OOP Mid 2

Tuesday, 19 March 2024 6:24 pm

Inheritance is a feature or a process in which new classes are created from the existing classes.

- The new created class is called "derived" or "child" class
- The existing class is called "base" or "parent" class.

Derived class is inherited from base class.

Think of it in terms of parent-child class and it becomes easier.
The child gets to inherit whatever the parent owns (the methods, the attributes) just depends on what the parent gives (private, protected or public).

Let's understand from an example:

```cpp
// Base class: Course
class Course {
private:
    string courseName;
    int courseId;
public:
    Course(string name, int id) : courseName(name), courseId(id) {}
    void displayInfo()  {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

Here we have a base class Course will public members.

```cpp
// Derived class: Student
class Student : public Course {
private:
    string studentName;
    int studentId;

public:
    Student(string name, int id, string sName, int sId)
        : Course(name,id) , studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }
};
```

Inherited Course

So now the class Student can access everything the class Course has.
Now before we understand everything let's first understand which keyword allows us to access what from the parent class.

There are 3 modes of inheritance:

1- **Public:** All the public members of the parent class will be publicly available in the child class. Meaning you can access these members outside as well. Protected members will become protected in child class.

```
// Derived class: Student
class Student : public Course {
private:
    string studentName;
    int studentId;

public:
    Student(string name, int id, string sName, int sId)
        : Course(name,id) , studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }
};
```

→ Attributes of the Student class.

Note: Because there is a constructor in Course class, we have to call that Constructor in the member initialization list.

→ Because displayInfo() is a public member in Course class, we are able to call it inside Student class.

Now let's create an object of the Student Class:

```
int main() {

    Student s("Math", 101, "Ali", 1001);
    s.displayStudentInfo();

    return 0;
}
```

→

```
Course Name: Math
Course ID: 101
Student Name: Ali
Student ID: 1001
```

! Because Student inherited Course, it also got all the public members, thus was able to execute the function.

Now if I make the method private:

```
// Base class: Course
class Course {
private:
    string courseName;
    int courseId;
    void displayInfo()  {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
public:
    Course(string name, int id) : courseName(name), courseId(id) {}

};
```

Private members cannot be accessed with inheritance, I am not be to call displayInfo() in my derived class

Errors

```
D:\[redacted]: In member function 'void Student::displayStudentInfo()':
D:\[redacted]:30:9: error: 'void Course::displayInfo()' is private within this context
   30 |         displayInfo(); // Calling base class displayInfo function
      |         ^~~~~~~~~~~
D:\[redacted]:10:10: note: declared private here
   10 |     void displayInfo()  {
      |          ^~~~~~~~~~~
D:\[redacted]:30:20: error: 'void Course::displayInfo()' is private within this context
   30 |         displayInfo(); // Calling base class displayInfo function
      |                    ^~
D:\[redacted]:10:10: note: declared private here
   10 |     void displayInfo()  {
      |          ^~~~~~~~~~~
```

Let's see what happens if we access private attributes:

```cpp
// Derived class: Student
class Student : public Course {
private:
    string studentName;
    int studentId;

public:
    Student(string name, int id, string sName, int sId)
        : Course(name,id) , studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }

    void updateCourse()
    {
        courseName = "PF";
    }
};
```

Trying to change the value of private variable courseName from child class.

```
D:\[redacted]p: In member function 'void Student::updateCourse()':
D:\[redacted]:36:17: error: 'std::string Course::courseName' is private within this context
   36 |                     courseName = "PF";
      |                     ^~~~~~~~~~
D:\[redacted]p:8:12: note: declared private here
    8 |     string courseName;
      |            ^~~~~~~~~~
```

2) Protected: All the public members of the base class become protected while private remain as private.

Before we discuss protected inheritance, let's look at protected keyword itself.

It basically is an access modifier that allows any attribute with it to be access through other classes but not outside the class.

Above we had an issue accessing private variable in the derived class so we can use protected, so this way that variable can be accessed by other classes but not outside.

