

CS 32: Data Structures + Algorithms



## Objective

Data abstraction

C++ Classes

Pointers, Dynamic Arrays, Resource Management

Linked Lists

Stacks and Queues

Inheritance and Polymorphism

OOP

Recursion

Templates, Iterators, STL

Algorithmic Efficiency

Sorting

Trees

Tree-based tables, Hash tables

Priority Queues, Heaps

Graphs

## CS 31 Review

### Pointers

- Another way to implement pass by reference
- Traverse arrays
- Manipulate dynamic storage
- Represent relationships in data structures

double is a number

double *f* is a reference to a double  
(another name for a pre-existing object)

double\* is a pointer to a double.

*&x* generate a pointer to *x*

*\*p* the object that *p* points to

## Reference parameters

→ used to make a function return multiple values

void polarToCartesian ( double rho, double theta,  
double xx, double yy )

{

  xx = rho \* cos(theta);  
  yy = rho \* sin(theta);

}

int main()

{

  double x;  
  double angle;

  ... get r and angle ...

  double x;  
  double y;

  polarToCartesian ( r, angle, x, y )

while x and y obtain the correct value,  
they pass when the function passes.

what I'd like to do: change variable values OUTSIDE the function.

void polarToCartesian( double rho, double theta,  
double& xx, double& yy )

double rho and double theta create brand new doubles that the values are copied into.

But double& xx and double& yy are references — essentially, they point to the same double as x/y.

So if I change the value of x, even xx changes.

i.e. the parameter is just another name for the original argument. NO COPY IS MADE.

x: 5

angle: 0

x: ??? : xx

y: ??? : yy

5: rho

0: theta

Upon ending the function,  $xx$  and  $yy$  go away - but they do their job, i.e. alter the values of  $x$  and  $y$ .

### Pointers to pass by reference

$r : 5$

$angle : 0$

$x : ???$

$y : ???$

$5 : rho$

$0 : theta$

$x \leftarrow xx$

$y \leftarrow yy$

void polarToCartesian ( double rho, double theta,  
double\* xx, double\* yy )

{

$*xx = rho * \cos(theta);$

$*yy = rho * \sin(theta);$

}

int main()

{

double r;

double angle;

... .. get r and angle ...

```
double x;  
double y;
```

polarToCartesian( r, angle, &x, &y )

}

The pointer `&x` points to the main routine's `x`  
" " " `yy` " " " " `y`

Pointers are arrows pointing to something.

They don't actually hold a double, but tell you how to locate one.

To create an arrow/pointer to `x`, we append `&` before the variable name.

`&x` generates a pointer to `x`.

`double * xx` → data type for pointer

`&x` → pointer itself (address of `x`)

`* p` → the object that `p` points to (follow the pointer)

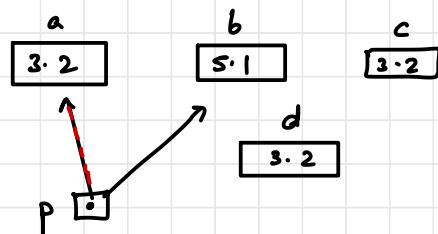
## Mechanics of pointers

```
double a = 3.2;  
double b = 5.1;
```

```
double* p = &a;
```

```
double c = a;
```

```
double d = *p; // Now p points to b
```



$$p = b$$

$p = \&b;$

make p point to b.

$*p = b;$

take b and store it  
where p points to.

```
int k = 2;
```

$p = \&k;$

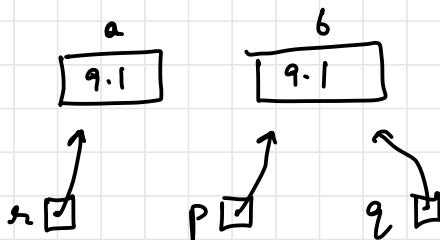
// different type, cannot happen.

pointer types cannot be converted.

