Dynamic constructors

Dynamic constructors

- Constructors too, can allot memories to objects at run time and prevent unnecessary memory wastage.

- A constructor allocating dynamic memory using new operator is called as ***dynamic constructors.***

Example:

```
class DynamicCon
{
        int * ptr;
        public:
        DynamicCon()
        {
          ptr=new int;
         *ptr=100;
        }
        DynamicCon(int v)
        {
          ptr=new int;
         *ptr=v;
        }

         int display()
        {
          return(*ptr);
        }

};

void main()
{

         DynamicCon obj1, obj2(90);
         cout<<"\nThe value of obj1's ptr  is:";
         cout<<obj1.display();
         cout<<"\nThe value of obj2's ptr is:"<<endl;
         cout<<obj2.display();
}
```

The value of obj1's ptr is:100
The value of obj2's ptr is: 90

Friend Functions

What is the use of the functions

- access the data in c++.
- the class private data members are accessible only with the member functions of the same class.
- Which is called data hiding (outers are not allowed)
- Suppose several classes want the same members then we need to create several member functions (for each class one member function so program size is increased)
- How to access the private data members outside class members (non member function). And instead of creating multiple member functions only one function is sufficient to access the private data members.
- Friend function
- The same way we go for operator overloading

Friend Functions

- A friend function is not a member of a class but still has access to its private elements.
- A friend function can be
  - A global function not related to any particular class
  - A member function of another class
- Inside the class declaration for which it will be a friend, its prototype is included, prefaced with the keyword friend.

- Why friend functions ?
  - Operator overloading
  - Certain types of I/O operations
  - Permitting one function to have access to the private members of two or more different classes

- Unlike member functions, it cannot access the member names directly and has to use an object name and dot operator with each member name.
- It can be declared either in the public or private sections
- Usually, it has the objects as arguments.

```
class Circle
{
    private:
                        float radius;
    public:
                        Circle() { radius = 0.0;}
                        Circle(int r);
                        void setRadius(float r){radius = r;}
                        float getDiameter(){ return radius *2;}
                        float getArea();
                        float getCircumference();
};
```

- Friend functions need to access the members (private, public or protected) of a class through an object of that class.

- The object can be declared within or passed to the friend function.

Declaration of friend function

```
class class_name {
                    ... .. ...
                    friend return_type function_name(argument/s);

                    ... .. ...
                }
```

```
return_type functionName(argument/s)
{
        ... .. ...
// Private and protected data of className can be accessed from //
this function because it is a friend function of className.
        ... .. ...
}
```

```
class MyClass
{
  int a; // private member
public:
  MyClass(int a1) {
    a = a1;
  }
  friend void ff1(MyClass obj);
};
```

```
// friend keyword not used
void ff1(MyClass obj)
{
  cout << obj.a << endl;
   // can access private member 'a'
   directly
  MyClass obj2(100);
  cout << obj2.a << endl;
}
void main()
 {
  MyClass o1(10);
  ff1(o1);
Cout<<o1.a; //error
}
```

- A friend function is not a member of the class for which it is a friend.

obj.ff1(obj2);

- // wrong, compiler error

Using a friend function with two classes

```
class YourClass; // a forward
    declaration
class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  friend int compare (MyClass obj1,
    YourClass obj2);
};
class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
  friend int compare (MyClass obj1,
      YourClass obj2);
};
void main() {
  MyClass o1(10); YourClass o2(5);
  int n = compare(o1, o2); // n = 5
}

int compare (MyClass obj1, YourClass
      obj2) {
  return (obj1.a – obj2.a);
}
```

- Member functions of one class can be friend functions of another class

```
class YourClass; // a forward
    declaration
class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  int compare (YourClass obj) {
    return (a – obj.a)
  }
};
```

```
class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
  friend int MyClass::compare
    (YourClass obj);
};
void main() {
  MyClass o1(10); Yourclass o2(5);
  int n = o1.compare(o2);
}
```

Friend Class

All the member functions of one class as the friend functions of another class. the class is called a friend class.

```
class Z
{
.........
friend class X;   // all member functions
                  // of X are friends to Z
};
```

```
class Node
{
private:
  int key;
  Node *next;
  /* Other members of Node Class */

  friend class LinkedList; // Now class  LinkedList can
                           // access private members of Node
};
```

```
... .. ...
class B;
class A
{

// class B is a friend class of class A
friend class B;
            ... .. ...
 }

class B
 {
            ... .. ...
}
```

```
class alpha
{
private:
        int data;
public:
        alpha() { data = 99; }
friend class beta;
//beta is a friend class
};
class beta
{
public:
void func(alpha a)
{
cout << "\ndata=" << a.data;
}
};
```

```
void main()
{
alpha a;
beta b;
b.func(a);
cout<< endl;
}
```

Rules for using Default Arguments

- When we mention a default value for a parameter while declaring the function, it is said to be as default argument.

