Table of Contents

Ways to write Code:

Sequential programming:

It is programming paradigm in which a program is executed sequentially, that is based on linear, one statement after the other (it is way to think as code each line after other)

Procedural programming:

It is a programming paradigm that organizes a program as a sequence of procedures or functions, which are reusable blocks of code that perform specific tasks (it way to think of procedures)

It achieves Readability: code divides into blocks

Reusability : block can use any time

Maintainability and extensibility : can add /adjust in the one place(block).

Object-Oriented Programming (OOP):

Paradigm? Not concept, it can give up on it .

It is a programming paradigm that is based on the concept of "objects".(it is way to think of data type).

It makes program more productive because of achieving readability, Reusability, Maintainability and extensibility.

key OOP concepts (4pillars) include:

Encapsulation: the practice of hiding the internal implementation details of an object from the outside world, and exposing only the essential features or interfaces that can be used to interact with that object that is done by **data types**.

Data type is a way to group related attributes and their actions. It also prevents access to the attributes from other data types; only the actions or children have access to the attributes.

Inheritance: The ability of one datatype to gain characteristics and actions from another datatype. The children have the option of keeping implementing on the parent's attributes and actions exactly as they were or to change them or add new ones.

Polymorphism: The ability to treat actions that have the same name but a different signature or implementation as though they were one.

Abstraction: The approach for calling or reusing an action without being concerned with the implementation.

New Words needs to know in OOP

Attribute : it is a variable, is a piece of data that belongs to a class or struct

Ex datatype car contains: color (attribute with typing string)

Action: a method or member function, is a behavior that an object can perform. It defines the operations that an object can do.

Es datatype car contains: start engine (action)

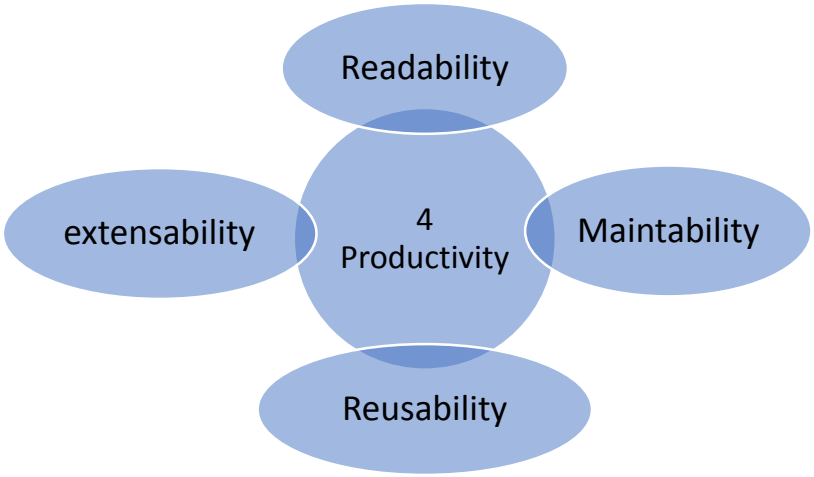
Member: is a general term that refers to both attributes and actions of a class/struct.

Encapsulation:

is a way to group related attributes and their actions. It also prevents access to the attributes from other data types; only the actions or children have access to the attributes.

Advantage:

It makes program more productive by achieving



Readability: it makes code more readable by adding all attributes and actions that are related to each other in one group.

Reusability: it can do multiple objects with worrying how datatype is implemented or duplicate code.

Maintainability /extensibility: by can adjust / add implementation of data type without effect on other parts of the program.

How to do it

1. Uses **struct or class** keyword to create data type :
   struct / class {
   Attriubtes ;
   Actions {}
   };

Ex in cpp :

```
struct Employee{
        int id ;
       string name ;
       display(){
        cout<<"The is id is "<<id <<"and the Name is "<<name<<endl;
      }
 }
```

2. Add **access** modifier.
   Access modifier: is keyword that used to adjust the access of Member of data type outside data type. it can be
   Private: it makes a member only access inside scope of datatype.
   Public: it makes a member only access inside/outside scope of datatype.
   Protected: it makes a member only access inside scope of datatype and scope of child datatype.
   Ex :

```
struct Employee{
private :
        int id ;
```

```
        string name ;
public :
display(){
        cout<<"The is id is "<<id <<"and the Name is "<<name<<endl;
       }
 }
```

Abstractions:

When object is created, it can call its actions to preform task without being concerned to how it implements.