Cannot multiply a pointer with any data type, obviously. Cannot perform numerical operations.

Following an uninitialized pointer throws an error. Randomly replaces some bit pattern in/out your system. Might point anywhere and can be potentially very dangerous.

When we compare two pointers, we compare the address. While the value of the pointer might be the same, the address might not.



$(p == s)$  // false

$\downarrow$   
6      9.1

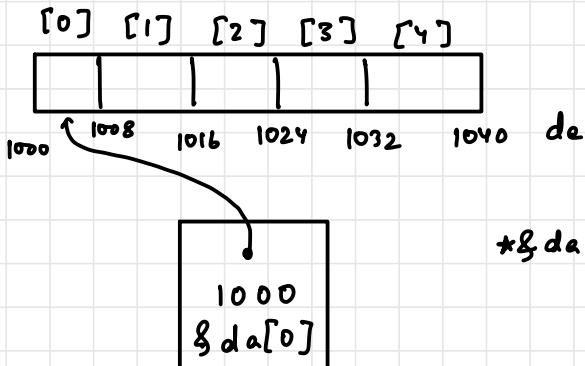
$(p == q)$  // true

$(\star p == \star s)$  // true

## Pointers to traverse arrays

```
for (double *dp = &da[0]; dp < &da[5]; dp++)
```

```
*dp = 3.6;
```



$$\&da[0] = da[0].$$

## Algebra rules

$$\rightarrow \&a[i] \pm j = \&a[i \pm j]$$

Integer + pointer = pointer [sub + integer]

→ C++ standard allows position 1040 (just pass the end) but that pointer cannot be followed

$$\rightarrow \&a[i] < \&a[j] \Rightarrow i < j$$

>

<=

>=

= =

!=

These compare positions in an array

→ In an expression, writing the name of an array without the sub is a pointer to element 0.

$$a \leftrightarrow \&a[0]$$

$$\&da[5]$$

$$\&da[0 + 5]$$

$$\&da[0] + 5$$

$$a + 5$$

Template to traverse an array

```
for (double * dp = da; dp < da + MAXSIZE;  
     dp++)
```

→  $\text{double } b[]$  is practically equivalent to  $\text{double* } b$ .

→  $p[i] \leftrightarrow *(p + i)$  because  $p[]$  is a pointer to the beginning of the array.

→ well-defined pointer that does not point to any object → nullptr. Used for error.

## NULLPTR

- ① int \* ip = nullptr;
- ② ip = nullptr;
- ③ if (ip == / != nullptr)

...

undefined : \*ip = 42 (if ip = nullptr)

An integer constant 0 in a context where a pointer is required is a nullptr.

## Structs

Define your own datatype. Instead of having multiple corresponding arrays, use structs.

Structs : common convention  
→ start with a capital letter

```
struct Employee
{
    string name;
    double salary;
    int age;
};

int main ()
{
    Employee e1;

    e1.name = "Fred"
    e1.salary = 50000
    e1.age = 51;

    Employee company [100]
    company [2].name = "Ricky"
}
```

0	1	2	n
		RICKY	

## Passing to functions : structs

use object of struct type . name of member of struct type  
to use attributes of the struct.

```
void printPayCheck (const Employee& e)
{
    cout << e.name << endl;
}
```

guarantees no change  
to e

Based on size of a struct,

large size - pass by reference (cheap)

small size - pass by value (expensive)

```
void f (Blah b);
```

or

```
void f (const Blah b);
```

To change a value, obviously use pass  
by reference or pointers.

