OOPS

https://www.geeksforgeeks.org /encapsulation-in-cpp/

eg: #include <iostream>

#include <string>

class Student {

private:

std::string name;

int age;

public:

// Setter methods

void setName(const std::string& newName) {

name = newName;

}

void setAge(int newAge) {

if (newAge >= 0) {

age = newAge;

}

}

// Getter methods

std::string getName() const {

return name;

}

int getAge() const {

return age;

}

};

int main() {

Student student1;

student1.setName("Alice");

student1.setAge(20);

std::cout << "Student's name: " << student1.getName() << std::endl;

std::cout << "Student's age: " << student1.getAge() << std::endl;

return 0;

}

In this example, the class Student encapsulates the data members name and age as private members. The public methods setName, setAge, getName, and getAge are used to access and modify these private members. This ensures that the data is not directly accessible from outside the class and can only be manipulated through the defined methods. This concept of encapsulation helps in maintaining the integrity of the data and allows you to control how the data is accessed and modified.

<https://www.geeksforgeeks.org/abstraction-in-cpp/>

eg: #include <iostream>

using namespace std;

class Summation {

private:

// private variables

int a, b, c;

public:

void sum(int x, int y)

{

a = x;

b = y;

c = a + b;

cout<<"Sum of the two number is : "<<c<<endl;

}

};

int main()

{

Summation s;

s.sum(5, 4);

return 0;

}

diff bet abs and enc:

<https://www.geeksforgeeks.org/difference-between-abstraction-and-encapsulation-in-c/>

why c++is called semi oops language:

<https://www.geeksforgeeks.org/c-partially-object-oriented-language/>

<https://www.geeksforgeeks.org/can-a-c-class-have-an-object-of-self-type/>

polymorphism:

<https://www.geeksforgeeks.org/cpp-polymorphism/>

Inheritance:

<https://www.geeksforgeeks.org/inheritance-in-c/>

**Hiding of all Overloaded Methods with Same Name in Base Class in C++**

<https://www.geeksforgeeks.org/hiding-of-all-overloaded-methods-with-same-name-in-base-class-in-cpp/>

# What is operator overloading?

# Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++.

# Overloaded operators are functions with special names: the keyword "operator"

## **Rules for Operator Overloading**

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
* When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

U can use this add 2 objects.

## **Encapsulation**

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

How can we restrict dynamic allocation of objects of a class using new?

By making an empty private new and new[] operators

#include <stdlib.h>

#include <stdio.h>

#include <iostream>

using namespace std;

class Test {

private:

void\* operator new(size\_t size) {}

void\* operator new[](size\_t size) {}

};

int main()

{

Test \*obj = new Test;

Test \*arr = new Test[10];

return 0;

}

Which of the following operators are overloaded by default by the compiler in every user defined classes even if user has not written?

1) Comparison Operator ( == )

2) Assignment Operator ( = )

Assign operator is by default available in all user defined classes even if user has not implemented. The default assignement does shallow copy. But comparison operator "==" is not overloaded.

#include<iostream>

using namespace std;

class Complex {

private:

int real, imag;

public:

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

};

int main()

{

Complex c1(10, 5), c2(2, 4);

// For example, below code works fine

c1 = c2;

// But this code throws compiler error

if (c1 == c2)

cout << "Same";

return 0;

}

In C++, when you overload both the pre-increment (**++obj**) and post-increment (**obj++**) operators for a class, you can differentiate between them based on their function signatures. The key difference is the presence of an additional dummy parameter in the post-increment operator. Specifically:

1. **Pre-increment Operator (++obj):**
   * It is overloaded as a member function without any parameters.
   * Its signature is **T& operator++();**, where **T** is the class type.
2. **Post-increment Operator (obj++):**
   * It is overloaded as a member function with a dummy parameter of type **int**.
   * Its signature is **T operator++(int);**, where **T** is the class type.

