

Lecture 3

OBJECT ORIENTED PROGRAMMING A REVIEW

September 09, 2021
Thursday

PROGRAMMING PARADIGMS

How you **Write** Or **Organize** your code.

Procedural

List of instructions.
Top-to-Bottom Approach.
Functions with clear purpose.
Not suited for large projects.
Unrestricted access, unrelated data & functions.
Poor model of the real world

C, Pascal, FORTRAN.

Object Oriented

Revolves around the concept of an Object (data & function combined and encapsulated in a single logical entity).
Provides better data security and avoids redundancy.
Well suited for large projects.
Simulates the real world better.

C++, Java.

Functional

Revolves around the concept of a pure function with clearly defined tasks. Immutable Data is passed around as parameters.
Provides better readability.
Modular design increases productivity.

Haskell, F#

Object Oriented Programming

Everything revolves around the Object

Thinking in terms of Objects, rather than Functions

OBJECT

*A real life entity having
a **state** and **behavior***

- Why there are so many objects having similar state & behavior?
- Each object was built from the same set of blueprints and therefore contains the same component.

Properties or State:

1. Color
2. Model
3. Make
4. Fuel type
5. No of seats



Behaviour or Actions:

1. Start
2. Accelerate
3. Shift
4. Reverse
5. Apply brake
6. Stop

CLASS

*The **blueprint** from which individual objects are created*

- Provides data specification and functions to manipulate the data.
 - Functions of the class are referred as methods, member functions or function members. **Methods define the behavior of the objects.**
 - Variables of the class are referred to as data members. **Data members define the state of the object.**
- Object is an instance of a class.

CLASS | C++ EXAMPLE

```
#include <iostream>
using namespace std;
```

```
class Node {
```

```
    // data members defined
```

```
    unsigned int age;
    string name;
```

}

Data Members

```
    int returnAge ()
        { return age; }
```

}

Member Functions

```
    void displayName ()
        { cout<<name; }
```

```
};
```

