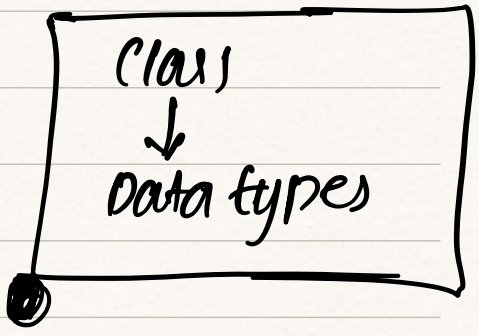
C++ :- Object oriented programming

- encapsulation
- abstraction
- inheritance
- polymorphism



→ classes are the fundamental unit of object oriented programming in C++.

→ class allows us to build user-defined types

→ Again as a summary:-

a) C++ is a compiled language, where another program called the compiler, converts your code into an executable program that the computer can run. It involves two steps:-

→ compile our code with a compiler
→ Run the executable file that the compiler outputs.

Structures

→ Structures allow developers to create their own types ("user-defined" types) to aggregate data relevant to their needs.

```
eg: struct Rectangle
    {
      float length;
      float breadth;
    };
```

Types

→ Every C++ variable is defined with a type.

```
int value;
Rectangle rectangle;
sphere Earth;
```

→ C++ includes fundamental types such as int, float. These fundamental types are sometimes called primitives.

→ The standard library provides or includes additional types such as `std::size_t` and `std::string`

User-defined Types

→ Structures are user-defined types. Structures are a way for programmers to create types that aggregate and store data in a way that makes sense in the context of a program.

→ For example, C++ doesn't have a fundamental type for storing a date. A programmer might desire to create a type to store a date.

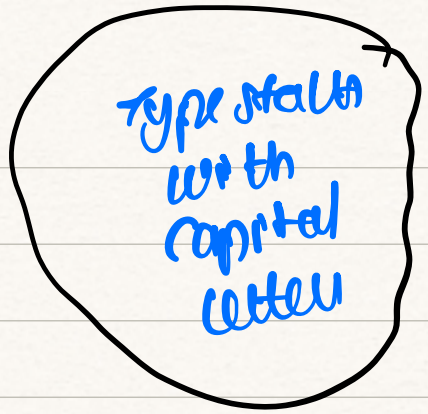
```
Struct Date
    {
    int day;
    int month;
    int year;
    };
```

// number variables

Create an instance of the Data structure is.

```
Date date;
```

```
date.day = 1;
date.month = 10;
date.year = 2019;
```



assert

→ assert is a macro provided by <cassert> header, used for debugging purposes.

```
#include <cassert>

assert (expression)
```

→ assert is active only in Debug mode.

Member initialisation in struct

```
Struct Date
    {
  int Day {1};
  int month {1};
  int year {2000};

   };
```

Access Specifiers

→ Members of a structure can be specified as public or private.

→ By default, all members of a structure are public, unless they are specifically marked private.

→ Public members can be changed directly, by any user of the object whereas private members can only be changed by the object itself.

Private members

```
Struct Date
    {
Private:
    int Date{13};
    int month{23};
    int year{2000};
};
```

→ Private members of a class are accessible only from within other member functions of the same class.

Protected

→ It implies that members are accessible from other member functions of the same class (or from their friends) and also from members of their derived class.

Accessors & Mutators

→ To access private members, we define accessor and mutator member functions.

→ It is also sometimes called "getter" and "setter" functions.

```
Struct Date
  {
Public:
  int Date ()
    { return day; }

Void day (int day)
```

```
{
this.day = day;
}

int month()
{
return month;
}

Void month (int month)
{
this.month = month;
}
int year ()
{
return year;
}
Void year (int year)
{
this.year = year;
}

Private:
int day[3];
int month[3];
int year[02];
};
```

Classes

·) Classes like structures, allow C++ programmers to aggregate data together in a way that make sense in the context of a specific program.

·) By convention, programmers use structures when member variables are independent of each other and use classes when member variables are related by an invariant.

⊙ IF RULES OR LOGIC ARE NEEDED TO ENSURE THE DATA IS VALID → USE A class

⊙ IF THE DATA CAN SIMPLY EXIST AS-IS WITHOUT ANY RESTRICTIONS, → USE A struct

When to use a class & a struct ?

Invariants

→ An 'invariant' is a rule that limits the values of member variables.

For eg: in a Date class, an invarrant would specify that the member variable day cannot be less than 0. Another invarriant would specify that the value of day cannot exceed 28, 29, 30 or 31, dipending on the month and year. Yet another invarriant would limit the value of month to the range of 1 to 12.

eg:

```
class Date
  {
  int day{13;
  int month{13;
  int year{03;
  };
```

// Simple class example.

The above class definition provides no invarrants. The data members can vary indipendently of each other.

→ Struct : All members are default to public.

→ Class : All members are default to private.

Initialization

- You can use in-class member initialization from (C++11)
  int x = 10; or int x {10};
- Otherwise, initialize private members in a Constructor.

- Why Brace initialization ?

→ Avoids narrowing conversions

int x = 10.5 ; // truncate to 10 → Allows

Int x {10.5} ; // Error (narrowing not allowed).