#include <iostream>

class Counter {

private:

int count;

public:

Counter(int initialCount) : count(initialCount) {}

// Pre-increment operator

Counter& operator++() {

++count;

return \*this;

}

// Post-increment operator

Counter operator++(int) {

Counter temp(\*this); // Create a copy of the current object

++count;

return temp; // Return the copy (the original value before increment)

}

int getCount() const {

return count;

}

};

int main() {

Counter c1(5);

// Pre-increment

++c1;

std::cout << "After pre-increment: " << c1.getCount() << std::endl;

// Post-increment

Counter c2 = c1++;

std::cout << "After post-increment: " << c1.getCount() << std::endl;

std::cout << "Value before post-increment: " << c2.getCount() << std::endl;

return 0;

}

In this example, the **Counter** class overloads both pre-increment and post-increment operators. The **operator++()** is the pre-increment operator, and the **operator++(int)** is the post-increment operator.

#include<stdlib.h>

#include<stdio.h>

#include<iostream>

**using** **namespace** std;

**class** Test {

**int** x;

**public**:

**void**\* operator **new**(**size\_t** size);

**void** operator **delete**(**void**\*);

    Test(**int** i) {

        x = i;

        cout << "Constructor called n";

    }

    ~Test() { cout << "Destructor called n"; }

};

**void**\* Test::operator **new**(**size\_t** size)

{

**void** \*storage = **malloc**(size);

    cout << "new called n";

**return** storage;

}

**void** Test::operator **delete**(**void** \*p )

{

    cout<<"delete called n";

**free**(p);

}

**int** main()

{

    Test \*m = **new** Test(5);

**delete** m;

**return** 0;

}

New

Const

Destrutor

Delete

### Why do we give semicolons at the end of class?

Many people might say that it’s a basic syntax and we should give a semicolon at the end of the class as its rule defines in cpp. But the main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.

Yes just like structure and union, we can also create the instance of a class at the end just before the semicolon. As a result, once execution reaches at that line, it creates a class and allocates memory to your instance.

#include <iostream>

using namespace std;

class Demo{

int a, b;

public:

Demo() // default constructor

{

cout << "Default Constructor" << endl;

}

Demo(int a, int b):a(a),b(b) //parameterised constructor

{

cout << "parameterized constructor -values" << a << " "<< b << endl;

}

void print()

{

cout<<"done"<<endl;

}

}instance;

int main()

{

instance.print();

}

**Why a pure virtual destructor requires a function body?**

The reason is that destructors (unlike other functions) are not actually ‘overridden’, rather they are always called in the reverse order of the class derivation. This means that a derived class destructor will be invoked first, then the base class destructor will be called. If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore the compiler and linker enforce the existence of a function body for pure virtual destructors.

// C++ program to demonstrate if the value of

// of pure virtual destructor are provided

// then the program compiles & runs fine.

#include <iostream>

// Initialization of base class

class Base {

public:

virtual ~Base() = 0; // Pure virtual destructor

};

Base::~Base() // Explicit destructor call

{

std::cout << "Pure virtual destructor is called";

}

// Initialization of derived class

class Derived : public Base {

public:

~Derived() { std::cout << "~Derived() is executed\n"; }

};

int main()

{

// Calling of derived member function

Base\* b = new Derived();

delete b;

return 0;

}

virtual ~ITooldMRILXXLTelemMRILD () override = default;

1. **override**: This keyword is used to indicate that this destructor is intended to override a virtual destructor in a base class. The use of **override** is optional but recommended for clarity and to catch errors at compile time.
2. **= default**: This part is specifying the default implementation for the virtual destructor. In this context, it means that the compiler should generate the default implementation for the virtual destructor. The **= default** syntax is used to explicitly request the compiler to generate the default implementation.

The default implementation for a virtual destructor in C++ is essentially an empty destructor. When you declare a virtual destructor with **= default**, you are instructing the compiler to generate the default implementation for the destructor. This default implementation does not contain any additional code and essentially performs no specific cleanup actions.