```cpp
// Base class: Course
class Course {
protected:
    string courseName;
    int courseId;
public:
    Course(string name, int id) : courseName(name), courseId(id) {}
    void displayInfo()  {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

→ Now courseID is protected

```cpp
// Derived class: Student
class Student : public Course {
private:
    string studentName;
    int studentId;

public:
    Student(string name, int id, string sName, int sId)
        : Course(name,id) , studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }

    void updateCourse()
    {
        courseName = "PF";
    }
};
```

Accessing a protected member

```cpp
s.updateCourse();
```

No more errors →

```
Course Name: Math
Course ID: 101
Student Name: Ali
Student ID: 1001
```

Now lets make displayInfo() protected:

```cpp
s.displayInfo();
```

Right now displayInfo() is public so it works →

```
Course Name: Math
Course ID: 101
```

```cpp
// Base class: Course
class Course {
protected:
    string courseName;
    int courseId;
    void displayInfo()  {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
public:
    Course(string name, int id) : courseName(name), courseId(id) {}

};
```

Protected

let's run it again:

```
D:\...\...pp: In function 'int main()':
D:\...\...:47:18: error: 'void Course::displayInfo()' is protected within this context
   47 |     s.displayInfo();
      |     ~~~~~~~~~~~~~~^~
D:\...\...pp:10:10: note: declared protected here
   10 |     void displayInfo()  {
      |          ^~~~~~~~~~~
```

Cannot access protected member outside class

Now let's look at protected inheritance:

This will convert all the public members of base class to protected in derived class.

```cpp
// Base class: Course
class Course {
private:
    string courseName;
    int courseId;
public:
    Course(string name, int id) : courseName(name), courseId(id) {}
    void displayInfo()  {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

Private

Public

Will become protected in derived class thus unable to access outside of class

```cpp
// Derived class: Student
class Student : protected Course {
private:
    string studentName;
    int studentId;

public:
    Student(string name, int id, string sName, int sId)
        : Course(name,id) , studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }
};
```

Protected Inheritance

```
D:\...\...p: In function 'int main()':
D:\...\...:41:18: error: 'void Course::displayInfo()' is inaccessible within this context
   41 |     s.displayInfo();
      |     ~~~~~~~~~~~~~~^~
D:\...\...:12:10: note: declared here
   12 |     void displayInfo()  {
      |          ^~~~~~~~~~~
D:\...\...:41:18: error: 'Course' is not an accessible base of 'Student'
   41 |     s.displayInfo();
      |     ~~~~~~~~~~~~~~^~
```

3) Private: All public and protected members become private in derived class.

Pretty much same as above, just that all inherited members become private.

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance:

1- **Single Inheritance:** A class is allowed to inherit from only one class.

The example done above was of single inheritance as Student inherited from Course only.

2- **Multiple Inheritance:** A class can inherit from more than one class.

To demonstrate this, let's create another class:

```cpp
// Another base class: Teacher
class Teacher {
private:
    string teacherName;
    int teacherId;
public:
    Teacher(string name, int id) : teacherName(name), teacherId(id) {}
    void displayTeacherInfo()  {
        cout << "Teacher Name: " << teacherName << endl;
        cout << "Teacher ID: " << teacherId << endl;
    }
};
```

Now to inherit this class to Student

```cpp
// Derived class: Student
class Student : public Course, public Teacher {
private:
    string studentName;
    int studentId;

public:
    Student(string courseName, int courseId, string teacherName, int teacherId, string sName, int sId)
        : Course(courseName, courseId), Teacher(teacherName, teacherId), studentName(sName), studentId(sId) {}

    void displayStudentInfo() {
        displayTeacherInfo(); // Calling base class displayTeacherInfo function from Teacher
        displayInfo(); // Calling base class displayInfo function from Course
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }
};
```

Second Inheritance

We have to call Teacher constructor aswell

Let's execute it:

```cpp
Student s("Math", 101, "Mr. Smith", 201, "Ali", 1001);
s.displayStudentInfo();
```

