UNBC Laboratory Assignment
Computer Science 101—Winter 2007

Forward Class Declarations and the Compiler Firewall Idiom

Purpose:

To understand two of the purposes of using forward class declarations: decoupling tangled classes, and hiding class information from the compiler, and from class users.

Due Date:

This assignment is due 5 March 2007 *at the beginning of lecture.*

Forward Class Declarations

Recall that it is possible to say that `Person` will be a class without actually listing the members of the class, by using a *forward class declaration*, namely,

```
class Person ;
```

In this set of exercises, we explore two reasons why forward class declarations are useful.

Tangled Classes

The C++ programming allows objects of one class to contain physically member variables that are objects of a second class (this contrasts with JAVA). Of necessity, these containment relations are linearly ordered. If a `Firetruck` object contains an `Engine` object, then the **`Firetruck.h`** file must include the **`Engine.h`** file. More generally, if you wish to declare a variable of a particular class type, or if you wish to use the members of an object of a particular class type, you must be able to see the complete class declaration at that point.

However, sometimes you may have two or more classes that refer to each other in some way other than physical containment. For the sake of amusement, assume that we are modelling `Snakes` and `Lizards`, and that they can eat each other, and that once eaten, they die. Thus the `Snake` class has member functions something like those shown in Figure 1, and the `Lizard` class has symmetric functions. In order for the **`Snake.cpp`** file to be able to call `Lizard::die`, it must include **`Lizard.h`**. The file inclusion structure ends up looking like that shown in Figure 2.

This **`.pdf`** created February 25, 2007.

```
void Snake::eat(Lizard& godzilla)
{
    cout << "A snake is about to eat a lizard" ;
    godzilla.die() ;
}

void Snake::die()
{
    if (!dead++)
        cout << "A snake dies."
}
```

Figure 1: `Snake` member functions

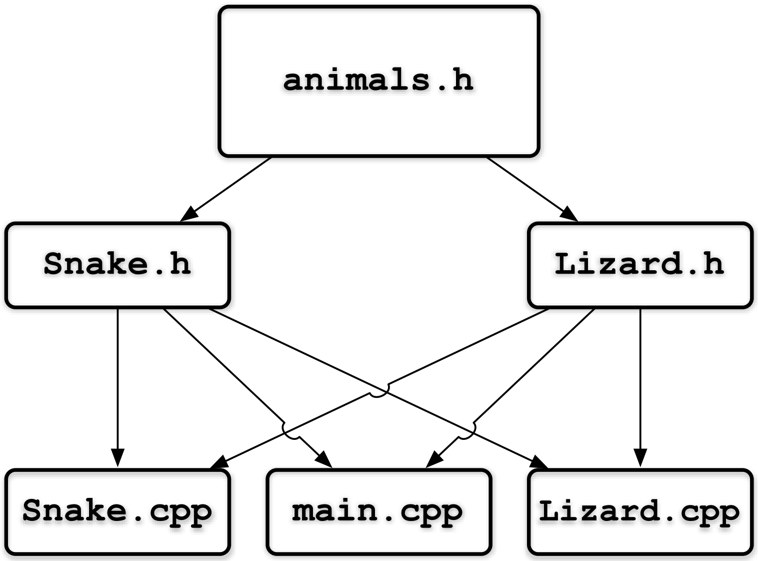

Figure 2: File inclusion diagram

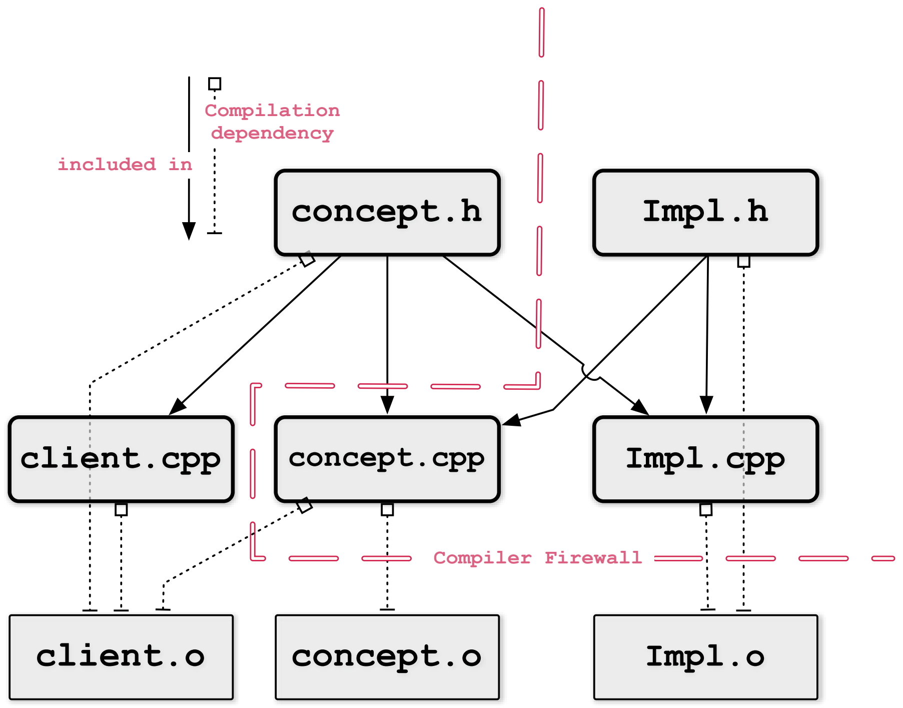

Figure 3: The Compiler Firewall idiom

Programming:

Implement a very small program consisting of the six files shown in Figure 2, that at least give each of the two animal classes the functions illustrated in Figure 1. Your driver program should show the unhappy fate of at least one snake and one lizard.

