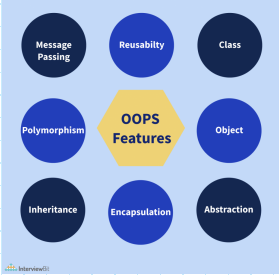
# C++

31 May 2023 13:00

1. Mostly similar to C but differs largely since its an OOP rather than a POP




2. **How different OOP objectives are achieved in C++:**
    a. **Encapsulation: classes and objects**
    b. **Abstaraction: ADTs (Abstraction Data Types)**
    c. **Polymorphism: function, operator overloading, dyanamic binding (virtual function)**
    d. **Single Inheritance**
    e. **Multiple Inheritance**
    f. **Dlegation: making object composition**
    g. **Genericity: function templates, class templates**
    h. **C++ does not support Persistence**

1.

| **C++ specific keywords** | asm | new | this | template | catch | virtual | friend | delete | new |
|---|---|---|---|---|---|---|---|---|---|
| operator | protected | private | inline | public | **`nullptr`** | | | | |

2. **C++ also supports C style strings**

```
char c[100] = "O mighty Smaug, O great tyrrant, truly majestic.";
```

**or better and more intuitive way**

```
std::string str = "My wings are a hurricane, my breath is fire and I AM DEATH.";
```

3. **STRUCTS-> with function overloading applied**

```cpp
#include <iostream>
using namespace std;

struct date{
    int date;
    int month;
    int year;

    void show();            //shows date of object
    void show(int date, int month, int year); //different date
};

void date::show(int a, int b, int c){
    cout<<"\n"<<a<<"-"<<b<<"-"<<c<<endl;
}

void date::show(){
    std::cout<<"\n"<<date<<"-"<<month<<"-"<<year<<endl;
}


int main(){
    date d1 = {27,11,2003};
    date d2 = {26,7,2007};

    d1.show();       //shows 27-11-2003
    d2.show(1,1,0); //shows 1-1-0 instead of 26-7-2007
    return 0;
}
```

4. **Unions in C++: support functions as well**

```cpp
union Shakespeare_play
{
    char narration[1000];
    char dialogue[50000];
    void narrator_speak();
    void dialogue_speak();
};
```

5. **new and delete operators: to alloacate memory dynamically (when script is running) on the heap memory**

```
int * ptr = new int;
int * arr = new int[4];     //an space of 4 integers allocated
```

```
//we can also initialize the variables during allocation
int * ptr = new int(600);

//freeing the memory using delete
delete ptr;
```

6. **CLASSES -> defined with constructors and destructors.** Constructors are special functions that are automatically invoked at the time of creation of an object. They have the same name as its class. Destructors are declared by plpacing a tilde (~) in front of the classname. Destructirs are invoked just before the block ends. Constructors can be overloaded, but destructors cannot be. Yes, classes are very similar to structs. Data members of classes are private by default. Data members of structs are public by default.

```
class deep_sky_stars{
    private:
        int azimuth;
        float luminosity;
        float dayperiod;

    public:

    //constructors, overloaded
        deep_sky_stars()
        {   azimuth = 0;    luminosity = 0.0;   dayperiod = 0.0;   }

        deep_sky_stars(int az, float lum, float day)
        {   azimuth = az;   luminosity = lum;   dayperiod = day;}


        void printinfo()
        {   cout<<"Azimuth = "<<azimuth<<endl;
            cout<<"luminosity = "<<luminosity<<endl;
            cout<<"Dayperiod = "<<dayperiod<<endl;}

    //destructor, cannot be overloaded
        ~deep_sky_stars();

};
```

- **In local blocks, Local variables are prioritized over global and class variables. To override this behaviour, either use scope resolution operator :: or use 'this' keyword.**
- **member functions can be defined within the class body (declared inline, if inside) or outside (not declared inline by default)**. To define outside first declare prototype inside the body and then use scope resolution opertaor :: to define outside like so,

```
void deep_sky_stars:: printgraph(){
    blah blah blah
    }
```

