Basic Operator Overloading

Overview

- What is Operator Overloading
- When to Use it (and when not)
- Binary and Unary Operators
- Assignment Operator
- The ‘this’ Pointer
- The Canonical Header File

2

Operator Overloading

- Powerful feature in C++
- Use keyword 'operator'
- Define new behaviour for operator
- Watch out for misuse

3

OPERATOR OVERLOADING

In C++ the user is offered a number of ways to extend the language. One of those features is called operator overloading. Basically this comes down to defining operators for user-defined types. This can be very handy when creating mathematical classes (for example matrices, vectors etc.). By defining our own operators for these classes the code will become more readable for these objects; however operator overloading can make the code more unreadable if we do not apply operator overloading logically.

Overloading Binary Operators

- Declare like normal member function

```
Complex c1, c2, c3;
c3 = c1 + 10;  // Complex::operator + (int)
c3 = c1 + c2;  // Complex::operator + (const Complex&);
```

- Left object is receiver

```
a + b;  // 'a' receives message operator + (b)
        // a.operator + (b);
```

4

Overloading Binary Operators

Suppose we have a class representing matrices. We want to create a function that can multiply two matrices and give back the result. One possibility could be to create a normal function with two matrices as arguments and use the accessing functions of the matrices to implement the multiplication. It would be more logical to write a multiplication function for the matrix class.

The most logical thing to do is to add a member function to the matrix class.

Matrix Multiplication

```
class Matrix
{
      Matrix multiply(const Matrix& mat); // Multiply a matrix
};
```

Notice the const and '&', we receive another object as argument.

If we want to add two matrices using this member function the code would look like as follows:

```
Matrix m1, m2, m3;
m3 = m1.multiply(m2);
```

This piece of code is still readable, but it can become quite unreadable if we want to use subtraction as well. If we want to create a more readable piece of code we would like to see some kind of multiply operator just as you would see in a mathematics book. This can be done by overloading the binary operator* for our matrix class.
We want to define new behaviour for the operator '*' in order to multiply two matrices.

```
m3 = m1 * m2;
```

This code is far more readable than the first option. In this example we are using the operator '*' and the operator '='. The assignment operator will be covered later. Let us first concentrate on the operator*.

We want to define the operator * for the class matrix. The operator* must become a member function of the class matrix. The syntax for an operator overload is:

<return_type> operator <op_name>(<argument_type>)

If we implement this for our matrix class the header would look like:

Operator*

```
class Matrix
{
public:
      Matrix multiply(const Matrix& mat);
      Matrix operator*(const Matrix& mat);
};
```

The only difference between the multiply() function and the '*' operator is the name of the function. The code for both functions is the same. Another difference between these functions is the way we call them.

```
1) m3 = m1.multiply(m2);
2) m3 = m1 * m2;
```

When calling the operator function you do not have to specify the operator keyword. The object m1 is in both cases the object that receives the message.

In option 1) the object m1 receives the message mutliply(m2), and in option 2) the object receives the message operator*(m2). As you see the only difference is the message that the object receives.

Through operator overloading we can declare several operators* with different arguments.

Multiple Operator*

```
class Matrix
{
public:
      Matrix operator*(const Matrix& mat);
      Matrix operator*(int value);
      Matrix operator*(const Vector& vec);
};
```

What you always need to remember is that the left object is always the receiver. This is the object that receives the message.

When you want to use operator overloading think about the logical interface of your class. Only implement the operator in your class if it is a logical part of the interface of that class.

Overloading Unary Operators

- No arguments

```
Complex c1, c2;
c2 = -c1;        // c1.operator - ()
```

- Normally returns a copy of negated current object

5

OVERLOADING UNARY OPERATORS

The operators +, -, * and & can be overloaded in both their unary and binary form. We have already seen how to overload binary operators in the previous example.