OOP | PRINCIPLES

What

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

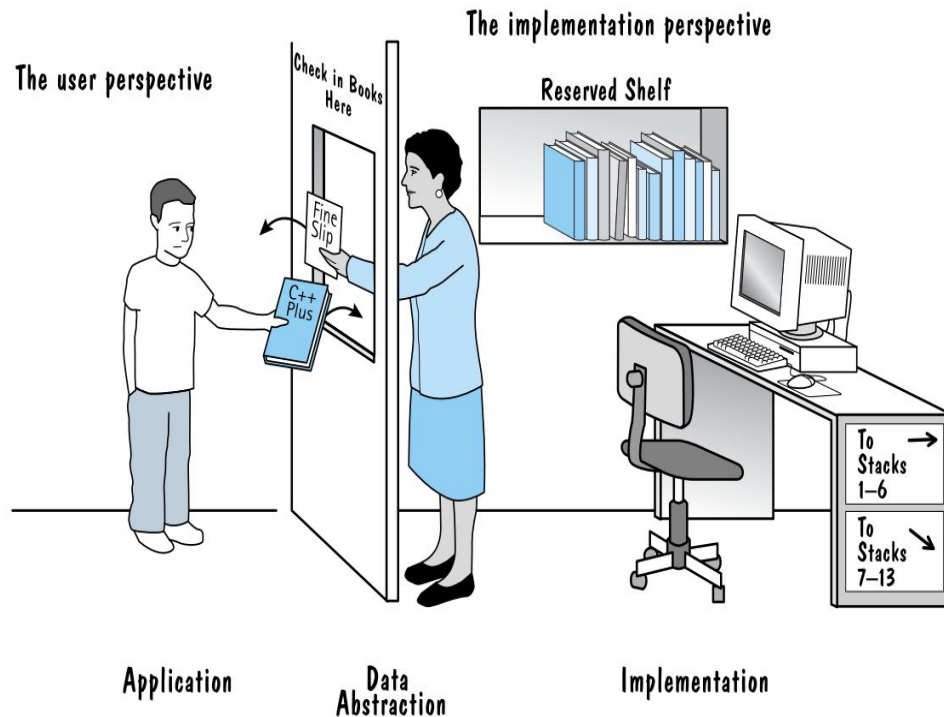
Why

- Modularity
- Reusable Code
- Pluggability & Debugging
- Information Hiding

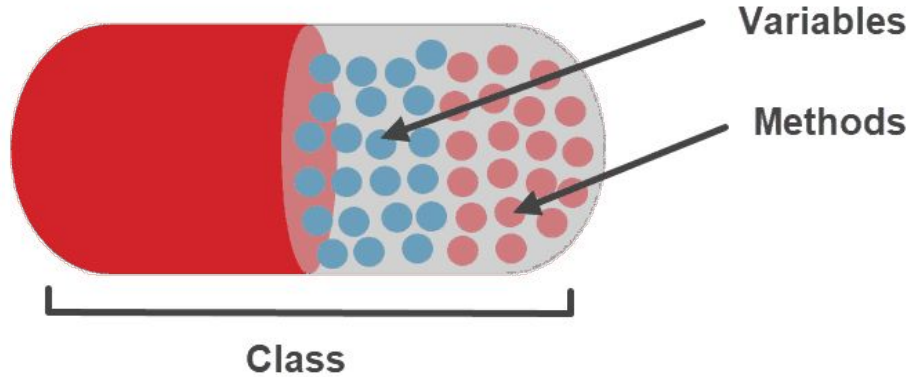
ABSTRACTION

Design that allows user to utilize complex functionality without showing how it works.

Only relevant information for the user.



ENCAPSULATION



Grouping related information and relevant functions into one unit and defining where that information is accessible.

ENCAPSULATION

- Classes
 - Private
 - Only accessible within the class.
 - Protected
 - Only accessible within the class and in inherited classes.
 - Public
 - Accessible outside the class.
- Header files
 - `math.h` provides
 - `pow(x,y)`, `sqrt(x)`, `exp(x)`, `log(x)`, `sin(x)`, `cos(x)`, `tan(x)` etc.

CLASS | C++ EXAMPLE

```
#include <iostream>
```

```
class Node {
```

```
    private:
```

```
        unsigned int age;
```

```
        string name;
```

```
    protected:
```

```
        int returnAge ()
```

```
        { return age; }
```

```
    public:
```

```
        void displayName ()
```

```
        { cout<<name; }
```

```
};
```

FRIEND CLASS & FUNCTION

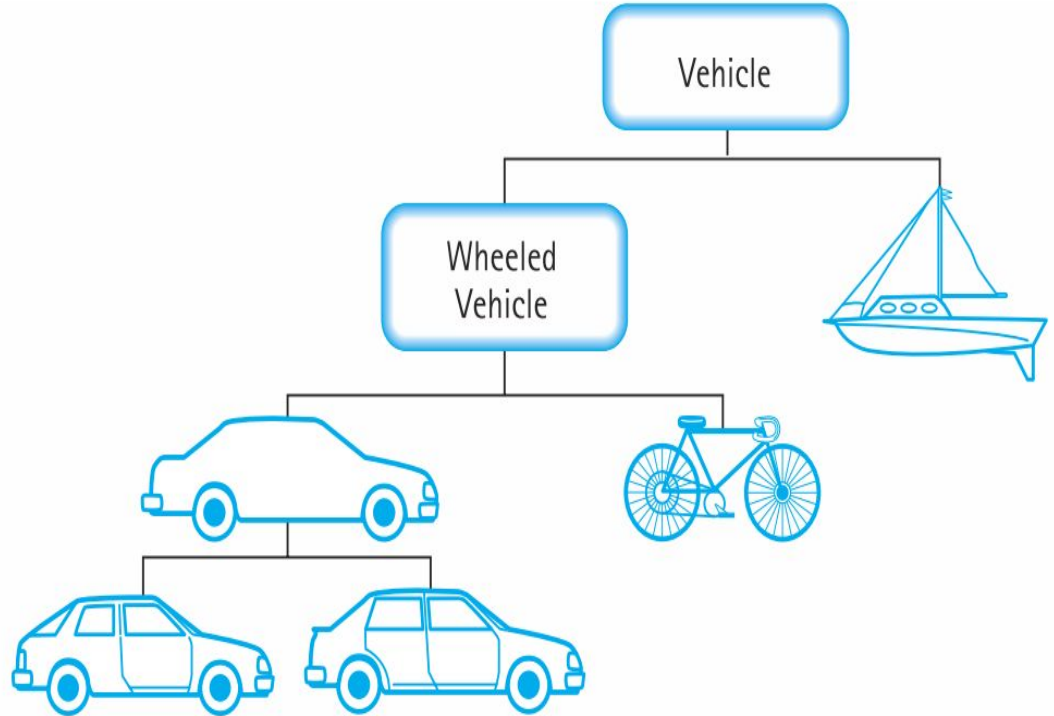
- A friend class can access private and protected members of other class in which it is declared as friend.
- A friend function can be given special grant to access private and protected members. A friend function can be
 - A method of another class
 - A global function
- A class has to tell who is its Friend.
 - `friend int sum(A, B);`

CLASS Vs STRUCT

- First difference between class & struct is the encapsulation defaults.
 - A struct defaults to public members.
 - A class defaults to private members.
- Structs are allocated on Stack, while classes are allocated on heap.

