ECE 244, Lecture 7

Last lecture: Classes

Today: Constructors, and dynamic memory allocation in objects

Student.h

```
class Student {
    private:
        int ID; string name;
    public:
①must be public  →  Student();  - Constructor
        void setName(string n );
        string getName( );
        void print();
};
```

Student.cpp

```
                      ②same name as class
    Student::Student(){
        ID = 0;               } Typically used to initialize
        name = " ";             data members of a class
    }
③no return
  type
```

main.cpp

```
int main( void ){
    Student x;        //constructor called

    Student y[10];    // constructor called 10 times

    Student* z;  ←  // no constructor called
                 ←  // no object is instantiated
}
```

What if I want to initialize ID with a specific value?

We can have multiple constructors

Student.h

```
    class Student {
        private:
            int ID:
            string name;
        public:
            Student();
            Student (int id);
            Student (int id, string name);
                :
    };
```

multiple constructors:
Same function name
different arguments
"Function overloading"

Student.cpp

```
    Student:: Student () {
        ID = 0;     name = " ";
    }
    Student:: Student (int id ) {
        ID = id;    name = " ";
    }
    Student:: Student (int id, string n ) {
        ID = id;
        name = n;
    }
```

main.cpp

```
      Student x;                      - Default constructor
      Student y(2307);                - 2nd constructor
      Student z(8731, "Osiris");      - 3rd constructor
```

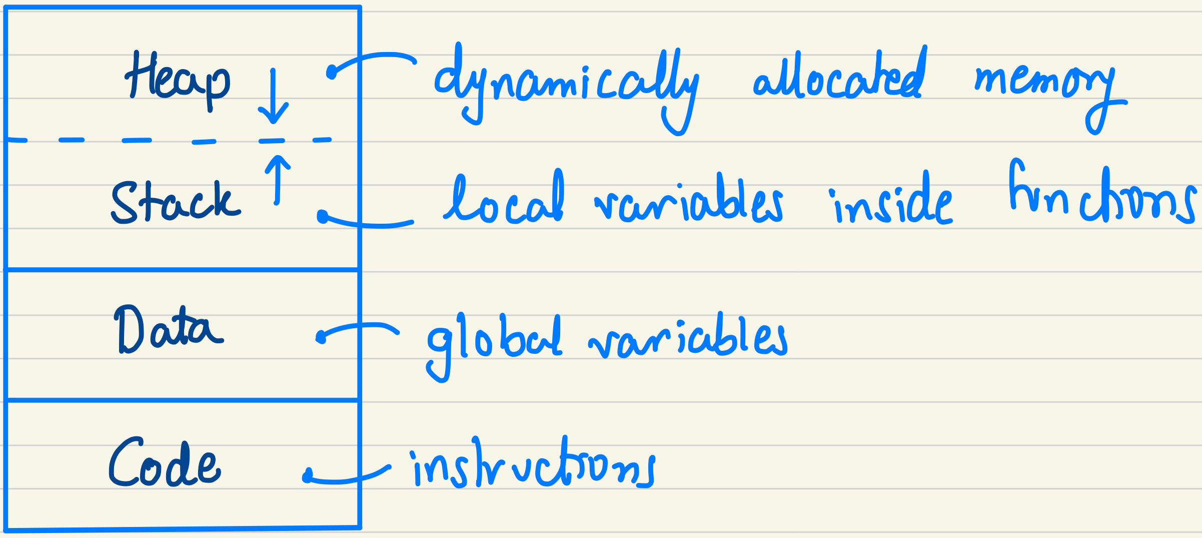
Respective constructors are called depending on arguments

V. Important! If default constructor Student() is not implemented, but Student(int) is implemented, then Student x; will cause an error as it will call Student() that is not defined.

Dynamically allocated memory in an object

Recall dynamic memory allocation!

A program's memory space



Heap ↓
Stack ↑
Data
Code
dynamically allocated memory
local variables inside functions
global variables
instructions

Memory on stack gets freed when a function returns. All local variables in a function disappear when the function returns or when they go out of scope.

