Using Incomplete (Forward) Declarations

David Kieras, EECS Dept., Univ. of Michigan

December 19, 2012

An *incomplete declaration* is the keyword `class` or `struct` followed by the name of a class or structure type. It tells the compiler that the named class or struct type exists, but doesn't say anything at all about the member functions or variables of the class or struct; this omission means that it is a (seriously) incomplete declaration of the type. Usually the compiler will be supplied with the complete declaration later in the compilation, which is why an incomplete declaration is often called a *forward* declaration - it is an advance forward announcement of the existence of the type. Since an incomplete declaration doesn't tell the compiler what is in the class or struct, until the compiler gets the complete declaration, it won't be able to compile code that refers to the members of the class or struct, or requires knowing the size of a class or struct object (to know the size requires knowing the types of the member variables)

Use an incomplete declaration in a header file whenever possible. By using an incomplete declaration in a header file, we can eliminate the need to `#include` the header file for the class or struct, which reduces the coupling, or dependencies, between modules, resulting in faster compilations and easier development. If the .cpp file needs to access the members of the class or struct, it will then `#include` the header containing the complete declaration.

In the following discussion, `X` will be the class or struct type that is being incompletely declared, and A.h will be the header file that contains the incomplete declaration of `X`. A.cpp will be the implementation file which will typically `#include` the complete declaration in X.h so that it can make full use of `X`. Instead of "class or struct", we'll just say "class" in what follows.

When will an incomplete declaration work in a header file?

1. *If the class type* `X` *appears only as the type of a parameter or a return type in a function prototype*. When it comes time for the compiler to actually compile a call to the function in A.cpp, it will have the complete declaration `X` so that it knows how to generate the code for the call - for example, it will need to know how big an object to push on to the function call stack. But the header file needs only the incomplete declaration to successfully declare a function that takes an `X` parameter or returns an `X` value. For example, we could declare a function prototype in A.h like this:

```
class X;
X foo(X x);
```

2. *If the class type* `X` *is referred to only by pointer (*`X*`*) or reference (*`X&`*), even as a member variable of a class declared in A.h*. The compiler does not have to know how big an object is, nor what its contents are, to understand a pointer (or reference) to that type of object (references are usually implemented in terms of pointers). This is because an address is an address, regardless of what kind of thing is at that address, and addresses are always the same size regardless of what they point to. The compiler can enforce type-safety of pointers perfectly well without having to know anything more than the type name about the pointed-to objects. So A.h could contain something like the following:

```
class X;
class A {
/* other members */
private:
    X* x_ptr;
    X& x_ref;
};
```

Of course, any code that actually dereferences the pointer, or uses member variables or functions of `X`, or requires knowing how big `X` is, will require the complete declaration; thus the A.cpp file will typically `#include "X.h"`.

3. *If you are using an opaque type* `X` *as a member variable of a class declared in A.h*. This is a type referred to only through a pointer, and whose complete declaration is not supposed to be available, and is not in any header file. Thus an incomplete declaration of the type is the only declaration your code will ever make or need either in A.h or A.cpp.

When will an incomplete declaration *not* work in a header file?

1. *If your A.h header file declares a class* `A` *in which the incompletely declared type* `X` *appears as the type of a member variable*. The class type `A` itself cannot be completely declared unless the compiler at least knows how big an object of that type is, which requires that all of the the member variable types be completed declared. The following will produce a compile error:

```
class X;
class A {
private:
    X x_member; // error - can't declare a member variable of incomplete type!
};
```

2. *If your A.h header file declares a class* `A` *in which the incompletely declared type* `X` *is a base class (*`A` *inherits from* `X`*)*. The class type `A` itself cannot be completely declared unless the compiler at least knows how big an object of that type is, which requires that it know the types of all of the the member variables in the base class; the complete declaration is necessary for this. So the following also fails to compile:

```
class X;
class A : public X { // error - base class is incomplete type!
```