Membership Function (action in datatype) vs Standalone function(function outside any scope like main)

	Membership function	Standalone function
in	Data type	global
It can access	Private, public and protected attribute of its datatype or parent data type	Only public attribute of any data type
Pass instance address	implicitly (call it by object name)	explicitly as parameter for it
Call it by Ex :	Object name.itsname e1.display()	Itsname Display()

This

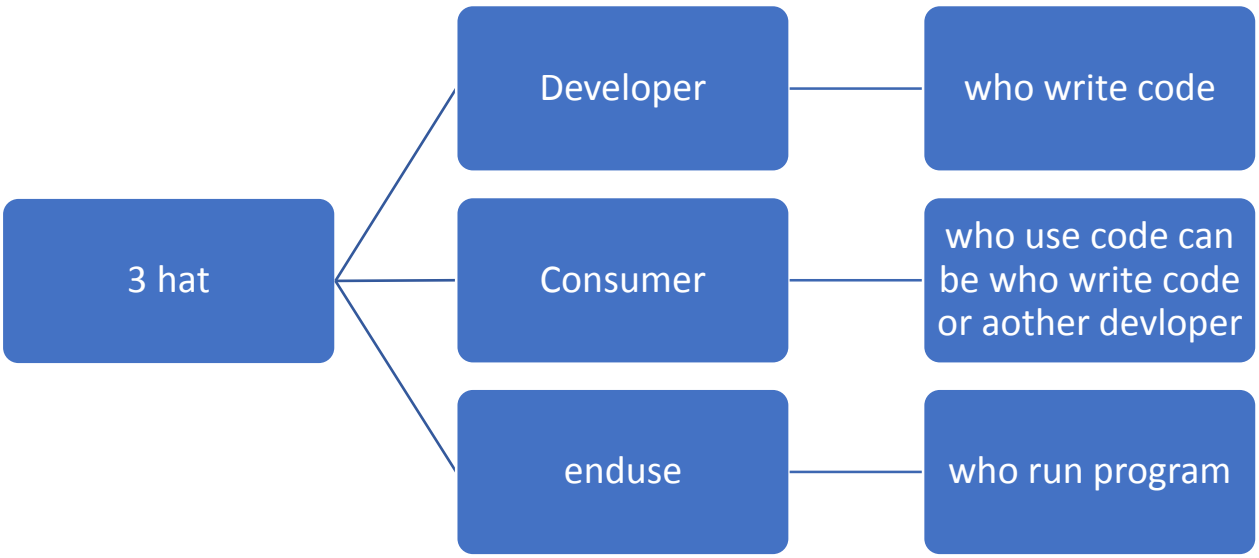
Pointer to caller object, it holds the address of the object that call its action.

Namespace

A collocation of attribute and action in scope in global with specific name

It used to 1. Organized data 2. Avoided conflict data type .

In evert program 3 Hats to know



Construct

It is special action/function/method that is automatic calling by compiler and its name with same name of datatype so complier can recognize it. It is used to initialize attribute once object is created using single line code

constructor Overriding

It is two constructure with the same name and signature (number of parameter is zero)

Signature means

1. number of parameters
2. datatype of parameters
3. order of parameter with different datatype.

It is (compile time/ early binding /apparent/not true) **polymorphism**

When an object is created, the compiler by default calls the constructor, or creates one that initializes the attribute with garbage values, if there is no one already exist. However, when a developer creates a constructor in a datatype, it overrides the compiler's constructor and is known as an overwrite constructor. So When an object is created, the compiler by default

calls the constructor and if the parameter less than the developer ‘s constructor parameter, it will be got compiler error.

Constructor Overloading

* if there is overloading there must be overriding

It is two or more constructure with the same name and different signature it is used when creating objects with different initializations.

It allows for greater flexibility and convenience when creating objects, as it allows you to provide different ways to initialize the object's state depending on the needs of the application.

It is (compile time/ early binding /apparent/not true) **polymorphism**

Constructor chaining

* if there is chaining there must be overloading and overriding

It is two or more constructure with the same name and one of them call the other that is used when you have multiple constructors with different sets of parameters, but some of the initialization tasks are common to all of them. instead of write the same code for all constructor, it can be written in one and the others ‘ll call it .

Copy Constructor

It is a special constructor that creates a new object by copying an existing object of the same type.
It is used when an object is copied **by value to other object, passed by value as a function argument**, or **returned by value from a function**.

The default copy constructor provided by the compiler performs a **shallow copy**, which means that it copies the values of the member variables of the object being copied. However, if the object being copied contains pointers or other dynamically allocated resources, **a deep copy** may be necessary to ensure that the new object has its own copy of the resources.

