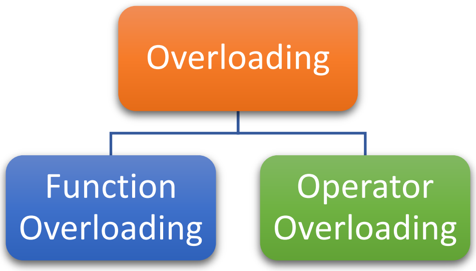
Overloading in C++

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.



An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

I. Function Overloading

> SAMPLE CODE

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

float float_max(float a, float b)
{
    return (a > b) a ? b;
}

int int_max(int a, int b)
{
    return (a > b) a ? b;
}

int main()
{
    cout << "int max = " << int_max(4, 5);  // Output: int max = 5
    cout << "int max = " << int_max(4.4, 5.5);  // Output: int max =  5
    cout << "float max = " << float_max(4.4, 5.5) << endl; // Output: float max = 5.5
    return 0;
}
```

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

> SAMPLE CODE

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

int max(int a, int b)
{
    return (a > b) a ? b;
}

int max(float a, float b)
{
    return (a > b) a ? b;
}

int main()
{
    cout << "int max = " << max(4, 5) << endl;          // Output: int max = 5
    cout << "float max = " << max(4.4, 5.5) << endl;    // Output: float max = 5.5
    return 0;
}
```

II. Operator Overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

> SAMPLE CODE

```
class Box {
    Box operator+(const Box&);

    Box operator+(const Box&, const Box&);

};
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below:

> SAMPLE CODE

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

> CONSOLE OUTPUT

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

III. Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded:

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

Following is the list of operators, which can not be overloaded:

::	.*	.	?:

IV. Operators Overloading Examples

1. Unary Operators Overloading

The unary operators operate on a single operand and following are the examples of Unary operators

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

> SAMPLE CODE

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

class Distance
{
private:
    int feet;   // 0 to infinite
    int inches; // 0 to 12

public:
    // required constructors
    Distance()
    {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i)
    {
        feet = f;
        inches = i;
    }

    // method to display distance
    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches << endl;
    }

    // overloaded minus (-) operator
    Distance operator-()
    {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
};

int main()
{
    Distance D1(11, 10), D2(-5, 11);

    -D1;                    // apply negation
    D1.displayDistance(); // display D1

    -D2;                    // apply negation
    D2.displayDistance(); // display D2

    return 0;
}
```

> CONSOLE OUTPUT

```
F: -11 I: -11
F: 5 I: -11
```

2. Binary Operators Overloading

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

> SAMPLE CODE

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

class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box

   public:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength( double len ) {
      length = len;
   }
   void setBreadth( double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

> CONSOLE OUTPUT

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

3. Relational Operators Overloading

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

> SAMPLE CODE

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }

      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }

   return 0;
}
```

> CONSOLE OUTPUT

```
D2 is less than D1
```

4. Input/Output Operators Overloading

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Following example explains how extraction operator >> and insertion operator <<.

> SAMPLE CODE

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

class Distance {
   private:
      int feet;              // 0 to infinite
      int inches;            // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output, const Distance &D ) {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }

      friend istream &operator>>( istream  &input, Distance &D ) {
         input >> D.feet >> D.inches;
         return input;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11), D3;

   cout << "Enter the value of object : " << endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;

   return 0;
}
```

```
> CONSOLE OUTPUT
```

```
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

5. Overloading Increment ++ & Decrement --

The increment (++) and decrement (--) operators are two important unary operators available in C++.

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

> SAMPLE CODE

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

class Time {
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59

   public:
      // required constructors
      Time() {
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m) {
         hours = h;
         minutes = m;
      }

      // method to display time
      void displayTime() {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }

      // overloaded prefix ++ operator
      Time operator++ () {
         ++minutes;          // increment this object
         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }

      // overloaded postfix ++ operator
      Time operator++( int ) {

         // save the orignal value
         Time T(hours, minutes);

         // increment this object
         ++minutes;

         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }

         // return old original value
         return T;
      }
};

int main() {
   Time T1(11, 59), T2(10,40);

   ++T1;                    // increment T1
   T1.displayTime();        // display T1
   ++T1;                    // increment T1 again
   T1.displayTime();        // display T1

   T2++;                    // increment T2
   T2.displayTime();        // display T2
   T2++;                    // increment T2 again
   T2.displayTime();        // display T2
   return 0;
}
```

> CONSOLE OUTPUT

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

6. Assignment Operators Overloading

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

The syntax and handling here is similar to the copy constructor.

> SAMPLE CODE

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};
int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

> CONSOLE OUTPUT

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

7. Function Call Operator () Overloading

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Following example explains how a function call operator () can be overloaded.

> SAMPLE CODE

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // overload function call
      Distance operator()(int a, int b, int c) {
         Distance D;

         // just put random calculation
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }
      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

> CONSOLE OUTPUT

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

8. Subscripting [] Operator Overloading

The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.

Following example explains how a subscript operator [] can be overloaded.

