Object Oriented Programming in C++

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

Outline

- Review of Object Oriented Programming
- Four pillars - Abstraction, encapsulation, inheritance and polymorphism
- Pointers and their uses
- Dynamic Memory Management

Object Oriented Programming (OOP)

- In OOP, we define a class which is a template and create an instance or object of that class.
- A class has members (variables and functions) which are inherited by the object of the class.
- A class allows new user-defined data type and works as an object constructor.

OOP vs Procedural Programming

- OOP improves on procedural programming in several ways.
- Procedural programming uses procedures (functions) which contains computation steps related to a task. A procedure can be called multiple times and from any where in the program (even from another procedure) which allows code reuse.
- OOP is faster and easier to execute, provides a clear program structure, code becomes easier to maintain, modify and debug, and creates full reusable applications.

Encapsulation

- Encapsulation is binding together the data and the functions that manipulate the data.
- In OOP, encapsulation is achieved by means of classes.
- A class encapsulates the data and the functions related to that data.
- Encapsulation protects an object from unwanted access by users of the class. Therefore, encapsulation also ensure the security of the data.
- To ensure good encapsulation, must declare class variables as private (cannot be accessed from outside the class).
- It reduces human error, simplifies code maintenance, and makes code understandable.

Abstraction

- Abstraction is hiding the background (complex) details and exposing the information actually needed by the user to operate.
- In OOP, abstraction is achieved by classes.
- In a class, we decide which class member (variables and functions) is accessible by the outside world (class's user).
- Access specifiers (private and protected) are used to limit access of a member to the outside world.
- Make data members private, but the functions which manipulate the data members should be public.

Abstraction

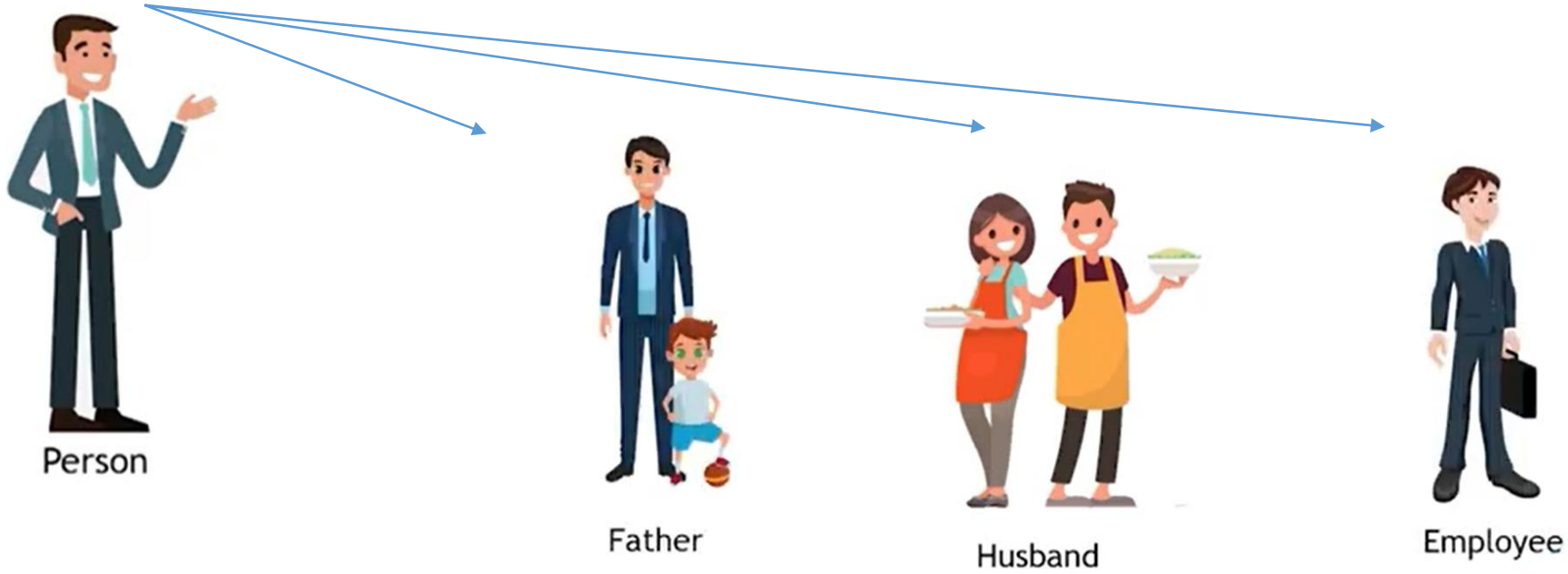
- In C++, header files also provide abstraction. For example, the *pow()* function in *math.h* header file.
- Internal implementation can be changed without affecting the usability.
- Improves code reusability and security of an application.

