Lecture 21

Multiple Inheritance

What is Multiple Inheritance?
- In C++ code, multiple inheritance looks just like single inheritance, except there are "multiple" base classes specified.
- They are separated by commas.
- They may individually be declared as public, private or protected.
- If a given class is not declared as being of type public, private or protected, the default is private.
- Inherited member variables are accessible according to the rules of single inheritance.

```cpp
class Instructor : public Person, public Employee
{
    ..
};
```

Arguing about it...
- One can argue that multiple inheritance shouldn’t be necessary if you’ve defined your class hierarchy properly.
- As a matter of fact, most books don’t cover it.
- Since every Employee really is a Person, should we be able to do this?

```cpp
class Employee : public Person
{
    ..
};
```

```cpp
class Instructor : public Employee
{
    ..
};
```

```cpp
class Student : public Person
{
    ..
};
```

What are Mix-Ins?
- A mix-in is really just what it sounds like.
- It’s a combination of two classes through multiple inheritance.
- Think back to interfaces for a moment.
- Consider a DataPrinter interface which allows us to derive a class used to print either an array of string or List data types...

```cpp
// The following class is an example of an interface
class DataPrinter
{
    public:
        virtual void printData(string *, int)=0;
        virtual void printData(List *)=0;
    };
```

Arguing... Well, certainly you can. Remember, this is C++. You can do just about anything you want to!
- Here’s another wrench in the puzzle...
- What happens when a Student is an Employee as well?
- You still wouldn’t derive every Student from Employee either through single or multiple inheritance.
- What it might suggest is that you need to re-think your object hierarchy before starting to code.
- This is usually the point of view of those who don’t believe in multiple inheritance.
- We won’t settle the argument here.
- We will concentrate on the other side of the argument, which was that sometimes you need to derive from two pre-existing classes. It’s called a "mix-in".

What is Multiple Inheritance?
- We defined inheritance earlier in the semester as a relationship between classes.
- If class C inherits from class B it is normally the case that C is a B or that C is a kind of B.
- But what happens when we discover that C is a kind of B and C is also a kind of D?
- For example, remember our Student and Instructor classes? They both were derived from Person.
- But let’s throw another class into the mix: Employee.
- An Instructor is certainly a kind of Employee, but an Instructor is also a kind of Person.
- This seems like a possible candidate for multiple inheritance!
What are Mix-Ins?
- Now, suppose you had to “derive” a class from DataPrinter to do printing to cout.
- It might be called CoutPrinter and looked like this:

```cpp
class CoutPrinter : public DataPrinter
    {
    public:
        void printData(string *, int);
        void printData(List &);
    };
```

```cpp
void CoutPrinter::printData(string *stringArray, int size)
{
    cout << "ARRAY OUTPUT: " << endl;
    for (int i=0; i<size; i++)
    {
        cout << i << ". " << stringArray[i] << endl;
    }
}
```

What are Mix-Ins?
- Now, consider that you would like to define another class to output to a dot matrix printer instead of cout.
- You might call it DMDataPrinter.
- The thing is, you already have a class for representing dot matrix printers:

```cpp
class DotMatrixPrinter {
    public:
        DotMatrixPrinter(): port("") {}
        DotMatrixPrinter(string argPort): port(argPort) {}
        void setPort(string argPort);
        string getPort();
        bool openPort();
        void closePort();
    protected:
        ofstream outStream;
    private:
        string port;
    };
```

What are Mix-Ins?
- You don’t want to duplicate the functionality of DotMatrixPrinter which appears to take care of some of the overhead involved in opening a connection to the printer.
- So, you make use of multiple inheritance to create a mix-in of two existing classes, namely DotMatrixPrinter and DataPrinter.
- The argument that perhaps our object hierarchy needs rethinking is still somewhat valid, but not as strong.
- You see, DataPrinter is an interface.
- To try and incorporate it into the DotMatrixPrinter hierarchy (which is probably derived from a generic “Printer” class in the real world) doesn’t make sense because you would then be forcing every printer to implement the printData method (remember, it’s pure virtual).
- No, this is clearly a case for multiple inheritance:

```cpp
class DMDataPrinter : public DataPrinter, public DotMatrixPrinter
    {
    public:
        void printData(MyString *, int);
        void printData(List &);
    };
```

