedPassword = hashFunction(password);

        // Store in database
        std::string query = "INSERT INTO users (username, password)
VALUES ('"
                            + username + "', '" + hashedPassword + "')";
executeSQL(query);

        // Send welcome email
        std::string emailBody = "Welcome, " + username + "!";
sendEmail(username + "@example.com", "Welcome", emailBody);

        // Log activity
logActivity("User registered: " + username);
    }

private:
    std::string hashFunction(const std::string& input) { /* ... */ }
    void executeSQL(const std::string& query) { /* ... */ }
    void sendEmail(const std::string& to, const std::string& subject,
const std::string& body) { /* ... */ }
    void logActivity(const std::string& activity) { /* ... */ }
};

// After refactoring (better maintainability)
class PasswordHasher {
public:
    std::string hashPassword(const std::string& password) { /* ... */ }
};

class DatabaseManager {
public:
```

```
    void storeUser(const std::string& username, const
std::string&hashedPassword) { /* ... */ }
};

class EmailService {
public:
    void sendWelcomeEmail(const std::string& username) { /* ... */ }
};

class ActivityLogger {
public:
    void logUserRegistration(const std::string& username) { /* ... */ }
};

class UserManager {
private:
PasswordHasherpasswordHasher;
DatabaseManagerdatabaseManager;
EmailServiceemailService;
ActivityLoggeractivityLogger;

public:
    void registerUser(std::string username, std::string password) {
        if (username.empty() || password.empty()) return;

        std::string                hashedPassword                =
passwordHasher.hashPassword(password);
databaseManager.storeUser(username, hashedPassword);
emailService.sendWelcomeEmail(username);
activityLogger.logUserRegistration(username);
    }
};
```

Extensibility

OOP designs are naturally extensible:

- **Open-Closed Principle**: Classes are open for extension but closed for modification
- **Interface-based programming**: Code to interfaces, not implementations

- **Pluggable components**: New implementations can be substituted without changing client code

```
// Base plugin interface
class TextProcessor {
public:
   virtual void processText(std::string& text) = 0;
   virtual ~TextProcessor() {}
};

// Concrete implementations
class SpellChecker : public TextProcessor {
public:
   void processText(std::string& text) override {
      std::cout<< "Spell checking text...\n";
      // Implementation
   }
};

class GrammarChecker : public TextProcessor {
public:
   void processText(std::string& text) override {
      std::cout<< "Grammar checking text...\n";
      // Implementation
   }
};

class PlagiarismDetector : public TextProcessor {
public:
   void processText(std::string& text) override {
      std::cout<< "Detecting plagiarism...\n";
      // Implementation
   }
};

// Extensible document processor using plugins
class DocumentProcessor {
private:
   std::vector<std::unique_ptr<TextProcessor>> processors;
```

```
public:
   void addProcessor(std::unique_ptr<TextProcessor> processor) {
processors.push_back(std::move(processor));
   }

   void processDocument(std::string& document) {
      for (auto& processor : processors) {
         processor->processText(document);
      }
   }
};

// Usage
int main() {
DocumentProcessordocProcessor;

   // Add existing processors
docProcessor.addProcessor(std::make_unique<SpellChecker>());
docProcessor.addProcessor(std::make_unique<GrammarChecker>());

   // Later, add a new processor without changing DocumentProcessor

docProcessor.addProcessor(std::make_unique<PlagiarismDetector>())
;

   std::string document = "Sample text to process";
docProcessor.processDocument(document);

   return 0;
}
```

Reliability and Robustness

OOP enhances software reliability through:

- **Data hiding**: Prevents unauthorized access and modification
- **Strong typing**: Catch errors at compile time
- **Exception handling**: Structured approach to error management

- **Invariant maintenance**: Classes can ensure their data remains valid

```
// Example of robust class design
class RationalNumber {
private:
    int numerator;
    int denominator;

    // Helper to reduce fraction to lowest terms
    void reduce() {
        if (numerator == 0) {
            denominator = 1;
            return;
        }

        int gcd = findGCD(std::abs(numerator), std::abs(denominator));
        numerator /= gcd;
        denominator /= gcd;

        // Ensure denominator is positive
        if (denominator < 0) {
            numerator = -numerator;
            denominator = -denominator;
        }
    }

    // Calculate greatest common divisor
    int findGCD(int a, int b) const {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

public:
    // Constructors with validation
```

```
RationalNumber(int num = 0, int denom = 1) {
        if (denom == 0) {
            throw std::invalid_argument("Denominator cannot be zero");
        }

        numerator = num;
        denominator = denom;
        reduce();
    }

    // Arithmetic operations
RationalNumber add(const RationalNumber& other) const {
        int num = numerator * other.denominator + other.numerator *
denominator;
        int denom = denominator * other.denominator;
        return RationalNumber(num, denom);
    }

    // More operations...

    // Safe access methods
    int getNumerator() const { return numerator; }
    int getDenominator() const { return denominator; }
    double toDouble() const { return static_cast<double>(numerator) /
denominator; }

    std::string toString() const {
        if (denominator == 1) {
            return std::to_string(numerator);
        }
        return          std::to_string(numerator)          +          "/"          +
std::to_string(denominator);
    }
};
```

Real-World Modeling

OOP naturally maps to real-world entities and relationships:

- **Object correspondence**: Software objects represent real entities

- **Natural hierarchies**: Inheritance models "is-a" relationships
- **Has-a relationships**: Composition models containment
- **Behavior modeling**: Methods capture actions and operations

```
// Real-world university modeling example
class Person {
protected:
   std::string name;
   int age;
   std::string id;

public:
   Person(std::string n, int a, std::string i)
      : name(n), age(a), id(i) {}

   virtual void displayInfo() const {
      std::cout<< "Name: " << name << ", ID: " << id << std::endl;
   }

   virtual ~Person() {}
};

class Student : public Person {
private:
   std::string major;
   double gpa;
   std::vector<std::string>enrolledCourses;

public:
   Student(std::string n, int a, std::string i, std::string m)
      : Person(n, a, i), major(m), gpa(0.0) {}

   void enrollInCourse(const std::string& course) {
enrolledCourses.push_back(course);
   }

   void displayInfo() const override {
      Person::displayInfo();
      std::cout<< "Type: Student, Major: " << major << std::endl;
```

```
        std::cout<< "Enrolled courses: ";
        for (const auto& course : enrolledCourses) {
            std::cout<< course << " ";
        }
        std::cout<< std::endl;
    }
};

class Professor : public Person {
private:
    std::string department;
    std::vector<std::string>taughtCourses;

public:
    Professor(std::string n, int a, std::string i, std::string d)
        : Person(n, a, i), department(d) {}

    void assignCourse(const std::string& course) {
taughtCourses.push_back(course);
    }

    void displayInfo() const override {
        Person::displayInfo();
        std::cout<< "Type: Professor, Department: " << department <<
std::endl;
        std::cout<< "Teaching: ";
        for (const auto& course : taughtCourses) {
            std::cout<< course << " ";
        }
        std::cout<< std::endl;
    }
};

class Course {
private:
    std::string courseCode;
    std::string title;
    Professor* instructor;
```

```
    std::vector<Student*> students;

public:
    Course(std::string code, std::string t, Professor* prof)
        : courseCode(code), title(t), instructor(prof) {
        if (instructor) {
            instructor->assignCourse(courseCode);
        }
    }

    void addStudent(Student* student) {
students.push_back(student);
        student->enrollInCourse(courseCode);
    }

    void displayCourseInfo() const {
        std::cout<< "Course: " <<courseCode<< " - " << title <<
std::endl;
        if (instructor) {
            std::cout<< "Taught by: " << instructor->getName() <<
std::endl;
        }
        std::cout<< "Enrolled students: " <<students.size() << std::endl;
    }
};

class Department {
private:
    std::string name;
    std::vector<Professor*> faculty;
    std::vector<Course*> offerings;

public:
    Department(const std::string& n) : name(n) {}

    void addProfessor(Professor* prof) {
faculty.push_back(prof);
    }
```

```
    void addCourse(Course* course) {
offerings.push_back(course);
    }

    void displayDepartmentInfo() const {
        std::cout<< "Department: " << name << std::endl;
        std::cout<< "Number of faculty: " <<faculty.size() << std::endl;
        std::cout<< "Number of courses: " <<offerings.size() << std::endl;
    }
};

class University {
private:
    std::string name;
    std::vector<Department*> departments;

public:
    University(const std::string& n) : name(n) {}

    void addDepartment(Department* dept) {
departments.push_back(dept);
    }

    void displayUniversityInfo() const {
        std::cout<< "University: " << name << std::endl;
        std::cout<< "Departments:
```

1.2 Object and Class

What is an Object?

An object is a real-world entity that has two main properties:

- **State (Attributes/Data Members):** The characteristics of an object (e.g., color, model, speed of a Car object).
- **Behavior (Methods/Member Functions):** The actions an object can perform (e.g., startEngine (), accelerate() for a Car object).

Example of an Object:

A Car object has attributes like color, brand, model, speed, and behaviors like start(), brake(), accelerate(), etc.

An object is an instance of a class. It represents a specific entity with its own unique attribute values.

Example of an Object in C++

cpp

Copy

Edit

```
int main() {
   Car car1;  // Creating an object of class Car
   car1.brand = "Toyota";
   car1.model = "Corolla";
   car1.speed = 0;

   car1.accelerate(50);
   car1.display();

   return 0;
}
```

Table 1.1: Relationship between Class and Object

Class (Blueprint)	**Object (Instance of Class)**
Defines attributes and behavior	Stores actual data and executes behavior
Acts as a template	Created based on the class
Exists once in the program	Multiple objects can be created from one class

An **object** is an instance of a class. Using a class, multiple objects can be created, each with its own values and behavior.

Example in C++

cpp

```
#include <iostream>
using namespace std;

class Car {
private:
   string brand;
   int speed;

public:
   // Constructor
   Car(string b, int s) {
      brand = b;
```

```
        speed = s;
    }

    // Member function
    void showDetails() {
cout<< "Brand: " << brand << ", Speed: " << speed << " km/h"
<<endl;
    }
};

int main() {
    Car car1("Toyota", 120);
    Car car2("Honda", 140);

    car1.showDetails();
    car2.showDetails();

    return 0;
}
```

Output:

yaml

Brand: Toyota, Speed: 120 km/h

Brand: Honda, Speed: 140 km/h

What is a Class?

A class is a blueprint/template for creating objects. It defines:

- The attributes (data members) an object will have.
- The methods (functions) the object can perform.

Example in C++

Cpp

```
class Car {
private:
  string brand;
   string model;
  int speed;
public:
   // Constructor
 Car(string b, string m, int s) {
    brand = b;
```

```
 model = m;
 speed = s

  }
    // Method to accelerate
void accelerate(int value) {
  speed += value;
}

 // Method to display details
void display() {
cout<< "Brand: " << brand << ", Model: " << model << ", Speed: "
<< speed <<endl
  }
};
int main() {

 Car car1("Toyota", "Corolla", 0); // Creating an object
 car1.accelerate(50);
car1.display();
    return 0;
}
```

A class is a blueprint or template that defines the structure and behavior of objects. It contains attributes (data members) and methods (functions) that define how the object behaves.

Example of a Class in C++

Cp

```
class Car {
public:
   string brand;
string model;
    int speed;
  void accelerate(int value) {
    speed += value;
  }
   void display() {
cout<< "Brand: " << brand << ", Model: " << model << ", Speed: "
<< speed << " km/h" <<endl;
```

```
}
};
```

Here, Car is a class with three attributes (brand, model, speed) and two methods (accelerate() and display()). A **class** is a blueprint or template used to create objects. It groups data (attributes) and functions (member functions) that operate on that data.

Unit 2: Member Functions

1.4 Member Functions

What are Member Functions?

Member functions are functions defined inside a class that define the behavior of objects.

Types of Member Functions

- **Simple Member Functions:** Perform basic actions.
- **Inline Functions:** Defined inside the class for efficiency.
- **Const Functions**: Do not modify class attributes.
- **Virtual Functions:** Used for polymorphism in inheritance.
- **Pure Virtual Functions:** Used in abstract classes.
- **Static Member Functions:** Belong to the class, not to any object.
- **Function Overloading:** Multiple functions with the same name but different parameters.
- **Function Overriding:** Redefining a function in a derived class.
- **Simple Member Functions** – Basic functions that perform specific tasks.
- **Inline Functions** – Defined inside the class for efficiency.
- **Const Functions** – Cannot modify class attributes.
- **Virtual Functions** – Used for **polymorphism** in inheritance.
- **Pure Virtual Functions** – Used in **abstract classes**.
- **Static Member Functions** – Belong to the class, not an object.
- **Function Overloading** – Multiple functions with the same name but different parameters.
- **Function Overriding** – Redefining a function in a derived class.

Example of Member Functions

Defining Member Functions Inside a Class

cpp

Copy

Edit

```
class Car {
public:
   int speed;
```

```
    void accelerate(int value) {
        speed += value;
    }
};
```

Defining Member Functions Outside a Class

cpp

Copy

Edit

```
class Car {
public:
    int speed;
    void accelerate(int value); // Function prototype
};

// Function definition outside the class using ::
void Car::accelerate(int value) {
    speed += value;
}
```

Virtual and Pure Virtual Functions (Polymorphism Example)

cpp

Copy

Edit

```
class Vehicle {
public:
    virtual void honk() {  // Virtual function
cout<< "Vehicle Honk!" <<endl;
    }
};

class Car : public Vehicle {
public:
    void honk() override { // Function Overriding
cout<< "Car Honk!" <<endl;
    }
};
```

If we use Vehicle *v = new Car(); v->honk();, it will call Car’s honk() method due to dynamic binding.

A **member function** is a function defined inside a class. It performs operations on the class's attributes.

Defining Member Functions Inside a Class

cpp

```
class Car {
public:
   int speed;

   void accelerate(int value) {
      speed += value;
   }
};
```

Member functions are functions defined within a class that operate on the objects of that class.

cpp

```
class Rectangle {
public:
   int length, width;

   void setValues(int l, int w) {
      length = l;
      width = w;
   }

   int area() {
      return length * width;
   }
};
```

Defining Member Functions Outside a Class

cpp

```
class Car {
public:
   int speed;
   void accelerate(int value); // Function prototype
};

// Function definition outside the class using ::
void Car::accelerate(int value) {
```

```
    speed += value;
}
cpp
#include <iostream>
using namespace std;

class Rectangle {
private:
    int length, width;

public:
    void setValues(int, int);
    int area();
};

// Defining member functions outside the class
void Rectangle::setValues(int l, int w) {
    length = l;
    width = w;
}

int Rectangle::area() {
    return length * width;
}

int main() {
    Rectangle rect;
rect.setValues(5, 10);
cout<< "Area: " <<rect.area() <<endl;
    return 0;
}
```

Virtual and Pure Virtual Functions (Polymorphism Example)

```
cpp
class Vehicle {
public:
    virtual void honk() {  // Virtual function
cout<< "Vehicle Honk!" <<endl;
    }
```

```
};

class Car : public Vehicle {
public:
   void honk() override { // Function Overriding
cout<< "Car Honk!" <<endl;
   }
};
```

If we use Vehicle *v = new Car(); v->honk();, it will call **Car's honk()** method due to **dynamic binding**.

Output:

makefile

Area: 50

Constructors and Destructors

(A) Constructor

A **constructor** is a special function that gets called automatically when an object of a class is created.

(1) Default Constructor

cpp

```
class Car {
public:
   Car() { // No parameters
cout<< "Car object created!" <<endl;
   }
};
```

(2) Parameterized Constructor

cpp

```
class Car {
private:
   string brand;

public:
   Car(string b) { // Constructor with one parameter
      brand = b;
   }

   void showBrand() {
cout<< "Car Brand: " << brand <<endl;
```

```
    }
};
```

(3) Copy Constructor

```
cpp
class Car {
private:
    string brand;

public:
    Car(string b) {
        brand = b;
    }

    Car(const Car& c) { // Copy constructor
        brand = c.brand;
    }

    void showBrand() {
cout<< "Car Brand: " << brand <<endl;
    }
};
```

(B) Destructor

A **destructor** is a special function that is automatically called when an object is destroyed.

Example of Destructor in C++

```
cpp
class Car {
public:
    Car() {
cout<< "Car object created!" <<endl;
    }

    ~Car() { // Destructor
cout<< "Car object destroyed!" <<endl;
    }
};
```

Constructors and Destructors in Java

Constructor in Java

java

```
class Car {
   String brand;

   Car(String b) {  // Constructor
      brand = b;
   }

   void showBrand() {
System.out.println("Car Brand: " + brand);
   }

   public static void main(String[] args) {
      Car car1 = new Car("Toyota");
      car1.showBrand();
   }
}
```

Destructor in Java (Using Finalizer)

```
class Car {
   protected void finalize() throws Throwable {
System.out.println("Car object is being destroyed");
   }

   public static void main(String[] args) {
      Car car = new Car();
      car = null; // Mark object for garbage collection
System.gc(); // Request garbage collection
   }
}
```

1.5 Array within the Class

The first and one of the basic concepts of OOP have been the inclusion of arrays into a class structure. Embedding array as class members allows us to create powerful data containers capable of holding collections of values, all with the added benefits of encapsulation, inheritance, and polymorphism that class members bring to the table. This helps developers to write a very good structured code, flexible and manageable code infrastructure that can perform complex operations on data relationships.

Basic Concepts

What is an Array?

It means that an array is series of element which usually be stored in memory locations that are placed one after another directly. It provides a way to store and manipulate multiple values in a single variable name, with individual elements accessed via indices.

What is a Class?

In object-oriented programming, a class is a blueprint for creating objects. It specifies the properties (attributes) and behaviors (methods) that objects of that class will have. Classes are used to organize the data functionality that you require.

Arrays as Class Members

Integrating arrays within a class means that we define an array as a member variable (or attribute) of that class. V simple app that can also be used as a tiny example app for data collection.

Implementation Approaches

Static Arrays within Classes

Static arrays have fixed sizes determined at compile time. When included within a class, they provide a predictable memory footprint for each object instance.

```
public class StudentRoster {
   private String className;
   private String[] studentNames = new String[30]; // Fixed size array
   private int currentSize = 0;

   public void addStudent(String name) {
      if (currentSize<studentNames.length) {
```

```
studentNames[currentSize] = name;
currentSize++;
      }
   }

   public String getStudent(int index) {
      if (index >= 0 && index <currentSize) {
         return studentNames[index];
      }
      return null;
   }
}
```

Here, a given StudentRoster object has an array of size 30 that stores names of students. The class contains methods to add and get student information in a safe manner.

Dynamic Arrays within Classes

Dynamic arrays, on the other hand, can grow and shrink during runtime, providing flexibility when we do not know in advance how many elements we are going to have. In different programming languages, this gets implemented as built-in collection classes.

```
class ShoppingCart:
   def __init__(self):
self.items = []  # Dynamic array (list in Python)

   def add_item(self, item_name, price):
self.items.append({"name": item_name, "price": price})

   def remove_item(self, index):
      if 0 <= index <len(self.items):
         return self.items.pop(index)
      return None

   def calculate_total(self):
      return sum(item["price"] for item in self.items)
```

This Python ShoppingCart class maintains a dynamic list of items, allowing for an unlimited number of additions and removals.

Multi-dimensional Arrays

Classes can also contain multi-dimensional arrays to represent more complex data structures, such as matrices or grids.

```
class GameBoard {
private:
   char board[3][3]; // 3x3 grid for tic-tac-toe

public:
GameBoard() {
      // Initialize empty board
      for (int i = 0; i< 3; i++) {
         for (int j = 0; j < 3; j++) {
            board[i][j] = ' ';
         }
      }
   }

   bool makeMove(int row, int col, char player) {
      if (row >= 0 && row < 3 && col >= 0 && col < 3 &&
board[row][col] == ' ') {
         board[row][col] = player;
         return true;
      }
      return false;
   }

   char getCell(int row, int col) {
      if (row >= 0 && row < 3 && col >= 0 && col < 3) {
         return board[row][col];
      }
      return '\0';
   }
};
```

This C++ class uses a 2D array to represent a tic-tac-toe board, with methods to make moves and retrieve cell values.

Memory Management Considerations

Stack vs. Heap Allocation

When using arrays within classes, understanding memory allocation is crucial:

- **Stack allocation:** Arrays with fixed sizes declared directly within the class typically use stack memory, which is automatically managed but limited in size.
- **Heap allocation:** Dynamically allocated arrays use heap memory, which requires manual management in languages without garbage collection but offers more flexibility in size.

```
class DataProcessor {
private:
   int stackArray[100]; // Stack-allocated fixed-size array
   int* heapArray;      // Pointer for heap-allocated array
   int heapSize;

public:
DataProcessor(int size) {
heapSize = size;
heapArray = new int[size]; // Heap allocation
   }

   ~DataProcessor() {
      delete[] heapArray; // Manual cleanup required in C++
   }
};
```

In this C++ example, the class manages both a stack-allocated fixed array and a heap-allocated dynamic array, including proper cleanup in the destructor.

Reference vs. Value Semantics

Arrays within classes can follow either reference or value semantics, depending on the language:

- In languages like Java and Python, arrays are reference types, so copying an object with an array member typically creates a shallow copy.
- In languages like C++, arrays can follow value semantics if explicitly copied.

```
public class ArrayHolder {
```

```
    private int[] numbers;

    public ArrayHolder(int[] initialNumbers) {
this.numbers = initialNumbers; // Reference is copied, not the array contents
    }

    // Method to create a deep copy
    public ArrayHoldercreateDeepCopy() {
ArrayHolder copy = new ArrayHolder(new int[numbers.length]);
System.arraycopy(numbers, 0, copy.numbers, 0, numbers.length);
        return copy;
    }
}
```

This Java class demonstrates the reference semantics of arrays and provides a method for deep copying when needed.

Design Patterns Using Arrays in Classes

Iterator Pattern

The Iterator pattern allows sequential access to elements without exposing the underlying implementation. Classes containing arrays often implement iterators to provide safe traversal.

```
public class CustomCollection {
    private String[] elements;
    private int size;

    public CustomCollection(int capacity) {
        elements = new String[capacity];
        size = 0;
    }

    public void add(String element) {
        if (size <elements.length) {
            elements[size++] = element;
        }
    }

    public Iterator getIterator() {
        return new CollectionIterator();
```

```
    }

    // Inner iterator class
    private class CollectionIterator implements Iterator {
        private int currentIndex = 0;

        @Override
        public booleanhasNext() {
            return currentIndex< size;
        }

        @Override
        public String next() {
            if (!hasNext()) {
                return null;
            }
            return elements[currentIndex++];
        }
    }
}
```

This implementation allows clients to traverse the collection without direct access to the array.

Composite Pattern

Arrays within classes can facilitate the Composite pattern, which composes objects into tree structures to represent part-whole hierarchies.

```
public class Department {
    private String name;
    private Employee[] employees;
    private int employeeCount;
    private Department[] subDepartments;
    private int subDepartmentCount;

    public Department(String name, int maxEmployees, int
maxSubDepts) {
        this.name = name;
        employees = new Employee[maxEmployees];
subDepartments = new Department[maxSubDepts];
```

```
    }

    public void addEmployee(Employee emp) {
        if (employeeCount<employees.length) {
            employees[employeeCount++] = emp;
        }
    }

    public void addSubDepartment(Department dept) {
        if (subDepartmentCount<subDepartments.length) {
subDepartments[subDepartmentCount++] = dept;
        }
    }

    public int getTotalEmployeeCount() {
        int total = employeeCount;
        for (int i = 0; i<subDepartmentCount; i++) {
            total += subDepartments[i].getTotalEmployeeCount();
        }
        return total;
    }
}
```

This Department class uses arrays to manage both employees and sub-departments, creating a hierarchical structure.

Practical Applications

Data Structures Implementation

Arrays within classes form the foundation of many custom data structures, such as stacks, queues, and hash tables.

```
public class Stack<T> {
    private Object[] elements;
    private int top;
    private static final int DEFAULT_CAPACITY = 10;

    public Stack() {
        elements = new Object[DEFAULT_CAPACITY];
        top = -1;
    }
```

```
    public void push(T item) {
        if (top == elements.length - 1) {
            // Resize array if full
            Object[] newElements = new Object[elements.length * 2];
System.arraycopy(elements, 0, newElements, 0, elements.length);
            elements = newElements;
        }
        elements[++top] = item;
    }

    @SuppressWarnings("unchecked")
    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        T item = (T) elements[top];
        elements[top--] = null; // Help garbage collection
        return item;
    }

    public booleanisEmpty() {
        return top == -1;
    }
}
```

This generic stack implementation encapsulates an array, hiding the internal representation while providing a clean interface.

Buffering and Caching

Arrays in classes are ideal for implementing buffers and caches that temporarily store data.

```
public class CircularBuffer<T> {
    private Object[] buffer;
    private int readIndex;
    private int writeIndex;
    private int size;
    private int capacity;

    public CircularBuffer(int capacity) {
this.capacity = capacity;
```

```
        buffer = new Object[capacity];
readIndex = 0;
writeIndex = 0;
        size = 0;
    }

    public void write(T item) {
        if (size == capacity) {
            // Overwrite oldest item
readIndex = (readIndex + 1) % capacity;
            size--;
        }

        buffer[writeIndex] = item;
writeIndex = (writeIndex + 1) % capacity;
        size++;
    }

    @SuppressWarnings("unchecked")
    public T read() {
        if (size == 0) {
            return null;
        }

        T item = (T) buffer[readIndex];
readIndex = (readIndex + 1) % capacity;
        size--;
        return item;
    }

    public booleanisEmpty() {
        return size == 0;
    }

    public booleanisFull() {
        return size == capacity;
    }
}
```

This circular buffer class uses an array to efficiently store and retrieve data in a first-in-first-out manner, with automatic overwriting of the oldest data when full.

Game Development

Arrays within classes are essential in game development for representing game state, character attributes, and level designs.

