بسم الله الرّحمٰن الرّحيم

CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

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

CASE STUDY 1

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).

CASE STUDY 2

A student wants to write an application that calculates his CGPA & finds if he is eligible for a Thesis.

Eligibility for Thesis requires 26 credit hours passed & CPGA of 3.0 or higher.

CASE STUDY 3

You need to write a function that encrypts a given variable by taking its reference and increasing its value by 5.

But you do not know beforehand the type of variable (it can either be an integer, character or float).

CONSTRUCTOR

- A *constructor* is called whenever an object of a class is created

```
class MyClass
{
  public:        //constructors are usually public
  MyClass()
  { //Things you want to do as soon as object is created }
}
```

CONSTRUCTOR

- C++ requires a constructor call for each object that is created

- A constructor *****looks***** similar to a function except that it does not return anything (thus it does not have a return type)

CONSTRUCTOR

- A constructor is used to initialize the data members/class variables for an object when that object is created

- If you don’t define a constructor in a class, the compiler itself provides a *default (no-parameter) constructor* for that class

CONSTRUCTOR

- 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");
}
```

CONSTRUCTOR

- 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

```
class A
{
    int var;
    string str;
  public:

   A(int v)
    {   var = v;
        cout << "Constructor v1: " << var << endl;
    }

   A()
    {
       var = 0;
       cout << "Default constructor: " << var << endl;
   }

   A(string s)
   {
        str = s;
        cout << "Constructor v2: " << str << endl;
   }
};
```

```
int main()
   {
       A dOb;
       A iOb(50);
       A sOb("Hello");
}
```

```
class A

{
   int var;
   string str;


   public:
   A(int v)
   {

               var = v;

               cout << "Constructor v1: " << var << endl;

   }


   A(string s)

   {

               str = s;

               cout << "Constructor v2: " << str << endl;

   }

};
```

```
int main()
{
        A dOb;    \\This line will give an error
        A iOb(50);
        A sOb("Hello");
}
```

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;
  if ( name.length() > 25 )
  {
        courseName = name.substr( 0, 25 );
        cout << "Name exceeds max length (25)“ << endl;
  }
}
```

SEPARATING INTERFACE FROM IMPLEMENTATION

- It’s a good practice to define member functions *outside* class definition
- Implementation details can be hidden from client code
- Do not write client code that depends on class’ implementation

SEPARATING INTERFACE FROM IMPLEMENTATION

```
#include <string.h>
using namespace std;

class A
{
  int a;

  public:
  int getA();
};
```

SEPARATING INTERFACE FROM IMPLEMENTATION

```
#include <iostream>
#include "A.h"
using namespace std;

int A::getA()
{
  a = 10;
  return a;
}
```

EXAMPLE: BANK ACCOUNT

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.

EXERCISE

Create a class called Payroll that includes three pieces of information as data members—a first name, a last name and a monthly salary. Your class should initialize the three data members as soon as an object is created. First name must always be provided beforehand, last name is an optional field, monthly salary should also be provided. However, if the salary isn't provided, we know that it can never be less than Rs20,000. Also make getter function for each data member. If the monthly salary input is not positive, set it to 0.

Write a test program that demonstrates class Payroll's capabilities. Create two objects and display each object's yearly salary.

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

- Has the same name as that of class
  Example: ~MyClass() { . . . }

- The destructor itself does not release object's memory, it just performs *termination tasks* right before the memory is reclaimed

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

ORDER OF CALLING DESTRUCTORS

```
class MyClass
{
    int objectID;

    MyClass(int objectID)
    {
            this->objectID = objectID;
    }

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

ORDER OF CALLING DESTRUCTORS

```
MyClass ob1 (1);

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

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

ORDER OF CALLING DESTRUCTORS

```
MyClass ob1 (1);   \\ destroyed fifth


void func()

{

   MyClass ob3 (3);                \\ destroyed second
   MyClass ob4 (4);                \\ destroyed first

}


int main()

{

   MyClass ob2 (2);                \\ destroyed fourth
   func();
   MyClass ob5 (5);                \\ destroyed third

}
```

WHY ARE DESTRUCTORS USEFUL?

- Useful for garbage collection

- Garbage-collected languages like JAVA do not have a destructor
  - There is no guarantee of when an object will be destroyed

EXERCISE

An application for a firm requires that records for different clients are maintained. There can be three different type of clients (categ1, categ2 & categ3). For each client, his full name, contact number, type, thumb impression & contractID must be recorded.

If the client wants to end his association with the firm, his record is deleted but before that, for categ1 & categ2 clients, a call is made to try & convince that client to reconsider his decision.