A copy constructor can be defined **explicitly** by the programmer to customize the copying behavior of the object. This can be useful in situations where the default copy constructor is insufficient or where a deep copy is required

Adv.

Readability: it makes code more readable by initialize code in one line

Reusability: by allowing objects to be created and initialized in a single step. This can save time and reduce the amount of code that needs to be written. Additionally, constructors can be overloaded to allow objects to be created with different sets of parameters, further increasing their reusability.

Maintainability /extensibility: by can adjust / add implementation of constructor (initialize value) without effect on other parts of the program.

How to do it?

```
Struct Employee{
Private :
int id ;
string name;
public :
employee() :employee (0,””){} // constructor overwrite  and chaining

employee(int _id,string name) //constuctor loading
{
Id=_id;
name=_name;
}
}
```

Destructor:

it is a special method/function that is automatically called when an object is destroyed or deleted.it has the same name of datatype.

Its main purpose is to execute a certain code or perform any necessary cleanup tasks that may be required before the memory occupied by the object (it is not prevent destroying object but it is execute a wish before destroying).

Ex

```
~employee {
```

cout<<” I’m dying”;

}

Action/Function or method

It way to organize code in one place it can be called wherevers it needs in the program.

Function overloading

It is (compile time/ early binding /apparent/not true) **polymorphism**
It is two or more function with the same name and different signature and similar functionality it is used when creating objects with different initializations.

It defines multiple functions with the same name in a program, each of which performs a different task based on the input arguments provided.

When a function is called, the compiler determines which version of the function to use based on its signature. This allows you to write more concise and readable code by using a single function name for similar operations, rather than having to create multiple functions with different names. ** **Readability.**

Reusability: you can reuse the same function name in different parts of the program with different parameter types. This can reduce code duplication and make the code more modular.

Overloading functions can make the code more **extensible** by allowing you to add new functions with the same name and different parameter types.

If you need to modify a function, you can do so without affecting other functions with the same name. This means you can make changes to one function without worrying about breaking other parts of the codebase as well as easy access to it ** **maintainability**

Ex

Sum (int x, int y){}
Sum (float x, int y){}
Sum (int x, int y, int z){}

Default parameter

A default parameter is a value that is automatically assigned to a function parameter if the caller of the function does not provide a value for that parameter.

ex

```
int Sum (int x=1,int y=2){return x+y;}
int main(){
int z=sum(2,3) ;// z=5
int s=sum() ;// s=3
return 0;
}
```

Static

It is keyword that used to make a member that is associated with a datatype rather than with an instance/object of that data type.

Instance (Non static) action can access both of static and instance attribute but static action can access only static attribute. (It is logic because static attribute is part of each object while the reverse is not true) . ex:

```
static int count; // static attribute
static void resetCount{count=0;} // static action
```

Adv

It keep the encapsulation concept of access attribute by actions only or child datatype while attribute can related to datatype not to instance (its value is updated with each instance but can use of previous instance)

Dynamic memory allocation implementation in c++

To create variable in heap, use new keyword.

```
Int * x=new int(size);
```

To release variable in heap ,use delete method.

```
delete(x);
```

Template

It is a prototype for creating objects. It defines the properties and behaviors that objects of a certain type will have.

It allows developers to define a datatype that can be used with different datatype of attributes without having to rewrite the code for each specific type.

Advantage

Readability: Developers can quickly understand an object's capabilities and intended uses by providing generic datatype of attributes.

Reusability: With the aid of templates, programmers can provide a set of attributes that can be applied to a variety of objects, each of which may have a different data type.(one structure ca work at any datatype of attribute)

Maintainability and extensibility: By promoting good software design principles such as code reuse, modularity, and extensibility, templates make it easier to maintain code over time. Changes can be made to the template without affecting the objects that are created from it, and updates to the template can be propagated to all instances of the object

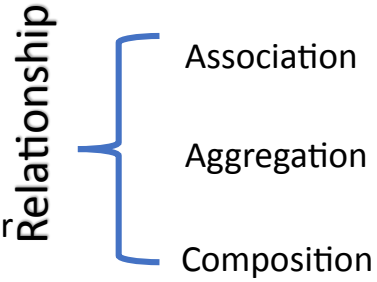
Operators overloading

Operator overloading is a feature in many programming languages that allows operators, such as +, -, *, /, and others, to be used with user-defined types. By overloading an operator, you can define what it means to apply that operator to instances of your own datatype.

