Object Oriented Programming

P.S. These slides are useless if you do not attend classes

Note: All the materials taken from external websites are linked in the reference section

Getter / Setter Functions

Getter functions (or accessor functions) are used to read value of a private member of some class.

Setter functions (or mutator functions) are used to modify the value of a private member of some class.

Providing public set and get functions allows clients of a class to access the hidden data, but only indirectly

Getter / Setter Functions

Getter functions (or accessor functions) are used to read value of a private member of some class.

```
class BankAccount

{

int PIN; //private variable

int get_PIN()

{

return PIN;

}

};
```

Getter / Setter Functions

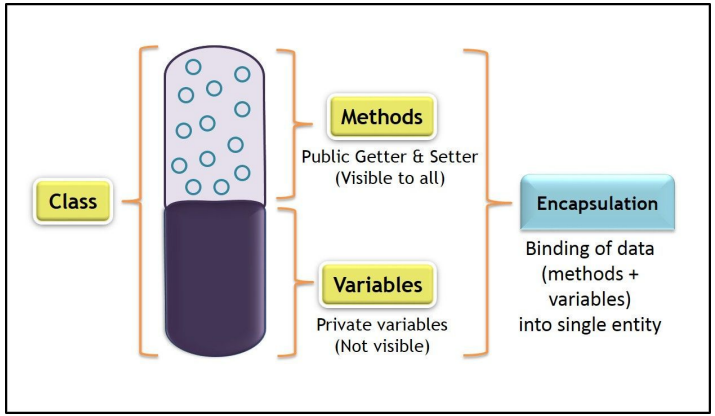
Setter functions (or mutator functions) are used to modify the value of a private member of some class.

```
class BankAccount

{

int accountNo; //private variable

void set_accountNo(int num)

{

accountNo = num;

}

}
```

Me: *use "public" access modifier to get & set Attributes*

Setter & Getter Methods:

Encapsulation



In C++, encapsulation is used along with the classes and access specifier concept.

It is the process of combining data and function into a single unit.

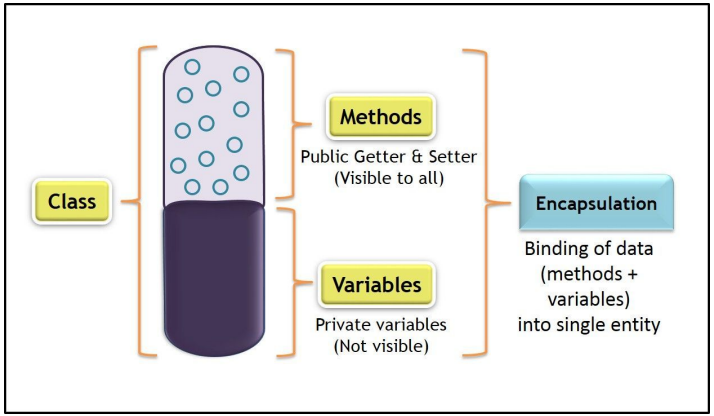
Using the method of encapsulation the programmer cannot access the class directly.

You can assume it as a protective wrapper that stops random access of code defined outside that wrapper.

Encapsulation

It binds the data & functions together which keeps both safe from outside interference.

Data encapsulation led to data hiding.

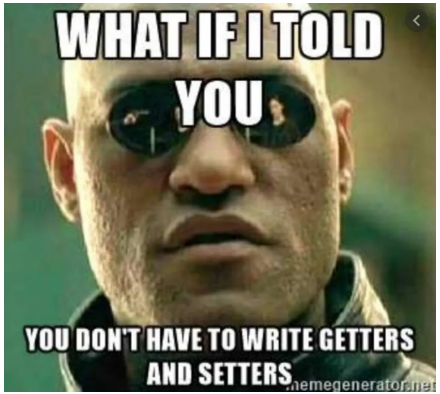


Encapsulation

```
#include<iostream>
using namespace std;
class Example
{
private:  // data hidden from outside world
int num;
public:
void set(int a) // function to set value of variable num

{
num =a;
}
int get() // function to return value of variable num
{
return num;
}
};
```

```
// main function

int main()
{
Example obj;
obj.set(5);
cout<<obj.get();
return 0;
}
```