```
class Battlefield:
    def __init__(self, width, height):
self.width = width
self.height = height
self.terrain = [[0 for _ in range(width)] for _ in range(height)]  # 2D array for terrain
self.Module s = [[None for _ in range(width)] for _ in range(height)]
# 2D array for Module s

    def place_terrain(self, terrain_type, x, y):
        if 0 <= x <self.width and 0 <= y <self.height:
self.terrain[y][x] = terrain_type

    def place_Module (self, Module , x, y):
        if 0 <= x <self.width and 0 <= y <self.height and self.Module
s[y][x] is None:
self.Module s[y][x] = Module
            return True
        return False

    def move_Module (self, from_x, from_y, to_x, to_y):
        if (0 <= from_x<self.width and 0 <= from_y<self.height and
            0 <= to_x<self.width and 0 <= to_y<self.height and
self.Module s[from_y][from_x] is not None and
self.Module s[to_y][to_x] is None):

self.Module s[to_y][to_x] = self.Module s[from_y][from_x]
self.Module s[from_y][from_x] = None
            return True
        return False
```

This Battlefield class uses two 2D arrays to represent the terrain and Module positions in a game, with methods to manipulate these elements.

Advanced Topics

Thread Safety

When using arrays in multi-threaded environments, synchronization becomes essential to prevent data corruption.

```
public class ThreadSafeBuffer {
   private final Object[] buffer;
   private int count = 0;

   public ThreadSafeBuffer(int size) {
      buffer = new Object[size];
   }

   public synchronized void add(Object item) throws
InterruptedException {
      while (count == buffer.length) {
         wait(); // Buffer full, wait for space
      }

      buffer[count++] = item;
notifyAll(); // Notify waiting threads
   }

   public synchronized Object remove() throws InterruptedException
{
      while (count == 0) {
         wait(); // Buffer empty, wait for items
      }

      Object item = buffer[0];
System.arraycopy(buffer, 1, buffer, 0, --count);
notifyAll(); // Notify waiting threads
      return item;
   }
}
```

This Java class implements a thread-safe buffer using synchronization to ensure data integrity in concurrent scenarios.

Serialization

Serializing classes with array members requires special attention to ensure the entire data structure can be properly saved and restored.

```
public class SerializableDataContainer implements Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private int[] values;
    private transient int[] cachedCalculations; // Not serialized

    public SerializableDataContainer(String name, int[] values) {
        this.name = name;
this.values = values.clone(); // Deep copy to ensure encapsulation
this.cachedCalculations = new int[values.length];
        recalculate();
    }

    private void recalculate() {
        for (int i = 0; i<values.length; i++) {
cachedCalculations[i] = values[i] * values[i]; // Example calculation
        }
    }

    // Called when object is deserialized
    private void readObject(ObjectInputStream in) throws
IOException, ClassNotFoundException {
in.defaultReadObject(); // Read the non-transient fields
cachedCalculations = new int[values.length];
        recalculate(); // Reconstruct the transient field
    }
}
```

This example demonstrates proper serialization of a class with array members, including handling of transient (non-serialized) calculated values.

Generic Arrays

Creating truly generic arrays in languages like Java presents special challenges due to type erasure.

```
public class GenericArrayWrapper<T> {
    private final Object[] array;
    private final Class<T> type;

    @SuppressWarnings("unchecked")
    public GenericArrayWrapper(Class<T> type, int size) {
this.type = type;
        // Cannot create generic arrays directly due to type erasure
        array = new Object[size];
    }

    public void set(int index, T item) {
        if (index >= 0 && index <array.length) {
            array[index] = item;
        } else {
            throw new IndexOutOfBoundsException();
        }
    }

    @SuppressWarnings("unchecked")
    public T get(int index) {
        if (index >= 0 && index <array.length) {
            return (T) array[index];
        } else {
            throw new IndexOutOfBoundsException();
        }
    }

    @SuppressWarnings("unchecked")
    public T[] toArray() {
        T[] result = (T[]) Array.newInstance(type, array.length);
        for (int i = 0; i<array.length; i++) {
            result[i] = (T) array[i];
        }
        return result;
```

```
    }
}
```

This class works around Java's limitations with generic arrays by using Object[] internally and providing type-safe access methods.

Language-Specific Implementations

Java

Java arrays are objects with a fixed length. When used within classes, they are often encapsulated with accessors and mutators.

```
public class TemperatureTracker {
    private final double[] hourlyTemperatures;
    private final String location;

    public TemperatureTracker(String location) {
this.location = location;
this.hourlyTemperatures = new double[24]; // 24 hours in a day

        // Initialize with default value
Arrays.fill(hourlyTemperatures, Double.NaN);
    }

    public void recordTemperature(int hour, double temperature) {
        if (hour >= 0 && hour < 24) {
hourlyTemperatures[hour] = temperature;
        }
    }

    public double getAverageTemperature() {
        int validReadings = 0;
        double sum = 0;

        for (double temp : hourlyTemperatures) {
            if (!Double.isNaN(temp)) {
                sum += temp;
validReadings++;
            }
        }

        return validReadings> 0 ? sum / validReadings : Double.NaN;
```

```
    }

    public double getMaxTemperature() {
        double max = Double.NEGATIVE_INFINITY;

        for (double temp : hourlyTemperatures) {
            if (!Double.isNaN(temp) && temp > max) {
                max = temp;
            }
        }

        return max != Double.NEGATIVE_INFINITY ? max :
Double.NaN;
    }
}
```

This Java class tracks hourly temperatures, providing methods to record readings and calculate statistics.

Python

Python's dynamic nature simplifies array handling within classes through lists.

```
class Playlist:
    def __init__(self, name):
        self.name = name
self.songs = []  # Dynamic list (Python's equivalent of a dynamic
array)
self.current_index = -1

    def add_song(self, song):
self.songs.append(song)

    def remove_song(self, song):
        if song in self.songs:
            index = self.songs.index(song)
self.songs.remove(song)
            # Adjust current index if necessary
            if index <= self.current_index and self.current_index> 0:
self.current_index -= 1
            return True
```

```
        return False

    def next_song(self):
        if not self.songs:
            return None

self.current_index = (self.current_index + 1) % len(self.songs)
        return self.songs[self.current_index]

    def previous_song(self):
        if not self.songs:
            return None

self.current_index = (self.current_index - 1) % len(self.songs)
        return self.songs[self.current_index]

    def current_song(self):
        if not self.songs or self.current_index == -1:
            return None
        return self.songs[self.current_index]

    def shuffle(self):
        import random
random.shuffle(self.songs)
self.current_index = -1  # Reset current index
```

This Python class implements a music playlist with various operations, using a list to store songs.

C++

C++ offers both static and dynamic arrays, with the additional complexity of manual memory management.

```
class ImageProcessor {
private:
    unsigned char* imageData;
    int width;
    int height;

public:
ImageProcessor(int width, int height) : width(width), height(height) {
```

```
        // Allocate memory for the image (assuming grayscale - one byte per pixel)
imageData = new unsigned char[width * height];
        // Initialize to black
        std::memset(imageData, 0, width * height);
    }

    // Copy constructor - essential for proper resource management
ImageProcessor(const ImageProcessor& other) : width(other.width), height(other.height) {
imageData = new unsigned char[width * height];
        std::memcpy(imageData, other.imageData, width * height);
    }

    // Move constructor - for efficient transfers of ownership
ImageProcessor(ImageProcessor&& other) noexcept : width(other.width), height(other.height), imageData(other.imageData) {
other.imageData = nullptr;
other.width = 0;
other.height = 0;
    }

    // Assignment operator
ImageProcessor& operator=(const ImageProcessor& other) {
        if (this != &other) {
            delete[] imageData;

            width = other.width;
            height = other.height;
imageData = new unsigned char[width * height];
            std::memcpy(imageData, other.imageData, width * height);
        }
        return *this;
    }

    // Destructor
    ~ImageProcessor() {
```

```
        delete[] imageData;
    }

    // Get pixel value
    unsigned char getPixel(int x, int y) const {
        if (x >= 0 && x < width && y >= 0 && y < height) {
            return imageData[y * width + x];
        }
        return 0;
    }

    // Set pixel value
    void setPixel(int x, int y, unsigned char value) {
        if (x >= 0 && x < width && y >= 0 && y < height) {
imageData[y * width + x] = value;
        }
    }

    // Apply blur filter
    void applyBlur() {
        // Create temporary buffer for result
        unsigned char* tempData = new unsigned char[width * height];

        // Simple box blur
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int sum = 0;
                int count = 0;

                // 3x3 kernel
                for (int dy = -1; dy<= 1; dy++) {
                    for (int dx = -1; dx <= 1; dx++) {
                        int nx = x + dx;
                        int ny = y + dy;

                        if (nx>= 0 &&nx< width &&ny>= 0 &&ny< height) {
                            sum += imageData[ny * width + nx];
                            count++;
```

```
                    }
                }
            }

tempData[y * width + x] = sum / count;
        }
    }

    // Swap buffers
    std::swap(imageData, tempData);
    delete[] tempData;
  }
};
```

This C++ class demonstrates advanced memory management for a class containing a dynamically allocated array representing an image, including proper copy semantics and a sample image processing algorithm.

Performance Optimization

Array Resizing Strategies

Efficiently resizing arrays is crucial for dynamic data structures.

```
public class DynamicArray<T> {
  private Object[] array;
  private int size;
  private static final int DEFAULT_CAPACITY = 10;

  public DynamicArray() {
    array = new Object[DEFAULT_CAPACITY];
    size = 0;
  }

  public void add(T element) {
ensureCapacity(size + 1);
    array[size++] = element;
  }

  @SuppressWarnings("unchecked")
  public T get(int index) {
    if (index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException();
        }
        return (T) array[index];
    }

    public int size() {
        return size;
    }

    private void ensureCapacity(int minCapacity) {
        if (minCapacity>array.length) {
            int newCapacity = Math.max(array.length * 2, minCapacity);
            Object[] newArray = new Object[newCapacity];
System.arraycopy(array, 0, newArray, 0, size);
            array = newArray;
        }
    }

    public void trimToSize() {
        if (size <array.length) {
            Object[] newArray = new Object[size];
System.arraycopy(array, 0, newArray, 0, size);
            array = newArray;
        }
    }
}
```

This implementation demonstrates efficient array resizing using exponential growth (doubling) to achieve amortized constant-time additions.

Memory Layout Optimization

Understanding how arrays are laid out in memory can lead to performance improvements through cache-friendly access patterns.

```
class MatrixOperations {
private:
    int rows;
    int cols;
    double* data; // Row-major order for better cache locality
```

```
public:
MatrixOperations(int r, int c) : rows(r), cols(c) {
    data = new double[rows * cols]();
  }

  ~MatrixOperations() {
    delete[] data;
  }

  // Row-major access
  double get(int row, int col) const {
    return data[row * cols + col];
  }

  void set(int row, int col, double value) {
    data[row * cols + col] = value;
  }

  // Cache-friendly matrix multiplication
MatrixOperations multiply(const MatrixOperations& other) const {
    if (cols != other.rows) {
      throw std::invalid_argument("Matrix dimensions do not match
for multiplication");
    }

MatrixOperations result(rows, other.cols);

    // Traditional multiplication (row by column)
    for (int i = 0; i< rows; i++) {
      for (int j = 0; j <other.cols; j++) {
        double sum = 0.0;
        for (int k = 0; k < cols; k++) {
          sum += get(i, k) * other.get(k, j);
        }
result.set(i, j, sum);
      }
    }
```

```
        return result;
    }

    // Cache-optimized matrix multiplication using blocking
MatrixOperationsmultiplyOptimized(const MatrixOperations& other)
const {
        if (cols != other.rows) {
            throw std::invalid_argument("Matrix dimensions do not match
for multiplication");
        }

MatrixOperations result(rows, other.cols);
        const int blockSize = 32; // Adjust based on cache size

        // Zero initialize result
        for (int i = 0; i< rows * other.cols; i++) {
result.data[i] = 0.0;
        }

        // Blocked multiplication for better cache utilization
        for (int ii = 0; ii < rows; ii += blockSize) {
            for (int jj = 0; jj<other.cols; jj += blockSize) {
                for (int kk = 0; kk < cols; kk += blockSize) {
                    // Process block
                    for (int i = ii; i< std::min(ii + blockSize, rows); i++) {
                        for (int j = jj; j < std::min(jj + blockSize, other.cols);
j++) {
                            double sum = result.get(i, j);
                            for (int k = kk; k < std::min(kk + blockSize, cols);
k++) {
                                sum += get(i, k) * other.get(k, j);
                            }
result.set(i, j, sum);
                        }
                    }
                }
            }
        }
```

```
        return result;
    }
};
```

This C++ class demonstrates cache-friendly matrix operations using row-major storage and blocked multiplication algorithms.

Testing and Debugging

Module Testing Arrays in Classes

Thorough testing of classes containing arrays requires verifying boundary conditions, empty states, and full capacity scenarios.

```
import org.jModule .jupiter.api.Test;
import static org.jModule .jupiter.api.Assertions.*;

class CircularBufferTest {
    @Test
    void testEmptyBuffer() {
CircularBuffer<String> buffer = new CircularBuffer<>(5);
assertTrue(buffer.isEmpty());
assertFalse(buffer.isFull());
assertNull(buffer.read());
    }

    @Test
    void testWriteAndRead() {
CircularBuffer<Integer> buffer = new CircularBuffer<>(3);

buffer.write(1);
buffer.write(2);

assertFalse(buffer.isEmpty());
assertFalse(buffer.isFull());

assertEquals(Integer.valueOf(1), buffer.read());
assertEquals(Integer.valueOf(2), buffer.read());
assertTrue(buffer.isEmpty());
    }

    @Test
```

```
   void testFullBuffer() {
CircularBuffer<Character> buffer = new CircularBuffer<>(2);

buffer.write('A');
buffer.write('B');

assertTrue(buffer.isFull());

      // When full, new writes overwrite oldest data
buffer.write('C');

assertEquals(Character.valueOf('B'), buffer.read());
assertEquals(Character.valueOf('C'), buffer.read());
   }

   @Test
   void testCyclicBehavior() {
CircularBuffer<Integer> buffer = new CircularBuffer<>(3);

      for (int i = 1; i<= 10; i++) {
buffer.write(i);

         // Read every third item to create circular pattern
         if (i % 3 == 0) {
            for (int j = 0; j < 3; j++) {
buffer.read();
            }
assertTrue(buffer.isEmpty());
         }
      }
   }
}
```

This JModule test class demonstrates comprehensive testing of a CircularBuffer implementation, covering various scenarios and edge cases.

Common Bugs and Pitfalls

When working with arrays in classes, several common issues arise:

1. Off-by-one errors

2. Null reference handling
3. Boundary checking
4. Out-of-bounds access
5. Memory leaks (in languages without garbage collection)

```
public class BuggyArrayHandler {
   private int[] data;

   // Bug 1: No null check in constructor
   public BuggyArrayHandler(int[] initialData) {
this.data = initialData; // Should check for null and make a defensive copy
   }

   // Bug 2: Off-by-one error in loop
   public int sum() {
      int total = 0;
      // Should be i<data.length
      for (int i = 0; i<= data.length; i++) {
         total += data[i];
      }
      return total;
   }

   // Bug 3: No bounds checking
   public void setValue(int index, int value) {
      data[index] = value; // Should check if index is within bounds
   }

   // Fixed implementation
   public static class FixedArrayHandler {
      private final int[] data;

      public FixedArrayHandler(int[] initialData) {
         // Null check and defensive copy
         if (initialData == null) {
            throw new IllegalArgumentException("Initial data cannot be null");
         }
```

```
this.data = initialData.clone();
    }

    public int sum() {
        int total = 0;
        // Correct loop bounds
        for (int i = 0; i<data.length; i++) {
            total += data[i];
        }
        return total;
    }

    public void setValue(int index, int value) {
        // Bounds checking
        if (index < 0 || index >= data.length
```

MCQs:

1. **Which of the following is NOT a feature of C++?**
   a) Object-Oriented Programming
   b) Platform Independence
   c) Low-Level Programming
   d) Garbage Collection
2. **Which of the following is NOT an Object-Oriented Programming concept?**
   a) Encapsulation
   b) Inheritance
   c) Polymorphism
   d) Compilation
3. **What is an Object in C++?**
   a) A function that stores data
   b) An instance of a class
   c) A type of variable
   d) A control statement
4. **Which of the following best describes a class?**
   a) A collection of functions
   b) A blueprint for creating objects
   c) A memory allocation technique
   d) A type of loop

5. **What does a member function in C++ do?**
   a) Stores objects in memory
   b) Allows functions to access class attributes
   c) Creates a new variable
   d) Converts one data type to another
6. **Which of the following is an advantage of OOP?**
   a) Reusability of code
   b) Increased compilation speed
   c) Reduced security
   d) High memory usage
7. **How is an object of a class created in C++?**
   a) class obj;
   b) object obj();
   c) className obj;
   d) object = new className;
8. **Which feature of OOP allows wrapping data and functions together?**
   a) Abstraction
   b) Encapsulation
   c) Polymorphism
   d) Inheritance
9. **An array within a class in C++ is used to:**
   a) Store multiple objects
   b) Store multiple values of the same type
   c) Convert data types
   d) Implement control statements
10. **Which feature of OOP allows objects to share characteristics?**
    a) Inheritance
    b) Encapsulation
    c) Polymorphism
    d) Data Hiding

Short Questions:

1. What are the key features of C++?
2. Define Object-Oriented Programming (OOP).
3. What are the advantages of OOP over procedural programming?
4. Explain the difference between classes and objects in C++.

5. What is the role of a member function in a class?
6. How is an object created and used in C++?
7. What is encapsulation, and why is it important in OOP?
8. Explain how arrays can be used within a class.
9. What is the difference between data abstraction and data hiding?
10. How does OOP improve code reusability?

Long Questions:

1. Explain the structure of a C++ program with an example.
2. What are the key Object-Oriented Programming (OOP) concepts? Explain each with an example.
3. Discuss the advantages of OOP and how it differs from procedural programming.
4. Define classes and objects in C++ and provide an example of how they are implemented.
5. What is encapsulation? Explain how it is achieved in C++ with an example.
6. How do member functions work inside a class? Provide a code example.
7. Explain the importance of arrays within a class with an example program.
8. How does OOP help in software development? Discuss real-world applications.
9. Discuss how data hiding and abstraction improve security in OOP.
10. Write a C++ program to demonstrate the use of objects, classes, and member functions.

MODULE 2
FUNCTIONS, CONSTRUCTORS, AND DESTRUCTORS

2.0 LEARNING OUTCOMES

- Understand the memory allocation of objects in C++.
- Learn about friend functions and how they interact with private data.
- Understand the concept of local classes in C++.
- Learn about constructors (Parameterized, Multiple, Default Argument) and their applications.
- Explore dynamic initialization of objects, copy constructors, and dynamic constructors.
- Understand the role of destructors in C++.

2.1 Memory Allocation of Objects

What is Object Memory Allocation?

CMPS 101: Memory Allocation Memory allocation is one of the most fundamental operations in modern computing systems. If program is to operate with the objects then how such objects are allocated, accessed and finally deallocated has great impact on performance, reliability, and utilization of resources of the program. This part of your learning journey discusses about how object memory allocation works under the hood, the different approaches anyone can take in different programming paradigms, and the problems of object lifetime management. So what it boils down to is that memory allocation for objects is allocating a section of a computer's memory to hold the state and structure information of an object. Though conceptually-simple, the actual implementation reaches into myriad complexities such as memory hierarchies, allocation algorithms, garbage collection mechanisms, and optimization techniques. These concepts give us significant knowledge to help software developers build efficient, reliable, and performant applications.

Computer Memory – a layman's explaination

Before we dive specifically into object allocation, it is important to explain the memory layout not only of object allocation but of modern computing systems in general. Computer architecture employs a memory hierarchy which efficiently allocates data across high-speed but low-capacity caches, slower, large main memory, and even more cost-efficient but much slower, larger forms of persistent storage. Even closer to the CPU are registers, ultra-fast, but very small storage areas that hold the most frequently accessed data and instructions at a high level. Instead, when they access any information, they use cache memory to hold recently accessed data, which results in a much shorter access time because its capacity is minimal. Cache memory is actually split into several levels, L1, L2, and L3, where L1 is the fastest but smaller, and L3 is slower but larger. The primary working area, where currently running programs and data reside, is called main memory (RAM). RAM is the term used to denote the much larger but significantly slower data storage compared to the cache.

Virtual memory is a concept for unifying RAM and persistent storage that has been explored a great deal, and there is a hardware-software combination of paging that allows the memory that would be allocated to the system to far exceed the physical memory limit of RAM. Secondary storage, in the form of SSDs and HDDs, is used for long-term data retention but works at a much lower speed. Types of behaviour are the interaction between elements of the memory hierarchy, the effectiveness of caching strategies, how virtual memory influences performance and so on. Virtual memory is an abstraction on lower-level storage, offering isolated address spaces to processes, protection, and the ability for programs to address more memory than the system actually holds. This abstraction depends on the Memory Management Module (MMU) and page tables for seamless conversion from virtual addresses to physical locations. The memory layout of a process divides memory into regions that lenses through a systematic and efficient Where the text segment holds code, the data segment stores initially defined globals and statics. This is the BSS segment (Block Started by Symbol), which holds all the uninitialized global and static variables, growing as needed. Another important memory area is heap memory, which enables dynamic memory allocation, which allows programs to allocate and release memory as needed at runtime. Unlike this, the stack handles function calls and local variables, working in a last-in, first-out (LIFO) fashion. Object memory models specify how objects are allocated and accessed in memory, which can affect the efficiency and speed of the program. There are two fundamental categories of data and they are value types and reference types. Value types hold their data directly so that when they are assigned, or passed to a function, they are copied rather than reference copied. On the other hand, reference types hold memory location pointers, which means that more than one variable can point to a single object. For example, in languages like Java, every object is reference type, while in languages like C# and Swift, there is an explicit notion of value types and reference types. This difference is significant for behaviors such as performance optimization, garbage collection, and object lifetime management in varying types of languages.

Object Layout in Memory

An object gets allocated in memory, which consists of various things as part of its structure, which helps manage, access, and use it in a particular runtime environment. Although not the most prominent feature, one of the most important aspects of an object is type information (metadata about the object's class/type). In languages that implement polymorphism, for example C++ or Java, this is usually stored as a pointer to a type descriptor, or virtual method table (vtable). In general, an object header stores synchronization/coding information(gcod data), garbage collection flags and at times hashed codes generation (for speed so they won't be computed in runtime, when two objects are compared or retrieved). You have instance fields which store the data members of the object and how they are organized on the memory is based on the language rules to make sure that we can work with the classes in an efficient manner. In addition, padding is also essential for aligning fields, as well as to keep track of memory layouts when it comes to CPUs. The precise memory layout of an instance can differ dramatically between languages and runtime environments. Like for example, C++ objects have vtable pointers for polymorphism, Java objects have pointers to class and synchronization information, C# objects have pointers to type objects and synchronization information blocks, and Python objects maintain reference counting and pointers to their types for memory management. Of course, those differences emphasize how approaches like this are tuned for performance vs. memory or function vs. runtime environment etc. An important aspect of object memory representation is **memory alignment and padding**, which significantly influences performance. CPUs are optimized to access memory in **aligned blocks**, meaning a 4-byte integer, for instance, should ideally be stored at a memory address **divisible by 4**. To ensure this alignment, compilers often introduce **padding bytes** between fields, increasing the object's overall memory footprint but improving access speed by reducing unaligned memory accesses. Consider the following C++ class:

```
class Example {
   char a;      // 1 byte
   int b;       // 4 bytes
   char c;      // 1 byte
   double d;    // 8 bytes
```

};

So initial total of field sizes is 14 bytes. However, due to alignment restrictions, the real object size may well be 24 bytes because of padding inserted in between fields. The char a is stored first, but to be able to align the int b, three padding bytes are put after char a. Likewise, after char c we may have 7 padding bytes inserted before double d to place it at an 8-byte boundary. This extra memory consumption optimizes CPU performance by making sure a part of the memory is accessed faster and preventing needless calculations where a structure's data are not aligned in memory. Though padding does come at a cost in memory consumption, with the acceptable trade-off for maximum-performance systems in modern computing architectures. For this reason, it is important for programmers to understand these concepts when writing the most efficient code possible, especially when working on performance-critical applications, including game engines, high-performance computing, and low-level systems programming.

Allocation Strategies

Efficiently allocating memory for objects is paramount to the efficiency of an application and by extension to its performance and memory consumption. Stack allocation is the most straightforward and fastest since it just manipulates the stack pointer, ensuring that allocations and deallocations take place in a strict LIFO manner. It frees allocation automatically when a function returns, is thus very efficient for local primitive variables, and small and short-lived objects and no fragmentation is the concern here. For example, stack allocation for value types in C and C++, and escape analysis for short-lived objects in Java and C#. But the most significant limitation of stack allocation lies in its rigidly-function scope dependence in contrast to the stack; a form of allocation that is not appropriate for objects whose lifespan must extend beyond their active function. Heap allocation, on the other hand, provides more flexibility as it allows objects to outlive the function that created them. It allows dynamic memory growth and allows allocations and deallocations in arbitrary order. Because it takes time to find the right chunk of unused memory, heap allocation is much slower than allocating memory on the stack. However, it also introduces problems like fragmentation, where free memory is broken into dispersed pieces not next to each

other, and the cost of extra structures needed to manage memory, tracking which blocks are used or free. We implement fixed-size allocators for commonly used object sizes to reduce heap fragmentation and improve speed. Most modern memory managers use a mixture of both for maximum efficiency. There are many heap allocation algorithms established to improve memory management. With the first-fit, the first free block that is greater than or equal to the requested memory size is selected versus the best-fit, the smallest free block that is equal to or greater than the requested size is selected ensuring that memory is allocated in the most efficient way to minimize fragmentation. The worst-fit strategy, on the other hand, cuts pieces of memory from the biggest available chunk, trying to leave large free portions for future allocations. The Buddy system divides memory in to blocks of power of 2 sizes which allows easier coalescing of blocks which become free, whereas slab allocation pre-allocates pools of objects of fixed size, making them especially useful for system objects which are allocated frequently. Many allocators have a hybrid approach and use different techniques depending on the size and patterns of allocations. Different programming languages have their own unique philosophies when it comes to memory management, and that affects how they implement object allocation strategies. C: Memory allocation is entirely manual, Allocates an object with malloc(), calloc() and releases it with free(). However, this gives you full control, but you also need to manage it properly otherwise you would encounter issues like memory leaks, use-after-free and double-free problems. Automatic garbage collection to reclaim and free up unused memory is used by languages like Java and Python, which does reduce the amount of memory-related bugs but at the expense of runtime overhead. This is where C++ comes in and helps with manual memory management and allows for the use of smart pointers to prevent resource leaks. That being said, memory allocation strategies continue to evolve, and most systems nowadays are hybrid between allocating small memory chunks and a big and suited memory chunk, balancing performance, memory efficiency, and the ease of programming across a wide variety of applications and computing environments.

Example:

```
struct Person {
```

```
    char* name;
    int age;
};

// Allocation
struct Person* person = (struct Person*)malloc(sizeof(struct Person));
person->name = (char*)malloc(50); // Allocate space for name
strcpy(person->name, "Alice");
person->age = 30;

// Later: deallocation
free(person->name);
free(person);
```

This approach offers maximum flexibility and performance but places a significant burden on developers to manage memory correctly.

C++ and RAII

C++ enhances C's model with several object-oriented features:

- Objects can be allocated on the stack or heap (new operator)
- Constructors and destructors enable the Resource Acquisition Is Initialization (RAII) pattern
- Smart pointers (unique_ptr, shared_ptr) automate memory management for heap objects
- Custom allocators allow specialized allocation strategies

Example:

```
class Person {
private:
    std::string name;
    int age;
public:
    Person(std::string n, int a) : name(n), age(a) {}
    ~Person() { /* Resources automatically cleaned up */ }
};

// Stack allocation
Person alice("Alice", 30);

// Heap allocation with smart pointer
auto bob = std::make_shared<Person>("Bob", 25);
```

```
// No explicit delete needed; memory freed when last reference disappears
```

This model combines manual control with safer abstractions, reducing common memory management errors.

Java and Garbage Collection

Java represents a fully managed approach:

- All objects are allocated on the heap using the new operator
- A garbage collector automatically reclaims memory when objects are no longer referenced
- No explicit deallocation mechanism is exposed to developers
- Memory layout and allocation details are abstracted away

Example:

```
class Person {
   String name;
   int age;

   Person(String name, int age) {
      this.name = name;
this.age = age;
   }
}

// Allocation
Person person = new Person("Alice", 30);

// No deallocation needed; garbage collector handles it
```

This approach simplifies development but sacrifices some control over memory management timing and behavior.

Python's Reference Counting and Garbage Collection

Python uses a hybrid mechanism for memory management where it uses both reference counting and cycle-detecting garbage collector to increase the usability of object deallocation. The main mechanism is reference counting, where every object keeps track of how many references to it exist; when the reference count reaches zero, the object is freed immediately. Reference counting alone is not enough to deal with circular references (two or more objects referencing each other), which keeps their reference counts from reaching zero. To

overcome this, Python's garbage collector has a cycle detection algorithm that is responsible for finding and deleting these unreachable objects. Furthermore, allocation in Python is separate from memory management meaning that developers have to do things like cleanup. One optimization is object pooling — commonly used small objects, like integers and strings, are reused for efficiency. Thus the balance of user-friendliness vs performance. For example, you have a Python code which allocates an instance of the Person class in memory and then sets it to None, this instance can be collected by a garbage collector later:

```
class Person:
   def __init__(self, name, age):
      self.name = name
self.age = age

# Allocation
person = Person("Alice", 30)

# Setting reference to None allows object to be collected
person = None
```

This automatic memory management system requires less effort from the developer, but ensures that the memory is freed as soon as objects are no longer required. However, it brings additional runtime overhead due to tracking references and detecting circular dependencies. In contrast, Rust has a completely different approach with an ownership-based memory model, which allows it to be garbage-collector-free while still providing compile-time memory-safety guarantees. Rust is a systems programming language in which every value has a single owner, and when that owner goes out of scope, the value is automatically cleaned up. This model avoids memory leaks and dangling pointers without incurring any runtime garbage collection. Another aspect Rust brings in is the concept of borrowing, which enables references to data without transferring ownership. Borrowing, on the other hand, has rigid rules that are checked at compile time to ensure that references never outlive the data they point to. These capabilities prevent many memory bugs, including use-after-free and double free. Here's an example that demonstrates Rust's ownership model:

```
struct Person {
   name: String,
   age: u32,
}

fn main() {
   // Allocation
   let person = Person {
      name: String::from("Alice"),
      age: 30,
   };

   // Automatic deallocation when `person` goes out of scope
}
```

In this example, the Person instance is created inside the main function and gets deallocated once main completes. This model of memory management gives the safety benefits of garbage collection without the runtime costs. Rust achieves this by enforcing these rules at compile time, meaning that safe and efficient memory usage is guaranteed by default, well ahead of runtime, making it a perfect choice for performance-critical applications like systems programming and embedded development.

Item Pools and Specialized Allotment Techniques

Along with default allocation algorithms, advanced allocation strategies (such as object pooling) can greatly improve performance if you have strict constraints on allocations per frame (for example creating and destroying many objects every frame). In this approach, instead of allocating or deallocating an object you reuse the existing object which helps in improving performance. Specifically, the technique consists of pre-allocating a pool of objects that can be reused; whenever an object of that type is required, it is taken from the pool rather than allocated. When the object is released, it doesn't get deallocated; it gets returned to the pool, reducing fragmentation and improving overall system performance. When appropriate, the pool size can grow dynamically as needed. In managed environments this way of using (not over allocating) memory could trigger a lot of

garbage collections and lead to performance issues. Moreover, the locality of object usage is improved since the frequently used objects remain close to one another in memory, which helps improve cache behavior and overall system responsiveness. In high-performance computing and real-time applications, the characteristics of object pooling make it an indispensable technology. For example, in situations where small objects are created and destroyed frequently within the same frame, such as in game development, network packet processing, or UI frameworks, object pooling is a good way to keep frame rates and system responsiveness at an acceptable level. The same applies to resource-capped objects such as database connections, threads, or file handlers. Object pooling avoids intermittent pauses due to memory allocation and garbage collection making memory allocation timing deterministic as needed by real-time systems which impose strict performance constraints. These applications are typical of high-throughput systems (e.g. financial systems, or high-frequency trading platforms), because of the lower allocation overhead, and therefore the uniform performance when running them with heavy workloads. Object pooling reduces memory fragmentation and allocation overhead, optimizing overall resource management and stability and efficiency of software systems.

Region-Based Allocation

Region-Based Allocation: Benefits and Advantages

This consists of allocating a large block of memory and then performing allocations that move a pointer in that memory. When objects are no longer needed, the whole region is freed at once, unlike more complex garbage collection, which needs to keep track of individual objects. It is particularly good for phases where lots of temporary objects are created and abandoned at the same time. This methodology is commonly used in both parsers and compile implementations, where ephemeral objects are created for each stage of processing. For example, request processing in web servers lends itself well to region based allocation, generating many temporary objects in the process where allocating and deallocating is cheap and memory management costs are minimized. This strategy improves performance and mitigates fragmentation that commonly occurs when applications frequently allocate and free memory and is widely

adopted in a variety of systems in environments running numerous memory allocation and deallocation cycles.

Optimizing Strategies: Escape Analysis and Custom Allocations

Thus in addition to allocation based on region, modern compilers and virtual machines do escape analysis to optimize memory usage. This method analyzes whether an object leaves the method that creates it. If an entity stays within the boundary of the method, it can be allocated on stack (instead of the heap) which greatly improves performance. Stack allocation is more direct because it bypasses heap overhead and collection. In addition, escape analysis allows compilers to remove redundant synchronization for non-escaping objects, this again contributes to the elimination of execution overhead in multi-threaded environment. Custom memory allocators can provide additional configurability and performance for specialized allocation patterns. One way of doing this is through thread-local allocators, which use a separate memory pool for each thread in order to avoid contention (in effect, this reduces synchronization overhead in multi-threaded applications). Different sizes of object are handled with different strategies in hierarchical allocators. In addition, the use of specialized domain allocators can optimize memory management for specific object types or access patterns, improving performance even more. These techniques, which can help developers enhance application performance while managing memory effectively, are the subject of great interest among developers.

Memory Fragmentation

Fragmentation is a common issue in dynamic memory allocation due to which it wastes a lot of memory and reduces the performance of computer system. The two common types of fragmentation are external fragmentation and internal fragmentation. You learn about external fragmentation, which happens when free memory is broken into several small block of memory that are not contiguous with each other and cannot meet an allocation request, even though the total free memory is large enough. This fragmentation can be attributed to the allocation and deallocation of small objects of different sizes, varying lifetimes of objects, and the lack of a compaction mechanism in the memory management system. There are gaps between the blocks allocated as memory is allocated and freed dynamically, making it difficult to find contiguous space for larger allocations. This

results in enormous memory wastage over a period of time which makes the system reject allocation requests even though it has enough free memory in fragmented form. In order to reduce external fragmentation different strategies are implemented. Compaction, for example, organizes the objects in memory to merge free space into larger contiguous blocks, but usually involves high computational overhead. Another strategy is coalescing, in which contiguous free blocks are merged to form larger chunks of usable memory. Buddy system is a commonly implemented memory management algorithm which divides and combines memory blocks to satisfy allocation requests. Other memory allocators implement a number of heaps per process, where each heap covers a range of size classes, in an effort to minimize fragmentation by allocating allocations of similar sizes. Opposite to external fragmentation, it is called internal fragmentation which happens when allocated memory blocks are larger than necessary, leaving nonutilized space in each block. This is often due to several factors, including alignment requirements, where memory addresses must be aligned on certain boundaries for hardware access to be efficient, resulting in wasted space in the blocks of memory that are allocated. In general, fixed-size allocation blocks also create internal fragmentation, when Memory block assigned is greater than the requested memory chunk. This is further complicated with memory management overhead and size class rounding within pool allocators, where your object would generally be rounded up to fit into a pool block of memory. Internal fragmentation is common in systems which favour speed and simple allocation strategies, but compromise memory utilization due to fixed-size granularity. Optimizations aim to customize memory allocation according to real requirements to reduce internal fragmentation. A strategy that is working especially well is to organize objects to minimize padding by maintaining efficient struct alignment. Pool allocators can also be implemented on a more fine-grained basis, where the requested size is always greater than or equal to the allocated size. Moreover, having different allocation strategies for different size ranges helps improve speed and memory conservation. For this reason, external and internal fragmentation can be reduvely reduced with ensuring more efficient allocation policies and better internal management techniques.

Related Items Garbage Collection and Automatic Memory Management

Many modern programming languages, including Java, Python, and C#, utilize garbage collection to automatically manage memory allocation, thus easing the burden on developers and mitigating memory leaks. Garbage collection is fundamentally concerned with determining which objects are still alive (reachable in the program) and which objects can be cleaned up (unreachable in the program). Some garbage collection techniques also include memory compaction to mitigate fragmentation, enhancing performance and efficiency. Garbage collection abstracts memory out of a programmer's responsibility, so they could focus on logic and functionality, and not manual allocation and deallocation. But such convenience has a price; the developers traded precise control of memory usage, leading to inefficiency, unpredictable pauses, and performance trade-offs for resource-intensive apps. Garbage collection can be tuned in such a way as to take some of these tradeoffs into account, and there are in fact different garbage collection strategies, each with its pros and cons. The simplest garbage collection algorithm is reference counting, in which each object stores a count of how many references are pointing to it. If the reference count reaches zero, the object is deallocated immediately. This gives predictable cleanup and minimal pause time for algorithm which is beneficial but has the problem of dealing with cycles as it needs additional way to manage that type of links. In contrast, tracing garbage collection identifies reachable memory by following from root objects. Some popular techniques under this umbrella are mark-sweep (mark the current/live objects then sweep through memory and delete unmarked ones), mark-compact (mark current/live objects, then compact memory to minimise fragmentation), and copying collection (live objects are copied to a new space and the old space is freed) Generational garbage collection takes this a step further by grouping objects based on how long they live, collecting the short-lived ones often while postponing collection of the long-lived ones. This allows for better performance by eliminating overhead. While garbage collection tuning is all about performance at scale it is important because pause times, cpu usage, memory footprint, and a lot of other metrics have a substantial influence on an application, responsiveness. Garbage

collection is often an automated process, but developers can adjust parameters like heap size, the frequency of collections, and more, to improve specific workloads while retaining as much automation as possible.

Memory Allocation for Special Object Types

Managing memory for various types of objects efficiently is a key aspect of modern computing. To avoid cache misses and improve access patterns, arrays and collections are stored continuously in memory. But their dynamic nature often requires you to over-allocate to meet future growth needs because it is costly to resize. On the other hand, when an array or collection exceeds its capacity, a new and bigger block of memory is allocated and all elements are copied to that new block, resulting in overhead in terms of performance. To mitigate this problem, many implementations apply a geometric growth strategy, for example doubling the overall size of the structure on an append (which amortizes the reallocation costs over many operations). There are added complications with larger objects like large buffers of data or media content. Since allocation of these objects within the normal heap may cause memory fragmentation, they are usually allocated within comparatively smaller regions of memory. For large objects, some runtime environments sidestep standard allocation mechanisms altogether, relying on memory-mapped files or specialized memory management strategies that are more compatible with virtual memory. Furthermore, the garbage collection for large objects is generally different from small objects, with separate collection thresholds (or less frequent collection) to minimize performance impacts. Small objects, on the other hand, present a whole new set of efficiency problems because of their small size. The allocation overhead like the space taken up for metadata can sometimes outstrip an object's real size, making inefficient use of your memory. Of course one has to deal with fragmentation since small objects are more frequent and their scattered allocations can hit performance in the long run. The above approach causes various problems, which many runtime environments use dedicated allocation techniques to solve. For example, object inlining puts small objects, which are frequently accessed, inside their parent objects, avoiding pointer dereferencing. They are typically combined with dedicated small object heaps or memory pools to reduce

fragmentation. The heaps in question are often size classes pre-allocated so all objects of the same size get grouped together for maximum memory efficiency. Bitmap-based allocation tracking additionally contributes to lower overall overhead of headers by representing allocation metadata in a packed format. These pointed optimizations ensure that the allocation of small and large objects is done efficiently so as to maintain a balance of higher performance with optimal memory usage.

The Best Practices for Memory Management in Multi-threaded Environments

Memory allocation presents a greater challenge to multi-threaded programs as they require concurrent access to the same resources. The first major challenge that arises is thread safety between the two memory allocators, where multiple threads may request, allocate, or deallocate memory simultaneously. There are several approaches to this issue. While global locks offer a straightforward approach through serialization of memory allocation operations, they create contention that decreases performance. Fine-grained locking increases concurrency, by locking smaller regions of memory, which allows multiple threads to allocate memory independently. Using atomic operations, lock-free algorithms remove the need for explicit locks altogether, minimizing contention even more. A different and less level two approach is thread-local allocation where each thread maintains its own pool of memory, what reduces lock contention. Finally it reduces contention on the global allocator and improves performance by having the memory requests from different threads dont foul up the memory cache with each others requests. However, misimplementing these strategies can result in fragmentation, memory wastage and potential race conditions that degrade the stability and efficiency of the program. What modern memory allocators do is per-thread caching, which prevents contention on the global allocator. Each thread has its private cache of free memory blocks, so the number of synchronization requirements for allocating to allocation events is very small, since you typically allocate what is already in the cache. When a thread requires memory, it first tries to allocate one from its local cache and it only contacts the global allocator when that is exhausted, saving for expensive calls in the global allocator. To prevent memory waste, if the cache grows too

big, excess memory blocks are returned back to the global pool. When the local cache becomes exhausted though, more memory is pulled from the global allocator. Although this strategy increases performance significantly, it comes with issues like false sharing, where multiple threads share the same cache line by accident, resulting in cache evictions and performance disturbances. Mitigations involve padding out objects to all cache line isolation, alignment aware allocation to place objects strategically, and thread aware placement strategies to keep hot objects apart. These techniques enable memory allocators to achieve high throughput and scalability in multi-threaded settings while reducing contention and cache-related inefficiencies.

Debugging and Profiling Memory Allocations

Understanding allocation patterns and potential issues that can degrade memory performance are parts of effective memory management. Memory leaks: Memory leaks is one of the most bad issues in memory management, which occurs when allocated memory is not released properly, and results in gradual loss of resources. There are various detection techniques to find such leaks, such as reference tracking tools, which identify objects without references, allocation tracking which helps detect unbalanced allocate/free operations, and heap differencing, which looks at the increase of memory over time. Statistical sampling further helps in identifying allocation hotspots, which allows for timely intervention. Another major difficulty is use-after-free and double-free bugs. A use-after-free error occurs when a program tries to read or write to a location in memory after the code has freed the memory, while a double-free error occurs when a program tries to free the same piece of memory multiple times, leading to possible undefined behavior. To lesson these hazards, developers use techniques such as delay-free mechanisms that place issued memory in quarantine for a period of time, memory poisoning to overwrite deallocated parts of memory with known patterns, and guard pages to catch access to element that was not intended by the programmer. Garbage collection and reference counting also play strong preventative roles, automatically freeing memory rather than risk deallocation too early. Allocation profiling not only detects errors, but also helps to optimize memory usage. Profiling can reveal where allocation hotspots are, where the

allocations are concentrated over time, what distributions of sizes are used, and how much and how long the objects allocated live. This perspective leads to memory optimization techniques like object pooling for commonly allocated objects, creating custom allocators for specific size classes, arranging your data in a manner that reduces allocation overhead, and reserving memory ahead of time based on anticipated workloads. But, even after 40 years, the trends in memory management are still relevant due to the latest hardware and software updates. Among them, NUMA (Non-Uniform Memory Access) awareness is a key improvement in that it is optimized for multi-socket architecture memory allocation, which means that the memory will be allocated on the same NUMA node as the thread that wants to access it, thereby reducing latency and improving performance. It leverages access patterns across nodes and provides APIs for explicit placement control as well as automatic migration of objects based on access behaviors. Emergence of persistent memory technologies like Intel Optane also introduce a new paradigm for memory allocations, allowing objects to live across power cycles, which require a different approach for durability, atomicity, and the recovery of incomplete operations. Further, hardware supported memory management is just starting to be adopted, exposing hardware transactional memory to achieve efficient concurrent allocations, memory tagging to catch errors, and page usage tracking to improve garbage collection. Hardware acceleration can also help reducing memory overhead in address translation. In addition, new domain-specific allocators are being created more in line with the specific needs of different computing domains. This opens up new avenues for real-time systems with predictable, bounded-time guarantees on allocations, and GPU computing with allocations handled as unified memory so as to allow an easy movement of data between CPU and GPU. Machine learning frameworks depend on specific tensor and gradient-optimized allocation schemes, while edge computing has memory-efficient allocation schemes for computing resources-constraint environments. With memory management evolving, these improvements work together to increase trustworthy system capabilities and guarantee that applications run precisely in various computing architectures.

Optimized strategies for distribution of objects

In addition to basic allocation strategies, you get several advanced optimization tricks that make memory usage and performance much better. One of these techniques is object inlining, where all of the child objects are directly embedded in their parents, eliminating the cost of pointer dereferencing, improving cache locality, and lowering the cost of allocation. So this is an effective way to deal with small objects that are not often shared because it makes memory management easier and involves much faster runtime. Another excellent method modern compilers use is escape analysis which checks whether an object can be accessed outside its scope. Stack allocation: Because an object can be allocated on the stack instead of the heap (which reduces the pressure on garbage collection), the object need not escape its method or thread. Escape analysis allows, among other things, scalar replacement to split objects into fields and thread-local objects to avoid unnecessary synchronization which boosts execution speed. As value types are allocated on the stack and immutable and favor time and space efficiency by not needing synchronization when being used on multi-threaded environments. Immutable objects are inherently thread-safe and have room for memory optimizations including structural deduplication and specialized memory layouts. Cache-aware object allocation optimizes for alignment with CPU caches to place frequently accessed fields at the start of objects, or to store related objects together in memory, to improve performance. Remember, prefetching and cache-line-aware data ordering are highly encouraged to reduce memory latency and increase throughput. Last but not least, dynamic adjustments in memory allocation strategies through serial optimizing compilers and runtime systems via profiling-based memory allocations, speculative optimizing, and Just-In-Time (JIT) compilation enable dynamic adaptation of matrix algorithms based on profile information. Dead field elimination improves memory footprint even more, since it trims unused object fields, making applications leaner and more performant.

2.2 Friend Function

Friend Function in C++: A Comprehensive Explanation

In object-oriented programming (OOP), **encapsulation** is a fundamental concept that ensures data security by restricting direct access to an object's internal state. However, there are situations

where we may need to allow an external function or class to access private and protected members of a class while still maintaining the integrity of the class structure. In C++, this is achieved using the **friend function**. A **friend function** is a special function that is not a member of a class but is granted permission to access the class's private and protected data members. It is declared inside the class with the keyword friend and defined outside the class. Unlike member functions, a friend function is not invoked using the object of the class; instead, it is called like a normal function.

Declaration and Definition of a Friend Function

A friend function is declared within the class using the friend keyword but defined outside the class without the scope resolution operator (::). Below is a basic example to illustrate its syntax:

Example: Using a Friend Function to Access Private Members

```
#include <iostream>
using namespace std;

class Sample {
private:
   int num;
public:
   Sample(int n) : num(n) {}
   friend void display(const Sample& s); // Friend function declaration
};

void display(const Sample& s) { // Friend function definition
cout<< "The value of num is: " <<s.num<<endl;
}

int main() {
   Sample obj(10);
   display(obj); // Calling the friend function
   return 0;
}
```

Key Features of a Friend Function

1. **Not a Member Function:** A friend function is not a member of the class but is declared inside the class.

2. **Defined Outside the Class:** Even though it is declared inside the class, it is defined externally.
3. **Access to Private and Protected Members:** The primary purpose of a friend function is to access private and protected members of a class.
4. **Called Like a Normal Function:** Unlike member functions, a friend function does not use the dot (.) or arrow (->) operator.
5. **Can Be Used for Multiple Classes:** A friend function can be used to access private members of multiple classes.

Table 2.1: Friend Function vs. Member Function

Feature	Friend Function	Member Function
Access Specifier	Needs friend keyword	No special keyword required
Access to Private Data	Yes	Yes (only for its own class)
Invocation	Called like a normal function	Called using an object of the class
Scope Resolution	Not required	Required for definition outside the class
Belongs to Class	No	Yes

Advantages of Friend Functions

1. **Facilitates External Function Access:** Sometimes, it is necessary to allow non-member functions to access private data.
2. **Useful in Operator Overloading:** Friend functions play a crucial role in overloading operators like +, -, and <<.
3. **Improves Code Modularity:** Certain operations can be kept separate from the class while still having access to private data.

Disadvantages of Friend Functions

1. **Breaks Encapsulation:** Since a friend function can access private data, it slightly weakens the concept of data hiding.
2. **Less Secure:** The use of friend functions increases the risk of accidental modification of private members.
3. **Harder to Maintain:** Excessive use of friend functions can make the code harder to manage and maintain.

Friend Function in Operator Overloading

One of the most common applications of friend functions is in operator overloading. Let's consider an example where we overload the + operator using a friend function.

Example: Overloading + Operator Using a Friend Function

```
#include <iostream>
using namespace std;

class Complex {
private:
   int real, imag;
public:
   Complex(int r, int i) : real(r), imag(i) {}
   friend Complex operator+(const Complex& c1, const Complex& c2);
   void display() {
cout<< real << " + " <<imag<< "i" <<endl;
   }
};

Complex operator+(const Complex& c1, const Complex& c2) {
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
   Complex c1(3, 4), c2(1, 2);
   Complex c3 = c1 + c2;
   c3.display();
   return 0;
}
```

Friend Class and Friend Function

A **friend class** is another concept related to friend functions. If multiple functions of a class need access to another class's private members, instead of making each function a friend, we can declare an entire class as a friend.

Example: Using a Friend Class

```
#include <iostream>
```

```
using namespace std;

class B; // Forward declaration

class A {
private:
   int numA;
public:
   A(int a) : numA(a) {}
   friend class B; // Declaring B as a friend class
};

class B {
public:
   void display(A obj) {
cout<< "Value of numA: " <<obj.numA<<endl;
   }
};

int main() {
   A objA(100);
   B objB;
objB.display(objA);
   return 0;
}
```

When to Use Friend Functions?

1. When external functions need access to private data without being part of the class.
2. When overloading certain operators that require access to private members.
3. When maintaining the encapsulation of a class while allowing specific functions to interact with private members.

2.3 Local Class

Introduction

Local classes are one of the several class types supported in Java and other object-oriented languages, and they have importance related to encapsulation and scoping. Local class — To define a class inside of a

block of a code like this block can be method, constructor or initialization block. In this document, we go in-depth on local classes, including what they are, their properties, their benefits, their limitations, and how to use local classes in practice, including some examples.

Definition and Characteristics

A **local class** is a nested class defined inside a method or block of code. It has the following characteristics:

1. **Limited Scope**: A local class is only accessible within the block where it is defined.
2. **Encapsulation**: Since it is defined within a method, it cannot be accessed outside that method, ensuring better encapsulation.
3. **Access to Enclosing Scope**: It can access variables and methods of the enclosing class, provided they are effectively final (i.e., they do not change after initialization).
4. **No Static Members**: Local classes cannot have static members, except for constant declarations (static final variables).
5. **Can Implement Interfaces and Extend Other Classes**: Like other classes, local classes can extend other classes and implement interfaces.

Syntax of Local Class

A local class is defined inside a method or block as follows:

