Friend Function / Class

Local Class, nested Class

Simulating Final Class

Friend function

Inline Functions

Malloc+calloc+realloc

Static data Member and Function

const

---------------------------------------

@friend functions

Friend functions are those functions that have the right to access the private data members of class even though they are not defined inside the class.

It is necessary to write the prototype of the friend function.

One main thing to note here is that if we have written the prototype for the friend function in the class it will not make that function a member of the class

class Complex{

int a, b;

friend Complex sumComplex(Complex o1, Complex o2);

public:

void setNumber(int n1, int n2){

a = n1;

b = n2;

}

// Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)

void printNumber(){

cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;

}

};

Complex sumComplex(Complex o1, Complex o2){

Complex o3;

o3.setNumber((o1.a + o2.a), (o1.b+o2.b))

;

return o3;

}

int main(){

Complex c1, c2, sum;

c1.setNumber(1, 4);

c1.printNumber();

c2.setNumber(5, 8);

c2.printNumber();

sum = sumComplex(c1, c2);

sum.printNumber();

return 0;

}

/\* Properties of friend functions

1. Not in the scope of class

2. since it is not in the scope of the class, it cannot be called from the object of that class. c1.sumComplex() == Invalid

3. Can be invoked without the help of any object

4. Usually contans the objects as arguments

5. Can be declared inside public or private section of the class

6. It cannot access the members directly by their names and need object\_name.member\_name to access any member.

\*/

**Friend Function**

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

A class’s friend function is defined outside that class’s scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend function in C++ is a function that is preceded by the keyword “friend.”

class class\_name {

friend data\_type function\_name(argument);

// syntax of friend function.

};

The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator.

#include <iostream>

using namespace std;

class Rectangle {

private:

int length;

public:

Rectangle() { length = 10; }

friend int printLength(Rectangle); // friend function

};

int printLength(Rectangle b) {

b.length += 10;

return b.length;

}

int main() {

Rectangle b;

cout << "Length of Rectangle: " << printLength(b) << endl;

return 0;

}

Output : Length of Rectangle : 20

**Characteristics of friend function:**

* A friend function can be declared in the private or public section of the class.
* It can be called a normal function without using the object.
* A friend function is not in the scope of the class, of which it is a friend.
* A friend function is not invoked using the class object as it is not in the class’s scope.
* A friend function cannot access the private and protected data members of the class directly. It needs to make use of a class object and then access the members using the dot operator.
* A friend function can be a global function or a member of another class.

Friend Function:

In C++, encapsulation restricts direct access to private and protected members of a class, ensuring data protection. However, in some situations, we may want a non-member function to access the private members of a class, and this is where a friend function comes into play.

A friend function is a function that is not a member of the class but is granted access to its private and protected data. By using the friend keyword, we allow the function to bypass access restrictions while keeping the function itself external to the class.

Example:

Consider a scenario where we have a Box class, and we want a non-member function to access the width of the box:

cpp

Copy code

class Box {

private:

int width;

public:

Box() : width(0) {}

// Declaring a friend function

friend void printWidth(Box box);

};

// Friend function definition

void printWidth(Box box) {

cout << "Width of the box: " << box.width << endl;

}

In this case, the printWidth function is a non-member of the Box class but can still access the private member width.

Friend Class:

A friend class allows another class full access to the private and protected members of the class that declares the friendship. This is particularly useful when two classes are closely related.

For instance, if a Car class needs to access the private data of an Engine class (like horsepower), we can declare Car as a friend of Engine:

cpp

Copy code

class Engine {

private:

int horsepower;

public:

Engine(int hp) : horsepower(hp) {}

// Declaring the Car class as a friend

friend class Car;

};

class Car {

public:

void displayEngineDetails(Engine engine) {

cout << "Car engine horsepower: " << engine.horsepower << endl;

}

};

Here, the Car class can directly access Engine’s private members.

------------------------------

Inline Functions in C++

Definition and Use

Inline functions are functions that the compiler attempts to expand in place rather than calling them through the usual function call mechanism. This can help reduce the overhead associated with function calls, such as pushing arguments onto the call stack and jumping to a different location in memory.

Syntax

inline int add(int a, int b) {

return a + b;

}

How Inline Functions Help

Reduced Function Call Overhead: By expanding the function body at the place where the function is called, inline functions eliminate the overhead associated with a function call. This includes avoiding the overhead of stack operations and jump instructions.

Improved Performance: For small, frequently called functions, using inline can lead to performance improvements by avoiding function call overhead.

Increased Code Size: One potential downside is that extensive use of inline functions can increase the code size, which might negatively impact performance due to instruction cache misses.

-------------------------

malloc

Allocates a specified number of bytes.

Returns a pointer to the allocated memory.

Memory content is not initialized.

int\* ptr = (int\*)malloc(5 \* sizeof(int)); // Allocates memory for 5 integers

calloc

Allocates memory for an array of elements, initializes all bytes to zero.