- Only the last argument must be given default value. You cannot have a default argument followed by non-default argument.

```
sum (int x,int y);
sum (int x,int y=0);
sum (int x=0,int y); // This is Incorrect
```

- If you default an argument, then you will have to default all the subsequent arguments after that.

```
sum (int x,int y=0);
sum (int x,int y=0,int z); // This is incorrect sum
(int x,int y=10,int z=10); // Correct
```

- You can give any value a default value to argument, compatible with its datatype.

Function overloading

- There are two types of polymorphism in C++.
- Compile time polymorphism and runtime polymorphism.
- Function overloading comes under the Compile time polymorphism.
- Function overloading is a process in which functions have same name but either the number of arguments or the data type of arguments has to be different.

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

- It does not depend upon the return type of functions because function will return a value when it is called.

```
int test(int a) { }
double test(int b){ }
```

Example:

```
class FuncOverloading
{
public:
        void sum(int x,int y)
        {
                cout<<"\n Sum of two integers ::"<<x+y;
        }
        void sum(double x,double y,double z)
        {
                 cout<<"\n Sum of three double variable ::"<<x+y+z<<endl;
        }
};
void main()

{
        FuncOverloading obj;
        obj.sum(10,20);
        obj.sum(10.1,20.2,30.3);

}
```

Output:

Sum of two integers :: 30
Sum of three double variable :: 60.6

Operator Overloading

User defined data types and pre defined data types

In “normal” overload we can write multiple functions and C++ invokes the one that matches the parameter list. We have been doing this for constructors.

we can make operators to work for user defined classes.

```
return_type classname :: operator op(arglist)
{
Function body // task defined
}
```

return_type: is the type of value returned by the specified operation
Op : is the operator being overloaded
“operator op ” : is the function name

- Is a process using operator instead of function to do a particular concept
- Existing operators only overload new one cant do
- It never overrides existing behavior of the operator (we cant change basic meaning of the operator)
- In general operator overloading we pass the objects as the arguments
- At least one user defined data type use in operator overloading (left side operand should be user defined data type)

We can't overload ::, (scope) ., .* (membership) operators

sizeof, conditional operator

Operator member function should be the member of the same class

Explicitly send only one object as argument because implicitly using another object to call the operator overloading (for binary operator overloading case)

In case of unary explicitly object is not required implicit object is sufficient

Overloading Unary Operators

```
class space
{
private:
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display()
{ cout << x <<" "<< y <<" "<< z <<"\n";
}
void operator++(); //overload increment
void operator-(); //overload unary minus
};
void space :: getdata(int a, int b, int c)
{
x = a;
y = b;
z = c;
}
```

```
void space :: operator++()
{
x++;
y++;
z++;
}
void space :: operator-()
{
x = -x;
y = -y;
z = -z;
}
void main()
{
space S;
S.getdata(10,-20, 30);
cout<<"S : ";
S.display();
++S; // activates operator++() function
cout<<"S : ";
S.display();
-S; // activates operator-() function
cout<<"S : ";
S.display();
}
```

```
S : 10 -20 30
S : 11 -19 31
S : -11 19 -31
```

S2 = ++S1;
will not work because the function operator++() does not return any value.

It can work if the function is modified to return an object

Overloading Binary Operators

Binary operators act on two operands. Examples include +, -, *, and /.

```
class complex
{
private:
float x;
float y;
public:
complex(){ }
complex(float real, float imag)
{ x = real; y = imag; }
complex operator+(complex);
void display();
};
complex complex :: operator+(complex c)
{
complex temp;
temp.x = x + c.x;
temp.y = y + c.y;
return temp;
}
```

```
void complex :: display()
{
cout<< x << " + j"<< y << "\n";
}
void main()
{
complex C1(2.5, 3.5), C2(1.6, 2.7), C3;
C3 = C1 + C2;
cout<<"C1 = ";
C1.display();
cout<<"C2 = ";
C2.display();
cout<<"C3 = ";
C3.display();
}
```

C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2

Another way of calling binary operator overloading function is to call like a normal member function as follows,

