# MQF 633 C++ FOR FINANCIAL ENGINEERING

# Lecture 4: C++ Object Oriented Programming

## Part I: Class and Object Revisit

### What is class

In C++, a class is a user-defined data type that represents a blueprint for creating objects. An object is an instance of a class.

### What is an object

An object is an instance of a class. It is created based on the blueprint provided by the class.

Class Car:

{

// something

}

int main() {

// Create objects of the Car class

Car myCar1;

Car myCar2;

return 0;

}

Definition provides the actual implementation of the function.

**Function Call**

int add(int a, int b) {

return a + b;

}

Invoking a function involves providing arguments

Some more example on Object of a Class, car class.cpp

#include <iostream>

using namespace std;

// Define a class

class Car {

public:

// Member variables

string brand;

int year;

// Member function (method)

void displayInfo() {

cout << "Brand: " << brand << ", Year: " << year << endl;

}

};

int main() {

// Create an object of the class Car

Car myCar;

// Access and modify member variables

myCar.brand = "Toyota";

myCar.year = 2022;

// Call a member function

myCar.displayInfo();

return 0;

}

### Why OOP?

Using C++ classes provides several benefits, and it's a fundamental aspect of object-oriented programming (OOP). Here are some reasons why you might want to use C++ classes:

1. **Encapsulation**:

Data Hiding: Classes allow you to encapsulate data within an object, restricting direct access to internal details. This concept is known as data hiding, and it helps in creating a clear separation between the interface and implementation details.

1. **Modularity**:

Code Organization: Classes help organize code into logical units. This modular approach makes it easier to manage and maintain large codebases.

1. **Reusability**:

Once a class is defined, it can be reused in different parts of your program or even in other projects. This promotes code reuse and reduces redundancy.

1. **Abstraction**:

Abstract Data Types: Classes allow you to create abstract data types, where you focus on what an object does rather than how it achieves it. This abstraction simplifies the complexity of the program.

1. **Inheritance**:

Code Reuse: Inheritance allows you to create a new class by inheriting the properties and behaviors of an existing class. This promotes code reuse and helps in building relationships between different classes.

1. **Polymorphism**:

Flexibility: Polymorphism allows objects of different types to be treated as objects of a common base type. This flexibility simplifies code and enables you to work with objects at a higher level of abstraction.

1. **Encapsulation of Complexity:**

Complex Systems: In larger projects, classes help encapsulate the complexity of the system. Each class can represent a specific component or subsystem, making the overall system more manageable.

1. **Code Readability and Maintenance:**

Clear Structure: Classes provide a clear and structured way to represent entities in your program. This clarity enhances code readability and makes it easier for other developers (or yourself) to understand and maintain the code.

1. **Constructor and Destructor:**

Initialization and Cleanup: Classes support the use of constructors for initializing objects and destructors for cleanup. This ensures proper resource management and avoids memory leaks.

1. **Operator Overloading:**

Custom Behavior: C++ allows you to overload operators within classes. This feature enables you to define custom behaviors for operators when applied to objects of your class.

1. **Templates:**

Generic Programming: C++ supports the use of templates, allowing you to write generic classes and functions. This promotes generic programming and code that is independent of specific data types.

In summary, using C++ classes enhances code organization, promotes modularity, and provides a powerful mechanism for creating reusable and maintainable code. It is a key feature of the object-oriented programming paradigm, allowing you to model real-world entities in a structured and efficient way.

### Constructors and Destructors

Definition:

* Constructors are **special** member functions used for **initializing** objects.
* Destructors are **special** member functions to **clean up resources** when an object is destroyed.

Here is an example, student class.cpp

class Student {

public:

// Constructor

Student(string name, int age) : studentName(name), studentAge(age) {

cout << "Student object created" << endl;

}

// Destructor

~Student() {

cout << "Student object destroyed" << endl;

}

void displayInfo() {

cout << "Name: " << studentName << ", Age: " << studentAge << endl;

}

private:

string studentName;

int studentAge;

};

int main() {

// Create an object with constructor

Student myStudent("John", 20);

// Call a member function

myStudent.displayInfo();

// Destructor will be called when the object goes out of scope

return 0;

}

initializer list

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon. Remember the order matters, need to follow the order of member fields being declared.

for most of cases, it is matter of style of construction, however, ...

