ECE 462
Object-Oriented Programming using C++ and Java

Lecture 12

Yung-Hsiang Lu
yunglu@purdue.edu

```
class User {
    String name;
    int age;
    User(String nm, int a) {name=nm;
      age=a;}
    public String toString()
    {
        String str = name + " " + age;
        return str;
    }
    public void func1(final User usr)
    {
        // usr = new User("Charlie", 8);
        // usr.name = "David";
        // name = usr.name;
        // func2(usr);
    }
    public void func2(User usr)
    {
        // usr = new User("Edward", 61);
        // usr.name = "Frank";
        name = usr.name;
    }
}

class FinalMethod {
    public static void main(String[] args)
    {
        User u1 = new User("Amy", 95);
        User u2 = new User("Bob", 98);
        u2.func1(u1);
        System.out.println(u1);
        System.out.println(u2);
    }
}
```

```
#include <iostream>
#include <string>
using namespace std;
class User {
private:
  string name;
  int age;
public:
  User(string nm, int a) {name=nm;
      age=a;}
  void print(void) const
  {
    cout << "name = " << name << " age = "
      << age << endl;
    // name = "Tom";
  }
  void func1(const User usr) const
  {
    // usr.name = "David";
    // name = usr.name;
    func2(usr);
  }
  void func2(User usr)
  {
    // usr.name = "Frank";
    // name = usr.name;
  }
  void func3(const User & usr) const
  {
    // usr.name = "David";
    // name = usr.name;
    // func2(usr);
  }
};
int main(int argc, char * argv[])
{
  User u1("Amy", 95);
  User u2("Bob", 98);
```

```
  u2.func1(u1);
  u2.func2(u1);
  u2.func3(u1);
  u1.print();
  u2.print();
  return 0;
}
```

final in Java

- disallow inheritance

```
final class ClassA { ... }
class ClassB extends ClassA // error
```

- constant (usually with public static) // similar to C++ static

```
class Math {
        public static final double PI = 3.14159;
}
...
Math.PI // do not have to create an object
Color.WHITE
```

Overloading (by Parameters Types)

Overloading: same function name, same return type, different input parameters (numbers, types, or both)

example: Java Color class has 7 constructors

- **Color** (ColorSpace cspace, float[] components, float alpha)
  Creates a color in the specified ColorSpace with the color components specified in the float array and the specified alpha. (alpha: transparency)
- **Color**(float r, float g, float b)
  Creates an opaque sRGB color with the specified red, green, and blue values in the range (0.0 - 1.0).

- **Color**(float r, float g, float b, float a)
  Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0.0 - 1.0).
- **Color**(int rgb)
  Creates an opaque sRGB color with the specified combined RGB value consisting of the red component in bits 16-23, the green component in bits 8-15, and the blue component in bits 0-7.
- **Color**(int r, int g, int b)
  Creates an opaque sRGB color with the specified red, green, and blue values in the range (0 - 255).

  ... 2 more constructors ... check

http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Color.html

Why to overload? convenience: you may have different types of information
frequently used in constructors

Overload Resolution
(how to find the right function to call)

- If there is an exact match, the function is called.
  - (int, double, User) → f(int, double, User)
  - (String, User, Color) → f(String, User, Color)
- If there is no exact match, the compiler will find the "closest" match by
  - type conversion through promotion (no information loss), eg. short → int, char → int, float → double
  - not const → const
  - C++: standard conversion (information may be lost), eg . double → int, int → double
  - derived class → base class

Overload Resolution

- determined at compilation time, not run-time
- C++ allows default values, as one format of overloading

```
func(int x, int y = 8) { ... }
func(5); // same as calling func(5,8);
func(5, -11); // x = 5 and y = -11
// default values always start from the last parameter
func(int x = 3, int y) { ...  } // error
func(int x, double y = 0.0, int z = -10) { ... }
```

```java
//Overload.java
class Employee { String name; }
class Manager extends Employee { int level; }
class Test {
    static void foo( Employee e1, Employee e2 ) {
        System.out.println( "first foo" );
    }
    static void foo( Employee e, Manager m ) {
        System.out.println( "second foo" );
    }
    static void foo( Manager m, Employee e) {
        System.out.println( "third foo" );
    }
```

```
public static void main( String[] args )
{
    Employee emp = new Employee();
    Manager man = new Manager();
    foo( emp, man );      // will invoke the second foo
    //foo( man, man );    // Error because of ambiguity in
                          // overload resolution (2nd and 3rd)
 }
}
```

Overload cannot be resolved by the return type.

Examples of Overloading

```
func(int x, int y, int z)
{ ... } // func1

func(int x, int y, double z)
{ ... } // func2

func(String str)
{ ... } // func3

func(int x, String str)
{ ... } // func4
```

```
int x = 3;
int y = 4;
int z = 5;
double w = 0.6;
String str = “hello”;

func(x, y, z); // func1
func(x, y, w); // func2
func(str); // func3
func(x, str); // func4
```

Handle (Detect or Prevent) Problem

- assert (something must be true);
  - Your program crashes when it is not true.
  - Do **not** use assert. Instead, you should **handle** the situation when it is false.
- always check the return value of function calls
  - memory allocation
  - open a file
  - send data through network
  - ...

