Pointers & Dynamic memory management

Why do we learn pointers?

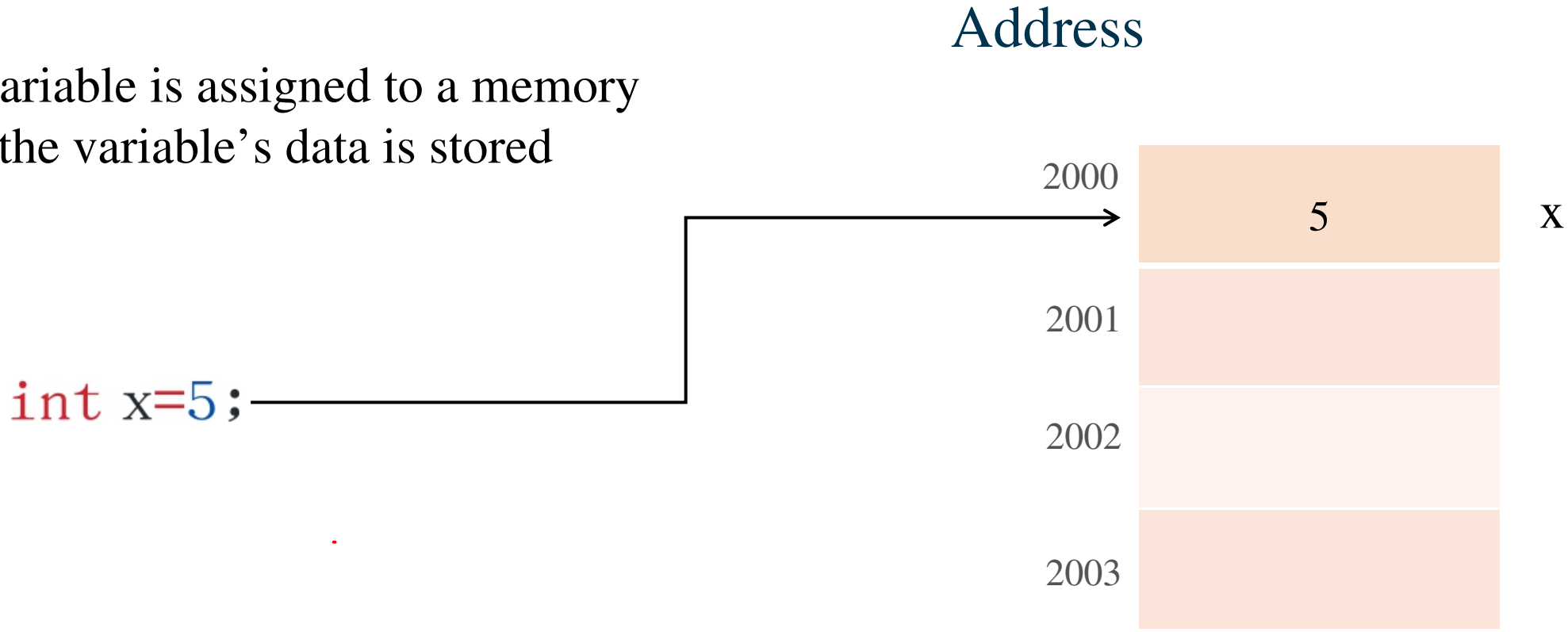
Pointers provide you with alot of flexability; as they give you a great control over memory, so they are used for alot of things like:

- creating dynamic arrays `int arr[5];` `int arr[x];`
- implementing data structures
- important for understanding & using oop concepts(inheritance & polymorphism)
- ...etc

so to sum it up, they are essential for dealing with dynamic related behaviour and that's why our second topic of today's sessions will be about "Dynamic memory management"

Computer memory

- Each variable is assigned to a memory slot & the variable's data is stored there