→ In-class initialisation
→ Brace initialisation
→ constructor initialisation

Here is an example of data members (or member variables) with invariants.

```
class Date
    {
public:
    int Day() { return day_; }

    void Day(int d) {
      if (d >= 1 && d <= 31) day_ = d;
      }
private:
    int day_ = 13;
    int month_ = 8;
    int year_ = 33;
};
```

invariants

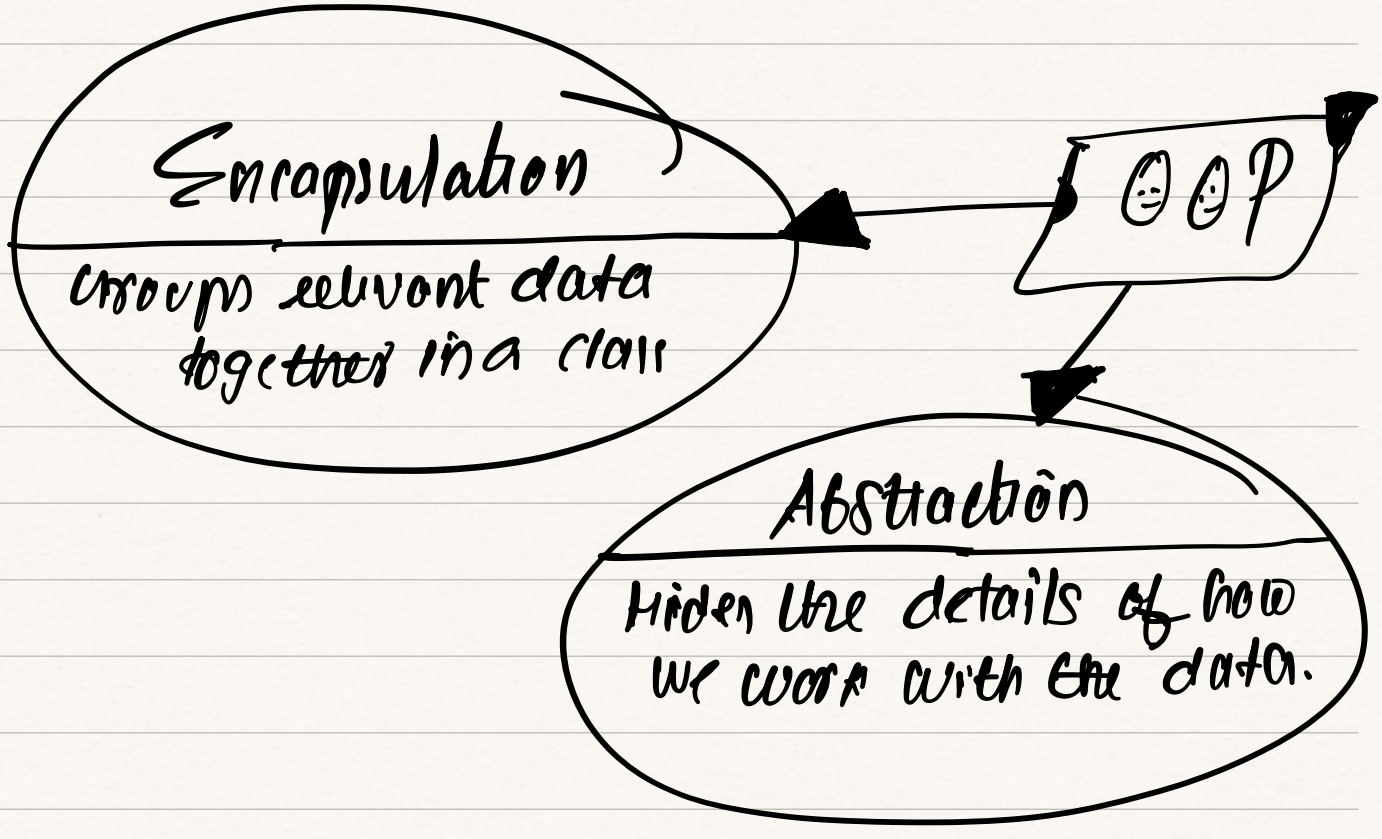
int day_ = 13

↳ underscore to differentiate that it is a member variable (naming convention)

Encapsulation & Abstraction

Encapsulation :- Means that we bundle related properties together in a single class and sometimes we protect those properties from unauthorized or accidental modifications.

Abstraction :- Means that users of our class only need to be familiar with the interface we provide, they don't have to know how we actually store member data or implement member functions.



Constructors

→ To instantiate an object of the class.

- Constructors are member functions of a class or struct that initialize an object.

- A constructor can take arguments, which can be used to assign values to member variables.

→ Default constructor

- We can initialize an object of this class, even though this class doesn't explicitly define a constructor.

- This is possible because of the default constructor. The compiler will define a default constructor which accepts no arguments, for any class or structure that doesn't contain an explicit-defined constructor.

Scope Resolution operator

- C++ allows different identifiers to have the same name, as long as they have different scopes.
- But in some cases, scopes can overlap, in which case the compiler may need assistance in determining which identifier the programmer meant to use. The process of determining which identifier to use is called scope resolution.

:: → scope resolution operator

→ ∴ can be used to specify which namespace or class to search in order to resolve an identifier.

Class

→ Each class provides its own scope. We can use the scope resolution operator to specify identifiers from a class.

```
class Date {
  public:
    int Day() { return day; }
    void Day(int day);
    int Month() { return month; }

  private:
    int day{1};
    int month{1};
    int year{2000};
};

void Date::Day(int day)
{
  if((day >= 1) && (day <= 31))
    Date::day = day;
}
```

// here the scope resolution operator is used.