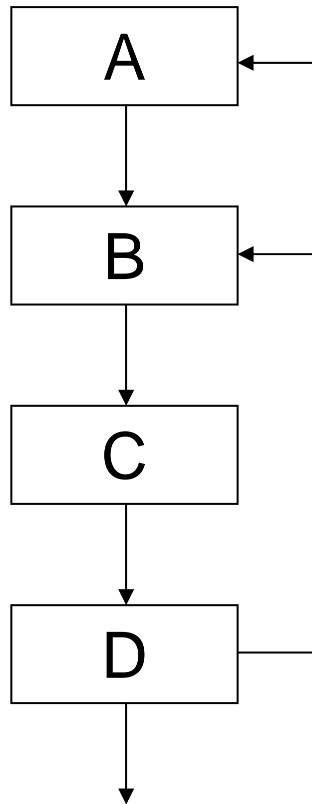
Handle Problems

- A typical approach is to check before proceeding:

```
retval = func(parameters);
if (retval < 0) {
  ...
}
```

- However, sometimes the immediate caller does not know how to handle the problem. This is especially common for reused code.

Call Stack

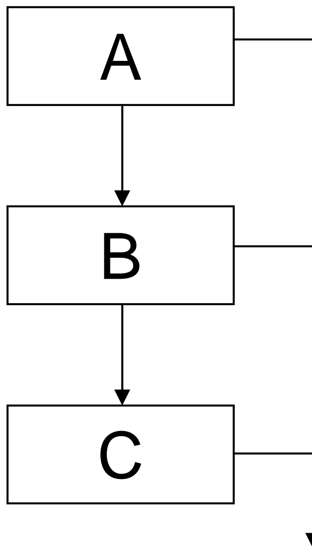


- Should D create the file?
- Can D make that decision?
- Maybe, A intends to ask the user to give a different file name.
- Maybe, B wants to simply skip the file.

$\Rightarrow$ need a way to inform a caller that is several "frames" away

Concepts of Exception Handling

try {



} catch (Exception) {

handle exception;

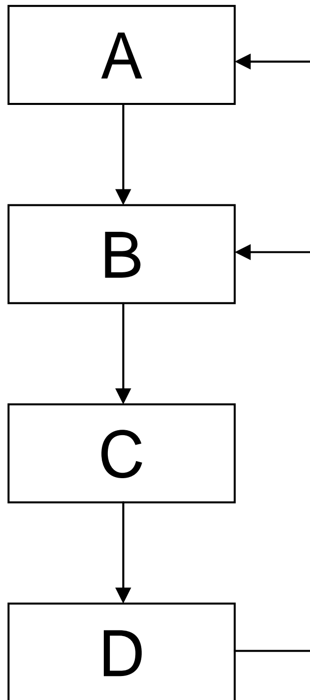
}

- A, B, or C can throw an exception and it will be caught by

  try { ... } catch

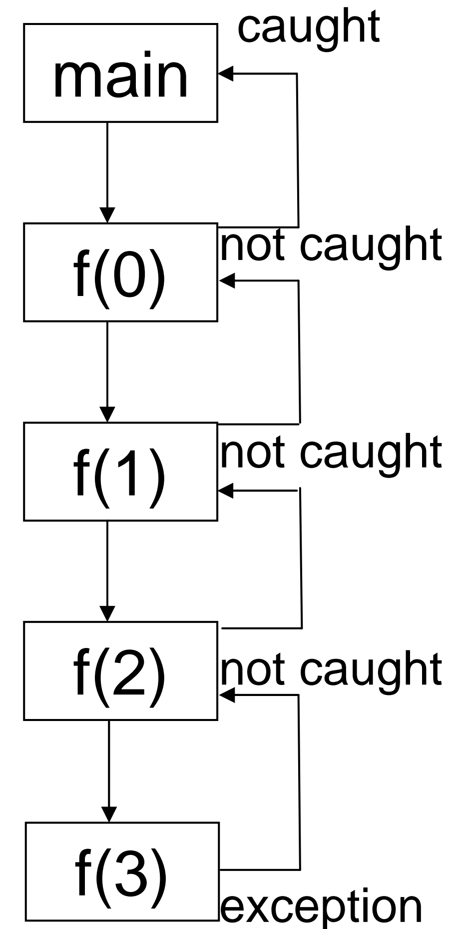
- If an exception is not caught, the program terminates (like "crash").

Catching an Exception



- An exception is caught by the closest caller in the stack.
- If both A and B catch the same exception and D throws and exception, B will catch it.
- The handling code **can throw** the exception **again** to its call stack; the code can also throw a **different** exception.
- An exception may pass **parameters** through the call stack.

```
//TryCatch.java
import java.io.*;
class Err extends Exception { }
class Test {
    public static void main( String[] args )
    {
        try {
            f(0);
        } catch( Err e ) {
            System.out.println( "Exception caught in main" );
        }
    }
    static void f(int j) throws Err {
        System.out.println( "function f invoked with j = " + j );
        if (j == 3) throw new Err();
        f( ++j );
    }
}
```



```
//TryCatch.cc
#include <iostream>
#include <cstdlib>
using namespace std;
class Err {}; // not a derived class of any particular class
void f(int j) { // does not declare that it may throw an exception
    cout << "function f invoked with j = " << j << endl;
    if (j == 3) throw Err();
    f( ++j );
}
int main()
{
    try {
        f(0);
    } catch( Err ) {
        cout << "caught Err" << endl;
        exit(0); // you don't need this
    }
    return 0;
}
```

