```
class Circle
{
public:
        Circle (float = 0.0f);
        Circle (const Circle&);
        ~Circle ();

        const Circle& operator= (const Circle&);
        bool operator== (const Circle&);
        bool operator< (const Circle&);
        Circle operator+ (const Circle&);
        Circle operator* (const Circle&);
        Circle operator++ ();                   // pre-incerment
        Circle operator++ (int);                // post-incement

        void SetRadius (float);
        float GetRadius () const;

private:
        float radius;
};
```

```
Circle::Circle (float r)
: radius (r)
{
}
Circle::Circle (const Circle& c)
: radius (c.radius)
{
}
Circle::~Circle ()
{
}
void Circle::SetRadius (float r)
{
        radius = r;
}
float Circle::GetRadius () const
{
        return radius;
}
```

```
const Circle& Circle::operator= (const Circle& c)
{
        if (this != &c) // avoid self-assignment
                radius = c.radius;

        return *this;           // return the object
assigned
}

bool Circle::operator== (const Circle& c)
{
        if (radius == c.radius)
                return true;
        return false;
}

bool Circle::operator< (const Circle& c)
{
        if (radius < c.radius)
                return true;
        return false;
}
```

```
Circle Circle::operator+ (const Circle& c)
{
        Circle tmp;
        tmp.radius = radius + c.radius;
        return tmp;
}

Circle Circle::operator* (const Circle& c)
{
        Circle tmp;
        tmp.radius = radius * c.radius;
        return tmp;
}
```

```
Circle Circle::operator++ ()     // preincrement
{
        ++radius;
        return *this;
}

Circle Circle::operator++ (int u)// postincement
{
        Circle tmp = *this;
        ++radius;
        return tmp;
}
```

```
class Book
{
public:
        Book (string, int);
        Book (const Book&);
        ~Book ();
        void Print () const;
        void InsertAtPosition (string, int);

        const Book& operator= (const Book&);
        bool operator== (const Book&);

private:
        string Title;
        int NumChapters;
        string* Chapters;
};
```

```
Book::Book (string t, int num)
: Title (t), NumChapters (num)
{
        Chapters = new string[NumChapters];
        for (int i = 0; i < NumChapters; ++i)
                *(Chapters + i) = "";
                // or // Chapetrs[i] = "";
}
Book::Book (const Book& b)
: Title (b.Title), NumChapters (b.NumChapters)
{
        Chapters = new string[NumChapters];
        for (int i = 0; i < NumChapters; ++i)
                *(Chapters + i) = *(b.Chapters + i);
                // or // Chapetrs[i] = b.Chapters[i];
}

Book::~Book ()
{
        delete [] Chapters;
}
```

```
void Book::Print () const
{
        cout<<"Title: "<<Title<<endl;
        for (int i = 0; i < NumChapters; ++i)
                cout<<"Chapter "<<i<<":
"<<Chapters[i]<<endl;
}

void Book::InsertAtPosition (string chp, int pos)
{
        if (pos < 0 || pos >= NumChapters)
                return;

        Chapters[pos] = chp;
}
```

```
const Book& Book::operator= (const Book& b)
{
        if (this != &b) {
                Title = b.Title;
                NumChapters = b.NumChapters;

                 delete [] Chapters ;
                Chapters = NULL:
                Chapters = new string[NumChapters];
                for (int i = 0; i < NumChapters; ++i)
                        *(Chapters + i) = *(b.Chapters + i);
                        // or // Chapetrs[i] = b.Chapters[i];
        }
        return *this;
}
```

```
bool Book::operator== (const Book& b)
{
        return this == &b;
}
```

OR compare all members in Book

```
bool Book::operator== (const Book& b)
{
        if (Title != b.Title)
                return false;

        if (NumChapters != b.NumChapters)
                return false;

        for (int i = 0; i < NumChapters; ++i)
                if (*(Chapters + i) != *(b.Chapters + i))
                        return false;

        return true;
}
```

```
void main ()
{
        Book Cplusplus ("C++ Programming", 4);
        Cplusplus.Print ();
        Cplusplus.InsertAtPosition ("Arrays", 0);
        Cplusplus.InsertAtPosition ("Functions", 1);
        Cplusplus.InsertAtPosition ("Loops", 5);// will not proceed
        Cplusplus.Print ();

        Book Java ("Java Programming", 3);
        Java.Print ();

        Java = Cplusplus;
        Java.Print ();
        Java.InsertAtPosition ("Operator Overloading", 2);

        if (Cplusplus == Java)
                cout<<"Same Book\n";
        else
                cout<<"Different Books\n";
}
```

Postincement operator overloading

To distinguish between pre- and post- operator overloading, we use dummy parameter (of type int) in the function header.

```
Circle Circle::operator++ (int u)          // postincement
{
        Circle tmp = *this;
        ++radius;
        return tmp;
}
```

e.g.

```
Circle c1 (5);

c1++;               // interpreted by compiler as
                    // c1.operator++(0);

++c1;               // interpreted by compiler as
                    // c1.operator++();
```

Friend functions

A friend function is a nonmember function that has access to private data member of the class

```
class IllustrateFriend
{
        friend void Two (IllustrateFriend);

public:
        void Print ()
        { cout<<x; }
        void SetX (int a)
        { x = a; }

private:
        int x;
};
```

```
void Two (IllustrateFriend obj)
{
        obj.x = 20; // friend function can
        cout<<obj.x; // access private x
}

void main ()
{
        IllustrateFriend a;
        a.SetX (5);
        Two (a);
}
```

