OOPS!!!

IF YOU WANT TO READ A BOOK THAT HAS NOT WRITTEN YET, YOU MUST WRITE IT

The sheet provides a quick overview of the OOP concepts in few seconds, preferred language is c++

Content

Constructors & Destructors

constructor

- Initialize a class
- Automatically called when object is created
- Used to initialize the variables of the class
- Does not return anything (no need to specify anything)
- User can create multiple definition of constructors, based on the inputs it will decide which constructor to call

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;

    public:

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    coordinate (int a) {
        x = a;
        y = 0;
    }
```

```
    coordinate () {
        x = 0;
        y = 0;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    coordinate * point1 = new coordinate(3,4);
    coordinate * point2 = new coordinate(2);
    coordinate * point3 = new coordinate();

    pair<int,int> par1, par2, par3;
    par1 = point1->getpoint();
    par2 = point2->getpoint();
    par3 = point3->getpoint();

    print_pair(par1);
    print_pair(par2);
    print_pair(par3);

    // output:
    // 3 4
    // 2 0
    // 0 0


    return 0;
}
```

copy constructor

- Used to create exact copy of an object (copy one object to another)
- automatically called when object is created using copy constructor
- If the custom implementation is not found in class, then the compiler provide it's own default copy constructor

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:
```

```
    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    coordinate(coordinate * &new_obj) {
        cout << "calling cc" << endl;
        x = new_obj->x;
        y = new_obj->y;
    }

    coordinate(coordinate &new_obj) {
        cout << "calling cc obj" << endl;
        x = new_obj.x;
        y = new_obj.y;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    coordinate * point1 = new coordinate(3,4);
    coordinate * point2 = new coordinate(point1); // copy constructor is
called


    print_pair(point1->getpoint());
    print_pair(point2->getpoint());

    // # brain storming
    coordinate * point3 = new coordinate();
    point3 = point1; // copy constructor will not be called for assignment

    coordinate * point4 = point1; // copy constructor will not be called

    coordinate point5 = *point1; // copy constructor will be called

    // calling cc
    // 3 4
    // 3 4
    // calling cc obj

    return 0;
}
```

Remember

- The default copy constructor does the shallow copy
- The user defined copy constructor des the deep copy
- The copy constructor is called at the time of assignment as well (see above snippit)

Destructor

- Used to destroy memory, space taken by the class object

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    ~coordinate () {
        cout << "destructor is called" << endl;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    coordinate * point1 = new coordinate(3,4);
    print_pair(point1->getpoint());

    coordinate point2(4,5);
    print_pair(point2.getpoint());

    delete point1;

    // 3 4
    // 4 5
    // destructor is called
```

```
    // destructor is called

    return 0;
}
```

Remember

- free(pointer) -> does not call the destructor
- delete pointer -> calles the destructor
- for normal case (not a pointer) the destructor is called automatically

Access Modifiers

- C++ offers Three type of access modifiers
- Public
  : Accessable by other classes and objects
- Private
  : Not accessable by anyone (except friend function)
- Protected
  : Not accessable outside class (but childs can access the functions/variables)
- By default the access is **Private**

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    private:

    int x,y;

    public:

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }

    protected:

    pair<int,int> addtopoint(int val) {
        return pair<int,int>(x+val,y+val);
    }
```

```
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    coordinate * point = new coordinate(3,4);

    print_pair(point->getpoint());

    // output:
    // 3 4

    return 0;
}
```

Remember

- **Encapsulation** means binding the data only with the functions that must modify them
- For that Restrict the access to other users (Use access modifiers)
- Encapsulation also leads to abstraction or data hiding etc.

Inheritance

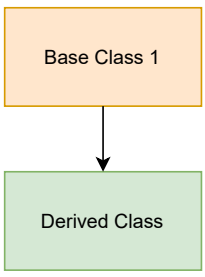
- Inheritance is deriving the properties of one class to another class
- The class which is deriving the property of another class is called as derived / child class
- The class which is derived is known as parent / base class

Not all functions are Inherited

- constructor and destructor don't inherit
- operator= function don't inherit
- friend functions don't inherit

