UIUC

→ C++ is a Strongly typed P. Lang

int value = 42 ; and location in memory.
↓ type ↓ name ↓ value

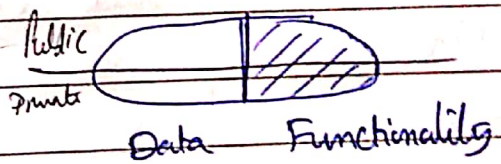
→ C++

(6) Primitive | User-defined
int, char, bool | → std::string
float, double, void | → std::vector

→ C++ → every program must have a starting point. main()

Encapsulation

1) - encloses data & functionality into a single unit called class.



2) In C++, the interface (.h file) to the class is defined separately from the implementation (.cpp file.

.h → like an API
also: all data
all fnctions

__/__/__

Cube.h (only the declaration)

pragma once → Always present in .h files
→ asks the compiler to compile the code only once

```
class Cube {
public:
    double getVolume();
    double getSurfaceArea();
    void setLength(double length);
private:
    double length_;
};
```

length_ ← indicates for private members.

Imp

- Note, this .h file does not have the definitions for the public member functions

Cube.cpp

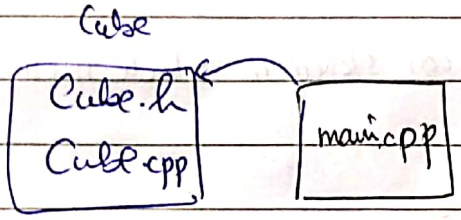
Cube → .h | .cpp

```
#include "Cube.h"

double Cube::getVolume() {
    return length_ * length_ * length_;
}

void Cube::setLength(double length) {
    length_ = length;
}

double Cube::getSurfaceArea() {
    return 6 * length_ * length_;
}
```

__/__/__

main.cpp

```
#include "Cube.h"
using namespace std;
int main() {
    Cube c;
    c.setLength(3.48);
    double vol = c.getVolume();
    cout << vol << endl;
    return 0;
}
```



Libs

1) iostream → using std::cout
using std::endl

Namespace

→ just like we used a Cube, there can be multiple other cubes throughout the organisation code. So we use namespace.

→ Cube.h

```
// namespace unic {
class Cube {
};
// }
```

Cube.cpp

```
namespace unic {
double getArea() {
}
}
```

We use namespace beg 2 diff libraries must use the same label for the same variable. __/__/____

main.cpp

```
//

main()

    (winc::Cube) c;
```

In C++, the default label for a Class is PRIVATE:.

⊛ Week 2

2.1 Stack Memory + Pointers (all var stored in stack memory)

every C++ variable has 4 things :-

```
int prime = 20;
```

datatype name val + memory address

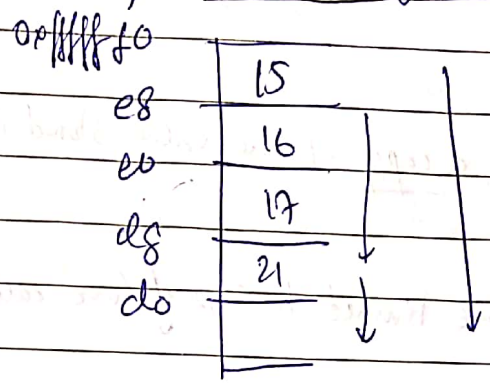
& operator → get the memory address
→ num
→ &num ← Mem address.

Lifecycle of a variable

→ all variables in C++ placed inside the stack memory
→ ⊛ Stack mem is associated with the CURRENT FUNC and lifecycle of var = lifecycle of func
— ⊛ whenever the fn returns/ends, memory space of the fn is released.