The overloading of unary operators is much the same. The only difference are the arguments of the functions. The unary operators do not have an argument.

```
class Matrix
{
public:
      ...operator-();
};
```

Unary operator-

The return type of the operator is left out intentionally. The return type of this operator is debatable. Basically it comes down how do you want to let the user use this operator.

The choice depends on whether you want to allow the use of this operator in a calculation or as a stand-alone operation.

Matrix m1;

-m1; // Stand alone operation changes m1
m2 = m3 * -m1; // In calculation does not change m1

If we allow the client to use the operator as a stand-alone operator, the return type can be void or a reference to the object.

In the second option the function must return a matrix, the changed matrix (preferable).

Assignment Operators Scenarios

- Assigning same types

```
Point pt1, pt2;
pt1 = pt2;       // pt1.operator = (pt2);
```

- Operator must make a deep copy of all the members of argument into current object
- Like copy constructor

6

ASSIGNMENT OPERATORS

The assignment operators are special operators that can be overloaded. Assignment operators are the operators which have a '=' in them.

=, +=, -=, *=, &=, <<=, >>= etc..

The most obvious assignment operator would be the one that copies one point to the other.

```
Point p1, p2;

p2 = p1;
```

Operator=

The assignment operator for the Point class is called, p2.operator=(p1).

The assignment operator does much of the same work as the copy constructor. In some cases however the assignment operator can become rather complex.

Assignment and Copy Constructor

- Copy constructor gets called when creating a new object
  - Object has no old values
- If assignment operators gets called object has data already
  - Clear the old data

7

Assignment Vs. Copy Constructor

The assignment operator and the copy constructor are two member functions with almost the same functionality they both copy data members of one object in the current. The difference between the two member functions is that the copy constructor is called when the object is created (the copy) and the assignment operator is called when the object that is destined to be the copy already exists.

In the case of the copy constructor we are creating a new object so no old data needs to be removed it is a fresh object. We can concentrate on copying the data appropriately.

Copy Constructor

```
Point pt(10.0,20.0);// A point is created with coordinates 10 and 20
Point pt2(pt);    // The copy constructor is called, pt2 is a new object
                  // which is initialised with the same values
                  // as the data members of pt1
```

The assignment operator is a different case. When the assignment operator is called on an object the object already had a life. During this life its data members gained values and meaning. When we want to overwrite these existing values with their meaning we have to eliminate them after which we can copy the data members of the second object into this object.

Assignment Operator

```
Point pt(10.0,20.0);// A point is created with coordinates 10 and 20
Point pt2;        // pt2 is a default object as wel
//pt2 has a life of its own

pt2 = p1;         // The assignment operator is called for pt2
                  // Its data members need to be reinitialized and
                  // the data of p1 needs to be copied to pt2
```

Assignment Operator

```
... Point::operator = (const Point& source)
{
  m_x = source.m_x;
  m_y = source.m_y;

  // What should be returned ?
}
```

Return Type Of Assignment

```
Point pt1, pt2, pt3;

pt1 = pt2;           // pt1.operator = (pt2);
pt3 = pt1 = pt2;     // pt3.operator = (pt1.operator = (pt2));
```

- Operator = should return pt1 (current object) or pt2

9

RETURN TYPE OF ASSIGNMENT

When defining the assignment operator, the first problem that arises is what does this operator return for value. In most cases you want to support the following:

```
p3 = p2 = p1;      // p3 = (p2 = p1)
```

This expression is evaluated from right to left. First p1 is assigned to p2 after which the result of this assignment is assigned to p3. So the result of the first assignment should be p2 or a value equivalent to p2 e.g. p1.

