CS 213 -- Lecture #21

"Late Night Guide to C++"
Chapter 12 pg. 311 - 330

MULTIPLE INHERITANCE

Administrative...

- Assignment #9 due today
- Assignment #10 is up on web site!
- Remember, second prelim on Tue. Nov. 23

Dialogue

A Conversation About Multiple Inheritance

Thank you Lois and Clark

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 is Multiple Inheritance? (cont)

- 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.

```
class Instructor : public Person, public Employee
{
…
};
```

Arguing about it...

- Just like we saw in the Lois & Clark example, one can argue that multiple inheritance shouldn't be necessary if you've defined your class hierarchy properly.
- Since every Employee really is a Person, should we be able to do this?

```
class Employee : public Person
{
…
};
class Instructor : public Employee
{
…
};
class Student : public Person
{
…
};
```

Arguing about it… (cont)

- 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 was Lois' point through the Lois and Clark dialogue.
- The argument of using multiple inheritance vs. redesigning your object hierarchy is a good one.
- We won't settle it here.
- We will concentrate on Clark's point, which was that sometimes you need to derive from two pre-existing classes. He called it a "mix-in".

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 MyString or List data types…

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

What are Mix-Ins? (cont)

- Now, suppose you had to "derive" a class from `DataPrinter` to do printing to `cout`.
- It might be called `CoutPrinter` and looked like this:

```
class CoutPrinter : public DataPrinter
{
public:
  void printData(MyString *,int);
  void printData(List &);
};
void CoutPrinter::printData(MyString *stringArray,int size)
{
  cout << "ARRAY OUTPUT: " << endl;
  for (int i=0; i<size; i++)
  {
    cout << i << ". " << stringArray[i] << endl;
  }
}
```

What are Mix-Ins? (cont)

```
void CoutPrinter::printData(List &aList)
{
  List::Node *aNode = aList.the_head;
  int i= 0;
  cout << "LIST OUTPUT:" << endl;
  while (aNode)
  {
    cout << i << ". " << aNode->the_value << endl;
    i++;
    aNode = aNode->next;
  }
}
```

- 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:

What are Mix-Ins? (cont)

```
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;
};
```

- You don't want to duplicate the functionality in either of the existing classes in `DMDataPrinter`.
- You need to derive `DMDataPrinter` from `DataPrinter` so that all the virtual functionality we rely on with interfaces still works.

What are Mix-Ins? (cont)

- 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`.
- Lois' 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:

What are Mix-Ins? (cont)

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

- Since this class is derived from `DotMatrixPrinter` as well, the `printData` member function can be implemented like this:

What are Mix-Ins? (cont)

```
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();
  }
}
```

- Let's see this work...

Demonstration

Simple Multiple Inheritance

A Few Words on Pre-Processing

- I've been keeping a secret from you.
- It's called the set of C PreProcessor directives.
- They could make your life easier
- Any text in your source file which begins with "#" is treated as a preprocessor command.
- Some compilers insist that the # appear in the first "column" (can you say FORTRAN?)
- Other compilers are more lenient.
- The commands are called "preprocessing commands" because they are evaluated during the compilation process.
- In fact, the result of their evaluation is treated as part of your source file and then compiled with the rest of your source code.
- This may help clear up some mysteries.
- Consider the single most common directive: `#include`

A Few Words on Pre-Processing (cont)

- You commonly see #include used to "bring in" other source or header files into the current file.
- Lets take our simple DataPrinter.h file. It looks like this:

```
// DataPrinter.h
#include <MyString.h>
class DataPrinter
{
public:
  virtual void printData(MyString *argArray,int size)=0;
};
```

A Few Words on Pre-Processing (cont)

- Now, consider that we have a simple TestIt.cpp file which wants to make use of this interface.
- It might look like this:

```
// TestIt.cpp
#include <DataPrinter.h>
int main()
{
  DataPrinter *aDataPrinter;
…
}
```

A Few Words on Pre-Processing (cont)

• Now, when you compile this code, the compiler will expand out the preprocessing directive and will have the following:

```
// TestIt.cpp
// DataPrinter.h
#include <MyString.h>
class DataPrinter
{
public:
  virtual void printData(MyString *argArray,int size)=0;
};
int main()
{
  DataPrinter *aDataPrinter;
…
}
```

A Few Words on Pre-Processing (cont)

• Now, there's another #include directive present. It gets expanded too!

```
// TestIt.cpp
// DataPrinter.h
//
// MyString.h
…  REST OF MYSTRING LEFT OUT FOR OBVIOUS REASONS  …
class DataPrinter
{
public:
  virtual void printData(MyString *argArray,int size)=0;
};
int main()
{
  DataPrinter *aDataPrinter;
…
}
```

A Few Words on Pre-Processing (cont)

• When you run into even more complicated situations you can inadvertently end up including a header file twice.

• That results in multiple-definition compiler errors :-(.

• There is a trick you can use involving other preprocessing directives to avoid the multiple-definition trap.

• Those directives are called `#ifdef`, `#define`, and `#endif`.

• #define is use to define a macro, the syntax is:

  – `#define <SYMBOLNAME> <EXPRESSION>`

• Once the compiler sees this, anywhere <SYMBOLNAME> is encountered it is immediately replaced with <EXPRESSION>

```
#define oldStyleConstant   4
main()
{
  if (oldStyleConstant < 5)
…
}
```

A Few Words on Pre-Processing (cont)

• `#define`s can be in any of the following forms:

• Note that there are no semi colons after expressions

```
#define aConstant   5
#define min(x,y)  (x < y) ? x : y
#define foobar      // doesn't work on all compilers
#define aString     "We only have five more classes left"
```

• Bascially the text that appears as part of "expression" is substituted in the source file whenever the "symbol" is encountered.

• There is no type checking involved, it's a straight substitution.

• The #ifdef/#endif pair is simpler.

• The syntax is:

  – `#ifdef <SYMBOL>`
  – `<Your code here>`
  – `#endif`

A Few Words on Pre-Processing (cont)

• #ifdef/#endif is a way to include or exclude code based on some compile time condition.

• So, for some cross-platform code you might see something like this:

```
// Assume that, depending on the system we are compiling for
// we've defined one of the following: MACINTOSH, WINDOWS,
// or UNIX

#ifdef MACINTOSH
#define DIRECTORYSEPARATOR ':'
#endif
#ifdef WINDOWS
#define DIRECTORYSEPARATOR '\\'
#endif
#ifdef UNIX
#define DIRECTORYSEPARATOR '/'
#endif
```

A Few Words on Pre-Processing (cont)

• Now consider the following technique for tackling the multiple definition errors you've gotten in the past:

```
// MyString.h
//
// #ifndef is means "if symbol is NOT defined"
#ifndef _MYSTRING
#define _MYSTRING

class MyString
{
public:
…
};

#endif
```

Back To Multiple Inheritance

- Multiple base classes may share member function names, but...
- The following code will generate a compiler error:

```
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 {
};
main()
{
  C c;
  c.printSomething();
};
```

Back To 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!
};
```

Final Thoughts

- Prelim #2 11/23
- Final Projects!
  - Example:
    - Write a console based checkers game using the Matrix homework.
  - Should feel like 3-4 homework assignments
  - Only needed if you do not have at least a “70” average assuming you take a 0 for the final project.