Inheritance

- Inheritance is using properties and characteristics from another class (Base class) due to some shared functionalities.
- The Subclass (class which inherits) can have some inherited members as well as some unique members.
- It improves code reusability.
- Inheritance can be in public, private or protected mode.
- Inheritance can be multiple and multilevel.

Polymorphism

- Poly means many and morph means form.
- Polymorphism means many forms or many behaviours.



Polymorphism

There are two types of polymorphism.

1. Compile time polymorphism (static polymorphism) is achieved through Method Overloading.
2. Run time polymorphism (dynamic polymorphism) is achieved through Method Overriding.

Overloading (function)

- A class can have many functions with the same name.
- Overloading needs to meet three conditions. Functions should,
  - have same name
  - be in the same class
  - have different parameters.
- It can be achieved by,
  - Different number of parameters
  - different type of parameters
  - different sequence of parameters

Overloading (operator)

- Operators can also be overloaded.
- For example, ‘+’ operator can be used to concatenate two strings.

Overriding

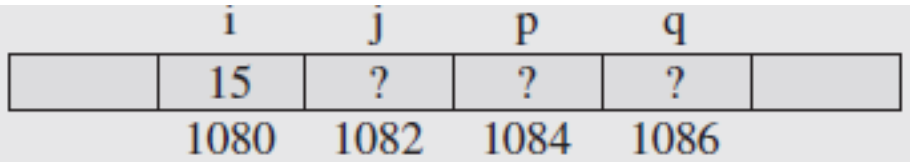
- Involves inheritance (specially for OOP)
- A method of a child class can override a method of base class.
- Derived classes can respond to a method of base class in their own unique way.
- Name of the parent and child methods should be same.
- Parameters of the parent and child methods should have same sequence, types, and count.
- For example, GetAccountInfo method in class Account can be implemented differently by the derived classes SavingAccount and CurrentAccount.

Pointers

- Pointer is a variable which holds address of another variable.
- The power of pointer is in directly modifying data at a memory location.
- It allows to access same memory space from different locations.
- Therefore, change made to a data using pointer in one location can be seen in other locations of the program.
- A pointer also has an address which can be saved in another pointer (called pointer to pointer).
- Pointers save space, as memory components are shared, and enables dynamic memory allocation.

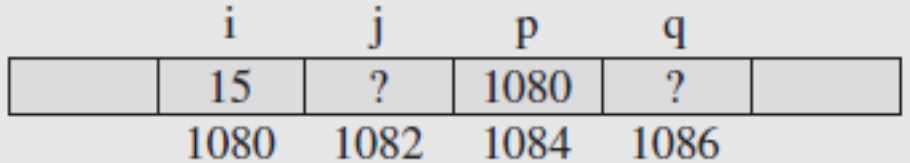
Pointers

```
int i = 15, j,
        *p, *q;
```

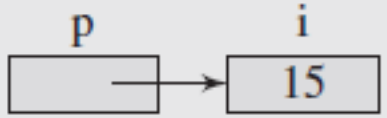


(a)

```
p = &i;
```



(b)



