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

Design Issues and Multiple Inheritance in C++
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.
• Should you create one class for each possible status of employee?
• Or, should you use attributes to distinguish their status?
Spaghetti code

10 i = 0
20 i = i + 1
30 GOTO 55
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
55 PRINT i; " squared = "; i * I
56 GOTO 40
60 PRINT "Program Completed."
70 END

// Done correctly
FOR i = 1 TO 10
   PRINT i; " squared = "; i * i
NEXT i
PRINT "Program Completed."
END

Goto considered harmful, Dykstra
Too many classes lead to spaghetti OO code

• Unconstrained use of DLLs that reference each other is another example of this
• This makes programming debugging and maintenance extremely difficult
What Does a Class Give You?

- interface (public inheritance) 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).
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.
Multiple Inheritance

Student

Teacher

TeachingAssistant

Car

Truck

SportUtilityVehicle
Repeated/Multiple Inheritance

class Vehicle
{
    int v_engineSize;
    int v_numberWheel;
};

Does SportUtilityVehicle have one v_engineSize or two?
⇒ two, unless you use \textit{virtual inheritance}
⇒ ambiguous, compilation error
Repeated Inheritance

Let us construct a SportUtilityVehicleObject

1. Construct a SportVehicleObject, but to do that we
2. Construct a Car, but to do that we
   a. Construct a Vehicle
   c. Construct a Truck, but to do that we
   a. Construct a Vehicle

vehicle
v_engineSize
v_numberWheel

Car

vehicle
v_engineSize
v_numberWheel

Truck

SUV
#include <string>
#include <iostream>
using namespace std;

class B1
{
public:
    B1(int a);
    B1( );
    void print( );
    ~B1();
    int age;
};

B1::B1(int a) {
    age=a;
}

B1::B1( ) {
    age=1;
}

void B1::print( ) {
    cout << age << endl;
}

B1::~B1( ) {cout << age << " deleted " << endl;};

class B2
{
public:
    B2(int a);
    B2( );
    void print( );
    ~B2();
    int age;
};

B2::B2(int a) {
    age=a;
}

B2::B2( ) {
    age=1;
}

void B2::print( ) {
    cout << age+1 << endl;
}

B2::~B2( ) {cout << " B2 deleted " << endl;};

B1 and B2 are identical except for the class names
```cpp
#include "B1.h"
#include "B2.h"

... class D : public B1, public B2 {
public:
    D(int a);
    ~D();
};

#include "D.h"
#include <iostream>
using namespace std;

D::D(int a) {
    age=a;
}

int main(int argc, char * argv[]) {
    D* d = new D(4);
    d->print( );
}
```
g++ main.cpp D.cpp B1.cpp B2.cpp
main.cpp: In function ‘int main(int, char**)’:
main.cpp:7: error: request for member ‘print’ is ambiguous
B2.h:11: error: candidates are: void B2::print()
B1.h:11: error:                 void B1::print()
D.cpp: In constructor ‘D::D(int)’:
D.cpp:6: error: reference to ‘age’ is ambiguous
B2.h:13: error: candidates are: int B2::age
B1.h:13: error:                 int B1::age
D.cpp:6: error: reference to ‘age’ is ambiguous
B2.h:13: error: candidates are: int B2::age
B1.h:13: error:                 int B1::age
#include <string>
using namespace std;

class B {
public:
    B(int a);
    B( );
    virtual void print( );
    ~B();
    int age;

    B::B(int a) {
        age=a;
    }
    B::B( ) {
        age=1;
    }
    void B::print( ) {
        cout << "B" << endl;
    }

    B::~B( ) {cout << age << " deleted " << endl;};
```cpp
#include "B.h"
#include <string>
using namespace std;

class D1 : public B {
public:
    D1( );
    D1(int a);
    ~D1();
};

#include "D1.h"
#include <iostream>
using namespace std;

D1::D1( ) {age = -1;}
D1::D1(int a) {
    age=a;
}

D2::D2( ) {age = -2;}
D2::D2(int a) {
    age=a;
}
```
```cpp
#include "D1.h"
#include "D2.h"
#include <string>
using namespace std;

class DD : public D1, public D2 {
public:
    DD(int a);
    ~DD();
};

#include "D.h"
#include <iostream>
using namespace std;

int main(int argc, char * argv[]) {
    DD* d = new D(4);
}
```
What happens when we compile?

