Operator Overloading in C++

Questions:

1. With regard to the meaning of the assignment operator '=' in Java, what can you say about the objects for which the variables u1 and u2 are holding references?

```
        User u1 = new User( .... );
        User u2 = u1;
```

2. How do we create a duplicate of an object in Java?

3. Is there anything wrong with this code?

```
class X implements Cloneable {

    // some data members are class types

    public X( ...  ) { .... }

    public X clone() {
        //  call super.clone()
        //  do something with the exception
        //     thrown by super.clone()
        //  clone class type data members individually
    }
}
```

4. Is there anything wrong with this code?
We have fixed the return type, but changed the access specifier. Note that the access specifier of Object.clone() is protected?

```
class X implements Cloneable {

    // some data members are class type

    public X( ...  ) { .... }

    private Object clone() {
        // call super.clone()
        // do something with the exception thrown
        //      by super.clone()
        //  clone class type data members individually
    }
}
```

5. Is there anything wrong with this code?
Note that Object.clone() throws
     CloneNotSupportedException
The clone of X throws Exception.

```
class X implements Cloneable {

    // has class type data members

    public X( ...  ) { .... }

    public Object clone() throws Exception {
        //  call super.clone()
        //  clone class type data members individually
    }
}
```

6. Is there anything wrong with this code?

```
class X {

    // has class type data members

    public X( ...  ) { .... }

    public Object clone()
        throws CloneNotSupportedException {
          //  call super.clone()
          //  clone class type data members individually
    }
}
```

7. Is there anything wrong with this code?

```
class X implements Cloneable {

    // has class type data members

    public X( ...  ) { .... }

    public Object clone() {
        try {
            super.clone();
        } catch( CloneNotSupportedException e ) {
            e.printStackTrace();
        }
        //  clone class type data members individually
    }
}
```

Operator Overloading in C++

You have already seen numerous examples of overloaded operators in C++ and Java. For example, when you say in Java

```
    String str = "hello" + "there";
```

or in C++

```
    string str = "hello" + "there";
```

you are using the overloaded definition of the '+' operator.

While you will find overloaded operators in both C++ and Java, you are not allowed to create your own operator overloadings in Java.

When an operator is overloaded in C++, at least one of its operands must be of class type or enumeration type.

Additionally, at least one of the operands must not be a built-in type. This keeps a programmer from, say, redefining the meaning of, for example, the '+' operator for the `int` type.

It is also useful to remember that the predefined precedence of an operator cannot be altered by an overload definition.

An overload definition also cannot alter the predefined arity of an operator, meaning that an operator that is designated to be only unary (for example, the logical NOT operator '!') cannot be turned into a binary operator by overloading.

For operators that can be used for both unary and binary operations (these being the four operators + - * and &), both arities can be overloaded.

Arguments can be passed to the parameters in the overload definitions either by value or by reference, but not by pointer.

Default arguments for overloaded operators are illegal, except for the operator `operator()`.

Operator Tokens and Operator Functions

```
 +   -   *   /   =   ->   &&   new   new[]   delete   delete[]
```

When the arguments supplied to any of these and other operators are of class type, what the compiler actually invokes is an *operator function* that is associated with the operator token.

For example, for the case of class type arguments, associated with the operator token + is the function with the following operator function

```
    operator+
```

Similarly, associated with the operator token, or operator for short, = is the operator function

```
    operator=
```

and with the operator << the operator function

```
    operator<<
```

Overload definitions for operator functions come in two forms:

Definitions that are global and definitions that are member functions for a class.

An overloaded operator works the same way regardless of which way its overload definition is implemented.