Namespaces

→ Namespaces allow programmers to group logically related variables and functions

together.

→ It also avoids conflicts between two variables that have the same name in different parts of a program.

```
namespace English
  {
  void Hello ()
    {
    std::cout << "Hello, English";
    }

namespace Spanish
        {

  void Hello ()
        {
    std::cout << "Hola";
    }

int main()
  {
  English :: Hello();
```

```
Spanish :: Hello();
}
```

std namespace

- std namespace is used by the c++ standard library.
- classes like `std::vector` and functions like `std::sort` are defined within the `std` namespace.

Initializer list

- Initialized lists initialize member variables to specific values, just before the class constructor runs.

- This initialization ensures that the class members are automatically initialized when an instance of the class is created.

```
Date::Date(int day, int month, int year):
  year_(year)
```

```
{
Day(day);
Month(month);
}
```

→ Here the member value `year` is initialized through the initializer list.

→ `Day` and `month` allows us to apply the invariants set in the mutator.

Prefer initialization to assignment

⊛ Why prefer initialization ?

1) Efficiency
2) const and reference members
3) Clarity.

→ Below are some of the examples of initialization to assignment.

Example: Assignment vs. Initialization

Assignment in Constructor Body (Inefficient):

cpp Copy code

```cpp
class MyClass {
private:
    int x;

public:
    MyClass(int value) {
        x = value; // Assignment after default initialization
    }
};
```

1. **Steps**:
   - The compiler first default-initializes `x` (e.g., to some uninitialized garbage value).
   - The value is then reassigned using `x = value;`.

Initialization List (Preferred):

cpp Copy code

//good practice

```cpp
class MyClass {
private:
    int x;

public:
    MyClass(int value) : x{value} {} // Initialization directly
};
```

1. **Steps**:
   - `x` is directly initialized with the value `value`.
   - This avoids redundant default initialization.

Handling Const and Reference Members

Cannot Use Assignment:

cpp Copy code

```cpp
class MyClass {
private:
    const int y;
    int &ref;

public:
    MyClass(int value, int &reference) : y{value}, ref{reference} {}
};
```

- `y` (a `const` variable) and `ref` (a reference) must be initialized using the **initialization list**. Assignment won't work because:
  - `const` members can't change after initialization.
  - References must bind to an object during initialization.

→ In general, prefer initialization to assignment. Initialization sets the value as soon as the object exists, whereas assignment sets the value only after the object comes into being. This means that assignment creates and opportunity to accidently use a variable before its value is set.

→ In fact, initialization lists ensure that member variables are initialized before

the object is created. This is why class members can be declared const, but only if the member variable is initialized through an initialization list. Trying to initialize a const class member within the body of the constructor will not work.

Initializing constant members

Initializer list exists for a number of reasons:

→ Compiler can optimize initialization faster from an initialization list than from within the constructor.

→ If you have a const class attribute, you can only initialize it using an initialization list. → Otherwise violate the const keyword.

→ The attributes defined as reference must use initialization lists.

Encapsulation

→ Encapsulation is the grouping together of data and logic into a single unit. In OOP, classes encapsulate data and functions that operate on that data.

Accessor Functions (Getter function)

→ Accessor functions are public members functions that allow users to access an objects data albeit indirectly.

`const`

Accessors should only retrieve data. They should not change the data stored in the object.

Mutator Function (Setter function)

→ A mutator ("setter") can apply logic ("invariants") when updating member data.

→ Make it clear through proper naming conventions.

```
void setEmployeeAge(int newAge);
```

camel case

Abstraction

→ Abstraction is how we separate the interface of a class from its implementation.

→ We abstract the way the implementation details so that the user doesn't have to worry about them.

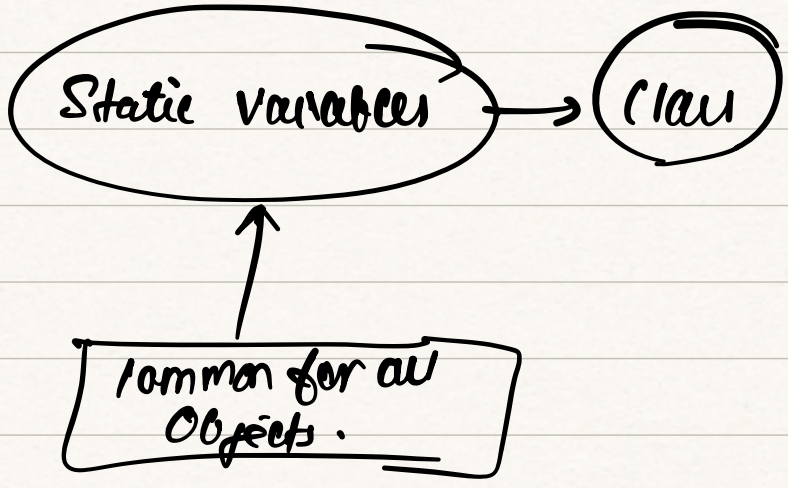
→ Abstraction refers to the separation of a class's interface from the details of its implementation. The interface provides a way to interact with an object, while hiding the details and implementation of how the class works.

`std::to_string` ← (converts to string)

→ Abstraction is used to expose only relevant information to the user. By hiding implementation details, we give ourselfs flexibility to modify how the program works.

Static variables

→ A static variable is a variable that belongs to the whole class and not just to an individual object of the class.



eg: Static float const exp Pi_ = 3.14513;

→ Class members can be declared static, which means that the member belongs to the entire class, instead of a specific instance of the class.