Let us take a closer look at the first assignment p2 = p1. Actually the operator= for the p2 object is called. The code can be written as p2.operator=(p1).
The function header could look like:

```
class Point
{
Point operator=(const Point& pt2);
};
```

Return A Copy

```
Point Point::operator = (const Point& source)
{
  m_x = source.m_x;
  m_y = source.m_y;

  return source;  // return the argument
}
```

Operator should return itself

Objects Returning Themselves

- Object can return it self (the current object)
- Objects reference themselves by using the ‘this’ pointer
- Can only be used inside member functions (for the current object)
- Used as an 'ordinary' pointer to an object

11

OBJECTS RETURNING THEMSELVES

When we are in the body of member functions we talk about the current object also referenced as self. The current object is the object on which the member function in which we are is called. Sometimes we need to reference this so-called current object. The problem however is that the current object does not have a name as a function argument does. To solve the problem of referencing the current object there is a so-called ‘this’ pointer.

The this pointer is defined as pointing to the current object. You only have a current object in a member function so only member functions have a this pointer. Constant member functions are said to have a constant this pointer.

Some programmers use the this pointer to keep a clear difference between global variables and data members in member functions. Compare the below functions, they do exactly the same.

Using this

```
int Date::day()
{// Without the this pointer
      return m_day;
}

int Date::day()
{// With the this pointer
      return this->m_day;
}
```

The use of ‘this’ as used in this manner is a bit outdated.

The this pointer is however very useful when we want to return the current object as a whole and not just a member from the current object. By returning *this we return the current object.

Returning Current Object

- With no reference copy is made

```
Point Point::operator = (const Point& source)
{
  m_x = source.m_x;
  m_y = source.m_y;

  return *this;  // Copy of current object is returned
}
```

12

Assignment Operator Returns Current Object

The assignment operator in the previous assignment example returned the argument object so that the assignment could be concatenated. However the object we returned was not the object we really should return. If we look at the previous example:

```
p3 = p2 = p1;      // p3 = (p2 = p1)}
```

Concatenating Assignment

The object p1 is assigned to object p2 that in its place is assigned to object p3. In our previous example of the assignment operator our assignment assigned object p1 to object p2 which is incorrect, object p2 should be assigned to object p3. Therefor our operator= should return object p2 the object on which the operator= was called, so this operator should return the current object.

```
Point Point::operator=(const Point& pt2)
{
      x = pt2.x;
      y = pt2.y;
      return *this;
}
```

Assignment Operator Without References

There is only one drawback to this solution the current object is copied before we return it, to solve this problem see the later sheets about returning references.

Other Examples of this

- The following statements are equivalent

```
return m_x;           // Preferred
return this->m_x;
return (*this).m_x;
```

13

Returning References

- Used for avoiding unnecessary copy of object

```
Point& Point::operator = (const Point& source)
{
  m_x = source.m_x;
  m_y = source.m_y;

  return *this;  // Current object is returned
}
```

14

RETURNING NON REFERENCES

When a function returns an object of some kind a copy is created and the copy is what the caller of that function receives.

Returning Non References

```
Point print(const Point& pt)
{
      cout << pt.ret_x() << endl;
      cout << pt.ret_y() << endl;

      return pt;        // Copy is created not necessary
}

void main()
{
      Point pt;

      pt = print(pt);   // pt is assigned what the print function
                        // returns in this case a copy of pt
}
```

Our previous assignment operator was a function that copied the current object because it returned *this.

RETURNING REFERENCES

By using reference as return values we prevent the system from creating copies of whatever is returned.

Returning References

```
Point& print(const Point& pt)
{
      cout << pt.ret_x() << endl;
      cout << pt.ret_y() << endl;

      return pt;        // No copy is created
}

void main()
{
      Point pt;

      pt = print(pt);   // pt is assigned what the print function
                        // returns in this case pt itself
}
```

Our previous assignment operator returned a copy of the current object we now how to prevent this copy process. We can let the assignment operator return not a copy but the original current object using a reference.

Assignment Operator With References

```
Point& Point::operator=(const Point& pt2)
{
      x = pt2.x;
      y = pt2.y;
      return *this;
}
```

It is not mandatory to let an operator= return a reference to the current object it is however advisable to make sure no unnecessary copies are created.