* Why sometimes we must use initializer list?

1. Efficiency: default state and exact state of class member
2. if class member is reference, has to use initializer list
3. if class member is none-static const, has to use initializer list
4. class member which has no default constructor

Refer to the example: initializer list.cpp

## Class Encapsulation

Encapsulation is the bundling of data and methods that operate on the data within a single unit (class). Here is an example

class Circle {

private:

double radius;

public:

// Setter method

void setRadius(double r) {

if (r > 0) {

radius = r;

} else {

cout << "Invalid radius" << endl;

}

}

// Getter method

double getArea() {

return 3.14 \* radius \* radius;

}

};

int main() {

Circle myCircle;

myCircle.setRadius(5.0);

cout << "Area of the circle: " << myCircle.getArea() << endl;

return 0;

}

### Class Inheritance

Inheritance allows a class to inherit properties and behaviors from another class. Here is a simple example, refer to inheritance.cpp

class Animal {

public:

void eat() {

cout << "Animal is eating" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog is barking" << endl;

}

};

int main() {

Dog myDog;

myDog.eat(); // Inherited from Animal class

myDog.bark();

return 0;

}

#### Access Control for inheritance

In C++, access control specifies the visibility of class members (attributes and methods) from outside the class. There are three access control specifiers: public, private, and protected. These specifiers determine how the members of a class can be accessed by other classes or functions.

Here's a breakdown of the access control rules:

* Public Access:

Public members are accessible from anywhere (inside and outside the class).

**Public inheritance retains public access for inherited members.**

* Private Access:

Private members are only accessible within the class that declares them.

**They are not accessible in derived classes.**

* Protected Access:

**Protected members are accessible within the class that declares them and in derived classes.**

**Protected members are not directly accessible from outside the class.**

Let see an example:

#include <iostream>

// Base class with public, private, and protected members

class Base {

public:

int publicVar; // Public member

void publicMethod() {

std::cout << "Public method in Base class" << std::endl;

}

private:

int privateVar; // Private member

void privateMethod() {

std::cout << "Private method in Base class" << std::endl;

}

protected:

int protectedVar; // Protected member

void protectedMethod() {

std::cout << "Protected method in Base class" << std::endl;

}

};

// Derived class

class Derived : public Base {

public:

void accessBaseMembers() {

// Accessing public members of the Base class

publicVar = 10;

publicMethod();

// Accessing protected members of the Base class

protectedVar = 20;

protectedMethod();

// Cannot access private members of the Base class

// privateVar = 30; // This would result in a compilation error

// privateMethod(); // This would result in a compilation error

}

};

int main() {

// Accessing public members of the Base class

Base baseObj;

baseObj.publicVar = 5;

baseObj.publicMethod();

// Cannot access private and protected members from outside the class

// baseObj.privateVar = 15; // This would result in a compilation error

// baseObj.privateMethod(); // This would result in a compilation error

// baseObj.protectedVar = 25; // This would result in a compilation error

// baseObj.protectedMethod(); // This would result in a compilation error

// Accessing members through a derived class

Derived derivedObj;

derivedObj.accessBaseMembers();

return 0;

}

## Interface & Abstract class

Abstraction involves hiding the complex implementation details and **exposing only the necessary features to the outside world**. Here's an extended example demonstrating abstraction through a simple shape hierarchy. Refer to abstract.cpp

#include <iostream>

#include <cmath>

// Abstract base class

class Shape {

public:

// Pure virtual function makes this class abstract

virtual double area() const = 0; // To be implemented by derived classes

virtual void display() const = 0; // To be implemented by derived classes

// Virtual destructor to ensure proper cleanup in polymorphic scenarios

virtual ~Shape() {}

};

// Derived class: Circle

class Circle : public Shape {

private:

double radius;

public:

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

// Implementation of virtual function for area calculation

double area() const override {

return 3.14 \* radius \* radius;

