Object-oriented Programming

Lecture 4

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:

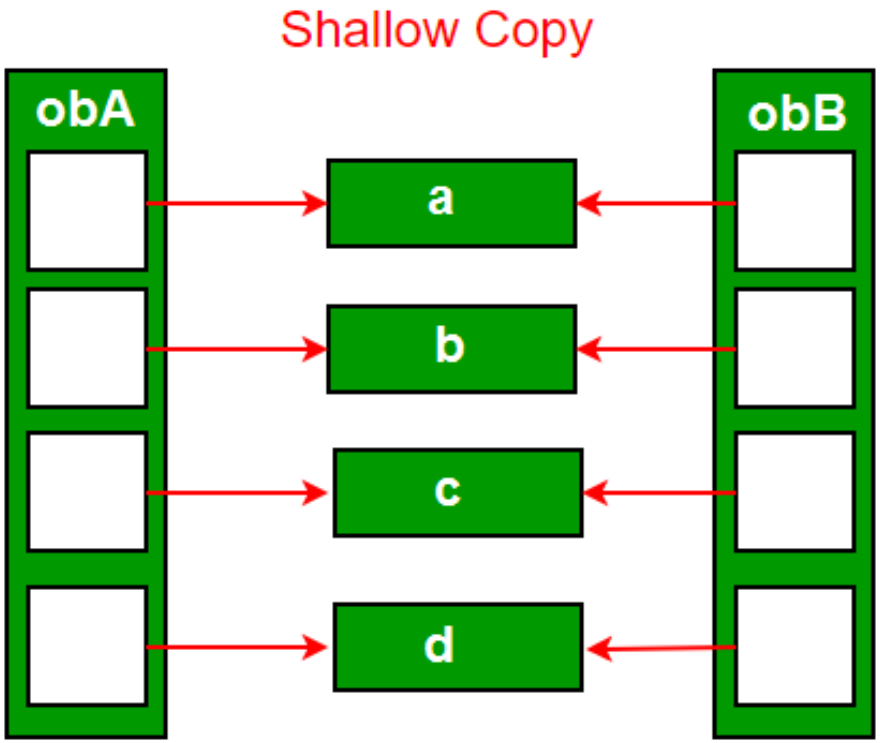
1) An object of the class is returned by value
2) An object of the class is passed (to a function) by value as an argument
3) An object is constructed based on another object of the same class
4) The compiler generates a temporary object