Getter / Setter

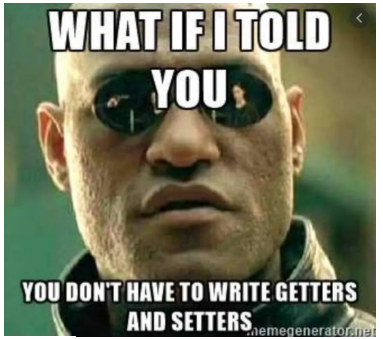


177

Having getters and setters does not in itself break encapsulation. What does break encapsulation is automatically adding a getter and a setter for every data member (every *field*, in java lingo), without giving it any thought. While this is better than making all data members public, it is only a small step away.

The point of encapsulation is not that you should not be able to know or to change the object's state from outside the object, but that you should have a reasonable *policy* for doing it.

Getter / Setter



- Some data members may be entirely internal to the object, and should have neither getters nor setters.
- Some data members should be read-only, so they may need getters but not setters.
- Some data members may need to be kept consistent with each other. In such a case you would not provide a setter for each one, but a single method for setting them at the same time, so that you can check the values for consistency.
- Some data members may only need to be changed in a certain way, such as incremented or decremented by a fixed amount. In this case, you would provide an `increment()` and/or `decrement()` method, rather than a setter.
- Yet others may actually need to be read-write, and would have both a getter and a setter.

Data validation / Validity Checking

Data validation is performed to ensure that class members are provided data in “correct format”

Validation rules vary according to requirements.

Where to perform validation?

Data validation / Validity Checking

```
void setCourseName(string name) // a setter function
{
if ( name.length() <= 25 )

      courseName = name; //data member updated

if ( name.length() > 25 )
{
  cout << "Name exceeds max length (25)" << endl; // error message
}

}
```

Class Activity

A bank wants a simple application module to manage the accounts of its customers.

For every new customer, the app must let us fill in the details including his Name, Age, NIC#, Address, Opening Balance, Current Balance, Contact# & PIN.

These details may be modified later except for the PIN. At any given time, the customer can check his balance.

Also, tax must be calculated (Tax is 0.15% of the current balance for customers aged 60 or above and 0.25% for all other customers).

Class Activity

Create an Account class to represent customers' bank accounts. Include a data member of type int to represent the account balance.

You need to initialize the data members. But while doing so, you should validate the initial balance to ensure that it's greater than or equal to 0. If not, set the balance to 0 and display an error message indicating that the initial balance was invalid.

Provide three member functions.

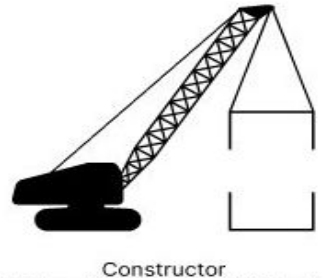
`Member function credit()` should add an amount to the current balance.

`Member function debit()` should withdraw money from the Account and ensure that the debit amount does not exceed the Account's balance. If it does, the balance should be left unchanged and the function should print a message indicating "Debit amount exceeded account balance.

`Member function getBalance()` should return the current balance.

Create a program that creates two Account objects and tests the member functions of class Account.

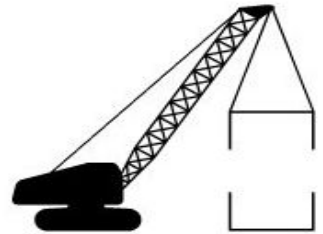
Constructors



A constructor is called whenever an object of a class is created.

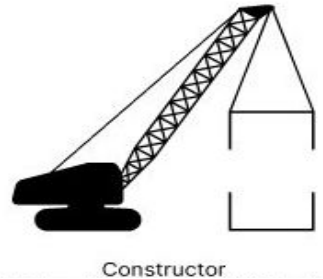
A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create.