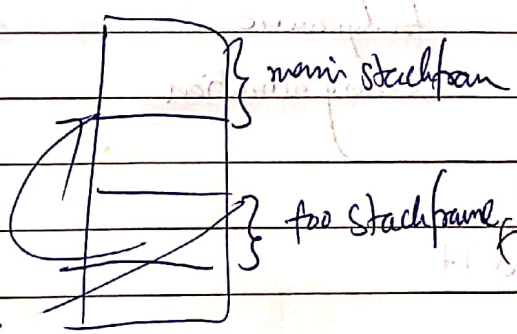
__/__/__

- Stack Memory starts at high addresses and grows down to 0.



Experiment to prove if first declared vars have higher MemAddr than vars declared later

```
void foo(){
    int num = 21;
    cout << &num << endl;   // 0xff...f8bc
}

int main(){
    int num = 7;
    cout << &num << endl;   // 0xff...f908
    foo();
}
```



Pointers

```
int *p = &num;
```

type name value

int num = 7

__/__/____

Dangling Pointer

→ int fun() {
int x = 2;
return x;
}

↓ Here, a copy of the value stored in x is returned.

int& funcl() {
int i; ← lifetime limited to scope of func call.
i=1;
return i;
}

↳ after this step, i has already been destroyed because now i refers to an object that DOES.

int main() {
int &p = funcl(); X
}

↳ p is garbage.

Heap
int* func2() {
int* p = new int;
*p = 1;
return p;
}

Stack
↓
automatic variables

Free Store (vs) Heap
↓ ↓
new/delete malloc/free
for dynamic + also dynamic
memory allocation

int* func() {
return &x;
}

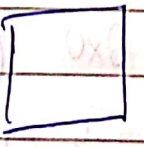
STACK memory starts high and goes down.
Heap starts low and goes up.

HEAP MEMORY

- Heap memory allows us to create memory independent of the lifecycle of the function
- If memory needs to stay longer than the lifecycle of the fn, we must use Heap memory.
- only way is to use `new` keyword.
- (new) keyword return a pointer to the memory storing the data, not the data itself

```
int * p = new int
*p = 5;
```

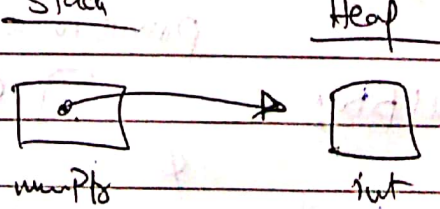
p → [5]

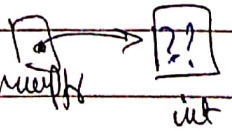


1) Allocate memory in heap
2) Initialize
3) return ptr to the start of the DS

② only when we use delete is it reclaimed

eg `int * numPtr = new int;`





`int * numPtr = new int;` | * numPtr = 42

`cout << * numPtr` → garbage | 42 ✓

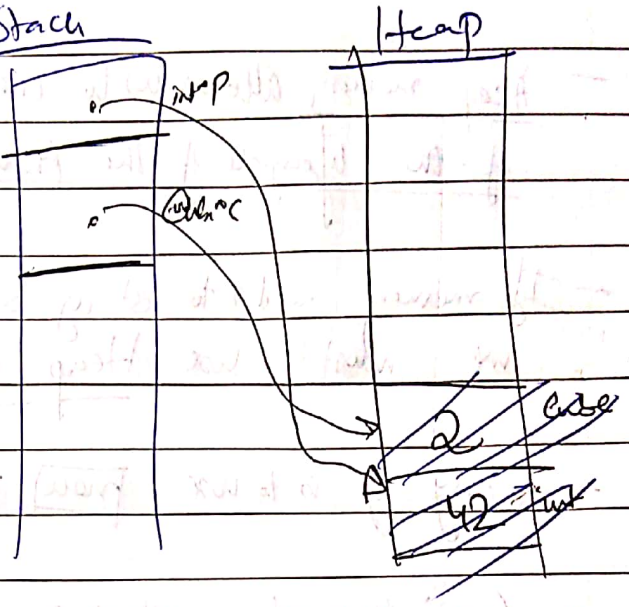
`<< numPtr` 0x1234 (low address) ✓

`<< &numPtr` 0x ffff (high address) ✓

__/__/____

Memory Heap

```
int main(){
    int *p = new int;
    Cube *c = new Cube;

    *p = 42;
    (*c).setL(2);

    delete c;
    delete p;
    return 0;
}
```