The Compiler Firewall Idiom

The compiler firewall idiom is a programming strategy that uses forward class declarations to make the private section of a class in effect even more private. It can be used for two purposes:

1. to prevent changes in the implementation of a class from affecting the clients of the class. In particular the clients neeed not be recompiled even if the implementation objects change shape.
2. to allow implementers of a class to keep all of the details of a class, including its list of private member variables, secret.

The compiler firewall idiom works by separating a class into a *proxy* class (corresponding to the **`concept.h`**, **`concept.cpp`** and **`concept.o`** files in Figure 3) that provides the public interface, and the implementation class (corresponding to the **`Impl.h`**, **`Impl.cpp`** and **`Impl.o`** files in Figure 3).

The proxy interface file

When you are using a *proxy* class in the compiler firewall idiom, its **.h**-file looks something like

```
#if !defined(Date_INCLUDED)
class DateImplementation ; // does all the work

class Date   {
    Date& operator=(const Date&) ; // forbids assignment of Date Objects
public:
    // constructors, etc. At least one for each of DateImplementation
    Date() ;
    Date(const Date&) ;         // must be defined, see elsewhere
    ~Date() ;                   // must be defined.
    // usual public member function declarations
private:
    DateImplementation * impl_ptr ;
} ;
#define Date_INCLUDED
#endif  // !defined(Date_INCLUDED)
```

Note that

- We avoid needing access to the **DateImplementation.h** file by forward declaring the class, and only using a pointer to it in the implementation.
- The shape of the `Date` class will not change regardless of changes to the `DateImplementation` class because`Date` objects always consist of exactly one pointer.
- We must explicitly declare all of the constructors and destructors of the `Date` class, even if they would normally be compiler supplied.

The proxy definition file

Note that the proxy definition file is behind the compiler firewall. It needs to include the declaration of the implementation class, because it refers to member functions of the implementation class.

Ordinary member functions

Almost all of the member functions of the proxy class simply refer to the implementation to do the work. For instance, one might write:

```
  int Date::getDay() const {return impl_ptr->getDay() ;}
```

or

```
  void Date::setDay(int d) {impl_ptr->setDay(d) ; return ;}
```

(the code here is condensed for space reason; this is generally not good coding style.)

Constructors and destructors

Unfortunately constructors and destructors are more complicated, because the implementation objects must live on the heap. The zero-argument constructor and copy constructor must be explicit and are likely coded as

```
Date::Date()
    : impl_ptr(new DateImplementation())
{}

Date::Date(Date const& rhs)
    : impl_ptr(new DateImplementation(*rhs->impl_ptr))
{}
```

Again, note how the `Date` class is really just referring the work to the corresponding `DateImplementation` class functions. Other constructors in the `DateImplementation` class must have corresponding analogs in the `Date` class. The destructor must free up the implementation storage

```
Date::~Date()
 {   delete impl_ptr ; impl_ptr = 0 ;   }
```

Although this looks ahead to Chapter 8, the assignment operator can be made public and coded as:

```
Date& Date::operator=(Date const& rhs)
{
    *impl_ptr = *rhs.impl_ptr ;
    return *this ;
}
```

Optional programming exercise

The following programming exercise is optional. You are not required to hand it in, although you are responsible for understanding the ideas that it contains.

In Lab assignment 4 you implemented two different versions of a **`Date`** class. By renaming them, use these to provide two distinct implementations of a **`DateImplementation`** class.

- Write a `Date` class that acts as a proxy class, using the ideas provided above. (The `tomorrow()` and `yesterday()` member functions require some additional thought.)

- Your driver and/or test programs from Lab assignment 4 should work with the new `Date` class. Test that your `Date` proxy class, one of your `DateImplementation` classes and your driver program work together.
- Show, through the use of a **`Makefile`** or otherwise, that you can substitute the **`.o`**-file of one of the `DateImplementation` objects for the other and relink the executable without being forced to recompile any code.

Review Questions:

You should now be able to answer the following questions about the compiler firewall idiom. *Do not hand these in*, but be prepared to answer similar questions on Midterm II.

1. If we did not use the compiler firewall idiom, why would we be forced to recompile the client program (**`client.o`** in Figure 3, the driver programs in the optional programming exercise) when we changed the implementation of the class (`Date` in Lab Assignment 5)?
2. Why must we use copy constructors in the proxy class?
3. Why must the proxy class implementation be able to see the full implementation class declaration?
4. Using a proxy class as in the compiler firewall idiom, how would you write a function like `Date::tomorrow()` that returns a `Date` object?