//

operator overloading

operator overloading is closely related to function overloading operator can be overloaded so that they perform special operations on objects of the class

operators are overloaded by writing operator overload functions these functions can be made either member functions or friend functions of a class

operator overloading functions are composed of Keyword "operator" followed by operator being overloaded.

The syntax for operator overloading function when designed as a member function is as shown below

```
class name
{
  ⋮
ret-type  operator<symbol>(arg-list)
{
  ⋮
 }
};
```

The syntax for operator overloading fun overload when designed as a friend function is shown below

```
        class name
        {
          ≡
    friend ret. type operator<symbol>(arg list)
        };

    ret_type operator<symbol>(arg list)
        {
          ≡
        }
```

Scanned with CamScanner

Overloading UNARY MINUS

```
# include <iostream>
using namespace std;
Class space
{
  int x;
  int y;
  int z;
 public:
    void getdata(int a, int b, int c);
    Void display (void);
    Void operator-();
};
void space :: getdata(int a, int b, int c)
 {
    x=a;
    y=b;
    z=c;
 }

void space :: display (void)
{   cout << x << " ";
    Cout << y << " ";
    cout << z << "\n";
}
```

Scanned with CamScanner

```
void space :: operator -()
{
  x = -x;
  y = -y;
  z = -z;
}
int main()
{
    Space s;
    s.getdata (10, -20, 30);
    cout << "s : ";
    s.display();
    -s;
    cout << "s : ";
    s.display();
    return 0;
}
```

o/p

```
s : 10 -20 30
s : -10 20 -30
```

The
t

Scanned with CamScanner

NOTE

The function operator-() takes no argument then what does this operator function do? it changes the sign of data members of the object s. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Creating prefix and post-fix forms of the Increment and decrement operators

-> In the preceding program only the prefix form of the increment operator was overloaded.

-> General forms for the prefix and postfix ++ and -- operator function

```
// prefix increment
  type operator ++ () {
        // body of prefix operator

  }
// post fix increment
  type operator ++ (int x) {
            // body of postfix operator }

  // prefix decrement
   type operator --() {
       }  // body of prefix operator
  // postfix decrement
     type operator -- (int x)
        { } // body of postfix operator
```

operator overloading Restrictions

Rules for operator overloading

→ only existing operators (such as +, *, /, + etc) can be overloaded

→ The new operators (such as ** indicating user defined exponentiation operator) can not be created.

→ The precedence of the operator can not be changed.

→ The associativity of the operator can not be changed

→ The following existing operators can not be overloaded.

. member selection

.* member selection through pointer.

:: scope resolution operator

?: condition operator (ternary) operator

→ The following operators can be overloaded only using non static member function

= assignment operator

() Function operator

[ ] Subscript operator

→ arrow operator

→ overloaded operators should not have default argument

```
Class x
{
public :
    void operator + (int = 0).

}
```

→ <u>what operator can be overload</u> ,

unary → ++, --, +, − etc

binary → +, −, /, *, <=, >= etc

new & delete → new, delete

Special → (), [ ] and →

Comma → ,

operator overloading using a Friend function

→ you can overload an operator for a class by using a nonmember function which is usually a friend of the class

→ since a friend function is not a member of the class it does not have a this pointer.

→ Therefore an overloaded friend operator function is passed the operands explicitly.

→ this means that a friend function that overloads a binary operator. has two parameters and a friend Function that overloads a unary operator has one parameter

→ when overloading a binary operator using a friend function the left operand is passed in the first parameter and the right operand is passed in the second parameter.

```
#include <iostream>
#include <conioh>
   class test2;
   class test1;
  {
     int a;
     public:
        void get a()
     {
      cout<<"enter a value";
       cin>>a;
     }
  friend void operator >(test1+test2),

    };

  class test2
  {
   int b;
   public:
  void get b;
  {
  cout<< enter b value;
   cin>>b;
  }
  friend void operators( test2 +testi
   };
```

```
void operator >(test1, test2)
{
t1.a>t2.b?->
 cout<<'a is big;
 Cout<<"b is big;
 }
 void main ()
  {
    test1 t1;
    test2 t2;
    clrscr()
    t1.get a();
    t2.get b();
     t1>t2;
      getch();
    }
```