C3 = C1.operator+ (C2);

Operator Overloading using Friend

- We can overload the operator friend keyword.
- Friend function not a member of that class. So we cant call with obj.
- Normally member functions are called with obj. member function
- Friend function declaration is inside the class and definition is outside the class.
- But in operator overloading we can define inside or outside of the class also.
- Another one is friend functions have objects as arguments because they are not member function so they not directly access the members. Then we can use obj. members to access the data.
- In operator overloading with friend for unary case one explicit argument (object) and in binary case two explicit arguments need to provide.

```
#include<iostream.h>
Class test2;
Class test1
{
Int a;
Public:
Void geta()
{
Cout<< "enter a value";
Cin>>a;
friend void operator > (test1, test2);
}
};
```

```
Class test1
{
Int b;
Public:
Void getb()
{
Cout<< "enter b value";
Cin>>b;
}
friend void operator > (test1, test2);
};
```

```
void > (test t1, test t2)
{

  t1.a > t2.b ? Cout<<" a is big" : "b is big"

}
```

```
Void main()
{
  test1 t1;
  test2 t2;
  t1.geta();
  t2.getb();
  t1>t2;
}
```

Overloading and Ambiguity

The compiler could not choose a function between two or more overloaded functions, the situation is called as an **ambiguity** in function overloading.

When the program contains ambiguity, the compiler will not compile the program.

The main cause of ambiguity in the program

- Type conversion
- Functions with default arguments
- Functions with pass by reference

```
#include <iostream>
using namespace std;
#include <conio.h>

void fun(int i)
{
   cout << i;
}

void fun(float f)
{
   cout << f;
}

int main()
{
    fun(10);
    fun(10.2);
    getch();
    return 0;
 }
```

call of overloaded 'fun(double)' is ambiguous.

Because 10.2 will be treated as double data type by the compiler.

(In c++ all the floating point values are converted by the compiler as double data type.) This is type conversion of float to double.

Functions with default arguments

```
#include <iostream>
using namespace std;
#include <conio.h>

void fun(int i)
{
  cout << i;
}

void fun(int i, int j=8)
{
 cout << i << j;
}


int main()
{
    fun(10);
    fun(10.2f);
    getch();
    return 0;
}
```

They are **(i)** function with one argument. fun(2) now the value of i is 2 and j is 8 (8 is the default value).
(ii) function with two arguments fun(5,6). So i is 5 and j is 6.

fun(int i) is a function with one argument. It should be invoked with one argument.

when we call a function with 1 argument the compiler could not select among the two functions fun(int i) and fun(int i, int j=8).

Function with reference parameter

```
#include <iostream>
using namespace std;
#include <conio.h>

void fun(int i)
{ cout << i;}

void fun(int &i)
{cout <<i;}

int main()
{
   fun(10);
   getch();
   return 0;
}
```

error message **call of overloaded ‘fun(int)’ is ambiguous**.

compilers do not know which function is intended by the user when it is called. Because there is no syntactical difference in calling the function between the function with single argument and function with single reference parameter.

Pointers to Objects

- when we are writing the program we don't know how many objects we want to create.
- we can use new operator to create objects at run time.
- The new operator returns a pointer to an unnamed object.
- object pointers are useful in creating objects at run time.
- can also use an object pointer to access the public members of an object.

```
class item
{
private:
int code;
float price;
public:
void getdata(int a, float b)
{
code = a;
price = b;
}
void show()
{
cout<< "Code : " << code << "\n";
cout<< "Price : " << price <<"\n";
}
};
```

```
void main()
{
int x;
float y;
item *p = new item;
cout<< "Input code and price for item: ";
cin >> x >> y;
p->getdata(x,y); // or (*p).getdata(x,y);
p->show();  // or (*p).show();
}
```

This Pointer

- Each object gets its own copy of data members and all objects share single copy of member functions.
- If only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
- Compiler supplies an implicit pointer along with the functions names as ‘this’.
- The ‘this’ pointer is passed as a hidden argument to all non static member function calls
- this’ pointer is a constant pointer that holds the memory address of the current object.
- this’ pointer is not available in static member functions as static member functions can be called without any object (with class

Friend functions do not have a **this** pointer, because friends are not members of a class.

this pointer is used to represent the address of an object inside a member function

```
class ClassName
{
  private:   int dataMember;
  public:   method(int a)
{ // this pointer stores the address of object obj and access dataMember
 this->dataMember = a;
... .. ...
 }
}
int main()
 {
ClassName obj;
 obj.method(5);
  ... .. ...
}
```