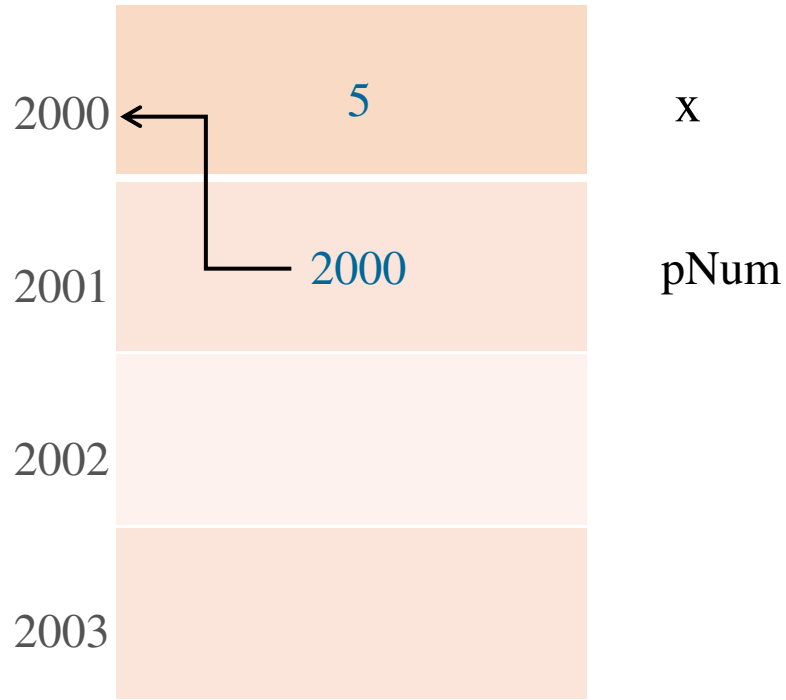
What are pointers ?

A pointer is :

1. variable
2. stores the adress of the pointed memory cell

```
int x=5;
int* pNum=&x;
```

3. Can be used to reference to this memory cell



What are pointers ?

Example 1 on using pointers:

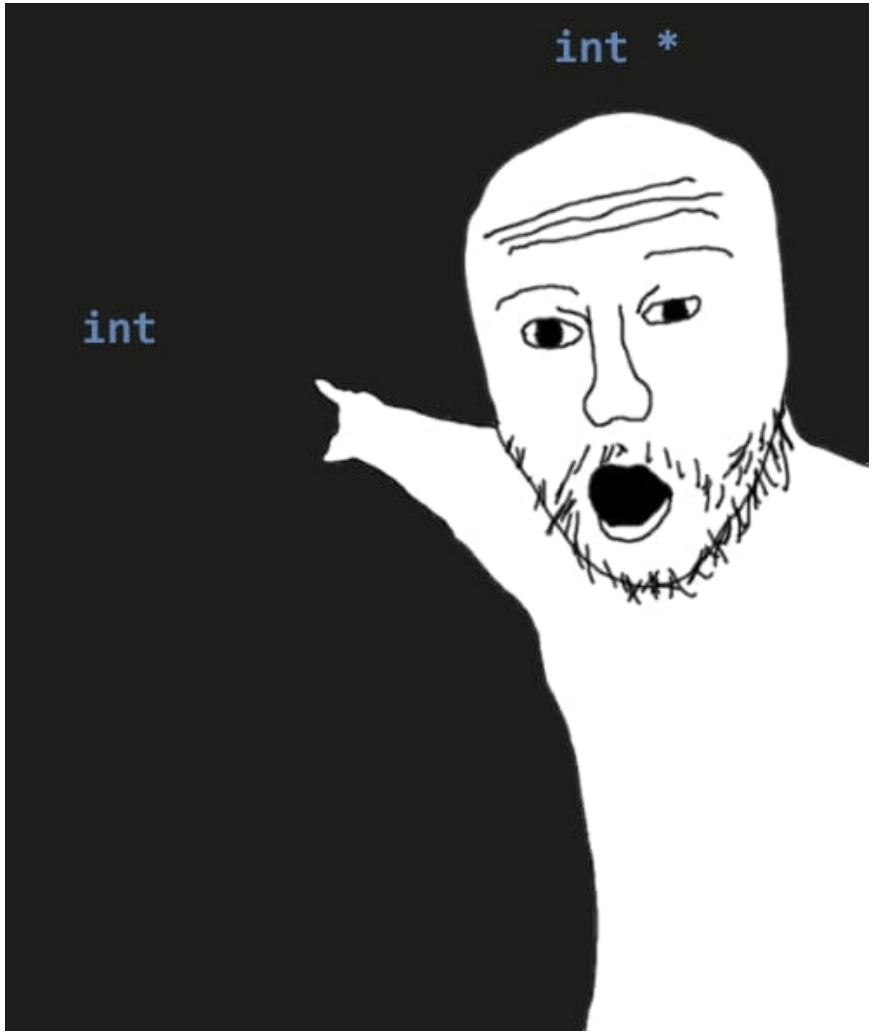
```
int x=5;
int* pNum=&x;
*pNum=0;
cout <<"x= "<<x<<endl;
cout <<"&x= "<<&x<<endl;
cout <<"pNum= "<<pNum<<endl;
cout <<"*pNum= "<<*pNum<<endl;
```