```
Teacher Name: Mr. Smith
Teacher ID: 201
Course Name: Math
Course ID: 101
Student Name: Ali
Student ID: 1001
```

Now we also see Teacher attributes in output

3- Multilevel Inheritance: In this a derived class is created from another derived class.

To demonstrate this, let's create a derived class from the already derived class Student

Current Derived Class:

```
// Derived class: Student
class Student : public Course {
private:
    string studentName;
    int studentId;

public:
    Student(string courseName, int courseId, string studentName, int studentId)
        : Course(courseName, courseId), studentName(studentName), studentId(studentId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Student Name: " << studentName << endl;
        cout << "Student ID: " << studentId << endl;
    }
};
```

Class derived from Student:

```
// Derived class: GraduateStudent
class GraduateStudent : public Student {
private:
    string researchTopic;

public:
    GraduateStudent(string courseName, int courseId, string studentName, int studentId, string researchTopic)
        : Student(courseName, courseId, studentName, studentId), researchTopic(researchTopic) {}

    void displayGraduateStudentInfo() {
        displayStudentInfo(); // Calling base class displayStudentInfo function
        cout << "Research Topic: " << researchTopic << endl;
    }
};
```

- As it inherits Student, it in turn also inherits Course meaning it inherits members of Course as well

Let's execute it:

```
GraduateStudent gs("Math", 101, "Ali", 1001, "Machine Learning");
gs.displayGraduateStudentInfo();
```

→

```
Course Name: Math
Course ID: 101
Student Name: Ali
Student ID: 1001
Research Topic: Machine Learning
```

Output course and Student details.

Note: Because it also inherits course I can do this as well.

```
gs.displayInfo();
```

→

```
Course Name: Math
Course ID: 101
```

4- Hierarchical Inheritance: In this more than one subclass is inherited from a single base class.

To demonstrate this let's change the code a bit. Let's create a base class Person, then inherit the other classes from it.

```cpp
// Base class: Person
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
    }
};
```

Base Class Person

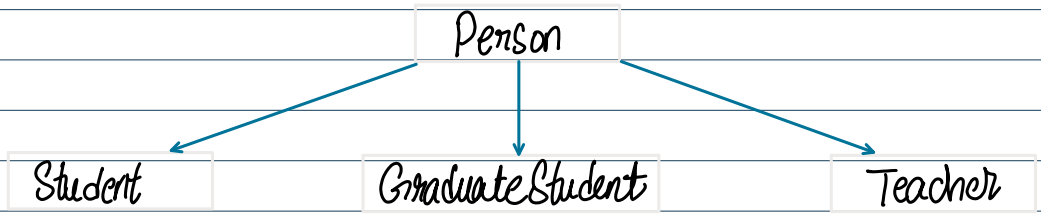
```cpp
// Derived class: Student
class Student : public Person {
private:
    string courseName;
    int courseId;
public:
    Student(string name, int id, string courseName, int courseId)
        : Person(name, id), courseName(courseName), courseId(courseId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

```cpp
// Derived class: GraduateStudent
class GraduateStudent : public Person {
private:
    string researchTopic;
public:
    GraduateStudent(string name, int id, string researchTopic)
        : Person(name, id), researchTopic(researchTopic) {}

    void displayGraduateStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Research Topic: " << researchTopic << endl;
    }
};
```

```cpp
// Derived class: Teacher
class Teacher : public Person {
private:
    string subject;
public:
    Teacher(string name, int id, string subject)
        : Person(name, id), subject(subject) {}

    void displayTeacherInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Subject: " << subject << endl;
    }
};
```

All the derived classes inherited from base class Person.

This can be seen as following:



We can further create more derived classes from the current derived classes. Person will be inherited to the new derived class aswell.