In this porogram the operator + () function is made into a friend

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
 loc() { }
loc(int lg. Int lt)
{
 longitude = lg;
 latitude = lt;
}
void show ()
 {
 cout << longitude <<" ";
 Cout << latitude <<"\n";
 }
friend loc operator + (loc op1, loc op2);
 loc operator - (loc op2);
 loc operator= (loc op2);
 loc operator++ ();
 };
loc operator+ (loc op1 , loc op2)
 {
  loc temp;
temp.longitude=op1.longitude+op2.longi
temp.latitude=op1.latitude+op2.latitu
 return temp;
}
 loc loc::operator-(loc op2)
 {
  loc temp;
  temp.longitude=longitude-op2.longitude;
  temp.latitude=latitude-op2.latitude;
  return temp;
  }
loc loc::operator=(loc op2)
{
  longitude=op2.longitude;
  latitude=op2.latitude;
   return *this;
   }
   loc loc::operator++()
   {
      longitude++;
      latitude++;
      return *this;
    }
```

}
loc

```
    . 1. <iostream>
        int main()
{
loc ob1(10, 20). ob2(5. 30):
ob1: ob1+ ob2;
ob1. show();
 return 0:
  }
```

using a friend to overload ++ or --

refer page no 393 & 396
chapter 15
C++ complete Reference

Scanned with CamScanner

overloading new and delete

→ it is possible to overload new and delete

→ For Example you may want allocation routines that automatically begin using a disk file as Virtual memory when the heap has been Exhausted

→ Syntax.

```
            // Allocate an object
  void * operator new (size_t size)
 {
   /* perform allocation Throw bad-alloc on
      failure. constructor called automatically
    return pointer-to-memory ;
  }


      // Delete an object

  void operator delete (void *p)
 {
  /* Free memory pointed to by p
     Destructor called automatically. */
  }
```

```
#include <iostream>
# include<stdlib h>
using namespace std:
class student
{
  String name:
  int age:
  public:
   Student ()
 {
 cout<<" constructor is called\n":
 }
Student (string name, int age)
  {
   this -> name = name:
   this -> age = age:
  }
   void display()
     {
Cout<<" Name:" <<name << endl
cout <<" Age:" <<age <<endl:
}
void *operator new (size_t size)
{
cout <<"overloading new operator with size:"
        << size <<endl;
    void *p=::new student();
    return p;

}
void operator delete (void *p)
{
cout<<"overloading delete operator"<<endl;
  free (p);

}

};
   int main()
  {
student *p=new student ("yash", 24);
    p -> display ();
        delete p;

      }
```

Scanned with CamScanner

Technically the parameter does not have to be of type int, but an operator [ ]() function is typically used to provide array subscripting and as such, an integer value is generally used

refer page no. 408-409
Example program

<u>overloading ( )</u>

→ when you overload the () function call operator you are not per se creating a new way to call a function

→ Rather you are creating an operator function that can be passed an arbitrary number of parameters.

double operator () (int a, float f, char

and an object O of its class then the statement

O(10, 23.34, "hi");

translates into this call to the operator () function

O.operator () (10, 23.34, "hi");

Scanned with CamScanner

overloading new and delete

refer page no 399-407
C++ complete Reference

overloading some special operators

→ C++ defines array subscripting, function calling and class member access as operations. The operators that perform these functions are the [ ], ( ), and →, respectively

→ one important restriction applies to overloading these three operators.
They must be nonstatic member functions.
They cannot be friends

overloading [ ]

In C++ the [ ] is considered a binary operator when you are overloading it.
∴ the general form of a member operator [ ]( ) function is as shown here.

```
type class-name :: operator[](int i)
{
    //...
}
```

refer Example
page no 412 - 413

Overloading ->

The -> pointer operator also called the class member access operator, is considered a unary operator when overloading. its general usage is shown here:

```
object->element;
```

Eg: refer page 413-414

Constructor Overloading in C++

Prerequisites: Constructors in C++

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;
 class construct
{
 public:
    float area;
          // Constructor with no parameters
    construct()
    {
        area = 0;
    }
          // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};
 int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);
      o.disp();
    o2.disp();
    return 1;
}
```

Output: 0 200

new and delete operators in C++ for dynamic memory

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap**.
and non-static and local variables get memory allocated on stack

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is it different from memory allocated to normal variables?
For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated.

For dynamically allocated memory like " int *p = new int[10] ", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

How is memory allocated/deallocated in C++?