→ A `Static` member is created only once and then shared by all instances of the class. → That means if the `static` member gets changed, either by a user of the class or within a member function of the class itself, then all members of the class will see that change the next time they access the `Static` member.

implementation

→ Static members are declared within their class but in most cases they must be defined within the global scope. That's because memory is allocated for `static` variables immediately when the program begins, at the same time any global variables are initialized.

```
#include <iostream>

class Foo {
  public:
    static int count;
    Foo() { Foo::count++; }
};

int Foo::count{0};
```

```
int main() {
    Foo f3;
    cout << (Foo::count == 1);
}
```

→ An exception to the global definition of **static** members is if such members can be marked as **constexpr**. In that case, the static member variable can be both declared and defined within the **class** definition

IMP

In C++, static members of a class are :-

1. Declared inside the class
2. Defined outside the class in the global scope to allocate memory.

Exception :- Unless it's a **constexpr** static member of integral or const type, which can be initialized in one.

→ Static member can be modified by using class namespace as well as objects.

Eg.:
```
Foo::static_member = 10;   // modified directly
Foo f;
```

f. static_member = 20; // modified using the object.

IMP

→ You can't declare a non-static data member as constexpr.

→ In c++, constexpr can only be used for static data members

Correct Usage for constexpr in Class: (IMP)

1. **For non-static members**: You can use `const` to indicate the value is constant for an instance but not evaluated at compile time.
2. **For static members**: You can use `constexpr` for compile-time evaluation.

Static Methods

→ Just like static members, static methods are instance independent. They belong to the class, not to any particular instance of the class.

→ Static methods can be invoked without ever creating an instance of the class.

→ Encapsulation
→ Abstraction
→ inheritance
→ polymorphism

} Concepts of OOPs

Inheritance

general stuff → specific stuff

→ It refers to a hierarchical structure of classes where one class inherits from another.

→ Top class or Base class or Parent class
↓
derived class or child class

```
class vehicle {
 public:
    int wheels = 0;
    std::string color = "blue";

 void print() const
    {
  std::cout << color;
    }};
```

Child classes

```
Class car : Public vehicle
    {
  Public:
    bool sunroof = false;
    };

Class Bicycle : Public vehicle {
  public:
    bool kickstand = true;
    };
```

Access Modifiers

```
Class Animal
    {
  Public:
      Void Talk() const { std::cout << "Talk"; }
    };

Class Human : Public Animal
      {
```

```
Public:
    void Talk( std::string content) const
        {
        std::cout << content << "\n"; };


Class Baby : Private Human
            {
    Public:
        Void Cry()
            {
            Talk(" Whaaa !");
                }
        };

Int main()
    {
    Human human;
    human.Talk ("Hello, world");
    Baby baby;
    baby.Cry();

    baby.Talk("Meeee");
    }
```

Access modifier

// doesn't work because it is inherited privately

→ So Baby object can't call the functions from Human class, but baby itself can call talk() function from inside baby class.

Inherited Access Specifiers

→ Just at access specifies **Public**, **Protected** and **private** which class members users can access, the same access modifiers also define which class members users of a derived classes can access.

Public inheritance

→ The public and protected members of the base class listed after the specifier keep their member access in the derived class.

Protected inheritance

→ The public and protected members of the base class listed after the specifier are protected members of the derived class.

Private inheritance

→ The public and protected members of the base class listed after the specifier are private members of the derived class.

Public Inheritance

- **Description**: The **public** and **protected** members of the base class retain their original access levels in the derived class.
- **Effect on Members**:
  - **Public** members of the base class become **public** members in the derived class.
  - **Protected** members of the base class remain **protected** in the derived class.
  - **Private** members of the base class are not directly accessible in the derived class (they remain private to the base class).

Syntax:

cpp Copy

```cpp
class Base {
public:
    int public_member;
protected:
    int protected_member;
private:
    int private_member;
};

class Derived : public Base {
    // public_member -> public
    // protected_member -> protected
    // private_member -> inaccessible
};
```

Example:

cpp Copy

```cpp
Base base;
Derived derived;
derived.public_member = 10; // Allowed
// derived.protected_member = 20; // Not allowed, still protected
```

Protected access specifier

→ Public members :- Become protected in the derived class

→ Protected :- Become protected in the derived class

→ Private :- Members of the base class are not directly accessible in the derived class.

Protected Inheritance

- **Description**: The **public** and **protected** members of the base class become **protected** in the derived class.
- **Effect on Members**:
  - **Public** members of the base class become **protected** in the derived class.
  - **Protected** members of the base class remain **protected** in the derived class.
  - **Private** members of the base class are not directly accessible in the derived class.

Syntax:

cpp Copy

```cpp
class Derived : protected Base {
    // public_member -> protected
    // protected_member -> protected
    // private_member -> inaccessible
};
```

Example:

cpp Copy

```cpp
Derived derived;
// derived.public_member = 10; // Not allowed, now protected
```

- Inside the derived class, `public_member` can still be accessed but is no longer public when exposed to users of the `Derived` class.

Private

→ Public and Protected members of the class become private in the child class.

Private Inheritance

- **Description**: The **public** and **protected** members of the base class become **private** in the derived class.
- **Effect on Members**:
  - **Public** members of the base class become **private** in the derived class.
  - **Protected** members of the base class become **private** in the derived class.
  - **Private** members of the base class are not directly accessible in the derived class.