}

// Implementation of virtual function for displaying information

void display() const override {

std::cout << "Circle - Radius: " << radius << ", Area: " << area() << std::endl;

}

};

// Derived class: Rectangle

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

// Implementation of virtual function for area calculation

double area() const override {

return length \* width;

}

// Implementation of virtual function for displaying information

void display() const override {

std::cout << "Rectangle - Length: " << length << ", Width: " << width << ", Area: " << area() << std::endl;

}

};

int main() {

// Creating objects of derived classes

Circle myCircle(5.0);

Rectangle myRectangle(4.0, 6.0);

// Using the abstraction to calculate and display areas

myCircle.display();

myRectangle.display();

return 0;

}

**Virtual keyword**

In C++, the virtual keyword is used to declare a member function in a base class as "virtual." This allows the function to be overridden by a function with the same signature in a derived class. The primary use of the virtual keyword is in the context of polymorphism, enabling dynamic dispatch of member functions.

1. Virtual Functions:

* A virtual function is declared in the base class using the virtual keyword.
* The virtual function in the base class provides a common interface that can be overridden by derived classes.
* The virtual function can have a default implementation in the base class, and derived classes can choose to override it.

1. Overriding Virtual Functions:

* A derived class that wants to provide its own implementation for a virtual function declares the function with the override keyword.
* Overriding ensures that the function in the derived class has the same signature as the virtual function in the base class.

1. Dynamic Dispatch (Polymorphism):

* When a base class pointer or reference is used to call a virtual function on an object, the actual implementation to be executed is determined at runtime.
* This mechanism is known as dynamic dispatch or runtime polymorphism.

1. Pure Virtual Functions:

* A pure virtual function is declared using virtual in the base class and set to 0 (pure virtual function) after the function declaration.
* A class containing at least one pure virtual function becomes an abstract class, and objects of abstract classes cannot be instantiated.

The use of virtual functions and dynamic dispatch is a key feature of polymorphism in C++, allowing you to write code that works with a base class interface while allowing for different implementations in derived classes.

**Interface class**

Class contains virtual function

**Abstract class**

Class contains pure virtual function

**Override keyword**

The override keyword in C++ explicitly specifies that a virtual function in a derived class is intended to replace (or “override”) a virtual function in the base class with the same name.

### Virtual Destructor

In C++, when you have a class hierarchy and you intend to use polymorphism (i.e., you might have pointers or references to base class objects pointing to derived class objects), it's a good practice to declare the base class destructor as virtual. This ensures that the proper destructor for the actual type of the object is called when the object is deleted through a pointer or reference to the base class.

#include <iostream>

// Base class

class Base {

public:

Base() {

std::cout << "Base class constructor" << std::endl;

}

// Virtual destructor

virtual ~Base() {

std::cout << "Base class destructor" << std::endl;

}

};

// Derived class

class Derived : public Base {

public:

Derived() {

std::cout << "Derived class constructor" << std::endl;

}

// Destructor (implicitly virtual due to the virtual destructor in the base class)

~Derived() {

std::cout << "Derived class destructor" << std::endl;

}

};

int main() {

// Creating a derived class object using a base class pointer

Base\* basePtr = new Derived();

// Deleting the object through the base class pointer

delete basePtr;

return 0;

}

## Polymorphism

Polymorphism allows objects of different **derived types to be treated as objects of a common base type**. Here's an example:

class Shape {

public:

virtual void draw() {

cout << "Drawing a shape" << endl;

}

};

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing a circle" << endl;

}

};

class Square : public Shape {

public:

void draw() override {

cout << "Drawing a square" << endl;

}

};

int main() {

Shape\* shape1 = new Circle();

Shape\* shape2 = new Square();

shape1->draw(); // Calls draw() from Circle

shape2->draw(); // Calls draw() from Square

delete shape1;

delete shape2;

return 0;

}

In this example, the printString function takes a const reference to a std::string. This means that the function cannot modify the content of the string. Attempting to modify the string inside the function would result in a compilation error.

Using const references is beneficial when you want to avoid unnecessary copies of large objects (like strings or complex data structures) and ensure that the function does not accidentally modify the input parameters.