```
class OuterClass {
   void display() {
      class LocalClass {
         void showMessage() {
System.out.println("This is a local class.");
         }
      }
LocalClass obj = new LocalClass();
obj.showMessage();
   }
}
```

In the above example, LocalClass is defined inside the display() method of OuterClass. It cannot be accessed outside this method.

Advantages of Local Classes

1. **Encapsulation**: Since they are defined within a method, they are not accessible from outside, reducing unwanted interference.
2. **Better Organization**: Local classes help organize code by keeping the class definition close to where it is used.
3. **Increased Readability**: They improve readability by keeping the scope of the class limited to its use case.
4. **Efficient Memory Utilization**: Local classes are only created when the method is invoked, ensuring efficient memory utilization.

Limitations of Local Classes

1. **Limited Accessibility**: They cannot be accessed outside their enclosing method, which can be restrictive in some scenarios.
2. **Cannot Have Static Members**: They do not support static variables or methods, except for constants.
3. **Complexity**: Overuse of local classes can lead to code that is harder to maintain and debug.

Use Cases of Local Classes

Local classes are particularly useful in scenarios where a small helper class is required within a method. Some common use cases include:

1. **Event Handling**: In GUI-based applications, local classes are used to handle events.
2. **Encapsulation of Logic**: When a specific logic is required only within a method, local classes provide a neat encapsulation.
3. **Threading**: Local classes can be used to create Runnable objects for multi-threading.

Example 1: Using Local Class for Event Handling

```
import java.awt.*;
import java.awt.event.*;

class ButtonDemo {
   public void createGUI() {
      Frame frame = new Frame("Local Class Example");
      Button button = new Button("Click Me");

button.addActionListener(new ActionListener() {
         class ButtonClickHandler implements ActionListener {
```

```
            public void actionPerformed(ActionEvent e) {
System.out.println("Button Clicked!");
            }
        }

        public void actionPerformed(ActionEvent e) {
            new ButtonClickHandler().actionPerformed(e);
        }
    });

frame.add(button);
frame.setSize(300, 200);
frame.setLayout(new FlowLayout());
frame.setVisible(true);
  }
  public static void main(String[] args) {
    new ButtonDemo().createGUI();
  }
}
```

In this example, a local class ButtonClickHandler is used to handle the button click event, encapsulating the logic within the event handler.

Example 2: Local Class in Multi-threading

```
class ThreadDemo {
  void startThread() {
    class MyThread implements Runnable {
      public void run() {
System.out.println("Thread is running...");
      }
    }
    Thread t = new Thread(new MyThread());
t.start();
  }
  public static void main(String[] args) {
ThreadDemo demo = new ThreadDemo();
demo.startThread();
  }
}
```

Here, MyThread is a local class used to create a Runnable object for threading.

Table 2.2: Comparison with Anonymous and Inner Classes

Feature	**Local Class**	**Anonymous Class**	**Inner Class**
Defined Inside	Method/Block	Expression	Class
Can Have a Name	Yes	No	Yes
Can Implement Interface	Yes	Yes	Yes
Can Extend Class	Yes	Yes	Yes
Access Enclosing Scope	Yes (Effectively Final)	Yes (Effectively Final)	Yes
Static Members	No	No	Yes (If Static Inner Class)

Best Practices for Using Local Classes

1. **Use When Necessary**: Local classes should be used when a small, specific functionality is required within a method.
2. **Avoid Overuse**: Overusing local classes can make code less readable and harder to debug.
3. **Prefer Anonymous Classes for Simplicity**: If the class is used only once, consider using an anonymous class instead.
4. **Ensure Encapsulation**: Use local classes to encapsulate logic specific to a method.

2.4 Constructors: Parameterized, Multiple, Default Argument

A constructor is a member function we use in an object oriented programming which invokes automatically when an object of a class is created. It is mainly utilized for object initialization and allocating resources. Constructors differ from regular functions as they share the name of the class that they belong to and they never return any value. They are responsible for establishing initial state, ensuring that an object is in a valid state upon creation. We can classify constructors on the basis of their implementation and method of accepting parameters. Constructors with default arguments In order to effectively write code with these types it's important to have a thorough understanding of each since they make it possible to initialize objects in a more structured and flexible way.

Parameterized Constructors

A parameterized constructor is a constructor with an argument with which an object can be initialized to specific values based on the inputted argument at the object creation time. By default constructor does not receive parameters and provide brokers, but a parameterized constructor directly helps.

Syntax of Parameterized Constructor:

```
class Student {
   string name;
   int age;

public:
   // Parameterized constructor
   Student(string n, int a) {
      name = n;
      age = a;
   }
   void display() {
cout<< "Name: " << name << ", Age: " << age <<endl;
   }
};

int main() {
```

```
    Student s1("John", 20); // Object creation with parameterized constructor
    s1.display();
    return 0;
}
```

Advantages of Parameterized Constructors:

1. **Customization of Object Initialization** – Users can define object properties at the time of creation rather than assigning values later.
2. **Eliminates the Need for Setter Methods** – Since values are initialized in the constructor itself, additional setter functions may not be required.
3. **Ensures Object Integrity** – It ensures that every object created has meaningful values and is not left in an uninitialized state.

Multiple Constructors (Constructor Overloading)

A class may have more than one constructor with different parameters. It's called constructor overloading. Constructors are overloaded by the number and type of what is passed in. This makes object creation more efficient with multiple constructors to initialize objects in a variety of ways.

Example of Constructor Overloading:

```
class Rectangle {
   int length, width;

public:
   // Default constructor
   Rectangle() {
      length = 0;
      width = 0;
   }
   // Parameterized constructor
   Rectangle(int l, int w) {
      length = l;
      width = w;
   }
   // Copy constructor
   Rectangle(const Rectangle &r) {
      length = r.length;
      width = r.width;
   }
   void display() {
cout<< "Length: " << length << ", Width: " << width <<endl;
   }
};

int main() {
   Rectangle r1;         // Default constructor called
   Rectangle r2(10, 20); // Parameterized constructor called
   Rectangle r3(r2);     // Copy constructor called

   r1.display();
   r2.display();
```

```
    r3.display();
    return 0;
}
```

Benefits of Constructor Overloading:

1. **Flexibility in Object Creation** – Different ways to instantiate objects based on available data.
2. **Improved Code Readability** – Different constructors make code intuitive and easy to understand.
3. **Enhanced Code Maintainability** – Overloaded constructors reduce the need for separate initialization methods.

Default Arguments in Constructors

A constructor can have **default arguments**, meaning some parameters can take predefined values if no explicit values are provided during object instantiation. This feature reduces redundancy and provides a convenient way to initialize objects with common values.

Example of Constructor with Default Arguments:

```
class Car {
    string model;
    int year;

public:
    // Constructor with default arguments
    Car(string m = "Toyota", int y = 2022) {
        model = m;
        year = y;
    }
    void display() {
cout<< "Model: " << model << ", Year: " << year <<endl;
    }
};

int main() {
    Car c1;          // Uses default values
    Car c2("Honda");  // Uses default year, custom model
    Car c3("BMW", 2020); // Uses custom values

    c1.display();
    c2.display();
```

```
    c3.display();
    return 0;
}
```

Advantages of Default Arguments in Constructors:

1. **Less Code Duplication** – Avoids creating multiple overloaded constructors for default values.
2. **Increased Usability** – Allows users to provide only necessary values while leaving others as defaults.
3. **Better Code Maintainability** – If default values need modification, they only have to be changed in one place.

Table 2.3: Comparison of Different Constructor Types

Feature	**Default Constructor**	**Parameterized Constructor**	**Multiple Constructors**	**Default Arguments**
Arguments	None	Yes	Varies	Some parameters have default values
Purpose	Initializes object with generic values	Initializes object with user-defined values	Provides multiple ways to create objects	Provides optional values for parameters
Example	Rectangle() {}	Rectangle(int l, int w)	Rectangle(), Rectangle(int), Rectangle(int, int)	Rectangle(int l = 10, int w = 5)
Flexibility	Low	Medium	High	High

2.5 Dynamic Initialization of Objects, Copy Constructor and Dynamic Constructor

Introduction

Beginning of Subject 2 (OOP): OOP is one of the most important patterns in modern software development, and a key concept in OOP is object initialization. Initialization is the part of program execution that describes how the objects will be created and managed. Object, Copy constructor and Dynamic constructor are some of the important and different types of initialization techniques that you are taught while learning C++ or any other OOP based programming code. These techniques allow for the effective management of memory resources, the duplication of objects, and the dynamic allocation of specific resources. We cover these aspects in-depth in this Module , with a focus on their importance, implementation, benefits, and actual use cases.

_ Do you have an idea for a story we should cover?

Initialization of Dynamic Objects

Concept and Need

In a dynamic initialization, an object is initialized at runtime using user-provided values or values calculated during execution of the program. This approach is not as inflexible as static initialization, which requires values to be known during compile time. Dynamic initialization is advantageous to use when your program requires dynamic memory allocation and typically takes user inputs, reads values from a file, or calculates a value before assigning it to an object.

Implementation in C++

In C++, dynamic initialization is often performed using constructors that accept arguments. It utilizes memory allocation functions such as new and delete to manage resources efficiently.

```
#include <iostream>
using namespace std;

class Product {
   string name;
```

```
    float price;
public:
    // Parameterized constructor with dynamic initialization
    Product(string pname, float pprice) {
        name = pname;
        price = pprice;
    }

    void display() {
cout<< "Product: " << name << ", Price: " << price <<endl;
    }
};

int main() {
    string pname;
    float pprice;

cout<< "Enter product name: ";
cin>>pname;
cout<< "Enter product price: ";
cin>>pprice;

    Product p(pname, pprice); // Dynamic initialization at runtime
p.display();
    return 0;
}
```

Advantages of Dynamic Initialization

1. **Flexibility** – Enables initialization based on user input or runtime conditions.
2. **Efficient Memory Usage** – Allocates resources only when necessary, avoiding unnecessary memory consumption.
3. **Scalability** – Supports complex data structures and dynamic resource allocation.
4. **Encapsulation and Data Integrity** – Keeps data members private and ensures controlled initialization.

Copy Constructor

Definition and Purpose

A **copy constructor** is a special constructor in C++ that initializes a new object as a copy of an existing object. It is used to duplicate objects while preserving their state. The copy constructor is particularly important in cases where objects contain dynamically allocated memory or when passing objects by value.

Syntax and Implementation

A copy constructor takes a reference to an object of the same class as its parameter.

```
class ClassName {
public:
ClassName(const ClassName&obj) {
      // Copy constructor implementation
   }
};
```

Example of Copy Constructor

```
#include <iostream>
using namespace std;

class Student {
   string name;
   int age;
public:
   // Parameterized constructor
   Student(string sname, int sage) {
      name = sname;
      age = sage;
   }

   // Copy constructor
   Student(const Student &obj) {
      name = obj.name;
      age = obj.age;
   }

   void display() {
cout<< "Name: " << name << ", Age: " << age <<endl;
   }
```

```
};

int main() {
   Student s1("Alice", 20);
   Student s2 = s1; // Invokes copy constructor

   s1.display();
   s2.display();
   return 0;
}
```

Advantages of Copy Constructor

1. **Ensures Deep Copy** – Essential for objects containing dynamically allocated memory.
2. **Efficient Object Duplication** – Allows copying objects without manually reassigning values.
3. **Simplifies Code Maintenance** – Reduces redundancy and enhances readability.
4. **Prevents Unintended Modifications** – Protects original data while working with copies.

Dynamic Constructor

Definition and Functionality

A **dynamic constructor** is a constructor that dynamically allocates memory to objects using new or malloc() during object creation. Unlike traditional constructors, which allocate memory statically, dynamic constructors allow objects to acquire memory space at runtime based on program requirements.

Implementation in C++

```
#include <iostream>
using namespace std;

class DynamicArray {
   int *arr;
   int size;
public:
   // Dynamic constructor
DynamicArray(int n) {
      size = n;
```

```
arr = new int[size]; // Allocating memory dynamically
   }

   void setValues() {
      for (int i = 0; i< size; i++) {
cout<< "Enter value for index " <<i<< ": ";
cin>>arr[i];
      }
   }

   void display() {
      for (int i = 0; i< size; i++) {
cout<<arr[i] << " ";
      }
cout<<endl;
   }

   ~DynamicArray() {
      delete[] arr; // Freeing memory
   }
};

int main() {
   int n;
cout<< "Enter the number of elements: ";
cin>> n;

DynamicArray d(n); // Dynamic constructor called
d.setValues();
d.display();
   return 0;
}
```

Advantages of Dynamic Constructor

1. **Efficient Memory Management** – Allocates memory as needed, preventing wastage.
2. **Scalability** – Suitable for handling large datasets and variable-size structures.

3. **Prevention of Memory Fragmentation** – Optimizes memory allocation and deallocation.
4. **Supports Real-time Applications** – Beneficial in cases where data size varies dynamically.

Unit 6 : Destructures

2.6 Destructors

Destructors in C++

In terms of memory management process in object-oriented programming (OOP), constructors and destructors are implicitly the two most vital structures. You work on constructors until you realize that the destructors are there to make sure things are clean when the cheese closes off. In this article, we will learn about C++ destructor, a key special member function that is called automatically on object destruction. This helps to ensure that any dynamically allocated resources, such as memory, file handles, or network connections, are properly released, preventing memory leaks and resource mismanagement.

Introduction to Destructors

A destructor in C++ is a member function with the same name as the class, prefixed with a tilde (~). Unlike constructors, destructors take no arguments and do not return any value. Every class in C++ can have at most one destructor.

Syntax of a Destructor

The syntax of a destructor in C++ is as follows:

```
class ClassName {
public:
   ~ClassName() {
      // Destructor body
   }
};
```

Characteristics of a Destructor

- It is automatically invoked when an object goes out of scope.
- It has the same name as the class but is prefixed with a tilde (~).
- It cannot be overloaded, meaning a class can have only one destructor.
- It does not take any parameters and does not return a value.
- It is commonly used to release dynamically allocated memory or close file handles.

Need for Destructors

In C++, dynamic memory allocation is done using the new operator, and it must be deallocated using delete. If objects allocate memory dynamically and fail to release it before being destroyed, memory leaks occur, leading to inefficient memory utilization. Destructors help in handling such situations by ensuring proper resource deallocation.

For instance, consider the following example:

```
#include <iostream>
using namespace std;

class Example {
public:
   Example() {
cout<< "Constructor called" <<endl;
   }
   ~Example() {
cout<< "Destructor called" <<endl;
   }
};

int main() {
   Example obj; // Constructor will be called
   return 0;    // Destructor will be called automatically
}
```

Output:

Constructor called
Destructor called

Destructor and Dynamic Memory Management

To illustrate the role of destructors in managing dynamic memory, consider a class that dynamically allocates memory inside the constructor:

```
#include <iostream>
using namespace std;

class DynamicExample {
private:
```

```
   int* ptr;
public:
DynamicExample() {
ptr = new int; // Allocating memory
cout<< "Memory allocated." <<endl;
   }
   ~DynamicExample() {
      delete ptr; // Releasing memory
cout<< "Memory deallocated." <<endl;
   }
};

int main() {
DynamicExample obj;
   return 0;
}
```

Output:

Memory allocated.

Memory deallocated.

Without the destructor, the dynamically allocated memory would not be released, causing a memory leak.

Destructors in Inheritance

In inheritance, it is required to call the destructor of base class and derived classes properly. In C++, the destructors of the inheritance hierarchy are called in a top down manner starting from the derived class to the base class.

Example of Destructors in Inheritance

```
#include <iostream>
using namespace std;

class Base {
public:
   Base() {
cout<< "Base Constructor" <<endl;
   }
   virtual ~Base() {
cout<< "Base Destructor" <<endl;
   }
```

```
};

class Derived : public Base {
public:
   Derived() {
cout<< "Derived Constructor" <<endl;
   }
   ~Derived() {
cout<< "Derived Destructor" <<endl;
   }
};

int main() {
   Base* obj = new Derived();
   delete obj; // Ensures proper destruction
   return 0;
}
```

Output:

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

Using a virtual destructor ensures that the destructor of the derived class is called before the base class destructor, preventing memory leaks when dealing with polymorphism.

Virtual Destructors

In C++, if a class contains virtual functions, it is recommended to declare its destructor as virtual. A virtual destructor ensures that when deleting an object through a base class pointer, the destructor of the derived class gets executed first, followed by the base class destructor.

Example of Virtual Destructors

```
class Parent {
public:
   Parent() {
cout<< "Parent Constructor" <<endl;
   }
   virtual ~Parent() {
cout<< "Parent Destructor" <<endl;
```

```
    }
};

class Child : public Parent {
public:
    Child() {
cout<< "Child Constructor" <<endl;
    }
    ~Child() {
cout<< "Child Destructor" <<endl;
    }
};

int main() {
    Parent* obj = new Child();
    delete obj;
    return 0;
}
```

If the destructor is not virtual, only the base class destructor will be called, leading to resource leaks in the derived class.

Explicitly Calling Destructors

Though destructors are called automatically when an object goes out of scope, they can be explicitly called using the scope resolution operator:

obj.~ClassName();

However, explicitly calling destructors is usually unnecessary and should be done cautiously.

Destructors and Smart Pointers

Modern C++ uses smart pointers (std::unique_ptr, std::shared_ptr) to manage memory automatically. These smart pointers have destructors that automatically release resources when they go out of scope.

Example using std::unique_ptr:

```
#include <iostream>
#include <memory>
using namespace std;

class Sample {
public:
```

```
    Sample() {
cout<< "Resource allocated." <<endl;
    }
    ~Sample() {
cout<< "Resource deallocated." <<endl;
    }
};

int main() {
unique_ptr<Sample>ptr = make_unique<Sample>();
    return 0;
}
```

Output:

Resource allocated.

Resource deallocated.

Using smart pointers eliminates the need for explicit destructors in many cases, making memory management safer and more efficient.

MCQs:

1. **Which memory is used for object storage in C++?**
   a) Stack
   b) Heap
   c) RAM
   d) ROM
2. **Which of the following statements is true about friend functions?**
   a) Friend functions can access private and protected members of a class
   b) Friend functions can only access public members
   c) Friend functions belong to the class
   d) Friend functions require object instantiation
3. **What is a local class in C++?**
   a) A class defined inside a function
   b) A global class
   c) A class that can only be used in files
   d) A static class
4. **What is a constructor in C++?**
   a) A function that is used to allocate memory
   b) A function that initializes objects

c) A function that destroys objects

d) A normal function inside a class

5. **Which constructor is called when an object is created without parameters?**

   a) Copy Constructor

   b) Dynamic Constructor

   c) Default Constructor

   d) Parameterized Constructor

6. **Which type of constructor allows an object to be initialized using another object?**

   a) Default Constructor

   b) Copy Constructor

   c) Multiple Constructor

   d) Dynamic Constructor

7. **Which constructor dynamically allocates memory at runtime?**

   a) Copy Constructor

   b) Dynamic Constructor

   c) Default Constructor

   d) Multiple Constructor

8. **What is the purpose of a destructor in C++?**

   a) To allocate memory

   b) To initialize objects

   c) To release memory and clean up resources

   d) To call functions

9. **How many destructors can a class have?**

   a) One

   b) Two

   c) Multiple

   d) None

10. **Which operator is used for dynamic memory allocation in C++?**

    a) malloc

    b) free

    c) new

    d) delete

Short Questions:

1. What is memory allocation of objects in C++?

2. Explain the concept of a friend function with an example.
3. What is a local class in C++?
4. Define constructors and explain their purpose.
5. What are the different types of constructors?
6. Explain the concept of copy constructors.
7. What is the difference between dynamic constructor and default constructor?
8. What is the significance of destructors in C++?
9. How does dynamic initialization of objects work?
10. Explain the syntax and purpose of a destructor in C++.

Long Questions:

1. Explain memory allocation of objects in C++ with examples.
2. What is a friend function? Discuss its advantages and limitations with an example.
3. Define local classes and explain their applications.
4. What are constructors? Discuss parameterized, multiple, and default argument constructors with examples.
5. Explain the process of dynamic initialization of objects in C++.
6. What is a copy constructor? Write a program to demonstrate its use.
7. Compare and contrast dynamic constructors and normal constructors.
8. What is a destructor, and how does it work in C++? Explain with an example.
9. Discuss the importance of object initialization and destruction in memory management.
10. Write a C++ program demonstrating different types of constructors and destructors.

MODULE 3
OPERATOR OVERLOADING AND INHERITANCE

3.0 LEARNING OUTCOMES

- Understand operator overloading and how to overload unary and binary operators.
- Learn how to overload binary operators using friend functions.
- Understand the rules of operator overloading and type conversion.
- Learn about inheritance and how derived classes function in C++.
- Explore different types of inheritance (Single, Multilevel, Multiple).
- Understand virtual base classes and abstract classes.
- Learn about constructors in derived classes and member classes.

Unit 7: Operator Overloading Basics

3.1 Operator Overloading: Unary and Binary

Introduction

One of the most influential features of object orientated programming (OOP) is operator overloading, it enables the programmer to specify or alter the actions of the built-in operators for user-defined types. This makes the code easier to read and write, allowing objects to act like primitive data types and use the same operators. Operator overloading in C++ is extensively used to make operations for classes like complex numbers, matrices and vectors intuitive. Operators can be unary or binary, which refers to the number of operands that they work on.

Introduction to Operator Overloading

Operator overloading allows us to redefine an operator so that we can perform a particular operation on user-defined data types. For fundamental data types, when an operator is overloaded, it has its normal meaning, while for class objects, it takes on new meaning. They are special functions used to overload called operator functions, they can either be class member functions or friend functions.

In C++, operators can be categorized into:

- **Unary Operators**: Operate on a single operand (e.g., ++, --, !, -, ~).
- **Binary Operators**: Require two operands (e.g., +, -, *, /, ==, !=, >, <).

The syntax for operator overloading follows this general format:

```
class ClassName {
public:
ReturnTypeoperatorSymbol(Arguments) {
      // Overloaded operator function body
   }
};
```

Unary Operator Overloading

Unary operators operate on a single operand. Some of the commonly overloaded unary operators include increment (++), decrement (--), negation (-), logical NOT (!), and bitwise complement (~).

Example 1: Overloading Unary - Operator

```
#include <iostream>
using namespace std;

class Number {
   int value;
public:
   Number(int v) : value(v) {}
   void display() { cout<< "Value: " << value <<endl; }
   Number operator-() {
      return Number(-value);
   }
};

int main() {
   Number n1(10);
   Number n2 = -n1;
   n1.display();
   n2.display();
   return 0;
}
```

Explanation:

- The - operator is overloaded using a member function.
- The overloaded operator- negates the value and returns a new object.
- The main function demonstrates the application of the overloaded operator.

Example 2: Overloading Increment (++) Operator

```
#include <iostream>
using namespace std;

class Counter {
   int count;
public:
   Counter() : count(0) {}
   void display() { cout<< "Count: " << count <<endl; }
   Counter operator++() { // Pre-increment
      ++count;
      return *this;
```

```
    }
};

int main() {
    Counter c1;
    ++c1;
    c1.display();
    return 0;
}
```

Explanation:

- The ++ operator is overloaded to increment count.
- The function returns the modified object.

Binary Operator Overloading

Binary operators require two operands and can be overloaded to define operations like addition, subtraction, multiplication, and comparison for user-defined types.

Example 3: Overloading + Operator

```
#include <iostream>
using namespace std;

class Complex {
    int real, imag;
public:
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}
    void display() { cout<< real << " + " <<imag<< "i" <<endl; }
    Complex operator+(const Complex &c) {
        return Complex(real + c.real, imag + c.imag);
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;
    c3.display();
    return 0;
}
```

Explanation:

- The + operator is overloaded to add two complex numbers.

- A new Complex object is returned, encapsulating the sum.

Example 4: Overloading == Operator

```
#include <iostream>
using namespace std;

class Point {
   int x, y;
public:
   Point(int a, int b) : x(a), y(b) {}
   bool operator==(const Point &p) {
      return (x == p.x&& y == p.y);
   }
};

int main() {
   Point p1(3, 4), p2(3, 4);
   if (p1 == p2)
cout<< "Points are equal" <<endl;
   else
cout<< "Points are not equal" <<endl;
   return 0;
}
```

Explanation:

- The == operator is overloaded to compare two Point objects.
- It returns true if both x and y coordinates match.

Overloading Operators as Friend Functions

Operators can also be overloaded using friend functions. This is useful when the left-hand operand is not an object of the class.

Example 5: Overloading * Operator using Friend Function

```
#include <iostream>
using namespace std;

class Multiply {
   int value;
public:
   Multiply(int v) : value(v) {}
   friend Multiply operator*(const Multiply &m1, const Multiply &m2);
```

```
    void display() { cout<< "Value: " << value <<endl; }
};

Multiply operator*(const Multiply &m1, const Multiply &m2) {
    return Multiply(m1.value * m2.value);
}

int main() {
    Multiply m1(5), m2(3);
    Multiply m3 = m1 * m2;
    m3.display();
    return 0;
}
```

Explanation:

- The * operator is overloaded as a friend function.
- It allows multiplication of objects without requiring a member function.

3.2 Overloading Binary Operators Using Friend Functions

Overloading Binary Operators Using Friend Functions in C++

Introduction

Operator overloading is a crucial feature in C++ that enables operators to work with user-defined data types. Specifically, binary operator overloading allows us to define custom behavior for operations such as addition (+), subtraction (-), multiplication (*), and division (/) when applied to objects of a class. One way to achieve this is by using **friend functions**. This article explores how to overload binary operators using friend functions in C++ with examples, applications, and best practices.

Understanding Binary Operator Overloading

A binary operator operates on two operands. In C++, built-in binary operators such as +, -, *, /, ==, and != can be overloaded to work with class objects. When overloading binary operators, we must define how they function when applied to objects of user-defined classes.

There are two primary ways to overload binary operators in C++:

1. Using **member functions**
2. Using **friend functions**

Friend Functions for Binary Operator Overloading

A **friend function** is a non-member function that has access to the private and protected members of a class. It is particularly useful for binary operator overloading when the left operand is not necessarily an object of the class.

Syntax of a Friend Function for Overloading a Binary Operator

```
class ClassName {
   private:
      // Data members
   public:
      // Constructor
      // Friend function prototype
      friend Return Type operator Op Symbol (const ClassName & obj1, const ClassName&obj2);
};
```

Steps to Overload a Binary Operator Using Friend Functions

1. **Define a class** with necessary data members.
2. **Declare a friend function** inside the class.
3. **Define the friend function** outside the class to perform the desired operation.
4. **Return the result** as an object of the class.
5. **Test the overloaded operator** in the main () function.

Example: Overloading the + Operator Using a Friend Function

Let's consider a Complex number class where we overload the + operator using a friend function.