Global Overload Definitions for Operators

```
class MyComplex {
  double re, im;
public:
  MyComplex( double r, double i ) : re(r), im(i) {}
  double getReal() { return re; }
  double getImag() { return im; }
};
```

```
MyComplex c1( 3, 5 );
MyComplex c2( 1, 4 );
```

```
MyComplex sum = c1 + c2;
MyComplex diff = c1 - c2;
...
...
```

```
MyComplex operator+( MyComplex arg1, MyComplex arg2 ) {
  double d1 = arg1.getReal() + arg2.getReal();
  double d2 = arg1.getImag() + arg2.getImag();
  return MyComplex( d1, d2 );
}
```

```
MyComplex operator-( MyComplex arg1, MyComplex arg2 ) {
  double d1 = arg1.getReal() - arg2.getReal();
  double d2 = arg1.getImag() - arg2.getImag();
  return MyComplex( d1, d2 );
}
```

```
ostream& operator<< ( ostream& os, const MyComplex& arg ) {
  double d1 = arg.getReal();
  double d2 = arg.getImag();
  os << "(" << d1 << ", " << d2 << ")" << endl;
  return os;
}
```

```
    cout << expr1 << exp2 << exp3  ..
```

```
//MyComplex.cc

#include <iostream.h>
using namespace std;

class MyComplex {
  double re, im;
public:
  MyComplex( double r, double i ) : re(r), im(i) {}
  double getReal() const { return re; }
  double getImag() const { return im; }
};


MyComplex operator+( const MyComplex arg1, const MyComplex arg2 ) {
  double d1 = arg1.getReal() + arg2.getReal();
  double d2 = arg1.getImag() + arg2.getImag();
  return MyComplex( d1, d2 );
}


MyComplex operator-( MyComplex arg1, MyComplex arg2 ) {
  double d1 = arg1.getReal() - arg2.getReal();
  double d2 = arg1.getImag() - arg2.getImag();
  return MyComplex( d1, d2 );
}


ostream& operator<< ( ostream& os, const MyComplex& arg ) {
  double d1 = arg.getReal();
  double d2 = arg.getImag();
  os << "(" << d1 << ", " << d2 << ")" << endl;
  return os;
}
```

```
int main()
{
  MyComplex first(3, 4);
  MyComplex second(2, 9);

  cout << first;                      // (3, 4)
  cout << second;                     // (2, 9)
  cout << first + second;             // (5, 13)
  cout << first - second;             // (1, -5)
}
```

Member-Function Overload Definitions for Operators

```
//MyComplexMem.cc
#include <iostream>
using namespace std;

class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    MyComplex operator+( MyComplex) const;
    MyComplex operator-( MyComplex) const;
    //  ostream& operator<< ( ostream& os );                  WRONG
    //  ostream& operator<<( ostream&, MyComplex& );          WRONG
    friend ostream& operator<< ( ostream&, const MyComplex& );
};


MyComplex MyComplex::operator+( const MyComplex arg ) const {
    double d1 = re + arg.re;
    double d2 = im + arg.im;
    return MyComplex( d1, d2 );
}


MyComplex MyComplex::operator-( const MyComplex arg ) const {
    double d1 = re - arg.re;
    double d2 = im - arg.im;
    return MyComplex( d1, d2 );
}



/*
ostream& MyComplex::operator<< ( ostream& os ) {              // WRONG
  os << "Real part: " << re << "  Imag part: " << im << endl;
  return os;
}
```

```
ostream& MyComplex::operator<< ( ostream& os, MyComplex c ) {  // WRONG
  os << "Real part: " << c.re << "  Imag part: " << c.im << endl;
  return os;
}
*/


ostream& operator<< ( ostream& os, const MyComplex& c ) {
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}


int main()
{
    MyComplex first(3, 4);
    MyComplex second(2, 9);

    cout << first;                          // (3, 4)
    cout << second;                         // (2, 9)
    cout << first + second;                 // (5, 13)
    cout << first - second;                 // (1, -5)
}
```

```
    Complex MyComplex::operator+( const MyComplex arg ) const {
        double d1 = re + arg.re;
        double d2 = im + arg.im;
        return MyComplex( d1, d2 );
    }
```

Given

```
    MyComplex c1( 3, 5);
    MyComplex c2( 2, 4);
```

The call

```
    MyComplex result =  c1 + c2;
```

gets translated by the compiler into

```
    MyComplex result =  c1.operator+( c2 );
```

Therefore, we can think of `c1` as the invoking object and `c2` as the argument object.