And it is ususally recommended to do so

- **Copy Constructors:** constructors which creates an object by initializing it with an object of the same class, previously created. In its parameters, it takes the reference to an object.
Obviously, since it is a constructor, it needs to have the same name as the class. Not only that, you might be surprised to know that copy constructors are automatically created if not defined by us (user).

```
/*-----------------------copy constructor demo-----------------------*/
#include <iostream>
#include <string.h>
using namespace std;
class acc{
    char name[10];
    char bio[100];
    int followers;
    public:

        acc(char* namein, char* bioin){
            strcpy(name,namein);
            strcpy(bio, bioin);
            followers=0;
        }
        //        COPY CONSTRUCTOR!!!!
        acc(acc & obj){
            strcpy(name, obj.name);
            strcpy(bio, obj.bio);
            followers = obj.followers;
        }

        void show(){
            cout<<"Name = "<<name<<endl;
            cout<<"Bio = "<<bio<<endl;
            cout<<"Followers = "<<followers<<endl;
        }

};
int main(){
    acc Amy("Amy", "Hi , I am Amy and I am studying Philosophy at the University of Oxford!");
    Amy.show();

    //copy constructor called below
    acc Bree=Amy;   //or acc Bree(Amy);
    Bree.show();
return 0;
}
```

- **Friend Classes and friend functions:** C++ friend class is special and can access even the private data members and functions of other classes.

```
/*Can a member function access the private data of its own objects?
```

```cpp
/*----------------DEMONSTRATING FRIEND FUNCTIONS----------------*/
#include <iostream>

class ball{
    private:
        int radius;
    public:

        //friend void print_radius(ball b);

        char color;
        ball(int x, char c){
            radius = x; //initialize radius
            color = c;  //initialize color
        }
        void add_radius(ball obj){
            //accessing the private members of another class- possible
            radius += obj.radius;
        }

};

void print_radius(ball b){
    //Erorr- a non-member function cannot access private members 'radius' of 'ball'
    std::cout<<"The radius of the ball = "<<b.radius;
    //The way to rectify this is to declare this function as a friend to ball class inside class body
    /*   friend ball print_radius(ball b);   */

    //although public data member 'color' can be accessed
    std::cout<<"The color is = "<<b.color;
}

int main(){
    ball b1(1,'g'),b2(7,'b');
    b1.add_radius(b2);  //OK
    print_radius(b1);
    return 0;
```

```cpp
/*----------------------------FRIEND CLASSES-------------------*/
#include <iostream>

//forward declaration of class
class Cockpit;
class Pilot;

class Cockpit{
    private:
        short controlacccess;
    public:
        void grantaccess(Pilot p);
};

class Pilot{
    private:
        short canfly;
    public:
        //friend class Cockpit; can also be declared which will
        //give access to every cockpit function
        friend void Cockpit::grantaccess(Pilot p);
};

int main(){
    Pilot Rohit;
    Cockpit Airbus;

}

void Cockpit::grantaccess(Pilot p){
    if(p.canfly == 1)
        controlacccess == 1;
}
```

- **Nested Classes:**

```cpp
/*----------------------NESTED CLASSES DEMO-----------------------*/
#include <iostream>
#include <string.h>

enum what {PHOTO, VIDEO, GIF, TEXT};
class webpage{

    private:

        char URL[20];
        char cookies[500];

    public:

    //nested class for media inserted into the page like photos, gifs, videos
        class media{
            public:
                char path[20];
                what t;

        } m0, m1;   //m0, m1 are objects of media

    webpage(char* url){
```

```
    webpage(char * url){
        strcpy(URL, url);
    };
    void display();
    void close();
    ~webpage();
};

int main(){
    webpage page1("www.foodblog.com\\sweet\\rasgolla");

    strcpy(page1.m0.path,"\\home\\monkey.jpg");      //page1.media.path will be wrong
    page1.m0.t = PHOTO;
return 0;
}
```

- **Empty class: classes can be left empty too, just in case**
  ***Class webpage{};***

- **Arrow operator:** It is used to access the public members of a class, structure, or members of union with the help of a pointer variable.
  //Dog is a class
  Dog * Tommy;
  Tommy->bark();

- Static data members and member functions of a class

- Constant data members and member functions of a class

7. **OBJECTS** can be created from classes like so

*deep_sky_stars alpha_centauri;* *//automatically calls the constructor*
*deep_sky_stars proxima_centauri(27, 0.0017, 11.2);*

**for reference, Alpha Centauri and Proxima Centauri are actual stars**

- objects can be passed to and returned from functions, constructors:

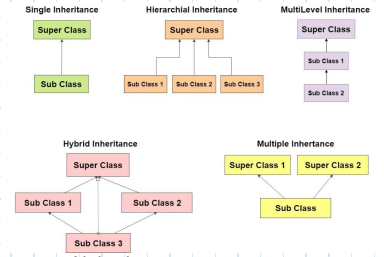
- Array of objects: suppose className is Dog, and member function bark()

```
Dog obj[5];
Obj[0].bark();   //first dog barks
Obj[1].bark();    //second dog barks

//or using new
Dog* ptr= new Dog[5];
ptr->bark();   //first dog barks
ptr++;
ptr->bark();   //second dog barks
ptr++;

//or just by using a pointer array
Dog * ptr[5];
ptr->bark();
ptr++;
ptr->bark();
```

- **'this'** keyword is a pointer that points to the object invoking the member function

```
/*-------------------------use of 'this' keyword---------------*/
#include <iostream>
using namespace std;
class Dog{
    public:
        int dog_number;
        void bark(){
            cout<<this->dog_number<<" arragharaghargh..."<<endl;
        }
        Dog get_dog_copy(){
            return *this;
        }
};

int main(){

    Dog Tommy;
    Tommy.dog_number=10;
    Tommy.bark();

    Dog Golu;
    Golu = Tommy.get_dog_copy();

return 0;
```

8. **Operator Overloading:** It allows us to provide an intuitive interface to our class users, plus makes it possible for templates to work equally well with classes and built-in types. All arithmetic(+ - * /), logical(&& || !), relational( <= !=), assignment( = += ), bitwise( ^ >>) operators can be overloaded. Even 'new' and 'delete' can be overloaded.



Types of inheritance

```cpp
#include <iostream>

class Index{
    int val;
    public:
    Index(){
        val=0;
    }
        Index(int n){
            val=n;
        }
        //note return type
        void operator ++(){
            val++;
        }
        //note return type
        Index operator +(Index &obj){
            Index t;
            t.val = val + obj.val;
            return t;
        }
        void show(){
            std::cout<<"\nval = "<<val<<"\n";
        }
};

int main(){

    Index i1(1), i2(3), i3;

    ++i2;
    i2.show();
    i3 = i2 + i1;
    i3.show();

    return 0;
}
```

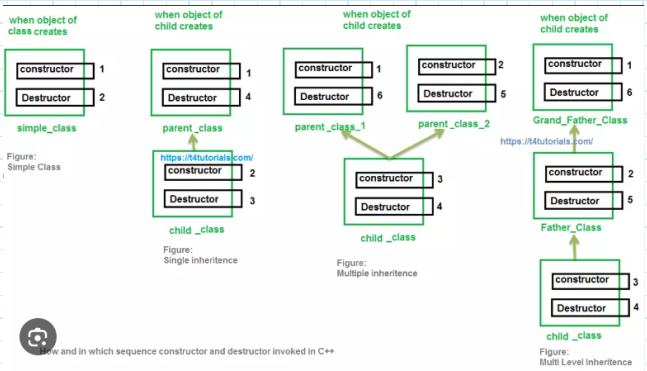
9. **Inheritance: the technique of building new classes from existing ones**
   Derived class declaration:
   *Class DerivedClass : [visibility mode] BaseClass*

```cpp
/*--------------------------------Single Inheritance demo----------------------*/
class B
{
    private:            //accessible only within the class
        int bpriv;
        void f1();
    protected:          //accessible within the class and within derived class
        int bpro;
        void f2();
    public:             //accessible within the class, derived class and through objbects
        int bpub;
        void f3();


};

class Derived: public B{};
```

**Visibility modes: public, private (default)**

| Base Class Visibility | Derived class inheriting publicly | Derived class inheriting privately |
|---|---|---|
| private | Not inherited | Not inherited |
| protected | protected | private |
| public | public | private |

