Data Structure (introduction 2)

Objectives

1. **Class templates**
2. **Operator overloading**
3. **Overloading various operators**

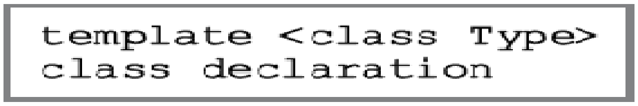
1. Class templates

– Writing a single code segment for a set of related classes

– Data types: parameters to templates

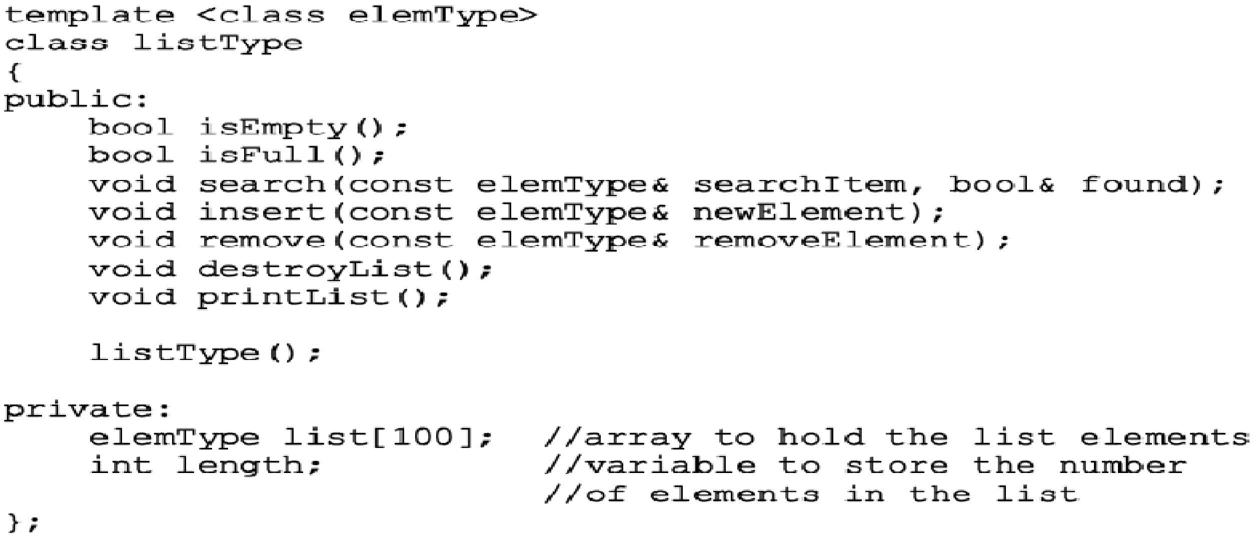
- Called parameterized types

- Syntax



```
template <class Type>
class declaration
```

- Example



```
template <class elemType>
class listType
{
public:
    bool isEmpty();
    bool isFull();
    void search(const elemType& searchItem, bool& found);
    void insert(const elemType& newElement);
    void remove(const elemType& removeElement);
    void destroyList();
    void printList();

    listType();

private:
    elemType list[100];   //array to hold the list elements
    int length;           //variable to store the number
                          //of elements in the list
};
```

Header File and Implementation File of a Class Template

- Not possible to compile implementation file independently of client code

- Solution

– Put class definition and definitions of the function templates directly in client code

– Put class definition and definitions of the function templates together in same header file

– Put class definition and definitions of the functions in separate files (as usual): include directive to implementation file at end of header file

2. Operator Overloading

- Why operator overloading is needed
  - ➢ Built-in operations on classes : Assignment operator and member selection operator
  - ➢ Other operators cannot be directly applied to class objects

- Relational operators, arithmetic operators, insertion operators for data output, and extraction operators for data input applied to classes

– Examples : Stream insertion operator (<<), stream extraction operator(>>), +, and –

– The operator << is used as both a stream insertion operator and a left shift operator.

– The operator >> is used as both a stream extraction operator and a right shift operator.

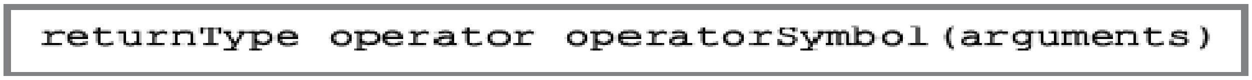
– Overload an operator : you must write functions (header and body)

– Function name overloading an operator: reserved word **operator** followed by operator to be overloaded

– Function name: operator>=

- **Syntax for Operator Functions**
  - ➢ Operator function: value-returning function
  - ➢ Include statement to declare the function to overload the operator in class definition
  - ➢ Write operator function definition

- **Operator function heading syntax:**



```
returnType operator operatorSymbol(arguments)
```

Overloading an Operator: Some Restrictions

- Cannot change operator precedence

- Cannot change associativity: Example: arithmetic operator + goes from left to right and cannot be changed

- Cannot use default arguments with an overloaded operator

- Cannot change number of arguments an operator takes
- Cannot create new operators
- Some operators cannot be overloaded: .* :: ?: sizeof

3. Operator Functions as Member Functions and Nonmember Functions

Operators can be overloaded as member functions or nonmember functions

- Function overloading operators (), [], ->, or = for a class must be declared as a class member function.