Constructors vs Member Functions



- Constructor has same name as the class itself.
- Constructors don’t have return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

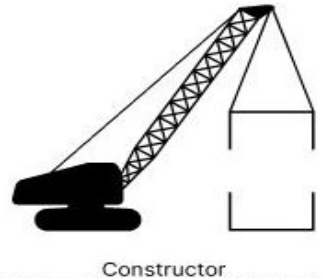
Constructors



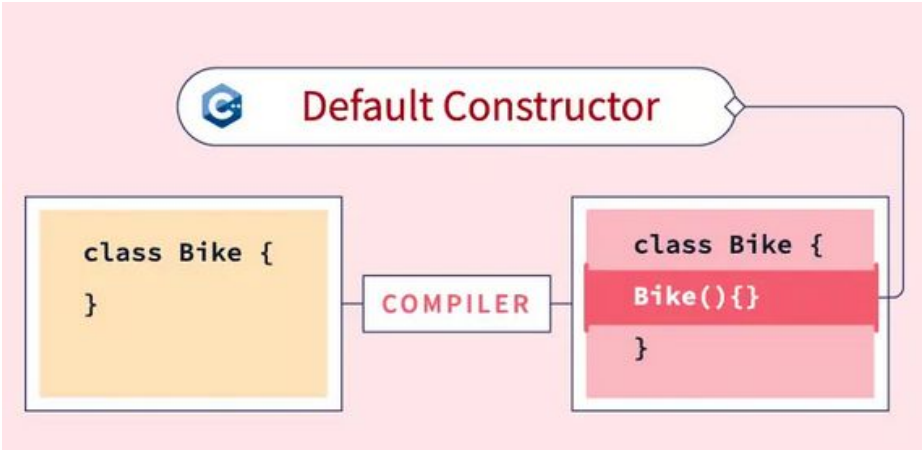
There are 3 types of Constructors

- Default
- Parametrized
- Copy

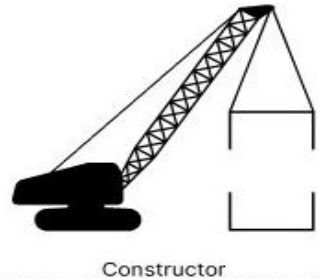
Default Constructors



- Default constructor is the constructor which doesn't take any argument. It has no parameters.
- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.



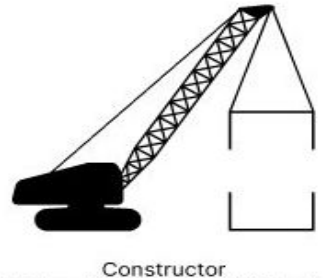
Default Constructors




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

class Line {

public:
Line() {  //Constructor Definition
cout << "Object is being created" << endl;
}

void setLength( double len ) { //Setter
length = len;
}

double getLength( void ) { //Getter
return length;
}

private:
double length;
};
```

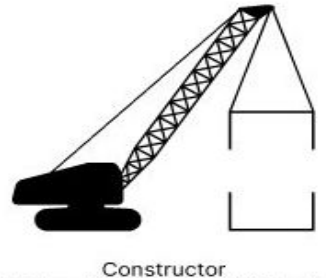
```
int main() {
Line line; // Constructor Call
line.setLength(6.0);
cout << "Length of line : " <<
line.getLength() <<endl;
return 0;
}
```

Default Constructors




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

class Line {

public:
void setLength( double len );
double getLength( );
Line(); // This is the constructor

private:
double length;
};
```

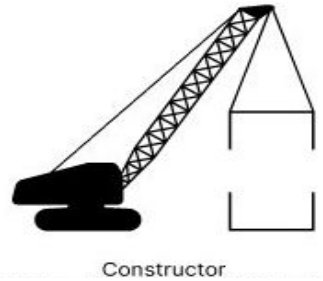
Member Function Definition Outside Class




```
Line::Line() {  //Constructor Definition
cout << "Object is being created" << endl;
}

void Line::setLength( double len ) {
length = len;
}

double Line::getLength( ) {
return length;
}
```

```
int main() {
Line line; // Constructor Call
line.setLength(6.0);
cout << "Length of line : " <<
line.getLength() <<endl;
return 0;
}
```

