Page: Date: / /

Classes

- All rules are same as of struct except a few small ~~stuff~~ changes.

example:

Struct

```
Struct Point {
    double x,y;
};

int main(){
    point p1;
    p1.x = 5;
    p1.y = 10;
}
```

✓ correct

Class

```
Class Point {
    double x,y;
};

int main() {
    Point p1;
    p1.x = 5;
    p1.y = 10;
}
```

✗ error

- if code is same then why error.

In struct it is not mentioned by struct is

```
struct {
public:
};
```

Day:

Date: / /

but class is

```
class  {
  private:


}
```

- Private members cannot be used or modified outside of class.

Access modifiers:

1) Private.
2) Public.
3) protected.

1) private:
   - This cannot be accessed outside of class.

2) Public.
   - Can be accessed anywhere.

- In classes, the default access modifier is private. but it was public in struct.

Why do we need private?

- We need to hide our codes from outside world. We can allow ~~our~~ people to use our classes but not see our research.
- we can validate x, y or any variable how ever we want and get input in our requirement.

Making Public / Private Classes

```
Class Point{
    private:
        double x,y;

    Public:
        void display (){
            ---
        }
};
```

```
int main(){
    Point p1;
    p1.x = 5;   error
    p1.y = 5;   error
    p1.display();   allowed
}
```

Day:

Date: / /

Encapsulation:

- Hiding data from outside world.

- we make private to make the code as we want it. Normally for now we make attribute variables as private;.

- But then How to change values of member variables:
  - Remember we can still change values of attributes inside the class. So we make member functions for this.

Example:

```
class Point {
    Private:
        double x,y;
    Public:
        void setX(int x1){
                x = x1;
            }
        void setY(int y1){
                y = y1;
            }
        void display(){ cout<< x <<","<< y <<endl;}
};
```

Yousaf

Day:

Date: / /

```
int main() {

    Point P1;

    p1.setX(3);
    p1.setY(4);

    p1.display();
}
```

Terminal:

> (3, 4)

so this is how we ~~check values~~ value change values for private members.

Adding a validation function:

```
Class Point {
private:
    double x, y;

    ~~void~~double validate (int* val) {
        if (*val < 0) {
            cout << "invalid value" << endl;
            return 0;
        }
        else return val
    }
```

if the value is < 0, 0 is returned to that x or y.

Day:

```
public:
    void setX(int x1){
        x = validate(x1);
    }
    void setY(int y1){
        y = validate(y1);
    }

    void display(){

    };

}
```

```
int main(){

    Point P1;
    P1.setX(-9);
    P1.setY(10);

    P1.display();
}
```

Terminal

0, 10

as -9 is < 0 x stored 0 as that was returned to x.

- These functions of setX and setY are called accessor manipulator

Yousaf

Day: Date: / /

Scope Resolution Operator / Class resolution operator:

Symbol: "::"

- This helps us use the class member variables if the ~~func~~ another veriable in another scope has same name in the objeet.

Example:

```
class Point {

    Private:
    double x,y;

    Public:

    void setX( double x) {
                Point :: x = x;
    }

};
```

- This also has usage in out-line functions (will be read later).

Youssef

Constructor:

- An object can only be made through a constructor.
- But we did not make constructors before, how was it running? Compiler always makes a default constructor if we don't make one.

- Name of constructor is same as name of class
- There can be more than one constructor.
- Once you make any constructor the default constructor is not made.
- Constructor should normally be public.
- There are no Return type for function.

```
Name of class ( Parameters ) {
        body;
    }
```

Yousaf

Day: Date: / /

E.g:

```
Class    Point {

    Private:
        double x;
        double y;
    double validator ( double val) { ------ };
    Public:

    Point () { } //default constructor;

    Point (double x1, double y1) {
        x = validator (x1);
        y = validator (y1);
    }

    Point ( double x1 ) {
        x = validator (x1);
    }
};

int main () {
        Point p;          ✓ allowed
        Point p (1);      ✓ allowed
        point p (1, 2);   ✓ allowed
        point p (1, 2, 3);  ✗ disallowed
                          as there is no 3rd
                          parameter
                          constructor
```

Yousaf

Date: / /

- if in the point example the default constructor did not exist then

Point p; ✗ This would have been an error.