A potential problem with Hybrid Inheritance: Diamond Problem

When we are doing hybrid inheritance and then we do multiple inheritance, we have what is known as Diamond Problem

Let's demonstrate it:

```cpp
// Base class: Person
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
    }
};
```

I have a base class Person

I have hybrid inheritance here

```cpp
// Derived class: Student
class Student : public Person {
private:
    string courseName;
    int courseId;
public:
    Student(string name, int id, string courseName, int courseId)
        : Person(name, id), courseName(courseName), courseId(courseId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

```cpp
// Derived class: Teacher
class Teacher : public Person {
private:
    string subject;
public:
    Teacher(string name, int id, string subject)
        : Person(name, id), subject(subject) {}

    void displayTeacherInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Subject: " << subject << endl;
    }
};
```

Now to do multiple inheritance

```cpp
// Derived class: GraduateStudent
class GraduateStudent : public Student, public Teacher {
private:
    string researchTopic;
public:
    GraduateStudent(string name, int id, string courseName, int courseId, string subject, string researchTopic)
        : Student(name, id, courseName, courseId), Teacher(name, id, subject), researchTopic(researchTopic) {}

    void displayGraduateStudentInfo() {
        displayStudentInfo(); // Calling base class displayStudentInfo function
        displayTeacherInfo(); // Calling base class displayTeacherInfo function
        cout << "Research Topic: " << researchTopic << endl;
    }
};
```

The issue which occurs is that GraduateStudent inherit the class Person twice, therefore duplicating all the variables and constructor meaning that constructor of Person gets called twice, and even when assigning values we would run into errors. Let's see this error in action.

```cpp
// Derived class: GraduateStudent
class GraduateStudent : public Student, public Teacher {
private:
    string researchTopic;
public:
    GraduateStudent(string name, int id, string courseName, int courseId, string subject, string researchTopic)
        : Student(name, id, courseName, courseId), Teacher(name, id, subject), researchTopic(researchTopic) {}

    void displayGraduateStudentInfo() {
        displayStudentInfo(); // Calling base class displayStudentInfo function
        displayTeacherInfo(); // Calling base class displayTeacherInfo function
        cout << "Research Topic: " << researchTopic << endl;
    }

    void set() {name = "TEST";}
};
```

↳ Now which inherited variable of Person is value TEST set to

Error:

```
D:\[redacted] In member function 'void GraduateStudent::set()':
D:\[redacted]63:17: error: reference to 'name' is ambiguous
   63 |     void set() {name = "TEST";}
      |                 ^~~~
D:\[redacted]9:12: note: candidates are: 'std::string Person::name'
    9 |     string name;
      |            ^~~~
D:\[redacted]:12: note:                 'std::string Person::name'
```

How to fix it? Use of virtual keyword (Will discuss virtual in detail later, for now understand how it helps here)

We need to add virtual in hybrid inheritance

```cpp
// Derived class: Student
class Student : virtual public Person {
private:
    string courseName;
    int courseId;
public:
    Student(string name, int id, string courseName, int courseId)
        : Person(name, id), courseName(courseName), courseId(courseId) {}

    void displayStudentInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

```cpp
// Derived class: Teacher
class Teacher : virtual public Person {
private:
    string subject;
public:
    Teacher(string name, int id, string subject)
        : Person(name, id), subject(subject) {}

    void displayTeacherInfo() {
        displayInfo(); // Calling base class displayInfo function
        cout << "Subject: " << subject << endl;
    }
};
```

What this does is, is that in GraduateStudent it creates a single instance of the class Person, therefore eliminating duplication of Person. However now because Person is directly inherited in GraduateStudent we now have to call Person in initializer list too - (Previously Person was called in hybrid classes).

```cpp
// Derived class: GraduateStudent
class GraduateStudent : public Student, public Teacher {
private:
    string researchTopic;
public:
    GraduateStudent(string name, int id, string courseName, int courseId, string subject, string researchTopic)
        : Student(name, id, courseName, courseId), Teacher(name, id, subject), researchTopic(researchTopic), Person(name, id) {}

    void displayGraduateStudentInfo() {
        displayStudentInfo(); // Calling base class displayStudentInfo function
        displayTeacherInfo(); // Calling base class displayTeacherInfo function
        cout << "Research Topic: " << researchTopic << endl;
    }

    void set() {name = "TEST";}
};
```