&= adress in memory (hex number)

*pNum means the value pointed at by pNum (equivlent to x=0;)

Run

What are pointers ?



Using Pointers.

Example 2:

```
int count=100;
char l='M';
short num=5;
Person person1;

int* pCount= &count;
char* pL= &l;
short* pNum= &num;
Person* pPerson1= &person1;
```

count	100
l	M
num	5
person1	.....
pCount	address of count
pL	addresss of l
pNum	address of Num
pPerson1	address of person1

Using Pointers.

Notes:

(*) was used 2 times : -on declaration (int* pCount)

- when using pointers[Derefrencing] (*pCount=&count)

Pointer to a Pointer.

```
int a=8;
int* pa=&a;
int** ppa=&pa;
cout<<"a= "<<a<<endl;
cout<<"pa= "<<pa<<endl;
cout<<"*pa= " << *pa <<endl;
cout<<"ppa= "<<ppa <<endl;
cout<<"*ppa= "<< *ppa<<endl;
cout<<"**ppa= " << **ppa<<endl;
```

Variable	Address	Value
a	0x7fff3b3ad824	8
pa	0x7fff3b3ad828	0x7fff3b3ad824
ppa	....	0x7fff3b3ad828

RUN

Becareful.

-only use suitable pointer datatype to avoid errors.

(Wrong: int count; double* pCount= &count)

-Always initialize pointers

```
int* pCount= 0;

int* pCount =NULL;

int* pCount= nullptr; (for c++ 11 or above) (prefered to use)
```

Arrays and Pointers

The variable name of an array is actually a pointer to the first element of the array

```
int Arr[4]={0,1,2,3};
cout<< Arr<<endl;
cout<< *Arr<<endl;
```

RUN

Arrays and Pointers

Passing Arrays to functions using 2 methods :

a)
```
void PrintArray(int* arr, int size){
for(int i=0;i<size;i++) cout<<arr[i]<<endl;
}
```

b)
```
void PrintArray(int arr[], int size){
for(int i=0;i<size;i++) cout<<arr[i]<<endl;
}
```

Arrays and Pointers

Example:

```
int* AddOne(int* arr, int size)
{
for(int i=0; i<size; i++){ arr[i]++;}
return arr;
}
```

The Problem with this is that we make the changes to the original array.

Becareful.

when using Pointers with a function:

-The function shouldn't return a pointer to a local variable.

The function should return a pointer:

to data that was passed to the function as an argument.

to dynamically allocated memory.

Part 2: Dynamic Memory Mangement.

Static vs Dynamic Memory

Static Memory: Memory is allocated at compilation time.

