Day6: Collections

1. Normal Static Declaration

   ⇒ **One Object**

   - Deleted when main Ends

```
Comlex c1;
Complex c2(7,2);
```

   ⇒ **Array of objects**

1. `Complex CompArr[5];`

An array of Complex are Constructed using the default constructor

2. `Complex myArr[3]; = { Complex(20,7),Complex(3),Complex(9)};`

An array of Complex each object constructed using different const

1. Dynamic allocation

   - we should delete it manually

```
Comlex *ptr1;
ptr1 = new Comlex();
ptr2 = new Complex(7,2);
delete ptr1;
```

1. `Complex *ptr;` `pAr = new Complex[5];`

5 objects Dynamically using default Const.

`delete[] pArr;`

if I forget[] it will delete 1st object only but in the case of a dynamic array of primitive types it won't make difference using [] or not

2. In Dynamic allocation, we Can't Choose which Constructor to constructor object by it.

Notes:

⇒ in One Object:

- Static : `c1.setReal(10);`
- Dynamic: `ptr1->setreal(10);` `ptr1->print();`

⇒ Array of Objects:

- Static:

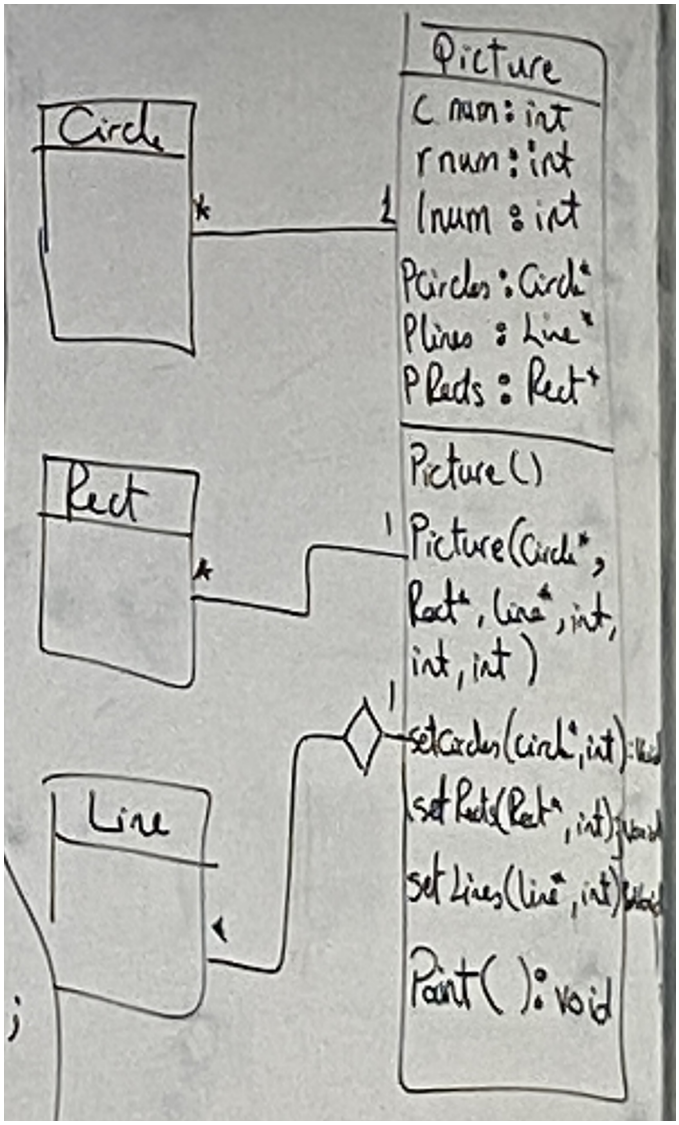
```
CompArr[i].setreal(10);
CompArr[i].print();
```

- Dynamic :

```
PArr[i].setreal(10);
pArr[i].print();
//Or
Complex *pCurr = pArr;
pCurr++;
pCurr->setreal(10);
pCurr->print();
```