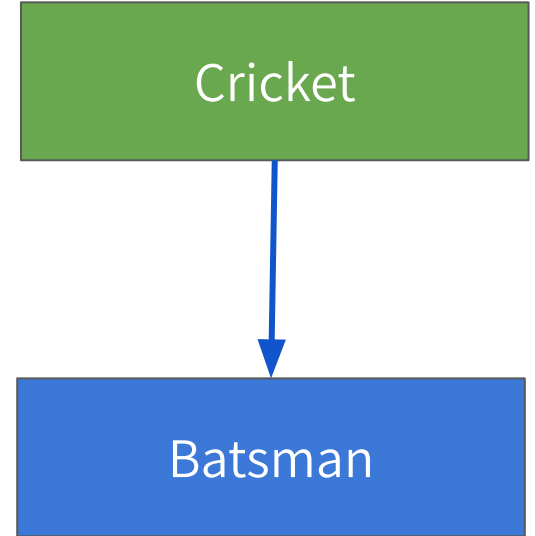
INHERITANCE

- Hierarchy of classes is constructed such that each descendent class inherits the properties of its ancestor class.
- The class being inherited is **Base Class**.
- The class inheriting the properties is **Derived Class**.



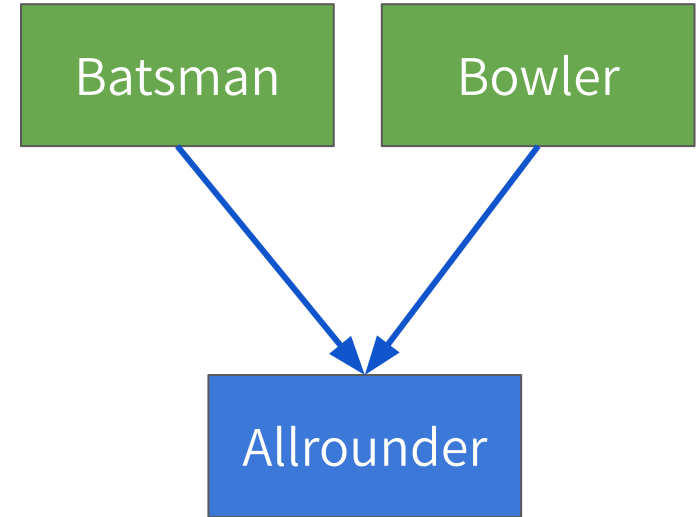
INHERITANCE | SINGLE

```
class Cricket {  
    ... ..  
};  
class Batsman: public Cricket {  
    ... ..  
};
```



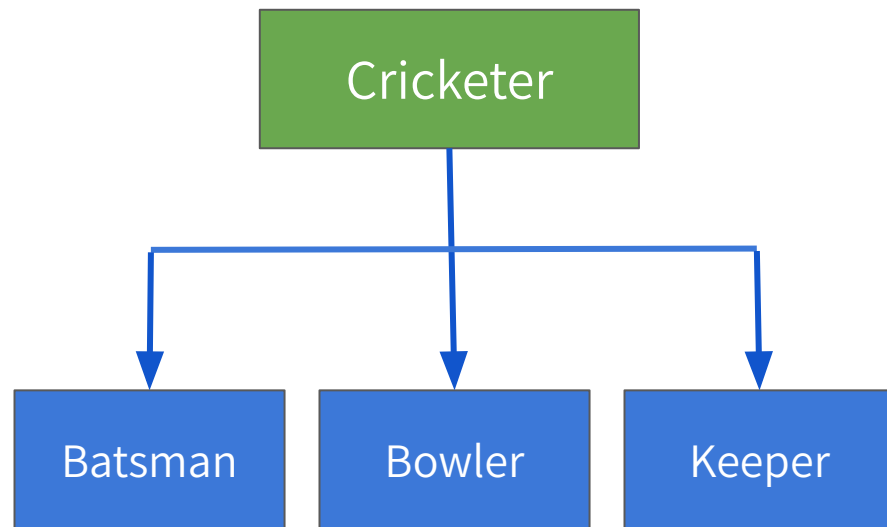
INHERITANCE | MULTIPLE

```
class Batsman {  
    ... ..  
};  
  
class Bowler {  
    ... ..  
};  
  
class Allrounder : public Batsman, public Bowler {  
    ... ..  
};
```