(c)

- ***p = 20;**

- **j = 2 * *p;**

Pointers

- **q = &i;**
- ***p = *q – 1;**
- **q = &j;**
- ***p = *q – 1;**

Pointers

Array as Pointer

- The name of an array, in a normal array declaration, is a pointer holding the initial address of a block of memory. *E.g. int a[5];*
- However, a's address can't be changed. *E.g. int *p; a=p; or a++;* ✖

For example, sum of elements of 'a' can be calculated in multiple ways.

```
for (sum = a[0], i = 1; i < 5; i++)
        sum += a[i];

for (sum = *a, i = 1; i < 5; i++)
        sum += *(a + i);

for (sum = *a, p = a+1; p < a+5; p++)
        sum += *p;
```

Memory Allocation in C++

- A C++ program uses mainly two types of memory.

Static memory (Stack):

- In static memory allocation, the size of memory needed for a variable is known during compilation. This memory is allocated in continuous locations. The variables belonging to a function call are allocated memory in stack automatically when the call is made. When the function call is over the memory for the variables gets deallocated. Also, the global variables are allocated in stack memory (and gets deallocated automatically when the program terminates).

Memory Allocation in C++

Dynamic memory (Heap):

- The need for memory allocation rise during the execution of a program (i.e., at run time) are handled via Heap. It is used when the exact size of memory is unknown at compilation. Therefore, a pointer is used to link to a dynamically allocated memory.
- Note: A program should deallocate memory when not using it anymore. Otherwise, it may lead to memory leak (which may slow down a computer's performance by reducing the memory's availability). It can be fatal for programs which never terminate like, servers.

Dynamic Memory Allocation

- Manual allocation and deallocation of memory by a program.
- ‘new’ operator is used to allocate and ‘delete’ operator is used to deallocate memory dynamically.
- ‘new’ takes from memory as much space as needed to store an object.
- The space will depend on the type (if it is an array then also on the size of array)

Dynamic Memory Allocation

Dynamic memory allocation example of a single variable:

```
int * ptr = new int;          // one dynamic integer
```

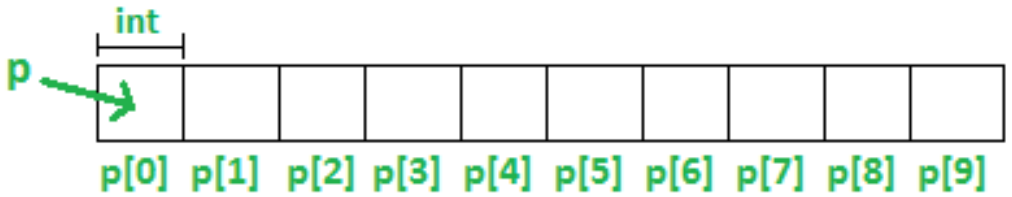
- To access data, dereference is needed,

```
cout << ptr; // prints the pointer contents (address of data)
cout << *ptr; // prints the data
```

Dynamic Memory Allocation

- Pointer can be used along with the ‘new’ operator to dynamically allocate a block of memory (or an array).

```
int * nums = new int[size];          // int size=10;
```



- For a dynamically created array, the pointer acts as the array name.

E.g. nums[2] = 33;

Dynamic Memory Allocation

Advantages:

- A program can assign memory of variable size.
- A program can allocate and deallocate memory as per need.
- This flexibility helps in implementing useful data structures, like Linked List, Tree, etc.

Pointers

- When the variable pointed by a pointer is not needed, then it is deleted using ‘delete’ instruction. E.g. *delete ptr;*
- It returns or frees the pointed memory back to operating system.
- Note, it does not actually delete the pointer or pointed memory. The pointer still points to the same mem location and the mem location can have the same data (or may have different data).
- A pointer that is pointing to deallocated memory is called a **dangling pointer.**
- After the delete command, set the pointer to 0 or nullptr. Otherwise, it can cause undefined behavior.