### Template class

In C++, templates provide a way to create generic classes and functions. A template allows you to define a generic type or function with placeholders for types or values that can be specified later. This enables you to write code that works with different data types without duplicating the code for each type. Here's an example of a simple template class that represents a generic container:

#include <iostream>

// Template class definition

template <typename T>

class Container {

private:

T data;

public:

Container(T value) : data(value) {}

T getValue() const {

return data;

}

};

int main() {

// Instantiate Container with int

Container<int> intContainer(42);

std::cout << "Integer Value: " << intContainer.getValue() << std::endl;

// Instantiate Container with double

Container<double> doubleContainer(3.14);

std::cout << "Double Value: " << doubleContainer.getValue() << std::endl;

// Instantiate Container with string

Container<std::string> stringContainer("Hello, C++!");

std::cout << "String Value: " << stringContainer.getValue() << std::endl;

return 0;

}

**Syntax**

**template<class T1, class T2, ...**>

class classname

{

...

...

};

Simple example, template class1.cpp

## Another example, template class2.cpp

#include <iostream>

using namespace std;

template <typename T1, typename T2>

class Pair {

private:

T1 first;

T2 second;

public:

Pair(T1 a, T2 b) {

first=a;

second=b;

}

T1 GetFirst() {

return first;

}

T2 GetSecond() {

return second;

}

};

int main() {

Pair<int, double> pair1(10, 3.14);

cout << "First: " << pair1.GetFirst() << ", Second: " << pair1.GetSecond() << endl;

Pair<string, bool> pair2("hello", true);

cout << "First: " << pair2.GetFirst() << ", Second: " << pair2.GetSecond() << endl;

return 0;

}

## Part II: Some Advanced Topics

### Memory Structure

The memory structure of a typical modern personal computer (PC) consists of several components that work together to store and retrieve data. Here's a simplified overview of the memory structure in a model PC:

**Registers:**

Location: Inside the CPU (Central Processing Unit).

Role: Registers are the smallest and fastest type of memory. They store data that the CPU is actively using for processing. The CPU uses registers to store intermediate results and control information.

**Cache Memory:**

Location: Between the CPU and RAM (Random Access Memory).

Role: Cache memory is faster than RAM and is used to store frequently accessed data and instructions. There are typically multiple levels of cache (L1, L2, L3) with varying sizes and speeds.

**RAM (Random Access Memory):**

Location: Connected to the motherboard.

Role: RAM is the main memory used by the CPU to store data that is actively being used or processed. It is volatile memory, meaning its contents are lost when the power is turned off.

**Storage (HDD/SSD):**

Location: Connected to the motherboard.

Role: Hard Disk Drives (HDDs) or Solid State Drives (SSDs) are used for long-term storage of data, applications, and the operating system. Unlike RAM, storage is non-volatile, and data persists even when the power is off.

**Virtual Memory:**

Location: Managed by the operating system (partly on storage).

Role: Virtual memory extends the computer's available memory by using a portion of the storage device as if it were additional RAM. It allows the system to run applications that require more memory than physically available.

**GPU Memory (Graphics Memory):**

Location: On the graphics card (GPU).

Role: Dedicated memory for the GPU to store textures, shaders, and other graphics-related data. It is separate from the main system memory (RAM) and is crucial for graphics rendering.

**BIOS/UEFI Firmware:**

Location: Firmware chip on the motherboard.

Role: Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) provides low-level control over the hardware. It contains the initial instructions that the computer executes during the boot process.

**Memory Controller:**

Location: On the CPU or integrated into the motherboard.

Role: The memory controller manages the flow of data between the CPU and RAM. It plays a crucial role in accessing and transferring data between the CPU and main memory.

This memory structure allows the computer to efficiently manage data at different speeds and capacities, balancing the need for speed and volatile storage (RAM and cache) with the need for persistent storage (HDD/SSD) and specialized memory (GPU memory). Keep in mind that the specifics can vary based on the architecture, type of CPU, and other hardware components used in a particular PC model.

