Example: IntWrapper.cpp

- Class definition
- public/private
- Constructor (with default value(s))
- Don’t forget the semi-colon at the end!

Separate Compilation, Revisited

As always, we prefer to separate the class **interface** from it's **implementation**.

Typically for class "foo":

- foo.h – declarations that are needed to compile any module that uses foo
- foo.cpp – implementation(s) of foo's member function(s).

Example: separatedIntWrapper

- intWrapper.h
- intWrapper.cpp
- Scoping operator
- #include “intWrapper.h” goes everywhere this class will be used

Annoying Problem

Suppose:

- Class **A** makes use of classes **B** and **C** in such a way that the file A.h must begin with:

```
#include "B.h"
#include "C.h"
```

- Class **B** uses **C** in such a way that the filethat B.h must start with:

```
#include "C.h"
```

What happens when we try to compile class A?

Standard Header File Trick

Foo.h looks like this:

```
#ifndef FOO_H
#define FOO_H
…
// Usual stuff goes here…
…
#endif
```

- No matter how many copies are #included together, it will only be fed to the compiler **once**.
- **Use this trick for all header files!**

More details about const…

(SeparatedIntWrapper.cpp, continued…)

Our IntWrapper class looks pretty good, right?

Wrong. Let's look at **driver2.cpp**…

The compiler is being very careful…

Mutators vs. Accessors

Some member functions mutate the state (mutators), others do not (accessors).

The compiler can't tell which is which and it doesn't want to run a mutator on a **const** object.

Rule: If your object is **const**, then you can only run member functions marked as **const**. (These are the accessors.)

We must always mark all accessors as const!

Use Const-ant Good Habits!

- Declare variables **const** unless the variable will modify its value.
- In particular: Use **const** references for function parameters that will not be modified by the function.
- If a member function will not modify the current object, declare it as **const**

Although viewed by some as inconvenient and obtrusive, experienced programmers view C++ const-enforcement as a major advantage over Java.

3 Kinds of Implicit Constructor Calls

Three kinds of constructors are called **implicitly**, as needed:

1. **The Default (no-arg) constructor** is called implicitly when:

- When an object is instantiated with no arguments
- When an array of objects is created
- When an "interior object" is constructed (later today…)
- When a derived class does not explicitly call the base class constructor (later next week…)
- Frequently used in the STL for initial values (later if there is time…)

3 Kinds of Implicit Constructor Calls

2. **The Copy Constructor** is called implicitly when:

- When passing an object by value
- When returning an object by value
- When "thrown" (Later, if there is time…)
- When variable is declared and simultaneously initialized:

```
Cat x = y; // This is NOT the assignment operator
```

Note: If you don't write one, the system provides one for you that does a shallow copy.

Note: The compiler may sometimes optimize-out the copying!

3 Kinds of Implicit Constructor Calls

3. **Conversion Constructor(s)**

Any constructor that takes **one** parameter is a "conversion constructor":

```
Foo(Bar x) {…}
```

- Called whenever a Foo is expected, but a Bar is provided. **It will automatically construct a Foo from a Bar!**

- Can be disabled by labeling the declaration as "explicit":
  `explicit Foo(Bar x);`

Example: ImplicitConstructors.cpp

- Default Constructor
- Copy Constructor
- Conversion Constructor

Objects within Objects? Not in Java!

Java:

```
class CatOwner {
  String name;
  Cat pet;
}
```

A CatOwner object contains two references to externally stored objects.

In Java it is impossible to have an object inside another!

Objects within Objects in C++

C++:

```
class CatOwner
{
  string name;
  Cat pet;
}
```

A CatOwner object actually contains a string object and a Cat object!

Objects within Objects: Constructors

When an object is a member of another, the inner object is constructed BEFORE the outer object's constructor runs!

Potential constructor:

```
CatOwner(string nameIn, Cat petIn)
{
  name = nameIn;
  pet = petIn;
}
```

This constructor may look good to a Java programmer, but in C++…

How many times will the state of a Cat be initialized?
How many times will the state of a string be initialized?

Compound Objects: Constructors

First fix: Pass arguments by (const) reference, not by value.

```
CatOwner(string const &nameIn, Cat const &petIn)
{
  name = nameIn;
  pet = petIn;
}
```

Still… "Double Initialization" occuring:

1. Inner objects are constructed using default (no-arg) constructors
2. Assignment overwrites these objects

Member Initialization Lists

To avoid “double initialization” we use **member initialization lists**.

```
CatOwner(string const &nameIn, Cat const &petIn) :
   name(nameIn), pet(petIn)
{
}
```

We are explictly calling constructors for each member.

Initializers run before the code for **this** constructor.

The only time NOT to use member initializers is when default construction is desired.

Example: MemberInitializers

Let's carefully look at:

- Cat.h
- Cat.cpp
- CatOwner.h
- CatOwner.cpp
- Driver.cpp

When you MUST use Member Initializer

- Member does not have a default constructor
- Member is declared as const
- Member is declared as a reference