Dynamic Memory: Memory is allocted and deallcoted at running time.

Dynamic variable

```
int* DM=new int; //allcation
delete DM; //deallocation
```

Dynamic Memory

Example :

```
int* pNum=nullptr;
pNum=new int;
*pNum=5;
cout<< *pNum<< endl;
delete pNum;
cout<< *pNum;
```

RUN

Dynamic Array

```
int* DA=new int[size]; //allcation
delete[] DA; //deallocation
```

Arrays and Pointers

Example:

```
int* AddOne(int* arr, int size)
{

for(int i=0; i<size; i++){
   arr[i]++;
   }
return arr;
}
```

The Problem with this is that we make the changes to the original array.

Dynamic array

Example:

```
int* AddOne(int* arr, int size)
{
int* pArr = new int[size];
for(int i=0; i<size; i++){
   pArr[i]=arr[i]+1;
   }
return pArr;
}
```

The Problem is now fixed.

Dynamic Objects

```
string* DynamicObject=new string(); //no argument construction

string* DynamicObject=new string; //no argument construction

string* DynamicObject=new string("random"); // construction with argument
```

Static vs Dynamic Memory

	Static Memory	Dynamic memory
Time of allocation	**allocated at compilation time**	**allocated at run time**
Scope	**defined automatically**	**defined Manually(using the words new and delete)**
expansion and shrinkage	**It can't expand or shrink during run time**	**expands and shrink**

Back to OOP

```
Circle::Circle(){
radius=1;
}
Circle::Circle(double radius){
this->radius = radius;
}
```

The keyword “this” is a built-in pointer that points to the calling object

Note: (this -> radius) is the same as saying (*this.radius)

Destructors

Remeber that:

-A function that is called when an object is destroyed

-Destructor name is ~className, e.g, ~Circle();

-It has no return type, takes no arguments

-Cannot be overloaded

now we will learn that:

-if constructor allocates dynamic memory, destructor should deallocate it

Dynamic Memory

```
class trial{
  private:
    int* num;
  public:
    trial(){
      num=new int;
      *num=5;
      cout<<"This is the constructor,
   *num= "<<*num<<endl;
}
  ~trial(){
      cout<<"this is the destructor, *num
   was equal to "<<*num<<endl;
      delete num;
      cout<<"*num is equal to "<<*num;
}
};
```

```
int main(){
   trial t1;
   cout<<"object hasn't been destructed
            yet"<<endl;
}
```

RUN

Dynamic Memory

```
class trial{
  private:
  public:
   trial(){
    int* num=nullptr;
    num=new int;
    *num=5;
    cout<<"This is the constructor,
  *num= "<<*num<<endl;
}
 ~trial(){
    cout<<"this is the destructor, *num
  was equal to "<<*num<<endl;
    delete num;
    cout<<"*num is equal to "<<*num;
}
};
```

```
int main(){
  trial t1;
  cout<<"object hasn't been destructed
        yet"<<endl;
}
```

Will it work???

RUN

Dynamic Memory

```
class trial{
   private:
   public:
     trial(){
      int* num=nullptr;
      num=new int;
      *num=5;
      cout<<"This is the constructor,
   *num= "<<*num<<endl;
}
  ~trial(){
      cout<<"this is the
      destructor”<<endl;
}
};
```

```
int main(){
   trial t1;
   cout<<"object hasn't been destructed
          yet"<<endl;
}
```

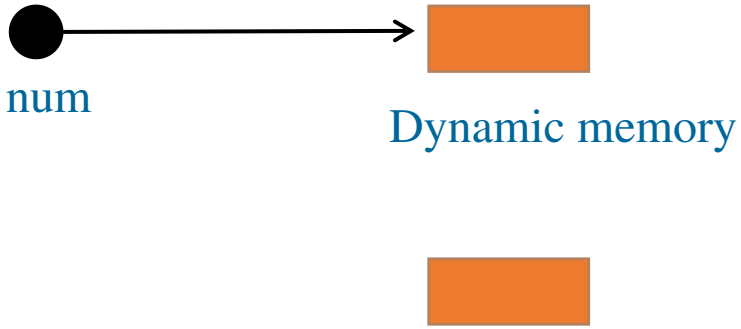
Will it work???