### Memory Leak

A memory leak in C++ occurs when a program allocates memory dynamically using operators like new or malloc but fails to deallocate or free that memory before the program exits. This can lead to a gradual accumulation of unreleased memory over time, which may result in degraded performance or, in extreme cases, program crashes due to insufficient memory.

Here's a simple example illustrating a memory leak:

#include <iostream>

int main() {

// Allocating memory for an integer array

int\* intArray = new int[5];

// Assigning values to the array elements

for (int i = 0; i < 5; ++i) {

intArray[i] = i \* 2;

}

// Program exits without deallocating the memory (memory leak)

return 0;

}

In this example:

* Memory is dynamically allocated for an integer array of size 5 using new int[5].
* Values are assigned to the array elements.
* The program exits without deallocating the memory using delete[] intArray.

To fix the memory leak, you should use delete[] to release the allocated memory before the program exits.

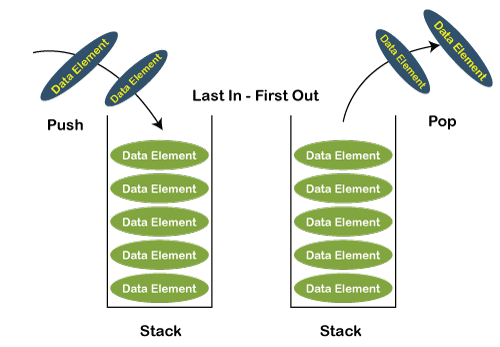
Remember whenever new is used, delete should always be used in hand-in-hand. There is the reason.

### Stack Vs Heap

In C++, memory can be allocated in two primary areas: the stack and the heap. Understanding the differences between these two memory areas is crucial for writing efficient and correct programs.

### Stack allocation

The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated. This all happens using some predefined routines in the compiler. A programmer does not have to worry about memory allocation and de-allocation of stack variables. This kind of memory allocation is also known as Temporary memory allocation because as soon as the method finishes its execution all the data belonging to that method flushes out from the stack automatically. This means any value stored in the stack memory scheme is accessible as long as the method hasn’t completed its execution and is currently in a running state.

****

**Key features:**

* **Memory Management:**

Automatic: Memory on the stack is managed automatically by the compiler. When a function is called, local variables are pushed onto the stack, and when the function exits, the stack space is automatically reclaimed.

* **Scope:**

Local Variables: Typically, variables with a short lifespan, such as local variables within a function, are allocated on the stack.

* **Limited Size:**

The stack has a limited size, and it is generally smaller than the heap. Therefore, large data structures or objects are usually allocated on the heap.

* **Speed:**

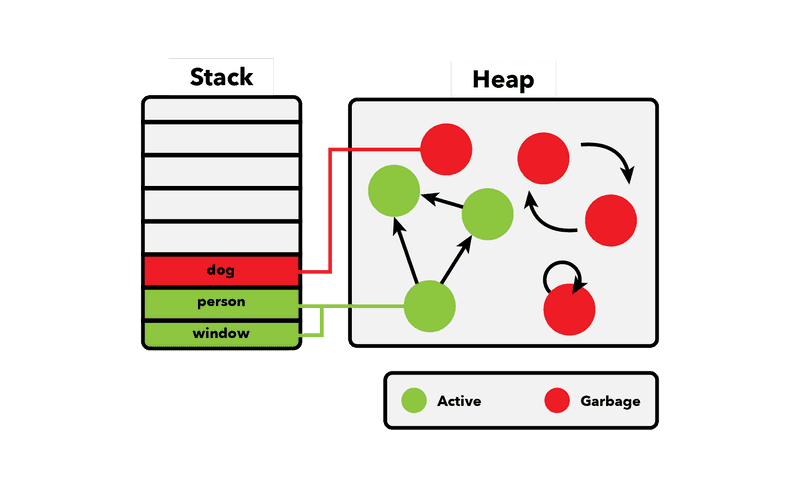
Fast Access: Accessing variables on the stack is faster than dynamic memory allocation on the heap because it involves simple pointer manipulation.

* **Lifetime:**

Short-lived: Memory on the stack is short-lived. It is automatically deallocated when the function or block of code in which the variable is declared exits.