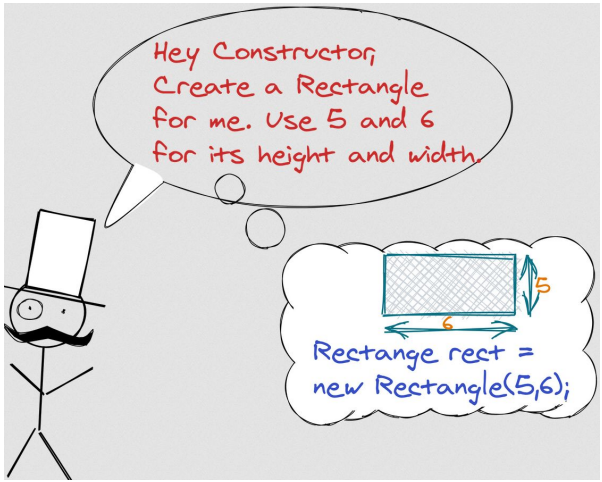
Parameterized Constructors



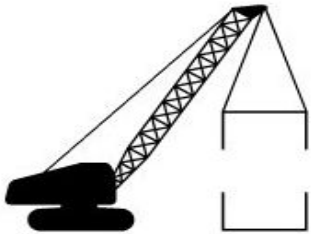
A constructor can take parameters. These parameters are used to initialize class variables for the object.

```
class Employee
{
    string name;

    public:
    Employee(string eName)
    {
        name = eName;
    }
}
```

```
int main()
{
    Employee e1("Ali");
    Employee e2("Shuja");
}
```



Parameterized Constructors

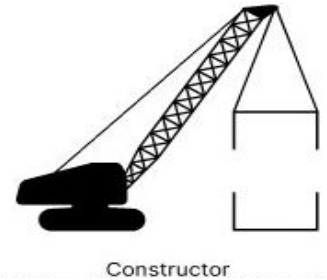


Can we have more than one parameterized constructors of a class?

Yes, we can. But not having the same parameter signature.

If you define any parameterized constructor(s) in the class, C++ will not implicitly create a default constructor for you.

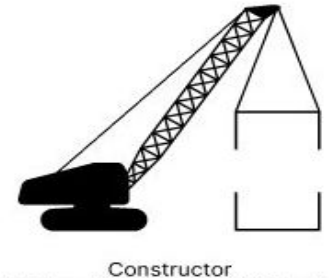
Parameterized Constructors




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

class Line {

public:
Line() {  //Default Constructor
 cout << "Object is being created" << endl;
}

Line(int x) {  // parameterized Constructor
length=x;
}
void setLength( double len ) { //Setter
length = len;
}
double getLength( void ) { //Getter
return length;
}
private:
double length;
};
```

```
int main() {
Line line1; // Default Constructor Call
line1.setLength(6.0);
Line line2(10.0) // Parametrized Constructor Call
cout << "Length of line1: " << line1.getLength()<<endl;
cout << "Length of line2: " << line2.getLength()<<endl;
return 0;
}
```

Parameterized Constructors




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

class Rectangle {

public:
Rectangle() {  //Default Constructor
 cout << "Object is being created" << endl;
}

Rectangle(int x) {  // parameterized Constructor
length=x;
}
Rectangle(double y) {  // parameterized Constructor
width=y;
}
Rectangle(int x, double y) {  // parameterized Constructor
length=x;
width=y;
}
private:
int length;
double width;
};
```

```
int main() {
Rectangle rect1;
Rectangle rect2(5);
Rectangle rect3(12.5);
Rectangle rect4(3,4.5);
return 0;
}
```

Parameterized Constructors

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

class Rectangle {

public:

Rectangle(double y) {
 width=y;
}
Rectangle(int x=4, double y=2) {  // Default Parameters
length=x;
width=y;
cout<<length<<"\t"<<width<<endl;
}
private:
int length;
double width;
};
```

```
int main() {
Rectangle rect1;
Rectangle rect2(5);
Rectangle rect3(12.5);
Rectangle rect4(3,4.5);
return 0;
}
```

Parameterized Constructors

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

class Rectangle {

public:

Rectangle(double y) {
 width=y;
}
Rectangle(int x=4, double y=2) {  // Default Parameters
length=x;
width=y;
cout<<length<<"\t"<<width<<endl;
}
private:
int length;
double width;
};
```

```
4    2
5    2
3    4.5
```

```
int main() {
Rectangle rect1;
Rectangle rect2(5);
Rectangle rect3(12.5);
Rectangle rect4(3,4.5);
return 0;
}
```