– Two rules when including operator function in a class definition, suppose operator **op** overloaded for class **opOverClass** :

1. If leftmost operand of op is an object of a different type: Function overloading operator **op** for **opOverClass** must be a nonmember (friend of class opOverClass)

• If operator functions overloading operator **op** for class **opOverClass** is a member of the class **opOverClass**: When applying **op** on objects of type **opOverClass**, leftmost operand of op must be of type **opOverClass**.

– Functions overloading insertion operator (<<) and extraction operator (>>) for a class must be nonmembers

- C++ consists of binary and unary operators

– C++ contains a ternary operator : they cannot be overloaded

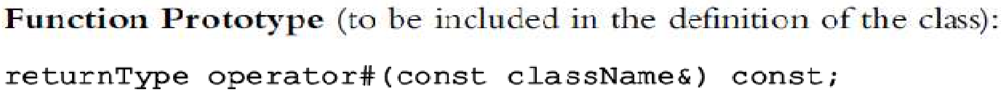
Overloading Binary Operators

Two ways to overload

– As a member function of a class

– As a friend function

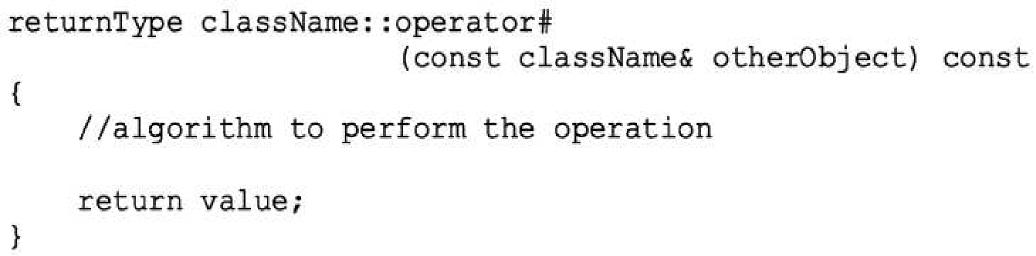
As member functions: General syntax:



Function Prototype (to be included in the definition of the class):

```
returnType operator#(const className&) const;
```

– **Function definition**



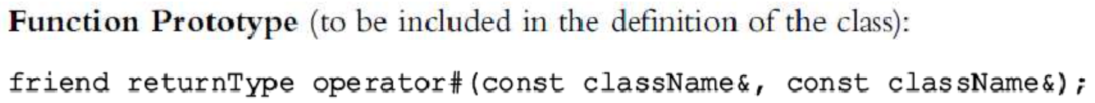
```
returnType className::operator#
                       (const className& otherObject) const
{
    //algorithm to perform the operation

    return value;
}
```

# stands for the binary operator.

The return type of the function that overloads a relational operator is **bool.**

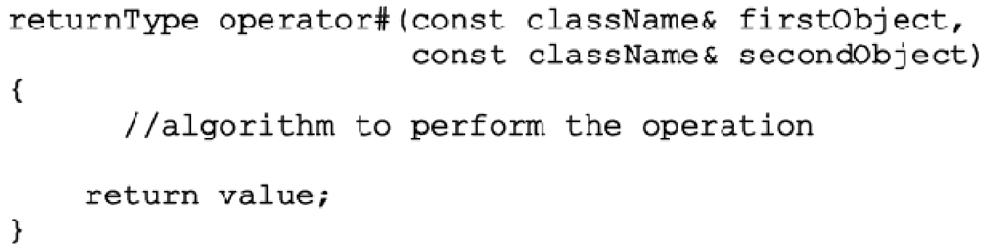
As nonmember functions: General syntax



Function Prototype (to be included in the definition of the class):

```
friend returnType operator#(const className&, const className&);
```

Function definition



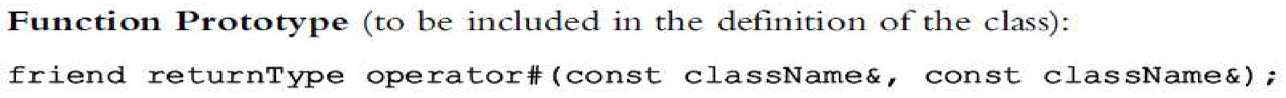
```
returnType operator#(const className& firstObject,
                     const className& secondObject)
{
      //algorithm to perform the operation

    return value;
}
```