INITIALIZING ONE OBJECT WITH ANOTHER

```
class A
{
   int val;

   public:
   A(int val) { this->val = val; }
   A(){ }
   void setVal(int val) { this->val = val; }
   void showVal() { cout << "Value: "  << val  << endl; }
};
```

INITIALIZING ONE OBJECT WITH ANOTHER

```
int main()
{
  A  a1( 10 );
  a1.showVal( );

  A a2  =  a1;
  a2.showVal( );
  a2.setVal( 20 );

  a1.showVal( );
  a2.showVal( );
}
```

INITIALIZING ONE OBJECT WITH ANOTHER

```
int main()
{
   A a1( 10 );
   a1.showVal( );

   A a2 = a1;
   a2.showVal( );
   a2.setVal( 20 );

   a1.showVal( );
   a2.showVal( );

}
```

OUTPUT:

```
Val: 10   a1.val
Val: 10   a2.val

Val: 10   a1.val
Val: 20   a2.val
```

INITIALIZING ONE OBJECT WITH ANOTHER

```
int main()
{
  A a1( 10 );
  a1.showVal( );

  A a2;
  a2 = a1;
  a2.showVal( );
  a2.setVal( 20 );


  a1.showVal( );
  a2.showVal( );


}
```

OUTPUT:

```
Val:  10   a1.val
Val:  10   a2.val

Val:  10   a1.val
Val:  20   a2.val
```

COPY CONSTRUCTOR

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

- A copy constructor has the following prototype:

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

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

In C++, a Copy Constructor may be called when:

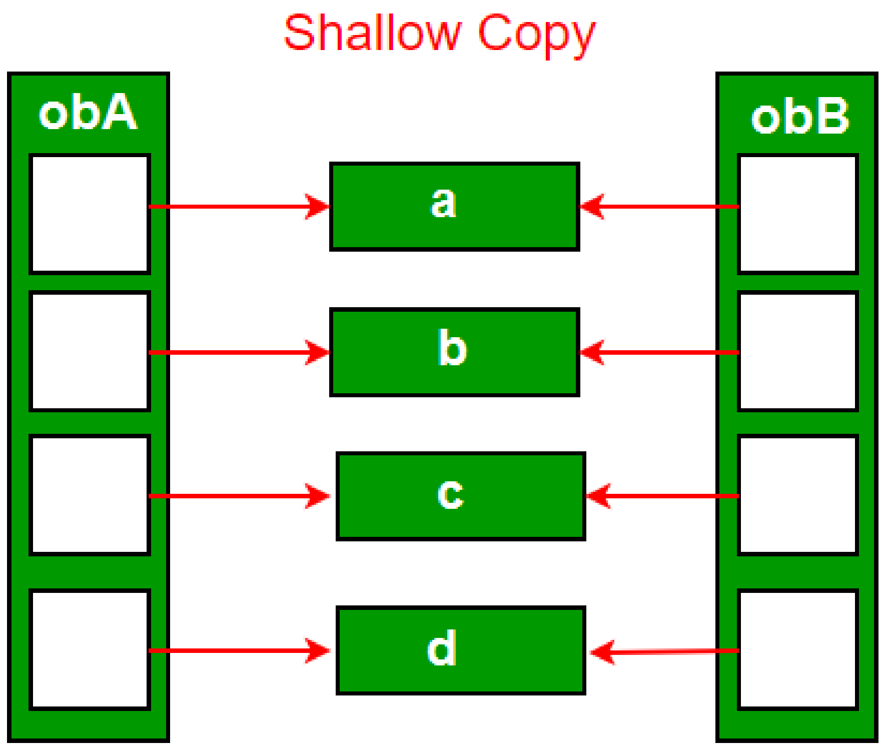
1) An object of the class is returned by value
2) An object of the class is passed (to a function) by value as an argument
3) An object is constructed based on another object of the same class
4) The compiler generates a temporary object