Combining all the constructors:

```
Point (double x1 = 0, double y1 = 0) {
    x = validator (x1);
    y = validator (y1);
}
```

- Now we have used default arguments.

So now if we write:

✓ Point p; // (0,0)
✓ point p1 (1,2); // (1,2)
✓ point p2 (2); // (2,0).

Yousuf

Ambiguity error:

eg
if we write

1) Point (double x=0, double y=0){
}

2) Point () { }

Now which will be called for
Point p;

Both can be used as one stores (0,0) and other garbage.

- This is called Ambiguity error.

Day: Date: / /

An important points

1) int main(){

Point * p_ptr = new Point (3, 2);

allowed if this constructor is available.

}

Date: / /

Destructor

- Just like constructors are needed to make an object, destructor is needed to destroy an object.

- ~~destructor~~ destructor is only one
- It calls it self when an object ends

Default destructor

```
~Point () {
}
```

- Same as constructor but with a "~" in start.

- But this only destroys the memory in stack. what if it was in heap? then?

Yousaf

Prog: Date: / /

```
class Point {

    int *x;
    int y;

Public:

    [redacted]

    void x_size (int size) {
      x = new int [size];
      }


~ Point () {
      delete [] x;
      }

}
```

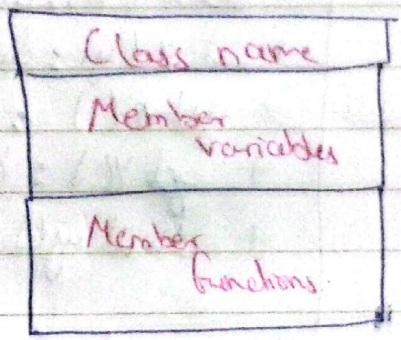
Now this will also delete the heap memory.

Day: Date: / /

Class diagrams:
Unified Model language (UML):

In a UML diagram everything that needs to be in a class is written and accepted as an unified way for understanding the class.

Layout:



- Private things are written with a "-" sign.
- Public things are written with a "+" sign.

"-" or "+" sign show access modifiers.

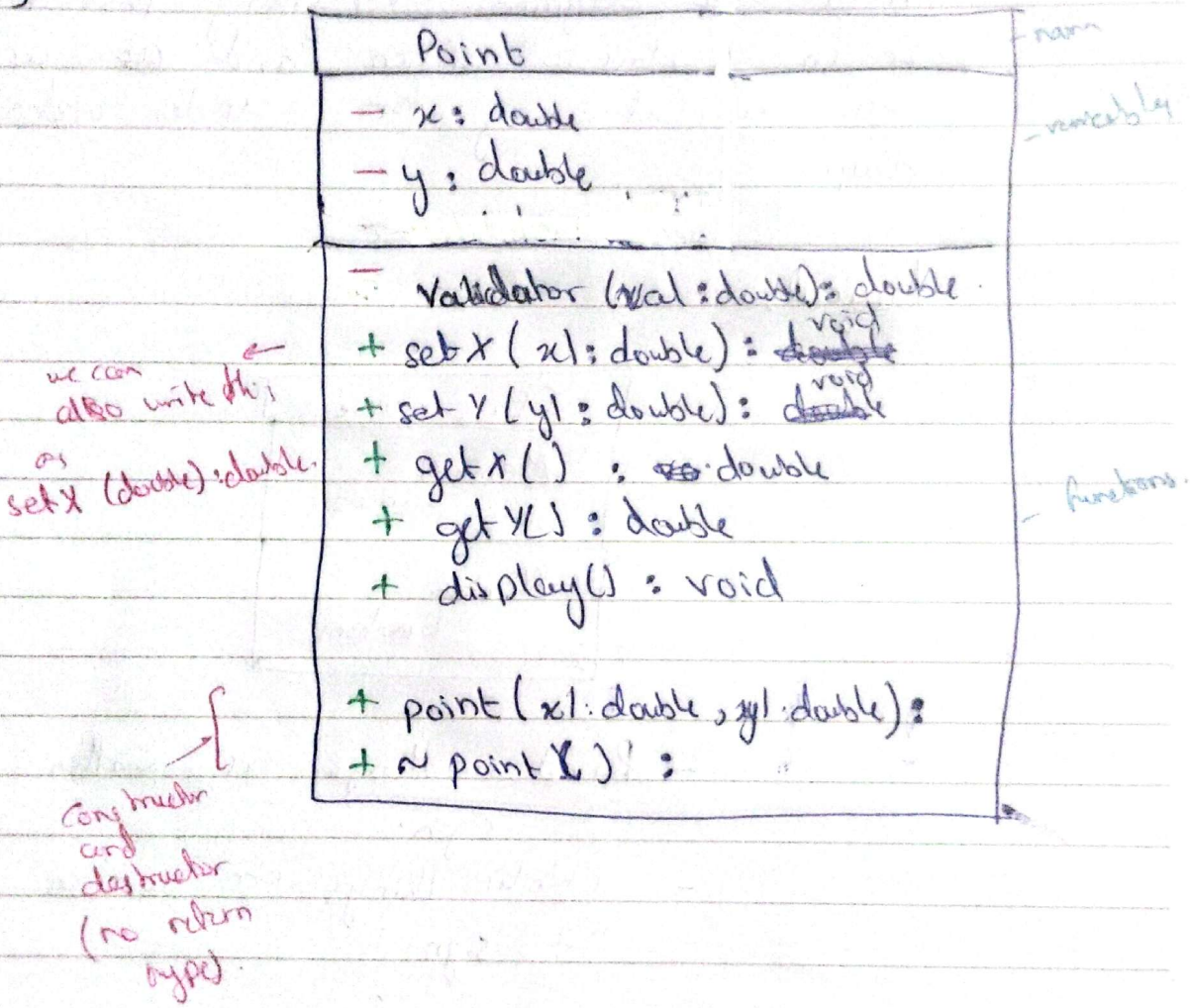
- To show a variable, Name is first: Then datatype. (same for function.