### Pointer approach

\*ep.name ← dot operator takes precedence

(\*ep).name ← ugly

ep → name ← solution

a pointer of an object of some struct → the name of a member of the struct

## Member functions

Necessary to impose constraints within a struct.

struct Target

{

void init();  
bool move (char dir);  
void replayHistory();

] member functions

int pos;  
string history;

] data members

}

void Target:: init()

] :: implies a member function

{

this → pos = 0;  
this → history = ""

] 'this' is used by an object to access itself

}

The **this** arrow can be left off since the usage is implicit.

```
bool Target::move (char dir)
{
    switch (dir)
    {
        case 'R':
        case 'r':
            pos++;
            break;
        case 'L':
        case 'l':
            pos--;
            break;
        default:
            return false;
    }
    history += toupper (dir);
    return true;
}
```

**Access modifiers** - enforcement mechanism

public / private - impose your own functions

If I want my private variables to be used only by the public functions and not otherwise, I use access modifiers.

```
struct Target
```

```
{ public:
```

```
    void init();
```

```
    bool move(char dir);
```

```
    void replayHistory();
```

```
private:
```

```
    int pos;
```

```
    string history;
```

```
}
```

Public can be accessed by other classes  
and functions too.

Private can only be accessed by member  
functions inside the class, or indirectly  
by using public member functions.

Private functions cannot even be viewed  
outside the public functions. Allows  
changing internal implementation without  
changing external operation.

simply add a public

```
int Target::position()
```

```
{
```

```
    return pos;
```

```
}
```

common convention while naming private elements is to use a common prefix [to identify data members ]

For eg.

**private:**

```
int m_health;  
string m_name;
```

Name of some struct :: Name of a member of  
that type

If not specifically listed, all data members are public. This is due to C integration.

STRUCT

vs.

CLASS

→ Starts off public if not specified

→ Starts off private if not specified

otherwise, they are literally the same.

To be safe, list public and private members independently and clearly → use interchangeably.

const member functions

C++ does not allow, for eg.

```
void report (const Target& x)  
{ cout << x.position() << endl; }
```

because this might  
change values, and  
the compiler cannot  
see the implementation  
of the function

Therefore, the correct approach is:

```
int position() const;
```

which promises not to modify the object it is called on.

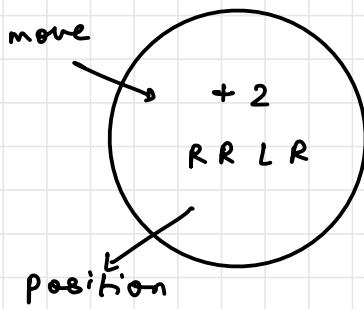
const also needs to be appended to :: implementation of a member function.

We need to make sure that the Target is initialized immediately. If some functions are called without initializing, that can then create.

We use a constructor for this. It is a special member function that is automatically called everytime an object is created.

```
class Target
{
public:
    Target();           // constructor
};                  // no return type
```

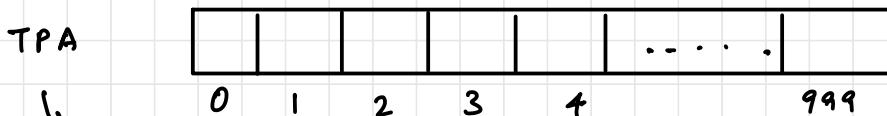
```
Target :: Target ()  
{  
    pos = 0;           // replaces int  
    history = "";     // initializes Target  
}  
                                // constructed immediately
```



flow of information

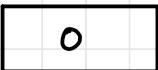
## Pointers and Dynamic storage

It doesn't make sense to initialize a Target till it is required. Initializing a 1000 targets when only 100 are needed would be a waste of processing. Therefore, we use pointers as such:



target pointer array

nTargets



# of targets being initially used

```
Target* tpa [1000];  
int nTargets = 0;
```

Syntax to construct a brand new target and give me a pointer assigned to a new array position:

$\text{tpa}[\text{nTargets}] = \underline{\text{new Target;}}$   
 built into C++  
 calls constructor  
 assigns a new pointer to  
 $\text{tpa}[\text{nTargets}]$   
 $\text{nTargets}++;$