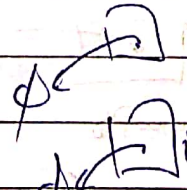
but notice, the pointers still remain even after deleting the heap memory.

nullptr → 0x0 (seg fault)

- → nullptr
- → points to nowhere
- → address is reserved & never used
- → calls to delete 0x0 are ignored.

good practice

```
delete c;      // heap of c is freed. but c pointer still points to the same memory heap garbage
c = nullptr;
```




```
c -> getVolume();
```

same as

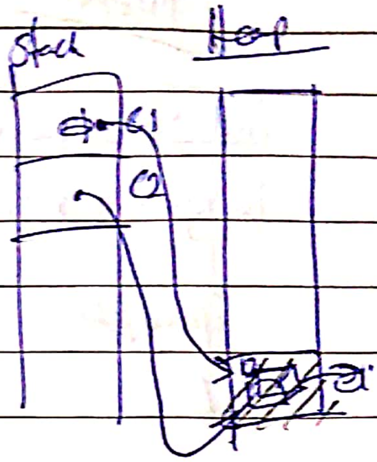
```
(*c).getVolume();
```

Imp

(*). int main() {
Cube* c1 = new Cube;
Cube* c2 = c1;

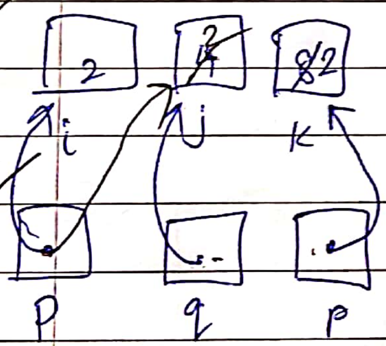
c2 → setLength(10);

delete c2; // memory freed
delete c1; // !!! (error, bcoz we want to double-free
// the memory i.e, free a memory which is already
// freed.
return 0;
}



Heap memory puzzles

Puzzle 1



p ← q

LHS affected p = q; RHS

K = i ; → unaffected

⇒ K ← i

→ p = q

*q = *r

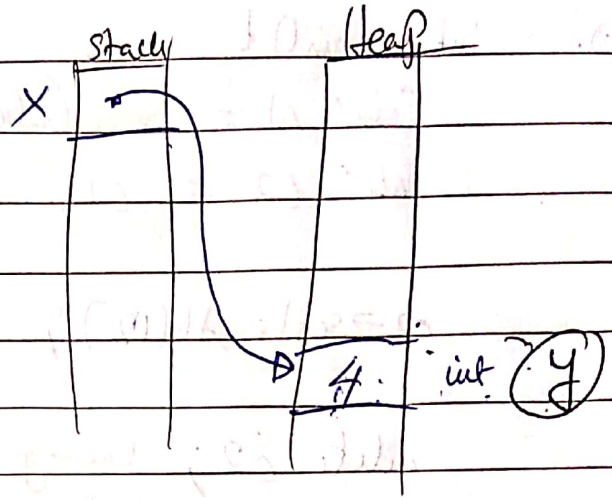
__/__/__

puzzle2

```
int *x = new int;
int &y = *x;
y = 4;
```

(So basically, give a name to ~~x~~ *x)

reference/alias



&x → 0xfff (layer-stack)

x → 0x123 (small-heap)

