# Object Oriented Programming

**Using C++ Programming Language**

# RECAP

Information hiding

Encapsulation

Implementation

Interface

Messages

Abstraction

# Lecture # 5

Functions, Function arguments, Function overloading, Classes and Objects

# From Structures to classes

```
struct POINT
{

        int x-cord;
        int y-cord;
};

int main()
{
        POINT pt1;

        pt1.x-cord = 3;
        pt1.y-cord = 7;
        return 0;
}
```

```
class POINT
{
 public:
                int x-cord;
                int y-cord;
};

int main()
{
                POINT pt1;

                pt1.x-cord = 3;
                pt1.y-cord = 7;
                return 0;
}
```

# Classes and Objects

Simple C++ Class

```
class Car{
public:
    int nModel;
    void DisplayModelNumber()
    {
        cout<<“Model Number:”<<nModel;
    }
}
```

# Declaring and using class object

```
class Car
{
....
}
int main(void)
{
    Car  myCar;
    myCar.nModel = 48952734;
    myCar.DisplayModelNumber();
}
```

# Access modifiers

- private
  - Only visible inside a class, not accessible directly. Information hiding
- public
  - Visible to the world, directly accessible.

Now how to access private member?

We use public functions to modify private members values. (Security)

e.g. GetModelNumber, SetModelNumber

# Information Hiding

- Hiding data provides security
- Hidden from whom?

- Data hiding mean state variables can not be accessible from other parts of program
- One class members are hidden from other.

*“Data hiding is designed to protect well intentioned programmers from honest mistakes”*

# Functions are public and data is private.

Data variables are declared under private access modifies

e.g.

```
private:
 int nAccountBalance;
```

Function that we want to expose to the world/other users are declared under the public access modifies.

e.g.

```
public:
void CreditToAccount(int accNo, double dAmount);
```

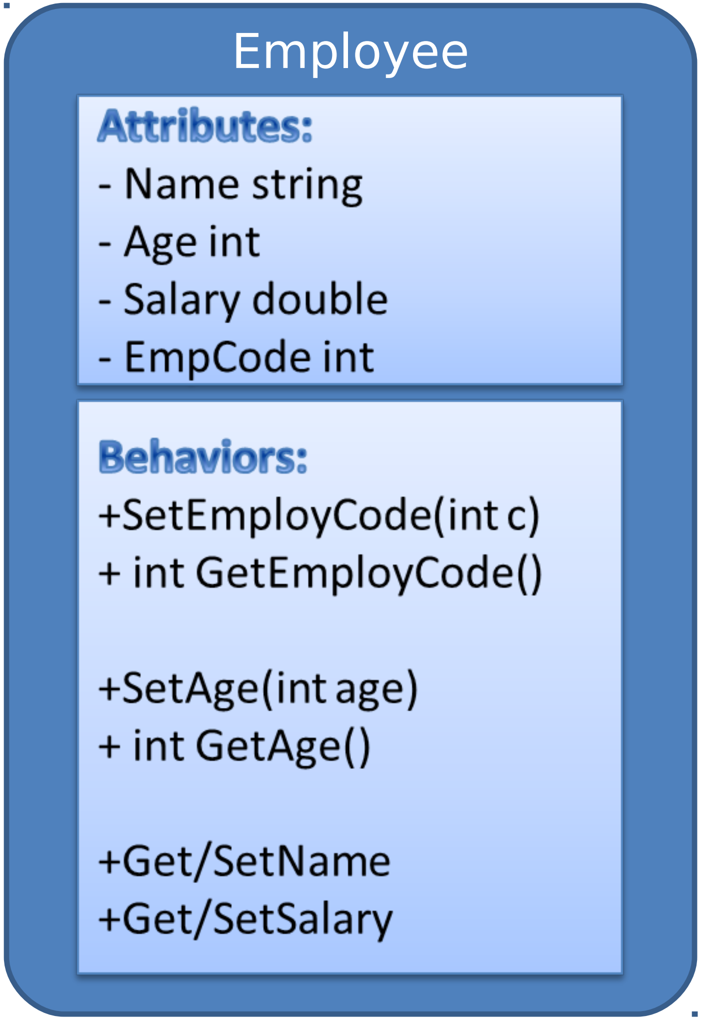
# Example 1

```
class Circle
{
private:
        int nRadius;
        POINT ptCenter;
public:
        void SetRadius(int rad);
        void SetCenter(POINT pt);
        void SetCenter(int x, int y);
        void GetArea();
};
```

# Example 1…

```
void Circle::SetRadius(int rad){
      if(rad > 0)
                  nRadius = rad;
      else
                  cout<<“Radius cannot be -ve ”;
}
void Circle:: SetCenter(POINT pt) { ptCenter = pt;}
void Circle:: SetCenter(int x, int y) {pt.x-cord = x; pt.y-cord = y;}
int Circle:: GetArea()
{
      return PI*nRadius*nRadius;
}
```

# Example 2



Employee

**Attributes:**

- Name string
- Age int
- Salary double
- EmpCode int

**Behaviors:**

+SetEmployCode(int c)
+ int GetEmployCode()

+SetAge(int age)
+ int GetAge()

+Get/SetName
+Get/SetSalary

# Constructor

Sometimes it is convenient that an object can initialize itself when it is first created.

A special member function of a class called constructor helps in automatic initialization of data members.

Constructor is executed automatically as the object is declared.

Abbreviation ctor

# Constructor. A Counter Example

```
class Counter
{
private:
int  nCount;
public:
 void Increment() {
   nCount++;
}
 int GetCount() { return count;
}
} // end class Counter
```

```
int void main()
{
Counter cntr;
cntr.increment();
std::cout<<cntr->GetCount();
}
```

# Constructor. A Counter Example

```
class Counter
{
private:
int  nCount;
public:
 void Increment() {
    nCount++;
}
 int GetCount() { return count;}
Counter()
{
nCount = 0;
}
} // end class Counter
```

1

```
int void main()
{
Counter cntr;
cntr.increment();
std::cout<<cntr->GetCount();
}
```

# Initializer  List

```
class Counter
{
private:
int  nCount;
int nMaxCount;
public:
 void Increment()
{
    nCount++;
}
Counter():nCount(0):nMaxCount(20)
{
}
} // end class Counter
```

# Overloaded Constructors

```
class  Counter{
….
Counter() { // Default Constructor
nCount = 0;
}
Counter(int count) { // Overloaded Constructor
nCount = count;
}
Counter(int count):nCount(count){ // with initializer list
}
…..
}
```

```
int void main()
{
Counter cntr(10);
cntr.increment();
std::cout<<cntr-
>GetCount();
}
```

# Overloaded Constructors

```
class  Counter{
….
Counter() { // Default Constructor
nCount = 0;
}
Counter(int count) { // Overloaded Constructor
nCount = count;
}
Counter(int count):nCount(count){ // with initializer list
}
…..
}
```

```
int void main()
{
Counter cntr(10);
cntr.increment();
std::cout<<cntr-
>GetCount();
}
```

# Destructor

As constructor is called automatically when the object is created similarly there is a function known as destructor called automatically when the object is destroyed or its lifetime ends.

```
Class Foo{
private:
int Data;
public:
Foo():Data(0)  // same name as class
{}
~Foo()              // same name with a tilde
{}
};
```

# Destructors…

- Like constructors Destructor also do not have return types
- They also take no arguments (No overloading :) ) ?
  - Only one way to destroy object.

- Most common use of destructor is to deallocate memory that may be allocated in constructor.

# Q & A