OOPs 2

Shallow and Deep Copy

Shallow Copy:

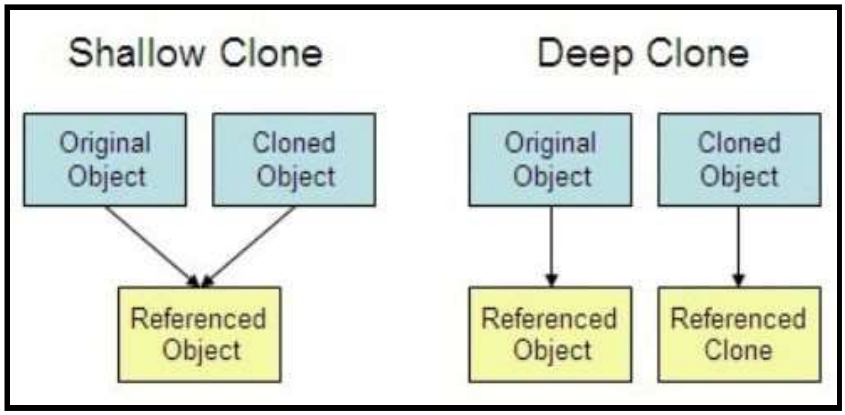
It copies all of the member field values. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred.

Note: Assignment operator and the default copy constructor makes a shallow copy.

Deep Copy:

It copies all the fields but creates a copy of dynamically allocated memory pointed to by the fields. Here, we need to make our copy constructor and overload the assignment operator. Advantages of the deep copy are:

- When we need to initialise the class variables to some value or NULL, then a parameterized constructor can be used.
- A destructor that can be used to delete the dynamically allocated memory.



Kindly refer to the code below:

```
class Student {
```

```
        int age;
        char *name;
        public :
        Student(int age, char *name) {   // parameterized constructor
                this -> age = age;
                // Shallow copy
                // this -> name = name;

                // Deep copy
                this -> name = new char[strlen(name) + 1];  // Created a new memory
                strcpy(this -> name, name);
        }
};
```

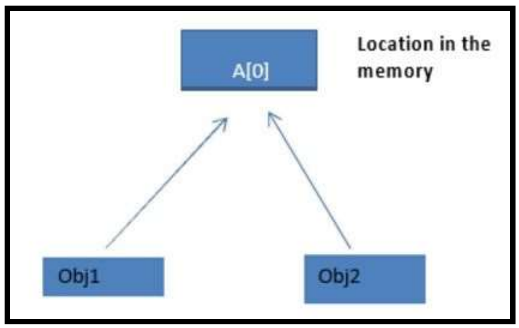
Copy Constructor

The copy constructor is used to create a copy of an already existing object of class type. We can make this copy in two ways:

Shallow copy constructor:

Let's take an example to understand it better. Suppose, two students are entering their details simultaneously from two different machines that are connected over the same network. It will reflect the changes made by both of them in the database because they both are entering their details in the same database.

The same is the case with the shallow copy as both share the same locations. It copies references to the original object. **Default copy constructor** uses the same way of copying. It is generally preferred in the cases when the class does not contain any dynamically allocated memory.



Deep copy constructor:

Suppose you have to submit the homework but are short of time. So, you decide to copy the same from your friend. Now, you and your friend have the same work but possess different copies. Now, you choose to modify yours's a bit to prevent the risk of the teacher figuring it out. But the changes you did will not affect your friend's work. Deep copy follows the same concept.

Deep memory allocates different memory for the copied task. It is generally used when we assign the dynamic memory using pointers.

Kindly refer to the code below:

```
// Copy constructor
Student(Student const &s) {      // keyword const assured us that original copy is
                                 // unchanged
        this -> age = s.age;
        // this -> name = s.name;          // Shallow Copy

        // Deep copy
        this -> name = new char[strlen(s.name) + 1];
        strcpy(this -> name, s.name);
}
```

Initialisation list

Using the initialisation list, we can directly assign the values to the data members of the class. Though we can use the initialisation list as needed, in the following cases, we mandatorily need to use the same:

- When no base class default constructor is present
- When the data members are of type **const**
- When the data member and parameter have the same name
- When the data member of the reference type is used.

The syntax begins with a colon(:) and then each variable along with its value separated by a comma. It does not end in a semicolon.

Kindly refer to the code below for better understanding:

```
class Student {
        public :

                int age;
                const int rollNumber;     // Const type data member
                int &x;                   // age reference variable
                // Parameterised list is used
                Student(int r, int age) : rollNumber(r), age(age), x(this -> age) {
                        //rollNumber = r;
                }
};
```

Constant functions

Constant functions are those which don't change any property of current objects. Only constant objects of the class could invoke these.

Syntax:

```
datatype function_name const();
```

Static Members

Suppose, we have a student database and we are creating objects for each student specifying their name, roll number and school name. As discussed earlier, deep copy constructor will be used for the same. For each student, their names and roll number can be different so we will be allotting distinct memories to each. But the school name should be the same. By using a deep copy, we will allocate each student a different memory for the school name. To overcome this, we will be using the **static** keyword.

Syntax for declaring static members:

```
static datatype variable_name;
```

Since it is a property of the class and not of an object, we can't simply access these values from the object name followed by the variable name.

