

Overloading is considered “ad-hoc” polymorphism.

Can define new meanings (functions) of operators for specific types.

Compiler recognizes which implementation to use by signature (the types of operands used in the expression).

Overloading is already supported for many built-in types and operators:

```
17 * 42
4.3 * 2.9
cout << 79 << 'a' << "overloading is profitable" << endl;
```

The implementation used depends upon the types of operands.

Support natural, suggestive usage:

```
Complex A(4.3, -2.7), B(1.0, 5.8);  
Complex C;  
C = A + B;      // '+' means addition for this type as well
```

Semantic integrity (assignment for objects with dynamic content must ensure a proper deep copy is made).

Uniformity with built-in types (??)

Able to use objects in situations expecting primitive values

# Operators That Can Be Overloaded

Only the following operator symbols can be overloaded:

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

*Rule:* if you don't know what it means, don't overload it!

Declared and defined like other methods or functions, except that the keyword `operator` is used.

As method of the `Name` class:

```
bool Name::operator==(const Name& RHS) {  
    return ( (First + Middle + Last) ==  
            (RHS.First + RHS.Middle + RHS.Last) );  
}
```

As nonmember friend function:

```
bool operator==(const Name& LHS, const Name& RHS) {  
    return ( (LHS.First + LHS.Middle + LHS.Last) ==  
            (RHS.First + RHS.Middle + RHS.Last) );  
}
```

It is probably most natural here to use the member operator approach.

# Using Overloaded Operators

If `Name::operator==` defined as *member* function, then

```
nme1 == nme2
```

is the same as

```
nme1.operator==(nme2)
```

If `operator==` defined as *nonmember* function, then

```
nme1 == nme2
```

is the same as

```
operator==(nme1, nme2)
```

**With a binary operator, the LHS becomes the implicit object and the RHS the parameter.**

**No self-respecting C++ programmer would use either of these forms.**

A class member subtract operator for `Complex` objects:

```
Complex Complex::operator-(const Complex& RHS) const {  
    return ( Complex(Real - RHS.Real, Imag - RHS.Imag) );  
}
```

To be a class member, the left operand of an operator must be an object of the class type:

```
Complex X(4.1, 2.3), Y(-1.2, 5.0);  
OK:      X + Y  
Not OK:  cout << X;
```

It is typical to pass by constant reference to avoid the overhead of copying the object.

A non-member subtract operator for `Complex` objects:

```
Complex operator-(const Complex& LHS, const Complex& RHS) {  
    return ( Complex(LHS.getReal() - RHS.getReal(),  
                    LHS.getImag() - RHS.getImag()) );  
}
```

As a non-member, this subtract operator must use the public interface to access the private data members of its parameters...

... unless the class `Complex` declares it to be a friend .

If an operator or function is declared to be a `friend` of a class then it can access private members as if it were a member function.

Here, the class `Complex` declares a non-member arithmetic operator to be a `friend`:

```
class Complex {
private:
    double Real;
    double Imag;
public:
    Complex();
    // . . .
    bool    operator==(const Complex& RHS) const;
    // . . .
    friend Complex operator*(const Complex& LHS,
                             const Complex& RHS);
    // . . .
};
```

**The granting of friend status must be in the class declaration.**

**The keyword is NOT also used in the implementation.**

The declaration of a `friend` is unaffected by the access controls `public` and `private`.

**Bug note: Visual C++ does not handle friend declarations properly unless Service Pack 4 is installed.**

Here, the class `Complex` declares a non-member arithmetic operator to be a `friend`:

```
Complex operator*(const Complex& LHS,
                  const Complex& RHS) {

    double prodReal = LHS.Real * RHS.Real -
                      LHS.Imag * RHS.Imag;
    double prodImag = LHS.Real * RHS.Imag +
                      LHS.Imag * RHS.Real;

    return (Complex(prodReal, prodImag));
}
```

The accesses to private data members are now direct, even though `operator*` is NOT a member of the class `Complex`.

In general, `friend` status should be granted grudgingly. Most operators should be made members instead. However, this is not always possible...

# Unary Operators

A negation operator for the Complex class:

```
Complex Complex::operator-() const {  
    return ( Complex(-Real, -Imag) );  
}
```

**Note the use of the unary  
- operator for the built-in  
type double.**

```
Complex A(4.1, 3.2); // A = 4.1 + 3.2i  
Complex B = -A;      // B = -4.1 - 3.2i
```

Note that a unary member operator takes NO parameters.

# Prefix and Postfix Operators

A member prefix increment operator for the `DisplayableNumber` class:

```
DisplayableNumber DisplayableNumber::operator++() {  
    ++Count;  
    return DisplayableNumber(Count, Out);  
}
```

A member postfix increment operator for the `DisplayableNumber` class:

```
DisplayableNumber DisplayableNumber::operator++(int Dummy) {  
    Count++;  
    return DisplayableNumber(Count-1, Out);  
}
```

The parameter is necessary to make the signature of the postfix operator different from that of the prefix operator. Note the difference in the returned objects.