Write a Design for a picture that uses Different Shapes inside it

⇒ Aggregation Relation



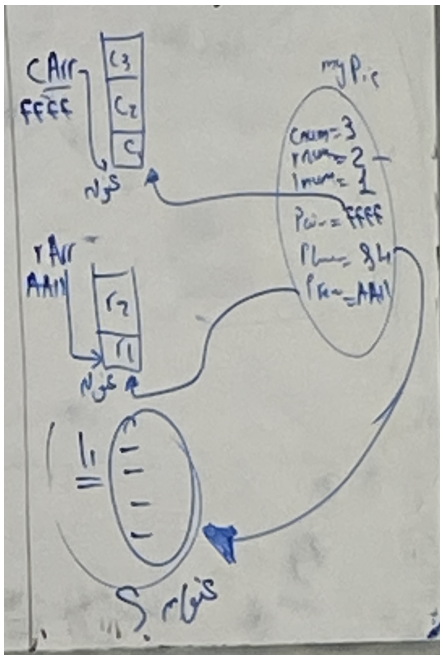

```
class  Picture
{

    int cnum,rnum,lnum;
    Circle *pcircle;
    Rect *prect;
    Line *plines;
    public:
        Picture()
          {
            cnum =0;
            rnum =0;
            lnum=0;
            pcircle = NULL;
            prect = NULL;
            plines = NULL;
```

```
          }

        Picture(int cn,int rn,int ln, Circle *pc, Rect* pr,line *pl)
          {
              cnum = cn;
              rnum = ln;
              lnum = ln;
              pcircle = pc;
              prect = pr;
              plines = pl;
          }

        void setCircles(int cn, Circle *pc)
          {
              cnum = cn;
              pcircle = pc;

          }
        void setRects(int rn, Rect *pr)
          {
              rnum = rn;
              prect = pr;
          }
        void setLines(int ln ,Line *pl)
        {
            lnum =  ln;
            plines = pl;
        }
        void Paint();
};

void Picture::Paint()
{
    for (int i = 0; i <cnum; i++)
    {
        pcircle[i].draw();
    }
    for (int i = 0; i <rnum; i++)
    {
        prect[i].draw();
    }
    for (int i = 0; i <lnum; i++)
    {
        plines[i].draw();
    }
}
```

Static

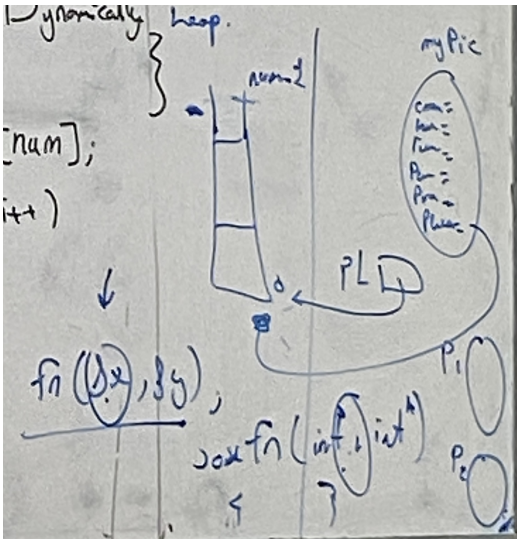
```
void main()
```

```
{
    Picture myPic;
    //Declare Object Statically
    Circle c1(50,50,50);
    Circle c2(200,50,50);
    Circle c3(300,150,200);

    Circle cArr[3] = {c1,c2,c3};
    Rect r1(200,150,350,150);
    Rect r2(100,100,200,350);

    Rect rArr[2] ={r1,r2};

    Line l1(200,300,250,150);
    myPic.setCircles(3,cArr);
    myPic.setRects(2,rArr);
    //because l1 its an normal var not array we have
to pass the address of this var
    myPic.setLines(1,&l1);
    myPic.Paint();

}
```



Dynamic

```
void main()
{
Picture myPic;
Point p1,p2;
int num,x1,x2,y1,y2;
cin>> num;
Line *pL = new Line[num];
for (int i = 0; i < num; i++)
{
    cin>>x1;
    cin>>y1;
    cin>>x2;
    cin>>y2;
    p1.setx(x1);
    p1.sety(y1);
    p2.setx(x2);
    p2.sety(y2);
    pL[i] = Line(p1,p2);
}
myPic.Paint();
}
```