> SAMPLE CODE

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
   private:
      int arr[SIZE];

   public:
      safearay() {
         register int i;
         for(i = 0; i < SIZE; i++) {
           arr[i] = i;
         }
      }

      int &operator[](int i) {
         if( i > SIZE ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
         }

         return arr[i];
      }
};

int main() {
   safearay A;
   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;

   return 0;
}
```

> CONSOLE OUTPUT

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

9. Class Member Access Operator (->) Overloading

The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply.

The operator-> is used often in conjunction with the pointer-dereference operator * to implement "smart pointers." These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion either when the pointer is destroyed, or the pointer is used to point to another object.

The dereferencing operator-> can be defined as a unary postfix operator. That is, given a class:

> SAMPLE CODE

```
class Ptr {
   //...
   X * operator->();
};
```

Objects of class **Ptr** can be used to access members of class **X** in a very similar manner to the way pointers are used. For example:

> SAMPLE CODE

```
void f(Ptr p ) {
   p->m = 10 ; // (p.operator->())->m = 10
```

}

The statement p->m is interpreted as (p.operator->())->m. Using the same concept, following example explains how a class access operator -> can be overloaded.

> SAMPLE CODE

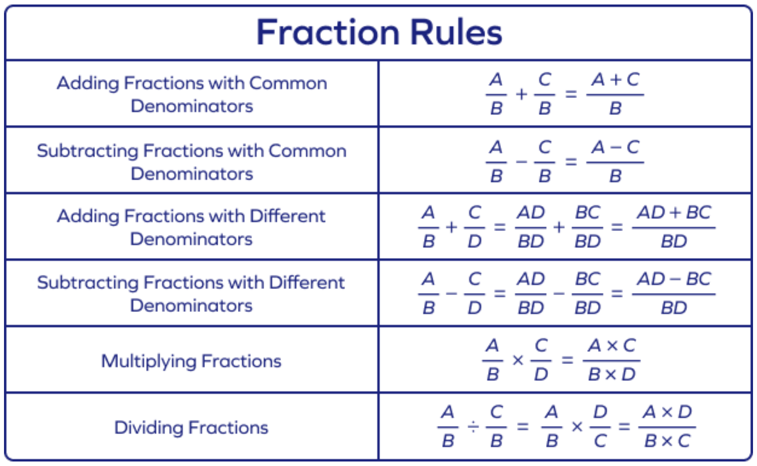
```
#include <iostream>
#include <vector>
using namespace std;

// Consider an actual class.
class Obj {
   static int i, j;

public:
   void f() const { cout << i++ << endl; }
   void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 10;
int Obj::j = 12;

// Implement a container for the above class
class ObjContainer {
   vector<Obj*> a;

   public:
      void add(Obj* obj) {
         a.push_back(obj);  // call vector's standard method.
      }
      friend class SmartPointer;
};

// implement smart pointer to access member of Obj class.
class SmartPointer {
   ObjContainer oc;
   int index;

   public:
      SmartPointer(ObjContainer& objc) {
         oc = objc;
         index = 0;
      }

      // Return value indicates end of list:
      bool operator++() { // Prefix version
         if(index >= oc.a.size()) return false;
         if(oc.a[++index] == 0) return false;
         return true;
      }

      bool operator++(int) { // Postfix version
         return operator++();
      }

      // overload operator->
      Obj* operator->() const {
         if(!oc.a[index]) {
            cout << "Zero value";
            return (Obj*)0;
         }

         return oc.a[index];
      }
};
int main() {
   const int sz = 10;
   Obj o[sz];
   ObjContainer oc;

   for(int i = 0; i < sz; i++) {
      oc.add(&o[i]);
   }

   SmartPointer sp(oc); // Create an iterator
   do {
      sp->f(); // smart pointer call
      sp->g();
   } while(sp++);

   return 0;
}
```

```
> CONSOLE OUTPUT
```

```
10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21
```

V. Exercises

1. Create class **Fraction** (分數) which contains two data members **num** as *numerator* (分子) and **den** as *denominator* (分母) following these requirements:
   a. Create 2 constructors with/without argument and with 2 parameters to assign *num* and *den*.
   b. Create **getters** and **setters** to assign values of *num* and *den*. Beware that denominator cannot equal to 0.
   c. Create **unary operator** overloading: - *fraction object* = - *num/den*
   d. Create **binary operators** (+,-,*,/) overloading by following rules:



Fraction Rules	
Adding Fractions with Common Denominators	$\frac{A}{B}+\frac{C}{B}=\frac{A+C}{B}$
Subtracting Fractions with Common Denominators	$\frac{A}{B}-\frac{C}{B}=\frac{A-C}{B}$
Adding Fractions with Different Denominators	$\frac{A}{B}+\frac{C}{D}=\frac{AD}{BD}+\frac{BC}{BD}=\frac{AD+BC}{BD}$
Subtracting Fractions with Different Denominators	$\frac{A}{B}-\frac{C}{D}=\frac{AD}{BD}-\frac{BC}{BD}=\frac{AD-BC}{BD}$
Multiplying Fractions	$\frac{A}{B}\times\frac{C}{D}=\frac{A\times C}{B\times D}$
Dividing Fractions	$\frac{A}{B}\div\frac{C}{B}=\frac{A}{B}\times\frac{D}{C}=\frac{A\times D}{B\times C}$

   e. Create **relational operator** overloading to compare two fractions: <, >, ==, <=, >=, !=
   f. Create **input/output operators** overloading (>>, <<) to prompt entering values *num* and *den* from console. User will input *num* and *den* values.
   g. Create ++ and -- operators overloading by adding or subtracting a number or a fraction to the current fraction.

E.g.
```
Fraction f1(1,2), f2(3,4);
f2 -= f1;    // f2 = f2 - f1;
f1 += 2;     // f1 = f1 + 2
++f1;        // f1 += 1
--f2:        // f2 -= 1
```

   h. Create **assignment operator** to assign a fraction to another one.
   i. Create a data function **simplify()** to simplify that fraction:

E.g.
```
Fraction f(6,12);
cout >> "f = " >> f.simplify();       // f = 1/2
```
Because the greast common divisor (最大公約數) of 6 and 12 is 6.

Students write the main function themselves to test the functions described above.