```
#include <iostream>
using namespace std;

class Complex {
private:
   int real, imag;

public:
   Complex (int r = 0, int i = 0) : real(r), imag(i) {}

   // Friend function declaration
   friend Complex operator+(const Complex &c1, const Complex &c2);
```

```
   void display() {
cout<< real << " + " <<imag<< "i" <<endl;
   }
};

// Friend function definition
Complex operator+(const Complex &c1, const Complex &c2) {
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
   Complex c1(3, 4), c2(5, 6);
   Complex c3 = c1 + c2; // Overloaded + operator
cout<< "Sum: ";
   c3.display();
   return 0;
}
```

Explanation

- The Complex class has private data members real and imag.
- A **friend function**operator+ is declared inside the class.
- The function definition is written outside the class, performing addition on real and imag parts of the two objects.
- The main() function demonstrates how c1 + c2 works seamlessly due to operator overloading.

Example: Overloading the - Operator Using a Friend Function

Similarly, we can overload the - operator for the Complex class.

```
friend Complex operator-(const Complex &c1, const Complex &c2);
Complex operator-(const Complex &c1, const Complex &c2) {
   return Complex(c1.real - c2.real, c1.imag - c2.imag);
}
```

Example: Overloading the * Operator Using a Friend Function

To overload the multiplication operator for matrix multiplication:

```
class Matrix {
private:
   int data;

public:
   Matrix(int val = 0) : data(val) {}
   friend Matrix operator*(const Matrix &m1, const Matrix &m2);
   void display() {
cout<< "Value: " << data <<endl;
   }
};

Matrix operator*(const Matrix &m1, const Matrix &m2) {
   return Matrix(m1.data * m2.data);
}

int main() {
   Matrix m1(4), m2(3);
   Matrix m3 = m1 * m2;
cout<< "Product: ";
   m3.display();
   return 0;
}
```

Applications of Overloaded Operators

- **Mathematical computations** (e.g., complex numbers, matrices, vectors)
- **Custom string operations** (e.g., concatenation)
- **Smart pointers and iterators**
- **Graphics and game development** (e.g., vector arithmetic)

- **Scientific computing** (e.g., statistical calculations, financial models)

Advantages of Using Friend Functions

1. **Allows flexibility** when the left operand is not a class object.
2. **Directly accesses private members** without needing accessors.
3. **Enhances readability** by keeping operator logic separate from the class definition.

Disadvantages of Using Friend Functions

1. **Breaks encapsulation** since private members are accessible.
2. **Cannot use this pointer** as it is a non-member function.
3. **May lead to performance overhead** if not optimized properly.

Best Practices for Overloading Operators Using Friend Functions

1. **Use friend functions only when necessary**, such as when the first operand isn't a class object.
2. **Return objects by value**, unless performance requires otherwise.
3. **Ensure operators maintain expected mathematical behavior** to avoid confusion.
4. **Minimize access to private data** to maintain encapsulation where possible.
5. **Keep operator functions simple and efficient** to avoid unnecessary computational overhead.

3.3 Rules of Overloading Operators, Type Conversion

Introduction

Operator overloading is a powerful feature in object-oriented programming (OOP) that allows operators to be redefined for user-defined data types. This feature enhances code readability and expressiveness, making it possible to perform intuitive operations on objects. However, operator overloading must adhere to specific rules to maintain program consistency and prevent ambiguity. Similarly, type conversion enables seamless interactions between different data types, ensuring type safety and proper interpretation of values.

Rules of Overloading Operators

Operator overloading follows well-defined constraints imposed by programming languages like C++ to avoid unexpected behavior. The following rules must be considered when overloading operators:

1. **Only Existing Operators Can Be Overloaded** Operator overloading does not allow the creation of new operators. It only modifies the behavior of existing operators for user-defined types.
2. **Certain Operators Cannot Be Overloaded** Some operators are fundamental to language syntax and cannot be overloaded, such as:
    - Scope resolution operator (::)
    - Member access operators (. and .*)
    - Ternary conditional operator (?:)
    - sizeof and typeid
3. **Operator Overloading Must Involve a User-Defined Type** At least one operand in an overloaded operator function must be a user-defined type (such as a class or structure). This prevents overloading operators for built-in types, avoiding conflicts.
4. **Preserving Operator Precedence and Associativity** Overloading does not change an operator's precedence or associativity. The compiler applies the same precedence rules as for built-in types.
5. **Overloading Must Preserve Natural Semantics** Operators should be overloaded in a way that maintains logical consistency. For example, an overloaded + operator should perform an addition-like operation, not something unrelated.
6. **Overloaded Operators Can Be Member or Non-Member Functions** Operators can be implemented as:
    - **Member functions**, where the left operand must be an instance of the class.
    - **Non-member functions**, often implemented using friend functions to allow direct access to private data.
7. **Some Operators Must Be Overloaded as Member Functions** Certain operators, such as =, (), [], and ->, must be overloaded as member functions since they inherently belong to a specific object.
8. **Binary Operators Need Two Parameters When Non-Member Functions** A binary operator, when overloaded as a non-member function, takes two explicit parameters.

However, if defined as a member function, it implicitly takes the invoking object as the first operand.

9. **Unary Operators Take No Parameters When Member Functions** When overloaded as member functions, unary operators (like -, !, ++, --) do not take explicit parameters. If implemented as non-member functions, they take one parameter.
10. **Friend Functions for Private Member Access** To allow operator overloading functions to access private class members, they are often declared as friend functions within the class.
11. **Operators Must Return the Correct Type** Overloaded operators should return an appropriate type. For example, operator+ typically returns a new object rather than modifying the existing one.
12. **Assignment (=) Operator Requires Proper Memory Management** Overloading the assignment operator (=) must handle deep copies for objects containing dynamically allocated memory to prevent memory leaks and shallow copy issues.
13. **Overloaded Increment and Decrement Operators Must Differentiate Prefix and Postfix**
    - Prefix (++x, --x) takes no arguments.
    - Postfix (x++, x--) takes an int argument to differentiate it from the prefix version.
14. **Logical and Bitwise Operators Should Return Meaningful Results** Logical operators (&&, ||, !) should return Boolean values, whereas bitwise operators should return modified versions of the object.
15. **Stream Insertion (<<) and Extraction (>>) Operators Must Be Non-Member Functions** Since cout<< obj requires cout (an ostream object) on the left side, these operators should be overloaded as friend or non-member functions.

Type Conversion

Type conversion is essential for ensuring compatibility between different data types. It can be categorized into three types:

1. **Implicit Type Conversion (Type Promotion)**
    - Automatically performed by the compiler.

- Converts smaller data types to larger ones (e.g., int to float).
- Avoids data loss and type mismatches.

2. **Explicit Type Conversion (Type Casting)**
    - Performed using cast operators (static_cast, dynamic_cast, reinterpret_cast, const_cast in C++).
    - Can lead to data loss if improperly used.
    - Used when implicit conversion is not sufficient.
3. **User-Defined Type Conversion**
    - Allows custom conversion between user-defined types.
    - Implemented using:
        - **Conversion Constructor** (Single-argument constructor that converts other types into the class type.)
        - **Conversion Operator (operator type())** (Defines conversion from class type to another type.)

3.4 Inheritance and Derived Classes

Inheritance is one of the key concepts of object-oriented programming (OOP) that allows a class to inherit properties and behaviour from any

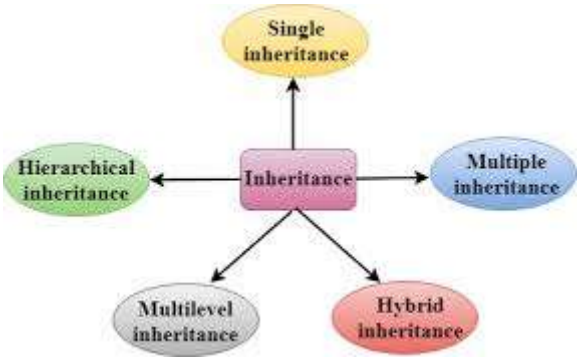


Figure 3 Types of Inheritance
[Source: https://medium.com]

other class. This mechanism fosters code reuse, creates a hierarchical between classes. Inheritance is a concept in OOP where a class, known as derived or child class, inherits the properties and behavior (methods) of another class known as parent base class, essentially establishing relationships between classes, and is much useful as it allows the creation of specialized implementations while preserving a common interface.

Inheritance Explained: The Fundamentals

Inheritance, at its core, signifies an "is-a" relationship (for the base or parent class), it inherits everything from the parent class, and can add few features or modify features if required.

The syntax for creating a derived class in C++ is:

```
class DerivedClass : [access-specifier] BaseClass {
   // Additional members and methods
};
```

The access specifier (public, private, or protected) determines how the members of the base class are inherited by the derived class.

Access Specifiers in Inheritance

The access specifier used during inheritance affects how the members of the base class are accessible in the derived class:

1. **Public Inheritance**: The public members of the base class become public members of the derived class, and protected members remain protected. This is the most common form of inheritance as it preserves the interface of the base class.
2. **Protected Inheritance**: The public and protected members of the base class become protected members of the derived class.
3. **Private Inheritance**: Both public and protected members of the base class become private members of the derived class.

For example:

```
class Base {
public:
   int publicVar;
protected:
   int protectedVar;
private:
   int privateVar;
};

class DerivedPublic : public Base {
   // publicVar remains public
   // protectedVar remains protected
   // privateVar is not accessible
};

class DerivedProtected : protected Base {
   // publicVar becomes protected
   // protectedVar remains protected
   // privateVar is not accessible
};

class DerivedPrivate : private Base {
   // publicVar becomes private
   // protectedVar becomes private
   // privateVar is not accessible
};
```

Member Access in Derived Classes

Just like C++ in which when we derive a class from a base class, the derived class would have access to all the public and protected

members of the base class. Although private members can not be accessed directly, they are accessible indirectly through public or protected member functions of the base class.

```
class Shape {
protected:
   double width;
   double height;
public:
   void setDimensions(double w, double h) {
      width = w;
      height = h;
   }
};

class Rectangle : public Shape {
public:
   double area() {
      return width * height;  // Access to protected members
   }
};
```

Function Overriding

Derived classes are allowed to implement a function that had been previously defined in the base class. This is called function overriding:

```
class Animal {
public:
   void makeSound() {
      std::cout<< "Some generic sound" << std::endl;
   }
};

class Dog : public Animal {
public:
   void makeSound() {  // Overrides the base class method
      std::cout<< "Woof!" << std::endl;
   }
};
```

To call the base class version of an overridden function, you can use the scope resolution operator (::):

```
class Dog : public Animal {
public:
   void makeSound() {
      Animal::makeSound();  // Call base class version
      std::cout<< "Woof!" << std::endl;
   }
};
```

Polymorphism Through Inheritance

Polymorphism — thanks to inheritance, a pointer/reference to a base class can point to an object of a derived class. This is an incredibly powerful feature that paves the way for flexible and extensible code:

```
Animal* pet = new Dog();
pet->makeSound();      // Calls Dog::makeSound() if properly implemented with virtual
```

In order for polymorphism to function correctly, the functions of the base class must be declared virtual, which we will examine in detail in section 3.6.

3.5 Inheritance: Single, Multilevel, Multiple

Different styles of inheritance structures provided for different design needs is supported by Object-oriented programming. Single inheritance, multilevel inheritance, and multiple inheritance are the major types.

Single Inheritance

Single inheritance is the simplest form where a derived class inherits from only one base class. This creates a direct parent-child relationship:

```
class Animal {
   // Base class members
};

class Dog : public Animal {
   // Single inheritance
};
```

Single inheritance is straightforward to implement and understand. It's supported by virtually all object-oriented programming languages, including Java, C#, and C++.

Multilevel Inheritance

Multilevel inheritance involves a derived class that inherits from another derived class, creating a chain of inheritance:

```
class Animal {
   // Base class members
};

class Mammal : public Animal {
   // First level derived class
};

class Dog : public Mammal {
   // Second level derived class
};
```

This structure allows each level to add specialized features while inheriting all the properties of its ancestors. The Dog class in the example has access to features from both Animal and Mammal classes.

Multiple Inheritances

Multiple inheritances occurs when a derived class inherits from two or more base classes:

```
class Engine {
   // First base class
public:
   void start() {
      std::cout<< "Engine started" << std::endl;
   }
};

class Wheels {
   // Second base class
public:
   void rotate() {
      std::cout<< "Wheels rotating" << std::endl;
   }
};

class Car : public Engine, public Wheels {
```

```
    // Derives from two base classes
public:
    void drive() {
        start();  // From Engine
        rotate(); // From Wheels
        std::cout<< "Car is moving" << std::endl;
    }
};
```

In the next section, we will talk about the complexities that arise from multiple inheritance, such as the diamond problem.

Hierarchical Inheritance

Another common pattern is hierarchical inheritance, in which multiple derived classes inherit from a single base class:

```
class Animal {
    // Base class
};

class Dog : public Animal {
    // First derived class
};

class Cat : public Animal {
    // Second derived class
};

class Bird : public Animal {
    // Third derived class
};
```

This practice is effective when common behavior is defined in one base class, and inherited by several specialized classes.

Hybrid Inheritance

Hybrid inheritance is obtained by combining two or more types of inheritance types. A class can even use both multiple and multilevel inheritance, for example:

```
class A { /* ... */ };
class B { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };
```

In this example, class D uses multiple inheritance (inheriting from B and C) and is also part of a multilevel inheritance chain (A to C to D).

3.6 Virtual Base Classes and Abstract Classes

The Diamond Problem

Multiple inheritance can lead to ambiguity known as the "diamond problem." This occurs when a class inherits from two classes that both inherit from a common base class:

```
  A
 / \
B   C
 \ /
  D
```

In this structure, class D inherits from both B and C, which both inherit from A. This means D inherits A's members twice, creating ambiguity.

```
class A {
public:
   int value;
};

class B : public A { };
class C : public A { };

class D : public B, public C {
   // Problem: D has two copies of A's members
};

int main() {
   D d;
d.value = 10;  // Ambiguous: which 'value' - from B or from C?
```

```
}
```

Virtual Base Classes

C++ resolves the diamond problem using virtual inheritance. By declaring base classes as virtual, a class ensures only one instance of the common ancestor:

```
class A {
public:
   int value;
};

class B : virtual public A { };
class C : virtual public A { };

class D : public B, public C {
   // D now has only one copy of A's members
};

int main() {
   D d;
d.value = 10;  // No ambiguity: only one 'value' exists
}
```

When you Multiple Inheritance, it is possible that there are 2 copies of the base class, so Virtual Base classes helps us to avoid this, so in Virtual Base classes there is only one base class copy no matter how many multiple inheritance paths there are.

Abstract Classes

An abstract class is a class that cannot be instantiated, but can be inherited from. It usually has one or more pure virtual function, which is a declared but not implemented type of function in the base class:

```
class Shape {
public:
   // Pure virtual function (makes Shape abstract)
   virtual double area() = 0;
```

```
    virtual void draw() {
        std::cout<< "Drawing a shape" << std::endl;
    }
};
```

The = 0 syntax marks a function as pure virtual, meaning derived classes must implement it. Any class with at least one pure virtual function becomes an abstract class.

```
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function
    double area() override {
        return 3.14159 * radius * radius;
    }
};
```

Interface vs. Implementation Inheritance

Abstract classes emphasise the difference between interface inheritance and implementation inheritance:

- **Interface Inheritance:** The derived classes get the interface (the methods that can be called), but they have to implement those themselves. This interface is defined by pure virtual functions.

So the first implementation inheritance is implementation inheritance: derived classes inherit both the interface and the implementation. Regular (non-pure) virtual and non-virtual functions provide implementation inheritance. Abstract classes are often a mix of both, providing some implementation (common utilities) while requiring implementations for others.

Virtual Functions and Runtime Polymorphism

Virtual functions enable runtime polymorphism, allowing a base class pointer to correctly call a derived class's implementation:

```
Shape* shape = new Circle(5.0);
double a = shape->area();  // Calls Circle::area(), not Shape::area()
```

This behavior is crucial for building flexible and extensible software systems.

The override and final Specifiers

Modern C++ provides additional keywords to better manage virtual functions:

- **Override**: Indicates that a function is intended to override a virtual function from a base class. The compiler checks that this is indeed the case:

```
class Derived : public Base {
public:
   void foo() override;  // Compiler checks that Base has a virtual foo()
};
```

- **final:** Prevents further overriding of a virtual function or inheritance from a class:

```
class Base {
public:
   virtual void foo() final;  // Cannot be overridden further
};

class FinalClass final { };  // Cannot be used as a base class
```

Unit 9: Constructors in Derived Classes and Member Classes

3.7 Constructors in Derived Classes and Member Classes

Constructor Execution Order

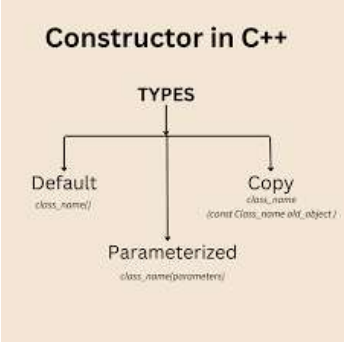


Figure 4 Types of Constructor
[Source https://logicmojo.com]

When a derived class object is created, the constructors execute in a specific order:

1. Base class constructors (in order of declaration for multiple inheritance)
2. Member object constructors (in order of declaration)
3. Derived class constructor body

Likewise, destructors execute in the reverse order.

Initializing Base Classes

A derived class constructor can explicitly call a base class constructor in its initializer list:

```
class Base {
private:
   int value;
public:
   Base(int v) : value(v) {}
};

class Derived : public Base {
private:
   double data;
public:
   // Calls Base(5) and initializes data to 10.5
```

```
    Derived() : Base(5), data(10.5) {}

    // Calls Base(v) and initializes data to d
    Derived(int v, double d) : Base(v), data(d) {}
};
```

When an instance of K is created and the base constructor is not explicitly called, the default (parameterless) base constructor is invoked automatically.

Virtual Base Class Constructors

When deriving a class from a virtual base class, construction of the virtual base is done by the most-derived class, regardless of how deep the inheritance chain goes:

```
class A {
public:
    A(int x) {}
};

class B : virtual public A {
public:
    B() : A(10) {}  // Calls A's constructor
};

class C : virtual public A {
public:
    C() : A(20) {}  // Calls A's constructor
};

class D : public B, public C {
public:
    // D must initialize A, even though B and C also do
    D() : A(30), B(), C() {}  // The A(30) call takes precedence
};
```

In this example, despite B and C initializing A, D's initialization of A takes precedence because D is the most derived class.

Member Classes (Nested Classes)

C++ allows classes to be defined within other classes, creating nested or member classes:

```
class Outer {
private:
   int value;

public:
   class Inner {
   private:
      int data;
   public:
      Inner(int d) : data(d) {}

      void display() {
         // Inner can't directly access Outer::value
         std::cout<< "Inner data: " << data << std::endl;
      }
   };

   Inner createInner(int d) {
      return Inner(d);
   }
};

int main() {
   Outer outer;
   Outer::Inner inner = outer.createInner(42);
inner.display();
}
```

Key characteristics of member classes:

1. A nested class is a member of its enclosing class and has the same access rights as other members.
2. The nested class can be declared in private, protected, or public sections of the enclosing class, affecting its visibility.

3. The nested class does not have direct access to the enclosing class's members without an instance of the enclosing class.
4. The relationship is purely logical nesting; there's no automatic containment relationship.

Initialization of Member Objects

When a class has member objects (composition), these members are initialized in the constructor's initializer list:

```
class Engine {
public:
   Engine(int power) {}
};

class Wheel {
public:
   Wheel(int size) {}
};

class Car {
private:
   Engine engine;
   Wheel wheels[4];

public:
   Car() : engine(150), wheels{18, 18, 18, 18} {}
};
```

If member objects aren't explicitly initialized, their default constructors are called.

Delegating Constructors

Modern C++ allows a constructor to delegate to another constructor in the same class:

```
class Example {
private:
   int x, y, z;

public:
   // Primary constructor
   Example(int x, int y, int z) : x(x), y(y), z(z) {}
```

```
    // Delegates to the primary constructor
    Example() : Example(0, 0, 0) {}

    // Delegates with some default values
    Example(int x) : Example(x, 0, 0) {}
};
```

This reduces code duplication and centralizes initialization logic.

Practical Applications of Inheritance

Class Hierarchies in GUI Frameworks

Graphical user interface (GUI) frameworks extensively use inheritance to create component hierarchies. Consider a simplified example:

```
class Widget {
protected:
    int x, y, width, height;

public:
    Widget(int x, int y, int w, int h) : x(x), y(y), width(w), height(h) {}
    virtual void draw() = 0;
    virtual bool handleEvent(const Event& event) = 0;
};

class Button : public Widget {
private:
    std::string label;
    std::function<void()>onClick;

public:
    Button(int x, int y, int w, int h, const std::string& label)
        : Widget(x, y, w, h), label(label) {}

    void draw() override {
        // Draw button with label
    }

    bool handleEvent(const Event& event) override {
        // Handle click events
```

```
        if (event.type == Event::Click &&containsPoint(event.x,
event.y)) {
            if (onClick) onClick();
            return true;
        }
        return false;
    }

    void setOnClickHandler(std::function<void()> handler) {
onClick = handler;
    }
};

class Checkbox : public Widget {
private:
    bool checked;
    std::string label;

public:
    Checkbox(int x, int y, int w, int h, const std::string& label)
        : Widget(x, y, w, h), label(label), checked(false) {}

    void draw() override {
        // Draw checkbox with label and check status
    }

    bool handleEvent(const Event& event) override {
        // Toggle checked state on click
        if (event.type == Event::Click &&containsPoint(event.x,
event.y)) {
            checked = !checked;
            return true;
        }
        return false;
    }

    bool isChecked() const { return checked; }
};
```

Object-Oriented Database Design

Inheritance is crucial in database object models, allowing for specialized entities while maintaining common traits:

```
class DatabaseEntity {
protected:
    int id;
    std::string createdAt;
    std::string updatedAt;

public:
    virtual ~DatabaseEntity() = default;
    virtual void save() = 0;
    virtual void load(int id) = 0;
    virtual void remove() = 0;
};

class User : public DatabaseEntity {
private:
    std::string username;
    std::string email;
    std::string passwordHash;

public:
    void save() override {
        // Implementation for saving user data
    }

    void load(int id) override {
        // Implementation for loading user data
    }

    void remove() override {
        // Implementation for removing user data
    }

    // User-specific methods
    bool authenticate(const std::string& password) {
        // Authentication logic
```

```
        return true;
    }
};

class Product : public DatabaseEntity {
private:
    std::string name;
    double price;
    int stockQuantity;

public:
    void save() override {
        // Implementation for saving product data
    }

    void load(int id) override {
        // Implementation for loading product data
    }

    void remove() override {
        // Implementation for removing product data
    }

    // Product-specific methods
    bool isInStock() const {
        return stockQuantity> 0;
    }
};
```

Game Development Entity System

Game engines often use inheritance for entity systems:

```
class GameObject {
protected:
    Vector2D position;
    bool active;

public:
GameObject() : active(true) {}
    virtual ~GameObject() = default;
```

```
    virtual void update(float deltaTime) = 0;
    virtual void render() = 0;

    void setPosition(const Vector2D& pos) { position = pos; }
    Vector2D getPosition() const { return position; }

    void setActive(bool isActive) { active = isActive; }
    bool isActive() const { return active; }
};

class Player : public GameObject {
private:
    int health;
    float speed;

public:
    Player() : health(100), speed(5.0f) {}

    void update(float deltaTime) override {
        // Update player state, handle input, etc.
    }

    void render() override {
        // Render player sprite
    }

    void takeDamage(int amount) {
        health -= amount;
        if (health <= 0) {
setActive(false);  // Player is defeated
        }
    }
};

class Enemy : public GameObject {
private:
    int health;
```

```
    float speed;
    Player* target;

public:
    Enemy(Player* player) : health(50), speed(3.0f), target(player) {}

    void update(float deltaTime) override {
        if (target && target->isActive()) {
            // Move toward player
            Vector2D direction = target->getPosition() - position;
direction.normalize();
            position = position + direction * speed * deltaTime;
        }
    }

    void render() override {
        // Render enemy sprite
    }

    void takeDamage(int amount) {
        health -= amount;
        if (health <= 0) {
setActive(false);  // Enemy is defeated
        }
    }
};
```

Best Practices for Inheritance

Use Inheritance Judiciously

Inheritance creates tight coupling between classes which can make code a little less flexible. Use composition or interfaces instead if applicable:

- Use inheritance to model “is-a” relationships
- Composition falls more favorably in “has-a” relationships
- Use interface inheritance instead of implementation inheritance if you can

Plan for Inheritance or Avoid It

A class should either be designed specifically to be the base of subclasses or explicitly disallowed from being used as a base class:

```
// Designed for inheritance
class Base {
public:
   virtual ~Base() = default;  // Virtual destructor
   virtual void operation() = 0;  // Pure virtual function
};

// Prohibited from inheritance
class Utility final {
public:
   static void helperFunction();
};
```

The Liskov Substitution Principle

Derived classes must be substitutable for their base classes without affecting the correctness of the program. This is one of the SOLID design principles, which helps to ensure that inheritance hierarchies are well-formed:

```
void processShape(Shape* shape) {
   // Any shape should work here without special cases
   double area = shape->area();
   shape->draw();
}

int main() {
   Circle circle(5);
   Rectangle rectangle(4, 6);

processShape(&circle);      // Should work correctly
processShape(&rectangle);   // Should work correctly
}
```

Virtual Destructors

Always declare destructors as virtual in base classes to ensure proper cleanup of derived objects:

```
class Base {
public:
   virtual ~Base() = default;  // Virtual destructor
};

class Derived : public Base {
private:
   Resource* resource;
public:
   Derived() : resource(new Resource()) {}
   ~Derived() override { delete resource; }  // Will be called correctly
};

int main() {
   Base* ptr = new Derived();
   delete ptr;  // Without virtual destructor, Derived's destructor wouldn't be called
}
```

Rule of Three/Five/Zero

When defining custom destructors in a class hierarchy, follow the Rule of Three (or Five in modern C++):

1. If you need a destructor, you probably need copy constructor and copy assignment
2. In modern C++, also consider move constructor and move assignment
3. Or follow the Rule of Zero: define no custom destructor, copy/move operations if possible

MCQs:

1. **Which of the following operators cannot be overloaded in C++?**
   a) +
   b) =
   c) ::
   d) *

2. **What is operator overloading in C++?**
   a) Using an operator with multiple values
   b) Assigning multiple operators to one function
   c) Redefining an operator to work with user-defined data types
   d) Making an operator a function
3. **Which keyword is used to declare a friend function in C++?**
   a) friend
   b) private
   c) virtual
   d) public
4. **Which of the following is NOT a rule of operator overloading?**
   a) At least one operand must be a user-defined type
   b) Overloading an operator must preserve its basic functionality
   c) Operators ::, sizeof, and .* can be overloaded
   d) Overloading cannot change operator precedence
5. **Which type of inheritance allows a derived class to inherit from more than one base class?**
   a) Single Inheritance
   b) Multilevel Inheritance
   c) Multiple Inheritance
   d) Hierarchical Inheritance
6. **Which function is called first when a derived class object is created?**
   a) Derived class constructor
   b) Base class constructor
   c) Destructor
   d) Member function
7. **What is the purpose of a virtual base class?**
   a) To improve program execution speed
   b) To avoid multiple instances of the base class in multiple inheritance
   c) To allow redefinition of private members
   d) To restrict inheritance
8. **An abstract class is a class that:**
   a) Cannot have objects

b) Must have all pure virtual functions
c) Cannot be inherited
d) Can only contain static functions

9. **Which function type must be overridden in a derived class when declared in a base class?**
   a) Friend function
   b) Virtual function
   c) Inline function
   d) Destructor
10. **Which of the following is an example of a constructor in a derived class?**
    a) Base() { }
    b) Derived() : Base() { }
    c) void Derived();
    d) Derived(int x);

Short Questions:

1. What is operator overloading, and why is it used?
2. Explain the difference between unary and binary operator overloading.
3. How can a friend function be used to overload binary operators?
4. What are the rules of operator overloading?
5. Define inheritance in C++ and give an example.
6. Differentiate between single and multiple inheritance.
7. What is the purpose of virtual base classes?
8. Explain the concept of abstract classes in C++.
9. How do constructors work in derived classes?
10. What is the role of type conversion in operator overloading?

Long Questions:

1. Explain operator overloading with examples of unary and binary operator overloading.
2. Write a C++ program demonstrating operator overloading using friend functions.
3. Discuss the rules and restrictions of operator overloading.
4. What is inheritance? Explain different types of inheritance with examples.
5. How does multiple inheritance work? Write a program to illustrate it.

6. What are virtual base classes, and how do they prevent ambiguity in multiple inheritance?
7. Define abstract classes and explain their significance with an example.
8. How do constructors work in derived classes? Write a program to demonstrate their usage.
9. Compare and contrast function overloading, operator overloading, and method overriding.
10. Explain the process of type conversion in operator overloading and provide examples.

MODULE 4
POINTER, VIRTUAL FUNCTION, AND POLYMORPHISM

4.0 LEARNING OUTCOMES

- Understand the concept of pointers to objects and this pointer.
- Learn about pointers to derived classes and how they work.
- Understand virtual functions and their role in achieving runtime polymorphism.
- Learn about pure virtual functions and abstract classes.
- Explore compile-time and runtime polymorphism.
- Understand the difference between overloading and overriding.

Unit 10: Pointers in C++

4.1 Pointers: Pointers to Objects, This Pointer

Introduction to Pointers in C++

A Pointer in C++ is a variable that contains the address of another variable. They form the base for many high-level programming abstractions and allow you to manipulate memory in an efficient manner. In the context of objects and classes, pointers become particularly powerful, as they enable dynamic memory allocation, polymorphism, and efficient object manipulation

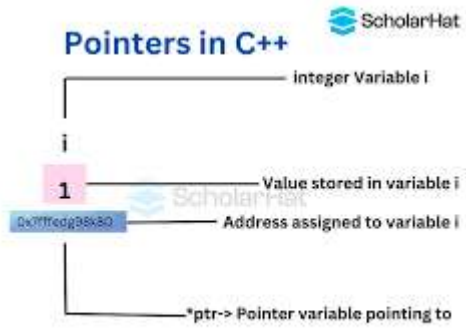


Figure 5 Concept of Pointers in OOP'S
[Source https://www.scholarhat.com]

Basic Pointer Syntax

A pointer is declared using the asterisk (*) symbol:

```
int* pInteger;       // Pointer to an integer
double* pDouble;     // Pointer to a double
char* pChar;         // Pointer to a character
```

To initialize a pointer, we assign it the address of a variable using the address-of operator (&):

```
int number = 10;
int* pNumber = &number;  // pNumber now points to number
```

To access the value at the memory address stored in a pointer (dereferencing), we use the asterisk operator:

```
int value = *pNumber;  // value = 10
```

Pointers to Objects

Just like we can create pointers to primitive data types, we can create them to objects. It is especially helpful during dynamic memory allocation or polymorphism implementation.

```
class Rectangle {
private:
   int length;
   int width;
public:
   Rectangle(int l = 0, int w = 0) : length(l), width(w) {}
   int getArea() { return length * width; }
};

// Creating a pointer to a Rectangle object
Rectangle rect(5, 3);
Rectangle* pRect = &rect;

// Accessing members using the pointer
int area = pRect->getArea();  // Using arrow operator -> to access members
```

When working with pointers to objects, we use the arrow operator (->) to access the object's members. This is equivalent to dereferencing the pointer and then using the dot operator.

```
int area1 = pRect->getArea();      // Using arrow operator
int area2 = (*pRect).getArea();    // Dereferencing and then using dot operator
```

Dynamic Memory Allocation for Objects

We can use pointers to dynamically allocate memory for objects using the new operator:

```
Rectangle* pDynamicRect = new Rectangle(10, 5);
```

When we're done with the dynamically allocated object, we must release the memory using the delete operator:

```
delete pDynamicRect;
pDynamicRect = nullptr;  // Good practice to set pointer to nullptr after deletion
```

For arrays of objects:

```
Rectangle* pRectArray = new Rectangle[5];  // Array of 5 Rectangle objects
// ...
delete[] pRectArray;  // Note the square brackets for deleting arrays
pRectArray = nullptr;
```

The this Pointer

In C++, every non-static member function receives a hidden pointer called this, which points to the object that called the function. The this pointer is implicitly used when accessing members of the class.

```
class Counter {
private:
   int count;
public:
   Counter(int c = 0) : count(c) {}

   // Using this pointer explicitly
   void increment() {
      this->count++;  // Equivalent to count++
   }

   // Returning *this allows for method chaining
   Counter& add(int value) {
      count += value;
      return *this;
   }

   int getCount() {
      return count;  // Implicitly uses this->count
   }
};
```

The this pointer serves several purposes:

1. **Resolving Name Conflicts**: When a parameter has the same name as a member variable.

```
class Person {
private:
   std::string name;
public:
   Person(const std::string& name) {
      this->name = name;  // Resolves the name conflict
   }
};
```

2. **Enabling Method Chaining**: By returning a reference to the current object.

```
Counter counter;
counter.add(5).add(10).add(15);  // Method chaining
```

3. **Identifying the Current Object**: Particularly useful in complex scenarios or when passing the current object to functions.

```
class Node {
private:
   int data;
   Node* next;
public:
   Node(int value) : data(value), next(nullptr) {}

   void setNext(Node* node) {
      next = node;
   }

   void connectToSelf() {
      next = this;  // Creating a self-referential structure
   }
};
```

Common Mistakes and Best Practices with Pointers

1. **Memory Leaks**: Always delete dynamically allocated memory when it's no longer needed.
2. **Dangling Pointers**: After deleting memory, set pointers to nullptr to avoid accessing freed memory.
3. **Null Pointer Dereferencing**: Always check if a pointer is nullptr before dereferencing it.
4. **Smart Pointers**: Consider using C++11's smart pointers (unique_ptr, shared_ptr) to avoid manual memory management issues.

```
#include <memory>

// Using smart pointers instead of raw pointers
std::unique_ptr<Rectangle>pRect                                    =
std::make_unique<Rectangle>(10, 5);
// No need to manually delete, memory is automatically managed
```

4.2 Pointer to Derived Classes

Inheritance and Pointers

One of the highlights of C++ is the ability to point to any derived class from a base class pointer. This enables polymorphism:

```
class Shape {
public:
   virtual void draw() {
      std::cout<< "Drawing a shape" << std::endl;
   }
};

class Circle : public Shape {
public:
   void draw() override {
      std::cout<< "Drawing a circle" << std::endl;
   }
};

class Rectangle : public Shape {
public:
   void draw() override {
      std::cout<< "Drawing a rectangle" << std::endl;
   }
};
```

Base Class Pointers to Derived Class Objects

A key feature of C++ is that a pointer to a base class can point to objects of any derived class. This enables polymorphism:

```
Shape* pShape1 = new Circle();
Shape* pShape2 = new Rectangle();

pShape1->draw();  // Outputs: "Drawing a circle"
pShape2->draw();  // Outputs: "Drawing a rectangle"

delete pShape1;
delete pShape2;
```

This works as a derived class object has a base class subobject inside it, hence it is type-compatible to the base class.

Upcasting and Downcasting

Pointers in inheritance hierarchies can be used for two types of conversions:

1. **Upcasting**: Converting a derived class pointer to a base class pointer.
    - This is implicit and always safe.

```
Circle* pCircle = new Circle();
Shape* pShape = pCircle;  // Implicit upcast, always safe
```

2. **Downcasting**: Converting a base class pointer to a derived class pointer.
    - This requires explicit casting and can be dangerous if not done correctly.

```
Shape* pShape = new Circle();
// Two ways to downcast:

// 1. Using static_cast (no runtime check)
Circle* pCircle1 = static_cast<Circle*>(pShape);

// 2. Using dynamic_cast (safer, includes runtime type checking)
Circle* pCircle2 = dynamic_cast<Circle*>(pShape);
if (pCircle2) {
   // Successfully cast to Circle*
}
```

Dynamic Cast and Runtime Type Identification (RTTI)

dynamic_cast provides type-safe downcasting. It checks at runtime whether the conversion is valid:

```
Shape* pShape = new Rectangle();

// This will return nullptr since pShape points to a Rectangle, not a Circle
Circle* pCircle = dynamic_cast<Circle*>(pShape);
if (pCircle) {
pCircle->draw();
} else {
   std::cout<< "Not a Circle object" << std::endl;
}
```

```
// This will succeed
Rectangle* pRect = dynamic_cast<Rectangle*>(pShape);
if (pRect) {
pRect->draw();
}

delete pShape;
```

Arrays of Pointers to Objects

Using arrays of pointers to objects is a common way to manage collections of objects in an inheritance hierarchy:

```
const int SIZE = 3;
Shape* shapes[SIZE];

shapes[0] = new Circle();
shapes[1] = new Rectangle();
shapes[2] = new Circle();

// Polymorphic behavior through base class pointers
for (int i = 0; i< SIZE; i++) {
   shapes[i]->draw();
}

// Cleaning up memory
for (int i = 0; i< SIZE; i++) {
   delete shapes[i];
}
```

4.3 Virtual Function, Pure Virtual Function

Virtual Functions

C++ strives for polymorphism via virtual functions. A member function that is declared in a base class as virtual and redefined (overridden) in the derived classes. The virtual keyword tells the compiler to look for an overridden version of the function in derived classes at runtime.

```
class Base {
public:
   virtual void display() {
      std::cout<< "Display from Base" << std::endl;
   }
};

class Derived : public Base {
public:
   void display() override {   // override keyword is optional but recommended (C++11)
      std::cout<< "Display from Derived" << std::endl;
   }
};

// Using polymorphism
Base* ptr = new Derived();
ptr->display();  // Outputs: "Display from Derived"
delete ptr;
```

How Virtual Functions Work: The Virtual Function Table (vtable)

If a class has any virtual functions, the compiler will create a virtual function table (vtable) for that class. Each class object has a hidden pointer (vptr) to this table. The virtual table (vtable) contains function pointers for the virtual functions that need to be invoked for objects of that class.

At runtime, when a virtual function is called through a base class pointer:

1. The program uses the object's vptr to locate its class's vtable
2. It looks up the appropriate function address in the vtable
3. It calls the function at that address

This mechanism allows the correct derived class function to be called, even when accessed through a base class pointer.

The Importance of Virtual Destructors

When using polymorphism, it's crucial to make the base class destructor virtual. This ensures that when an object is deleted through a base class pointer, the correct destructor chain is called.

```
class Base {
public:
   virtual ~Base() {
      std::cout<< "Base destructor" << std::endl;
   }
};

class Derived : public Base {
public:
   ~Derived() override {
      std::cout<< "Derived destructor" << std::endl;
   }
};

// Without a virtual destructor, only the Base destructor would be called
Base* ptr = new Derived();
delete ptr;  // Calls both Derived and Base destructors
```

Pure Virtual Functions

A pure virtual function is a virtual function that has no implementation in the base class and must be overridden by any concrete derived class. It is declared by adding = 0 to the function declaration:

```
class Shape {
public:
   virtual void draw() = 0;  // Pure virtual function
   virtual double area() = 0;  // Another pure virtual function
};
```

Abstract Classes

A class containing at least one pure virtual function is called an abstract class. Abstract classes:

1. Cannot be instantiated directly
2. Are used as interfaces or base classes
3. Require derived classes to implement all pure virtual functions to be concrete

```
// Shape is an abstract class
// Shape shape;  // Error: cannot instantiate abstract class

class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}

   void draw() override {
      std::cout<< "Drawing a circle" << std::endl;
   }

   double area() override {
      return 3.14159 * radius * radius;
   }
};

// Circle is a concrete class that can be instantiated
Circle circle(5.0);
```

Interfaces in C++

While C++ doesn't have a specific "interface" keyword like some other languages, abstract classes with only pure virtual functions serve as interfaces:

```
class Drawable {
public:
   virtual void draw() = 0;
   virtual ~Drawable() = default;  // Virtual destructor
};

class Resizable {
public:
```

```
    virtual void resize(double factor) = 0;
    virtual ~Resizable() = default;
};

// Multiple interface implementation
class Rectangle : public Drawable, public Resizable {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    void draw() override {
        std::cout<< "Drawing a rectangle" << std::endl;
    }

    void resize(double factor) override {
        width *= factor;
        height *= factor;
    }
};
```

Virtual Function Override Rules

When overriding virtual functions, several rules must be followed:

1. The function signature (return type, name, parameters) must match exactly (except for covariant return types)
2. The function must be declared virtual in the base class
3. The function must be accessible to the derived class (public or protected in the base class)
4. The function cannot have a more restrictive access modifier in the derived class

Using the override keyword (C++11) helps catch errors by explicitly stating that a function is intended to override a virtual function:

```
class Base {
public:
    virtual void func(int x) {
        // Implementation
    }
};
```

```
class Derived : public Base {
public:
   // The override keyword causes a compiler error if this doesn't actually
   // override a base class function
   void func(int x) override {
      // Implementation
   }

   // This would cause a compiler error with override:
   // void func(double x) override;  // Error: doesn't override anything
};
```

4.4 Polymorphism: Compile Time, Run Time

Polymorphism (meaning "many forms") is a foundational principle of object-oriented programming that allows objects of different classes to be treated as objects of the same class through a common base class. There are two major types of polymorphism in C++: compile time and run time.

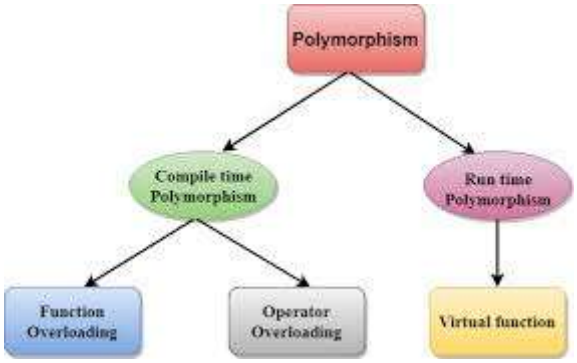


Figure 6 Types of Polymorphism

[https://www.tpointtech.com]

Compile-Time Polymorphism (Static Binding)

Compile-time polymorphism is resolved during compilation. It includes:

1. **Function Overloading**: Multiple functions with the same name but different parameters.
2. **Operator Overloading**: Redefining the behavior of operators for custom types.
3. **Templates**: Generic programming that handles different types.

Function Overloading

```
class Calculator {
public:
   int add(int a, int b) {
      return a + b;
   }

   double add(double a, double b) {
      return a + b;
   }

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

```
        return a + b + c;
    }
};

Calculator calc;
int sum1 = calc.add(5, 10);          // Calls add(int, int)
double sum2 = calc.add(3.5, 7.2);     // Calls add(double, double)
int sum3 = calc.add(1, 2, 3);         // Calls add(int, int, int)
```

The compiler determines which function to call based on the number and types of arguments.

Operator Overloading

```
class Complex {
private:
    double real;
    double imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overloading the + operator
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // Overloading the << operator for output
    friend std::ostream& operator<<(std::ostream&os, const Complex&
c) {
os<<c.real<< " + " <<c.imag<< "i";
        return os;
    }
};

Complex c1(3.0, 4.0);
Complex c2(2.0, 5.0);
Complex c3 = c1 + c2;  // Uses the overloaded + operator
std::cout<< c3;        // Uses the overloaded << operator, outputs: "5 +
9i"
```

Templates (Generic Programming)

```
template <typename T>
```

```
T maximum(T a, T b) {
return (a > b) ? a : b;
}

int maxInt = maximum<int>(10, 20);           // maxInt = 20
double maxDouble = maximum<double>(3.14, 2.72); // maxDouble = 3.14
char maxChar = maximum<char>('A', 'Z');         // maxChar = 'Z'
```

Templates allow you to write generic functions or classes that work with any data type, providing compile-time polymorphism.

Runtime Polymorphism (Dynamic Binding)

Runtime polymorphism is resolved during program execution. It's achieved through:

1. **Virtual Functions**: Member functions that can be overridden in derived classes.
2. **Function Overriding**: Redefining a base class function in derived classes.

Runtime polymorphism relies on virtual functions and inheritance hierarchies:

```
class Animal {
public:
   virtual void makeSound() {
      std::cout<< "Animal makes a sound" << std::endl;
   }
};

class Dog : public Animal {
public:
   void makeSound() override {
      std::cout<< "Dog barks: Woof!" << std::endl;
   }
};

class Cat : public Animal {
public:
   void makeSound() override {
      std::cout<< "Cat meows: Meow!" << std::endl;
   }
```

```
};

// Runtime polymorphism using base class pointers
Animal* animal1 = new Dog();
Animal* animal2 = new Cat();

animal1->makeSound();  // Outputs: "Dog barks: Woof!"
animal2->makeSound();  // Outputs: "Cat meows: Meow!"

delete animal1;
delete animal2;
```

Table 4.1: Key Differences Between Compile-Time and Runtime Polymorphism

Feature	Compile-Time Polymorphism	Runtime Polymorphism
Binding	Static (early) binding	Dynamic (late) binding
Performance	Generally faster	Slightly slower due to vtable lookups
Flexibility	Less flexible, fixed at compile time	More flexible, determined at runtime
Implementation	Function overloading, operator overloading, templates	Virtual functions, inheritance
Resolution	Resolved by the compiler	Resolved at runtime

Practical Example Combining Both Types of Polymorphism

```
class Shape {
public:
   // Runtime polymorphism through virtual functions
   virtual double area() const = 0;
   virtual void draw() const = 0;

   // Compile-time polymorphism through function overloading
   void print() const {
```

```
        std::cout<< "Area: " << area() << std::endl;
    }

    void print(const std::string& prefix) const {
        std::cout<< prefix << " area: " << area() << std::endl;
    }
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout<< "Drawing a circle" << std::endl;
    }
};

class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }

    void draw() const override {
        std::cout<< "Drawing a rectangle" << std::endl;
    }
};
```

```
// Using both types of polymorphism
Shape* shapes[2] = { new Circle(5.0), new Rectangle(4.0, 6.0) };
for (int i = 0; i< 2; i++) {
   shapes[i]->draw();          // Runtime polymorphism
   shapes[i]->print();              // Compile-time polymorphism (first version)
   shapes[i]->print("Shape");   // Compile-time polymorphism (second version)
}

delete shapes[0];
delete shapes[1];
```

Virtual Function Tables and Runtime Polymorphism Implementation

Knowing the internals of vtables will shed some light on how runtime polymorphism works:

- Every class with virtual function have a vtable which holds the address of the functions
- Every instance of such a class will have a secret vptr (virtual table pointer) to its class's vtable
- When a virtual function is invoked by means of a pointer or reference, the process:
- Retrieves the vptr of the object to locate the vtable
- Finds the corresponding function in the vtable
- Invokes the function at that address in memory

This dynamic dispatch mechanism allows runtime polymorphism. The name from the related function table is matched to the name you have used in your source code.

4.5 Overloading and Overriding

While both overloading and overriding are forms of polymorphism, they serve different purposes and operate differently.

Function Overloading

Function overloading allows multiple functions with the same name but different parameter lists to coexist in the same scope. It is a form of compile-time polymorphism.

Key Characteristics of Function Overloading:

1. Same function name with different parameter lists
2. Can differ in:

- Number of parameters
- Type of parameters
- Order of parameters

3. Cannot differ only in return type
4. All overloaded functions exist independently
5. Resolution is done at compile time based on the arguments passed

Example of Function Overloading:

```
class MathOperations {
public:
   // Overloaded functions
   int add(int a, int b) {
      return a + b;
   }

   double add(double a, double b) {
      return a + b;
   }

   int add(int a, int b, int c) {
      return a + b + c;
   }

   double add(int a, double b) {
      return a + b;
   }

   double add(double a, int b) {
      return a + b;
   }
};

MathOperations math;
int sum1 = math.add(5, 10);            // Calls add(int, int)
double sum2 = math.add(3.5, 7.2);      // Calls add(double, double)
int sum3 = math.add(1, 2, 3);          // Calls add(int, int, int)
double sum4 = math.add(5, 3.14);       // Calls add(int, double)
double sum5 = math.add(2.71, 8);       // Calls add(double, int)
```

Function Overriding

Function overriding occurs when a derived class provides a specific implementation for a function already defined in its base class. It is a form of runtime polymorphism.

Key Characteristics of Function Overriding:

1. Occurs in an inheritance hierarchy
2. Base class function must be declared as virtual
3. Derived class function must have the **exact same signature** (name, parameters, and return type) as the base class function
    - Exception: Covariant return types are allowed (returning a derived type when the base returns the base type)
4. Access specifier in the derived class cannot be more restrictive than in the base class
5. Resolution is done at runtime based on the object's actual type

Example of Function Overriding:

```
class Vehicle {
public:
   virtual void start() {
      std::cout<< "Vehicle starting..." << std::endl;
   }

   virtual void stop() {
      std::cout<< "Vehicle stopping..." << std::endl;
   }

   // Not virtual, can't be overridden polymorphically
   void maintenance() {
      std::cout<< "Vehicle maintenance" << std::endl;
   }
};

class Car : public Vehicle {
public:
   // Override virtual function
   void start() override {
      std::cout<< "Car engine starting..." << std::endl;
   }
```

```
    // Override virtual function
    void stop() override {
        std::cout<< "Car engine stopping, applying brakes..." << std::endl;
    }

    // Not an override, just a new function with the same name
    void maintenance() {
        std::cout<< "Car maintenance" << std::endl;
    }
};

Vehicle* vehicle = new Car();
vehicle->start();       // Calls Car::start() - polymorphic
vehicle->stop();        // Calls Car::stop() - polymorphic
vehicle->maintenance(); // Calls Vehicle::maintenance() - not polymorphic

Car car;
car.maintenance();     // Calls Car::maintenance()

delete vehicle;
```

Operator Overloading

Operator overloading is a special case of function overloading that allows custom implementations of C++ operators for user-defined types.