RUN

Destructors

Memory leak:

-The previous code worked but it contains a problem.



Forgetting to deallocate memory
will result in a **Memory leak.**

How to delete dynamic memory in a function

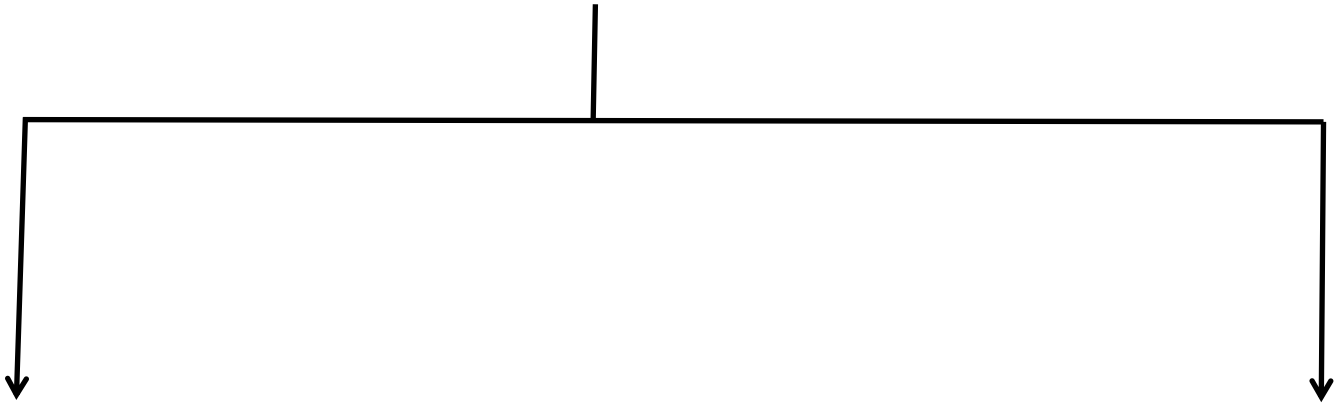
Remember this:

```
int* AddOne(int* arr, int size)
{
int* pArr = new int[size];
for(int i=0; i<size; i++){
   pArr[i]=arr[i]+1;
   }
return pArr;
}
```

```
int main(){
int* pArray= AddOne(Arr,4);
//do something with it
delete[] pArray;
}
```

Copy Constructor.

-It creates an object using another object's attributes (copying them).



Default Copy Constructor
(copies all attributes)

Overloaded Copy Constructor
(customaized one)

Default Copy Constructor.

-you don’t have to implement it

in order to use it: ClassName object2(object1);

object2:object you wan to create

object1: object to copy from

Note: object2 and object1 are 2 independent objects

Default Copy Constructor

```
class Circle{
private:
   double radius;
public:
   Circle(){
        radius=1;
}
   Circle(double radius){
        this->radius=radius;
}
   double getRadius(){
        return radius;
}
   void setRadius(double radius){
        this->radius=radius;
}

   ~Circle(){}
};
```

```
int main(){
   Circle c1(10);
   Circle c2(c1);
   cout<< c2.getRadius()<<endl;
```

RUN

```
    c1.setRadius(15);
    cout<< c1.getRadius()<<endl;
    cout<< c2.getRadius()<<endl;
    }
```