*x → 4

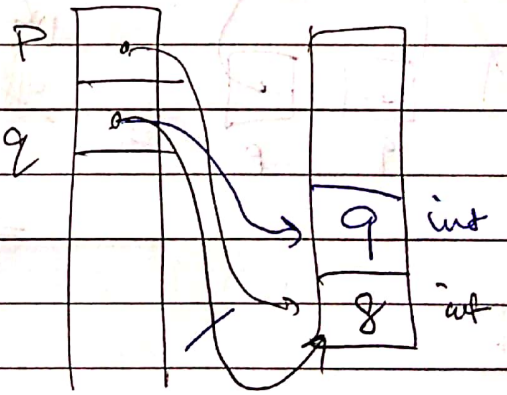
&y → 0x123

y → 4

*y → !!!! (dereference of a non-pointer)

puzzle3

```
int *p, *q;
p = new int;
q = p;
*q = 8;
cout << *p << endl;
q = new int;
*q = 9;
```



*p → 8

*p → 8

*q → 9

Puzzle 4

```
int *x;
int size=3;
x= new int[size];

for (i=0; i<size; i++)
    x[i] = i+3;

delete [] x;
}
```

C++ comments : //

/* ——— */

Header files .h / .hpp

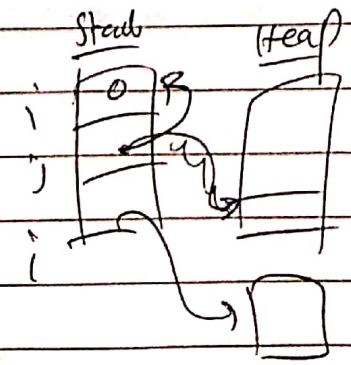
Implementation file .cpp

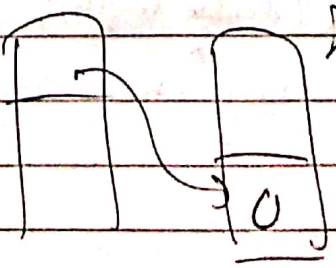
0x0 → address of nullptr

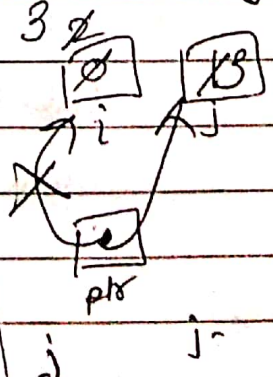
set unused ptrs to this address.

```
int *p = NULL;
  or
int *p = nullptr;
```

#include "abch" → quotes signify that 'abch.h' file is present in the same directory.







__/__/__

Week 3 (Developing C++ Classes)

Class Constructor

- if no constructor, then an automatic default constructor that is present

→ So we use a custom default constructor

default

Cube::Cube()
- same name as that of the class
- no return type
- no parameters

```
class Cube{
    int x;
public:
    Cube();
};
```

```
Cube::Cube(){
    x = 1;
}
```

④ Ctors need not be unique.

```
class Cube {
    public:
        Cube();
        Cube(double length);
};
```

↑ parameterized ctor

__/__/____

→ If any custom constructor is defined, an automatic default ctor is not defined.

Note

```
int main(){
    Cube c;  //!!
}
```

Should call 0 param ctor

"Cube.h"

```
class Cube {
public:
    Cube(double length);
};
```

↳ 1 Param ctor
↳ no default ctor

∴ This results in an error! as C++ is a strongly typed language.

→

Copy Constructors

— The C++ provides an automatic (copy) ctor for free!

— It will copy the contents of all member variables.

Custom copy ctor
↳ class ctor
↳ Has exactly 1 argument.
— The argument must be const reference of the same type as the class.

```
Cube::Cube(const Cube &obj){
    len_ = obj.len_;
```

Copy ctor invocation

1. Passing an obj as a parameter (by val) → anywhere
2. Returning an object from a fn
3. Initializing a new object

(1)
```
int main() {
    Cube c;      ← DEFAULT CTOR
    foo(c);      ← COPY CTOR
}                  ↑ here, an object is passed to as a param.
```

(2)
```
Cube foo() {
    Cube c;      ← DEFAULT CTOR
    return c;    ← COPY CTOR
}
```

```
int main() {
    Cube c2 = foo();   ← COPY CTOR for copying from the main stack frame
                  ||
                  c
}
```

o/p
Default ctor
Copy ctor
Copy ctor

(3)
```
int main() {
    Cube c;          ← DEFAULT CTOR
    Cube myCube = c; ← COPY CTOR
}
```