BUT Memory allocated on the heap dynamically has to be explicitly **freed**. It doesn't get freed when a variable goes out of scope. It is memory leak if we don't free.

E.g.

```
int x ;
int *p;
x = 7;
p = NULL;
p = &x;
```

& — address of

	Memory	Address
x	~~7~~ 5	0x120
p	~~NULL~~ ~~0x124~~ 0x560	0x124
	3	0x560

`cout << *p;` prints 7

* — dereference p / value at address in p

`cout << p;` prints 0x 124

`*p = 5;` changes value of x to 5

`p = new int;` change address stored in p to a newly allocated memory space

`*p= 3;` change value at address 0x560

⋮

Before exiting our program, we need to return dynamically allocated memory to operating system.

Recall in C, for every malloc there has to be a free.

```
delete p;        p now has address of expired data
p = nullptr;     Good practice!
```

In C++, we have new and delete.

Integer

```
int * pNum = new int;
delete pNum;
```

de-allocate memory at pNum

return address of an int variable created at run-time!

Array

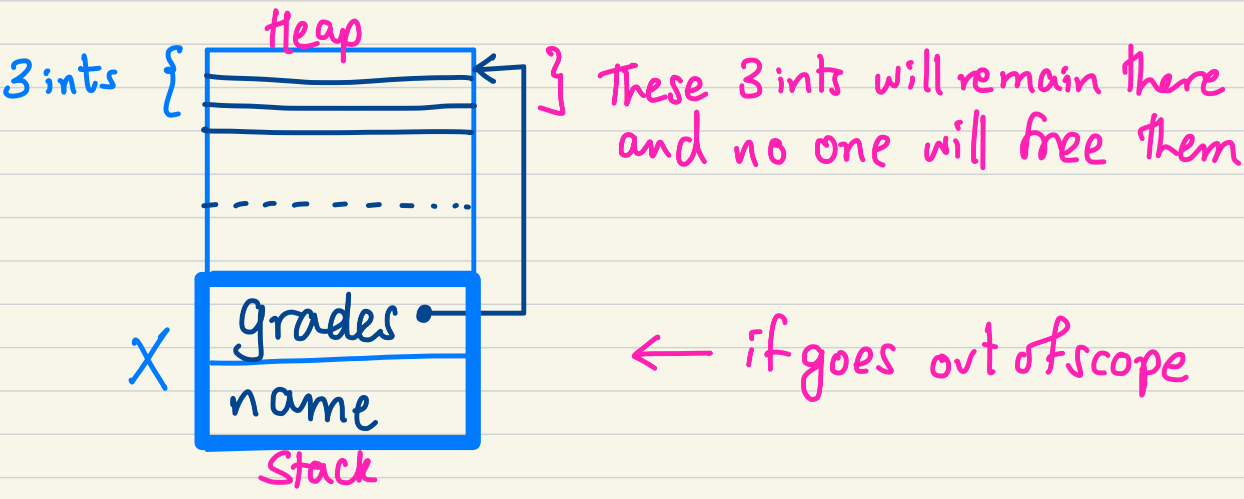
```
int * arr = new int[10];
delete [] arr;
```

Example in a class

```
class Student {
        private:
                int * grades;
                string name;
        public:
                Student ();
                Student (int );
};

Student :: Student () {
        grades = nullptr;
}
Student :: Student (int numLabs) {
        grades = new int[numLabs];
}
int main ( void ) {
        Student x (3);        → dynamically allocates
                                3 integers.
        return 0;
}
```

We didn't de-allocate them! — Memory leak



These 3 ints will remain there and no one will free them!

← if goes out of scope

Solution is in defining destructors

A destructor is the complement of constructor.
It's automatically called when an object is destroyed/goes out of scope. It's empty if it is not defined

If you dynamically allocate memory in your class, you will need a destructor to free up this memory space.

Student.h

```
class Student {
        private:
              int * grades;   string name;
        public:
Must be public        Student();
(like constructors)  Student(int);
                     ~Student(); — no return (like
                             |              constructors)
};                    no parameters
```

Student.cpp

```
        :
        :
Student :: ~Student () {
        if (grades != nullptr) {
              delete [] grades;
        }
}
```