→ Now this works

Now that we know inheritance, let's learn about the order of constructor and destructor call in multiple inheritance

When we do inheritance, there is always an order in which constructors are called. To demonstrate this, let's make a few changes to our constructors.

```cpp
Person(string name, int id) : name(name), id(id) {cout << "Base Class Person Constructor\n";}
```

```cpp
Student(string name, int id, string courseName, int courseId)
    : Person(name, id), courseName(courseName), courseId(courseId) {cout << "Derived Class Student Constructor\n";}
```

```cpp
Teacher(string name, int id, string subject)
    : Person(name, id), subject(subject) {cout << "Derived Class Teacher Constructor\n";}
```

```cpp
GraduateStudent(string name, int id, string courseName, int courseId, string subject, string researchTopic)
    : Student(name, id, courseName, courseId), Teacher(name, id, subject), researchTopic(researchTopic), Person(name, id)
    {cout << "Derived Class GraduateStudent Constructor\n";}
```

Added cout statements to all the constructors

Let's run the program and see in which we get the outputs:

```cpp
int main() {
    GraduateStudent gs("Ali", 1001, "Math", 101, "Physics", "Machine Learning");
```

Output :

```
Base Class Person Constructor
Derived Class Student Constructor
Derived Class Teacher Constructor
Derived Class GraduateStudent Constructor
```

- First the base class constructor was called thus proving that always base class is called first.
- Then the two inherited classes constructor was called, proving that once again that the class which we are inheriting from, its constructor is always called first.

Now what about the order of destructors?

Let's make destructors:

```
~Person() {cout << "Base Class Person Destructor\n";}
~Student() {cout << "Derived Class Student Destructor\n";}
~Teacher() {cout << "Derived Class Teacher Destructor\n";}
~GraduateStudent() {cout << "Derived Class GraduateStudent Destructor\n";}
```

} Destructors of all the classes

Let's run it and see (before that go through how destructors work if you dont remember):

Output:

```
Derived Class GraduateStudent Destructor
Derived Class Teacher Destructor
Derived Class Student Destructor
Base Class Person Destructor
```

- First the last derived class destructor was called thus proving that the destructor of the object is called first.
- Destructor of base class is called last

So constructor goes top down while destructor goes bottom up

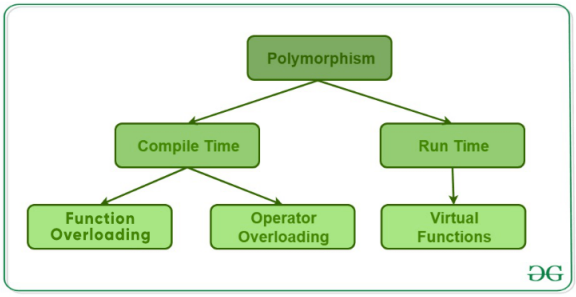
Note:

Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.

The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

Polymorphism:- The ability of a message to be displayed in more than one form.



1: Compile-time Polymorphism: It is called Early binding. In this the method or the operator with the same name performs different actions. The methods have different number of parameters or different parameters type. It is called early binding because compiler is aware of the functions with the same name and which overloaded function is to be called.

a) Function Overloading:

Let's see an example:

```
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

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

    void displayInfo(string name) {
        this->name = name;
        cout << "Changed Name: " << this->name;
    }
};
```

This is called because I passed no parameter

Functions with same names but different parameters

Let's see how they work

```
Person person("Ali", 1234);
person.displayInfo();
```

Output →

```
Name: Ali
ID: 1234
```

```
Person person("Ali", 1234);
person.displayInfo("Andrew");
```

Because I passed a parameter the second function was called →

```
Changed Name: Andrew
```

We can also do function overloading by changing parameter type:

```
void setInfo(string value)
{
    name = value;
}

void setInfo(int value)
{
    id = value;
}
```

Same number of parameters but different data type

```
Person person("Ali", 1234);

person.setInfo("Andrew");
person.setInfo(5678);

person.displayInfo();
```

calling same function → but with different data type

```
Name: Andrew
ID: 5678
```