Pointers

- Also, if reinitialize the pointer (p= new int) without deleting it first then it causes memory leak.
- The old pointed memory can’t be used by OS (for the duration of the program).
- Similarly, delete a dynamically assigned array. E.g. *delete [] nums;*
- It will free the entire array pointed by *nums*.
- Otherwise, the array remains in heap until the program execution.
- The brackets indicate that *nums* points to an array.

Pointers

Pointer to Class:

- This pointer stores address of an object of a class.
- The pointer should have same type as the object.
- To call a function of the class, '->' is used instead of dot operator.

```
Rectangle *r1, *r2, *rlist;
r1 = new Rectangle;              //uses default constructor
r2 = new Rectangle(4,3.0);       // uses constructor with an int and a float
                                 // argument.
rlist = new Rectangle[10];                  // uses default constructor for 10 objects.
r1->setData(5, 9.5);             // calls function setData from Rectangle class.
```

Pointers

Pointer to Class:

```
class Rectangle{
   private:
            int length;
            float breadth;
    public:
            Rectangle(int l=0, float b=0){          length = l;  breadth = b; }

            void setData(int l, int b) {
                        length=l;  breadth=b;
                       }
                       int getArea() {
                                  return 2*length*breadth;
                       }
};
```

Outline

- Functions and Reference Parameters
- Function Parameters in C++
- Function pointers and (assignments and call)
- Member Function Pointers
- Polymorphic Functions (Overriding)
- Constructor, Destructor, Copy Constructor, and Copy Assignment Operator.

Pointers and Reference Variables

- A reference variable is declared using &.

  E.g. int n, &r = n; // r becomes an alias of n.

- It should be initialized during declaration and its reference can't be changed after the declaration.

  - int n = 5, *p = &n, &r = n;
  - cout << n << ' ' << *p << ' ' << r << endl; // 5 5 5
  - n=7; cout << n << ' ' << *p << ' ' << r << endl; // 7 7 7
  - *p=4; cout << n << ' ' << *p << ' ' << r << endl; // 4 4 4
  - r=3; cout << n << ' ' << *p << ' ' << r << endl; // 3 3 3

Function Parameters in C++

Pass by Reference:

- Reference variable is used when passing an argument as reference in a function call.
- Copies the reference of an argument to function's formal parameter.
- Changes made inside the function are reflected in the passed argument.

```
void f1(int i, int& k) {
        i = 1;
        k = 3;
}
```

Function Parameters in C++

- Reference type is also used in defining the return type of a function.

```
int& f2(int a[], int i) {
    return a[i];
}
```

- declaring the array, int a[] = {1,2,3,4,5};
- f2() can be used on either side the assignment operator.

n = f2(a,3); // returns 3rd element, 4, with its reference;

f2(a,3) = 6; //assigns 6 to a[3] so that a = [1 2 3 6 5].

Function Parameters in C++

Pass by Address:

- Also called pass by pointer, it copies the address of the argument to the function's formal parameter.
- Similar to pass by reference, changes made to the function's parameters are reflected in the arguments passed.

Function Parameters in C++

Rules of Thumb:

- Call-by-value is appropriate for small objects that should not be altered by the function.
- Call-by-constant-reference is appropriate for large objects that should not be altered by the function and are expensive to copy.
- Call-by-reference is appropriate for all objects that may be altered by the function.

Pointers to Functions

- Just like a variable has a memory address which can be used by a pointer, a function also has an address which can be used with pointers.
- A function's address tells where in memory is the start of the body of the function.
- When a function is called, the system transfers control to this address to start the execution of the function.
- A pointer to function is useful in implementing functionals (which is a function which takes another function as its argument) and to create a callback function.