Types of Inheritance

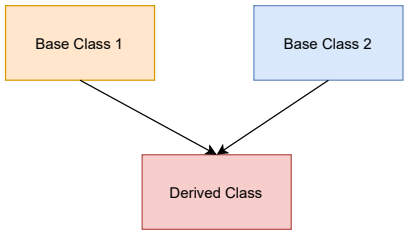
- Single Inheritance



```
#include <bits/stdc++.h>

using namespace std;

class base {
    int x;
    public:

        base() { x = 0; }

        base(int x) { this->x = x; }

        void show() {
            cout << x << endl;
        }

    protected:

        void set(int y) { x = y; }
};

class derived: public base {
    public:
        derived(int x): base(x) {}
        void show_x() { show(); }
};

int main() {
    derived * der = new derived(3);
    der->show_x();

    return 0;
}
```

- Multiple Inheritance
- In multiple inheritance the order of execution of constuctor is the order of declaration of the class
- for example if the inheritance is as follows (A → C) and (B → C) then the order of constructor calling is A then B

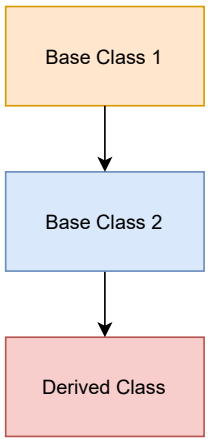
```
class A;
class B;
class C: public A, public B {
    C(): A(), B() {} // order of execution of constructor is A then B
    C(): B(), A() {} // order of execution remains the same as the
declaration is (public A, public B) so first A will be called and then B,
the order does not matter
}
```




```
#include <bits/stdc++.h>

using namespace std;

class base1 {
    int x;
    public:

        base1() { x = 0; }

        base1(int x) { this->x = x; }

        void show() {
            cout << x << endl;
        }

    protected:

        void set(int y) { x = y; }
};

class base2 {
    int x;
    public:

        base2() { x = 0; }

        base2(int x) { this->x = x; }

        void show() {
            cout << x << endl;
        }

    protected:

        void set(int y) { x = y; }
};

class derived: public base1, public base2{
    public:
        derived(int x, int y): base1(x), base2(y) {}
        // void show_xy() { show(); } if we write only this the compiler
got confuse which show to call (base1 or base2)
        void show_xy() {  // to solve above ambiguity we use scope
resolution operator;
```

```
            base1 :: show();
            base2 :: show();
        }
};

int main() {
    derived * der = new derived(3,4);
    der->show_xy();

    // 3
    // 4

    return 0;
}
```

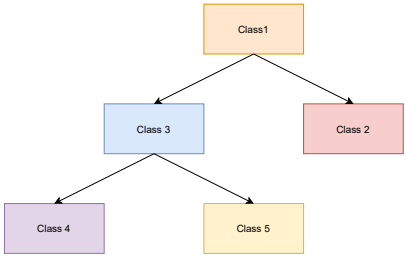
- Multilevel Inheritance
- In multilevel inheritance the constructors are called in the order of the inheritance
- for example inheritance is like this (A → B → C) then the order of calling constructor is (A → B → C)




```
#include <bits/stdc++.h>

using namespace std;

class base1 {
    int x;
    public:

        base1() { x = 0; }

        base1(int x) { this->x = x; }

        void show() {
            cout << x << endl;
        }
```

```
    protected:

        void set(int y) { x = y; }
};

class base2: public base1 {
    int z;
    public:

        base2() { z = 0;}

        base2(int z, int x): base1(x) { this->z = z; }

        void show() {
            cout << z << endl;
        }

    protected:

        void set(int y) { z = y; }
};

class derived: public base2{
    public:
        derived(int x, int y): base2(x,y) {}
        void show_xy() { // again use resolution operator to call
functions from different classes
            show(); // call base2 class show
            base1 :: show(); // call base1 class show

        }
};

int main() {
    derived * der = new derived(3,4);
    der->show_xy();

    // 3
    // 4

    return 0;
}
```

- Hierarchical Inheritance

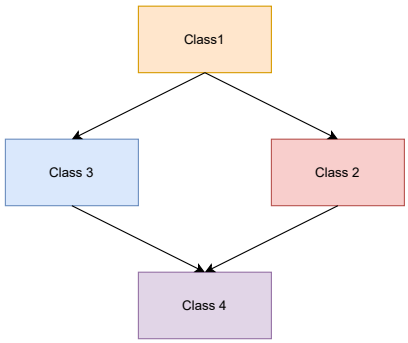


```
can be easily implemented using above concepts
```