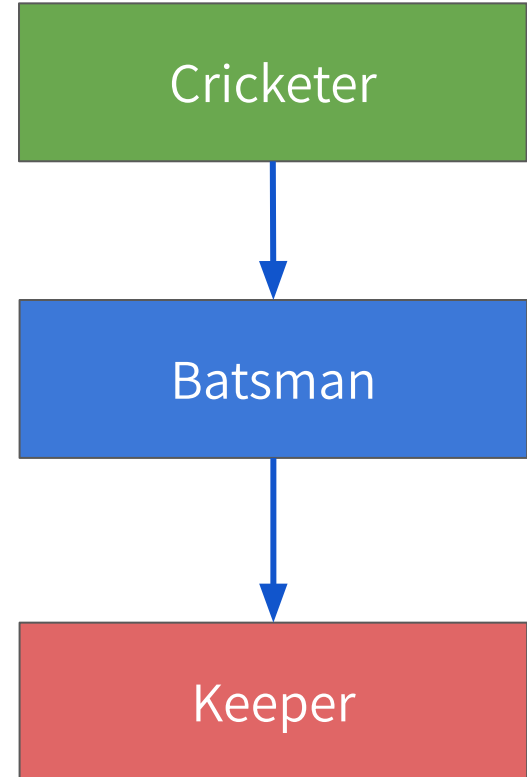
INHERITANCE | HIERARCHICAL

```
class Cricketer {  
    ... ..  
};  
  
class Batsman : public Cricketer {  
    ... ..  
};  
  
class Bowler : public Cricketer {  
    ... ..  
};  
  
class Keeper : public Cricketer {  
    ... ..  
};
```



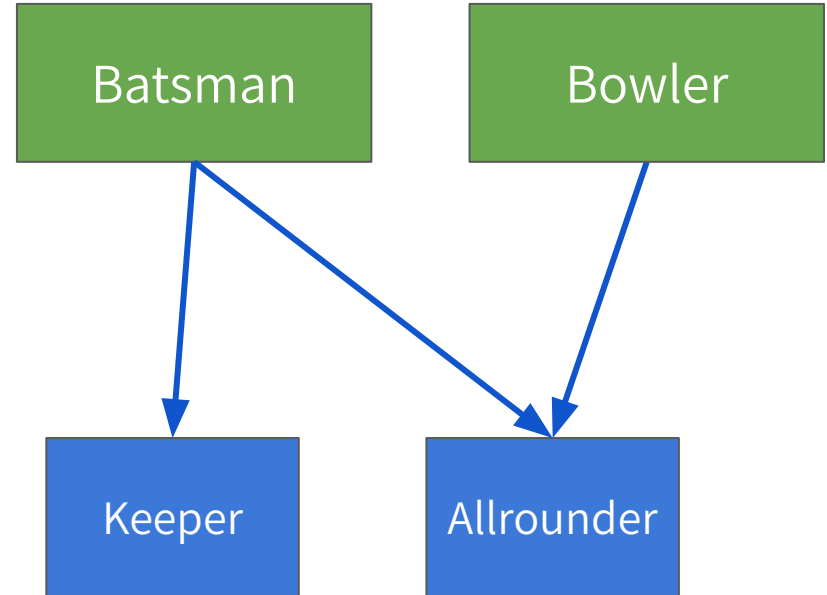
INHERITANCE | MULTILEVEL

```
class Cricketer {  
    ... ..  
};  
  
class Batsman : public Cricketer {  
    ... ..  
};  
  
class Keeper : public Batsman {  
    ... ..  
};
```