Syntax: `return_type (*pointer_name)(arguments)`

Pointers to Functions

- For a function f,

  double f(double x) { return 2*x;}

- f is the pointer to function f(), *f is the function itself, and (*f)(2.2) is the call to function with 2.2 as the argument.
- Consider writing a function to compute the sum, $\sum_{i=n}^{m} f(i)$
- It will take ‘n’ and ‘m’ as well as the function ‘f’ as the arguments.

```
double sum(double (*f)(double), int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

Pointers to Functions

- Now ‘sum’ can be called like,
  - Cout << sum(f, 3, 8); // using user defined function f
  - Cout << sum(sin, 3, 8); // using built-in function sin

- Note, the function and the pointer to function must have the same return type and the same (number of) argument(s).
- A programmer's type-defined declaration (using keyword ‘typedef’) can be used to create an alias to a pointer to function.

```
typedef double (*Diameter)(double);
```

Classes and Pointers to Functions

- Using the type-defined declaration, we can define pointer to function as a class member.

```
typedef double (*Multiply)(const double a);
class CSquare {
public:
    Multiply Perimeter;
};
```

Member (Function) Pointers

- It is a pointer to a (public) member function.
- Unlike ordinary functions, member functions have a special parameter called ‘this’ which means this object or an object is needed to serve as ‘this’.
- Therefore, you need an object on which you can call a member function, when calling via a member function pointer.

Declaration Syntax:

```
return_type (class_name::*pointer_name) (parameter types)
```

Setting Syntax:

```
pointer_name = &Class_name::function_name;
```

Calling Syntax:

```
result = (object.*pointer_name)(arguments);
```

using pointer to object,

```
result = (object_ptr->*pointer_name)(arguments);
```

Member (Function) Pointers

- Pointers to members are used to refer to non-static members of class objects.
- They can be used to assign values to them, and to call member functions.
- The pointer to member operators .* and ->* are used to bind a pointer to a member of a specific class object.
- Because the precedence of () (function call operator) is higher than .* and ->*, you must use parentheses to call the function pointed to by ptf.
- A member function pointer can be reassigned to point to other member function.

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

class X {
public:
  int a;
  void f(int b) {
    cout << "The value of b is "<< b << endl;
  }
};

int main() {

  // declare pointer to data member
  int X::*ptiptr = &X::a;

  // declare a pointer to member function
  void (X::* ptfptr) (int) = &X::f;

  // create an object of class type X
  X xobject;

  // initialize data member
  xobject.*ptiptr = 10;

  cout << "The value of a is " << xobject.*ptiptr << endl;

  // call member function
  (xobject.*ptfptr) (20);
}
```

Polymorphic Functions (Overriding)

- In OOP, polymorphic functions mean the same function name can be used by other functions that are members of different classes.
- Note, a pointer of base class type can point to the objects of base class as well as to the objects of a derived class.

```
Base* bptr;        // A pointer to base class called Base
Derived d;         // A pointer to a derived class called Derived
bptr = &d;         // A valid assignment
```

- To achieve runtime polymorphism, we can define a member function of the base class using the keyword ‘virtual’.
- Then, this function is overridden in the derived classes.

Polymorphic Functions (Overriding)

- When referring to a derived class's object using a pointer to the base class, the virtual functions allow to use the derived class's version of the function.
- Therefore, virtual functions should be accessed using pointer or reference of base class type.
- They ensure that regardless of the type of object which is calling the function, the correct version of the function is executed.
- Virtual function call is resolved at the run time.
- Virtual functions can be used to achieve polymorphism without inheritance.

Polymorphic Functions (Overriding)

- Pure virtual functions are virtual functions without an implementation in the base class.
- Instead, they are declared by assigning zero to them.
- A class with one or more pure virtual function becomes an abstract class.
- An abstract class is one which can’t be instantiated. However, we can have pointer or references to it.
- If pure virtual function is not overridden in a derived class, then the derived class also becomes abstract class.