Initializer List

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.

```
 Point(int i = 0, int j = 0):x(i), y(j) {}

    /*  The above use of Initializer list is optional as the
        constructor can also be written as:
        Point(int i = 0, int j = 0) {
            x = i;
            y = j;
        }
    */
```

Initializer List

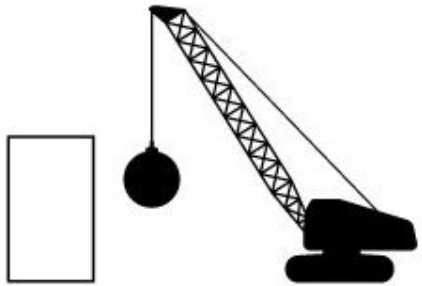
```
#include <iostream>
using namespace std;

class Rectangle {

public:
Rectangle(int x=4, double y=2) : length(x), width(y)
{  }
void show()
{
     cout<<"Length:"<<length<<" Width:"<<width<<endl;
}
private:
int length;
double width;
};
```

Output

```
/tmp/E7NFiNtTIY.o
Length:4 Width:2
Length:5 Width:2
Length:12 Width:2
Length:3 Width:4.5
```

```
int main() {
Rectangle rect1;
Rectangle rect2(5);
Rectangle rect3(12.5);
Rectangle rect4(3,4.5);
rect1.show();
rect2.show();
rect3.show();
rect4.show();
return 0;
}
```

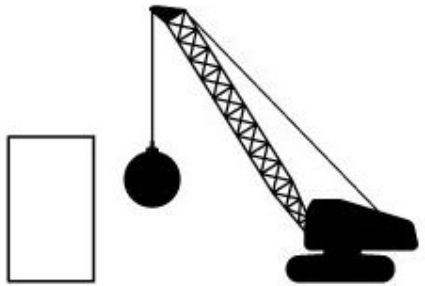
Destructor



A class' destructor is automatically called when an object of that class is "destroyed".

Destruction of an object means when program execution leaves the scope in which object was instantiated

Destructor



Destructor

A destructor cannot return a value and cannot take any arguments.

A destructor cannot be overloaded.

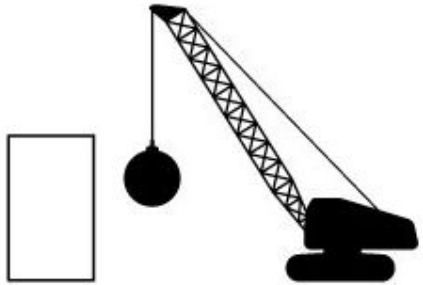
A class can thus have only one destructor.

If you do not explicitly define a destructor, the compiler provides a default “empty” destructor.

```
class Example
{
    //Constructor
    public Example(int x, int y)
    {
        //Initialize the Members
    }

    //Destructor
    ~Example()
    {
        //Cleanup Statements
    }
}
```

Destructor

```
class MyClass
{
      int objectID;

      MyClass(int objectID1)
      {
      objectID = objectID1;
      }

     ~MyClass()
      {
      cout << objectID << " deleted";
      }
}
```



Destructor

Destructor

```
#include <iostream>
using namespace std;
class ABC
{
public:
ABC () //constructor defined
{
cout << "Hey look I am in constructor" << endl;
}

~ABC() //destructor defined
{
cout << "Hey look I am in destructor" << endl;
}

};
```

```
int main()
{
ABC cc1; //constructor is called
ABC cc2; //constructor is called

cout << "function main is terminating...."
<<endl;
/*....object cc1 goes out of scope,now
destructor is being called...*/
return 0;
} //end of program
```

Destructor

```
#include <iostream>
using namespace std;
class ABC
{
public:
ABC () //constructor defined
{
cout << "Hey look I am in constructor" << endl;
}

~ABC() //destructor defined
{
cout << "Hey look I am in destructor" << endl;
}

};
```

```
int main()
{
ABC cc1; //constructor is called
ABC cc2; //constructor is called

cout << "function main is terminating...."
<<endl;
/*....object cc1 goes out of scope,now
destructor is being called...*/
return 0;
} //end of program
```

```
Hey look I am in constructor
Hey look I am in constructor
function main is terminating....
Hey look I am in destructor
Hey look I am in destructor

--------------------------------
Process exited after 0.127 seconds with return value 0
Press any key to continue . . .
```