Applications of this pointer

Return Object

```
return *this;
```

Method Chaining

```
positionObj->setX(15)->setY(15)->setZ(15);
```

each method return *this pointer.
This is equivalent to

```
positionObj->setX(15);
 positionObj->setY(15);
positionObj->setZ(15);
```

When local variable's name is same as member's name

```
class Test
{
private:
int x;
public:
void setX (int x)
{
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
}
void print() { cout << "x = " << x << endl; }
};

int main()
{
Test obj;
int x = 20;
obj.setX(x);
obj.print();
return 0;
}
```

Distinguish Data Members

this pointer to distinguish local members from parameters.

```
class sample
 {
       int a,b;
       public:
        void input(int a,int b)
        {
              this-> a=a+b;
              this->b=a-b;
       }
       void output()
       {
             cout<<"a = "<<a<<endl<<"b = "<<b;
        }
};
int main()
 {
       sample x;
        x.input(5,8);
        x.output();
        return 0;
        }
```

To return reference to the calling object

```
class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
Test obj1(5, 5);

// Chained function calls. All calls modify the same object
// as the same object is returned by reference
obj1.setX(10).setY(20);

obj1.print();
return 0;
}
```

```
x = 10 y = 20
```

```
Class Demo{
        private: int num; char ch;
        public:
        Demo &setNum(int num)
        {
            this->num =num;
            return *this;
        }
        Demo &setCh(char ch)
        {
                this->num++; this->ch =ch; return *this;
        }
        void displayMyValues()
        {
            cout<<num<<endl;
            cout<<ch;
        }
};
 int main()
{
    Demo obj; //Chaining calls
    obj.setNum(100).setCh('A');
    obj.displayMyValues();
    return 0;
}
```

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

class Test
{
private:
int x;
public:
Test(int x = 0) { this->x = x; }
void change(Test *t) { this = t; }
void print() { cout << "x = " << x << endl; }
};

int main()
{
Test obj(5);
Test *ptr = new Test (10);
obj.change(ptr);
obj.print();
return 0;
}
```

```
class Test
{
private:
	int x;
	int y;
public:
Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
static void fun1()
{
		cout << "Inside fun1()";
}
static void fun2()
{
		cout << "Inside fun2()"; this->fun1();
 }
};

int main()
{
	Test obj;
	obj.fun2();
	return 0;
}
```

1. Write a program to swap the values two integer members of different classes using friend function .

2. Write a program for addition of two complex numbers using friend function (use constructor function to initialize data members of complex class).

3. Define a class string and overload == to compare two strings and + operator for concatenation of two strings.

4. Write a program to perform matrix addition using operator overloading concept.

Matrix

a[100][100], m,n

void getdata()
void show()
matrix operator+(matrix &x,matrix &y)

5. Write a program to maintain the records of person with details (name and age) and find the eldest among them. The program must use this pointer to return the result.

How to pass an object within the class member function as an argument

1) Call by value
2) Call by reference

With Return object from a function

Inline Functions

Static data members

Write a C++ program to count the number of persons inside a bank, by increasing count whenever a person enters a bank, using an increment(++) operator overloading function, and decrease the count whenever a person leaves the bank using a decrement(--) operator overloading function inside a class

Write a program to accept the student detail such as name and 3 different marks by get_data() method and display the name and average of marks using display() method. Define a friend class for calculating the average of marks using the method marrk_avg().

C++ program to display the record of student with highest percentage.

```
class student
{
 char name[100];
 int age,roll;
float percent;
public:
void getdata()
 {
cout<<"Enter data"<<endl;
cout<<"Name:"; cin>>name;
cout<<"Age:"; cin>>age;
cout<<"Roll:"; cin>>roll;
cout<<"Percent:"; cin>>percent;
cout<<endl;
}
```

```
void display()
{

cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Roll:"<<roll<<endl;
cout<<"Percent:"<<percent;
}
```

```
student & max(student &s1,student &s2)
 {
if(percent>s1.percent && percent>s2.percent)
 return *this;
else if(s1.percent>percent && s1.percent>s2.percent)
return s1;
else if(s2.percent>percent && s2.percent>s1.percent)
return s2;
}

};
int main()
{
student s,s1,s2,s3;
 s1.getdata();
 s2.getdata();
 s3.getdata();
s=s3.max(s1,s2);
cout<<"Student with highest percentage"<<endl;
s.display();
getch();
return 0;
}
```