Yousuf

Day: Date: / /

Example:

1)



Yousaf

Day:

Date: / /

An important

Out line function:

- Before we were only making function inside the class, we can also make them outside but they will contain a prototype inside.

```
class ClassName {

    Datatype functionname ();

};

Datatype ClassName :: functionname() {

}
```

Example:

Day: Date: / /

```
class point {

    Private:
    int x,y;

    Public:

        void setX( int );
        void setY (int);
        int getX ();
        int getY ();
        void display ();
        Point ( int =0 , int=0); //constructor
        ~Point (); // destructor.

};

void  Point:: setX (int x1) {
                x = x1; }
void  Point :: setY (int y1) {
                y = y1; }

int Point:: getX() { return x; }
int Point :: getY() { return y; }
void Point:: display() { cout << x << ", " << y << endl; }
Point :: Point ( int x1 , int y1) { x=x1; y=y1; }
Point :: ~Point () { cout << "~"; display (); } ;
```

Date: / /

Important points:

- Default arguments can be ONLY be written once either in prototype or function.

e.g

```
display (int x1 = 0, int y1 = 0);
display (int x1 = 0, int y1 = 0) {
```

✗ Not allowed error

```
}
display (int x1, int y1) {
```

✓ allowed

```
}
```

```
display2(int x1, int y1=0);
display2 (int x1=0, int y1);
```

✓ allowed

- A constructor or destructor have no return type.

Day: Date: / /

Constant functions:

- Sometimes we don't want to change the contents of a function. for this we can create constant functions.

Syntax:

```
Datatype. Name. (Parameters) const {

}
```

- This addition of const makes sure that no member attribute is changed.

- But only member attributes CANNOT change. variables in parameters or variable created inside function can change.

e.g

A display function no needs to change attributes or even getter functions.

```
void display () const {
        cout << x << ", " << y << endl;
    };
```

Yousaf

Day: Date /

important point:

- Why these const are needed?

When we create a CONST user defined datatype e.g

const Point p;

it can ONLY use constant functions.

e.g:

```
class Example {
    private:
        int x;
    public:
        void display () { cout << x << endl; }
};

int main () {
    const Example E1;
    E1.display (); X error, Not allowed.
}
```

Yousaf

Day: Date: / /

- so to be safe from this error we need to make a const function.