Constructor Order

```
#include <iostream>
using namespace std;
class MyClass
{

int id;

public:

MyClass (int x) //constructor defined
{
id=x;
cout << "Object "<<id<<"created" << endl;
}

};
```

```
MyClass ob1(1);
void func()
{
MyClass ob3 (3);
MyClass ob4 (4);
}

int main()
{
MyClass ob2 (2);
func();
MyClass ob5 (5);
}
```

Constructor Order

```
#include <iostream>
using namespace std;
class MyClass
{

int id;

public:

MyClass (int x) //constructor defined
{
id=x;
cout << "Object "<<id<<"created" << endl;
}

};
```

```
MyClass ob1(1);
void func()
{
MyClass ob3 (3);
MyClass ob4 (4);
}

int main()
{
MyClass ob2 (2);
func();
MyClass ob5 (5);
}
```

Output

```
/tmp/E7NFiNtTIY.o
Object 1created
Object 2created
Object 3created
Object 4created
Object 5created
```

Destructor Order

```
#include <iostream>
using namespace std;
class MyClass
{
int id;
public:
MyClass (int x) //constructor defined
{
id=x;
cout << "Object "<<id<<"created" << endl;
}
~MyClass() //destructor defined
{
cout << "Object "<<id<<"deleted" << endl;
}

};
```

```
MyClass ob1(1);
void func()
{
MyClass ob3 (3);
MyClass ob4 (4);
}

int main()
{
MyClass ob2 (2);
func();
MyClass ob5 (5);
}
```

Destructor Order

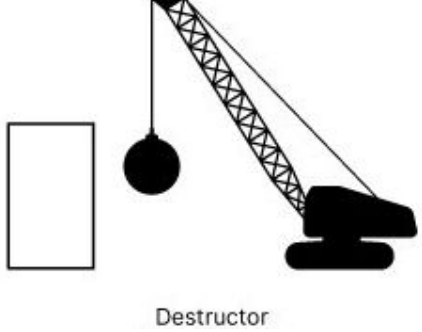
```
#include <iostream>
using namespace std;
class MyClass
{
int id;
public:
MyClass (int x) //constructor defined
{
id=x;
cout << "Object "<<id<<"created" << endl;
}
~MyClass() //destructor defined
{
cout << "Object "<<id<<"deleted" << endl;
}

};
```

```
MyClass ob1(1);
void func()
{
MyClass ob3 (3);
MyClass ob4 (4);
}

int main()
{
MyClass ob2 (2);
func();
MyClass ob5 (5);
}
```

Output

```
/tmp/E7NFiNtTIY.o
Object 1created
Object 2created
Object 3created
Object 4created
Object 4deleted
Object 3deleted
Object 5created
Object 5deleted
Object 2deleted
Object 1deleted
```

Why Destructors are Useful?



Useful for garbage collection.

If we do not write our own destructor in class, compiler creates a default destructor for us.

It works fine unless we have dynamically allocated memory or pointer in class.

When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed.

This must be done to avoid memory leak.

User Defined Destructor

Only variables allocated on the stack are deallocated automatically when they go out of scope.

```
1 int a = 5; // allocated on the stack
2 int* b = new int(5); // allocated dynamically, on the heap. Must be deleted
```

User Defined Destructor

```
#include <iostream>
using namespace std;
class ABC
{
int *p;
public:
ABC () //constructor defined
{ p=new int;
}
~ABC() //destructor defined
{  delete p;
}
};
```

Class Activity

Write a program to print the names of students by creating a Student class. If no name is passed while creating an object of the Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating the object of the Student class.

Class Activity

Create a class called water bottle.

The water bottle has a company (made by), color and water capacity. The water capacity is stored in both liters(l) and milliliters(ml).

Create variables and methods for your class. Methods should include getters and setters.

Also create an additional method that updates the water capacity (both in l and ml) after asking user how much water a person has drank. Assume that the user always enters the amount in ml.

Demonstrate the functionality of the water bottle in your main method.

Copy Constructor

Creating a copy of an object means to create an exact replica of the object having the same literal value, data type, and resources.