How to pass information across call stack

```
//ExceptionUsage9.cc
#include <iostream>
#include <string>
using namespace std;
class MyException {
    string message;
public:
    MyException( string m ) { message = m; }
    string getMessage() { return message; }
};
void f() throw( MyException );
int main(){
    try {
      f();
    } catch( MyException ex ) { cout << ex.getMessage(); }
    return 0;
}
void f() throw( MyException ) { throw MyException( "hello there" );}
```

```java
//ExceptionUsage7.java
class MyException extends Exception {
    String message;
    public MyException( String mess ) { message = mess; }
}
class Test {
    static void f() throws MyException {
        throw new MyException( "Hello from f()" );
    }
    public static void main( String[] args )
    {
        try {
            f();
        } catch( MyException e ) {
            System.out.println( e.message );
        }
    }
}
```

Exceptions in C++ and Java

- C++ can throw any type (including integer); Java can only throw derived classes of Exception
- In Java, a function has to explicitly declare that it may throw an exception.
- A function that may throw an exception should be called only within a try-catch block
- In Java, catch must identify the object.
- A function can throw different types of exceptions.

```
//ExceptionUsage3.java
class MyException extends Exception {}
class Err extends Exception {}
class Test {
    static void f( int j ) throws MyException, Err {
        if ( j == 1 ) throw new MyException();
        if ( j == 2 ) throw new Err();
    }
    public static void main( String[] args )
    {
        try {
            f( 1 );
        } catch( MyException e ) {
            System.out.println("caught MyException -- arg must be 1");
        } catch( Err e ) {
            System.out.println("caught Err -- arg must be 2");
        }
```

```
 try {
        f( 2 );
      } catch( MyException e ) {
        System.out.println("caught MyException -- arg must be 1");
      } catch( Err e ) {
        System.out.println("caught Err -- arg must be 2");
      }
   }
}
```

```
//ExceptionUsage4.java
class MyException extends Exception {}
class Err extends Exception {}

class Test {
   static void f() throws MyException {
      throw new MyException();
   }
   static void g() throws Err {
      throw new Err();
   }
   static void h() throws MyException, Err { // h may throw either type
      f();
      g();
   }
```

```
 public static void main( String[] args ) {
        try {
            h();
        } catch( MyException e ) {
            System.out.println( "caught MyException" );
        } catch( Err e ) {
            System.out.println( "caught Err" );
        }
    }
}
```

```
//ExceptionUsage5.java
import java.io.*;
class Test {
    static void foo() throws Exception { throw new Exception(); }
    static void bar() throws Exception {
        FileReader input = null;
        try {
            input = new FileReader( "infile" );
            int ch;
            while ( ( ch = input.read() ) != -1 ) {
                if ( ch == 'A' ) { System.out.println( "found it" ); foo(); }
            } finally { // execute regardless whether an exception is thrown
            if ( input != null ) {
                input.close();
                System.out.println("input stream closed successfully");
            }
        }
        System.out.println( "Exiting bar()" );
    }
```

```
 public static void main( String[] args ) {
        try {
            bar();
        } catch( Exception e ) {
            System.out.println( "caught exception in main" );
        }
    }
}
```

Exception after Being Caught

```
try {
        f();
    } catch( MyException e ) {
        System.out.println( "catching and re-throwing in g" );
        throw new MyException();
    }
```

Exception and Class Hierarchy
(15.9.4 and 15.15.3)

If a method throws an exception of type X; the overriding method in a derived class can throw an exception of type X or its derived classes.

```
class Base {
        virtual void foo () throw (E1) { …}
};
class Derived: public Base {
        void foo() throw (E2) { … } // E2 = E1, or E2 is derived from E1
};
```

Overhead of Exception Handling

- Do not overuse exceptions.
- In many cases, if the errors can be detected by the return code

```
retval = function(...);
if (retval < 0) { ... /* handle error */ }
```

  then, you should do so without using try-catch.
- Detect problems early, instead of using exceptions. For example, check whether network connection is valid before sending data (and catch exception when the sending fails).
- Catch an exception early and prevent its propagation.

ECE 462
Object-Oriented Programming using C++ and Java

Lecture 13

Yung-Hsiang Lu
yunglu@purdue.edu

Self Reference (11.6 and 11.7)

How to Cascade Methods?

Can we call the same methods repetitively without creating intermediate objects?
object.method1(parameters).method2(parameters)