SHALLOW COPY

- Default constructor always perform a *shallow copy*

- Changes made by one object are also made for the other object

SHALLOW COPY

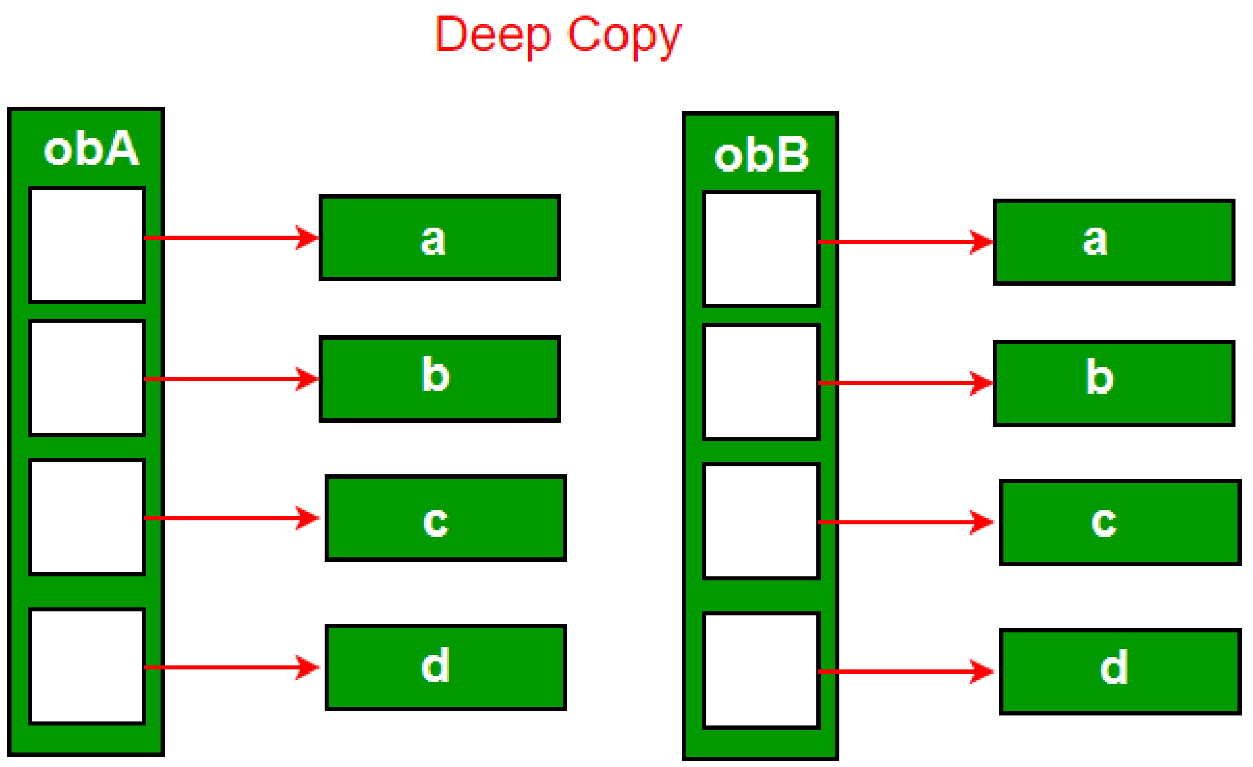


DEEP COPY

- 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

DEEP COPY



```
class A

{

   char *s; int size;

   public:


   A(const char *str = NULL)

   {

            size = strlen(str);

            s = new char[size+1];

            strcpy(s, str);

   }


   A(const A& ob)

   {

                         size = ob.size;

                         s = new char[size+1];
```

```
~A( ) { delete [] s; }

void print() {cout << s <<
endl;}

void change(const char
*str)
{
      delete [] s;
      size = strlen(str);
      s = new char[size+1];
      strcpy(s, str);
}


};
```

```
int main()
{
  A a1("Old string");
  A a2 = a1;

  a1.print();
  a2.print();

  a2.change("New string");

  a1.print();
  a2.print();
}
```

OUTPUT:

```
Old string    a1.s
Old string    a2.s

Old string    a1.s
New string    a2.s
```

WHY COPY CONSTRUCTOR?

- If we remove copy constructor from the above program, we don't get the expected output

- The changes made to a2 reflect in a1 as well which is never expected.

```
int main()
{
// with no copy constructor
  A a1("Old string");
  A a2 = a1;

  a1.print();
  a2.print();

  a2.change("New string");

  a1.print();
  a2.print();
}
```

OUTPUT:

```
Old string    a1.s
Old string    a2.s

New string    a1.s
New string    a2.s
```

DISCUSSION

- Can we make copy constructor private?
- Why argument to a copy constructor must be passed as a reference?
- Why argument to a copy constructor should be const?

CONST ARGUMENT FOR COPY CONSTRUCTOR

```
class Test
{
/* Class data members */
public:
Test(Test &t) { /* Copy data members from t*/}
Test()    { /* Initialize data members */ }
};
```

CONST ARGUMENT FOR COPY CONSTRUCTOR

```
Test func()
{
   cout << "func() Called\n";
   Test t;
   return t;
}

int main()
{
   Test t1;
   Test t2 = func();        \\Error at this line
```

SOLUTIONS

- Solution 1: Modify copy constructor:

```
Test(const Test &t) { /* Copy data members*/}
```

- Solution 2: Or do this (overloaded assignment operator):

```
Test t2;
t2 = func();
```

WITH THANKS TO:

SYED ZAIN-UL-HASSAN

Lecturer (Computer Science)