DD.cpp: In constructor ‘DD::DD(int)’:
DD.cpp:6: error: reference to ‘age’ is ambiguous
B.h:13: error: candidates are: int B::age
B.h:13: error: int B::age
DD.cpp:6: error: reference to ‘age’ is ambiguous
B.h:13: error: candidates are: int B::age
B.h:13: error: int B::age
Virtual Inheritance
Virtual Inheritance

• Change D1 and D2 to inherit B virtually and then compile.
• All is ok because only one base B object is created, and there is not ambiguity.

```cpp
class D1 : public virtual B {
public:
  D1();
  D1(int a);
  ~D1();
};

#include /*D1_H_*/
#include "D1.h"
#include <iostream>
using namespace std;

D1::D1() {age = -1;}
D1::D1(int a) {
  age = a;
}
D1::~D1() {}
```

```
Car  truck  vehicle
  v_engineSize
  v_numberWheel
SUV

YHL/SPM 2016
```
A: aval, A(int), void print()
B: bval, B(int,int), void print()
C: cval, C(int,int), void print()
D: dval, D(int,int,int,int,int), void print()
Some more examples
#include <iostream>
class B {
    public:
    int age;
    B(int a) {
        age=a;
        std::cout << "B::B(" << a << ")" << std::endl;
    }
    B() {
        age=1;
        std::cout << "B::B(void)" << std::endl;
    }
    void print() {
        std::cout << "B: " << age << std::endl;
    }
    ~B() {std::cout << age << " deleted " << std::endl;};
};
many Inheritance / no virtual

class D1 : public B {
public:
    D1( ) {
        age = -1;
        std::cout << "D1::D1(void)" << std::endl;
    }
    D1(int a) : B(a) {
        age=a;
        std::cout << "D1::D1(" << a << ")" << std::endl;
    }
};

class D2 : public B {
public:
    D2( ) {
        age = -2;
        std::cout << "D2::D2(void)" << std::endl;
    }
    D2(int a) : B(a) {
        age=a;
        std::cout << "D2::D2(" << a << ")" << std::endl;
    }
};
class DD : public D1, public D2 {
public:
    DD(int a):D1(-a),D2(a) {
        std::cout << "DD::DD(" << a << ")" << std::endl;
    }

    DD() {};
};

int main(int argc, char * argv[]) {
    DD* d = new DD(4);
    d->D1::print();
    d->D2::print();
}

YHL/SPM 2016
Execution gives: (note that B constructor called twice because not virtually inherited by D1, D2)

B::B(-4)
D1::D1(-4)
B::B(4)
D2::D2(4)
DD::DD(4)
B: -4
B: 4
Note how order of declaration is
determined for multiply inherited classes

class DD : public D2, public D1 {
public:
    DD(int a):D1(-a),D2(a) {
        std::cout << "DD::DD(" << 
        a << ")" << std::endl;
    }

    DD() {}
};

int main(int argc, char * argv[]) {
    DD* d = new DD(4);
    d->D1::print( );
    d->D2::print( );
}

B::B(4)
D2::D2(4)
B::B(-4)
D1::D1(-4)
DD::DD(4)
B: -4
B: 4
Multiple Inheritance with Virtual inheritance

class B {
public:
    int age;
    B(int a);
    B();
    virtual void print();
    virtual ~B();
};
B::B(int a) {
    age=a;
    std::cout << "B::B(" << a << ")" << std::endl;
}
B::B() {
    age=1;
    std::cout << "B::B(void)" << std::endl;
}
Multiple Inheritance with Virtual inheritance

class D1 : virtual public B {
   public:
      D1( );
      D1(int a);
};

D1::D1( ) : B(-100) {
   std::cout << "D1::D1(void)" << std::endl;
}

D1::D1(int a) : B(a*100) {
   std::cout << "D1::D1(" << a << ")" << std::endl;
};
D2 is the same as D1

class D2 : virtual public B {
    public:
    D2( );
    D2(int a);
};