References and LHS

- Functions returning references can be used on the LHS side of the assignment

```
class Point
{
public:
  double& GetX() const;
};

Point pt;
pt.GetX() = 10.0;
```

15

REFERENCES ON THE LHS SIDE

When a function returns a reference to a class this return value can be placed on the left side of the assignment operator. Our function returns a value that we can change.

Simple Class

```
class SimpleValue
{
private:
      int value;  // a simple value
public:
      SimpleValue();
      ~SimpleValue();
      void SetValue(int newVal);
      int& GetValue();
};

void SimpleValue::SetValue(int newVal)
{
      value = newVal;
}

int& SimpleValue::GetValue()
{
      return value;
}
```

The SetValue() in the above example changes the internal value of our simple class. In the above class file this should be the only way to change this value. But if we let the GetValue() function return a reference we make it possible to change the internal function with that function as well.

References on the LHS Side

```
void main()
{
      SimpleValue Simp;
      Simp.SetValue(10);
      cout << Simp.GetValue();      // Should display the value 10
      Simp.GetValue() = 20;
      cout << sim.GetValue();       // Will display 20
}
```

In the above example the call to the GetValue() function returns the internal value as a reference supplying the caller the original internal value which is changed in this case into 20.

So be careful to not automatically let a function return references when possible it can have unwanted results.

Pitfalls With Returning References

- Can only return reference to an object with a lifetime longer than function body

```
Point& Point::Add(const Point& pt)
{
  Point res;

  ...

  return res;          // ERROR reference to temporary object
}
```

16

PITFALLS WITH RETURNING REFERENCES

There are restrictions to values returned by reference. The object the function returns must outlive the lifetime of that function.

Incorrect Reference

```
Point& print(const Point& pt)
{
      Point pt2;
      ..
      return pt2; // NOT POSSIBLE
                  // Because it returns a temporary object
}
```

Custom Assignments

- Not only for same type as object

```
Point pt1;
pt1 = 10.0;    // pt1.operator = (10.0) -> x,y,z = 10.0
```

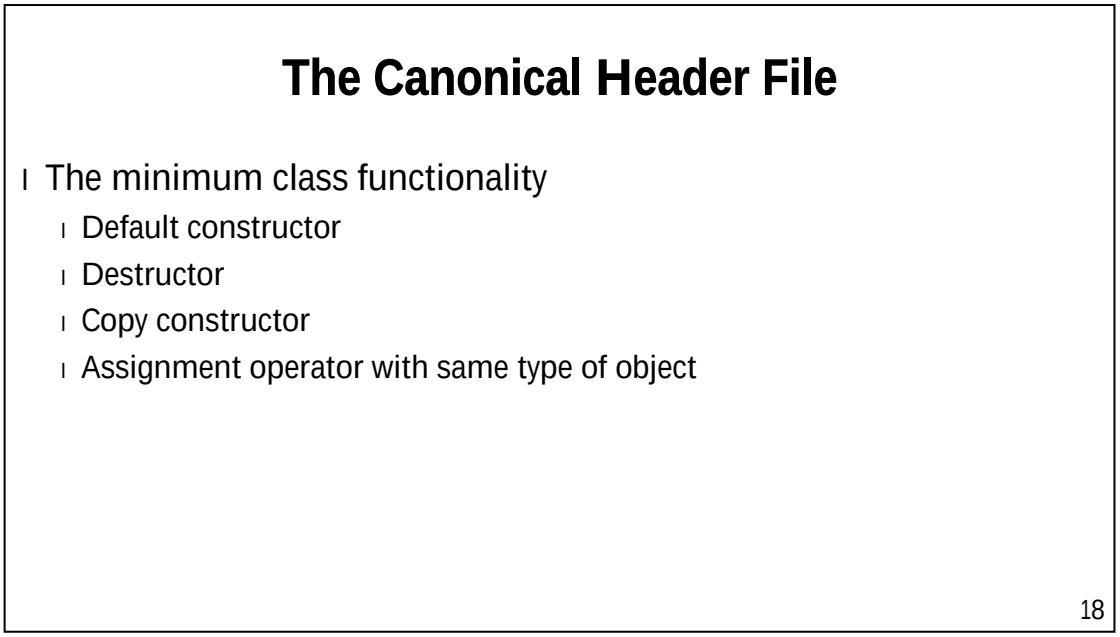
- Can combine Assignment with all operators
  - = += -= *= etc..