(4) int main() {

Cube c; ← DEFAULT CTOR

Cube myCube; ← DEFAULT CTOR

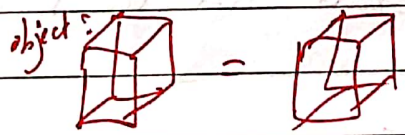
myCube = c; ← ~~COPY~~ Imp

Note, here myCube & C have ~~also~~ already been constructed. So, no more copy ctor here!

} ← simple assignment.

Copy Assignment Operator.

→ a copy assignment operator defines the behaviour when an object is copied ~~or~~ using '=' operator.



→ Copy ctor = Creates a new obj

assignment op = assigns value to existing obj

→ Cube& Cube::operator=(const Cube& obj) {

length = obj.length;

cout << "I am op=" << endl;

return *this;

}

(1) int main() {

Cube c; ← Def

Cube myCube; ← Def

myCube = c; ← Assignment operator

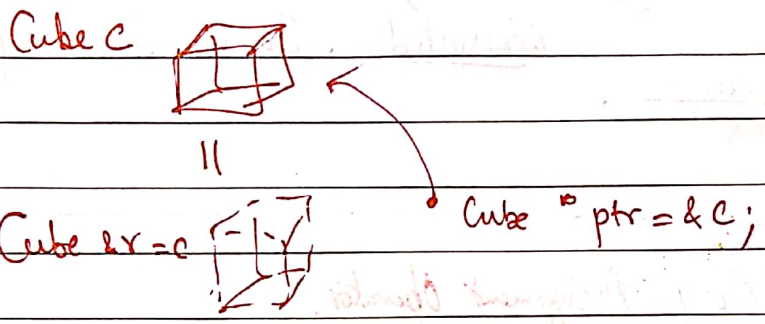
}

__/__/____

Variable Storage

→ An instance of a variable can be stored in 3 diff ways.

1) directly in memory
2) accessed by pointer
3) accessed by reference



① Direct Storage

→ the type of the variable has no modifiers
→ object takes up exact size in memory

```
Cube c;
int i;
minc:: HSLAPixel p;
```

② Storage by Pointer

- type modified with (*)
- a pointer takes a "mem-address width" of memory
  eg → 64 bits on a 64-bit architecture

- ptrs

```
Cube * c;
int * i;
minc:: HSLAPixel * p;
```

③ Storage by Reference

- a reference variable is an alias to an existing memory
- It does NOT store memory itself, it is only an alias to another variable
- alias must be assigned during initialization (Imp)

```
Cube &e = cube;
int &i = count;
uiuc::HSLAPixel &p;  // X wrong
```

↳ must alias something when variable is initialized.

example — Cube currency

eg → if we transfer cube currency, we transfer it by reference otherwise we might cause inflation.

④. Transfer by Value

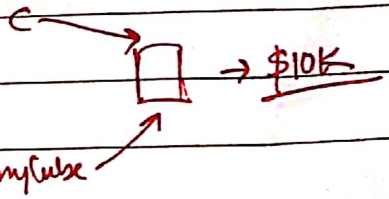
```
Cube c(10);   <- def
Cube myCube = c;   <- copy
```

10 → total = $20K

10 → So NOT feasible

⑤ Transfer by ref

```
Cube c(10);
Cube & myCube = c;
```

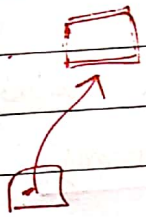


__/__/____

by Pointer

→ Cube c(10);

→ Cube* myCube = &c;



→ only $10K

→ send by value

```
Cube c(10);
sendCube(c);
```

```
bool sendCube(Cube c) {
}
```

