

# Object Oriented Programming (CS1004)

## Lecture 11

### Dynamically allocating objects and Destructors Arrays of Objects

```
class Point {  
private:  
    int x, y; // Private data members
```

public:

Point(int x = 0, int y = 0); // Constructor with default arguments

int getX() const; // Getter

void setX(int x); // Setter

int getY() const;

void setY(int y);

void setXY(int x, int y);

void print() const;

};

## Arrays of Objects

// Constructor - The default values are specified in the declaration

Point::Point(int x, int y) : x(x), y(y) { }

int Point::getX() const { return x; }

int Point::getY() const { return y; }

```
void Point::setX(int x_) { x = x_; }
```

```
void Point::setY(int y_) { y = y_; }
```

```
void Point::setXY(int x_, int y_) { x = x_; y = y_; }
```

```
void Point::print() const {  
    cout << "Point @ (" << x << ", " << y << ")";  
}
```

# Arrays of Objects-Static Memory

## Allocation • Static Memory Allocation

- Use default constructor for all elements of the array

```
Point ptsArray1[2]; // Array of Point objects
```

```
ptsArray1[0].print(); // Point @ (0,0)
ptsArray1[1].setXY(11, 11);
ptsArray1[1].print() // Point @ (11,11)
```

## Arrays of Objects-initializing objects using non-default constructor

- **Static Memory Allocation**
- Initialize array elements via non-default constructor

```
Point ptsArray2[3] = { Point(21, 21), Point(22, 22), Point() };
```

```
ptsArray2[0].print(); // Point @ (21,21)
```

```
cout << endl;
```

```
ptsArray2[0].print(); // Point @ (0,0)
```

## Arrays of Objects-creating objects

**dynamically • Array of Object Pointers - Need to allocate**

**elements dynamically**

```
Point * ptrPtsArray3 = new Point[2]; //objects created using default
```

```
constructor ptrPtsArray3[0].setXY(31, 31);
```

```
ptrPtsArray3[0].print(); // Point @ (31,31)
```

```
cout << endl;
```

```
ptrPtsArray3[1].setXY(32, 32);
```

```
ptrPtsArray3[1].print(); // Point @ (32,32)
cout << endl;
```

```
delete[] ptrPtsArray3; // Free storage
```

## Destructors

- Destructors

- used to destroy the objects that have been created by a constructor are member function of class
- called automatically when local objects go out of scope
- perform termination housekeeping before the system reclaims the object's memory
- Complement of the constructor

- Destructor naming:

- Name is tilde (~) followed by the class name (i.e., ~Time)
  - Recall that the constructor's name is the class name
- Receives *no parameters*, returns *no value*
- One destructor per class

- *No overloading* allowed

- If you do not explicitly provide a destructor, the compiler will create an empty destructor.

## Syntax for user-defined Destructor

```
Classname::~~Classname() {  
  
}
```

- Cleanup is as important as initialization and is guaranteed through the use of destructors.

- Destructor never has any arguments, because it does not need any options.

## Constructor/Destructor Example class

```
Employee {  
    public:  
    Employee () {  
        cout << "Employee's class object is  
        created"<<endl; }  
  
    ~Employee () {  
        cout << "Employee's class object is  
        deleted"<<endl; }  
};  
void main (void)
```

```
{
    Employee emp;
} // destructor will call here
```

**Program Output:**

Employee's class object is created

Employee's class object is deleted

## Simple Destructor Example

```
class Rectangle {
public:
    Rectangle(int w=5,int l=10){
        Width=w; length=l; }
    ~Rectangle() {
        cout<<"Rectangle object being destroyed: "; private:
        int width;
        int length;
```

```
Rectangle width = 5 and lenth = 10
Rectangle width = 2 and lenth = 10
```

```
};  
int main() {  
  
    Rectangle r1;  
    Rectangle r2(2,40);  
    Rectangle r3(3,60);  
    //Destructors for objects are implicitly called here.  
}
```

## Destructor Example: For array of objects

- The default constructor must be provided for creating array of objects:
- Destructor will be called automatically.
  - There is no need to call destructor explicitly.

```
Int main() {
```

```
Employee Emp[3]; //it will create three objects. Only  
default constructor will be called
```

```
Emp[0].setID(1);
```

```
Emp[1].setID(2);
```

```
Emp[2].setID(3);
```

```
} //destructors will be called implicitly
```

02/27/24

## When is a destructor called for Dynamically allocated Object?

- Destructors shall not be explicitly called.

```
DayofYear::~~DayofYear() {  
    cout<<"DayofYear object being destroyed: "<<endl  
}  
main() {  
    DayofYear *D1 = new DayofYear(10,3,2024);  
} // what happens??
```

## When is a destructor called for Dynamically allocated Object?

- Destructors shall not be explicitly called.
- The destructor `DayofYear::~~DayofYear()` will automatically get called when you delete it using  

```
delete D1;
```
- Remember: `delete D1` does two things: it calls the destructor and it deallocates the

memory.

```
DayofYear::~~DayofYear() {  
    cout<<"DayofYear object being destroyed: "<<endl  
} main() {  
    DayofYear *D1 = new DayofYear(10,3,2024);  
    ...  
    delete D1; //Automatically calls D1->~DayofYear()  
    ...  
}
```