must be an object

consider x = 3 + 5 + 19 + 28;

```
//SpecialInt.java
class SpecialInt {
    int i;
    int accumulator;
    SpecialInt( int m ) throws Exception {
        if ( m > 100 || m < -100 ) throw new Exception();
        i = m;
        accumulator = m;
    }
    int getI() { return i; }
```

```
SpecialInt plus( SpecialInt sm ) throws Exception {
    accumulator += sm.getI();
    if ( accumulator > 100 || accumulator < -100 )
      throw new Exception();
    return this;
  }
  public static void main( String[] args ) throws Exception {
    SpecialInt s1 = new SpecialInt( 4 );
    SpecialInt s2 = new SpecialInt( 5 );
    SpecialInt s3 = new SpecialInt( 6 );
    SpecialInt s4 = new SpecialInt( 7 );
    s1.plus( s2 ).plus( s3 ).plus( s4 );
    System.out.println( s1.accumulator );      // 22
    //SpecialInt s5 = new SpecialInt( 101 );   // range violation
  }
}
```

```
//SpecialInt.cc
#include <iostream>
using namespace std;
class Err {};
class SpecialInt {
   int i;
public:
   int accumulator;
   SpecialInt( int m ) {
      if ( m > 100 || m < -100 ) throw Err();
      i = m;
      accumulator = m;
   }
   int getI() { return i; }
   SpecialInt& plus( SpecialInt m );
};
```

```
SpecialInt& SpecialInt::plus( SpecialInt sm ) {
    accumulator += sm.getI();
    if ( accumulator > 100 || accumulator < -100 ) throw Err();
    return *this;
}
int main()
{
    SpecialInt s1( 4 );
    SpecialInt s2( 5 );
    SpecialInt s3( 6 );
    SpecialInt s4( 7 );
    s1.plus( s2 ).plus( s3 ).plus( s4 );
    cout << s1.accumulator << endl;        // prints out 22
    // SpecialInt s5( 101 );               // range violation
    return 0;
}
```

No Copy Constructor in Java

Java's garbage collection makes shallow / deep copy a less-relevant problem.

```
//AssignTest.java
class User {
    public String name;
    public int age;
    public User( String str, int n ) { name = str; age = n; }
}
class Test {
    public static void main( String[] args )
    {
        User u1 = new User( "ariel", 112 );
        System.out.println( u1.name );      // ariel
        User u2 = u1;
        u2.name = "muriel";
        System.out.println( u1.name );      // muriel
    }
}
```

Object Cloning in Java

without it CloneNotSupportedException

```
//ClonableX.java
class X implements Cloneable {
   public int n;
   public X() { n = 3; }
   public static void main( String[] args )
   {
      X xobj = new X();
      X xobj_clone = null;
      try {
         xobj_clone = (X) xobj.clone();
      } catch (CloneNotSupportedException e){}
```

Can we use xobj_clone = new X(xobj)? You need to implement deep copy yourself.

```
        System.out.println( xobj.n );              // 3
        System.out.println( xobj_clone.n );        // 3
        xobj_clone.n = 3000;
        System.out.println( xobj.n );              // 3
        System.out.println( xobj_clone.n );        // 3000 (different)
    }
}
```

```
class X {
   public int n;
   public X() { n = 3; }
   public Object clone() throws CloneNotSupportedException
    {       n = 3;     return this;     }
   public static void main( String[] args )
   {
      X xobj = new X();
      X xobj_clone = null;
      try {   xobj_clone = (X) xobj.clone();   }
      catch (CloneNotSupportedException e)
           { System.out.println("clone error"); }
```

```
        System.out.println( xobj.n );               // 3
        System.out.println( xobj_clone.n );         // 3
        xobj_clone.n = 3000;
        System.out.println( xobj.n );               // 3
        System.out.println( xobj_clone.n );         // 3000
    }
}
```

```
class X implements Cloneable {
   public int[] arr = new int[5];
   public X() {
      Random ran = new Random();
      int i=0;
      while ( i < 5 )
         arr[i++] = (ran.nextInt() & 0xffff)%10;
   }
   public Object clone() throws CloneNotSupportedException {
      X xob = null;
      xob = (X) super.clone();
      //now clone the array separately:
      xob.arr = (int[]) arr.clone();
      return xob;
   }
```

If a class contains object attributes, these objects' classes must also implement Cloneable.

Lab 7: Source Code Version Control in eclipse and netbeans

Version Control

- For any nontrivial project, version control is very important for maintaining the history of all changes.
- Version control also keeps messages for each version.
- You can roll back to an earlier version and compare the changes between versions.
- Version control allows “branches”: one for release without any new feature, meanwhile, another for the next release.
- Version control maintains a central repository and also allows multiple users to work concurrently. Version control helps team management.

- log into your account
- mkdir ECE462
- cd ECE462
- mkdir CVSROOT
- netbeans: right click the project, CVS, Import into Repository
- eclipse: Team, Share Project
- use cvs2cl.pl (http://www.red-bean.com/cvs2cl/) to create ChangeLog

Operator Overloading in C++

You have been using overloaded operators for years.

- 3 + 5 // integer addition (bit-wise + carries)
- 3.5 + 0.0059 // floating point addition
  // 1. align the decimal points
  // 2. add significands
  // 3. normalized and update exponent
- “hello” + “ world” // intuitively, it means append
- Operator overloading is not essential in object-oriented programming. In fact, Java does not allow operator overloading.