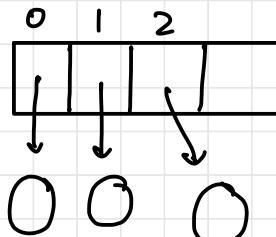
$\text{tpa}[1] \rightarrow \text{move}('R');$  // since this is a  
 pointer we use  $\rightarrow$   
 // I could potentially use  
 $(\star\text{tpa}[2]).\text{move}('R')$  but  
 verbose

### Pointers - Deleting a target

Theoretically, I can:

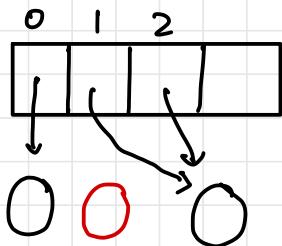
$\text{tpa}[1] = \text{tpa}[2];$

$\text{nTargets}--$



thereby reassigning [2] to [1] and deleting what was present in [1] initially. But this would involve two pointers pointing to the

same object, albeit briefly. However, I would NOT be looking at the second pointer, since



my nTargets would not include the second pointer element in the array.

A better way of looking at this would be:

$tpa[1] = tpa[2];$

$nTargets --$

$tpa[2] = \text{nullptr}; // \text{unnecessary}$

It is necessary to return storage to the operating system upon deletion of a certain element.

Here, for instance, the red blob is inaccessible, because the pointer pointing to it is lost. So we call it garbage.

A program with too much garbage leads to a **memory leak**. Memory leaks usually do not show up unless the program runtime is high.

Deleting the object that a pointer points to results in a **dangling pointer**. This may point to the previously stored information or may point to new information. Make sure to reassign the pointer.

---

Deleting the object that a pointer points to only makes sense if **New** is used. Otherwise, it causes an error.

In dynamic allocation, deleting a pointer deletes the object that the pointer points to

```
p = new Target;  
delete p;           // deletes the  
                    new target
```

Double delete is also a common error. Avoid deleting deleted memory.

```
int k;
```

```
int * p = &k;
```

```
delete p; // error, p cannot delete  
           a local variable
```

---

- Named local variables live on 'the stack'. They go away upon deletion.
- Variables declared outside any function live in the global storage area / static storage area.
- Dynamically allocated objects live on 'the heap.' They need to be deleted after usage.

The array of pointers lives on the stack. Upon leaving the function, they get deleted. This results in dynamically allocated objects becoming garbage.

Therefore, it is essential that we delete dynamically allocated objects.

```
for (int k = 0; k < nTargets; k++)  
    delete tpa[k];
```

---

To initiate a class member with set attributes, give constructor arguments.

public:

```
Pet (string nm, int initialHealth);
```

```
int main
```

```
{
```

```
Pet p1 ("Tommy", 10);
```

```
Pet p2; // Error
```

```
}
```

## Constructor Caveat

```
Blah b3(30, 20, 10);
```

```
Pet p2 ("Tommy", 20);
```

```
gleep g1 (20);
```

Target t(); // error: this is a function

Target t; // ditch parentheses, works.

## Destructors

To create an element which exists for some objects, we may use a pointer in private that points to the element value, and null pointer if it DNE. But since it exists inside a function, we must also cleanup before exiting the function, if we do not wish to leave garbage.

While this works, it may involve calling cleanup() too many times. Therefore, we use the concept of destructors.

## Syntax

`~Pet();` // replaces the cleanup function, never says void or contains arguments

## Destructors Caveats

```
Pet: ~Pet ()  
{  
    delete m_favToy;  
}
```

```
void Pet :: addToy ()  
{  
    m_favToy = new Toy;  
}
```

- Trying to delete the nullptr is harmless, in case `m_favToy` does not exist.
- If `addToy` is called twice, it will lead to a memory leak. Simply adding `delete m_favToy;` before assigning a new toy can help fix this.