- A Derived class inherits data members and member functions, but not the constructor or destructor from its base class.
- If a class is expected to be used as a base class then its members must be declared protected rather than private.
- Consructors of base class and derived class are automatically invoked when the derived class is initiated. The Derived class nedd not have a constructor as long as base has a no-argument constructor. If a base class has constructors with arguments then it is mandatory for the derived class to have a constructor and pass the arguments to the base class. However the no argument constructor need not be invoked explicitly. Remember constructors must be defined in the public section of class.
- Base ka constructor pehle run hoga fir derived ka. If there is a virtual base class then its constructors are called before every other constructor




```
Case2:
class A: public B, public C{
    // Order of execution of constructor -> B() then C() and A()
};

Case3:
class A: public B, virtual public C{
    // Order of execution of constructor -> C() then B() and A()
};
```

```
class Derived: public Base1, public Base2{
    int derived1, derived2;
    public:
        Derived(int a, int b, int c, int d): Base1(a), Base2(b){
            derived1 = c;
            derived2 = d;
            cout<< "Derived class constructor called"<<endl;
        }
        void printData(void)
        {
            cout << "The value of derived1 is " << derived1 << endl;
            cout << "The value of derived2 is " << derived2 << endl;
        }
};
```

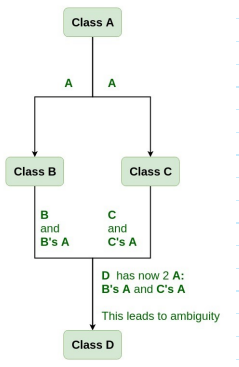
- **Member function accessibility**

| Function Type | Access directly to | | |
|---|---|---|---|
| | Private | Protected | Public |
| Class Member | Yes | Yes | Yes |
| Derived class member | No | Yes | Yes |
| Friend | Yes | Yes | Yes |
| Friend class member | Yes | Yes | Yes |

- **Ambiguity resolution in Inheritance:** Base class members with same name create ambiguity. How to resolve? If for example, base class has function joker() and derived class also has a its own joker() then calling derived.joker() will override Base.joker(). Otherwise if you want to call joker():

```
DerivedClass obj;
obj.joker();              //calls derived class joker()
obj.baseClass :: joker();   //calls base class joker
```

- **Virtual Classes:** Virtual base classes in C++ are used to prevent multiple instances of a given class from appearing in an inheritance hierarchy when using multiple inheritances.




```
/*---------virtuall class----------*/
```

```cpp
#include <iostream>
using namespace std;

class father{
    protected:
        int a;
};

class son1: virtual public father
{};

class son2: virtual public father
{};

class grandchild: public son1, public son2
{};
```

- **VIRTUAL FUNCTIONS:**
  https://www.codewithharry.com/videos/cpp-tutorials-in-hindi-57/
  1. They Cannot be static
  2. They are acccesed by object pointers
  3. They can be a friend of another class
  4. A virtual function in base class might not be used
  5. If a virtual function defined in the base class, there is no necessity of redefining in the derived class(that is if function is not defined in derived class base class function is called by defualt)
  6. **ABSTRACT BASE CLASS:** An abstract class is a class that is designed to be specifically used as a base class. **An abstract class contains at least one pure virtual function.** You declare a pure virtual function by using a pure specifier ( = 0 ) in the declaration of a virtual member function in the class declaration. **Pure virtual function:** (open text file below)

a. virtual-functions

```cpp
/*--------------pointer to derived class-----------------------
    polymorphism:
        1. compile-time:
            a. function-overloading
            b. operator overloading
        2. run time polymorphism:
            a. virtual functions
            */
#include <iostream>
using namespace std;
class base{
    public:
    int a;
    void show(){
        cout<<"in base: a = "<<a<<endl;
    }
};
class der : public base{
    public:
    int b;
    void show(){
        cout<<"in derived a = "<<a<<endl;
        cout<<"in derived b = "<<b<<endl;
    }
};
int main(){
    der obj_der, *der_pt;
    base *base_pt;
    base_pt = &obj_der;   //pointing base class pointer to derived class
    base_pt->a = 100;    //but it can only access base class members base::a, base::show()
    base_pt->show();     //also binding actually takes place duting runtime

    der_pt = &obj_der;    //pointing derived class pointer to derived class
    der_pt->a = 666;     //and it can access both derived and base members
    der_pt->b = 20;
    der_pt->show();


return 0;
}
```

```cpp
/*--------------pointer to derived class-----------------------
    polymorphism:
        1. compile-time:
            a. function-overloading
            b. operator overloading
        2. run time polymorphism:
            a. virtual functions
            */
#include <iostream>
using namespace std;
class base{
```

```
    public:
    int a;
    void show(){
        cout<<"in base: a = "<<a<<endl;
    }
};
class der : public base{
    public:
    int b;
    void show(){
        cout<<"in derived a = "<<a<<endl;
        cout<<"in derived b = "<<b<<endl;
    }
};
int main(){
    der obj_der, *der_pt;
    base *base_pt;
    base_pt = &obj_der;   //pointing base class pointer to derived class
    base_pt->a = 100;    //but it can only access base class members base::a, base::show()
    base_pt->show();     //also binding actually takes place duting runtime

    der_pt = &obj_der;    //pointing derived class pointer to derived class
    der_pt->a = 666;     //and it can access both derived and base members
    der_pt->b = 20;
    der_pt->show();


return 0;
}
```