**Destructor Example: Array of pointers of type rectangle** `class`

```
Rectangle {  
public:  
    Rectangle(int w=5,int l=10) {  
        Width=w;,length=l; }  
    ~Rectangle() {
```

```
cout<<"Rectangle object being destroyed: ";
```

```
private:
```

```
int width;
```

```
int length;
```

```
};
```

```
int main() {
```

```
{
```

```
Rectangle *rec[3];
```

```
for(int i=0;i<3;i++)
```

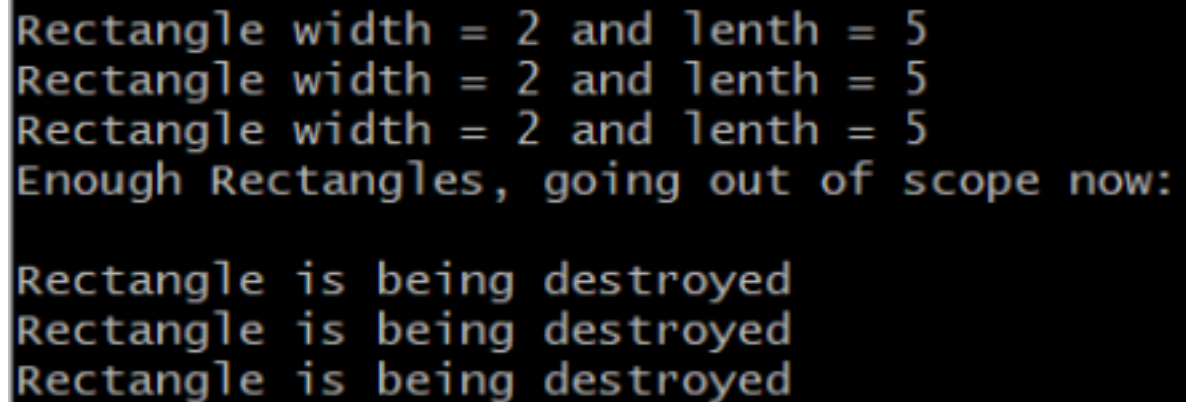
```
    rec[i] = new Rectangle(2,5);//invoking constructr for each obj.
```

```
cout<<"Enough Rectangles, now going to out of scope: "<<endl
```

```
for(int i=0;i<3;i++)
```

```
    delete rec[i];//destructors for the dynamically allocated objects  
    called here automatically.
```

```
}
```



```
Rectangle width = 2 and lenth = 5  
Rectangle width = 2 and lenth = 5  
Rectangle width = 2 and lenth = 5  
Enough Rectangles, going out of scope now:  
  
Rectangle is being destroyed  
Rectangle is being destroyed  
Rectangle is being destroyed
```

## Destructor Example: For dynamically allocated array of objects

- The default constructor must be provided for creating array of objects using below

method:

- Use `delete` for deallocating the dynamic array of objects.
- Without using `delete`, destructors will not be called.

```
Employee *c = new Employee[3]; //it will create three  
objects. Only default constructor will be called
```

```
Emp[0].setID(1);
```

```
Emp[1].setID(2);
```

```
Emp[2].setID(3);
```

```
delete [] c; //destructor will call here
```

# Class exercise#1

- Write a class myobj. The constructor will increment the variable count, number of objects, each time a new object is created and will print it on screen.
- Similarly, destructor will decrement the variable count, number of objects, each time an object goes out of scope.

```
class myobj{  
    Static int count;  
    public:  
    myobj () {  
        count++;  
        cout<<"No. of Object  
        created:"<<count; }  
    ~myobj () {
```

```

cout<<"No. of Object
destroyed:"<<count; --count; }

};

int myobj::count=0;

main() {
    myobj Obj,Obj1,Obj2,Obj3;
}

```

## Object holding variable on the heap

- When an object is deallocated, the dynamic storage that it "owns" must also be deallocated.
- this will not happen by default; we need to explicitly deallocate dynamic storage using `delete` or `delete[]`.

```

class dynamicvar {
private:
    int * ptr;
public:
    dynamicvar(int n) {

```

```

    cout<<"in constructor: Allocating variable on the heap: "<<endl;
    ptr = new int;
    *ptr = n;

}

~dynamicvar() {
cout<<"In Destructor: Dynamically allocated variable is being deleted:";
delete ptr;
};

int main() {
dynamicvar D1(5);
}

```

## Object holding variable on the heap

- When an object is deallocated, the dynamic storage that it "owns" must also be deallocated. • This will not happen by default; we need to explicitly deallocate dynamic storage using `delete` or `delete[]`.

```

class dynamicvar {
private:
    int * ptr;

```

```

public:
    dynamicvar(int) {
        cout<<"in constructor: Allocating variable on the heap: "<<endl;
        ptr = new int;
        *ptr = n;}
    ~dynamicvar(){
        cout<<"In Destructor: Dynamically allocated variable is being deleted:";
        delete ptr;
    };

int main() {
    dynamicvar *objptr = new dynamicvar(5);

    Delete objptr;
}

class myarray {
private:
    int * Arr;
    int size;
public:

```

```
    myarray(int); // constructor
    ~myarray(); // destructor
};

myarray::myarray(int n) {
    cout<<"In Constructor: Array is being allocated dynamically";
    Arr = new int[n];
}

myarray::~~myarray() {
    cout<<"In Destructor: Dynamically allocated array is being deleted:";
    delete[] Arr;
}

int main() {
    myarray A1(10);
}
```