Overloading Operators in C++

- We have seen overloaded operator for << and >>.
- at least one operand (for binary operators) must be an object or enumeration type (i.e. not a built-in type)
- precedence not changed
- arity not changed (! always unary)
- argument(s) may be passed by value (copy) or by reference, not by pointer
- default argument value(s) illegal
- cannot overload :: .* . ?:

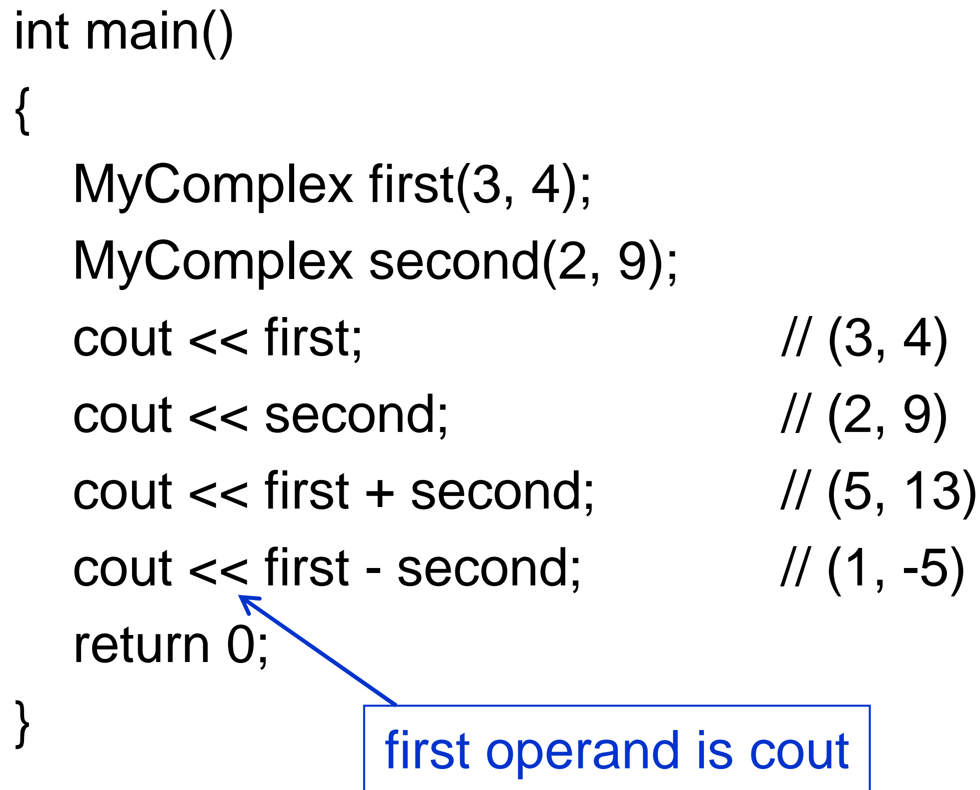
Binary Operators

- If the first operand is an object, a binary operator can be implemented in two forms
  - a member function, the first operand is ***this*** object
  - a “free” function (not a member of any class), usually declared as a **friend** to access private attributes
  - not both
- If the first operand is not an object (such as int), the operator must be a free function.
- In most cases, the operand object(s) should use reference to prevent calling copy constructor.

```
//OverloadBinaryGlobal.cc
#include <iostream>
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 ) {
    // not a member function
   double d1 = arg1.getReal() + arg2.getReal();
   double d2 = arg1.getImag() + arg2.getImag();
   return MyComplex( d1, d2 ); // create a new object
}
```

```
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)
    return 0;
}
```



```
//OverloadBinaryMemb.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<< ( const MyComplex& );            // WRONG
   friend ostream& operator<< ( ostream&, const MyComplex& );
    // friend can access private attributes (re and im)
};
```

```
MyComplex MyComplex::operator+( const MyComplex arg ) const {
    double d1 = re + arg.re;        // does not change re or im
    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& operator<< ( ostream& os, const MyComplex& c ) {         //(F)
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}
```

```
//OverloadUnaryGlobal.cc
#include <iostream>
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)
    return 0;
}
```

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

    cout << first;                  // (3, 4)
    cout << second;                 // (2, 9)
    cout << first + second;       // (5, 13), think of it as first .+ (second)
    cout << first - second;       // (1, -5)
    return 0;
}
```

```
//OverloadUnaryMemb.cc
#include <iostream>
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&, const MyComplex& );
};
//Member-function overload definition for "-"
MyComplex MyComplex::operator-() const {                    //(A)
   return MyComplex( -re, -im );
}
```

```
//This overload definition has to stay global
ostream& operator<< ( ostream& os, const MyComplex& c ) {
    os << "(" << c.re << ", " << c.im << ")" << endl;
    return os;
}
int main()
{
    MyComplex c(3, 4);
    MyComplex z = -c;
    cout << z << endl;                    // (-3, -4)
    return 0;
}
```

Performance Comparison

- An operator may be implemented in several different ways:
  - not a member function
    - as a friend to access private attributes directly
    - not a friend and call getReal / getImag function
  - member function
    - second parameter using reference
    - second parameter using object (calling copy constructor for passing the parameter)
- They have the same functionality but different performance.