NOT POSSIBLE: Changing only return type:

```
void displayInfo() {
    cout << "Name: " << name << endl;
    cout << "ID: " << id << endl;
}

string displayInfo() {
    return name;
}
```

Incorrect way of Polymorphism

```
D:\[redacted]p:19:12: error: 'std::string Person::displayInfo()' cannot be overloaded with 'void Person::displayInfo()'
   19 |     string displayInfo() {
      |            ^~~~~~~~~~~
D:\[redacted]p:14:10: note: previous declaration 'void Person::displayInfo()'
   14 |     void displayInfo() {
      |          ^~~~~~~~~~~
```

- Static functions cannot be overloaded either.

b) Operator Overloading: It is a compile time polymorphism. It gives a special meaning to an already existing operator without changing the original meaning

This mainly done when we wanna subtract classes and such

```
class StudentMarks {
public:
    int pf, oop;
    StudentMarks(int pf = 0, int oop = 0) : pf(pf), oop(oop) {}
};
```

Here I have a class with pf, oop

```
StudentMarks mid1(20, 30);
StudentMarks mid2(30, 40);
```

Here I created two classes mid1, mid2.

Now if I want to add mid1 marks to mid2:

```
StudentMarks final = mid1 + mid2;

cout << "Total Marks PF: " << final.pf << ", Total Marks OOP: " << final.oop;
```

I will get an error

```
D:\[redacted]p: In function 'int main()':
D:\[redacted]p:101:31: error: no match for 'operator+' (operand types are 'StudentMarks' and 'StudentMarks')
  101 |     StudentMarks final = mid1 + mid2;
      |                          ~~~~ ^ ~~~~
      |                          |    |
      |                          |    StudentMarks
      |                          StudentMarks
```

So here we will do operator overloading, so that we can add them:

```cpp
class StudentMarks {
public:
    int pf, oop;
    StudentMarks(int pf = 0, int oop = 0) : pf(pf), oop(oop) {}

    StudentMarks operator+(StudentMarks& stdmarks) {
        StudentMarks temp(0, 0);

        temp.pf = this->pf + stdmarks.pf;
        temp.oop = this->oop + stdmarks.oop;

        return temp;
    }
};
```

operator keyword then the operator you want to overload

The object to the left

The object to the right of the operator

Creating a new variable to store the addition so that we can return the newly added variable.

Adding the marks and storing in temp variable

Let's execute again:

```cpp
StudentMarks final = mid1 + mid2;

cout << "Total Marks PF: " << final.pf << ", Total Marks OOP: " << final.oop;
```

Output:

```
Total Marks PF: 50, Total Marks OOP: 70
```

The example we did above is known as Overloading Binary Operator. In Overloading Binary Operator there is one parameter passed. This overloading is the overloading of an operator on two operands.

mid1, mid2

Now Overloading Unary Operator:

In the unary operator function, no parameters are passed. It works with only one class objects.

Let's see what it means by this:

```cpp
class StudentMarks {
public:
    int pf, oop;
    StudentMarks(int pf = 0, int oop = 0) : pf(pf), oop(oop) {}

    void operator-() {
        pf -= 20;
        oop -= 20;
    }
};
```

No parameters

This time overloading the operator -

When - is placed on the object StudentMarks, 20 will be deducted from pf and oop.

```cpp
StudentMarks mid1(20, 30);

-mid1;

cout << "Total Marks PF: " << mid1.pf << ", Total Marks OOP: " << mid1.oop;
```

```
Total Marks PF: 0, Total Marks OOP: 10
```

Overloaded Operator can only be placed on left.

2- Run-time polymorphism: Also known as late Binding. The function call is resolved at run-time. In it function overriding is done when doing inheritance.