RUN

Default Copy Constructor

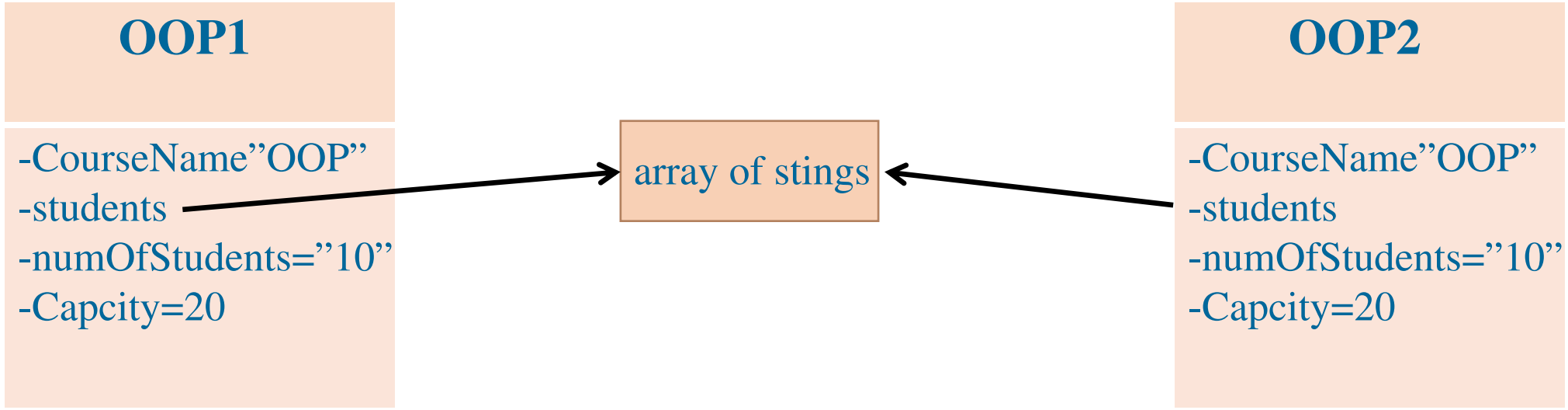
```
class Course{
private:
string courseName;
string* students;
int numOfStudents,capacity;
public:
//adding some functions
};
```

```
int main(){
   Course OOP1();
   Course OOP2(OOP1);}
```

Now that we have a dynamic array, how does it affect the copy constructor operation?

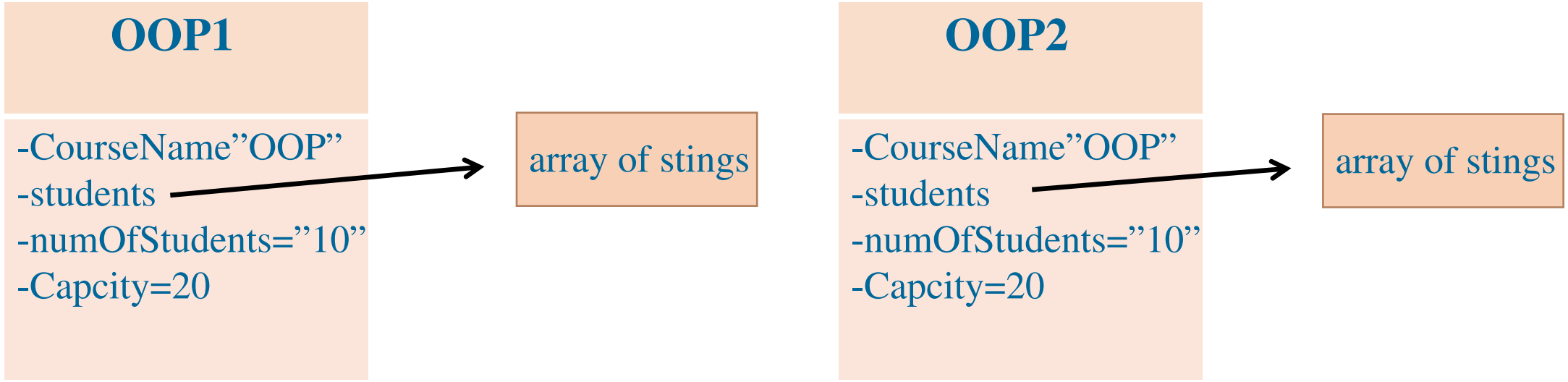
Shallow Copy

- OOP2 copies from OOP1, so the pointer in OOP2 now points to the same address in memory as OOP1



Deep Copy

- But what we want in our case is



Overloaded Copy Constructor.