### Heap allocation

The memory is allocated during the execution of instructions written by programmers. Note that the name heap has nothing to do with the heap data structure. It is called a heap because it is a pile of memory space available to programmers to allocate and de-allocate. Every time when we made an object it always creates in Heap-space and the reference information to these objects is always stored in Stack-memory. Heap memory allocation isn’t as safe as Stack memory allocation because the data stored in this space is accessible or visible to all threads. If a programmer does not handle this memory well, a memory leak can happen in the program.

****

* **Memory Management:**

Manual: Memory on the heap is managed manually by the programmer. You need to explicitly allocate and deallocate memory using operators like new and delete or, in modern C++, std::allocator and smart pointers.

* **Scope:**

Dynamic Variables: Objects allocated on the heap have a longer lifetime and persist until explicitly deallocated.

* **Size Limitation:**

Larger Size: The heap has a larger size compared to the stack. It is suitable for allocating large data structures or objects that need to persist across function calls.

* **Speed:**

Slower Access: Accessing variables on the heap is relatively slower than accessing variables on the stack due to the need for explicit memory management.

* **Lifetime:**

Long-lived: Memory on the heap can be long-lived, and it persists until explicitly deallocated by the programmer. Failure to deallocate memory properly can lead to memory leaks.

## Choosing Between Stack and Heap

* Use the stack for small, short-lived variables and data structures.
* Use the heap for large data structures, objects with a longer lifespan, or when the size of the data is not known at compile time.

| Parameter | STACK | HEAP |
| --- | --- | --- |
| Basic | Memory is allocated in a contiguous block. | Memory is allocated in any random order. |
| Allocation and De-allocation | Automatic by compiler instructions. | Manual by the programmer. |
| Cost | Less | More |
| Implementation | Easy | Hard |
| Access time | Faster | Slower |
| Main Issue | Shortage of memory | Memory fragmentation |
| Locality of reference | Excellent | Adequate |
| Safety | Thread safe, data stored can only be accessed by the owner | Not Thread safe, data stored visible to all threads |
| Flexibility | Fixed-size | Resizing is possible |
| Data type structure | Linear | Hierarchical |
| Preferred | Static memory allocation is preferred in an array. | Heap memory allocation is preferred in the linked list. |
| Size | Small than heap memory. | Larger than stack memory. |

// Example demonstrating stack and heap allocation

#include <iostream>

void stackExample() {

int stackVar = 5; // Variable allocated on the stack

// ...

} // stackVar is automatically deallocated when the function exits

void heapExample() {

int\* heapVar = new int; // Variable allocated on the heap

// ...

delete heapVar; // Explicitly deallocate memory to avoid memory leaks

}

int main() {

stackExample();

heapExample();

return 0;

}

In this example:

* The Base class has members with different access specifiers: public Var and public Method are public, private Var and private Method are private, and protected Var and protected Method are protected.
* The Derived class publicly inherits from the Base class. It can access the public and protected members of the Base class.
* In the main function, you can see how members with different access specifiers can be accessed or not accessed from outside the class or through a derived class.

It's important to note that modern C++ encourages the use of smart pointers (like std::unique\_ptr and std::shared\_ptr) and containers (like std::vector and std::string) to manage memory on the heap, reducing the risk of memory leaks and improving code safety.

### Deep copy vs Shallow copy

In C++, when dealing with objects and their copies, it's essential to understand the concepts of deep copy and shallow copy. These terms refer to how the data of an object is duplicated when creating a copy.

* **Shallow Copy:**

Definition: A shallow copy of an object is a copy of the object itself, but not of the data it points to.

Implementation: The default copy constructor or assignment operator performs a shallow copy.

Result: Both the original object and the copied object share the same memory locations for dynamically allocated data. Changes in one object affect the other.

class ShallowCopy {

private:

int\* data;

public:

ShallowCopy(int val) {

data = new int(val);

}

// Shallow copy constructor

ShallowCopy(const ShallowCopy& other) : data(other.data) {}

// Shallow copy assignment operator

ShallowCopy& operator=(const ShallowCopy& other) {

if (this != &other) {

delete data; // Deallocate old data

data = other.data;

}

return \*this;

}

~ShallowCopy() {

delete data;

}

};

* **Deep Copy:**

Definition: A deep copy of an object is a copy of both the object and the data it points to.