Syntax for invoking static members:

```
Class_name :: variable_name;
```

Note: (::) is called the Scope Resolution Operator.

Apart from variables, functions can also be made static. By declaring a function as static, we make it independent of any particular class object. A static function can be invoked even if no objects of the class exist and the static functions are accessed using only the class name and the (::) operator.

Kindly refer to the code below for better understanding:

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

class Student {
        static int totalStudents;          // total number of students present

        public :
        int rollNumber;
```

```
        int age;

        Student() {
                totalStudents++;
        }

        int getRollNumber() {
                return rollNumber;
        }

        static int getTotalStudent() {          // Static member function
                return totalStudents;
        }
};
int Student :: totalStudents = 0;   // initialize static data members

int main() {
        Student s1;
        Student s2;
        Student s3, s4, s5;

        cout << Student :: getTotalStudent() << endl;  // static function invoked
}
```

Operator Overloading

In C++, we can specify more than one definition of an operator in the same scope, which is called operator overloading. It makes the program more intuitive. Example, if we want to add two fraction numbers, then we can do the same by calling over the numerator and denominator of both fraction object F1 and F2. Using operator Overloading, it can be simply done by F1 + F2. You can see that we are adding two class objects, which is primitively not possible but is possible using operator overloading of '+' operator.

Syntax:

```
return_type operator symbol_to_be_overloaded(arguments){}
```

Here, **operator** is a keyword.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).
- Operators = and & are already overloaded in C++, so we can avoid overloading them.
- Precedence and associativity of operators remain intact.

List of operators that can be overloaded in C++:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

Kindly refer to the code below:

```
class Fraction {
        private :
                int numerator;
                int denominator;

        public :
                Fraction(int numerator, int denominator) {
                        this -> numerator = numerator;
                        this -> denominator = denominator;
```

```
    }

    void print() {
        cout << this -> numerator << " / " << denominator << endl;
    }

    void simplify() {
        int gcd = 1;
        int j = min(this -> numerator, this -> denominator);
        for(int i = 1; i <= j; i++) {
            if(this -> numerator % i == 0 && this -> denominator % i
== 0) {
                gcd = i;
            }
        }
        this -> numerator = this -> numerator / gcd;
        this -> denominator = this -> denominator / gcd;
    }

    Fraction add(Fraction const &f2) {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator+(Fraction const &f2) const {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator*(Fraction const &f2) const {
        int n = numerator * f2.numerator;
        int d = denominator * f2.denominator;
        Fraction fNew(n, d);
```

```
                fNew.simplify();
                return fNew;
        }

        bool operator==(Fraction const &f2) const {
                return (numerator == f2.numerator && denominator ==
f2.denominator);
        }

        void multiply(Fraction const &f2) {
                numerator = numerator * f2.numerator;
                denominator = denominator * f2.denominator;
                simplify();
        }

        // Pre-increment
        Fraction& operator++() {
                numerator = numerator + denominator;
                simplify();

                return *this;
        }
        // Post-increment
        Fraction operator++(int) {
                Fraction fNew(numerator, denominator);
                numerator = numerator + denominator;
                simplify();
                fNew.simplify();
                return fNew;
        }
        // Short-hand addition operator overloaded
        Fraction& operator+=(Fraction const &f2) {
                int lcm = denominator * f2.denominator;
                int x = lcm / denominator;
                int y = lcm / f2.denominator;

                int num = x * numerator + (y * f2.numerator);

                numerator = num;
                denominator = lcm;
                simplify();

                return *this;
        }
};
```

Dynamic array class

Let us now implement a dynamic array class where we will be creating functions to add an element to the array, extract the array element using the provided index and print the complete array. Kindly refer to the code below:

```
class DynamicArray {
        int *data;
        int nextIndex;
        int capacity;                   // total size

        public :
        DynamicArray() {
                data = new int[5];          // Starting with capacity  = 5
                nextIndex = 0;
                capacity = 5;
        }

        DynamicArray(DynamicArray const &d) {
                //this -> data = d.data;                // Shallow copy

                // Deep copy
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }
        // operator overloading: We are simply equating two arrays
        void operator=(DynamicArray const &d) {
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }

        // inserting a new element to the array
        void add(int element) {
                if(nextIndex == capacity) {         // If the capacity is not enough
                        int *newData = new int[2 * capacity];   // We doubled the
                                                                // capacity of the array
```

```
                for(int i = 0; i < capacity; i++) {
                        newData[i] = data[i];    // Elements copied to the array
                }
                delete [] data;
                data = newData;
                capacity *= 2;
        }
        data[nextIndex] = element;
        nextIndex++;
    }

    int get(int i) const {     // For returning the element at particular index
        if(i < nextIndex) {
            return data[i];
        }
        else {
            return -1;
        }
    }

    void add(int i, int element) {
            if(i < nextIndex) {
                data[i] = element;
            }
            else if(i == nextIndex) {
                add(element);
            }
            else {
                return;
            }
    }

    void print() const {                    // For printing the complete array
        for(int i = 0; i < nextIndex; i++) {
            cout << data[i] << " ";
        }
        cout << endl;
    }
};
```