- Hybrid Inheritance
- the virtual base class always has higher preference in calling the constructors

```
class A;
class B;
class C: public A, virtual public B {
    C(): A(), B() {} // order of execution of constructor is B then A
    C(): B(), A() {} // order of execution remains the same as B is
declared virtual base class which is always preferred
}
```

- If there are more virtual base classes then their order would decide the constructor calling




```cpp
#include <bits/stdc++.h>

using namespace std;

class base1 {
    public:
        void speak() {
            cout << "base 1 is speaking" << endl;
        }
};

class base2: virtual public base1 {
    public:
        void tell() {
            cout << "base2 is telling" << endl;
        }

        void speak() { // now speak will call this speak function
            cout << "base2 is speaking" << endl;
        }
};
```

```
class base3: virtual public base1 {
    public:
        void tell() {
            cout << "base3 is telling" << endl;
        }
};

class base4:public base2, public base3 {
    public:
        void chill() {
            cout << "base4 is chilling" << endl;
        }
        void tell () {
            base2:: tell();
        }
};


int main () {
    base4 * bs4 = new base4();
    bs4->speak();
    return 0;
}
```

Inheritance Access specifier Table

- The below table shows the access of the functions / variables into derived classes.
- for example if the base class is inherited private then all the things are private members of derived class
- the private members are still accessable inside the derived class

Base Class \ Inherited class	Public	Private	Protected
Public	public	private	protected
Private	private	private	private
Protected	protected	private	protected

Polymorphism

- polymorphism is one thing taking different forms, for example one function can take different forms depending on the type and number of arguments etc.
- It is of different type
    - Compile time
        - Function overloading
        - Operator overloading

    - Runtime
        - Virtual functions
        - Function overriding

Function overloading

- function with same name but different types and number of arguments
- functions are overloaded not on the basis of their return type but on the basis of their number of arguments and type of arguments

Remember

- The key to function overloading is a function's argument list
- This is also known as the function signature/ extended name.
- It is the signature, not the function type that enables function overloading.

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }

    pair<int,int> getpoint(int p) {
        return pair<int,int>(x+p,y+p);
    }

    pair<int,int> getpoint(int p, int q) {
        return pair<int,int>(x+p,y+q);
    }
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    coordinate * point1 = new coordinate(3,4);
    print_pair(point1->getpoint());
    print_pair(point1->getpoint(2));
```

```
    print_pair(point1->getpoint(3,4));

    // output:
    // 3 4
    // 5 6
    // 6 8


    return 0;
}
```

Remember

- Static functions cannot be overloaded

Operator overloading

- same operator with different operations

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    pair<int,int> setdata(int a, int b) {
        x = a;
        y = b;
        return pair<int,int>(x,y);
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }
};

void print_pair(pair<int,int> p1) {
    cout << p1.first << " " << p1.second << endl;
}

coordinate * operator+(coordinate &coor1, coordinate &coor2) {
    pair<int,int> p1 = coor1.getpoint();
```

```
    pair<int,int> p2 = coor2.getpoint();
    coordinate * coor3 = new coordinate(p1.first + p2.first,
p1.second+p2.second);
    return coor3;
}

coordinate * operator-(coordinate &coor1, coordinate &coor2) {
    pair<int,int> p1 = coor1.getpoint();
    pair<int,int> p2 = coor2.getpoint();
    coordinate * coor3 = new coordinate(p1.first - p2.first, p1.second -
p2.second);
    return coor3;
}

int main() {
    coordinate c1(1,2), c2(4,3);

    coordinate * c3 = c1 + c2;
    print_pair(c3->getpoint());
    coordinate * c4 = c1 - c2;
    print_pair(c4->getpoint());

    //output
    //5 5
    //-3 -1

    return 0;
}
```

Remember

- Ternary (?:) operator cannot be overloaded
- Dot (.) operator cannot be overloaded
- scope resolution (::) operator cannot be overloaded

Virtual Functions

- virtual functions are declared using **virtual** keyword, they are used to achieve

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
```

```
        x = a;
        y = b;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }

    virtual void show_point() { // declare virtual here for runtime
polymorphism
        cout << x << " " << y << endl;
    }
};

class extend_coor: public coordinate {
    public:

        extend_coor (int a, int b): coordinate(a,b) {
            cout << "the constructor is called" << endl;
        }

        void show_point() {
            pair<int,int> p =  getpoint();
            cout << "point: " << p.first << " " << p.second << endl;
        }