17

CUSTOM ASSIGNMENTS

Normally you always create an assignment operator for a class to copy an object of the same type. If you fail to create this operator the system provides one for you. This system created operator works correct if your object has nothing to do with references or pointers or things like those. The system creates a bitwise copy assignment. In other cases you need to provide an assignment operator so that the object is copied correctly. The assignment operator however can be overloaded for different types of assignment. Perhaps you would like to be able to assign an int value to a point object, or a double to a rational object. These types of assignments operators need to be defined by the developer him self, these will not be provided by the system.

Do not think that if you create an assignment operator and an operator+ that combinations like += will work automatically, you need to create a overloaded += separately.



THE CANONICAL HEADER FILE

Whenever you create a class you have to create a so-called canonical header file for that class. The canonical header file consists of four special member functions. When you do not provide one of these member functions the system will provide them. When the system provides these functions you lose a part of control of your class. And when creating a class you want absolute control of your class to make it a 'safe' class.

When creating a class the first thing you need to do is create a canonical header file which consists of the four member functions mentioned above.

Default constructor
Called when a default object is created.

```
Date d;     // Default constructor called
```

Destructor (virtual)
Called by the system when an object looses its scope

Copy constructor
Called when a copy of the object is required. This constructor is called in three situations:
1) Creating an object with another object of the same class.

```
Point p1;
Point p2(p1);
```

2) When an object is passed as a by value argument to a function

```
void print_point(Point pt);
```

3) When a function returns an object.

```
Point read_Point();
```

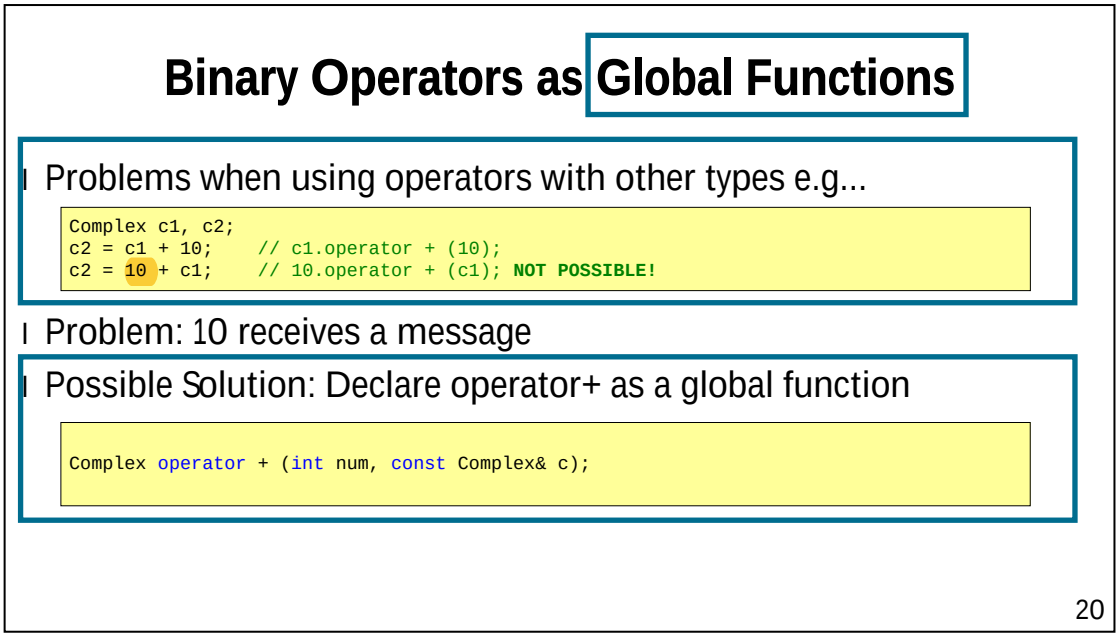
Assignment operators, with an object of the same type as argument
The assignment operator is called when you assign two objects to of the same class

```
Point p1, p2;
p1 = p2;
```