What are Mix-Ins?
- Let’s see this work...

```cpp
void DMDataPrinter::printData(MyString *anArray, int size)
{
    if (openPort())
    {
        outStream << "ARRAY OUTPUT: " << endl;
        for (int i=0; i<size; i++)
        {
            outStream << i << ". " << anArray[i] << endl;
        }
    }
    closePort();
}
```
Demonstration #1

Simple Multiple Inheritance

More About Multiple Inheritance
- This code is OK...

```
class A {
public:
    void printSomething() { cout << "Something from A" << endl; }
};
class B {
public:
    void printSomething() { cout << "Something from B" << endl; }
};
class C : public A, public B {
public:
    void printSomething() { A::printSomething(); }
};
main() {
    C c;
    c.printSomething();
    c.B::printSomething(); // That looks weird!
}
```

More About Multiple Inheritance
- Earlier we talked about additions to our old friends, the Student and Instructor classes.
- They were both derived from Person.
- We added Employee and showed how we could derive Instructor from both Person and Employee.
- That was our example of multiple inheritance.
- For another example, consider a hierarchy of "meals"
- Assume we have a base class called meal.
- Next we have three derived classes, breakfast, lunch and dinner.
- But, there’s this meal called brunch.
- It’s really a kind of breakfast and a kind of lunch, so we use multiple inheritance to derive brunch from breakfast and lunch.
- Graphically, that might look like this:

```
Meals:
  Breakfast
    Brunch
  Lunch
```

There is an interesting problem/side effect here!

Demonstration #2

Our Hierarchy of Meals
More About Multiple Inheritance

- There are actually two instances of Meal present in an instance of Brunch!

More About Multiple Inheritance

- Since Breakfast and Lunch are both derived from Meal, using multiple inheritance to derive Brunch from Breakfast and Lunch causes duplication in the Meal base class.
- This leads to ambiguous access errors when attempting to access Meal from Brunch.
- We can fix these problems with explicit references to which version of Meal we want.
- We do this by explicitly referencing the appropriate class derived from Meal (Breakfast or Lunch)
- This fixes the compile time problem(s)...

Virtual Base Classes

- OK, so now it works. But is it what we want?
- In general, probably not.
- You can avoid this sub-object duplication by making any classes which derive directly from Meal derive it virtually.
- Virtual again?

```cpp
class Breakfast : virtual public Meal
{
    
};
class Lunch : virtual public Meal
{
    
};
```

Virtual Base Classes

- In defining our classes this way we eliminate duplication if we derive through both of these classes via multiple inheritance later.
- But that means we have to know that we will be deriving from both of these classes via multiple inheritance later!
- One major point in the argument for multiple inheritance was that you would be more likely to use multiple inheritance for classes which already existed, that is, to create “mix-ins”.
- But in order to utilize virtual base classes the two classes you might be inheriting from must have inherited from a common base class virtually.
- If you didn’t write them as part of your current design effort, the programmer of those classes must have had foresight to realize that virtual base classes were appropriate.
- Our picture now looks like this:

Demonstration #3

Our Next Hierarchy of Meals

And now we can go back to our original code...
Demonstration #4

Our Best Hierarchy of Meals

Ambiguity

Last time we talked about ambiguous access when two base classes implement the same member function.

What follows is an interesting twist where a virtual base class is involved as well (from LNG):

```cpp
class B {
public:
    virtual void f() { cout << "She loves me!!!!" << endl; }
};

class D1 : virtual public B {
public:
    void g() { f(); }
};

class D2 : virtual public B {
public:
    void f() { cout << "She loves me not" << endl; }
};

class DD : public D1, public D2 {
    void g() { f(); }  // This is defined to call D2::f()
};
```

You see, this really isn’t a case of ambiguity.

- `D1::g()` is defined to call `B::f()`.
- `B::f()` is a virtual function in a virtual base class.
- Since `D2::f()` is derived from the same instance of `B` (via the virtual base class mechanism), `D2::f()` ends up getting called.
- If `me.g()` was defined to call `B::f()` instead of just `f()`, we’d see the other output!

OUTPUT:
She loves me not

Lecture 21

Final Thoughts