};


int main() {

    coordinate * coor;
    extend_coor * excoor = new extend_coor(2,3);

    coor = excoor; // base class pointer is pointing towards derived class
pointer

    // // if not declare virtual
    // coor->show_point(); // runs coor show_point
    // excoor->show_point(); // runs the extended coor show_point

    // if declare virtual
    coor->show_point(); // runs extended coor show_point
    excoor->show_point(); // runs the extended coor show_point

    return 0;
}
```

Function overriding

- Reimplement or override the function (which is in base class) in derived class

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    pair<int,int> setdata(int a, int b) {
        x = a;
        y = b;
        return pair<int,int>(x,y);
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }
};

class extension : public coordinate {
    public:
        extension(int x1, int y1): coordinate(x1,y1) {
            cout << "extension is created" << endl;
        }

        pair<int,int> getpoint() { // overwriting the function in the
derived class
            cout << "getting the coordinates" << endl;
            return coordinate :: getpoint();
        }
};

void print_pair(pair<int,int> par) {
    cout << par.first << " " << par.second << endl;
}

int main() {
    extension * lin = new extension(1,2);
    print_pair(lin->getpoint());

    return 0;
}
```

Abstraction

- Abstraction refers to hiding the details
- achieved using pure virtual functions in c++
- once declared a pure virtual function in base class, the object or pointer of that class cannot be created
- the function which is declared to be pure virtual must be implemented in the derived class

```
#include <bits/stdc++.h>

using namespace std;

class coordinate {
    int x,y;
    public:

    coordinate () { x = 0; y = 0; }

    coordinate (int a, int b) {
        x = a;
        y = b;
    }

    pair<int,int> getpoint() {
        return pair<int,int>(x,y);
    }

    virtual void show_point() = 0;// used for abstraction (now the
show_point function has to be defined in the derived class)
};

class extend_coor: public coordinate {
    public:

        extend_coor (int a, int b): coordinate(a,b) {
            cout << "the constructor is called" << endl;
        }

        void show_point() {
            pair<int,int> p = getpoint();
            cout << p.first << ">>" << p.second << endl;
        }
};


int main() {
    extend_coor * excoor = new extend_coor(2,3);

    excoor->show_point();

    // 2<<3
```

```
    return 0;
}
```

Cool stuff

Initialization list

- a way of initializing variables in the class

```
#include <bits/stdc++.h>

using namespace std;

class base {
    int x,y,z;

    public:
        // base (int a, int b, int c): x(a),y(b),z(c) { // outputs 1 2 3
        //     cout << "constructor called" << endl;
        // }

        // base (int a, int b, int c): x(a),y(b+x),z(c+y) { // outputs 1 3
6
        //     cout << "constructor called" << endl;
        // }

        base (int a, int b, int c): x(a), z(c+x), y(b+z) { // y becomes
garbage because the order of declaring the variables is x, y, z
            cout << "constructor called" << endl;
        }

        void print() {
            cout << x << " " << y << " " << z << endl;
        }
};


int main () {

    base * bs1 = new base(1,2,3);
    bs1->print();

    return 0;
}
```

Friend Function

- Friend function / classes are used to access the private members of one class

- They cannot be inherited to the derived class

```
#include <bits/stdc++.h>

using namespace std;

class base; // forward declaration

class friend_class {
    public:
    void friend_fun(base *, base *);
};

class base {
    friend base * addbase(base *, base *); // declared addbase as friend
function and so addbase can access the private and protected variables of
base class
    friend void friend_class::friend_fun(base *, base *); // declaring the
function from the class as friend
    friend class friend_class; // declaring whole class as friend class;
    int x;
    public:
        base(int x) {
            this->x = x;
        }

        void print() {
            cout << this->x << endl;
        }
};

void friend_class::friend_fun(base * b1, base *b2) {
    cout <<  b1->x + b2->x << endl;
}

base * addbase(base * base1, base * base2) {
    base * base3 = new base(base1->x + base2->x);
    return base3;
}


int main () {
    base * bs1 = new base(3);
    base * bs2 = new base(4);
    base * bs = addbase(bs1, bs2); // adding two base classes using friend
function
    bs->print();

    friend_class * fc = new friend_class();
    fc->friend_fun(bs1, bs2);

    // 7
    // 7
```

```
    return 0;
}
```