Constructor

- A constructor is a special member function of a class which is automatically called when an object is initialized, and it has no return type.
- Different constructors can be defined to allow object initialization using different number of parameters.
- Any constructor must be named same as the class name. So the rules of function overloading applies here.
- Every class has a default constructor which takes no arguments and has no body.

Destructor

- A member function to delete an object of the class.
- It is not a value-returning function (not even void).
- A class can have only one destructor, and the destructor has no parameters.
- The name of the destructor is same as the class name, preceded by a tilde (~) character.
- Destructor is automatically called when the class object goes out of scope.
- Compiler creates a default destructor if no definition is provided.

Destructor

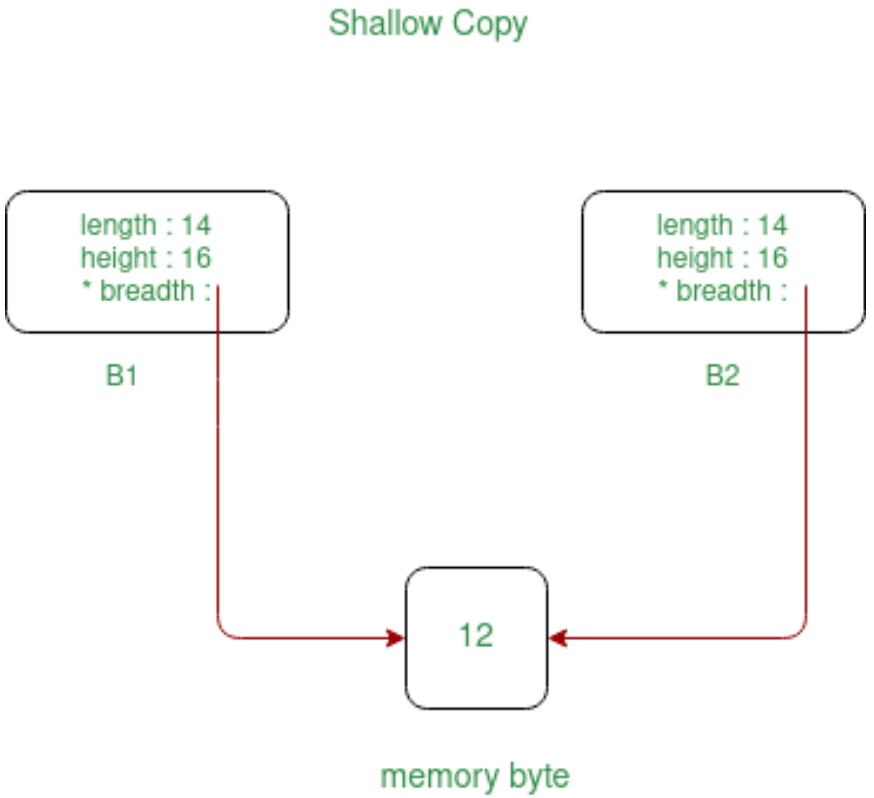
- If the class does not have dynamic memory allocation (pointer members), the default destructor work fine.
- If there are pointer members, write a destructor to explicitly release memory which was dynamically allocated (using delete operator). Otherwise, memory leak can happen.

Copy Constructor

- Apart from default constructor and the parameterized constructors, copy constructor is used in OOP.
- A copy constructor is used to properly initialize an object by copying data from another object of the same class.
- It is used when an object is passed by value to a function, returned (by value) from a function, or an object is initialized from an existing object of the same class.
- If the class contains any pointer member, then the default copy constructor (which is used when no copy constructor is provided) makes a shallow copy of the object.

Copy Constructor

- Shallow copy is a member-wise copy; simply copying the data of all variables of the original object.
- When having a pointer variable, the copied object's pointer variable will also reference the same memory location as the original object.
- Therefore, the default copy constructor causes aliasing for pointer variables and does not serve the purpose of making a replica.