- Can a normal User defined variable use a const function?
YES! it can. but priority will be given to non-constant function e.g.

```
class Example {
private:
    int x;
public

    void display () const {
        cout << x << endl; }
    void display () {
        x++;
        cout << " " << x }
```

Example (int x1) { //constructor
x = x1; }

Day: Date: / /

```
int main() {

    const Example E1(3);
    Example E2(4);

    E1.display(); This uses non const function
    E2.display(); This uses const function

}
```

Terminal

```
3
5
```

Yousaf

Day: Date: / /

This pointer:

The this pointer points to the class it is in.

- It is automatically created.
- It is not added to size of class (it is like an alias).

- this point only to object.

- this is a constant pointer. meaning its address cannot be changed.
  (this can be like:
  classname * const this;)

- this can ONLY be used inside the class or outline functions.

Using this; a simple example:
using the same point example.

```
1   void Point :: setX (int x) {
2
3       (*this).x = x;
4   }
```

- this points to the class so as both the ~~same~~ variable member variable and parameter had same name so we can use this to point to the x of the class which is the member variable.

```
void point::setX(int x) {
    this -> x = x;
}
```

changed in notation.

```
void point::display () {
    cout << "(" << this -> x << ", " << this -> y << ")" << endl;
}
```

Using this; A cascading effect (domino effect).

```
class Point {

    Private:
        int x,y;

    Public:
        int getX() const { return x; }
        int getY() const { return y; }

    Point & setX(int x) {

        this->x = x;
        return *this;
    }
    Point & setY(int x) {

        this->x = x;
        return *this;
    }

    Point & display() const {
        cout << x << ", " << y << endl;

        return* this;
    }
```

```
    Point (int x=0, int y=0){ //constructor.
        this->x = x;
        this->y = y;
    }

};

int main (){

    Point p1; //initially it is (0,0) due to constructor.

    p1.setx(5).sety(10);

    p1.display();

    p1.setx(15).sety(20).setx(5).display();

}
```

Terminal:

```
5, 10
6, 20
```

Day: Date:

But HOW?

p1. setx(5) . sety(10) .

This function returns the this or * this which is p1 itself.

So. by sequence this is happening.

p1. setx(5) . sety(10);

1) p1.setx(5)
2) p1.sety(10)

These two steps are happening behind.

Another way for cascading effect:

```
class Point {

    Private:
        int x,y;

    Public:
        int getX() { ---; }
        int getY() { ---; }

        Point * setX(int x){
            this->x = x;
            return this;
        }
        Point * setY (int y){
            this->y = y;
            return this;
        }
        Point * Display (){
            cout << x << " , " << y << endl;
            return this;
        }

};
```

```
int main () {

    Point p1;

    p1.setx(5) → sety(5);

    p1.display() → setx(2) → sety(-5) → display();

}
```

Terminal

```
5 , 5
2 , -5
```

But How?

Now we are returning a pointer which represents p1 so we needs first dereference it and then dot it with next function. This is what "→" does so we used it.

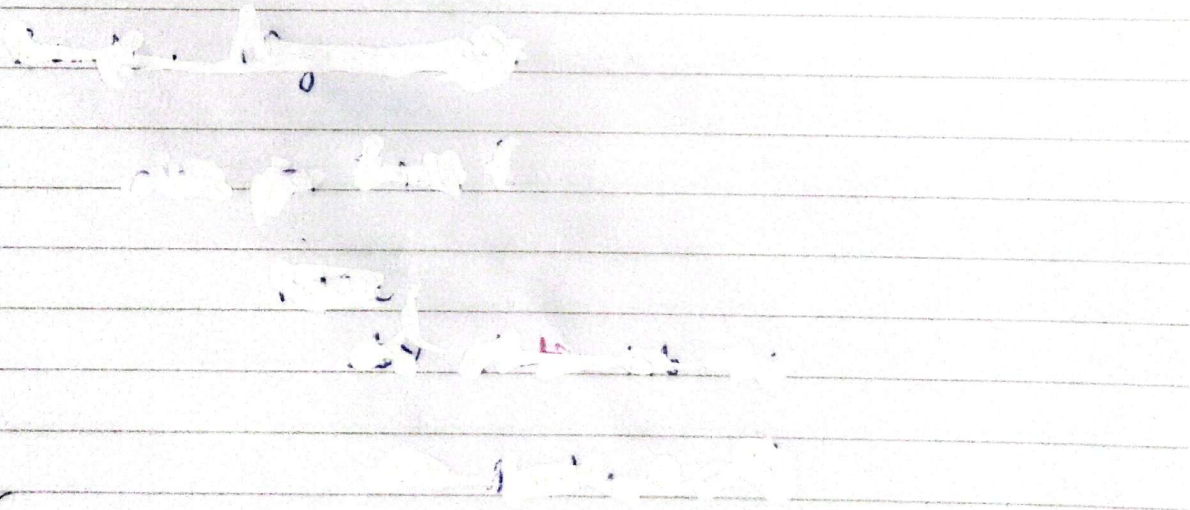
Day: Date: / /

Static variables:

- It is created the same as another member variable just write static before datatype. but also one more thing.

- BUT static is made in Data area of RAM.

- So every object ~~class~~ SHARES the same static variable. ~~if one~~



Syntax:

```
class Point {
    Public:
        static int c;
};  int Point :: c = 5;
```

c cannot write static here as it error

we need to declare in static always

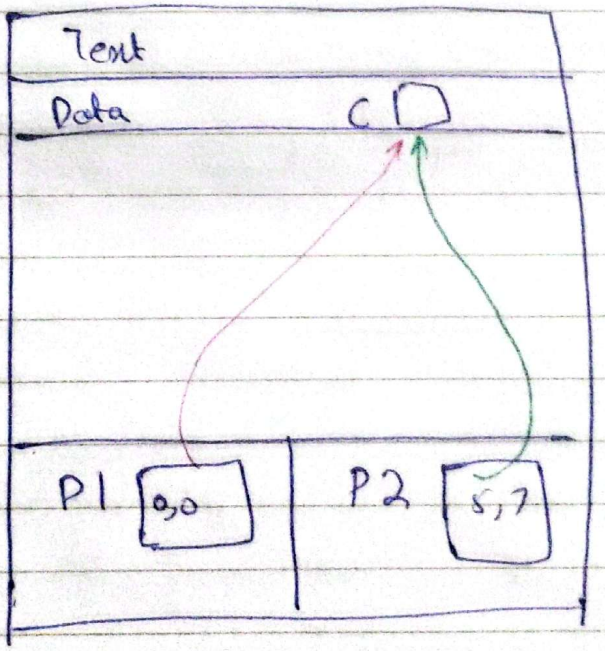
Yousaf

Day: Date: / /

- c need to be initialized globally.

Now how does shaping work?

e.g in int main we have two pointers, P1 and P2.



- both are connected to the same c.
- so if we change c, it will change for both.

= Now this pointer will not work for c e.g this → c, this is an error because c does not live in P1 or P2. It is in data.

Yousaf

= Now even in main Point :: c () Allowed as c is common for all.

✓ cout << Point :: c ; runs even in main.

but

✗ cout << c ; // This is error.

Page: Date: / /

Static variable : (Need of them):

- Static member ~~variable~~ variables could be used to count the number of objects running.

- The variable could be incremented in the constructor and decremented in the destructor.

Example:

```
class Point {

    int x, y;
    static int count;
Public:
    Point (int x=0, int y=0) {
        this->x = x;
        this->y = y;

        count++;
    }

    ~Point () {
        count--;
    }
```

Day: Date: / /

```
void getCount() {
    cout << count << endl;
}
};  int Point::count = 0;

int main() {
    Point p, p1, p2;
    p.getCount;
    Point p3, p4;
    p.getCount;
return 0;
}
```

Terminal:

```
3
5
```

So number of objects are counted.

Day: Date: / /

Static Functions:

- Before we used p.getCount but it required an object, what if we could call getCount without using an object. For this we will create a static function.

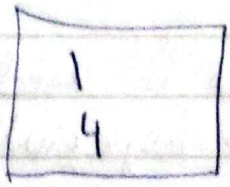
```
static int getCount () {
        return count;}
```

Now in main we can write:

```
int main () {

    Point p;

        cout << Point:: getCount () << endl;

    Point P1 [3];
        cout << Point :: getCount() << endl;
}
```

Date: / /

Terminal:



Important points:

- Static functions cannot use this pointer or call NON-static functions.
- A NON-static function can use static function
- A static can only use static functions and static variables.

- e.g if we try to write cout << x in static function, which object's x are we refering to. So this is error.

Date: / /

Member initializer list:

- A member initializer list is used to initialize variables especially constant variables.