So the important thing to bear in mind is that for overloaded member-function operators, one of the arguments is supplied implicitly by the object that invokes the operator and that in a call such as

In general, a construct such as

```
object1  χ  object2
```

in which χ is a member-function overloaded operator,

is interpreted by the compiler as

```
object1.operator χ ( object2 )
```

implying that `object1` is the invoking object for the operator χ.

On the other hand, if the operator χ was overloaded with a global definition, the construct

```
 object1  χ  object2
```

would be interpreted by the compiler as

```
operatorχ ( object1, object2 )
```

This implies that the global overloaded operator

```
    ostream& operator<< ( ostream& os, const Complex& arg )
    {
        double d1 = arg.getReal();
        double d2 = arg.getImag();
        os << "Real part: " << d1 << "  Imag part: " << d2 << endl
        return os;
    }
```

cannot be replaced by something like

```
    ostream& Complex::operator<< ( ostream& os ) {                /
        os << "Real part: " << re << "  Imag part: " << im << endl
        return os;
    }
```

or by something like this

```
    ostream& Complex::operator<< ( ostream& os, Complex c ) {    /
        os << "Real part: " << c.re << "  Imag part: " << c.im <<
        return os;
    }
```

Since the output stream operator << is used in the manner

```
    cout << expression;
```

which means that the left object will be the invoking object for this operator if it were defined for member-function overloading.

```
    friend ostream& operator<< ( ostream&, Complex& );
```

This declaration, which can be either in the private section or in the public section, allows us to define the overloading by

```
    ostream& operator<< ( ostream& os, Complex& c ) {
        os << "Real part: " << c.re << "  Imag part: " << c.im <<
        return os;
    }
```

It is important to note that if we had not declared `operator`<< to be a friend of `Complex`, the body of this function would have had no access to the `re` and `im` members of the parameter `c`, being private as they are.

Global Overload Definitions for Unary Operators

Just as is the case with binary operators, if the *arity* of an operator is unary with respect to built-in types, then such an operator can be overloaded for the user-defined types.

As before, this overloading may be accomplished either by supplying a global overload definition or a member-function overload definition.

Consider again the case of the `MyComplex` class, with just the unary operator '-' defined this time, and the previously presented overloaded binary operator << so that we can see the results of applying the unary operator to a `MyComplex` type.

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

class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    double getReal() const { return re; }
    double getImag() const { return im; }
};


// global overload definition for "-" as a unary operator
MyComplex operator-( const MyComplex arg ) {
    return MyComplex( -arg.getReal(), -arg.getImag() );
}


// global overload definition for "<<" as a binary operator
ostream& operator<< ( ostream& os, const MyComplex& arg ) {
    double d1 = arg.getReal();
    double d2 = arg.getImag();
    os << "(" << arg.getReal() << ", " << arg.getImag() << ")" << endl;
    return os;
}


int main()
{
    MyComplex c(3, 4);

    cout << c;                // (3, 4)
    cout << -c;               // (-3, -4)
}
```

Member-Function Overload Definitions for Unary Operators

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


class MyComplex {
    double re, im;
public:
    MyComplex( double r, double i ) : re(r), im(i) {}
    MyComplex operator-() const;
    friend ostream& operator<< ( ostream&, MyComplex& );
};


//Member-function overload definition for "-"
MyComplex MyComplex::operator-() const {
    return MyComplex( -re, -im );
}


//This overload definition has to stay global
ostream& operator<< ( ostream& os, MyComplex& c ) {
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}


int main()
{
    MyComplex c(3, 4);

    MyComplex z = -c;

    cout << z << endl;                  // (-3, -4)
}
```

The statement

```
    Complex z = -c;
```

is translated by the compiler into

```
    Complex z = c.operator-();
```

which makes `c` the invoking object. Thus the values of `re` and `im` available inside the definition of the `operator-` function would then correspond to the `Complex` number `c`.