Returns a pointer to the allocated memory.

int\* ptr = (int\*)calloc(5, sizeof(int)); // Allocates and initializes memory for 5 integers

realloc

Resizes previously allocated memory.

Can move the memory block if needed.

ptr = (int\*)realloc(ptr, 10 \* sizeof(int)); // Resizes the memory to hold 10 integers

--------------------------

**2. Local Class & Nested Class**

**Local Class:**

A **local class** is a class defined within the scope of a function. This means the class is only known within that function and cannot be used outside of it. A typical use case is when the functionality provided by the class is only relevant to the function, and there’s no need to expose it globally or to other parts of the program.

In interviews, I might give this example:

void exampleFunction() {

// Local class inside a function

class Local {

public:

void display() {

cout << "I am a local class!" << endl;

}

};

Local localObject;

localObject.display();

}

Here, the class Local is only accessible inside exampleFunction. This is useful for localizing functionality and keeping the code modular.

**Nested Class:**

A **nested class** is a class defined within another class. The idea here is that the nested class logically belongs to the outer class, and it may even be used to support the outer class’s functionality. It can access the members (including private members) of the enclosing class, depending on the access specifier.

class Outer {

private:

int outerData;

public:

class Nested {

public:

void display() {

cout << "I am a nested class!" << endl;

}

};

void outerFunction() {

Nested nestedObj;

nestedObj.display();

}

};

In this case, Nested is a part of Outer, and is usually used to assist Outer in some way. Nested classes are a good way to group closely related classes and encapsulate complex data structures.

**3. Simulating Final Class**

In C++, there’s no native final keyword to prevent a class from being inherited (like in Java or C#), but we can simulate this behavior by making the class’s constructor and destructor private and not providing a friend function for inheritance.

**Why Simulate a Final Class?**

A **final class** is one that cannot be extended. You might want to do this when:

* The class has a specific functionality that should not be altered.
* Extending the class could lead to errors or misuse.

**How to Simulate a Final Class:**

One way to do this is by making all constructors and destructors private. Then, no other class can inherit from this class, as they cannot call the constructor of the base class.

class FinalClass {

private:

FinalClass() {

cout << "FinalClass Constructor" << endl;

}

~FinalClass() {

cout << "FinalClass Destructor" << endl;

}

public:

// Factory method to create an object

static FinalClass\* createInstance() {

return new FinalClass();

}

};

Here, the class FinalClass cannot be inherited because the constructor and destructor are private, ensuring that no other class can extend it.

------------------------------------------------------------------------------------------------

**Static data Member and Function**

@static data member

In C++, **static data members** and **static member functions** are used when you want to share data or functionality across all instances of a class, rather than having each object maintain its own copy.

**1. Static Data Member:**

* A **static data member** is shared by all instances of the class. It is declared using the static keyword.
* This means that the static variable is **not specific to any object**, and it exists at the class level rather than at the object level.
* Static data members are **initialized outside the class** (because they are not tied to any particular object).

Key points:

* Only **one copy** of a static data member exists, regardless of how many objects of the class are created.
* It can be accessed using the **class name** or **object**.

#include <iostream>

using namespace std;

class Student {

public:

static int studentCount; // static data member

Student() {

studentCount++; // Increment static member

}

static void displayCount() {

cout << "Total students: " << studentCount << endl;

}

};

// Initialize static member outside the class

int Student::studentCount = 0;

int main() {

Student s1, s2, s3;

// Accessing static member through class name

Student::displayCount();

return 0;

}

Output

Total students: 3

**Static Member Function:**

* A **static member function** can only access static data members and other static functions. It does not have access to non-static members because it is called at the class level, not on an object.
* Static member functions are also declared using the static keyword.

Key points:

* It can be called **without creating an object** of the class, i.e., using the class name.
* Static member functions cannot access non-static members directly because they don’t have a this pointer.

#include <iostream>

using namespace std;

class Student {

public:

static int studentCount; // Static data member

Student() {

studentCount++;

}

static void displayCount() { // Static member function

cout << "Student count: " << studentCount << endl;

}

};

int Student::studentCount = 0; // Initialization of static data member

int main() {

Student s1, s2;

// Accessing static function without any object

Student::displayCount();

return 0;

}

Output:

Student count: 2

Const

the const keyword is used to define immutability. It can be applied in various contexts to restrict the modification of variables, objects, member functions, and parameters. Here's a breakdown of how const is used in OOP concepts:

**const Data Members**

* A **const data member** is a member of a class that is constant, meaning its value cannot be modified once initialized.
* It must be initialized **either in the constructor initializer list** or at the point of declaration (C++11 and later).

In this case, id cannot be modified after initialization.

#include <iostream>

using namespace std;

class Student {

public:

const int rollNumber;

Student(int r) : rollNumber(r) {} // Initialize const data member

void showRollNumber() const {

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

}

};

int main() {

Student s(101);

s.showRollNumber();

// s.rollNumber = 102; // Error: cannot modify const data member

return 0;

}

-------------------------------------------------------------------------------------------------------------------