Syntax:

```cpp
class Derived : private Base {
    // public_member -> private
    // protected_member -> private
    // private_member -> inaccessible
};
```

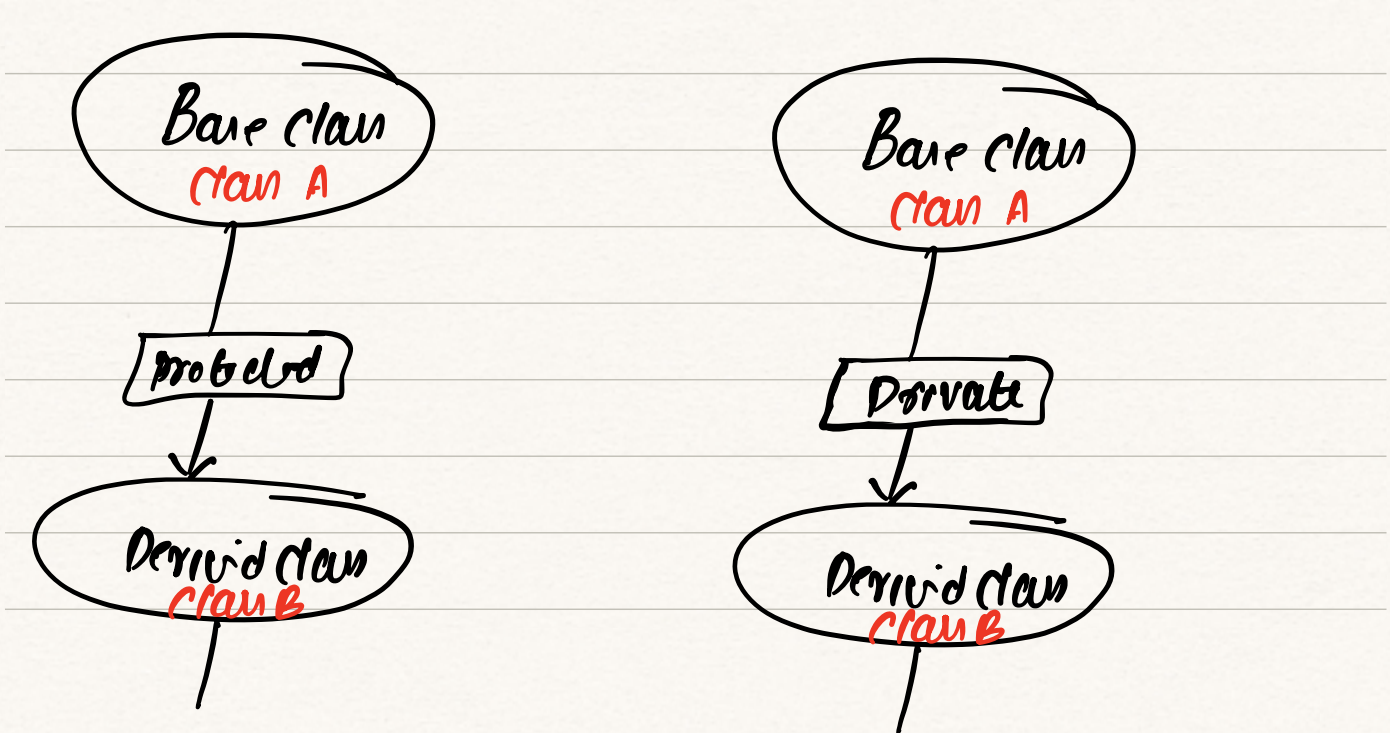
Example:

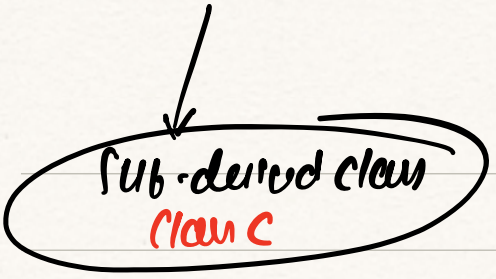
```cpp
Derived derived;
// derived.public_member = 10; // Not allowed, now private
```

- The derived class and its member functions can access `public_member` and `protected_member` from the base class, but these are not accessible outside the `Derived` class.

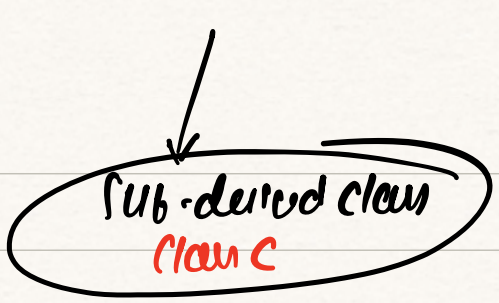
Difference Between protected and Other Specifiers:

Specifier	Accessible within the Class?	Accessible in Derived Classes?	Accessible Outside the Class?
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No





Here the class 'C' can access the class A since the class 'A' is exposed as protected.



Here class 'C' can't access the class 'A' since class 'A' is exposed as private.

When to use "protected" ?

→ When we need (derived class) to access and modify certain members while keeping those members hidden from external code.

IMP

To instantiate a class means to create an object of that class using its constructor. This process allocate memory for the object and initializes it according to the class definition

Composition

- In c++, composition is a design principle where one class contains objects of another classes as part of its data members.