3. *If you don't actually know the name of the type*. You can't forward declare a type unless you know its correct name. This can be a problem with some of the types defined in the Standard Library, where the normal name of the type is actually a `typedef` for a particular template instantiated with some other type, usually with multiple template parameters. For example, the following will not work to incompletely declare the `std::string` class:

```
class std::string;
```

This won't work because `std::string` is actually a `typedef` for the `std::basic_string<>` template instantiated to work with `char` data (as opposed to wide character data). Similarly, the `std::istream`, `std::ostream`, and other members of `<iostream>` are `typedefs` for templated classes instantiated for `char` data. Coming up with the correct incomplete declaration of these types involves knowing some pretty cryptic stuff about the exact template parameters involved.

Fortunately, the Standard Library provides a header file that contains a complete set of forward declarations for the `<iostream>` types, called `<iosfwd>`. The rule is to `#include <iosfwd>` instead of `<iostream>` whenever possible. Unfortunately, there is no such handy forward declaration file for the `std::string` family, so plan on `#including <string>` to access the complete declaration – an incomplete declaration is not practical in this case. So the following is an example of what you must put in a header file in both of these cases:

```
    #include <iosfwd>   // forward declarations of iostream classes
    #include <string>   // complete declaration of string class

    void write_string_to_file(const std::string& label_text, std::ofstream& outfile);
```

Declaring classes (or structs) that refer to each other

Forward declarations are essential for the following problem: Suppose we have two classes whose member functions make use of either parameters or member functions of the other class. Here is a simple, but nasty example - only a couple of member functions are shown, but having additional member variables and functions does not change the problem.

```
class A {
public:
     void foo(B b)    // Ex. 1: a parameter of the other class type
         {
            b.zap();  // Ex. 2: call a member function of the other class
         }
     void goo()
         {/* whatever */}
};

class B {
public:
     void zot(A a)    // Ex. 3: a parameter of the other class type
         {
            a.goo();  // Ex. 4: call a member function of the other class
         }
     void zap()
         { /* whatever */ }
};
```

The compiler will balk when it tries to compile this bit of code. The problem is that when it compiles the declaration of class `A`, it won't be able to understand the line labeled Ex. 1 - it hasn't seen a declaration of `B` yet. Obviously it would have a problem with Ex. 2, because it doesn't know about function `zap` either. But if the compiler could somehow understand the class `A` declaration, it would then have no problem with class `B`, because it would already know about class `A` when it sees Ex. 3 and Ex. 4. However, since the compiler can't compile the class `A` declaration, it will not be able to compile the class `B` declaration either.

Sometimes you can fix this *declaration-ordering problem* by simply putting the declarations in reverse order, so that the compiler has already seen everything it needs to know when it sees each declaration. But in this diabolical example, reversing the declarations won't work because class `B` uses things in class `A` - we would just get the same compiler error messages on the other class. So we might as well stick with the declarations in this order. To save space in what follows, the parts of the example declarations that aren't directly relevant will be omitted.

What we need is some way of telling the compiler *just enough* about class `B` to allow it to compile the class `A` declaration, and put off requiring any more information about `B` until it has processed the complete `B` declaration.

Incomplete declarations to the rescue! Oops, not yet

Adding a forward declaration of `B` to the above example would look like this:

```
class B; // forward incomplete declaration - class B will be fully declared later

class A {
public:
        void foo(B b)    // Ex. 1: a parameter of the other class type
        {
           b.zap();      // Ex. 2: call a member function of the other class
        }
        /* rest omitted to save space */
};

class B {
public:
/* rest omitted to save space */
};
```

This helps, but there is still a problem. When the compiler sees line Ex. 1, it is happy with the class name `B` because it has been told that `B` is the name of a class. But Ex. 2 is still a problem because the compiler hasn't seen the whole class declaration of `B`, and so doesn't know how to generate machine code for a call to the function named `zap`. This is a fatal compile error. So just providing the name of to-be-declared class in a forward declaration is not enough in this case.

A bit of re-arranging does the trick!