A copy constructor is used to initialize an object using another object of the same class.

```
ClassName (const ClassName &ob);
```

```
// Copy Constructor

Test Obj1(Obj);

Or

Test Obj1 = Obj;
```

```
// Default assignment operator

Test Obj2;

Obj2 = Obj1;
```

Copy Constructor

If we don’t define our own copy constructor, the compiler creates a default copy constructor for each class.

The default copy constructor performs member-wise copy between objects.

Default copy constructor works fine unless an object has pointers or any runtime allocation.

Copy Constructor

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

class Point {
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor
    Point(const Point& p1)
    {
        x = p1.x;
        y = p1.y;
    }

    int getX() { return x; }
    int getY() { return y; }
};
```

```
int main()
{
    Point p1(10, 15); // Normal constructor
    Point p2 = p1; // Copy constructor

   //Point p2(p1);

    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX()
         << ", p2.y = " << p2.getY();
    return 0;
}
```

Can we make copy constructor private ?

Yes, you can make a copy constructor private member in C++. Making so, you are restricting someone copying your object during object creation.

Objects of that class become non-copyable.

A common reason to make copy constructor private is to disable default implementation of these operations.

Why argument to a copy constructor must be passed as a reference?

If an object is passed as value to the Copy Constructor then its copy constructor would call itself, to copy the actual parameter to the formal parameter.

Thus an endless chain of call to the copy constructor will be initiated. This process would go on until the system run out of memory.

Why argument to a copy constructor should be const?

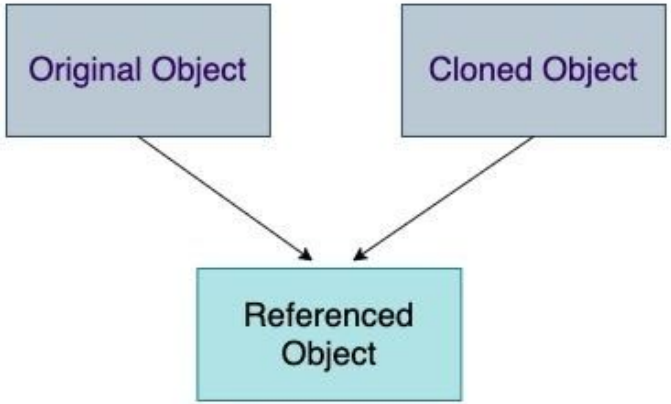
We should use const in C++ wherever possible so that objects are not accidentally modified

Shallow Copy

Default constructor always perform a shallow copy.

In shallow copy, an object is created by simply copying the data of all variables of the original object.

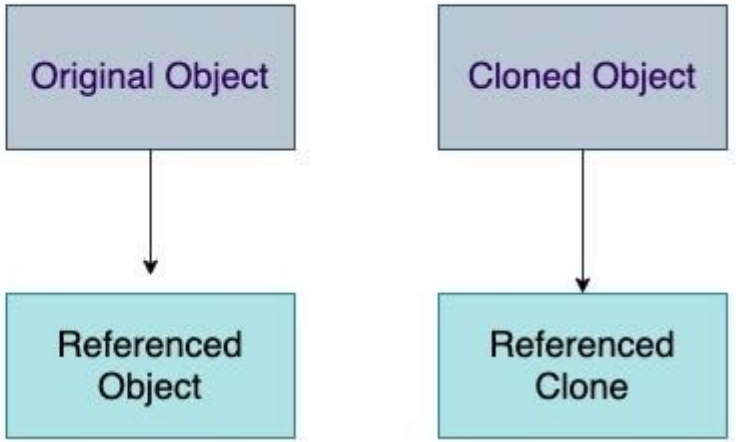