Overloading the Stream Insertion (<<) and Extraction (>>) Operators

Operator function overloading insertion operator and extraction operator for a class must be nonmember function of that class

Overloading the stream extraction operator (>>): General syntax and function definition



Function Prototype (to be included in the definition of the class):

```
friend returnType operator#(const className&, const className&);
```

Function Prototype: (to be included in the definition ofb the class)

frirend ostream & operator<< (ostream & , const className &);

Function defnition

ostream & operator<< ostream & osObj, const className & cObj)

```
{

// local declation, if any

//Output the members

return osObj;

}
```

Overloading unary operations

- Similar to process for overloading binary operators
- Difference: unary operator has only one argument

Process for overloading unary operators

- If operator function is a member of the class: it has no parameters
- If operator function is a nonmember (friend function of the class): it has one parameter

Example 2-6 (from the textbook)

Class Interface

```
#ifndef H_rectangleType
#define H_rectangleType
#include <iostream>
using namespace std;
class rectangleType
{
     //Overload the stream insertion and extraction operators
   friend ostream& operator << (ostream&, const rectangleType &);
   friend istream& operator >> (istream&, rectangleType &);
public:
   void setDimension(double l, double w);
   double getLength() const;
   double getWidth() const;
   double area() const;
   double perimeter() const;
   void print() const;
   rectangleType operator+(const rectangleType&) const;
     //Overload the operator +
   rectangleType operator*(const rectangleType&) const;
     //Overload the operator *
   bool operator==(const rectangleType&) const;
```

```
      //Overload the operator ==
    bool operator!=(const rectangleType&) const;
      //Overload the operator !=
    rectangleType();
    rectangleType(double l, double w);
private:
    double length;
    double width;
};
#endif
```

Class Implementation

```
#include <iostream>
#include "rectangleType.h"
using namespace std;
void rectangleType::setDimension(double l, double w)
{
    if (l >= 0)
        length = l;
    else
        length = 0;
    if (w >= 0)
        width = w;
    else
        width = 0;
}
double rectangleType::getLength() const
{
    return length;
}
double rectangleType::getWidth()const
{
    return width;
}
double rectangleType::area() const
{
    return length * width;
}
double rectangleType::perimeter() const
{
    return 2 * (length + width);
}
void rectangleType::print() const
{
    cout << "Length = "  << length
         << "; Width = " << width;
}
rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}
rectangleType::rectangleType()
{
    length = 0;
    width = 0;
}
rectangleType rectangleType::operator+
                          (const rectangleType& rectangle) const
{
    rectangleType tempRect;
    tempRect.length = length + rectangle.length;
    tempRect.width = width + rectangle.width;
    return tempRect;
}
rectangleType rectangleType::operator*
                       (const rectangleType& rectangle) const
{
    rectangleType tempRect;
    tempRect.length = length * rectangle.length;
    tempRect.width = width * rectangle.width;
    return tempRect;
}
bool rectangleType::operator==
                      (const rectangleType& rectangle) const
{
    return (length == rectangle.length &&
            width == rectangle.width);
}
bool rectangleType::operator!=
                       (const rectangleType& rectangle) const
{
    return (length != rectangle.length ||
            width != rectangle.width);
}
ostream& operator << (ostream& osObject,
                      const rectangleType& rectangle)
{
    osObject << "Length = "  << rectangle.length
             << "; Width = " << rectangle.width;
```

```
   return osObject;
}
istream& operator >> (istream& isObject,
                      rectangleType& rectangle)
{
   isObject >> rectangle.length >> rectangle.width;

   return isObject;
}
```

Test pogram

```
#include <iostream>                                  //Line 1
#include "rectangleType.h"                           //Line 2
using namespace std;                                 //Line 3
int main()                                           //Line 4
{                                                    //Line 5
   rectangleType myRectangle(23, 45);                //Line 6
   rectangleType yourRectangle;                      //Line 7
   cout << "Line 8: myRectangle: " << myRectangle
        << endl;                                     //Line 8
   cout << "Line 9: Enter the length and width "
        <<"of a rectangle: ";                        //Line 9
   cin >> yourRectangle;                             //Line 10
   cout << endl;                                     //Line 11
   cout << "Line 12: yourRectangle: "
        << yourRectangle << endl;                    //Line 12
   cout << "Line 13: myRectangle + yourRectangle: "
        << myRectangle + yourRectangle << endl;      //Line 13
   cout << "Line 14: myRectangle * yourRectangle: "
        << myRectangle * yourRectangle << endl;      //Line 14
   return 0;                                         //Line 15
}                                                    //Line 16
```