Overriding is done by defining same method with same name, parameters and signature.

```cpp
// Base class: Person
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
    }
};
```

```cpp
// Derived class: Student
class Student : public Person {
private:
    string courseName;
    int courseId;
public:
    Student(string name, int id, string courseName, int courseId)
        : Person(name, id), courseName(courseName), courseId(courseId) {}

    void displayInfo() {
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

Defined displayInfo() in derived class again, this replaces the method displayInfo() inherited from Person.

```cpp
Student student("Ali", 1001, "Math", 101);

student.displayInfo()
```

```
Course Name: Math
Course ID: 101
```

This method was called

While we are on function overriding, let's discuss Virtual Functions:

Virtual Functions: A virtual function is a member function that is declared within a base class and is overriden by a derived class.

In easier words: Virtual functions are functions that are written to be overriden in derived class.

```cpp
// Base class: Person
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

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

→ Because I know I am going to override this function in derived classes then I'll add the keyword virtual.

A few rules of Virtual Functions:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

From <https://www.geeksforgeeks.org/virtual-function-cpp/>

That is all that I think we need to know about virtual functions for now, however if you want you can read on it: https://www.geeksforgeeks.org/virtual-function-cpp/

Friend Function and Class:

Friend Class:

A friend class can access private and protected members of other classes in which it is declared as a friend.

(The difference between friend and inheritance is that we cannot access private members of base class in inheritance whereas through friend we can)

```cpp
// Base class: Person
class Person {
private:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

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

    friend class Student;
};
```

→ Declared class Student as a friend class.

```cpp
// Derived class: Student
class Student {
private:
    string courseName;
    int courseId;
public:
    Student(string courseName, int courseId)
        : courseName(courseName), courseId(courseId) {}

    void displayStudentInfo(Person& person) {
        person.name = "Andrew";
        person.id = 5678;
        person.displayInfo();
        cout << "Course Name: " << courseName << endl;
        cout << "Course ID: " << courseId << endl;
    }
};
```

→ Object of the class who this class is friend with.

→ Able to access private variables and changes their values

Can access public too.

```cpp
Person person("Ali", 1001);
Student student("Math", 101);

student.displayStudentInfo(person);
```

→

```
Name: Andrew
ID: 5678
Course Name: Math
Course ID: 101
```

→ Name changed from Ali to Andrew

→ ID changed from 1001 to 5678

Note: We can declare friend class or function anywhere in the base class body, whether its private, public or protected. It will work the same

Friend function:

Just like friend class, the friend function can be granted special access to private, protected members of a class.

A friend function can be global function or member function of another class.

Global:

```cpp
// Base class: Person
class Person {
private:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

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

    friend void displayDetails(Person& person);
};
```

→ Declared global function with the class itself as its parameters

```cpp
void displayDetails(Person& person) {
    cout << "Private Variables: Name = " << person.name << ", ID = " << person.id << endl;
}
```

} Function Global

↳ Private Variables

```cpp
Person person("Ali", 1001);

displayDetails(person);
```

Output →

```
Private Variables: Name = Ali, ID = 1001
```

Member function of another class:

```cpp
class Person; → Prototype
class Student {
private:
    string courseName;
    int courseId;
public:
    Student(string courseName, int courseId)
        : courseName(courseName), courseId(courseId) {}

    void displayStudentInfo(Person& person);
};
// Base class: Person
class Person {
private:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

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

    friend void Student::displayStudentInfo(Person& person);
};
```

Requirements:
The class which has the function should be above the class in which you declare the friend function.
Also the base class prototype must be at the top.

→ friend function declared

↳ Needed so it knows which Class function this is

Now below Person class is the friend function definition:

To specify which class the function is from

```
void Student::displayStudentInfo(Person& person) {
    cout << "Name: " << person.name << endl;
    cout << "ID: " << person.id << endl;
    cout << "Course Name: " << courseName << endl;
    cout << "Course ID: " << courseId << endl;
}
```

→ Accessing private members

```
Person person("Ali", 1001);
Student student("Math", 101);

student.displayStudentInfo(person);
```

Output →

```
Name: Ali
ID: 1001
Course Name: Math
Course ID: 101
```

Note: The order of class declarations, function definition matters alot in this so try to follow the same way I did.