```
class MyComplex {
  friend MyComplex operator + ( const MyComplex arg1, const MyComplex
     arg2 );
  ....
}
MyComplex operator + ( const MyComplex & arg1,  const MyComplex & arg2 )
 {
  double d1 = arg1.re + arg2.re;

/* not a friend, no need to change the class definition  */
MyComplex operator + ( const MyComplex & arg1,  const MyComplex & arg2 )
{
  double d1 = arg1.getReal() + arg2.getReal();
```

as Member Function

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

reference or not

member	friend	reference	execution time
-	-	-	3.43
-	Y	Y	0.86
Y	-	Y	0.88
Y	-	N	1.36

Template + Container + Overloading

```
#include <iostream>
#include <string>
using namespace std;
template <class TPL> class BinaryNode
{
  BinaryNode * bn_left;
  BinaryNode * bn_right;
  TPL bn_content;
public:
  BinaryNode(TPL content);
  ~ BinaryNode();
  void insertContent(TPL content);
  bool searchContent(TPL content) const;
  void print(int depth) const;
};
```

```
template <class TPL>
    BinaryNode<TPL>::BinaryNode(TPL
    content)
{
  bn_content = content;
  bn_left = NULL;
  bn_right = NULL;
  bn_right = NULL;
}
template <class TPL> void
    BinaryNode<TPL>::insertContent(TPL
    content)
{
  if (content == bn_content) { return; }
  // no duplicate
```

```
if (content < bn_content)
 {
  if (bn_left == NULL)
  {
   BinaryNode<TPL> * newnode = new
  BinaryNode<TPL>(content);
   bn_left = newnode;
  }
  else
  { bn_left -> insertContent(content); }
 }
else
 {
  if (bn_right == NULL)
  {
   BinaryNode<TPL> * newnode = new
  BinaryNode<TPL>(content);
   bn_right = newnode;
  }
  else
  { bn_right -> insertContent(content); }
 }
}
template <class TPL> void
   BinaryNode<TPL>::print(int depth)
   const
{
 if (bn_left != NULL) { bn_left -> print
   (depth + 1); }
 for (int identcnt = 0; identcnt < depth;
   identcnt ++) { cout << "\t"; }
 cout << bn_content << endl;
 if (bn_right != NULL) { bn_right -> print
   (depth + 1); }
}
template <class TPL> bool
   BinaryNode<TPL>::searchContent(TP
   L content) const
{
```

```
  if (content == bn_content) { return true; }
  if (content < bn_content)
   {
    if (bn_left == NULL) { return false; }
    return bn_left ->
    searchContent(content);
   }
  else
   {
    if (bn_right == NULL) { return false; }
    return bn_right ->
    searchContent(content);
   }
}
template <class TPL>
    BinaryNode<TPL>::~BinaryNode()
{
  if (bn_left != NULL) { delete bn_left; }
  if (bn_right != NULL) { delete bn_right; }
}
```

```
class Student
{
  string s_name;
public:
  Student(string name):s_name(name) {  }
  Student(const Student & orig):
      s_name(orig.s_name) { }
  Student() { s_name = ""; }
  bool operator < (const Student & arg2) {
      return (s_name < arg2.s_name); }
  /*
  Student & operator = (const Student &
      origin)
  {
    if (this != & origin) { s_name =
      origin.s_name; }
    return * this;
    } */
```

```
  bool operator == (const Student & arg2)
  {
    if (s_name == arg2.s_name) { return
      true; }
    else { return false; }
    }
  friend ostream & operator << (ostream&
      os, const Student & stu);
};
ostream & operator << (ostream& os,
      const Student & stu)
{
  os << stu.s_name << endl;
  return os;
}
class User
{
  int u_age;
public:
  User(int age): u_age(age) { }
  User(): u_age(0) {}
  bool operator < (const User & arg2) {
      return (u_age < arg2.u_age); }
  bool operator == (const User & arg2) {
      return (u_age == arg2.u_age); }
  friend ostream & operator << (ostream&
      os, const User & usr);
};
ostream & operator << (ostream& os,
      const User & usr)
{
  os << usr.u_age << endl;
  return os;
}
int main(void)
{
  BinaryNode<int> bnint(5);
  bnint.insertContent(3);
  bnint.insertContent(6);
```

```
bnint.insertContent(4);
bnint.insertContent(7);
bnint.insertContent('7');
bnint.insertContent(2);
bnint.insertContent(1);
bnint.insertContent(10);
bnint.insertContent(4);
bnint.insertContent(6);
bnint.insertContent(9);
bnint.print(0);
cout << "----------------------------" << endl;
Student stu1("John");
Student stu2("Mary");
Student stu3("Tom");
Student stu4("Amy");
Student stu5("Ted");
BinaryNode<Student> bnstu(stu1);
bnstu.insertContent(stu2);
bnstu.insertContent(stu3);
bnstu.insertContent(stu4);
bnstu.insertContent(stu5);
bnstu.print(0);
cout << "-----------------------------" << endl;
User usr1(21);
User usr2(28);
User usr3(19);
User usr4(17);
User usr5(22);
User usr6(20);
User usr7(18);
BinaryNode<User> bnusr(usr1);
bnusr.insertContent((User)3);
bnusr.insertContent(usr2);
bnusr.insertContent(usr3);
bnusr.insertContent(usr4);
bnusr.insertContent(usr5);
```

```
  bnusr.insertContent(usr6);
  bnusr.insertContent(usr7);
  bnusr.print(0);
  cout << "----------------------------------------------" << endl;
  cout << bnint.searchContent(11) << " "
       << bnint.searchContent(10) << endl;
  Student stu6("Ted");
  cout << bnstu.searchContent(stu6) << " "
       << bnstu.searchContent(stu2) << endl;
  cout << bnusr.searchContent(usr2) << " "
       << bnusr.searchContent(usr4) << endl;
  return 0;
}
```

Overloading and Constructor

- How to handle