If some variables are defined in heap memory, then shallow copy has the same reference.



Deep Copy

Deep copy is only possible with user-defined copy constructors.

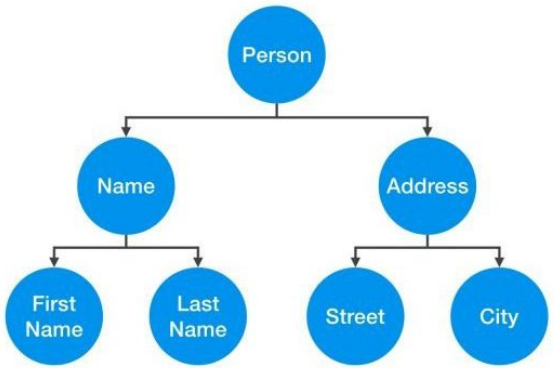
In user-defined copy constructors, we make sure that pointers (or references) of copied object point to new memory locations.

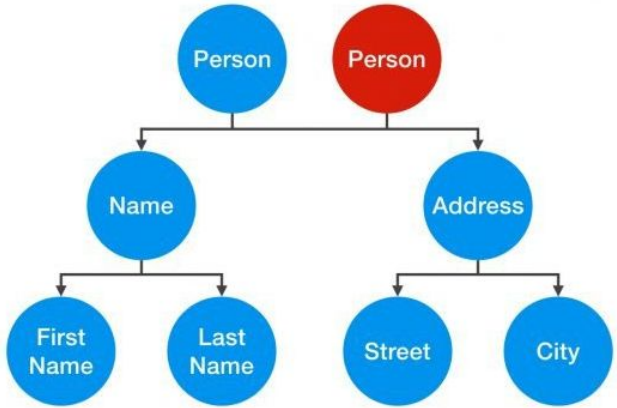


Shallow Copy

It copies the main object but does not copy the inner object. The inner object is shared between its original object and its copy as shown in the image below:

The problem with the shallow copy is that the two objects are not independent i.e if you modify for eg name object of one person it will reflect back to the other person object too.

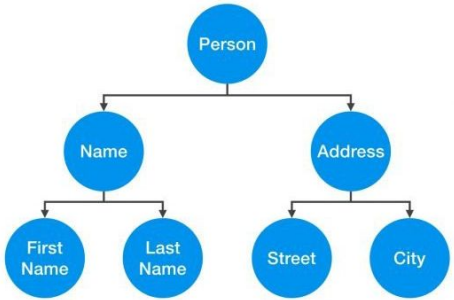


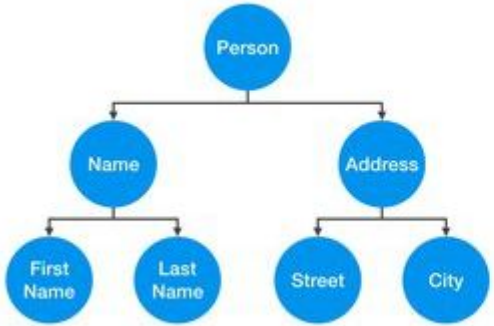


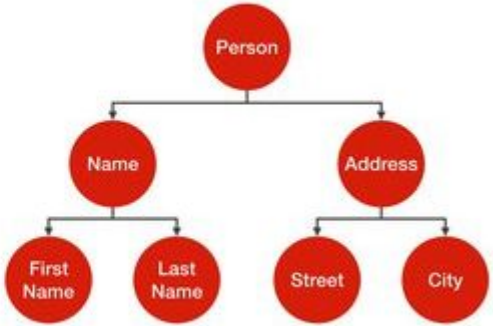
Deep Copy

In deep copy the complete structure is copied and not only the main object i.e there will be a replica of the original Person Object with same inner objects too. No sharing!!!.

Deep copy is totally independent of the original object i.e even if there is a change in the original object it won't get reflected in the copy.







Deep Copy

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

class box {
int length;
int* breadth;
int height;

public:
box() {
breadth = new int; }

box(box &sample)
{
length = sample.length;
breadth = new int;
*breadth = *(sample.breadth);
height = sample.height;
}
// Destructors
~box()
{
delete breadth;
}
};

void set_dimension(int l, int b, int h) {
length = l;
*breadth = b;
height = h;
}
void show_data() {
cout << " Length = " << length << "\n Breadth = " << *breadth << "\n Height =
" << height<< endl; }
```

Deep Copy

```
int main()
{
// Object of class first
box first;
// Set the dimensions
first.set_dimension(12, 14, 16);
// Display the dimensions
first.show_data();
box second = first;
// Display the dimensions
second.show_data();
return 0;
}
```

Topics Covered in Class

- Object's Array
- Passing an object to function as a parameter
- Object returning from a function

References

https://order66.medium.com/oop-series-what-is-an-object-b22fa34933f3

https://www.geeksforgeeks.org/copy-constructor-in-cpp/