Implementation: Requires a custom copy constructor or assignment operator to explicitly allocate new memory and copy the content.

Result: The original and copied objects have independent copies of the data. Changes in one object do not affect the other.

class DeepCopy {

private:

int\* data;

public:

DeepCopy(int val) {

data = new int(val);

}

// Deep copy constructor

DeepCopy(const DeepCopy& other) : data(new int(\*(other.data))) {}

// Deep copy assignment operator

DeepCopy& operator=(const DeepCopy& other) {

if (this != &other) {

delete data; // Deallocate old data

data = new int(\*(other.data));

}

return \*this;

}

~DeepCopy() {

delete data;

}

};

### Operator Overloading

Operator overloading in C++ allows you to define how an operator behaves when applied to objects of a class. This enables you to create more intuitive and expressive code by extending the functionality of operators beyond their default behavior.

Here's an overview of operator overloading with examples:

**Basic Syntax:**

Operator overloading is accomplished by defining a function that specifies the behavior of an operator for objects of a class. The function is called the "overloaded operator function."

return\_type operator+(const MyClass& obj1, const MyClass& obj2) {

// Define how the '+' operator should behave for objects of MyClass

// ...

}

Commonly Overloaded Operators:

* Arithmetic Operators: +, -, \*, /, %
* Comparison Operators: ==, !=, <, >, <=, >=
* Assignment Operators: =, +=, -=, \*=, /=, %=
* Increment/Decrement Operators: ++, --
* Logical Operators: &&, ||, !
* Bitwise Operators: &, |, ^, ~, <<, >>

Let’s see one example

#include <iostream>

class Complex {

private:

double real;

double imaginary;

public:

Complex(double r, double i) : real(r), imaginary(i) {}

// Overloading the '+' operator

Complex operator+(const Complex& other) const {

return Complex(real + other.real, imaginary + other.imaginary);

}

// Display the complex number

void display() const {

std::cout << real << " + " << imaginary << "i" << std::endl;

}

};

int main() {

// Creating two complex numbers

Complex c1(3.0, 4.0);

Complex c2(1.5, 2.5);

// Using the overloaded '+' operator

Complex result = c1 + c2;

// Displaying the result

std::cout << "Result: ";

result.display();

return 0;

}

In this example, the Complex class overloads the '+' operator to add two complex numbers. The operator+ function is defined as a member function of the Complex class, and it returns a new Complex object representing the sum of the two operands.

### Homework assignment

Based on L3 assignment, now we convert struct() into a trade class, and implement the similar code to compute the option pv and output the pv and trade information into a result file.

### Appendix

### Quiz

1. What is a class in C++?

A. A function

B. A data type

C. A program

D. A loop

1. How is an object created in C++?

A. create object;

B. Object = new Object();

C. Object obj;

D. new Object;

1. What does a constructor do in a class?

A. Initializes the class variables

B. Destroys the object

C. Creates a new object

D. Prints output

1. Which access specifier allows members to be accessible only within the same class?

A. public

B. private

C. protected

D. internal

1. What is the purpose of the new keyword in C++?

A. Create a new class

B. Allocate memory for an object

C. Create a constructor

D. Free memory

1. Which operator is used to access members of an object in C++?

A. ->

B. ::

C. =>

D. .

1. What is encapsulation in C++?

A. A loop construct

B. A way to bundle data and methods into a single unit

C. A memory allocation technique

D. A type of class inheritance

1. Inheritance in C++ allows a class to:

A. Inherit only data members

B. Inherit only member functions

C. Inherit properties and behaviours from another class

D. Inherit from multiple classes simultaneously

1. What is a destructor in C++ used for?

A. Create objects

B. Allocate memory

C. Initialize class members

D. Clean up resources and release memory

1. What does the virtual keyword do in C++?

A. Allocate virtual memory

B. Enable virtual functions and polymorphism

C. Define a virtual class

D. Enable virtual access specifier