-you have to implement it

-The function prototype is like this: ClassName (const ClassName&);

Notes:

-call by reference is used here for efficiency

-(const) is used as you don't want to make changes to the object passed

Overloaded Copy Constructor

```
class Course{
private:
string courseName;
string* students;
int numOfStudents,capacity;
public:
Course(const Course& x);
//adding some functions
};
```

```
Course::Course(const Course& x){
   this-> courseName= x.courseName;
   this->numOfStudents=x.numOfStudents;
   this->capacity=x.capacity;
   this-> students=new string[capacity];
    //if you want, you can copy the
   students array members in a loop
}
```

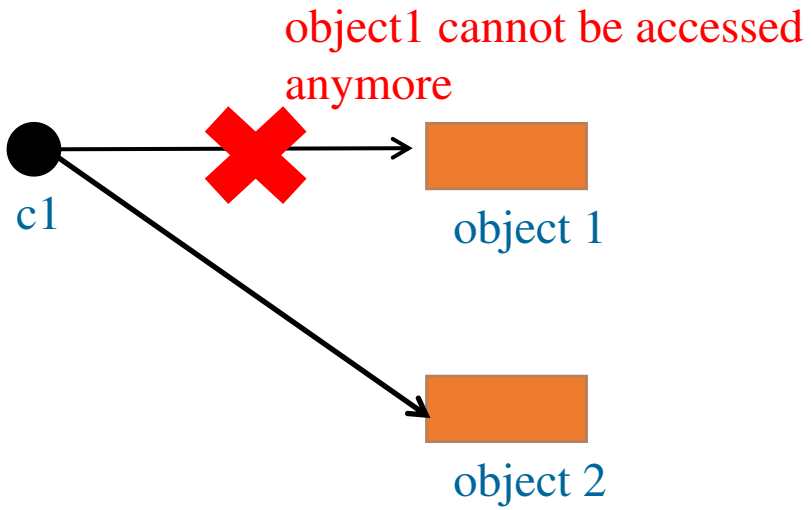
Problems with pointers

Memory leak:

-we have already talked about it, but let's show some common mistakes that lead to Memory leaks

```
Circle* c1=new Circle;
```

```
c1=new Circle;
```

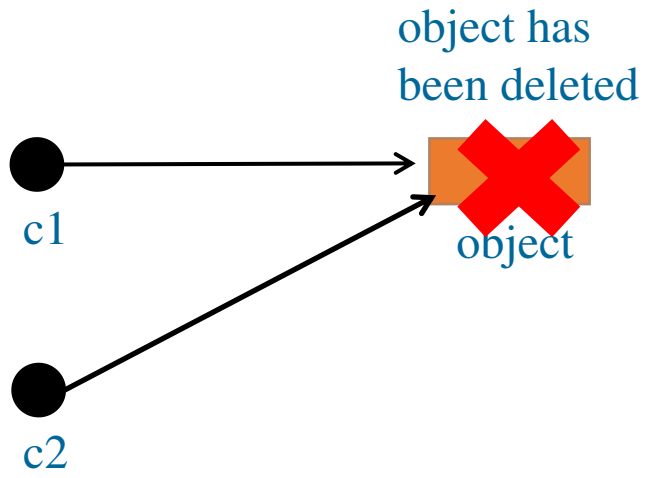


Problems with pointers

Dangling pointer:

```
Circle* c1=new Circle;
Circle* c2= c1;
```

```
delete c1;
cout<<c2.getRadius(); //invalid
```



Extras.

Google for (shared pointers), (unique pointers)

Thank you

presented by: Omar Khaled