↑ copy

→ send by Ref

```
Cube c(10);
sendCube(c);
```

```
bool sendCube(Cube &c) {
}
```

↑ ref

→ send by Pointer

```
Cube c(10);
sendCube(&c);
```

```
bool sendCube(Cube *c) {
}
```

↙ only 1 copy present

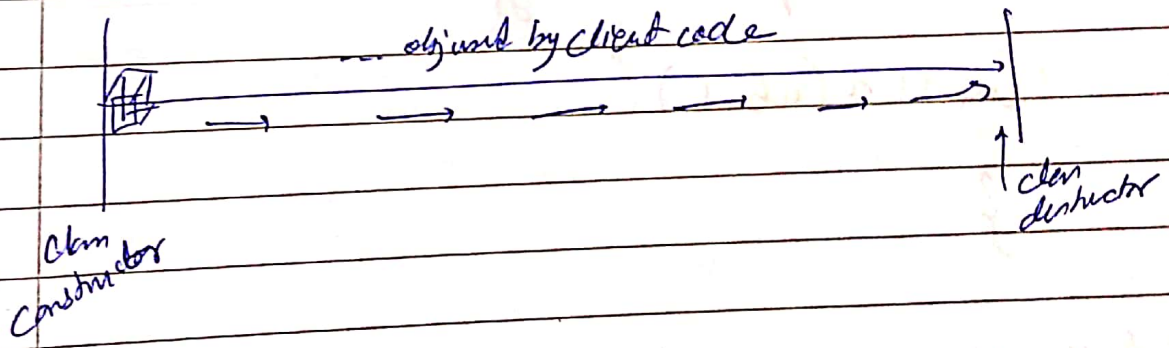
Return by ——

→ Return by value (def)
→ by pointer (*)
→ by reference (&)

!! – Never return a reference to a stack variable created on the stack of your current func.

Class Destructor.

→ When an instance of a class is cleaned up, the class destructor is the last call in a class's lifecycle



→ An automatic default Dtor is added to your class if no other dtor is defined.

→ The only action of the automatic default dtor is to call the default dtor of all member objs.

(R). Automatic default dtor is never called directly. It is automatically called when obj's memory is being reclaimed by sys.

- If obj is on the stack, when the func returns
- If obj is on the heap, when delete is used

→ Custom dtor

↳ a member fn
↳ Same as name of the class, preceded by ~ (tilde)
↳ 0 args, NO return type

```
Cube :: ~Cube ()
{
}
```

Ex:

```
double cube-on-stack() {
    Cube c(3); ← CTOR
    return c.getVolume();
}

void cube-on-heap() {
    Cube *c1 = new Cube(10); ← CTOR
    Cube *c2 = new Cube; ← CTOR
    delete c1; ← DTOR (c1)
}
```

```
int main() {
    cube-on-stack();
    cube-on-heap();
    cube-on-stack();

    return 0;
}
```

at program exit, dtor!

NO DTOR ← by heap memory is not destroyed when we use delete keyword.

__/__/____

Custom dtor

- Heap memory
- Open files
- Shared memory

for loop – temp & reference

```
for (int x : arr)
    x = 99;
}
```

↑ copy of arr value,

↳ D.N affect the original array

```
for (int& x : arr)
    x = 99;
```

↓ arr value directly affected

reference of arr value

→ for using both reference + but not changing, use a constant reference.

→ `for (const int& x : int_list) {`

`const cc xcc ewll;`

`x = 99; Error !!!`

`}`

__/__/____

Engineering C++ Software Solutions (Week 4)