Example Minimal Functionality

```
class Point
{
public:
  Point();                                  // Default constructor
  ~Point();                                 // Destructor
  Point(const Point& source);               // Copy constructor
  Point& operator = (const Point& source); // Assignment operator
};
```

19



BINARY OPERATORS AS GLOBAL FUNCTIONS

As we saw earlier operator overloading is best used with mathematically based classes.

In the previous operator overloading examples the left object was always an object that could receive a message, meaning that it can have different member functions for those messages.

```
Complex c1, c2, c3;

c3 = c1 + c2;
c3 = c1 + 10:
```

Operator+ scenarios

Object c1 is the object that receives the operator+ message al the times. We can implement the functionality of these operators by adding them as member functions to the Complex class. But now suppose the left object (or operand) is not an object of the class Complex, but for example an integer.

The left object (value 10 in this case) cannot receive any messages because it is not instance. We cannot implement the functionality of the operator+ by adding it to the instance 10. This is a problem and one solution would be by letting the user know that he cannot use the Complex in this way.

```
c3 = 10 + c1;
```

Left operand is a constant

But when delivering a Complex class you want to deliver it as complete as possible. The solution to this problem can be found by looking at the operator+ as a global function with two arguments. The first argument of the function is the left operand and the second argument is the right operand.

```
Operand1 + Operand2
```

This expression can be looked at as

<return_type> operator+(Operand1_Type Operand1, Operand2_type Operand2);

The expression is executed as if it is a normal global function with two arguments, thus the operator+ is not a member function of the left object (*Operand1*).

The function operator+ can use the accessing functions of both objects to perform the final addition.

```
Complex operator+(int value, const Complex& pt);
```

As a global function

Remember when defining this function that it is not a member function but a normal global function that can access only the public member functions of Point objects.

The function can be placed in any source file; the function must be defined without a scope resolution operator.

```
Complex operator+(int value, const Complex& pt)
{
    // Body of function
}
```

As a global function

Friends

- These are functions (or classes) that can directly access the private members of a given class
- Hardly ever use them (bad design), except...
  - Binary operators in which first operand has different type than second operand can be declared as friend
- A friend function is NOT a member function

Friend Classes and Friend Functions

- Friends can access all members
- Violate information hiding principle
- Classes select their friends
- Can have
  - Friend functions
  - Friend classes

22

FRIEND CLASSES AND FRIEND FUNCTIONS

The term 'friend' is used to specify that a class or function is not part of the class but can access all its members, including the private members. Specifying that a function or class is a friend can be dangerous; it is in fact a violation of the information hiding principle.

Friend classes and functions are used for the following reasons:

- Efficiency; functions can directly access the private parts of a class without having to use a function call of that class to access the private state.
- Tightly coupled classes can be friends of each other (e.g. matrices and vectors).
- Using global operator overloading (friend operators) leads to powerful class functionality and cleaner interfaces.

Always try to use friend as little as possible. The advantage of friends is that a class specifies its friends but not vice versa.

Whenever you specify a friend use the keyword 'friend'.