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
- Pointer - variable = **new** data-type;
  Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.
  Example:

```
// Pointer initialized with NULL

// Then request memory for the variable

int *p = NULL;

p = new int;
```

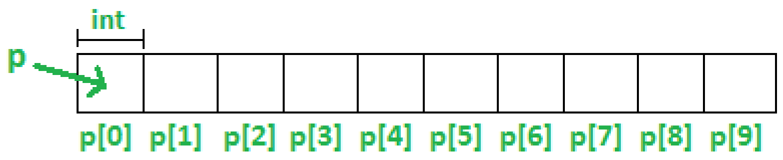
```
      OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

- **Initialize memory:** We can also initialize the memory using new operator:
- pointer-variable = **new** data-type(value);
- **Example:**
- int *p = new int(25);
- float *q = new float(75.25);
- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.
- pointer-variable = **new** data-type[size];

  where size(a variable) specifies the number of elements in an array.

```
Example:
     int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer (scroll to section “Exception handling of new operator” in this article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

delete operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
  delete p;
  delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;


Example:
   // It will free the entire array
   // pointed by p.
   delete[] p;
```

```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new(nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float *r = new float(75.25);

    cout << "Value of r: " << *r << endl;

    // Request block of memory of size n
    int n = 5;
    int *q = new(nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
```

Output:

```
Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5
```

Copy Constructor in C++

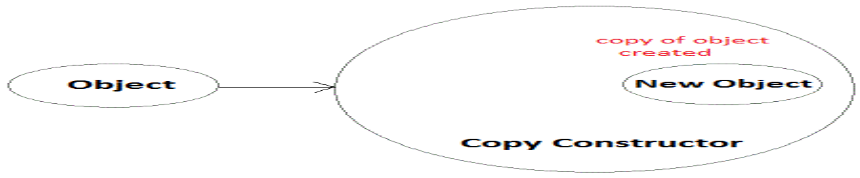
Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
    . . . .
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.

Copy constructors should not modify the object it is copying from which is why the **const** is preferred on the other parameter. Both will work, but the **const** is preferred because it clearly states that the object passed in should not be modified by the function. **const** is for the user only



Below is a sample program on Copy Constructor:

```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
    private:
    int x, y;   //data members

    public:
    Samplecopyconstructor(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    /* Copy constructor */
    Samplecopyconstructor (const Samplecopyconstructor &sam)
    {
        x = sam.x;
        y = sam.y;
    }

    void display()
    {
        cout<<x<<" "<<y<<endl;
    }
};
/* main function */
int main()
{
    Samplecopyconstructor obj1(10, 15);     // Normal constructor
    Samplecopyconstructor obj2 = obj1;      // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}
```

```
Normal constructor : 10 15
Copy constructor : 10 15
```

In the below example you can see both objects, c1 and c2, points to same memory location. When c1.concatenate() function is called, it affects c2 also. So both c1.display() and c2.display() will give same output.

```
#include<iostream>
#include<cstring>
using namespace std;
class CopyConstructor
{
    char *s_copy;
    public:
    CopyConstructor(const char *str)
    {
        s_copy = new char[16]; //Dynamic memory allocation
        strcpy(s_copy, str);
    }
    /* concatenate method */
    void concatenate(const char *str)
    {
        strcat(s_copy, str); //Concatenating two strings
    }
    /* copy constructor */
    ~CopyConstructor ()
    {
        delete [] s_copy;
    }
    void display()
    {
        cout<<s_copy<<endl;
    }
};
/* main function */
int main()
{
    CopyConstructor c1("Copy");
    CopyConstructor c2 = c1; //Copy constructor
    c1.display();
    c2.display();
    c1.concatenate("Constructor");    //c1 is invoking concatenate()
    c1.display();
    c2.display();
    return 0;
}
```

```
Copy
Copy
CopyConstructor
CopyConstructor
```

Why copy constructor argument should be const in C++?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.

One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that copy elision is not done by compiler.
When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization (sometimes referred to as RVO).

References:
http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html
http://en.wikipedia.org/wiki/Copy_constructor

When should we write our own copy constructor?

C++ compiler provide default copy constructor (and assignment operator) with class. When we don't provide implementation of copy constructor (and assignment operator) and tries to initialize object with already initialized object of same class then copy constructor gets called and copies members of class one by one in target object.

The problem with default copy constructor (and assignment operator) is – When we have members which dynamically gets initialized at run time, default copy constructor copies this members with address of dynamically allocated memory and not real copy of this memory. Now both the objects points to the same memory and changes in one reflects in another object, Further the main disastrous effect is, when we delete one of this object other object still points to same memory, which will be dangling pointer, and memory leak is also possible problem with this approach.

Hense, in such cases, we should always write our own copy constructor (and assignment operator).