# CS 213 -- Lecture #22

"Late Night Guide to C++"
Chapter 12 pg. 319 - 326

MORE MULTIPLE INHERITANCE, NAMESPACES

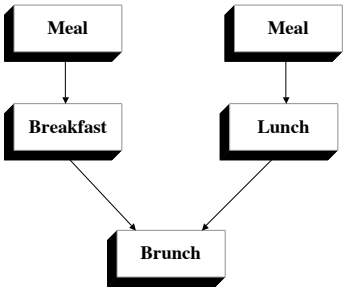
## *Administrative...*

- Assignment #10 due on Tuesday, 11/30
  - If you turn it in by 11/18, I will grade and make it available to you before the prelim.
  - This will be the last assignment
- Remember, second prelim on Tuesday, 11/23

## *More About Multiple Inheritance*

- Last time 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:

## *More About Multiple Inheritance (cont)*

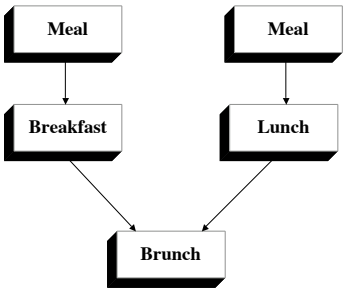



**There is an interesting problem/side effect here!**

# *Demonstration 1*

Our Hierarchy of Meals

## *More About Multiple Inheritance (cont)*




- 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)...

# *Demonstration 2*

Our Next Hierarchy of Meals

## *Virtual Base Classes*

- OK, so now it works. But is it what we want?
- In general, probably not. (Although LNG suggests there might be some cases where it is appropriate)
- You can avoid this sub-object duplication by making any classes which derive directly from Meal derive it *virtually*.
- As Lois said, "Virtual again?"

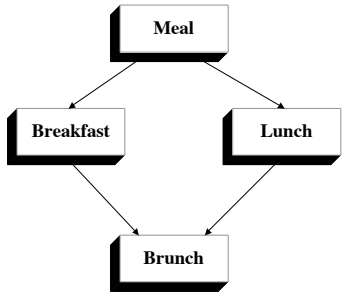
```
class Breakfast : virtual public Meal
{
…
};

class Lunch : virtual public Meal
{
…
};
```

## *Virtual Base Classes (cont)*

- 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!
- Clark's biggest point 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 (which, according to Clark's argument you haven't), the programmer of those classes must have had foresight to realize that virtual base classes were appropriate.
- Our picture now looks like this:

## *More About Multiple Inheritance (cont)*




- And now we can go back to our original code...

# *Demonstration 3*

Our Best Hierarchy of Meals

## *Assignment Involving Virtual Base Classes*

- Hang on, it gets bumpy here…
- Consider all we want to do is declare a Brunch and then assign it to another one.
- Our code might look like this:

```
int main()
{
  Brunch aBrunch("Scrambled Eggs");
  Brunch anotherBrunch;

  anotherBrunch = aBrunch;
  return 0;
}
```

## *Demonstration 4*

I'd Just Like 1 Copy, Please

## *Assignment Involving Virtual Base Classes (cont)*

- ARGH!
- Technically, it works…
- It's just inefficient to copy the same base item 3 times.
- It could also be more problematic where system resources are involved.
- We could take the brute force approach by defining an assignment operator for every class on the way...

```
Brunch &Brunch::operator=(Brunch &b){
  Lunch::operator=(b);
  Breakfast::operator=(b);
  return *this;
}
Lunch &Lunch::operator=(Lunch &);            // Definition omitted
Breakfast &Breakfast::operator=(Breakfast &); // Definition omitted
```

## *Demonstration 5*

Please, only one copy!!!!