Here's how we solve this problem. We take the function definitions out of the first class declarations, so that the compiler will see all of the second class before it has to compile the code for the first class functions. Here is how the example would look:

```
class B; // forward incomplete declaration - class B will be fully declared later
class A {
public:
     void foo(B b); // Ex. 1:
     /* rest omitted to save space */
};

class B {
public:
/* rest omitted to save space */
};

// the function definition later or in a separate .cpp file
void A::foo(B b)  // Ex. 1B: define A's foo function after the B declaration
{
     b.zap();      // Ex. 2: call a member function of the other class
}
```

This example now compiles successfully; here is the story: When the compiler sees line Ex. 1, which is now just a function prototype, it knows that `B` is the name of a class (from the forward declaration), and since we have only a function prototype here, we are no longer trying to call the still-unknown `zap` member function in `B`. The compiler simply records the `foo` prototype and continues. The compiler can then handle the class `B` declaration with no problem because it got all it needed from the class `A` declaration.

In line Ex. 1B, class `A`'s function `foo` is defined outside of the class `A` declaration, *after* the compiler has seen the complete class `B` declaration. Notice the scope operator `::` that tells the compiler that this `foo` is the one prototyped in the class `A` declaration. The compiler can handle the previously problematic Line Ex. 2 because it is now knows about B's zap member function.

Member variables in related classes: A lot messier

The above story is about function calls and parameters. What about member variables? Let's use the same classes, and try to declare member variables whose type is the other class:

```
class B; // forward incomplete declaration - class B will be fully declared later

class A {
public:
        /* rest omitted to save space */
private:
     B b_member; // Ex. 5
};

class B {
public:
/* rest omitted to save space */
private:
     A a_member;  // Ex. 6
};
```

If you guessed that the compiler will choke in the declaration of `A` at Ex. 5, you are right! In spite of the incomplete declaration of `B`, the compiler can't fully comprehend the declaration of `A` because it requires knowing the definition (at least the size) of a `B` object, which it hasn't seen. In contrast, Ex. 6 will be fine, if we can get class `A`'s declaration to work. We can get correctly compiling code by using a pointer to a `B` object as a member variable instead of a `B` object, taking advantage of the properties of pointer types mentioned above:

```
class B; // forward incomplete declaration - class B will be fully declared later

class A {
public:
        /* rest omitted to save space */
private:
     B* b_member; // Ex. 5 - no problem
};

class B {
public:
/* rest omitted to save space */
private:
     A a_member;  // Ex. 6 - no problem
};
```

Using a pointer to the `B` object gives us correctly compiling code, but we have a new problem: When a `B` object comes into existence, it will automagically get a fully initialized `A` object as a member variable. In contrast, when an `A` object comes into existence, it will not automatically get a fully functioning `b_member` object to use. Your code (e.g. in the `A` constructor) will have to know of or create (e.g. with `new`) a suitable `B` object and set the `b_member` to point to it. This is ugly, but in this situation, it's the best you can do.

What about using a `B&` reference instead of a `B*` pointer for `b_member`? It will also give correctly compiling code, and the same problem will arise that when an `A` object is created, a `B` object needs to be in existence for the reference to refer to. The compiler insists that you initialize a reference-type variable with the referred-to object at the point of definition. References must always refer to a single something - they can't be changed to refer to a different object. The only way to initialize a reference-type member variable is in a constructor initializer, which means that the `B` object needs to exist before the `A` constructor is called. This set of constraints means reference-type member variables are usually inconvenient, and so you rarely see them, and generally only when the clean syntax of a reference compared to a pointer is especially useful (such as for `ostream` objects).

A final note: Often the explicit incomplete declaration of a `class` or `struct` is redundant, because the first declaration of a pointer or reference using a `struct` or `class` name *implicitly* makes an incomplete declaration. However, it is good programming practice to write the explicit incomplete declaration to make it obvious to the reader that the `class` or `struct` declaration is not yet known at this point. Otherwise, they might be thinking "Did I miss something? Is there something in one of the `#included` .h files I didn't know about?"