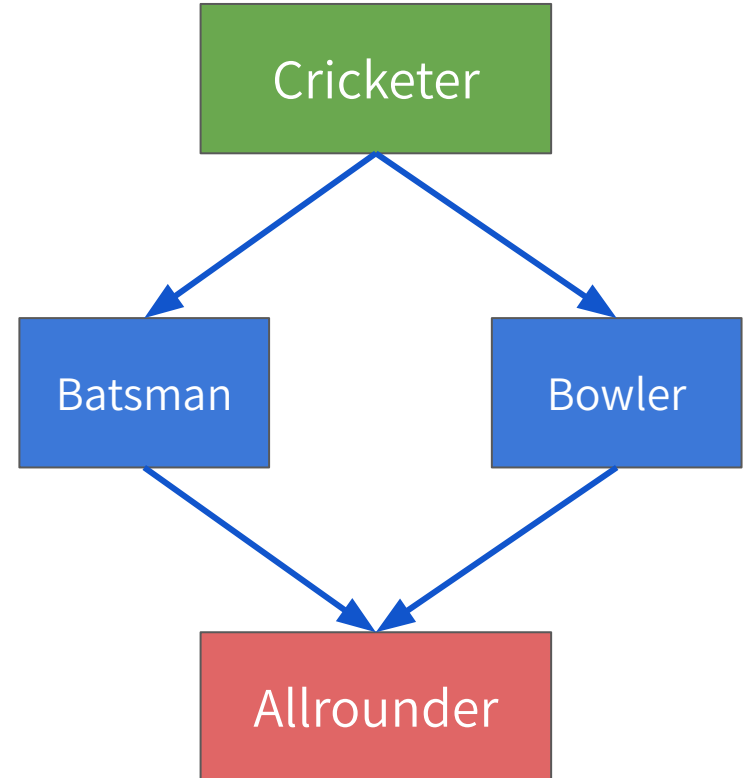
INHERITANCE | HYBRID

```
class Batsman {  
    ... ..  
};  
  
class Bowler {  
    ... ..  
};  
  
class Keeper : public Batsman {  
    ... ..  
};  
  
class Allrounder : Batsman, Bowler {  
    ... ..  
};
```



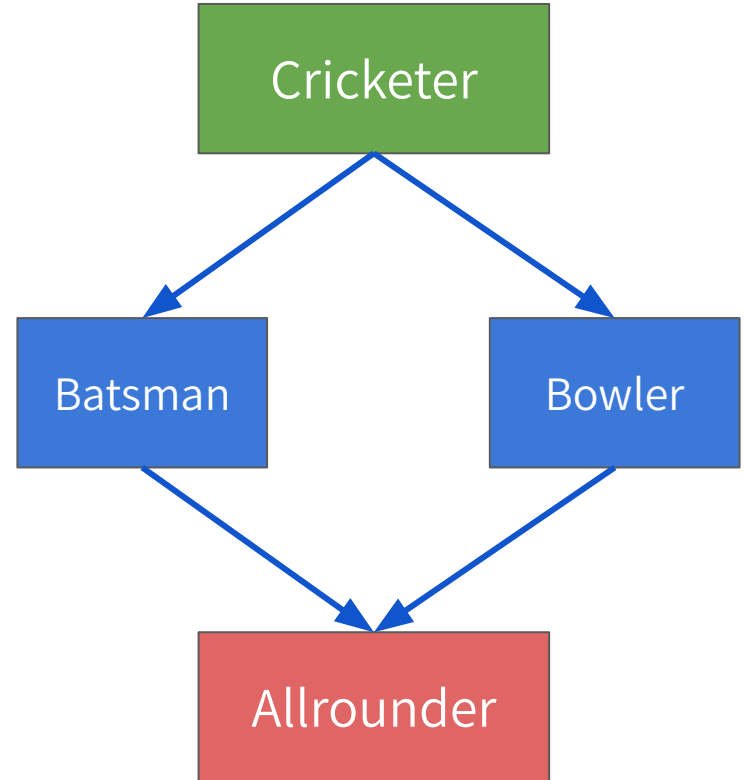
INHERITANCE | MULTIPATH

```
class Cricketer {  
    ... ..  
};  
  
class Batsman : public Cricketer {  
    ... ..  
};  
  
class Bowler : public Cricketer {  
    ... ..  
};  
  
class Allrounder : Batsman, Bowler {  
    ... ..  
};
```



INHERITANCE | MULTIPATH

```
class Cricketer {  
    ... ..  
};  
  
class Batsman : public Cricketer {  
    ... ..  
};  
  
class Bowler : public Cricketer {  
    ... ..  
};  
  
class Allrounder : Batsman, Bowler {  
    ... ..  
};
```



POLYMORPHISM

- Greek letters *Poly* means **many**, *Morphs* means **forms**. The ability of acquiring many forms.
- The ability to determine which of several functions with the same name is appropriate, combination of static & dynamic binding.