```
MyComplex comp1 (2.6, 3.8);
MyComplex comp2 = comp1 + 5.3;
MyComplex comp2 = 5.3 + comp1;
```

- Are they the same?
- one solution for comp2 = **5.3 +** comp1;

```
MyComplex operator + ( double val, const MyComplex & arg2 )
{
    return MyComplex( val + arg2.getReal(), arg2.getImag());
}
```

- two solutions for `comp2 = comp1 + 5.3;`

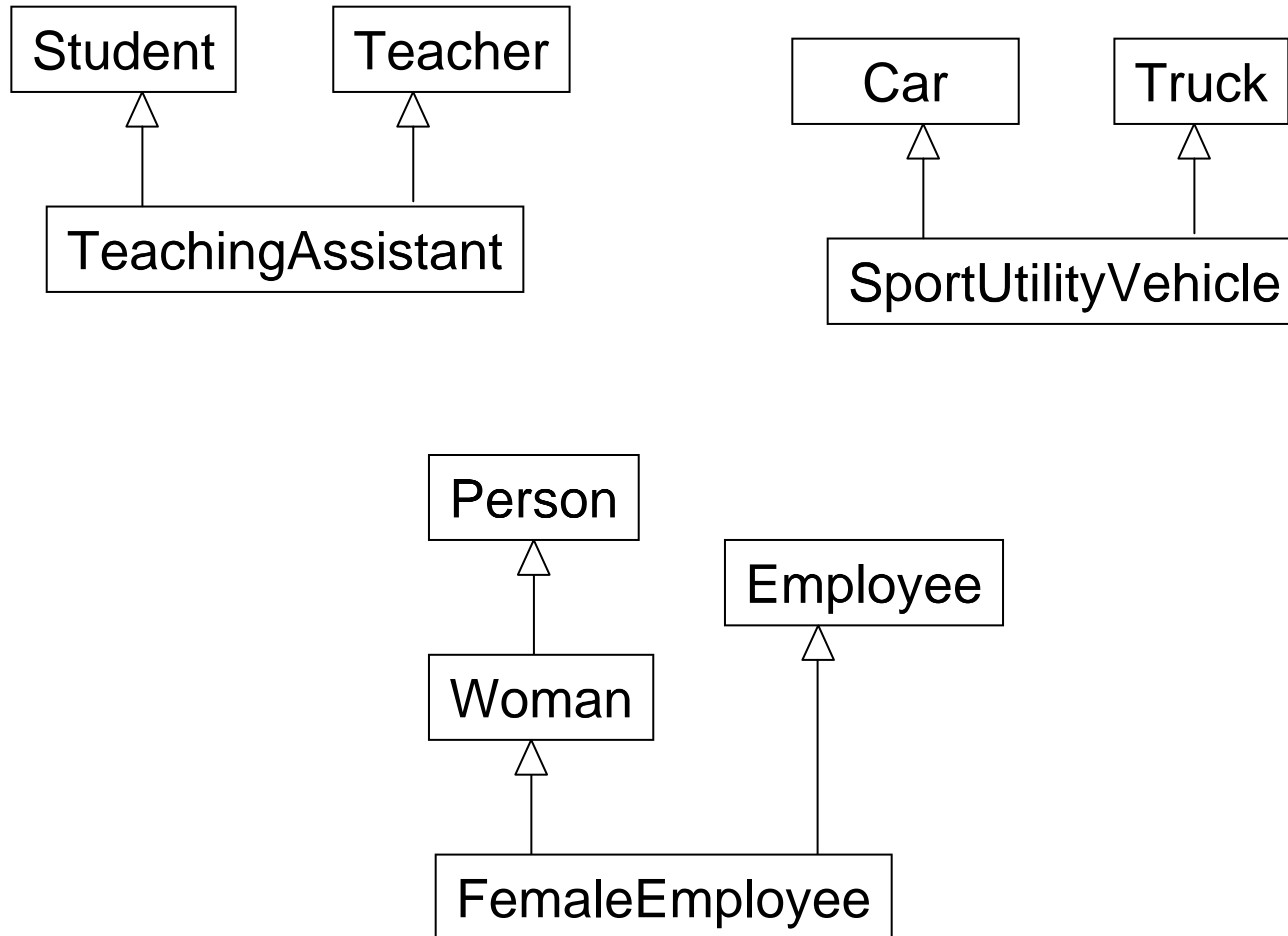
```
/* solution 1 */
MyComplex operator + ( const MyComplex & arg1, double val )
  { return MyComplex( arg1.getReal() + val, arg1.getImag());  }

