Programming Paradigm

Object Oriented Paradigm : Class and Object using C++

Structure

In “C”; it is possible put related items together, even if they are heterogeneous, in a structure.

Structures thus encapsulate related data, but it does not provide a mechanism by which we can also specify how the data can be acted upon

```
struct Person {
    char name[20];
    int age;
}
void Read(Person &x){
    cout << "enter name and age
:";
    cin >> z.name >> z.age;
}
```

```
void Write(const Person & y) const
{
    cout << "Name : "<< y.name << "
age : " << y.age << endl;
}
int main() {
    Person p;
    Read(p);
    p.age = -10; // it is possible
    Write(p)
    return 0;
}
```

Structure

Contd...

In "C++"; it is possible put not only data, but also operation with in a structure. It is new concept in C++

But in structure, by default all members are **Public** and therefore can be access by client program.

```
struct Person {
    char name[20];
    int age;
    void Read();
    void Write() const;
}
void Person::Read(){
    cout << "enter name and
age :";
    cin >> name >> age;
}
```

```
void Person::Write() const {
    cout << "Name : "<< name << " age
: " << age << endl;
}
int main() {
    struct Person p;
    p.Read();
    p.age = -10; // Still it is
possible
    p.Write()
    return 0;
}
```

Class

The extended feature of structure in C++, we would prefer to call as Class where we can have the **Access Specifier**.

Note that in Class, by default all members are **Private**

```
class Person {
    private : // Even if not specified
        char name[20];
        int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(){...}
void Person::Write() const {...}
```

```
int main() {
    Person p;  // p is an object of the class Person
    p.Read(); // p implicitly passed by reference

    p.age = -10; // It is NOT possible

    p.Write() // p is passed as a constant object
    return 0;
}
```

Class

Few points to note

- Encapsulates data and functions
- Same as structure, but all the members are private by default
- An object is an instance of the Class
- Size of the object depends only on the data members of the Class ad their layout and does not depends on the member functions
- Invoking member functions of the class is resolved at compile time and therefore no extra overhead at run time

Access Specifier

Private : default specifier for the class, can be access only by member function of the class or friend function (*This point will be revisited while discussing friend function*).

Public : can be access from the client function also

Protected : can be access from member function of the same class or any of it's publicly derived class – can not be access by the client function (*This point will be revisited while discussing inheritance*).

Static Members

A Class may have static data members and static member function

- A static data member belongs to the class and not to each object – any information related to the class can be maintained in the static data member
- All the objects of the same class will share the same static data member
- A static member is declared in the class definition like any other non-static data member, but it has to be defined in one of the implementation files explicitly
- A static member function can only access the static data members
- A non-static member function can also access the static data members of the class

Static Members

Example

```
class Person {
    private :
        char * name;
        int age;
        static int personCount;
    public :
        Person() { personCount++ } // Constructor accesses the static
data member
        ...
        ~Person() { personCount-- } // Destructor accesses the static
data member
        static void DisplayNumOfPerson();
};

int  Person::personCount = 0; // definition of the static data member
```

Static Members

Example contd...

```
// definition of the static member function
void Person::DisplayNumOfPerson() {
    cout << "Number of Person : " << personCount << endl;
}
int main() {
    Person::DisplayNumOfPerson(); // displays 0
    Person a;
    a.DisplayNumOfPerson(); // displays 1
    {
        Person b;
        b.DisplayNumOfPerson(); // displays 2
        a.DisplayNumOfPerson(); // displays 2
    }
    Person::DisplayNumOfPerson(); // displays 1
}
```

'This' pointer

- The keyword this identifies a special type of pointer.
- If an Object of a Class is created that has a non-static member function -
  - When the non-static member function is called – the keyword ***this*** in the function body stores the address of the Object (acts as a stack variable).
  - When a non-static member function is called, the ***this*** pointer is passed as an extra argument (hidden).
- Its not possible to declare the this pointer or make assignments to it
- A static member function does not have a this pointer

'This' pointer

Example

```
class Person {
    private :
        char name[20];  int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(Person * this){
    cout << "enter name and age :";
    cin >> this->name >> this->age;
}
void Person::Write(const Person * this) const {
    cout << "Name : "<< (*this).name << endl;
    cout << " age : " << (*this).age << endl;
}
```

Default member functions of a Class

- What are the member-functions a class has by default ?
- By default, if not implemented by the user, the compiler add some member functions to the class. Those are below four -
  - Default Constructor
  - Destructor
  - Copy constructor
  - Assignment operator

Note : This is true with some exceptions

Constructor

An object of the class can be initialize by special function called the constructor.