STATIC Vs DYNAMIC BINDING

Static Binding

- Determination of appropriate method at compile time.
- Known as Overloading & early-binding.
- Giving the same name to more than one method, or operators for more than one operations.
- Parameters must differ either by type Or by number.
- Methods are within the same scope. Doesn't require inheritance.
- Less flexible, but faster execution.

Dynamic Binding

- Determination of appropriate method at run time.
- Known as Overriding & Late-binding
- Giving the same method a different implementation in derived class than the base class function, with Virtual Functions
- Parameters must be same by type and number.
- Methods are in different scopes. Occurs with inheritance.
- More flexible, but slower execution.

CONSTRUCTOR

- Constructor is a member function with some creative powers
 - Allocates memory to the object when created.
 - Automatic initialization of data members on creation.
 - Same name as Class
 - Tells compiler this is a constructor.
 - No return type.
 - Another way of telling compiler this is a constructor.
 - The keywords like virtual, volatile, static, const can not be used.

CONSTRUCTOR | EXAMPLE

```
class Node {  
    // data members defined  
    private:  
        unsigned int age;  
    public:  
        Node () : age (0)  
        { /* empty body */ }  
  
        Node () {  
            age = 0;  
        }  
        void getAge ()  
        { return age; }  
};
```

Helpful in
initializing const
members.

If multiple members must be
initialized they are separated by
comma in **Initializer List**.

Can also initialize members inside
the constructor body.

CONSTRUCTOR | TYPES

1. Empty Constructor

- a. Does not have parameters and body.
- b. Gets called if no constructor is specified by the programmer.
- c. Useful:
 - i. No instantiation.
 - ii. Constructor overloading.

```
class Node {  
    public:  
        Node () {}  
};
```

2. Default Constructor

- a. Does not receive parameters but have a body.
 - i. Initializing the data members with a fixed value for all objects.

```
class Node {  
    private:  
        unsigned int age;  
    public:  
        Node ()  
            { age = 25;    }  
};
```

CONSTRUCTOR | TYPES

1. Parameterized Constructor

- a. When we want to initialize data members with user given values.

```
class Node {  
    private:  
        unsigned int age;  
    public:  
        Node (int x ) : age (x) { }  
};
```

2. Copy Constructor

- a. Special case, when we want to initialize the data members with an object of the same class.

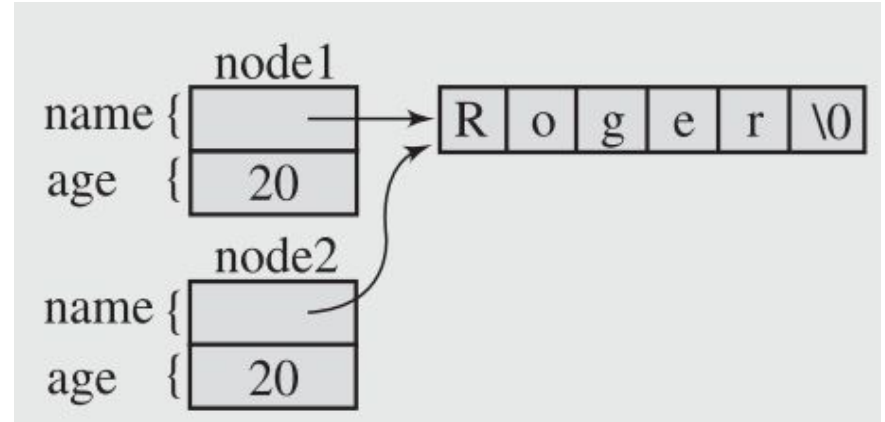
```
class Node {  
    private:  
        unsigned int age;  
    public:  
        Counter (Counter& obj ) {  
            age = obj.age;  
        }  
};
```

CONSTRUCTOR | TYPES

- Shallow Copy Constructor
 - By default provided by compiler if no copy constructor is provided.
 - Performs member-by-member copying, also known as bitwise copying.
 - Well suited, when objects are not using dynamic memory.
 - In case of dynamic memory allocation for objects. The shallow copy can lead to two objects updating the same memory location.
 - i. Memory allocated once would be freed twice, causing a system exception.