/* solution 2 */
class MyComplex {
    MyComplex( double r, double i = 0) : re(r), im(i) {}
    MyComplex operator + ( const MyComplex & ) const {
      return MyComplex(re + arg.re, im + arg.im );
}
```

Multiple Inheritance

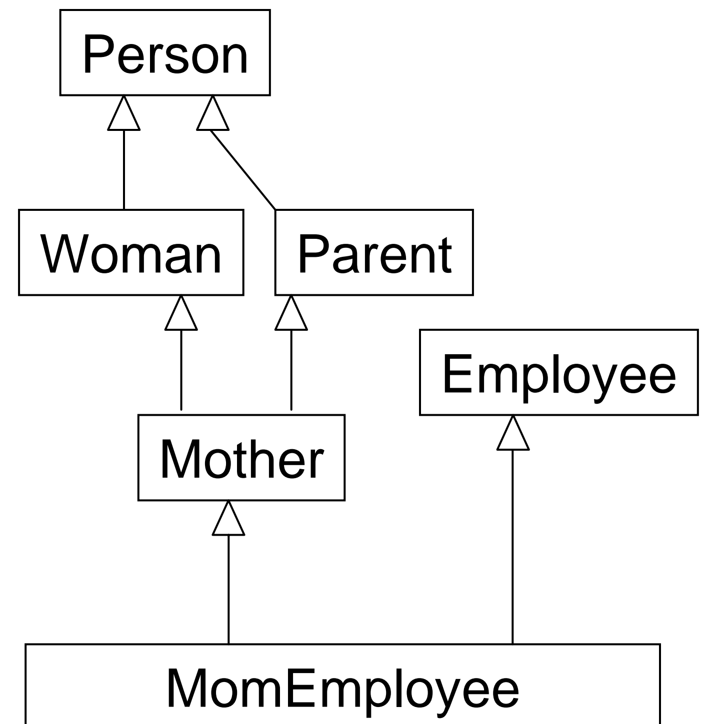
- One of the most (if not **the** most) controversial features in C++.
- Not supported in Java. Java uses interfaces.
- Most C++ books do not explain the concept and the problems.
- Understand the advantages and the problems and you can decide whether to use it.

When Multiple Inheritance Makes Sense



Design Issues

- A class provides interface and implementation.
- Code reuse is good but a class, once declared, is hard to change because of the code depending on this class.
- **If you have any doubt, do not create a class.**
- Avoid the proliferation of classes because they make code reuse harder.



Too Many Classes

- In many payroll systems, a person's status (for example tax withholding) depends on the person's role. If a person has children, the person's tax withholding is less.
- If an employee is also a mother, a company may send a gift to the children on their birthdays.
- Should you create one class for each possible status of employee?
- Or, use attributes to distinguish their status?

What Does a Class Give You?

- (usually) Interface and Implementation
- polymorphism
- object creation

- If you are not using polymorphism, think twice (or more) before creating a class.
- It is usually easier to change the behavior of an attribute (encapsulation) than changing the interface (code reuse).

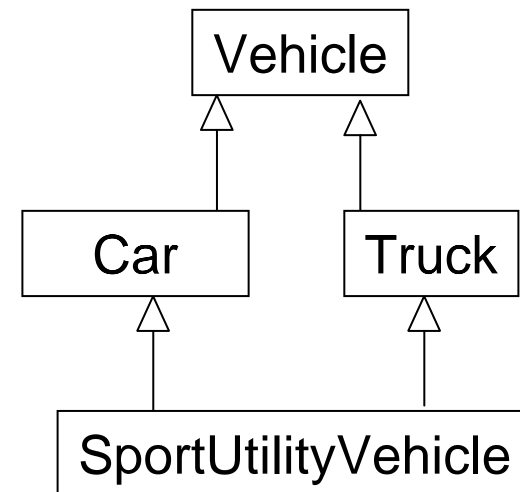
Flexibility with Attributes

```
class TaxStatus
{
};
class Employee
{
   TaxStatus e_status;
};
```

⇒ If one more status is added or removed, no change is needed in Employee.

Repeated Inheritance (Common Base)

```
class Vehicle
{
    int v_engineSize;
    int v_numberWheel;
};
```



Does SportUtilityVehicle have one v_engineSize or two?

⇒ **two**, unless you use virtual inheritance

```
#include <iostream>
using namespace std;
class A
{
protected:
  int a_val;
};
class B: public A
{
protected:
  int b_val;
};
class C: public A
{
protected:
  int c_val;
};
```

```
class D: public B, public C
{
public:
  D(int av, int bv, int cv)
  { a_val = av; b_val = bv;
              c_val = cv; }
};
int main(int argc, char * argv[])
{
  return 0;
}
```

In constructor `D::D(int, int, int)':
error: reference to `a_val' is ambiguous
error: candidates are: int A::a_val
error: int A::a_val
error: `a_val' was not declared in this scope

Resolve Duplicated Attributes

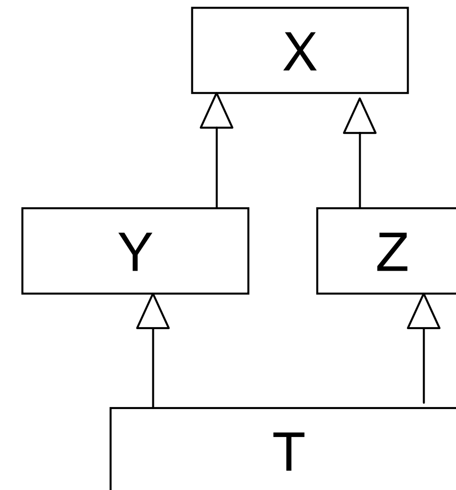
Specify where it comes from:

```
class D: public B, public C
{
public:
  D(int av, int bv, int cv)
  {
    B::a_val = av;
    C::a_val = av;
    b_val = bv;
    c_val = cv;
   }
};
```

Virtual Inheritance

```
class Y: virtual public X
{
};
class Z: virtual public X
{
};

public T: public Y, public Z
{
};
```



⇒ T has only one copy for the attributes from X