Templates

make our user Defined Types to be generic ⇒ working with Diff types based on Choice (Where I can save Data & Choose kind of datatype @Run Time

```
template<class T>
class Stack
{
    T *st;
    int top,size;
    static int counter;
    public:
        Stack()
            {
                counter++;
                this->size = 10;
                st = new T[10];
                top = 0;
            }

        Stack(int size)
            {
                counter++;
                this->size = size;
                st = new T[size];
                top =0;
            }
        ~Stack()
            {
                delete []st;
                counter--;
            }
        Stack(Stack&);
        void push(T);
        T pop();
        static int getCounter(){return counter;}
        Stack& operator=(Stack &);
        friend void viewContent(Stack);
};

template <class T>
int Stack<T>::counter = 0;

int main()
{
    Stack<int> s1(5);
    Stack<int> s2;
    s1.push(10);
    s1.push(21);
    s1.push(1);
    s1.push(2);
```

```
    s1.pop();
    cout<< "Int stack # is:"<< Stack<int>::getCounter()<<endl;
    Stack<char> mys;
    mys.push('a');
    mys.push('b');
    mys.push('c');
    cout<< Stack<char>::getCounter();
    return 0;
}



template <class T>
Stack<T>::Stack(Stack<T> & myst)
{
    top = myst.top;
    size = myst.size;
    st = new T[size];
    for(int i =0; i<myst.top;i++)
    {
        st[i] = myst.st[i];
        counter++;
    }
}

template <class T>
void Stack<T>::push(T n)
{
    if(top == size) cout<<"stack is full" <<endl;
    else
    {
        st[top] = n;
        top++;
    }

}

template <class T>
T Stack<T>::pop()
{
    T retval = 0;
    if(!top) cout<<"stack is empty" <<endl;
    else
    {
        top--;
        retval = st[top];
    }
    return retval;
}

template <class T>
Stack<T> & Stack<T>::operator=(Stack<T> &mys)
{
  delete []this->st;
```

```
    size = mys.size;
    top = mys.top;
    st = new T[size];
    for(int i=0;i<top;i++)
    {
        st[i] = mys.st[i];
    }
    return *this;
}

template <class T>
void ViewContent(Stack<T> mys)
{
    for(int i = 0; i < mys.size;i++)
    {
        cout<<"i + 1" << mys.st[i] <<endl;
    }
}
```

Notes:

→ This code will print 2 1 because we have created **2 objects (s1,s2)** of type `stack<int>` and **one object(mys)** of type `stack<char>`

→ We can not only primitive datatype we can use user-defined datatypes

---

Streams #include <iostream.h>

Cin

it is an object from class `istream`

ex) `cin>>x;`

use operator shift right to get data from screen to saved in x int in memory

⇒ That means ***operator >>*** is overloaded inside istream for primitives types `( int,long,double,float,char,char * [string])`

Cout

is object from class `Ostream`

`cout<<x`

use << operator to print value of primitive type x to console screen

⇒ ***operator<<*** is overloaded inside class `Ostream` for primitives types.

What we need is overload Operators '<<' or '>>' to work with complex class.

We can't overload the functions , So we have to use friend functions to implement our logic to make these functions work with complex objects

```
Complex c1(5.3,7.1);
cout<<c1;
cin<<c1;
```

```
class Complex
{
    friend void Complex operator+(float,Complex);
    friend void Complex operator>>(istream &,Complex &);
    friend void Complex operator<<(ostream &,Complex &);
};

void operator>>(istream& mys, Complex &c)
{
      mys>>c.real>>c.img;
}
void operator<<(istream& mys, Complex &c)
{
      mys<<c.real<<c.img;
}
```

when we use operators overloading with return void we can't do this `cin>>c1>>c2`

so we must use return of `istream` with `cin` and `ostream` with `cout`

```
istream & operator>>(istream& mys, Complex &c)
{
      mys>>c.real>>c.img;
}
ostream & operator<<(istream& mys, Complex &c)
{
      mys<<c.real<<c.img;
}
```