We can have two addition operators in a class:

```
Complex Complex::operator+(double RHS) const {  
    return (Complex(Real + RHS, Imag));  
}
```

This lets us write mixed expressions, like:

```
Complex X(4.1, 2.3);  
double R = 1.9;  
Complex Y = X + R; // Y.Real is 6.0
```

Signature of function used to resolve which is used:

```
Complex Z = Y + R; // complex plus double  
Complex W = Y + X; // complex plus complex
```

The compiler looks for a matching implementation of '+' in this order:

Member operator in X of form:

```
?? X::operator +(Y)
```

Nonmember operator of form:

```
?? operator +(X, Y)
```

The return type is not part of the lookup process.

In some cases, whole categories of operators make sense for a type.

For instance, it makes sense to overload all of the arithmetic operators for the class `Complex`. It also makes sense to overload all six relational operators for the class `Name`.

Often the implementation of one operator can "piggyback" off of another:

```
Complex Complex::operator+=(const Complex& RHS) {  
    *this = *this + RHS; // uses overloaded '+'  
    return (*this);  
}
```

For operators on primitive data types, such as:

```
Complex operator+(int LHS, const Complex& RHS);
```

When the source for the left operand of a binary operator is not available for modification:

E.g., ostream

For some type casting operations:

E.g.,

```
Complex::operator double() const  
{ return ( R ); }
```

We do not have access to the `istream` or `ostream` class code, so we cannot make overloadings of `<<` or `>>` members of those classes.

We also cannot make them members of a data class because the first parameter must then be an object of that type.

Therefore we must define `operator<<` as non-member function.

However, it must access private members of the data class, so we will typically make it a friend of that class. The alternative would be to have accessor functions for all the data members that will be written, and that is frequently unacceptable.

The general signature will be:

```
ostream& operator<<(ostream& Out, const Data& toWrite);
```

## operator<< for Complex Objects

This overloaded operator<< will write a nicely formatted Complex object to any output stream:

```
ostream& operator<<(ostream& Out, const Complex& toWrite) {  
  
    const int Precision = 2;  
    const int FieldWidth = 8;  
    Out << setprecision(Precision);  
  
    Out << setw(FieldWidth) << toWrite.Real;  
  
    if (toWrite.Imag >= 0)  
        Out << " + ";  
    else  
        Out << " - ";  
  
    Out << setw(FieldWidth) << fabs(toWrite.Imag);  
    Out << "i";  
    Out << endl;  
  
    return Out;  
}
```

**The formatting settings could also be added as data members of Complex, or...**

## operator>> for Complex Objects

This overloaded operator>> will read a complex number formatted in the manner used by operator<<:

```
istream& operator>>(istream& In, Complex& toRead) {  
  
    char signOfImag;  
  
    In >> toRead.Real;  
    In >> signOfImag;  
    In >> toRead.Imag;  
  
    if (signOfImag == '-')  
        toRead.Imag = -toRead.Imag;  
  
    In.ignore(1, 'i');  
    return In;  
}
```

**Of course, this depends on knowing exactly how the `Complex` objects are formatted in the input stream.**

**We could make this a lot more complicated if we had multiple formats to deal with.**

```
class Array {
private:
    int      Capacity;      // dimension of array
    int      Usage;        // number of cells in use
    string*  List;         // the list

    void ShiftTailUp(int Start);
    void ShiftTailDown(int Start);

public:
    Array(int initCapacity = 100, string Init = "");
    Array(const Array& Source);
    Array& operator=(const Array& Source);

    int  getCapacity() const;
    int  getUsage() const;

    bool isFull() const;
    bool isEmpty() const;

    bool Insert(const string& newValue);
    // . . . continues . . .
};
```

```
// . . . continued . . .  
  
bool DeleteAtIndex(int Idx);  
bool Delete(const string& toDel);  
  
int Find(const string& toFind) const;  
  
string& operator[](int Idx);  
const string& operator[](int Idx) const;  
  
friend ostream& operator<<(ostream& Out, const Array& toPrint);  
  
~Array();  
};
```

The first overloading of operator[ ]:

```
string& Array::operator[](int Idx) {  
  
    assert(0 <= Idx && Idx < Capacity);  
    return List[Idx];  
}
```

The first is the most commonly used. It provides the functionality you'd normally expect, allowing code like:

```
Array A(10);  
// . . .  
A[5] = "a string";  
cout << A[5] << endl;
```

The operator returns a reference to an element of the array, potentially dangerous, but also useful.

However, there's an important "gotcha" with the code above...

## Overloading operator[ ]

The second overloading of operator[ ]:

```
const string& Array::operator[](int Idx) const {  
    assert(0 <= Idx && Idx < Capacity);  
    return List[Idx];  
}
```

This is provided only to support cases where a constant return value is required. These are rare. Can you think of one?

Avoid violating expectations about the operator:

```
Complex Complex::operator~() const {  
    return ( Complex(Imag, Real) );  
}
```

Provide a complete set of properly related operators:  $a = a + b$  and  $a += b$  have the same effect and it makes sense to support both if either is supplied.

Define the operator overload as a class member unless it's necessary to do otherwise.

If the operator overload cannot be a class member, then make it a friend rather than add otherwise unnecessary member accessors to the class.