↳ alligator braces

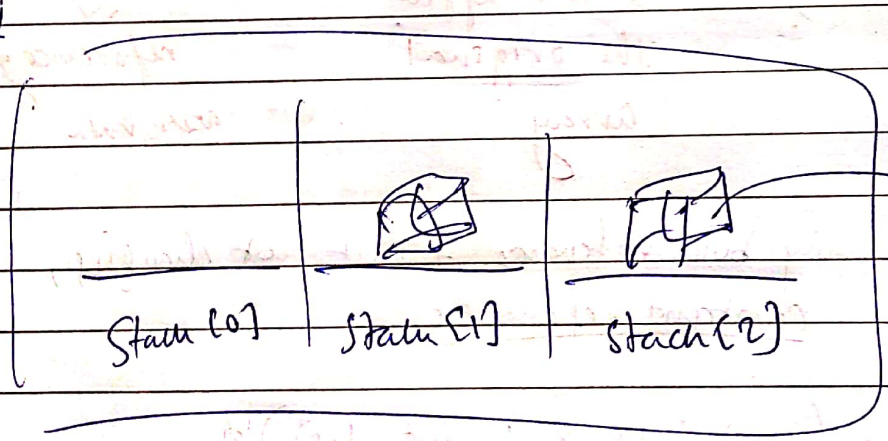
→ Tower of Hanoi

Template types

```
std::vector<char> v1;
```

Tower of Hanoi

Game



Templates & Classes

→ Template functions

→ A template variable is defined by declaring it before the beginning of a class/function.

```
template <typename T>
class List {
    private:
        T data;
};
```

```
template <typename T>
int max (T a, T b) {
    if (a > b) return a;
    return b;
}
```

→

(*) Templated variables are checked at compile-time, which allows for errors to be caught before running the program.

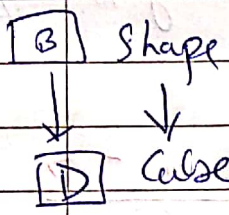
```
max (4, 7);                          ✓
max (Cube(3), Cube(5));              ✗
```

Error! because we haven't overloaded '>' operator.

```
max ('a', 'd');
```

Inheritance

→ Inheritance allows for a class to inherit all member functions & data from a base class into a derived class.



~~class~~ Cube must construct Shape via initializer list

→ Cube::Cube(double width, int col) : Shape(width) {
    color_ = color;
}

Access Control

When a base class is inherited, the derived class:

→ Can access all public

→ Cannot access private

INITIALIZER LIST

→ The syntax to initialize the base class is called the initializer list

→ Initialize a base class

→ Can " " current class using another constructor

→ " " default values of member vars.

pixel.l — $\frac{decx}{100}$ 2% $\frac{(100-2)}{100} \times$ pixel.l

__/__/___

1 PARAMETER

→ Shape :: Shape( ) : Shape(1) {
// nothing
}

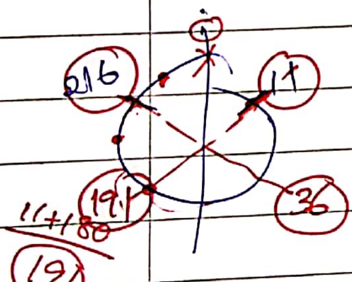
initilizer list

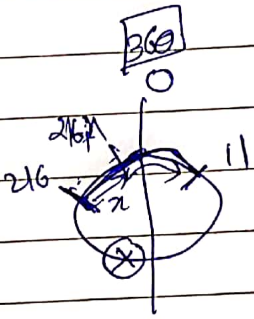
→ Shape :: Shape(double width) : width_(width) {
//Nothing
}

1) h = 11 (Illini Orange)
h = 216 (Illini Blue)

1 → 0.5
5 → 2.5

2) spotlight
(x, y)





216
180
36

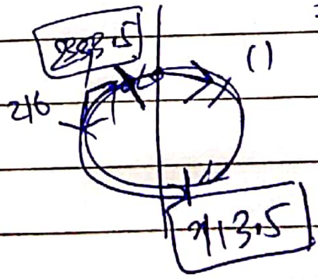
(216 + x)

$x = 11 + (360 - 216 - x)$

$2x = 371$
$- 216$

$2x = 155$

$x = \frac{155}{2}$ 77.5



216
77.5
333.5

216
77.5
293.5

$x - 11 = 216 - x$

$2x = 227$ 113.5

216
11
27