- It is a core concept for building complex systems by combining smaller, reuseable components.

```
#include <iostream>
#include <string>

class Engine {
  Private:
     int horsepower;

  Public:
     Engine (int hp): horsepower (hp) {}
    Void Start ()
       {
        Std::cout << "Engine with " << horsepower;
       }};

class wheel
       {
```

```
Private :
    std::string type;

Public :
    wheel (const std::string& t) : type(t)
                {}
    void rotate() {
     std::cout << "The " << type << "is rotating";
     };

class car
      {
  Private :
     Engine engine;    // car has an Engine (composition)
     Wheel wheel;      // car has a wheel (composition)

  Public :
    car (int engineHP, const std::string & wheelType)
            : engine (engineHP), wheel (wheelType)
                {}

    void start()
       {
       engine.start();
       wheel.rotate();
```

```
};
int main()
{
car myCar(150, "All-Terrain");
myCar.Start();
return 0;
}
```

→ Composition is a closely related alternative to inheritance. Composition involves constructing ("composing") classes from other classes, instead of inheriting traits from a parent class.

→ A common way to distinguish "composition" from "inheritance" is to think about what an object can do, rather than what it is. This is often expressed as "has a" versus "is a".

has a ⇒ is a composition relationship where one object belongs to another object.

has a → possessive hierarchy

→ "has a" and "is a" a fundamental concepts in OOP and describe the relationship between objects and classes.

* Has a → composition → ie (one object has other object)
* Is a → inheritance → (one class is a specialized version of another class)

OOPs

Key Differences Between "Has-a" and "Is-a":

Aspect	"Has-a" Relationship	"Is-a" Relationship
Definition	One class contains another as a member.	One class derives from another class.
Concept	Composition	Inheritance
Dependency	Classes are loosely coupled.	Classes are tightly coupled.
Example	A `Car` has-a `Engine`.	A `Dog` is-a `Animal`.
Reuse	Promotes code reuse by combining objects.	Promotes reuse by extending behavior.
Flexibility	(Easier to replace or modify components.)	(Modifications affect all derived classes.)

use "has-a" composition when

→ You want build a class from smaller, reuseable components.
→ The relationship is more about ownership than behaviour.
→ Example : A House has a Door

use "is-a" inheritance when

→ You need to create a hierarchy of specialized types.
→ The relationship is more about type compatibility or polymorphism.
→ Example : A cat is a Pet.

composition "has-a" × inheritance "is-a"

There is no hard and fast rule about when to prefer composition over inheritance. In general, if a class needs only extend a small amount of functionality beyond what is already offered by another class, it makes sense to inherit from that other class.
However if a class needs to contain functionality from a variety of otherwise unrelated classes, it makes sense to compose the class from other classes.

Friends

⇒ The `friend` keyword grants access to the private members of a class.

```
#include <iostream>

class Heart
{
public:
    int rate{80};
    friend class Human;     // declared a friend class.
```

```
};

class Human
    {
  Public:
   Heart heart;   //Object of Heart class.
   Void Exercise()
     {
    heart.rate = 150;
     }
   Int HeartRate()
        {
      return heart.rate;
        }};

int main()
  {
 Human human;
 human.HeartRate();
 human.Exercise();
  }.
```

Friend can access the private members of a class. But inheritance can't. So is the composition

→ The main difference between classical inheritance and friend inheritance is that a friend class can access private members of the base class, which isn't the case for classical inheritance. In classical inheritance, a derived class can only access public and protected members of the base class.

POLYMORPHISM : OVERLOADING

→ Overloading is a type of polymorphism that allows us to pass different arguments to the same function name and have different implementations of that function that respond to the different arguments.

→ polymorphism means "assuming many forms".

In the context of OOPs, polymorphism describes a paradigm in which a function may behave differently depending on how it is called. In particular, the function will perform differently based on its inputs.

Polymorphism can be achieved by two ways in C++ :-

→ Overloading
→ Overriding // 2 ways

Overloading

In C++, we can write two (or more) versions of a function with the same name. This is called overloading. Overloading requires that we leave the function name the same, but we modify the function signature.

For eg: we might define the same function name with multiple different configurations of input arguments.

```
#include <ctime>

Class Date
    {
    Public:
        Date (int day, int month, int year):
          day_(day), month_(month), year_(year) {}

        Date (int day, int month) : day_(day), month_
                                    (month)
            {
            time_t t = time(NULL);
            tm* timeptr = localtime(&t);
```

```
    year_ = timePtr -> tm_year;
}

private:
    int day_;
    int month_;
    int year_;
};
```

Operator overloading

→ operator overloading allows us to assign our own logic to operators.
For eg: Add two classes together or multiply them.

→ Operator overloading in C++ can only be done inside a class or struct. This is because operators are treated as member functions or friend functions of a class/struct, and the overloading mechanism is designed to work with user-defined types.

Why operator overloading only be done in classes or Structs?

→ Encapsulation of Behaviour.
→ Custom Data types.

```
Class Point {
  Public :
    int x, y;
    Public Cint x, int y) : x(x), y(y) {}

    Point operator+ (const point& other)
        {
      return Point (x + other.x, y + other.y);
        }
};
```

What operators can't be overloaded?

:: Scope resolution
.* Pointer to member
. member access operator.
Sizeof
typeid

Member Function Syntax //(Syntax of operator overloading)

cpp Copy code

```cpp
class ClassName {
public:
    ReturnType operatorSymbol(const ParameterType &parameter) {
        // Define the operator's behavior here
        return some_value; // The return type depends on the operator's behavior
    }
};
```

operatorsymbol :→ +, -, *, /

Returntype :→ Depends on the operation. could be the same type as the class or something else.