Friend Functions

- Friend functions can access all members of certain objects
- A class makes certain functions its friends
- Functions do not become member functions
- Friends are friends for all members not certain members (private and public have no effect)
- Friend functions cannot be const (not a member)

23

FRIEND FUNCTIONS

Friend Functions, Declaration

```
class Point
{
	friend Point MAX_POINT(const Point& p1, const Point& p2);
};
```

The above function MAX_POINT returns the point with the largest x value. You have to remember that a friend function is a global function and NOT a member function. The function can access all the members of the objects p1 and p2 but it is not a member function of the class Point. The function therefore can be placed in any source file, without using the scope resolution operator (::). The friend functions are mostly placed in the source file of the class, but this is not mandatory. It is the advised place, however.

Friend Functions, Implementation

```
Point MAX_POINT(const Point& p1, const Point& p2)
{
	if(p1.x < p2.x)
	{
		return p2;
	}
	return p1;
}
```

Friend Functions

```
class Point
{
private:
  double m_x;
  double m_y;

  friend void Display(const Point& pt);
};

void Display(const Point& pt)
{
  cout << pt.m_x << pt.m_y;  // Because it's a friend function
}
```

24

Friend Classes

- Classes can become friends of other classes
- Classes are friends of the whole class
- Class can use all members of the other class

25

FRIEND CLASSES

The previous example illustrates a typical use of a friend. The friend keyword can also be used for classes.

Friend Classes

```
class B;

class A
{
private:
	int a;
public:
	friend class B;
};

class B
{
private:
	int b;
public:
	void print(const A& obja);
};

void B::print(const A& obja)
{
	cout << b;
	cout << obja.a;
}
```

Friend Classes

```
class Point
{
private:
  double m_x;
  double m_y;

  friend class Other;
};

class Other
{
public:
  void Display(const Point& pt)
  {
    cout << pt.m_x << pt.m_y;
  }
};
```

26

Friend classes are used for tightly coupled classes. Take for example a binary tree consisting of nodes. We could create a class representing a node and a binary tree class being a friend class.

class Node

```
class Node
{
        Node* left;
        Node* right;
        friend Binary_Tree;
};
```

The member functions of the Binary_tree can access the left and right pointers of the node class, without using accessing functions.

We finish this section with some concluding remarks on friendship:
Friendship is not transitive, that is, if Y is a friend of X and Z is a friend of Y then it does not imply that Z is a friend of X.
Friendship is not inherited; doing so in the language would violate the Information Hiding Principle (However, it has consequences for software reusability, reimplementing the functions in derived classes).

- A function first declared in a friend declaration is equivalent to an extern declaration.
- A function can be a friend of two classes or more.
- A friend declaration must appear in the declaration of the class in which it is defined.
- A friend is as much a part of the interface of a class as the class members themselves.
- Friendship compromises the information hiding principle.

Binary Operators as Friend Functions

- Possible to use the data members
- More efficient
- Less object oriented

27

BINARY OPERATORS AS FRIEND FUNCTIONS

In the previous chapter we solved the problem of implementing operators which cannot be a part of any class. For example

Binary Operator Problem

```
Complex c1, c2, c3;

c3 = 10 + c1;
```

This problem was solved by implementing the binary operator+ as a global function. This global function had the disadvantage of not being a member function and thus not able to access the private members of the complex object which would make the function a bit simpler.

A solution to be able to access the private members of the complex object is to make the global function a friend function of the class Complex.

Friend Operator

```
class Complex
{
        friend Complex operator+(int value, const Complex& pt);
};
```

It is not mandatory but more efficient.

In the case of an operator+ with two different types of arguments, we can make the function friend of both classes.

Friend of More Classes

```
class Type1
{
	friend type operator+(const Type1& Oper1, const Type2& Oper2);
};

class Type2
{
	friend type operator+(const Type1& Oper1, const Type2& Oper2);
};
```

What if Int?

Kinds of Operator Overloading

- Mathematical: +, -, *, /
- Array indexing: [ ] (1 dimensional), () (multidimensional)
- Function call operator: () (looks just like a C function; called *function object/functor*)
- Relational operators: <, <=, ==, !=, etc.
- Input/output: <<, >>
- Exotic operators: , % ^ (if you like...)

28