D2::D2( ) : B(-100) {
    std::cout << "D1::D1(void)" << std::endl;
}

D2::D2(int a) : B(a*100) {
    std::cout << "D1::D1(" << a << ")" << std::endl;
};
Note the change in output for B prints

class DD : public D1, public D2 {
   public:
      DD(int a);
      DD( );
   }

DD::DD(int a):D1(-a),D2(a) {
   std::cout << "DD::DD(" << a << ")" << std::endl;
}

int main(int argc, char * argv[]) {
   DD* d = new DD(4);
   d->D1::print( );
   d->D2::print( );
}
• Execution gives what is below. Note that B constructor called \textit{once} because it is virtually inherited by D1, D2

\begin{verbatim}
B::B(void)
D1::D1(-4)
D2::D2(4)
DD::DD(4)
B: 1
B: 1
\end{verbatim}

• Who calls B's constructor?
• The answer is \textit{the most derived class}
The most derived class

DD is the most derived
• Note that since B(a) is not explicitly called by the DD constructor that the zero arg B() constructor is called from DD

B::B(void)
D1::D1(-4)
D2::D2(4)
DD::DD(4)
B: 1
B: 1
What if we want to create an object of a virtually inheriting class?

class B {
  public:
    int age;
    B(int a) {
      age=a;
      cout << "B::B(" << a << ")" << endl;
    }
    B( ) {
      age=1;
      cout << "B::B(void)" << endl;
    }
    void print( ) {
      cout << "B::B" << endl;
    }
    ~B( ) {cout << age << " deleted " << endl;}
  };

class D1 : virtual public B {
  public:
    int age;
    D1( ) {
      cout << "D1::D1(void)" << endl;
    }
    D1(int a) {
      cout << "D1::D1(int)" << endl;
    }
  };

What if we want to create an object of a virtually inheriting class?

class D2 : virtual public B {
public:
  int age;
D2( ) : B(5011) {
  cout << "D2::D2(void)" << endl;
}

D2(int a) : B(57) {
  cout << "D2::D2(int)" << endl;
}

~D2( ) {};
};

class DD : public D1, public D2 {
public:
  DD(int a):D1(a),D2(a),B(100) {
    cout << "DD::DD(int)" << endl;
  }

  ~DD( ) {};
};

int main(int argc, char * argv[]) {
  cout << "Creating DD object" << endl;
  DD* dd = new DD(4);
  cout << "Creating D2 object" << endl;
  D2* d2 = new D2(5);
}
D2 constructor call to B constructor is **used** when D2 is the most derived object in what is being created.

D2 constructor call to B constructor is **ignored** when D2 is **NOT** the most derived object in what is being created.

```cpp
B(int a) {
    age=a;
    cout << "B::B(" << a << ")" << endl;
}

D2(int a) : B(57) {
    cout << "D2::D2(int)" << endl;
}

D2* d2 = new D2(5); // in main
```
Why is D1 constructor called before D2 constructor?

class DD : public D1, public D2 {
    public:
        DD::DD(int a):D1(a),D2(a) {
            cout << "DD::DD(int)" << endl;
        }
}

B::B(void)
D1::D1(int)
D2::D2(int)
DD::DD(int)
• Why is D1 constructor called before D2 constructor?
• Because D1 is declared before D2.

```cpp
class DD : public D1, public D2 {
public:
    DD::DD(int a):D1(a),D2(a) {
        cout << "DD::DD(int)" << endl;
    }
}
```

• What defines the declaration order since files may be separately compiled with no clue as to which is compiled first?
See manyVirtD2D1Inheritance

• Why is D1 constructor called before D2 constructor?
• Because D1 is declared before D2.

B::B(void) class DD : public D1, public D2 {
  public:
  DD::DD(int a):D1(a),D2(a),B(a) {
    cout << "DD::DD(int)" << endl;
  }
}

• What defines the declaration order since files may be separately compiled with no clue as to which is compiled first and no strict chain of inheritance. The bold text above does.
• If the code below, the outcome below happens (D2 declared before D1)  
• B is called first because when creating the first declared D2 object the virtually inherited B's constructor must be called.  
• Because B is virtually inherited C++ gives the most derived class (DD) responsibility for creating it.  
• Ensures that it is created once and before any object created for another class needs it.

```cpp
B::B(void)
D2::D2(int)
D1::D1(int)
DD::DD(int)

class DD : public D2, public D1 {
public:
  DD::DD(int a):D1(a),D2(a),B(a) {
    cout << "DD::DD(int)" << endl;
  }
};
```