- Is a member function which is used to initialize the object
- Has the same name as the class
- Is invoked when an object is created
  - Is NOT invoked when a pointer of an Class is defined
  - Is invoked when an object is created dynamically
  - Is invoked as many times as the member of the elements in an array of objects
- Has no return type
- Can have parameter and default parameter
- Can be overloaded
- A constructor which takes no arguments is called Default Constructor
- A constructor can be private (will be discussed later)

Destructor

Any clean-up operation of an object can be done by another special function called the destructor.

- Is a member function which is used to release resources
- Has the same as the class, preceded by ~
- Is invoked when an object is removed
  - Not invoked when a pointer of a Class goes out of scope
  - Invoked when a dynamically allocated object is deleted
  - Invoked as many time as the number of elements in an array when array goes out of scope
- Has no return type
- Can have no parameters
- Can not be overloaded

Constructor and Destructor

Example

```
class Person {
    private :
        char name[20];
        int age;
    public :
        Person(); // default
        Person(char *, int=20);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(){
    name[0] = '\0'; age = 0;
}
```

```
Person::Person(char *s, int n){
    strcpy(name, s); age = n;
}
Person::~Person(){
    cout << "destructor called" <<
endl;
}
int main() {
    Person p;
    Person q("Raman", 53);
    Person * ptr = Null;
    ptr = new Person("Sundar");
    delete ptr;
}
```

Initialization List

- Initialization lists are used to initialize the data members of the class
- They can be used only in the constructor and no other member function
- Initialization list is executed before the body of the constructor is executed

Example code :

```
Person::Person() : age(0) {
    name[0] = '\0';
}
Person::Person(char *s, int n) : age(n) {
    strcpy(name, s);
}
```

Initialization List

Few points

- Used in constructor to initialize data members of the class
- Invoked in the order of declaration in the class and not in the order of the occurrence in the initialization list
- Efficient compared to making assignment
- Necessary to initialize constant or reference data member of the class

Some more points will be discussed while discussing inheritance

Dynamic Memory allocation using Constructors and Destructors

```
class Person {
    private :
        char * name; // a pointer
        int age;
    public :
        Person(); //default
        Person(char *, int);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(): name(NULL),
age(0) { }
```

```
Person::Person(char *s, int
n):age(n){
    name = new char[strlen(s)+1];
    strcpy(name, s);
}

Person::~Person(){
    delete []name;
}

int main() {
    Person p;
    Person q("Raman", 53);
}
```

Copy Constructor

There are instances in the code where an object is initialize with an existing object.

In such case, a special constructor called the copy constructor gets invoked.

```
int main() {
    Person p("raman", 53);
    p.write();
    Person q(p);
    // q is a new object instantiated by an existing object p,
  equivalent to "Person q = p;"
    q.write();
}
```

A default copy constructor provided by the compiler that does a member wise copy (shallow copy). This will not work if the object has a pointer.

Copy Constructor

Example

In that case programmer is required to provide an implementation of copy constructor if the object has some resource (for example : pointer data member having dynamically allocated memory) – provide deep copy instead of shallow copy.

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &); // object is passed by
  reference
        ~Person();
        void Read();
        void Write() const;
}
```

Copy Constructor

Example Contd...

```
Person::Person(const Person & rhs): age(rhs.age){
    name = new char[strlen(rhs.name)+1]),
    strcpy(name, rhs.name);
}

int main() {
    Person p("raman", 53);
    p.write();

    Person q(p); // q is a new object instantiated by an existing
object p using copy constructor
    q.write();

    Person r = p; // r is a new object instantiated by an existing
object p using copy constructor
    r.write();

}
```

Assignment Operator

There are instances in the code where an object is initialize with an existing object using assignment operator.

```
int main() {
    Person p("raman", 53);
    p.write();
    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
  assignment operator
    q.write();
}
```

A default implementation of assignment operator is provided by the compiler that does a member wise copy (shallow copy). This will not work if the object has a pointer.

Assignment Operator

Example

In that case programmer is required to provide an implementation of assignment operator if the object has some resource (for example : pointer data member having dynamically allocated memory) – provide deep copy instead of shallow copy.

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &);
        Person & operator=(const Person &);
        ~Person();
        void Read();
        void Write() const;
};
```

Assignment Operator

Example Contd...

```
Person & Person::operator=(const Person &rhs) {
    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
int main() {
    Person p("raman", 53);
    p.write();

    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
assignment operator
    q.write();

    p = p; // Also valid operation, but has side effect
    Person r("ravi", 33);
    r = p; // Also valid operation, but has side effect
}
```

Assignment Operator

Enhanced code

```
int main() {
    Person p("raman", 53);
    p = p; // This is valid operation, but may cause memory leak

    Person r("ravi", 33);
    r = p; // This is valid operation, but may cause memory leak
}

Person & Person::operator=(const Person &rhs) {
    if(this == &rhs) // self checking
        return *this;
    delete []name; // remove whatever existed before

    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
```

Automatic type conversion

- In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.
- In C++, this same effect for user-defined types can be achieved by defining automatic type conversion functions.