* operator overloading can be useful for many things. Consider the + operator. we can use it to add int s double, float, or even std::string

→ In order to overload an operator, we the operator keyword in the function signature.

```
Complex operator+ (const Complex& addend)
{
// logic to add complex numbers;
}
```

Virtual functions

→ Virtual functions allow us to define an abstract class that can function as an interface from which other classes can be derived.

```
class Animal
{
virtual void talk() const = 0;
};

class Human : public Animal
{
public:
    void talk() const
    {
        std::cout << "Hi";
    }
};

int main()
{
    Human human;
    human.talk()
}
```

Pure virtual function

// Virtual function means that the derived class can override the talk function.

We are not defining the function. ie. the class Animal will be an abstract class (we never gonna create an object of class Animal)

Here :-

talk() is a pure virtual function ⟹ Animal is an abstract class and can't able to create an object. It will serve as an interface for

Subsequent derived classes that will inherit from it.

Note:

Virtual functions are a polymorphic feature. These functions are declared (and possibly defined) in a base class, and can be overriden by derived class.

The approach declares an interface at the base level, but delegates the implementation of the interface to the derived classes.

→ A pure virtual function is a virtual function that the base class declares but does not define

→ A pure virtual function has the side effect of making its class abstract. This means that the class cannot be instantiated. Instead, only classes that derive from the abstract class and override the pure virtual function can be instantiated.

```
class shape {
public:
  shape() {}
  virtual double area() const = 0;
};
```

Function Hiding

→ Function hiding is closely related, but distinct from overriding.

→ Essentially the functions in the base class are always hidden by functions of the same name in a derived class. No matter if the functions in the derived class overrides a base class virtual function or 'not'.

Function overriding

→ Function signatures must match {same name, parameter types, and const/non-const qualifiers}.

Function Hiding

→ Function hiding is closely related, but distinct from overriding.

A derived class hides a base class function, as opposed to overriding it, if the base class function is not specified to be virtual

```cpp
class Cat { // Here, Cat does not derive from a base class
public:
  std::string Talk() const { return std::string("Meow"); }
};

class Lion : public Cat {
public:
  std::string Talk() const { return std::string("Roar"); }
};
```

In the above example,

Cat → base class

Lion → derived class

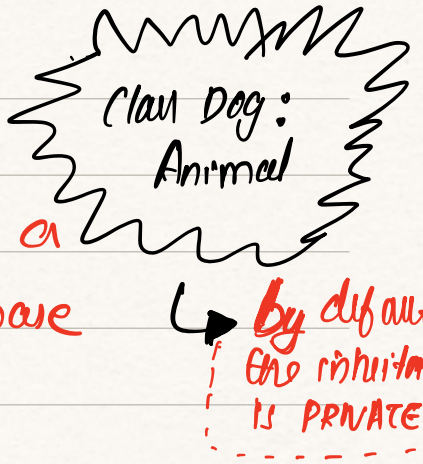
Both Cat and Lion has Talk() member functions. When an object of type Lion calls Talk(), the object will run Lion::talk(), not Cat::Talk()

In this situation, Lion::talk() is hiding Cat::talk(). If Cat::Talk() were virtual, then Lion::talk() would override Cat::talk(), instead of hiding it. Overriding requires a virtual function in the base class.

Override keyword

→ Override keyword specifies that the function talk is gonna override a virtual function somewhere.

→ Overriding of a function occurs when a derived class defines the implementation of a Virtual function that it inherits from a base class.




```
class Shape
  {
public:
    virtual double Area() const =0;
    virtual double Perimeter() const =0;
    };

class Circle : public Shape
    {
    Circle(double radius) : radius_(radius) {}
    double Area() const override
      {
      return pow(radius_,2) * PI ; }
    double Perimeter() const override
      {
      return 2 * radius_ * PI ;
      }
private:
    double radius_ ;
```

};

The "override" keyword tells both the compiler and the human programmer that the purpose of this function is to override a virtual function. The compiler will verify that a function specified as `override` does indeed override some other virtual function, or otherwise the compiler will generate an error.

→ Specifying a function as `override` is good practice, as it empowers the compiler to verify the code, and communicates the intention of the code to the future users.

Multiple inheritance

→ It allows to create a derived class from two different base classes

```
class car
  {
  public:
    std::string drive() { return "I am driving"; }};

car Boat {
  Public:
```

```
std::string cruise() { return "I am cruising"; }
};

class AmphibiousCar : public Boat, public car //
    {                                          derived class inherit from
    };                                         both parent class.

int main()
    {
    car car;
    Boat boat;
    AmphibiousCar duck;
    }
```