Shallow Copy

- Default constructor always perform a *shallow copy*

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

Shallow Copy

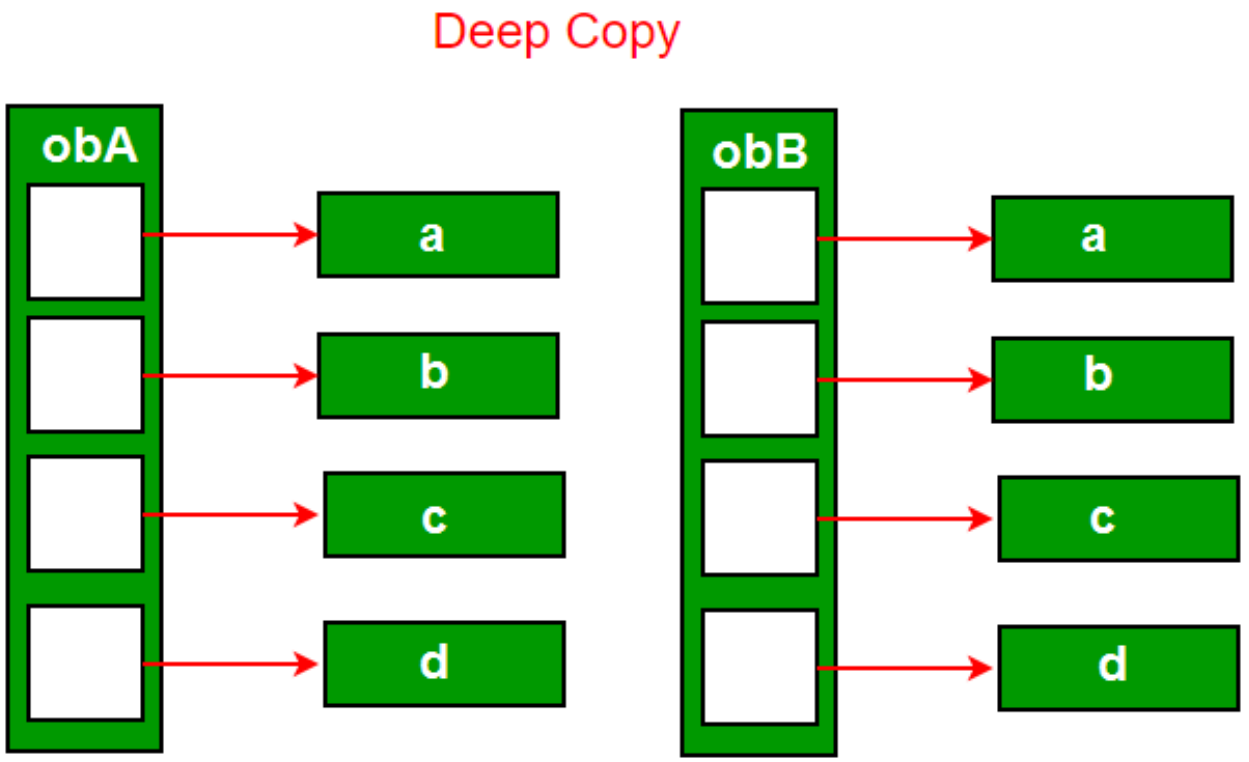


Deep Copy

- 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];
            strcpy(s, ob.s);
    }

    ~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*

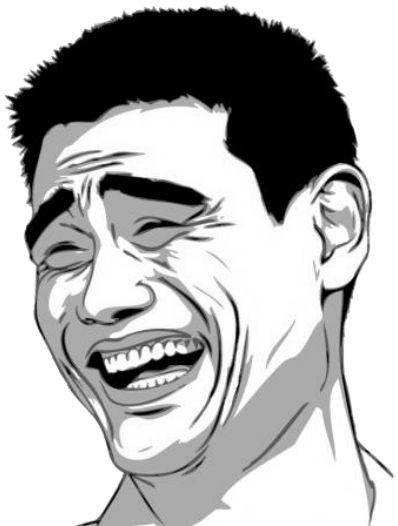
Why copy constructor?

- 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

Why copy constructor?

- We do not need user-defined copy constructor until there are pointers and dynamic memory allocations



```
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:

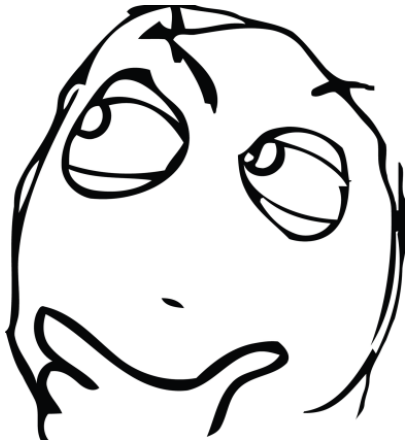
Old string *a1.s*
Old string *a2.s*

New string *a1.s*
New string *a2.s*

Discussion

- Can we make copy constructor **private**?

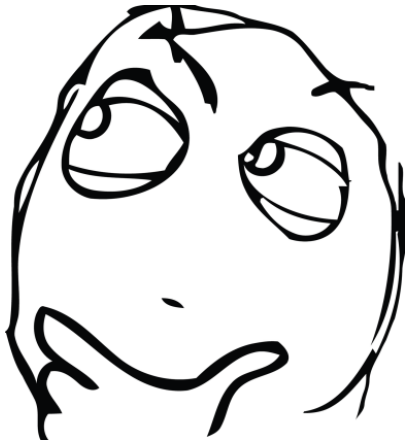
- Why argument to a copy constructor should be **const**?



Discussion

- Can we make copy constructor **private**?
  - Yes, when we don't want anyone to make copy of our objects

- Why argument to a copy constructor should be **const**?
  - To disallow changes in original object



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
    return 0;
}
```

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();