Copy Constructor

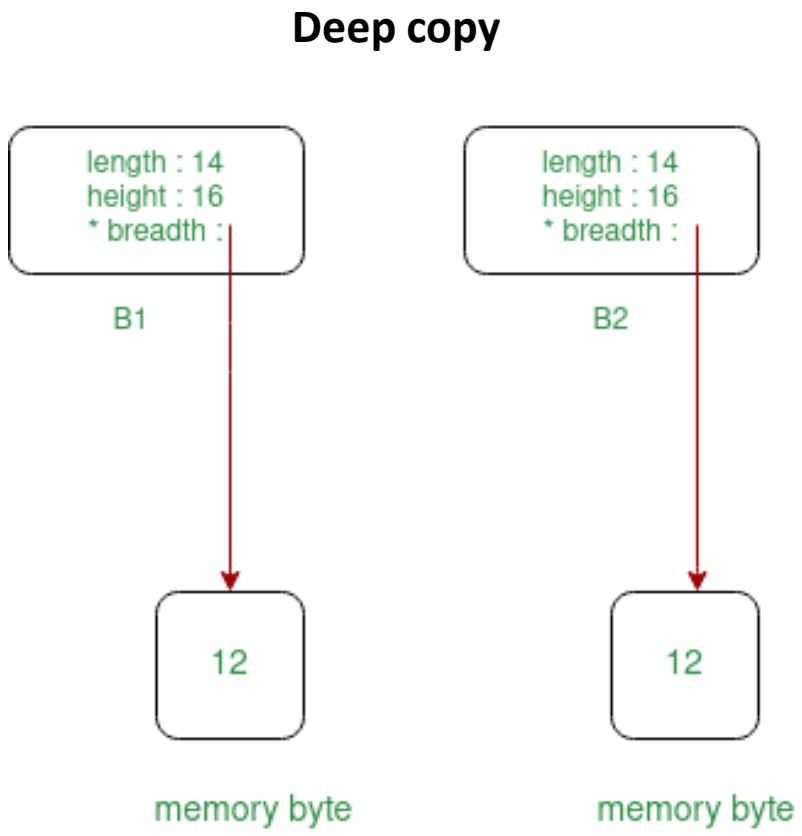
- Not writing own copy constructor causes two problems:

1. Pointer variables of the two objects point to the same mem location which results in overwriting data of one object by another.
2. Destructor is called on same mem location twice.

- Deep copy is created by explicitly defining a copy constructor and dynamically allocate memory for pointer variables of the copied object.

Syntex:

```
className(const className& otherObject);
```



Copy Constructor

- Argument is passed by reference to avoid non-terminating chain of calls. If we pass argument to copy constructor by value, then it needs to make a copy which calls the copy constructor by value which again calls it to make a copy, and so on.
- Copy constructor can be made private if we don’t want to make a deep copy or don’t want users to make copy of class object. So, an attempt to make copy will give compiler error.

Copy Assignment Operator

- The built-in assignment operator for classes with pointer member variables makes a shallow copy when it performs member-wise copying.
- It causes same issues as seen before with copy constructor.

```
className& operator=(const className& rightObject){
      // local declaration, if any
      if (this != &rightObject){
            // free previously allocated memory
            // allocate new memory
            // make deep copy of rightObject
      }
      return *this;
}
```

Copy Assignment Operator

- Every member function has access to a variable named *this* that is a pointer to the object whose member function was called. For example, when *object1 = object2;* is executed, *object1*'s member function *operator=* is called, so *this* is a pointer to *object1*.

  It differs from copy constructor in that,

  - Object which is copying from the other object has been initialized. It means it already has dynamic memory assigned (for pointer variable) which needs to be released. Then, new dynamic memory is allocated for that variable.
  - A variable assigned to itself is allowed and handled to avoid unnecessary computations.
  - It must return a value.