# Pointers in Data Structures

```
1 int main() {
2 Company myCompany;
3 myCompany.hire("Rick", 50000, 34);
4 myCompany.hire("Larry", 40000, 32);
5 Company yourCompany;
6 yourCompany.hire("Fred", 50000, 51);
7 myCompany.setBonusRate(.02);
8 myCompany.giveBonuses(); → class Company;
9 class Employee {                                (incomplete type
10 public:                                         declaration)
11 Employee(string nm, double sal, int ag);      ←
12 void receiveBonus() const;                      ←
13 private:                                       ←
14 string m_name;                                ←
15 double m_salary;                               ←
16 int m_age;                                    ←
17 Company* m_company;                           ←
18 }
19 class Company {
20 public:
21 void hire(string nm, double sal, int ag);
22 void setBonusRate(double pct);
23 void giveBonuses() const;
24 double bonusRate() const;
25 private:
26 Employee* m_employees[100];
27 int m_nEmployees;
28 double m_bonusRate;
29 };
30 Employee::Employee(string nm, double sal, int ag, Company *cp) {
31 m_name = nm;
32 m_salary = sal;
33 m_age = ag;
34 m_company = cp;
35 }
36 void Employee::receiveBonus() const {
37 cout << "Pay to the order of " << m_name >> " the amount $" << (m.salary*
38 m_comapny->bonusRate()) << endl;
39 }
40 Company::Company() {
41 m_nEmployees = 0;
42 }
43 void Company::hire(string nm, double sal, int ag) {
44 m_employees[m_nEmployees] = new Employee(nm, sal, ag, this);
45 m_nEmployees++;
46 }
47 void Company::setBonusRate(double pct) {
48 m_bonusRate = pct;
49 }
50 void Company::giveBonuses() const {
51 for (int k = 0; k != m_nEmployees; k++) {
52 m_employees[k]->receiveBonus();
53 }
54 }
55 double Company::bonusRate() const {
56 return m_bonusRate;
57 }
```

2.00

```

class Employee
{
public:
    Employee(string nm, double sal, int g);
    void receiveBonus(double rate) const;
private:
    string m_name;
    double m_salary;
    int m_age
};

Employee::Employee(string nm, double sal, int g)
{
    m_name = nm;
    m_salary = sal;
    m_age = ag;
}

void Employee::receiveBonus(double rate) const
{
    cout << "Pay to the order of " << m_name << " the amount $" 
        << (rate * m_salary) << endl;
}

```

I simply pass the rate to the Employee

```

class Company
{
public:
    Company();
    ~Company();
    void hire(string nm, double sal, int g);
    void setBonusRate(double rate);
    void giveBonuses() const;
private:
    Employee* m_employees[100];
    int m_nEmployees;
    double m_bonusRate;
};

Company::Company()
{
    m_nEmployees = 0;
}

Company::~Company()
{
    for (int k = 0; k < m_nEmployees; k++)
        delete m_employees[k];
}

void Company::hire(string nm, double sal, int ag)
{
    m_employees[m_nEmployees] = new Employee(nm, sal, ag);
    m_nEmployees++;
}

void Company::setBonusRate(double rate)
{
    m_bonusRate = rate;
}

void Company::giveBonuses() const
{
    for (int k = 0; k < m_nEmployees; k++)
        m_employee[k]->receiveBonus(m_bonusRate);
}

```

2.00

```

class Employee
{
public:
    Employee(string nm, double sal, int g, Company* cp);
    void receiveBonus() const;
private:
    string m_name;
    double m_salary;
    int m_age;
    Company* m_company;
};

Employee::Employee(string nm, double sal, int g, Company* cp)
{
    m_name = nm;
    m_salary = sal;
    m_age = ag;
    m_company = cp;
}

void Employee::receiveBonus() const
{
    cout << "Pay to the order of " << m_name << " the amount $" 
        << (m_company->bonusRate() * m_salary) << endl;
}

