C++ copy constructor

A copy constructor is an overloaded constructor used to declare and initialise an object from another object.

→ Two types

(i) Default copy constructor. - The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.

(ii) User defined copy constructor - User defined.

→ Syntax of user defined copy constructor

```
class_name ( const class_name & old_object);
```

ex.

```
class A
{
    A( A &x )      // copy constructor
    {
        // copy constructor.
    }
}
```

In above case, copy constructor can be called as -

```
(i)   A a2(a1);     → a1 initialises the a2 object.
(ii)  A a2 = a1;
```

ex.

```
class A {

  public :
        int x ;

        A ( int a) {              // parametrized constructor.

              x = a;
        }


        A (A & i) {               // copy constructor.

              x = i. x
        }

};

int main() {

      A  a1(20);
      A  a2(a1);                  // calling copy constructor.

      cout << a2 . x ;
```

output

```
20
```

→ Copy constructor is called when -

(i) When we initialise the object with another existing object of the same class type. For example,

```
Student  S1 = S2;
```

(ii) When the object of the same class type is passed by value as an argument. ex. A a2(a1)

(iii) When the function returns the object of the same class type by value.

→ Two types of copies are produced by the constructor,

(i) shallow copy (ii) Deep copy.

→ shallow copy

(i) The default copy constructor can only produce the shallow copy.

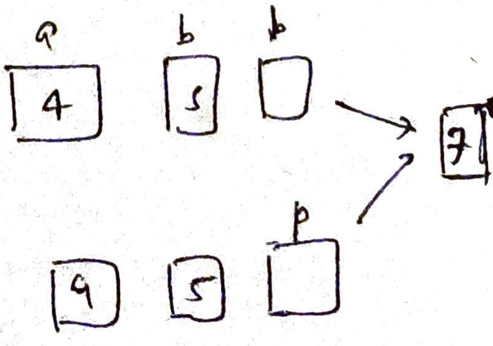
(ii) A shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

example →

```
class  Demo {

    int a;
    int b;
    int* p;

    public:

        Demo () {

            p = new int;

        }

        void   setdata ( int x,  inty,  int z ) {

            a = x ;
            b = y ;
            *p = z;

        }

        void   showdata () {

            cout   << a  << endl
            cout   <<  b   << endl
            cout    << *p   << endl

        }

    };

int  main () {

    Demo d1;
    d1 . setdata ( 4, 5, 7 );

    Demo d2  =  d1;

    d2 . showdata ();

}
```

Output

```
4
5
7
```



In above case, we have not defined a copy constructor, (C,
thus the statement d2 = d1 calls the default constructor
defined by the compiler.

Default constructor creates an exact copy / shallow copy
of existing object. Thus, the pointer p of both the objects
point to the same memory location.

Thus, if the memory of a field is freed, the memory of
another field is automatically freed as both fields point to
same memory location.

This problem is solved by user defined copy constructor
which produces a deep copy.

→ Deep Copy

Deep copy dynamically allocates the memory for the
copy and then copies the actual value, both the
source and copy have distinct memory locations.

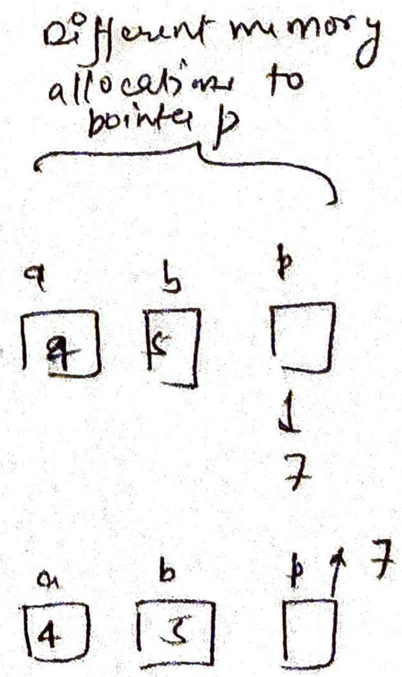
This way, both the souce and copy are distinct
and will not share the same memory location. Deep
copy requires to write user defined constructor.

example

```
class Demo {

  public:
      int a
      int b
      int *p

      Demo(){
          p = new int;
      }

      Demo ( Demo & d){          // copy constructor.

          a = d.a ;
          b = d.b ;
         *p = *(d.p);

      }

      void setdata ( int x, int y, int z){

          a = x ;
          b = y ;
          *p = z ;

      }

      void showdata (){

          cout << a << endl
          cout << b << endl
          cout << *p << endl

      }
  };

  int main (){

      Demo d1;
      d1. setdata (4,5,7);
      Demo d2 = d1;
      d2. show data();
  };
```

Output

```
4
5
7
```



→ Deep copy does not create copy of a reference type variable.

→ copy constructor → invoked when the new object is initialised with existing object.

assignment (=) → invoked when we assign existing object to a new object.

→ For copy constructor (user defined), both the existing and new object share different memory locations.

→ In assignment (and default copy constructor), the existing and new object share the same memory location.

C++ structs

In C++, classes, and structs are blueprints that are used to create the instance of a class. structs are used for lightweight objects like Rectangle, color, Point etc.

Unlike classes, structs in C++ are value type than reference type. It is useful if we have data that is not intended to be modified after creation of struct.

C++ structure is a collection of different data types. It is similar to the class that holds different types of data.

→ Syntax

```
struct struct_name {
        // member declarations
}
```

ex.

```
struct student {
        char name [20];
        int id;
        int age;
}
```

→ when structure is declared, no memory is allocated. when variable of a structure is created. then the memory is allocated.

→ Accessing struct member variables:

```
s. id ;              { using dot (.) operator }
s. name ;
```

example

```
struct Rectangle {
        int width, height;
    }
int main () {
        struct Rectangle rec;
        rec. width  = 8 ;
        rec. height = 5 ;
        cout << " area" << (rec. width * rec. height) << endl;
    }.
```

output

area 40.

→ In a struct, if access specifier is not declared explicitly, then default access specifier will be public.

→ If access specifier is not declared explicity, then by default, access specifier will be private.

example

```
struct Rectangle {
     int width, height;
     Rectangle ( int w, int h) {
          width = w;
          height = h;
          }

     void  area () {
          cout << (width * height);

        }
   };

int main () {
     struct Rectangle  rec = Rectangle (4, 6);
     rec. area ();

     }
```

output

```
24.
```