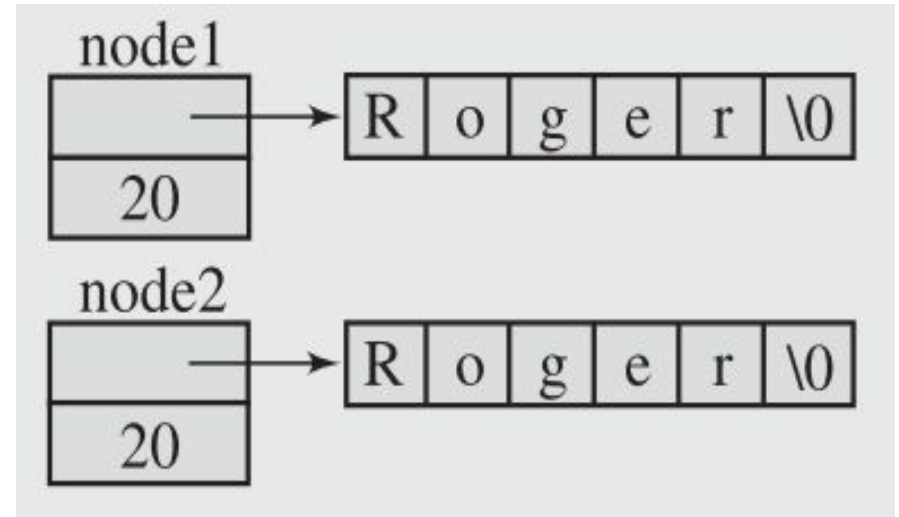
CONSTRUCTOR | SHALLOW COPY CONSTRUCTOR

```
class Node {  
    char* name;  
    int age;  
  
    Node ( char* n = "", int a ) {  
        name = strdup(n);  
        age = a;  
    }  
};
```



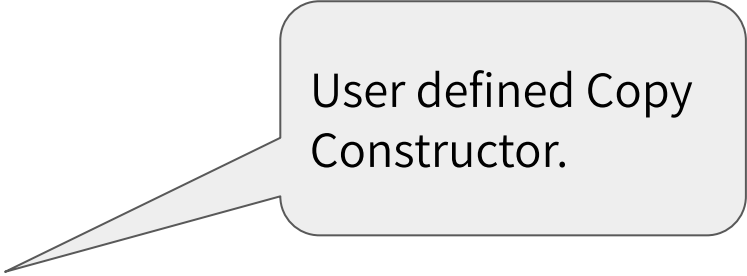
CONSTRUCTOR | DEEP COPY CONSTRUCTOR

- Deep Copy Constructor
 - Allocates similar memory resources with the same value to the new object.
 - Must be defined by the user.
 - Well suited for dynamic memory allocation.
 - The assignment to data members in one object does not affect the other.



CONSTRUCTOR | DEEP COPY CONSTRUCTOR

```
class Node {  
    char* name;  
    int age;  
  
    Node ( char* n = "", int a ) {  
        name = strdup (n);  
        age = a;  
    }  
  
    Node (const Node& n){  
        name = strdup (n.name);  
        age = n.age;  
    }  
};
```



User defined Copy
Constructor.

DESTRUCTOR

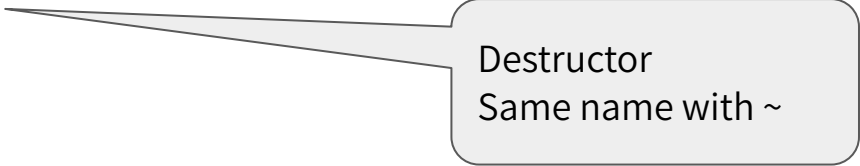
- Also a member function with some destructive powers
 - Gets automatically called when object is destroyed.
 - Deallocates or frees the memory allocated by the constructor
 - A class can have only one destructor, must be public.
 - Has same name as Class followed by a tilde ~
 - Takes no arguments.
 - Does not have a return type.

DESTRUCTOR | EXAMPLE

```
class Node {  
    char* name;  
    int age;  
  
    Node ( char* n = "", int a ) {  
        name = strdup (n);  
        age = a;  
    }  
  
    ~Node ( ) {  
        if (name != 0)  
            free name;  
    }  
};
```

In case of dynamically allocated memory.
The memory allocated by pointer data member is released but the memory taken by the value is not.

After object is destroyed the value is inaccessible.
Must define destructor explicitly to release



Destructor
Same name with ~