```

I

```

class Company
{
public:
    Company();
    ~Company();
    void hire(string nm, double sal, int g);
    void setBonusRate(double rate);
    void giveBonuses() const;
    double bonusRate() const;
private:
    Employee* m_employees[100];
    int m_nEmployees;
    double m_bonusRate;
};

Company::Company()
{
    m_nEmployees = 0;
}

Company::~Company()
{
    for (int k = 0; k < m_nEmployees; k++)
        delete m_employees[k];
}

void Company::hire(string nm, double sal, int ag)
{
    m_employees[m_nEmployees] = new Employee(nm, sal, ag, this);
    m_nEmployees++;
}

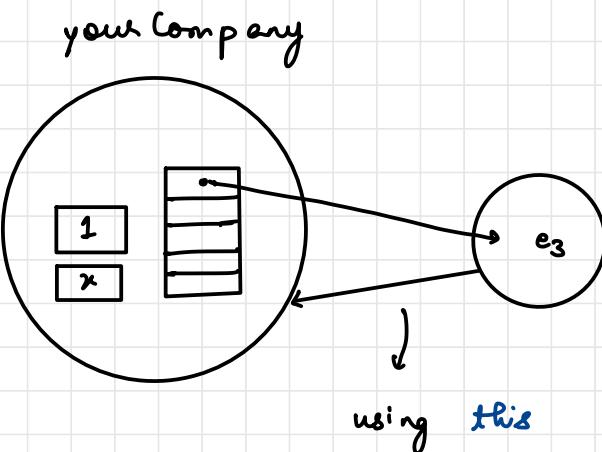
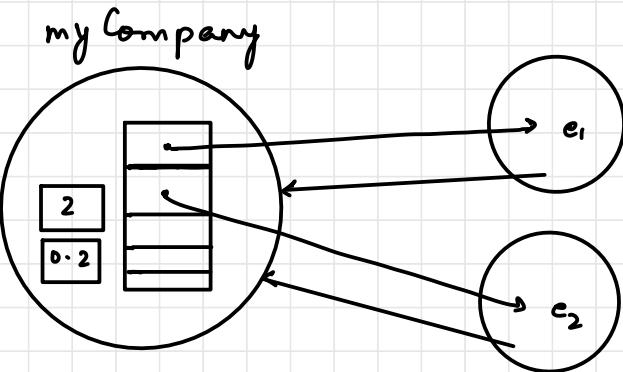
void Company::setBonusRate(double rate)
{
    m_bonusRate = rate;
}

double Company::bonusRate() const
{
    return m_bonusRate;
}

void Company::giveBonuses() const
{
    for (int k = 0; k < m_nEmployees; k++)
        m_employee[k]->receiveBonus();
}

```

using this in the environment of hire



## Pointers and overloading

```
1 =====
2 class Fan {
3 public:
4 void turnOn();
5 ...
6 };
7 class Rock {
8 public:
9 ...
10 double weight() const;
11 ...
12 };
13 class Robot {
14 ...
15 Fan m_cooler;
16 Rock* m_rock;    pointer to a rock or null if non-existent
17 ...
18 };
19 void Robot::blah() {
20 ...
21 if(m_rock != nullptr && m_rock->weight() > 50) {
22 m_cooler.turnOn();
23 }
24 ...
25 }
26 =====
27 class Complex {
28 public:
29 Complex(double re, double im);
30 double real() const;
31 double imag() const;
32 private:
33 double m_rho;
34 double m_theta;
35 };
36 Complex::Complex(double re, double im) {
37 m_rho = sqrt(re*re + im*im);
38 m_theta = atan(im, re);
39 }
40 double Complex::real() const {
41 return m_rho * cos(m_theta); } double Complex::imag() const {
42 return m_rho * sin(m_theta);
43 }
44 int main() {
45 Complex c1(4, -3); // 4-3i
46 cout << c1.real(); // writes 4
47 ...
48 Complex C2; // 0 + 0i
49 Complex ca[100]; // OK
50 }
51 You can overload a function name if the functions differ in the
number or type of parameters
52 void drawRect(Rectangle r); l
53 void drawCirc(Circle c);
54 int main() {
55 Rectangle rect1;
56 Circle circ1;
57 ...
58 drawRect(rect1);
59 drawCirc(circ1);
60 ...
61 }
```

order important

Complex();

Complex: Complex()

{  
    m\_rho = 0;  
    m\_theta = 0;  
}

If I declare a constructor, the compiler will not write a constructor for me.