For example, if you have a datatype representing a complex number, you might want to define what it means to add two instances of that datatype together using the + operator. By overloading the + operator, you can provide a specific implementation for this operation.

Operator overloading is a powerful **feature** that can make your code more expressive and easier to read, but it should be used judiciously. Overloading

operators can lead to unexpected behavior if not implemented carefully, so it's important to follow best practices and ensure that your code is well-tested.



Relationship between objects:

Association: (No ownership/Peer to peer)

It defines a situation in which one object is connected to another and might require interaction with it in some manner in order to carry out its tasks. However, since each of them is independent of the others, their deaths are not related. Nothing is owned.

Example: teacher and student relationship

How it implement

By adding attribute of datatype into other

```
struct Teacher {
private :
int id;
string name;
public :
Teacher(){}
};
```

```
struct Student{
 private :
  int id;
  string name;
   Teacher T1;
 public :
     Student(){}
      };
```

Aggregation (little ownership)

Aggregation is a relationship between two datatypes where one datatype holds a reference to another datatype as a member variable. (This implies that one datatype controls and is in charge of the management of the other datatype, but both of datatypes is independent if one is destroying the other will not destroy)

Example: teacher and department relationship

How it implement

By adding attribute of datatype into other

```
struct Teacher {
private :
int id;
string name;
public :
Teacher(){}
};
```

```
struct Department{
 private :
  int id;
  string name;
   Teacher T1;
 public :
     Department(){}

};
```

Composition (Full ownership)

In this relationship between two datatypes, one datatype is made up of one or more instances of the other datatype. As a result, the constructed datatypes is a dependency of the composite datatype and is a subset of it.

Although both composition and aggregation involve relationships between datatypes, the contained datatype in composition is dependent on the container datatype and cannot exist independently. In other words, the container datatype and the enclosed datatype both construct and destroy itself.

Example : car and engine

How it implement

By adding attribute of datatype into other

```
struct Engine {
private:
int id;
int RPM ;
```

```
struct Car{
 private :
  int id;
string model;
```

```
public :
Engine(){}
};
```

```
Engine * E1;
public :
    Car (){}
    };
```

Inheritance:

inheritance is a mechanism that allows a new class to be based on an existing class, inheriting its attributes and behaviors. The existing class is called the superclass or parent class, while the new class is called the subclass or child class. It can inherent the parent without any addition or add some members or modifies of action of parent.

Advantage

Inheritance enables code reuse and allows you to create a hierarchy of classes with shared attributes and behaviors. The subclass inherits all the public and protected attributes and methods of its parent class, and can add new attributes and methods or override existing ones.

Example

```
#include <iostream>
using namespace std;
// base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }
  protected:
    int width;
    int height;
};
```

```
// derived class
class Rectangle: Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main() {
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

Note

You can use constructor of child in creation a parent object because parent is part of child but not vice versa (slicing)

Example

Person P=Teacher(5,”Ahmed”,”1-April-2022”); /*compiler will take id and name and neglect the other attribute */

Every person has id and name but Teacher has other attribute needs initial value

Abstraction (True Abstraction)

Abstracted function

Its prototyping in the parent datatype and its implementation in the child datatype. they are used when it is necessary to implement a function based

on a child's relationship to the parent (generalization and protection of an object created by the parent).

Example:

```
#include <iostream>
using namespace std;
// Abstract class
struct Shape {
  public:
    // Pure virtual function
    virtual void draw() = 0; // "= 0" indicates that this function is pure virtual

    // Normal member function
    void display() {
      cout << "This is a shape." << endl;
    }
};

// Derived class
struct Circle: Shape {
  public:
    // Implementation of the draw function
    void draw() {
      cout << "Drawing a circle." << endl;
    }
};
struct Sequare: Shape {
  public:
    // Implementation of the draw function
    void draw() {
      cout << "Drawing a Sequare." << endl;
    }
};
```

```
int main() {
  Circle c;
  c.display(); // Call to Shape class member function
  c.draw();    // Call to Circle class member function

  return 0;
}
```

Polymorphism (true polymorphism)

when standalone function called abstracted function with object of parent.

When it is needs to implement a single function that will carry out the same action for various datatypes that are connected to one another (all of them have a common parent), this technique is utilized.

example

```
Void drawing Shape(shape*S){
S.draw() ; /* at this line complier doesn't know which draw will implement of circle or sequare (Dynamic/Runtime/Lead binding) */
}
```

Class Vs structure

	Class	structure
Default access of attribute	Private	Public
Inheritance mode of attribute	Private	Public