8. **Streams Computation:**

```
/*--------------------------I/O Streams-----------------------*/
/*  2 ways to read, write files
1. Using constructors
2.Using open() function
In this script-> using constructors*/
#include <iostream> //for console io
#include <fstream>  //for file io
using namespace std;
int main()
{

    // writing to file
    ofstream out("shakespeare.txt");  // ofstream object
    out << "My namei is oreiki";      // write first line
    out << "\nAnd my name is Durin."; // write second line
    out.close();                      // close 'out' object

    // reading from file
    string b;
    ifstream in("shakespeare.txt"); // ifstream object
    getline(in, b);               // read line 1
    // in>>a;
    cout << b<<endl;
    getline(in, b);                // read line 2
    cout << b<<endl;
    in.close(); // close 'in' object
    return 0;
}
```

```
/*In this script using open()*/
#include <iostream>                    //for console io
#include <fstream>                     //for file io
using namespace std;
int main(){

    //writing to file
    string s="i love snow!";
    ofstream out;
    out.open("shakespeare.txt");
    out<<s;
    out.close();

    //reading
    ifstream in;
    in.open("shakespeare.txt");
    getline(in,s);
    cout<<s;

    return 0;
}
```

**here's the code for printing an entire file:**

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void main(){
        string s;
```

```
ifstream in("logs.txt");
while(in.eof()!=0)
{
        getline(in,s);
        cout<<s;
}
}
```

- **File open modes using open function,**
  File = open("log.txt", ios::app);

| Sr. No | Open mode | Description |
|---|---|---|
| 1 | in | Open for reading |
| 2 | out | Open for writing |
| 3 | ate | Seek to end of file upon original open |
| 4 | app | Append mode |
| 5 | trunc | Truncate file if already exists |
| 6 | nocreate | Open fails if file does not exists |
| 7 | noreplace | Open fails if file already exists |
| 8 | binary | Opens file as binary |

9. **Generic programming with templates:**

- **Function templates**
  **Syntax:**

```
template <class T1, class T2...>
ReturnType funcname (arguments with atleast one template type)
{
//body
}
```

**Template arguments (passed) can also be user defined such as struct, or unions.**

```
#include <iostream>
template <class T>
int swap(T& x, T& y){
    T temp;
    temp=x;
    x=y;
    y=temp;
    return 0;
}

int main(){
    int a=1,b=2;
    swap(a,b);
    std::cout<<"\nThe numbers after swapping are "<<a<<" and "<<b;
    return 0;
}
```

- **Overloaded template function:**

```
/*overloaded template function*/

#include <iostream>
using namespace std;

template <class T>
void print(T t){
    cout<<t<<endl;
}

template <class T>
void print(T t, int nTimes){
    for(int i=0; i<nTimes; i++)
        cout<<t<<endl;
}

int main(){
    print('/');
    print('*', 20);
return 0;
}
```

- **Class templates**

```
/*-------------class tempates-----------------*/
```

```cpp
#include <iostream>
using namespace std;

template <class T>
class vector{
    int size;
    T * t;
    public:
        vector(int size){
            t = new t[size];
        }
        void getvector();
        void setvector();
        ~vector(){
            delete t;
        }
};
int main(){
    vector <int> V(10);
return 0;
}
```

- **Inheritance of a class template**

10. **STL= Standard Template library (Generic Classes and functions)**
    STL OFFERS Containers, Algorithms, Iterators (Iterators are handled just like pointers, connects algorithm with containers, movies as wanted ).

    Containers can be of 3 types (sequence = vectors, associative = set/multiset map/multimap, derived- real world modelling)

| vectors | strings | maps | |
|---|---|---|---|
| stl_cheatsheet_1 | cpp_strings | stl_cheatsheet_2 | |