If I do not declare a constructor, the compiler writes a default constructor for me.

'Default constructor' → a constructor with no parameters

Employee :: Employee ()

{}

Only throw in a default constructor if it contains a non-surprising value. Example of defaults : for complex:  $0 + 0i$ ; however you cannot have a default for, say, a UCLA student's ID.

OVERLOADING: giving more than one meaning to a function.

eg.      `Complex();`

`Complex( double re, double im );`

Any function can be overloaded, as long as the differ in terms of parameters.  
# or type

However, if arguments suggest different functions being called, the compiler throws an error due to ambiguity.

## Lecture 1 Part 1

### Undefined Behaviour

$a[k]$  where  $k$  is out of bounds.

$i/j$  where  $j = 0$ .

$p \rightarrow$  [element undefined]

Uninitialized variable assumed to be 0.

### Implementation-dependent behaviour

17/-5	17% -5
-3	2
-4	-3

If there's no return statement in a function, (except void), it is undefined behaviour.

```
double f
{
    if —
        return 0;
    if —
        return 1;
}
```

probably will not give an error, although it will give a warning. Ensure return in the general function code.

int

-2 billion to 2 billion

unsigned int  
(size\_t)

0 to 4 billion

0 - 2 billion → treated the same way

for (int k = 0; k < string.size() - 1; k++)  
    ↓                          ↓  
    unsigned                    signed  
    [size\_t]

an expression containing a signed and an unsigned int always converts the signed to unsigned.

Therefore, if string.size = 0, the value returned is 4 billion.

for (int k = 0; k+1 < s.size(); k++)

↳ fixed version

Alternatively,

int ssize = string.size()

CONVERT BOTH POTENTIAL CASES  
TO INT AND ACT UPON THAT.

## Lecture 1 Part 2

### Class Design and Issues

#### Circle type - interface

- Determine center and radius
- change size
- draw a circle
- find area (keeping it a non-member function for now)

class Circle

{ public:

    Circle( double x, double y, double r )  
    void scale( double factor )  
    void draw();

private:

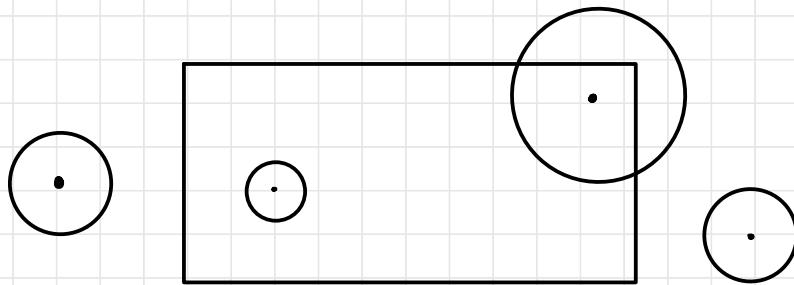
    double m\_x;  
    double m\_y;  
    double m\_r;

};

double area( Circle x );

Document all constraints while writing code.

Eg. The circle must be visible on the screen  
(bounds on the x and y).



Which one of these are valid?

### // class invariant

$m\_r > 0$  // r needs to be +ve

// r being 0 is silly in  
case of tangents

// constructor // the object is uninitialized here

```
Circle:: Circle (double x, double y, double r)
{
    if (r <= 0)
        exit (1);           // constructors cannot
    m_x = x;                return -1, so
    m_y = y;                we have to throw
    m_r = r;                exceptions
}
```

bool Circle:: scale (double factor)

```
{ if (factor <= 0)
    return false;
```

m-&star; = factor;  
between true;  
}

## Lecture 2