IMP

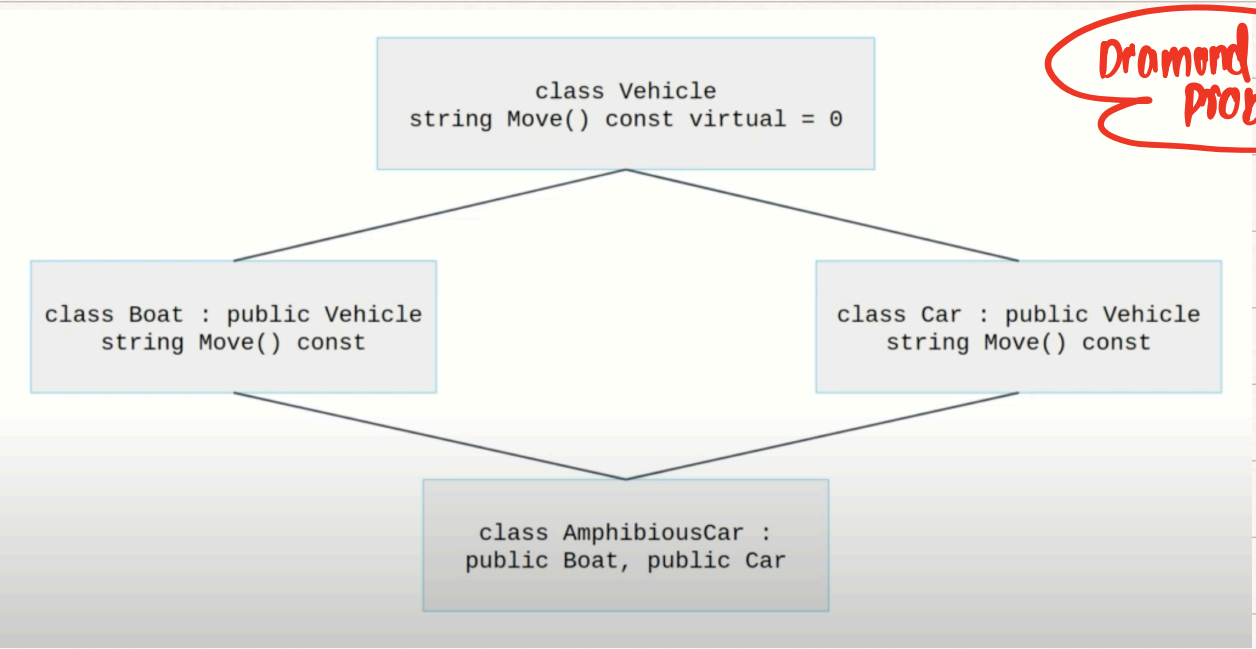
→ Multiple inheritance is risky and can present an issue known as the **diamond problem.**

→ The diamond problem occurs when a class inherits from two base classes, both of which themselves inherit from the same abstract class, in which case a conflict can emerge.

In the below example

→ An abstract class **vehicle** has a pure virtual function **Move()**

Diamond Problem



→ Now Boat and car inherits from the abstract class Vehicle and their own definition of move() function.

→ Now another class AmphibiousCar that inherits from both Boat and car. Now the problem is that the AmphibiousCar now inherits a move() function from both Boat and car and they are going to conflict.

→ Its not clear which function should actually be executed, if the user of an amphibious car calls the move() method.

Generic Programming

→ Generic programming is an example of polymorphism that generalizes the parameters of classes and functions. { template }

→ Using templates, we can build classes and functions that work with many different types of data.

Eg: Vector :→ we can create a vectos of anything.
(eg: int, String, user-defined type)

Alternative to inheritance :→ Generic programming

Templates

→ Tool for generic programming in C++
→ Template allow us to specify a generic type (for eg: T) and then use that generic type for the function signature and then within the function.

Syntax:

```
template <typename T>
```

```
#include <iostream>
#include <string>

template <typename T>
T Max (T a, T b)
    {
    Return  a > b ? a : b ;
    }

int main()
    {
    Max< int > (2, 4) ;
    Max< double > ( -1.0, -2.3) ;
    max < char> ('a', 'b') ;
    }
```

→ Templates enables generic programming by generalising a function to apply to any class.

```
template <typename type>
type sum ( type a, type b)
        {
        return a+b ;
        }
```

```
int main ()
{
    std::cout << Sum<double>(20.0, 17.3) << "\n";
}
```

When sum <double> calls

```
double Sum(double a, double b);
```

→ Vector's are indeed a generic class.

→ The keyword 'generic' specifies that the function is generic.

→ Generic code is the term for code that is independent of types.

→ Besides `typename`, the keyword `class` can also be used.

✱ In order to instantiate a templatized class, use a templatized constructor, for example :

```
Sum<double>(20.0, 13.7)
```

Generic programming in C++
→ template
→ concepts

Concepts {from c++ 20}

- Concepts are a new feature that provide a way to specify constraints on template parameters. They allow us to define requirements for types that can be used with a template, making code more readable, robust and easier to debug.

- A concept defines a set of requirements (constraints) that a type must satisfy. It is a boolean predicate evaluated at compile-time.

```
#include <concepts>

template <typename T>          // define a concept
concept Addable = requires (T a, T b)
    {
    {a+b} -> std::convertible_to <T>;
    };
```

```
template <Addable T>
T add (T a, T b)
    {
    return a+b;
    }
```

or

```
template <typename T>
    requires Addable <T>
T add (T a, T b)
    {
    return a+b;
    }.
```

can be added directly in the function signature for constraining function parameters.

```
auto add (Addable auto a, Addable auto b)
    {
    return a+b;
    }.
```

Template Deduction

→ Template deduction is when the compiler determines the type for a template automatically without us having to specify it.

```
Max<int>(1,2);
```

⇓

```
Max(1,2);
```

// compiler will deduct the type automatically

But what happens if we do this in std::vector?

normally: `std::vector<int> v{1,2,3};`

⇓

`std::vector v{1,2,3};`

// only from c++17 std it works.
So no need to specify the type of `std::vector <int>`