Overloading operators using friend functions

```
class Rectangle
{
        friend bool operator< (const Rectangle&, const Rectangle&);
        friend Rectangle operator+ (const Rectangle&, const Rectangle&);
        friend Rectangle operator-- (Rectangle&);     // pre
        friend Rectangle operator-- (Rectangle&, int); // post

public:
        Rectangle (float l = 0.0f, float w = 0.0f)
                : length (l), width (w)
        {}
private:
        float length, width;
};
```

Overloading operators using friend functions

```
bool operator< (const Rectangle& r1, const Rectangle& r2)
{
        return r1.length < r2.length &&
                        r1.width < r2.width;
}


Rectangle operator+ (const Rectangle& r1, const Rectangle& r2)
{
        Rectangle tmp;
        tmp.length = r1.length + r2.length;
        tmp.width = r1.width + r2.width;

        return tmp;
}
```

Overloading operators using friend functions

```
Rectangle operator-- (Rectangle& r)      // pre-decrement
{
        --r.length;
        --r.width;

        return r;
}

Rectangle operator-- (Rectangle& r, int u) // post-decrement
{
        Rectangle tmp (r);

        r.length--;
        r.width--;

        return tmp;
}
```

Template function

When implementing the function **Larger** to find the larger of two integers, floating points, characters, or strings you need to overloading the function 4 time as:

```
int Larger (int a, int b)
{
        if (a >= b)
                return a;
        return b;
}
float Larger (float a, float b)
{
        if (a >= b)
                return a;
        return b;
}
```

```
char Larger (char a, char b)
{
        if (a >= b)
                return a;
        return b;
}
string Larger (string a, string b)
{
        if (a >= b)
                return a;
        return b;
}
```

Template class

Since all 4 functions have the same body, we can use template as:

```
template <class Type>
Type Larger (Type, Type);

void main ()
{
        cout<< Larger (5, 10)<<endl;
        cout<< Larger (5.7, 40.9)<<endl;
        cout<< Larger ('a', 'D')<<endl;
        cout<< Larger ("Sun",
"Earth")<<endl;
}
```

```
template <class Type>
Type Larger (Type a, Type b)
{
        if (a >= b)
                return a;
        return b;
}
```

Template function

Important: when using templates, put both class interface and implementation in the .h file

```
#ifndef LIST_H
#define LIST_H

template <class Type>
class List
{
public:
        List (int);
        ~List ();
        void Insert (Type);
        bool IsFull () const;
        bool IsEmpty () const;
        void Print () const;
private:
        int size, count;
        Type* lst;
};

#endif
```

```
template <class Type>
List<Type>::List (int s)
: size (s), count (0)
{
        lst = new Type[size];
}

template <class Type>
List<Type>::~List ()
{
        delete [] lst;
        lst = NULL;
}

template <class Type>
void List<Type>::Insert (Type el)
{
        if (!IsFull ())
                lst[count++] = el;
}
```

```
template <class Type>
bool List<Type>::IsFull () const
{
        return count == size;
}

template <class Type>
bool List<Type>::IsEmpty () const
{
        return count == 0;
}

template <class Type>
void List<Type>::Print () const
{
        for (int i = 0; i < count; ++i)
                cout<<lst[i]<<endl;
}
```

```
void main ()
{
        List<int> IntList (5);
        IntList.Insert (10);
        IntList.Insert (20);
        IntList.Insert (30);
        IntList.Print ();

        List<string> StringList (4);
        StringList.Insert ("First");
        StringList.Insert ("Second");
        StringList.Insert ("Third");
        StringList.Print ();
}
```

File Input/Output

- Reading data from input files and saving output to output files:
- File I/O five steps
  - Include the header file fstream
    e.g. #include<fstream>
  - Declare file stream variables
    e.g. Ifstream inDate;
    ofstream outDate
  - Associate file stream variables with input/output sources
    e.g. inDate.open ("input.txt", ios::in);
    outDate.open ("input.txt", ios::in);
  - Use the file stream variables with >>, <<
    e.g. inData>>x>>y;
    outDate<<x<<y<<endl;
  - Close the files
    e.g. inData.close ();
    outDate.close ();

File Input/Output Example

- Read midTerm grades from an input file (midTermGrades.txt) and write the average into an output file (midTermAverage.txt).

midTermGrades.txt

```
20.5
30.0
15.0
17.0
23.0
25.5
13.0
15.5
11.0
27.0
8.0
-1
```

midTermAverage.txt

```
The average is 18.6818
```

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

void main ()
{
        ifstream in; // declare input stream variable
        in.open ("midTermGrades.txt", ios::in); // midTermGrades.txt is located in the current directory
        // or // in.open ("c:\\midTermGrades.txt", ios::in); // midTermGrades.txt is located in C:\ drive

        int count = 0;
        float grade, sum = 0.0f;
        in>>grade;
        while (grade != -1) {
                sum += grade;
                count++;
                in>>grade;
        }

        in.close ();

        ofstream out;           // declare output stream variable
        out.open ("midTermAverage.txt", ios::out); // empty midTermAverage.txt will be created
        out<<"The average is "<<sum / count<<endl;

        out.close ();
}
```

Sizeof

- The **sizeof** is a keyword, but it is a compile time operator that determines the size, in bytes, of a variable or data type.
- Syntax: sizeof (data type)

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

int main()
{
  cout << "Size of char : " << sizeof(char) << endl;
  cout << "Size of int : " << sizeof(int) << endl;
  cout << "Size of short int : " << sizeof(short int) << endl;
  cout << "Size of long int : " << sizeof(long int) << endl;
  cout << "Size of float : " << sizeof(float) << endl;
  cout << "Size of double : " << sizeof(double) << endl;

return 0;
}
```

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
```