## *Assignment Involving Virtual Base Classes (cont)*

- And you thought you had seen *all* the frustrating stuff with C++!
- The solution here is not elegant, but it does the trick.
- You need to define a protected function which does the actual copying *for the current class only*.
- Then you use `operator=` as a "forwarding" function.
- `operator=` needs to know what assignments need to be made both in its own class and any classes it inherits from.
- For all of those classes it will call the protected function.
- We'll call that function `assign`.
- The code now looks like this:

## *Demonstration 6*

May I *please* only have one copy?

## *More on Ambiguity*

- Phew! That took some time to get to.
- Last time we talked about ambiguous access when two base classes implement the same member function.
- The book gives an interesting twist where a virtual base class is involved as well:

```
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 { };
```

## *More on Ambiguity (cont)*

```
int main()
{
  DD me;
  me.g();
  return 0;
}

OUTPUT:
She loves me not
```

- 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!

## *Namespaces*

- At the beginning of the semester, I asked you to take it on faith that if you wanted to use the functions defined in <iostream> you would need to add the following line of code to your program:
  - `using namespace std;`
- And all I told you is that we'd cover namespaces "later".
- Well, here we are, near the end of the course and I'm finally getting to it :-)
- A *namespace* is a method by which you can encapsulate any combination of classes, constants, globals, types into a neat "package".
- Without using special directives, such classes, globals, etc., can only be accessed through the namespace's name.
- Consider that there are thousands of C++ courses given throughout the world.
- Consider that they all come to the conclusion that `string` doesn't cut it and they need to implement their own string class.

## *Namespaces (cont)*

- Now consider that with all that talent, all that potential programming genius, all of that raw brain power…
- 90% of those courses choose to name their string class `MyString`.
- Now, it's a catchy name, but what happens when when your friend at MIT decides to send you code that he/she has been working on in their C++ class.
- Your source code gets mixed up and, before you know it, your code is crashing all of the time.
- Then you discover that their MyString was being used instead of our MyString and since theirs doesn't do error checking or dynamic growth... ... you're having problems :-)
- Namespaces can help.
- Namespaces look somewhat like a class declaration.
- You define a name for your namespace and use curly braces to define a scope for it.

## *Namespaces (cont)*

- Everything which occurs in that scope is "hidden" in your namespace.
- It might look something like this:

```
//
// MyString.h
//
#ifndef _MYSTRING_H
#define _MYSTRING_H

namespace CornellCS213        // Defines a namespace named CornellCS213
{
  class MyString
  {
    public:
    …
  };
}
#endif
```

## *Namespaces (cont)*

- Given the previous namespace declaration, you can access `MyString` in one of three ways.
- First, you can use `MyString`'s new fully qualified name anywhere in your code:
  - `CornellCS213::MyString aString;`
- Second you can specifically designate that, for the current file being compiled, you'd like to use the `MyString` implementation in the `CornellCS213` namespace:
  - `using CornellCS213::MyString;`
  - `MyString aString;`
- Third, you can simply state that you'd like to use all the contents of a given namespace without qualification. This is what we've been doing all year with the `std` namespace:
  - `using namespace CornellCS213;`
  - `MyString aString;`

## *Namespaces (cont)*

- Now, a cool feature with namespaces is that a single namespace may span multiple files.
- Any time a namespace keyword is encountered, the contents that follow are "appended" to any previously existing entries in that namespace.
- So given our example which encapsulated MyString into CornellCS213, if the following were encountered:

```
namespace CornellCS213
{
  class Person  { … };
  class Student : public Person { … };
  class Instructor : public Person { … };
}
```

- It would also be accessible in the CornellCS213 namespace, along with MyString.

## *Namespaces (cont)*

- You may also define "aliases" to existing namespaces to shorten their names.
- Consider the following namespace:
  - `namespace AReallyLongNameSpaceName { … }`
- You can use the following call to shorten it:
  - `namespace X = AReallyLongNameSpaceName;`

## *Final Thoughts*

- Assignment #10 due 11/30
- Prelim next week (11/23), in class, closed book.
- We're just about done, need to cover Allocators next time.
- When I hand back the prelims on 11/30 I will let you know what you need to get on the last assignment to be "exempt" from the final project!