Rules for Operator Overloading:

1. Cannot change the operator precedence
2. Cannot change the number of operands
3. Cannot create new operators
4. Some operators cannot be overloaded (., ::, ?, sizeof)
5. Some operators can only be overloaded as member functions (=, [], (), ->)
6. Should maintain the semantic meaning of the operator

Example of Operator Overloading:

```
class Vector {
private:
```

```
    double x, y, z;
public:
    Vector(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z)
{}

    // Overload + operator (member function)
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y, z + other.z);
    }

    // Overload - operator (member function)
    Vector operator-(const Vector& other) const {
        return Vector(x - other.x, y - other.y, z - other.z);
    }
    // Overload * operator for scalar multiplication (member function)
    Vector operator*(double scalar) const {
        return Vector(x * scalar, y * scalar, z * scalar);
    }

    // Overload == operator (member function)
    bool operator==(const Vector& other) const {
        return (x == other.x&& y == other.y&& z == other.z);
    }

    // Overload << operator (friend function)
    friend std::ostream& operator<<(std::ostream&os, const Vector&
v) {
os<< "(" <<v.x<< ", " <<v.y<< ", " <<v.z<< ")";
        return os;
    }

    // Overload * operator for scalar multiplication (friend function)
    friend Vector operator*(double scalar, const Vector& v) {
        return v * scalar;  // Reuse the member function
    }
};

Vector v1(1.0, 2.0, 3.0);
```

```
Vector v2(4.0, 5.0, 6.0);

Vector v3 = v1 + v2;          // Using overloaded + operator
Vector v4 = v1 - v2;          // Using overloaded - operator
Vector v5 = v1 * 2.0;         // Using overloaded * operator (member)
Vector v6 = 3.0 * v2;         // Using overloaded * operator (friend)
bool equal = (v1 == v2);      // Using overloaded == operator
std::cout<< v3;               // Using overloaded << operator
```

Method Hiding (Name Hiding)

Method hiding is where a derived class defines a function with the same name as a function in the base class, except that the signature is different. In contrast to overriding, hiding is not a polymorphic action; it only hides the base class function behind the derived class..

```
class Base {
public:
   void display() {
      std::cout<< "Base display()" << std::endl;
   }

   void display(int x) {
      std::cout<< "Base display(int): " << x << std::endl;
   }
};

class Derived : public Base {
public:
   // This hides all Base::display() functions
   void display() {
      std::cout<< "Derived display()" << std::endl;
   }
};

Base b;
b.display();      // Calls Base::display()
b.display(5);     // Calls Base::display(int)

Derived d;
d.display();      // Calls Derived::display()
// d.display(5);  // Error: Derived::display() hides Base::display(int)

// To access the hidden base class function
d.Base::display(5);  // OK: Explicitly calling Base::display(int)
```

To prevent hiding and make all overloads from the base class available:

```
class Better : public Base {
public:
```

```
    // Bring all Base::display() functions into scope
    using Base::display;

    // Override just one version
    void display() {
        std::cout<< "Better display()" << std::endl;
    }
};

Better better;
better.display();    // Calls Better::display()
better.display(5);   // Calls Base::display(int) - now accessible!
```

Table 4.2: Key Differences between Overloading and Overriding

Feature	Overloading	Overriding
Purpose	Provides multiple functions with the same name but different parameters	Provides a specific implementation in derived classes
Scope	Same class or namespace	Base and derived class hierarchy
Function signatures	Same name, different parameters	Same name and parameters
Return type	Can differ as long as parameters differ	Must be same or covariant
Resolution time	Compile time	Runtime
Virtual keyword	Not required	Required in base class
Polymorphism type	Compile-time polymorphism	Runtime polymorphism

Best Practices for Overloading and Overriding

1. **Use override keyword** for functions that override virtual functions (C++11)
2. **Use final keyword** to prevent further overriding (C++11)
3. **Make base class destructors virtual** when using polymorphism

4. **Maintain consistent semantics** when overloading operators
5. **Consider using using declarations** to prevent unintended hiding
6. **Document the behavior** of overloaded and overridden functions

```
class Base {
public:
   virtual void func1() { /* ... */ }
   virtual void func2() final { /* ... */ }  // Cannot be overridden further
   virtual ~Base() = default;
};

class Derived : public Base {
public:
   void func1() override { /* ... */ }  // Clearly marked as override
   // void func2() override { /* ... */ }  // Error: cannot override final function
};
```

MCQs:

1. **What does the this pointer in C++ refer to?**
   a) The current object of a class
   b) A global object
   c) A derived class object
   d) A pointer to a base class
2. **What is the purpose of a pointer to an object?**
   a) To point to primitive data types
   b) To allow indirect access to class members
   c) To delete an object from memory
   d) To create a new object
3. **Which of the following correctly declares a pointer to an object?**
   a) Class obj;
   b) Class *ptr = new Class();
   c) Class ptr;
   d) int *ptr = &obj;
4. **What is a virtual function?**
   a) A function that is automatically executed at runtime

b) A function that allows function overriding in derived classes
c) A function that cannot be inherited
d) A function that is only used in templates

5. **Which keyword is used to declare a virtual function?**
   a) friend
   b) inline
   c) virtual
   d) static
6. **Which of the following is a characteristic of a pure virtual function?**
   a) It is implemented in the base class
   b) It has no implementation in the base class
   c) It cannot be inherited
   d) It must be private
7. **What is polymorphism in C++?**
   a) The ability of different objects to respond to the same function call in different ways
   b) The ability to store multiple data types in an array
   c) The ability to perform multiple loops at once
   d) The ability to overload an operator
8. **Which of the following is an example of compile-time polymorphism?**
   a) Virtual functions
   b) Function overloading
   c) Function overriding
   d) Dynamic binding
9. **Function overriding is an example of:**
   a) Compile-time polymorphism
   b) Run-time polymorphism
   c) Operator overloading
   d) Static linking
10. **Which of the following is NOT a feature of virtual functions?**
    a) They enable runtime polymorphism
    b) They can be overridden in derived classes
    c) They must be declared using the friend keyword
    d) They allow dynamic method dispatch

Short Questions:

1. What is a pointer to an object, and why is it useful?
2. Explain the purpose of the this pointer in C++.
3. How do pointers to derived classes work in C++?
4. Define virtual functions and explain their significance.
5. What is a pure virtual function, and how is it different from a virtual function?
6. What is polymorphism, and why is it important in OOP?
7. Differentiate between compile-time and runtime polymorphism.
8. Explain function overloading and function overriding with examples.
9. How does dynamic binding work in C++?
10. What are the advantages of using virtual functions in inheritance?

Long Questions:

1. Explain the concept of pointers to objects with an example program.
2. What is this pointer, and how does it help in object-oriented programming?
3. How does pointer to derived classes work? Provide an example.
4. Explain virtual functions and their role in achieving runtime polymorphism.
5. What is a pure virtual function? How does it relate to abstract classes?
6. Discuss the difference between function overloading and function overriding.
7. Explain compile-time vs. runtime polymorphism with examples.
8. Write a C++ program demonstrating the use of virtual functions.
9. How does method override support polymorphism in object-oriented programming?
10. Explain the advantages and disadvantages of polymorphism in C++.

MODULE 5
CONSOLE I/O OPERATIONS AND FILE HANDLING

5.0 LEARNING OUTCOMES

- Understand the stream classes in C++.
- Learn about formatted and unformatted I/O operations.
- Learn how to manage output using manipulators.
- Understand file stream operations and how to handle files in C++.
- Learn how to open and close files, detect the End-of-File (EOF) condition, and use file modes.
- Understand file pointers and their manipulations.
- Learn about sequential and random-access file operations.
- Understand error handling in file operations.

Unit 13: Console I/O Operations

5.1 Stream Classes

Intended for C++ File Stream classes are base for output and input operations. They offer a uniform interface for interacting with everything from the keyboard and display to files and memory buffers. It is very important that you understand what these classes are. There is an inheritance hierarchy for the C++ Standard Library stream classes, which are meant for specific use cases in the I/O system. It encourages the reuse of code and provides an abstract layer for dealing with different types of I/O.

Stream Class Hierarchy

The base classes ios_base and ios are at the top of this hierarchy. These classes define the basic attributes and behaviour common to all streams. Many important stream classes are derived from these:

Input Stream Classes

- **istream**: Handles input operations from various sources
- **ifstream**: Specializes in input operations from files
- **istringstream**: Manages input from string objects

Output Stream Classes

- **ostream**: Handles output operations to various destinations
- **ofstream**: Specializes in output operations to files
- **ostringstream**: Manages output to string objects

Bidirectional Stream Classes

- **iostream**: Combines the functionality of istream and ostream for bidirectional operations
- **fstream**: Provides bidirectional file operations
- **stringstream**: Enables bidirectional string operations

Template Classes and Character Types

The C++ I/O library uses template classes to support different character types:

```
// For char type (standard ASCII characters)
typedef basic_istream<char>istream;
typedef basic_ostream<char>ostream;
typedef basic_iostream<char> iostream;
// etc.
```

```
// For wchar_t type (wide characters)
typedef basic_istream<wchar_t>wistream;
typedef basic_ostream<wchar_t>wostream;
typedef basic_iostream<wchar_t>wiostream;
// etc.
```

This design allows the same code structure to handle both narrow and wide character streams, enhancing the library's flexibility.

Standard Stream Objects

C++ provides several predefined stream objects for common I/O operations:

- **cin**: Standard input stream (keyboard by default)
- **cout**: Standard output stream (screen by default)
- **cerr**: Standard error output stream (unbuffered)
- **clog**: Standard error output stream (buffered)

And their wide-character counterparts:

- **wcin**: Wide-character standard input
- **wcout**: Wide-character standard output
- **wcerr**: Wide-character unbuffered error output
- **wclog**: Wide-character buffered error output

These objects are automatically created when a C++ program starts, providing immediate access to standard I/O capabilities.

Stream Buffer Classes

Every stream object contains a buffer object that manages the actual transfer of data between the program and the external device. Key buffer classes include:

- **streambuf**: Base class for all stream buffers
- **filebuf**: Buffer for file operations
- **stringbuf**: Buffer for string operations

These buffer classes handle the low-level details of reading from and writing to different devices, allowing the stream classes to present a consistent interface regardless of the underlying I/O mechanism.

5.2 I/O Operations: Unformatted and Formatted

There are two main types of input and output operations in C++ — unformatted I/O and formatted I/O. Each type has its own use cases and provides varying degrees of control over data processing.

Unformatted I/O Operations

Unformatted I/O simply reads/writes the data as raw byte sequences, without any interpretation or conversion. In particular, these operations are well suited for binary data or when precise control of the input and output is necessary.

Unformatted Input Operations

The istream class provides several methods for unformatted input:

- **get()**: Reads a single character
- char ch;
- cin.get(ch); // Reads one character into ch
- *get(char*, int, char)**: Reads characters into a buffer until a delimiter is encountered
- char buffer[100];
- cin.get(buffer, 100, '\n'); // Reads up to 99 chars or until newline
- *getline(char*, int, char)**: Similar to get() but extracts and discards the delimiter
- char buffer[100];
- cin.getline(buffer, 100, '\n'); // Reads a line of up to 99 characters
- *read(char*, int)**: Reads a specified number of bytes
- char buffer[100];
- cin.read(buffer, 50); // Reads exactly 50 bytes
- **gcount()**: Returns the number of characters extracted by the last unformatted input operation
- cout<< "Characters read: " <<cin.gcount() <<endl;

Unformatted Output Operations

The ostream class provides these unformatted output methods:

- **put()**: Writes a single character
- cout.put('A'); // Outputs the character 'A'
- **write()**: Writes a specified number of bytes
- char buffer[10] = "Hello";
- cout.write(buffer, 5); // Writes 5 bytes from buffer
- **flush()**: Flushes the output buffer

- cout<< "Immediate output" << flush; // Ensures output is written immediately

Formatted I/O Operations

Formatted I/O interprets data according to its type, converting between internal representation and human-readable format. This is the more commonly used approach for most applications.

Formatted Input Operations

The primary mechanism for formatted input is the extraction operator (>>):

```
int num;
double value;
string name;

cin>> num >> value >> name;  // Reads formatted data into variables
```

Key points about the extraction operator:

- Skips leading whitespace by default
- Converts external text representation to the appropriate internal format
- Stops reading at whitespace or invalid characters for the target type
- Sets error flags if the input doesn't match the expected format

Formatted Output Operations

The insertion operator (<<) handles formatted output:

```
int num = 42;
double value = 3.14159;
string name = "C++";

cout<< "Number: " << num << ", Value: " << value << ", Name: " << name <<endl;
```

Key points about the insertion operator:

- Converts internal data to text representation
- Applies formatting according to the stream's current format state
- Does not automatically add spaces between items
- Can be chained for multiple output operations

Type Safety in I/O Operations

C++ I/O operations are type-safe, meaning that the compiler ensures that data is handled according to its type. The operators << and >> are

overloaded for different data types, allowing the same syntax to work correctly with integers, floating-point numbers, strings, and user-defined types.

User-defined types can participate in formatted I/O by overloading these operators:

```
class Person {
   string name;
   int age;
public:
   // Constructor and other methods...

   friend ostream& operator<<(ostream&os, const Person& p) {
      return os<< p.name << " (age " <<p.age<< ")";
   }

   friend istream& operator>>(istream& is, Person& p) {
      return is >> p.name >>p.age;
   }
};
```

This extensibility makes C++ I/O both powerful and flexible, accommodating a wide range of data types and formatting requirements.

5.3 Managing Output with Manipulators

Manipulators are special types of functions used to change how a format state is applied to a stream. They allow you to define a customizable way of inputting and outputting the data without modifying the actual data.

Basic Manipulators

C++ provides several basic manipulators that don't require additional parameters:

End-of-Line Manipulators

- **endl**: Inserts a newline character and flushes the buffer
- cout<< "Hello" <<endl; // Outputs "Hello" followed by a newline and flushes
- **ends**: Inserts a null character
- cout<< "Hello" << ends; // Outputs "Hello" followed by a null character
- **flush**: Flushes the output buffer without adding any characters

- cout<< "Hello" << flush; // Outputs "Hello" and flushes

Formatting Boolean Values

- **boolalpha**: Displays bool values as "true" or "false"
- cout<<boolalpha<< true; // Outputs "true" instead of "1"
- **noboolalpha**: Displays bool values as 1 or 0 (default)
- cout<<noboolalpha<< true; // Outputs "1"

Number Base Manipulators

- **dec**: Sets decimal base for integer I/O (default)
- cout<< dec << 16; // Outputs "16"
- **hex**: Sets hexadecimal base for integer I/O
- cout<< hex << 16; // Outputs "10"
- **oct**: Sets octal base for integer I/O
- cout<< oct << 16; // Outputs "20"

Floating-Point Format Manipulators

- **fixed**: Uses fixed-point notation
- cout<< fixed << 3.14159; // Outputs "3.141590"
- **scientific**: Uses scientific notation
- cout<< scientific << 3.14159; // Outputs "3.141590e+00"
- **defaultfloat**: Resets to default floating-point format
- cout<<defaultfloat<< 3.14159; // Outputs "3.14159"

Justification Manipulators

- **left**: Left-justifies output within its field
- cout<< left <<setw(10) << "Hello"; // Outputs "Hello "
- **right**: Right-justifies output (default)
- cout<< right <<setw(10) << "Hello"; // Outputs " Hello"
- **internal**: Uses internal justification (sign left-justified, value right-justified)
- cout<< internal <<setw(10) << -123; // Outputs "- 123"

Parameterized Manipulators

Some manipulators require parameters. To use these, you need to include the <iomanip> header:

- **setw(int)**: Sets the field width
- cout<<setw(10) << "Hello"; // Outputs " Hello" (with default right justification)
- **setprecision(int)**: Sets the precision for floating-point output
- cout<<setprecision(3) << 3.14159; // Outputs "3.14"
- **setfill(char)**: Sets the fill character for padded fields

- cout<<setfill('*') <<setw(10) << "Hello"; // Outputs "*****Hello"
- **setbase(int)**: Sets the base for integer I/O (8, 10, or 16)
- cout<<setbase(16) << 16; // Outputs "10" (hexadecimal)
- **setiosflags(fmtflags)** and **resetiosflags(fmtflags)**: Set or clear format flags
- cout<<setiosflags(ios::showpos) << 42; // Outputs "+42"
- cout<<resetiosflags(ios::showpos) << 42; // Outputs "42"

Creating Custom Manipulators

You can create your own manipulators to encapsulate complex formatting operations:

Parameterless Manipulators

```
ostream& currency(ostream&os) {
os<< "$" << fixed <<setprecision(2);
   return os;
}

// Usage:
cout<< currency << 12.5; // Outputs "$12.50"
```

Parameterized Manipulators

```
class repeat {
   int count;
   char ch;
public:
   repeat(int n, char c) : count(n), ch(c) {}

   friend ostream& operator<<(ostream&os, const repeat& r) {
      for(int i = 0; i<r.count; i++)
os<< r.ch;
      return os;
   }
};

// Usage:
cout<< "Start" << repeat(10, '-') << "End"; // Outputs "Start----------End"
```

Manipulator States and Persistence

Manipulators can have different persistence behaviors:

1. **One-time manipulators**: Affect only the next output operation
2. cout<<setw(10) << "Hello" << "World"; // Only "Hello" is affected by setw(10)
3. **Persistent manipulators**: Affect all subsequent operations until changed
4. cout<< fixed <<setprecision(2);
5. cout<< 3.14159 <<endl; // Outputs "3.14"
6. cout<< 2.71828 <<endl; // Also outputs with 2 decimal places: "2.72"

Understanding the persistence of manipulators is crucial for achieving consistent formatting throughout your program.

5.4 Classes for File Stream Operations

File stream classes in C++ provide specialized functionality for reading from and writing to files. These classes inherit from the basic stream classes and add file-specific capabilities.

File Stream Class Hierarchy

The file stream classes extend the general stream classes with file-specific functionality:

- **ifstream**: Derived from istream, specialized for file input operations
- **ofstream**: Derived from ostream, specialized for file output operations
- **fstream**: Derived from iostream, supports both input and output file operations

These classes include a filebuf object that manages the connection between the stream and the actual file on disk.

Including the Necessary Headers

To use file streams, include the <fstream> header:

```
#include <fstream>
using namespace std;
// Now you can use ifstream, ofstream, and fstream
```

File Stream Class Capabilities

Each file stream class provides specialized functionality:

ifstream (Input File Stream)

The ifstream class is designed for reading data from files:

```
ifstreaminputFile("data.txt");
if (inputFile.is_open()) {
   string line;
   while (getline(inputFile, line)) {
cout<< line <<endl;
   }
inputFile.close();
}
```

Key capabilities:

- Opening files for reading
- Reading data using formatted and unformatted operations
- Checking for end-of-file and error conditions

- Moving the file position pointer

ofstream (Output File Stream)

The ofstream class is designed for writing data to files:

```
ofstreamoutputFile("output.txt");
if (outputFile.is_open()) {
outputFile<< "Hello, file I/O!" <<endl;
outputFile<< 123 << " " << 3.14159 <<endl;
outputFile.close();
}
```

Key capabilities:

- Opening files for writing
- Writing data using formatted and unformatted operations
- Creating new files or truncating existing ones
- Appending to existing files
- Flushing the buffer to ensure data is written

fstream (File Stream)

The fstream class supports both reading and writing:

```
fstreamdataFile("data.dat", ios::in | ios::out | ios::binary);
if (dataFile.is_open()) {
   // Both read and write operations can be performed
   int value = 42;
dataFile.write(reinterpret_cast<char*>(&value), sizeof(value));

dataFile.seekg(0); // Move to the beginning of the file

   int readValue;
dataFile.read(reinterpret_cast<char*>(&readValue),
sizeof(readValue));

dataFile.close();
}
```

Key capabilities:

- Opening files for both reading and writing
- Supporting both input and output operations
- Allowing random access within files
- Supporting binary file operations

File Stream Construction and Initialization

File streams can be constructed and opened in several ways:

Default Construction and Later Opening

```
ifstreaminFile;
inFile.open("input.txt");
```

Construction with File Opening

```
ofstreamoutFile("output.txt");
```

Specifying Open Mode During Construction

```
fstreamdataFile("data.bin", ios::in | ios::out | ios::binary);
```

Working with File Paths

File streams work with file paths, which can be:

1. **Relative paths**: Relative to the current working directory
2. ifstream config("config.ini");
3. ifstream log("logs/app.log");
4. **Absolute paths**: Complete paths from the root directory
5. ifstreamdataFile("/home/user/data/info.txt");

On Windows, backslashes in paths need to be escaped or replaced with forward slashes:

```
ifstreamwinFile("C:\\Users\\Username\\Documents\\file.txt");
// or
ifstreamwinFile("C:/Users/Username/Documents/file.txt");
```

File Stream Buffers

Each file stream contains a file buffer (filebuf) that manages the connection to the physical file:

```
ifstreaminFile("data.txt");
filebuf* inBuf = inFile.rdbuf(); // Get the file buffer

// Buffer properties
streamsize size = inBuf->in_avail(); // Get available characters
```

Understanding file stream classes is essential for effective file I/O in C++ applications. These classes provide a type-safe, object-oriented approach to file operations that integrates seamlessly with the rest of the C++ I/O library.

5.5 Opening and Closing a File, Detecting End-of-File (EOF)

Proper file management in C++ requires understanding how to open files, close them when operations are complete, and detect when you've reached the end of a file.

Opening Files

Files can be opened in two ways: during stream construction or using the open() method.

Opening During Construction

```
ifstreaminputFile("data.txt");
ofstreamoutputFile("output.txt");
fstreamdataFile("data.bin", ios::in | ios::out | ios::binary);
```

Opening Using the open() Method

```
ifstreaminputFile;
inputFile.open("data.txt");

ofstreamoutputFile;
outputFile.open("output.txt", ios::app); // Open in append mode
```

Checking if a File Was Successfully Opened

It's essential to verify that a file was opened successfully before attempting operations:

```
ifstreaminputFile("data.txt");
if (!inputFile) {
cerr<< "Failed to open data.txt" <<endl;
   return 1;
}
// or
if (!inputFile.is_open()) {
cerr<< "Failed to open data.txt" <<endl;
   return 1;
}
```

Closing Files

Files should be closed when they're no longer needed to free system resources and ensure all data is properly written.

Using the close() Method

```
ifstreaminputFile("data.txt");
// File operations...
inputFile.close();
```

Automatic Closing

File streams are automatically closed when they go out of scope or when their destructors are called:

```
void processFile(const string& filename) {
ifstreaminputFile(filename);
   // Process file...
   // No explicit close needed; file will be closed when function returns
```

```
}
```

Reopening a File

After closing a file, you can reopen the same stream with a different file:

```
ifstreamdataFile("first.txt");
// Operations on first.txt...
dataFile.close();

dataFile.open("second.txt");
// Operations on second.txt...
```

Detecting End-of-File (EOF)

Detecting when you've reached the end of a file is crucial for processing file contents completely without attempting to read past the end.

Using the eof() Method

The eof() method returns true if an end-of-file condition has been encountered:

```
ifstreaminputFile("data.txt");
while (!inputFile.eof()) {
   string line;
getline(inputFile, line);
   // Process line...
}
```

However, this approach has a subtle issue: eof() only becomes true after an attempt to read past the end of the file.

Better Approach: Testing the Stream State

A more reliable pattern checks the success of each read operation:

```
ifstreaminputFile("data.txt");
string line;
while (getline(inputFile, line)) {
   // Process line...
}
```

Reading Individual Values

When reading individual values, check each extraction:

```
ifstreamdataFile("numbers.txt");
int value;
while (dataFile>> value) {
   // Process value...
```

```
}
```

Using fail() and bad()

For more detailed error checking:

```
ifstreaminputFile("data.txt");
string line;
while (true) {
getline(inputFile, line);

   if (inputFile.eof()) {
      // Reached end of file normally
      break;
   }

   if (inputFile.fail()) {
      // Read operation failed but recovery might be possible
inputFile.clear(); // Clear error flags
inputFile.ignore(numeric_limits<streamsize>::max(), '\n'); // Skip bad
line
      continue;
   }

   if (inputFile.bad()) {
      // Serious I/O error, recovery unlikely
cerr<< "I/O error while reading file" <<endl;
      break;
   }

   // Process line...
}
```

Best Practices for File Opening and Closing

1. **Always check if a file was opened successfully** before performing operations
2. **Close files explicitly** when they're no longer needed, especially for output files to ensure all data is written
3. **Use RAII (Resource Acquisition Is Initialization) principle** by wrapping file operations in classes or functions to ensure proper cleanup

4. **Handle file errors gracefully** using appropriate error checking and recovery mechanisms
5. **Use appropriate file modes** when opening files to avoid unintended data loss

Following these practices helps ensure robust file I/O operations in C++ applications.

5.6 File Modes, File Pointers, and Their Manipulations

C++ provides various modes for opening files and mechanisms for controlling the position within a file through file pointers.

File Open Modes

When opening a file, you can specify one or more mode flags to control how the file is accessed:

Basic Open Modes

- **ios::in**: Open for input operations (reading)
- **ios::out**: Open for output operations (writing)
- **ios::app**: Append mode; all output operations occur at the end of the file
- **ios::ate**: Position the file pointer at the end of the file upon opening
- **ios::trunc**: Truncate the file to zero length if it exists
- **ios::binary**: Open in binary mode (as opposed to text mode)

Combining Open Modes

Multiple modes can be combined using the bitwise OR operator (|):

```
// Open for both reading and writing in binary mode
fstream file("data.bin", ios::in | ios::out | ios::binary);
// Open for writing, creating a new file or truncating an existing one
ofstreamoutFile("output.txt", ios::out | ios::trunc);

// Open for appending
ofstreamlogFile("log.txt", ios::out | ios::app);
```

Default Modes

Each file stream class has default modes:

- **ifstream**: ios::in
- **ofstream**: ios::out | ios::trunc
- **fstream**: ios::in | ios::out

File Pointers and Their Manipulation

File streams maintain internal pointers that track the current position for reading (get pointer) and writing (put pointer).

Understanding File Pointers

- **get pointer (g)**: Controls where the next read operation occurs
- **put pointer (p)**: Controls where the next write operation occurs

In text mode, these pointers may not directly correspond to byte positions due to platform-specific line ending translations.