- It can only be used with constructors.

- If we create constant member variables we cannot change their value simply as they are not in stack. So there is only one way that is a member initializer list.

- This is also for member reference (&) variables.

Syntax:

```
Name (int v1, int v2) : v1(v1), v2(v2)
{}.
```

- after the constructor parameters write a colon and then write the name of member variable and in bracket write the name of the parameters

Yousaf

Date: / /

example:

```
Point (int x1, int y1): x(x1), y(y1) { };
```

- This initializes x & with x1 first then y& with y1.

Example:

```
class intArr {
    public:
    const int size;
    int * ptr;

    Private:

    intArr (int s=3): size(s) {
        ptr{new int [size]};
        for (int i=0; i< size; i++)
            ptr[i] = 0;

    }
```

Better way:

next page.

```
IntArr ( int s = 3) : size(s), ptr(new int[size]) {

    for(int i=0; i<size; i++)
        ptr[i] = 0;

    }
};
```

- In member initializer list we also include other function, a common one being the validator function e.g

```
IntArr ( int s = 3) : size(validator(s)),
```

- We can also use a ternary operator instead of a validator.

Day:

Date: / /

Member wise Assignments:

The "=" operator can be used to assign one objects data to another object or to initialize one object with another objects.

- Everything is copied, all variables, all arrays, all pointers.

Example:

```
class Example {
    Private:
    int x[3];
    int y;

    Public:
    Example( int x1=0, y1=0 ) {
        x[0] = x1;
        x[1] = x1+1;
        x[2] = x1+2;
    }. y = y1;

};
```

Day: Date: / /

```
int main(){
    Example E1(3,2).
    //Example E2 = E1; // This will not call = operator instead it uses copy constructor
    Example E2;
    E2 = E1.
}
```

after this E2 will have

same as E1.

- This equal to (=) operator will be read in detail in operator overloading.

Day: Date: / /

Copy constructor:

A copy constructor is a special constructor that is called whenever a new object is created and initialized with another objects data.

It works like:

Example E1;

Example E2(E1);

Now E1 is copied to E2. like in member wise assignment.

But Here is a problem:

if we have a member pointer for example int * ptr.

in E1 ptr → [ ] but then when we do E2(E1) E2's ptr also points to same place.

- This is a huge problem as now if we change value of E2's ptr, E1 also changes.

Yousaf

- secondly if the destructor is called it will delete E1 first and make the ptr memory as free but when E1 will be deleted it will cause memory corruption as ptr is already deleted. This error will be core dump.

- For this we will make a copy constructor of our own instead of just using the default.

Syntax:

```
Name of Class ( Name of Class &identifier) {
    body.
}
```

e.g.

```
Example ( Example & copy) {
    body.
}
```

- A better copy constructor will be to write const in parameter as we are should not be changing the contents.

Yousaf

Day: Date: / /

Example

```
class intArr {

    private:
    const int size;
    int * ptr;

    Public:

    intArr (int s=0) : size(s), ptr (new int [size]) { // constructor.
                for(int i=0; i< size; i++){
                    ptr [i] =0;
                }
    }

The default constructor // intArr (const intArr & copy) : size (copy.size),
                        //  ptr (copy.ptr) { }

    };
    intArr ( const intArr & copy): size (copy.size),
      ptr (new int [size]) { // copy constructor.

                for (int i=0; i<size ; i++ ) {
                    ptr [i] = copy.ptr [i];
                }
    }
```

Day: Date: / /

```
~int Arr(){
    delete[] ptr;
    }
};
```

- This example makes the perfect copy constructor.
- But if there was an array in member variables, you had to deal with that too. using a for loop.

When is a copy constructor called?:

This is mandatory.

① when e.g we write `int Arr brr = arr;` or brr(arr).

it should be initialized when created if else we do

② It is called when we pass by value

Same as = operator

③ At value returning copy constructor is called but in modern compilers Constructor Return value optimization (RVO) is used and copy constructor not called

Yousaf

Day:

Date: / /

Why is & needed in copy constructor:

if we did not add & then

```
int Arr (const int Arr copy) {
    body;
}
```

But this is created by copy constructor as this is pass by value. Then this will again create copy constructor, then again then again, infinite times so it is an error.

Yousaf