Virtual function in multilevel inheritance

- for multilevel inheritance just declare the base class functions as virtual (straight forward)

```
#include <bits/stdc++.h>

using namespace std;

class base1 {
    public:
        virtual void speak() {
            cout << "base 1 speaking" << endl;
        }
};

class base2 : public base1 {
    public:
        virtual void speak() {
            cout << "base 2 speaking" << endl;
        }
};

class derived: public base2 {
    public:
        void speak() {
            cout << "derived speaking" << endl;
        }
};


int main () {
    // base class pointer pointing to derived class pointer
    base1 * bs1 = new base1();
    base2 * bs2 = new base2();
    derived * dr = new derived();

    bs1 = bs2;
    bs1->speak(); // prints base 2 speaks

    bs1 = dr;
    bs1->speak(); // prints derived speaking
    return 0;
}
```

Accessing private functions using virtual keyword

```cpp
#include <bits/stdc++.h>

using namespace std;

class base1 {
    public:
        virtual void show_id() {};

};

class base2 : public base1 {
    private:
        int id; // very secret information
        void show_id() {
            cout << "the id is : " << id << endl;
        }
};

int main () {
    base1 * bs1 = new base1();
    base2 * bs2 = new base2();

    bs1 = bs2;
    bs1->show_id(); // runs the show id function
    return 0;
}
```

Can virtual functions be friend functions

- YES, The virtual functions can also become friend functions

```cpp
#include <bits/stdc++.h>

using namespace std;

// forward declaration

class base1;

class base2 {
    public:
        virtual base1 * add_two_base1(base1 *, base1 *);
};

class derived: public base2 {
    public:
        base1 * add_two_base1(base1 *, base1 *);
};

class base1 {
```

```
    friend base1 * base2 :: add_two_base1(base1 *, base1 *);
    friend base1 * derived :: add_two_base1(base1 *, base1 *);

    int x, y;
    public:
        base1 (int a, int b): x(a), y(b) {}
        void show() {
            cout << x << " " << y << endl;
        }
};



base1 * base2 :: add_two_base1(base1 * bs11, base1 * bs12) {
    base1 * bs1 =  new base1(bs11->x + bs12->x, bs11->y + bs12->y);
    return bs1;
}


base1 * derived :: add_two_base1(base1 * bs11, base1 *bs12) {
    base1 * bs1 =  new base1(bs11->x + bs12->x + 2, bs11->y + bs12->y +
2);
    return bs1;
}

int main () {
    base1 * bs1 = new base1(2,3);
    base1 * bs2 = new base1(4,2);
    base2 * bss2 = new base2();
    base1 * bs3 = bss2->add_two_base1(bs1,bs2);
    bs3->show(); // 6 5

    base2 * bs_ptr = new base2();
    derived * der_ptr = new derived();
    bs_ptr = der_ptr;
    base1 * bs4 = bs_ptr->add_two_base1(bs1,bs2);
    bs4->show(); // calls the add_two_base1 function of derived class
    return 0;
}
```

Name Mangling

- c++ allows function overloading but linker does not (so how the overloading is performed in c++?)
- the g++ compiler uses name mangling to assign a name to each function to overload it
- example mangling pattern
    - _ZNOP
    - Z is just Z (does not represent anything)
    - N is number of characters in original function name
    - O is the original function name

    - P is parameter encoded (int, char etc)
    - void myfun(int a, string c)
        - name = _Z5myfunis

Early and Late binding

- Binding basically refers to the conversion of functions or variables to addresses
- Early binding is also called as compile time polymorphism where at the compilation time the compiler does the binding
- Dynamic binding is also called as runtime polymorphism where at the runtime the compiler decides and does the binding (can be achieved using virtual functions)
- By default, C++ follows early binding

Keep in mind

- static functions cannot be declared virtual as the static function tied to class not object to class and cpp doesn't support pointer to class;
- virtual function can be a friend function for another class
- The function is overloaded based on the argument list not on the return type
- runtime polymorphism is also known as dynamic polymorphism or late binding
- ambiguity resolution can be done using (class_name:: fun_name) or by virtual base class

---

योगः कर्मसु कौशलम्