Retrieving Current File Position

To get the current file position:

```
ifstream file("data.txt");
streampos position = file.tellg(); // Get the current get pointer position

ofstreamoutFile("output.txt");
streamposoutPosition = outFile.tellp(); // Get the current put pointer position
```

Moving File Pointers

C++ provides several ways to move file pointers:

Absolute Positioning

Move to a specific position from the beginning of the file:

```
file.seekg(100); // Move get pointer to the 100th byte
file.seekp(200); // Move put pointer to the 200th byte
```

Relative Positioning

Move relative to the current position or file ends:

```
// Move get pointer 10 bytes forward from current position
file.seekg(10, ios::cur);
// Move get pointer 10 bytes backward from current position
file.seekg(-10, ios::cur);

// Move get pointer to the beginning of the file
file.seekg(0, ios::beg);

// Move get pointer to the end of the file
file.seekg(0, ios::end);

// Move get pointer 100 bytes before the end of the file
file.seekg(-100, ios::end);
```

The second parameter specifies the reference position:

- **ios::beg**: Beginning of the file
- **ios::cur**: Current position

- **ios::end**: End of the file

Synchronizing Get and Put Pointers

In fstream objects, get and put pointers are normally synchronized. To position both pointers:

```
fstreamdataFile("data.bin", ios::in | ios::out | ios::binary);
dataFile.seekg(100); // Move both pointers to position 100
```

To move them independently, you can use both seekg() and seekp().

Practical Applications of File Modes and Pointers

Creating a New File

```
ofstreamnewFile("new.txt", ios::out | ios::trunc);
// or simply
ofstreamnewFile("new.txt");
```

Appending to an Existing File

```
ofstreamlogFile("log.txt", ios::out | ios::app);
logFile<< "New log entry: " <<getCurrentTime() <<endl;
```

Reading and Writing to the Same File

```
fstreamdataFile("records.dat", ios::in | ios::out | ios::binary);

// Write a record
Record record = {1, "John Doe"};
dataFile.write(reinterpret_cast<char*>(&record), sizeof(Record));

// Move to the beginning and read it back
dataFile.seekg(0);
Record readRecord;
dataFile.read(reinterpret_cast<char*>(&readRecord), sizeof(Record));
```

Updating a Specific Record in a Binary File

```
fstream database("database.bin", ios::in | ios::out | ios::binary);

// Move to the position of the 5th record (assuming fixed-size records)
database.seekg(4 * sizeof(Record));

// Read the record
Record record;
database.read(reinterpret_cast<char*>(&record), sizeof(Record));

// Modify the record
record.value = 100;
```

```
// Move back to the same position and write the updated record
database.seekp(4 * sizeof(Record));
database.write(reinterpret_cast<const char*>(&record), sizeof(Record));
```

Understanding file modes and pointers is essential for complex file operations, especially when working with random access files or when updating specific parts of a file.

5.7 Sequential Input and Output Operations

Sequential file access is the most common pattern for file I/O, where data is read or written in order from the beginning to the end of a file.

Sequential File Input

Sequential input involves reading data from a file one item at a time, moving forward through the file.

Reading Text Files Line by Line

The most common approach for text files is to read them line by line:

```
#include <fstream>
#include <string>
#include <iostream>
using namespace std;

void readTextFile(const string& filename) {
ifstreaminputFile(filename);

   if (!inputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
       return;
   }

   string line;
   while (getline(inputFile, line)) {
cout<< line <<endl;
       // Process the line...
   }

inputFile.close();
}
```

Reading Formatted Data

For files with structured data, you can use the extraction operator:

```
void readStudentRecords(const string& filename) {
ifstreaminputFile(filename);

   if (!inputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
      return;
   }

   string name;
   int id;
   double gpa;

   // Assuming file format: ID Name GPA
   while (inputFile>> id >> name >>gpa) {
cout<< "Student: " << name << ", ID: " << id << ", GPA: "
<<gpa<<endl;
      // Process the student record...
   }

inputFile.close();
}
```

Reading Binary Data

For binary files with fixed-size records:

```
struct Record {
   int id;
   char name[50];
   double value;
};

void readBinaryRecords(const string& filename) {
ifstreaminputFile(filename, ios::binary);

   if (!inputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
      return;
   }
```

```
    Record record;
    while (inputFile.read(reinterpret_cast<char*>(&record),
sizeof(Record))) {
cout<< "ID: " << record.id << ", Name: " << record.name
<< ", Value: " <<record.value<<endl;
        // Process the record...
    }

inputFile.close();
}
```

Sequential File Output

Sequential output involves writing data to a file one item at a time, moving forward through the file.

Writing Text to a File

Basic text writing using the insertion operator:

```
void writeTextFile(const string& filename, const vector<string>&
lines) {
ofstreamoutputFile(filename);

    if (!outputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
        return;
    }

    for (const string& line : lines) {
outputFile<< line <<endl;
    }

outputFile.close();
}
```

Writing Formatted Data

Writing structured data with proper formatting:

```
void writeStudentRecords(const string& filename, const
vector<Student>& students) {
ofstreamoutputFile(filename);

    if (!outputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
```

```
        return;
    }

    // Set precision for floating-point output
outputFile<< fixed <<setprecision(2);

    for (const Student& student : students) {
outputFile<< student.id << " "
<< student.name << " "
<<student.gpa<<endl;
    }

outputFile.close();
}
```

Writing Binary Data

Writing fixed-size records to a binary file:

```
void writeBinaryRecords(const string& filename, const vector<Record>& records) {
ofstreamoutputFile(filename, ios::binary);

    if (!outputFile.is_open()) {
cerr<< "Error opening file: " << filename <<endl;
        return;
    }
    for (const Record& record : records) {
outputFile.write(reinterpret_cast<const char*>(&record), sizeof(Record));
    }

outputFile.close();
}
```

Combining Sequential Input and Output

Many applications need to read from one file and write to another:

```
void convertFileFormat(const string&inputFilename, const string&outputFilename) {
ifstreaminputFile(inputFilename);
ofstreamoutputFile(outputFilename);
```

```
   if (!inputFile.is_open()) {
cerr<< "Error opening input file: " <<inputFilename<<endl;
      return;
   }

   if (!outputFile.is_open()) {
cerr<< "Error opening output file: " <<outputFilename<<endl;
inputFile.close();
      return;
   }

   string line;
   while (getline(inputFile, line)) {
      // Transform or process the line as needed
outputFile<< line <<endl;
   }

inputFile.close();
outputFile.close();
}
```

Sequential Access Patterns

Scan-and-Process

The most basic pattern reads each record, processes it, and continues:

```
void scanAndProcess(const string& filename) {
ifstreaminputFile(filename);

   if (!inputFile.is_open()) {
cerr<< "Error opening file" <<endl;
      return;
   }
```

5.8 Random Access File

Data read/write files (also called random access files) are files which allow the programs to read from or write data at any position of a file without needing to sequentially read all the previous data x. This functionality has important use-cases in databases, game save systems and any scenario where you need to obtain direct access to specific sections of data.

Understanding Random Access

Unlike sequential access where files are processed from beginning to end, random access allows:

- Also Jumping to certain locations in a file
- Accessing data on these positions (reading or writing)
- To navigate forwards and backward through the file
- Updating individual records without rewriting the entire file

Random access is especially suited for working with binary files that contain fixed-size records, although it can work with text files with some constraints.

Prerequisites for Random Access

To effectively use random access, several conditions should be met:

1. The file should be opened in a mode that supports both reading and writing
2. The structure of the file should be known, particularly record sizes or offsets
3. The file pointer position must be precisely controlled

Opening Files for Random Access

Files intended for random access should be opened with the appropriate modes:

```
#include <fstream>
using namespace std;

// Open for both reading and writing in binary mode
fstreamdataFile("database.bin", ios::in | ios::out | ios::binary);
// If the file needs to be created if it doesn't exist
fstreamdataFile("database.bin", ios::in | ios::out | ios::binary | ios::trunc);
```

The binary mode is particularly important for random access because it ensures:

- No translation of newline characters
- Consistent byte counts
- Predictable file pointer positions

Positioning the File Pointer

Random access relies on precise control of the file pointer position using seekg() and seekp():

```
// Move to the 100th byte
dataFile.seekg(100);
```

```
// Move 50 bytes forward from the current position
dataFile.seekg(50, ios::cur);

// Move 200 bytes backward from the end of the file
dataFile.seekg(-200, ios::end);
```

Random Access with Fixed-Size Records

The most common application of random access is with fixed-size records:

```
struct Employee {
   int id;          // 4 bytes
   char name[50];    // 50 bytes
   double salary;    // 8 bytes
};  // Total: 62 bytes per record

void updateEmployeeSalary(fstream& file, int employeeId, double newSalary) {
   // Determine record size
   const int recordSize = sizeof(Employee);
   Employee emp;

   // Start from the beginning
file.seekg(0, ios::beg);

   while (file.read(reinterpret_cast<char*>(&emp), recordSize)) {
      if (emp.id == employeeId) {
         // Found the employee record
emp.salary = newSalary;

         // Move back to the beginning of this record
file.seekp(-recordSize, ios::cur);

         // Write the updated record
file.write(reinterpret_cast<const char*>(&emp), recordSize);

         return;
      }
   }
```

```
    // Employee not found
cout<< "Employee ID " <<employeeId<< " not found." <<endl;
}
```

Direct Access Using Record Index

When records have a fixed size, you can calculate the exact position of any record and jump directly to it:

```
void getEmployeeById(fstream& file, int recordIndex) {
    const int recordSize = sizeof(Employee);
    Employee emp;

    // Calculate the position
streampos position = recordIndex * recordSize;

    // Move to the calculated position
file.seekg(position);

    // Read the record
    if (file.read(reinterpret_cast<char*>(&emp), recordSize)) {
cout<< "Employee #" <<recordIndex<< ": "
<< emp.id << ", " << emp.name << ", $" <<emp.salary<<endl;
    } else {
cout<< "Record #" <<recordIndex<< " not found." <<endl;
    }
}
```

Implementing a Simple Random Access Database

Here's a complete example of a simple employee database using random access:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
using namespace std;

struct Employee {
    int id;
    char name[50];
    double salary;
};
```

```
class EmployeeDB {
private:
fstream file;
   const string filename;

public:
EmployeeDB(const string&fname) : filename(fname) {
      // Open file for reading and writing in binary mode
file.open(filename, ios::in | ios::out | ios::binary);

      if (!file) {
         // File doesn't exist, create it
file.clear();
file.open(filename, ios::out | ios::binary);
file.close();
file.open(filename, ios::in | ios::out | ios::binary);
      }
   }

   ~EmployeeDB() {
      if (file.is_open()) {
file.close();
      }
   }

   bool addEmployee(const Employee& emp) {
      // Move to the end to append
file.seekp(0, ios::end);
file.write(reinterpret_cast<const char*>(&emp), sizeof(Employee));
      return file.good();
   }

   bool getEmployee(int id, Employee& emp) {
file.seekg(0, ios::beg);

      while                    (file.read(reinterpret_cast<char*>(&emp),
sizeof(Employee))) {
```

```
            if (emp.id == id) {
                return true;
            }
        }
        return false;
    }

    bool updateEmployee(const Employee& emp) {
file.seekg(0, ios::beg);
        Employee temp;

        while (file.read(reinterpret_cast<char*>(&temp),
sizeof(Employee))) {
            if (temp.id == emp.id) {
                // Move back to the beginning of this record
file.seekp(-static_cast<int>(sizeof(Employee)), ios::cur);
file.write(reinterpret_cast<const char*>(&emp), sizeof(Employee));
                return file.good();
            }
        }
        return false; // Employee not found
    }

    bool deleteEmployee(int id) {
        // Note: This is a logical delete by marking ID as negative
file.seekg(0, ios::beg);
        Employee emp;

        while (file.read(reinterpret_cast<char*>(&emp),
sizeof(Employee))) {
            if (emp.id == id) {
                emp.id = -emp.id; // Mark as deleted
file.seekp(-static_cast<int>(sizeof(Employee)), ios::cur);
file.write(reinterpret_cast<const char*>(&emp), sizeof(Employee));
                return file.good();
            }
        }
        return false; // Employee not found
```

```
    }

    void displayAll() {
file.clear();
file.seekg(0, ios::beg);
        Employee emp;

cout<< "ID\tName\t\t\tSalary" <<endl;
cout<< "---------------------------------------" <<endl;

        while                    (file.read(reinterpret_cast<char*>(&emp),
sizeof(Employee))) {
            if (emp.id > 0) { // Skip deleted records
cout << emp.id << "\t"
<< emp.name << "\t\t"
<<emp.salary<<endl;
            }
        }
    }
};
```

Random Access with Variable-Length Records

Working with variable-length records is more complex but possible using an index:

```
struct IndexEntry {
    int id;          // Record identifier
    long position;    // Position in the data file
    int length;       // Length of the record
};

// Store variable-length records with an index file
void addRecord(fstream&dataFile, fstream&indexFile, int id, const
string& data) {
IndexEntry entry;
    entry.id = id;

    // Position at the end of data file
dataFile.seekp(0, ios::end);
entry.position = dataFile.tellp();
```

```
   // Write the variable-length data
dataFile<< data;
entry.length = data.length();

   // Write the index entry
indexFile.seekp(0, ios::end);
indexFile.write(reinterpret_cast<const                    char*>(&entry),
sizeof(IndexEntry));
}

// Retrieve a record using the index
string getRecord(fstream&dataFile, fstream&indexFile, int id) {
IndexEntry entry;

   // Search for the record in the index
indexFile.seekg(0, ios::beg);
   while              (indexFile.read(reinterpret_cast<char*>(&entry),
sizeof(IndexEntry))) {
      if (entry.id == id) {
         // Found the index entry, now retrieve the data
dataFile.seekg(entry.position);

         // Read the variable-length data
         char* buffer = new char[entry.length + 1];
dataFile.read(buffer, entry.length);
         buffer[entry.length] = '\0';

         string result = buffer;
         delete[] buffer;
         return result;
      }
   }

   return ""; // Record not found
}
```

Random Access in Text Files

Random access in text files is challenging due to variable line lengths, but can be implemented using line positions:

```
vector<streampos>buildLineIndex(ifstream&textFile) {
   vector<streampos>linePositions;
textFile.seekg(0, ios::beg);
linePositions.push_back(textFile.tellg());

   string line;
   while (getline(textFile, line)) {
linePositions.push_back(textFile.tellg());
   }

   return linePositions;
}

string getLine(ifstream&textFile, const vector<streampos>&linePositions, int lineNumber) {
   if (lineNumber>= 0 &&lineNumber<linePositions.size()) {
textFile.seekg(linePositions[lineNumber]);

      string line;
getline(textFile, line);
      return line;
   }

   return ""; // Invalid line number
}
```

Random access provides powerful capabilities for efficient file manipulation, especially in applications requiring direct access to specific portions of data without processing the entire file.

5.9 Error Handling During File Operations

File operations can fail for numerous reasons: files may not exist, permissions might be insufficient, disks could be full, or hardware errors might occur. Robust error handling is essential for creating reliable file I/O code.

Understanding I/O States and Error Flags

C++ file streams maintain several state flags that indicate the success or failure of operations:

- **good()**: True if no errors have occurred and the file is ready for I/O
- **eof()**: True if the end-of-file has been reached
- **fail()**: True if a formatting or extraction error has occurred
- **bad()**: True if a serious I/O error has occurred, such as disk failure

The ! operator on a stream returns true if either fail() or bad() is true, providing a quick way to check for errors:

```
ifstreaminputFile("data.txt");
if (!inputFile) {
cerr<< "Error opening file!" <<endl;
}
```

Basic Error Checking for File Operations

Checking if a File Was Successfully Opened

Always verify that files are opened successfully before attempting operations:

```
ifstreaminputFile("data.txt");
if (!inputFile.is_open()) {
cerr<< "Failed to open data.txt" <<endl;
   // Handle the error, perhaps by returning an error code
   return -1;
}
```

Checking I/O Operations

Verify the success of read and write operations:

```
int value;
inputFile>> value;
if (inputFile.fail()) {
cerr<< "Failed to read integer from file" <<endl;
   // Handle the error
}

outputFile<< data;
if (!outputFile) {
cerr<< "Failed to write data to file" <<endl;
   // Handle the error
}
```

Comprehensive Error Handling

A more detailed approach uses specific state checking methods:

```
void processFile(const string& filename) {
ifstream file(filename);

   // Check if file opened successfully
   if (!file.is_open()) {
cerr<< "Error: Could not open file " << filename <<endl;
      return;
   }

   string line;
   while (true) {
getline(file, line);

      if (file.eof()) {
         // Normal end of file reached
         break;
      }

      if (file.fail()) {
         // A recoverable error occurred
cerr<< "Warning: Failed to read a line. Clearing error state." <<endl;
file.clear(); // Clear error flags
file.ignore(numeric_limits<streamsize>::max(), '\n'); // Skip bad line
         continue;
      }
      if (file.bad()) {
         // A non-recoverable error occurred
cerr<< "Error: I/O error while reading file." <<endl;
         break;
      }

      // Process the line...
cout<< "Read: " << line <<endl;
   }

file.close();
```

```
}
```

Using Exceptions for Error Handling

C++ streams can be configured to throw exceptions when errors occur:

```
#include <iostream>
#include <fstream>
#include <stdexcept>
using namespace std;

void processFileWithExceptions(const string& filename) {
ifstream file;

   // Configure the file stream to throw exceptions
file.exceptions(ifstream::failbit | ifstream::badbit);

   try {
file.open(filename);

      string line;
      while (getline(file, line)) {
         // Process line...
cout<< line <<endl;
      }

file.close();
   }
   catch (const ifstream::failure& e) {
cerr<< "Error processing file: " <<e.what() <<endl;
      if (file.is_open()) {
file.close();
      }
   }
}
```

Error Recovery Strategies

Different types of errors require different recovery approaches:

Recovering from Formatting Errors

```
void readNumbers(const string& filename) {
```

```
ifstream file(filename);
   if (!file.is_open()) {
cerr<< "Error opening file" <<endl;
      return;
   }

   int number;
   while (file >> number) {
      // Process valid number
cout<< "Number: " << number <<endl;
   }

   if (file.fail() && !file.eof()) {
      // Failed to read a number but not at EOF
file.clear(); // Clear error flags

      string invalidInput;
      file >>invalidInput;
cerr<< "Invalid input: " <<invalidInput<<endl;

      // Continue reading after skipping the invalid input
file.clear();
      while (file >> number) {
cout<< "Number: " << number <<endl;
      }
   }

file.close();
}
```

Handling File Access Errors

```
bool saveData(const string& filename, const vector<int>& data) {
ofstream file;

   // Try primary location
file.open(filename);
   if (!file) {
cerr<< "Warning: Could not open " << filename <<endl;
```

```
        // Try backup location
        string backupFilename = "backup_" + filename;
file.clear();
file.open(backupFilename);

        if (!file) {
cerr<< "Error: Could not open backup location" <<endl;
            return false;
        }

cerr<< "Using backup location: " <<backupFilename<<endl;
    }

    // Write data
    for (int value : data) {
        file << value <<endl;
        if (!file) {
cerr<< "Error writing to file" <<endl;
file.close();
            return false;
        }
    }

file.close();
    return true;
}
```

Advanced Error Handling Techniques

Using errno for System-Level Error Information

For more detailed error information, you can use the C-style errno and related functions:

```
#include <cerrno>
#include <cstring>

void detailedErrorReport(const string& filename) {
ifstream file(filename);
    if (!file) {
cerr<< "Error opening file: " << filename <<endl;
```

```
cerr << "System error: " << strerror(errno) << " (errno: " << errno <<
")" << endl;
}
}
```

Creating a File I/O Error Class

For more sophisticated applications, creating a dedicated error class can help:

```
class FileError : public runtime_error {
private:
   string filename;

public:
FileError(const string& msg, const string&fname)
      : runtime_error(msg), filename(fname) {}

   const string&getFilename() const {
      return filename;
   }
};

void safeReadFile(const string& filename) {
ifstream file(filename);

   if (!file.is_open()) {
      throw FileError("Cannot open file", filename);
   }

   // File operations...

   if (file.bad()) {
      throw FileError("I/O error during read", filename);
   }
}

// Usage
try {
safeReadFile("important_data.txt");
}
```

```
catch (const FileError& e) {
cerr<< "Error: " <<e.what() << " - File: " <<e.getFilename() <<endl;

   // Log the error
logError(e.what(), e.getFilename());

   // Attempt recovery
   if (attemptRecovery(e.getFilename())) {
cerr<< "Recovery successful" <<endl;
   }
}
```

Best Practices for File Error Handling

1. **Always check if files are opened successfully** before performing operations
2. **Check the success of each I/O operation**, especially in critical applications
3. **Provide meaningful error messages** that help diagnose problems
4. **Implement appropriate recovery strategies** based on the type of error
5. **Close files properly**, even when errors occur
6. **Use exception handling** for systematic error management in larger applications
7. **Log detailed error information** to aid debugging
8. **Consider using backup mechanisms** for important data
9. **Test error scenarios** to ensure your code handles them correctly
10. **Use RAII (Resource Acquisition Is Initialization) principles** to ensure proper resource cleanup

Robust error handling is a key aspect of reliable file I/O code. By implementing these techniques, you can create applications that gracefully handle the many ways file operations can fail.

MCQs:

1. **Which header file is required for file handling in C++?**
   a) <iostream>
   b) <fstream>
   c) <stdio.h>
   d) <string>

2. **Which class is used for reading from a file in C++?**
   a) ofstream
   b) ifstream
   c) fstream
   d) file
3. **Which class is used for both reading and writing to a file?**
   a) ifstream
   b) ofstream
   c) fstream
   d) streambuf
4. **What is the purpose of the seekg() function in file handling?**
   a) Move the get (input) pointer
   b) Move the put (output) pointer
   c) Read the file
   d) Write to the file
5. **Which function is used to check if a file has reached the End-of-File (EOF)?**
   a) eof()
   b) close()
   c) seekg()
   d) fail()
6. **Which of the following is NOT a valid file mode in C++?**
   a) ios::in
   b) ios::out
   c) ios::print
   d) ios::app
7. **Which function is used to write data into a file?**
   a) write()
   b) insert()
   c) append()
   d) print()
8. **What is the default mode when opening a file using ofstream?**
   a) ios::app
   b) ios::out
   c) ios::binary
   d) ios::trunc

9. **What does the tellg() function do?**
   a) Returns the current position of the input (get) pointer
   b) Moves the file pointer to the beginning
   c) Reads data from a file
   d) Writes data to a file
10. **Which file mode allows both input and output operations?**
   a) ios::out
   b) ios::in | ios::out
   c) ios::trunc
   d) ios::binary

Short Questions:

1. What are stream classes in C++?
2. What is the difference between formatted and unformatted I/O operations?
3. Explain the purpose of manipulators in C++.
4. How do you open and close a file in C++?
5. What is the function of eof() in file handling?
6. Define file modes and explain their usage.
7. What is the difference between ifstream, ofstream, and fstream?
8. Explain the use of file pointers (seekg() and seekp()) in file operations.
9. What is the difference between sequential access and random access file handling?
10. How do you handle errors in file operations in C++?

Long Questions:

1. Explain stream classes in C++ with examples.
2. What is the difference between formatted and unformatted I/O operations? Provide examples.
3. Discuss different file handling classes (ifstream, ofstream, fstream) with examples.
4. Explain file modes and how they affect file operations.
5. Write a C++ program to open, write, and read a file using file handling.
6. How does the End-of-File (EOF) condition work? Explain with an example.
7. Explain sequential file access and randomaccess file operations with examples.

8. What is the role of file pointers (seekg(), seekp(), tellg(), tellp()) in file handling?
9. Write a C++ program to demonstrate error handling during file operations.
10. Discuss the importance of file handling in real-world applications.

References

Object-Oriented Concepts (Chapter 1)

1. Stroustrup, B. (2013). The C++ Programming Language (4th ed.). Addison-Wesley Professional.
2. Lippman, S. B., Lajoie, J., & Moo, B. E. (2012). C++ Primer (5th ed.). Addison-Wesley Professional.
3. Prata, S. (2011). C++ Primer Plus (6th ed.). Addison-Wesley Professional.
4. Lafore, R. (2001). Object-Oriented Programming in C++ (4th ed.). Sams Publishing.
5. Deitel, P., & Deitel, H. (2016). C++ How to Program (10th ed.). Pearson.

Functions, Constructors, and Destructors (Chapter 2)

1. Meyers, S. (2005). Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd ed.). Addison-Wesley Professional.
2. Schildt, H. (2003). C++: The Complete Reference (4th ed.). McGraw-Hill Education.
3. Eckel, B. (2000). Thinking in C++ (2nd ed.). Prentice Hall.
4. Josuttis, N. M. (2012). The C++ Standard Library: A Tutorial and Reference (2nd ed.). Addison-Wesley Professional.
5. McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction (2nd ed.). Microsoft Press.

Operator Overloading and Inheritance (Chapter 3)

1. Vandevoorde, D., & Josuttis, N. M. (2017). C++ Templates: The Complete Guide (2nd ed.). Addison-Wesley Professional.
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
3. Coplien, J. O. (1991). Advanced C++ Programming Styles and Idioms. Addison-Wesley Professional.
4. Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.
5. Alexandrescu, A. (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley **Professional.**

Pointer, Virtual Function, and Polymorphism (Chapter 4)

1. Meyers, S. (2014). Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media.

2. Booch, G. (2007). Object-Oriented Analysis and Design with Applications (3rd ed.). Addison-Wesley Professional.
3. Lister, A. M., & Eager, R. D. (1988). Fundamentals of Operating Systems. Springer.
4. Stroustrup, B. (2014). Programming: Principles and Practice Using C++ (2nd ed.). Addison-Wesley Professional.
5. Koenig, A., & Moo, B. E. (2000). Accelerated C++: Practical Programming by Example. Addison-Wesley Professional.

Console I/O Operations and File Handling (Chapter 5)

1. Musser, D. R., Derge, G. J., & Saini, A. (2009). STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library (3rd ed.). Addison-Wesley Professional.
2. Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment (3rd ed.). Addison-Wesley Professional.
3. Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.
4. Austern, M. H. (1998). Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Professional.
5. Stepanov, A., & Rose, D. (2015). From Mathematics to Generic Programming. Addison-Wesley Professional.

MATS UNIVERSITY

MATS CENTER FOR OPEN & DISTANCE EDUCATION

UNIVERSITY CAMPUS : Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 M : 9109951184, 9755199381 Toll Free : 1800 123 819999

eMail : admissions@matsuniversity.ac.in Website : www.matsodl.com