These functions come in two flavours:

1. a particular type of constructor
2. an overloaded **operator**.

Constructor conversion

If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.

```
class One {
  public:
    One() {...}
};
class Two {
  public:
    Two(const One&) {...}
};
void f(Two) {...}
int main() {
  One one;
  f(one); // Wants a Two, has a One
}
```

Preventing constructor conversion: Using keyword explicit

```
class Two {
  public:
    explicit Two(const One&) {...}
};

int main() {
  One one;
  //! f(one); // No auto conversion
allowed
  f(Two(one)); // OK -- user
performs conversion
}
```

Operator conversion

Create a member function that takes the current type and converts it to the desired type using the **operator** keyword followed by the type you want to convert to.

This form of operator overloading is unique because you don't appear to specify a return type – the return type is the *name* of the operator you're overloading

```
class Three {
  int elem;
  public:
    Three(int i = 0, int j = 10) : elem(i) {}
};
class Four {
  int num;
  public:
    Four(int x) : num(x) {}
    operator Three() const {
        return Three(num); }
};
```

```
void g(Three) {...}
int main() {
  Four four(1);
  g(four);
  g(1); // Calls
Three(1,10)
}
```

Friend Function

There are instance where a function might be required to access the data members of a class even though they are private and this function can not be made member of a class.

In such case, we declare that this function is a **friend** of the class. A friend function can access the private members of the class to which it is a friend.

Note that a friend function can be member function of one class also. As well it can be friend to entire class also.

Friend Function

Example code

Example code :

```
class Person {
    private :
    char * name; int age;
    public :
    ...
    friend void doctor(Person &);
    // friend declaration
generally  placed in public
section
};
```

```
void doctor(Person & p) {
    cout << "Administrating age
reduction tonic" << endl;
    p.age = p.age - 10;
    // accessing private member
}

int main() {
    Person p("suparman", 50);
    p.write();
    doctor(p);
    p.write();
    return 0;
}
```

Friend Class

There are instances where a class might require to use another class – the former class may want to access private members of the latter class. The former class is made friend to the latter class.

Example code : Please complete by your own

```
class Node {
    private :
        int info;
        Node* link;
        Node(int) // private constructor
        friend class Queue;
};
```

```
class Queue {
    public :
        Queue();
        ~Queue();
        void add(int); // Add at rear
        void remove(); // Remove from front
        bool isEmpty();

    private :
        Node * f, *r;
        // front and rear pointers
};
```

Operator Overloading

Two ways to overload an operator. Using -

1. global overloaded operators
2. member overloaded operators

- One of the most convenient reasons to use global overloaded operators instead of member operators is that in the global versions, automatic type conversion may be applied to either operand
- Whereas with member objects, the left-hand operand must already be the proper type.

Reflexivity of operators

```
class Number {
  int num;
  public:
    Number(int i = 0) : num(i) {}
    const Number
      operator+(const Number&) const;
    friend const Number
      operator-(const Number&, const Number&);
};
const Number
Number::operator+(const Number& n) const {
    return Number(num + n.num);
}
const Number
operator-(const Number& n1, const Number& n2){
   return Number(n1.num - n2.num);
}
```

```
int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 10; // 2nd arg
converted to Number
  //! 10 + a; // Wrong!
1st arg not of type
Number
  a - b; // OK
  a - 10; // 2nd arg
converted to Number
  10 - a; // 1st arg
converted to Number
}
```

Operator Overloading

There are several other aspects of operator overloading.
Those are for your detail study !!

Class Template

There are many data structures which have certain properties independent of the components they contain. For example :

- Queue : It has property of FIFO, doesn't not matter what type of component we put in the queue
- Stack : It has property of LIFO, doesn't not matter what type of component we put in the stack

- In c++ we can write such generic data structure using template class concept.
- In this approach, the class as well as the member functions are generated at compile time based on the definition of the object in the client code.

Like -

```
MyStack<int> s1(10); // stack of 10 integer
MyStack<double> s2(5); // stack of 10 integer
```

To carry out this process, the code has to be exposed to the client.

Class template

Example

```
template <class T> // can use the keyword typename also
class MyStack {
    public :
        MyStack(int m = 10);
        void Push(T x);
        T Pop();
        ...
    Private :
        T * elementList;
        ...
};
int main() {
    MyStack<int> S1(10);     S1.push(5);
    MyStack<double> S1(5);   S2.push(9.33);
}
```