Using Incomplete (Forward) Declarations

David Kieras, EECS Dept., Univ. of Michigan
Prepared for EECS 381, Fall 2004

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 members of the class or struct or their types; this omission means that it is a (seriously) incomplete declaration of the type. Usually the compiler will be supplied with the complete declaration later, 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, the compiler 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).

However, pointers or C++ references to the incompletely declared type are completely understandable to the compiler - it does not have to know how big an object is, or what its contents are, to know how to compile a pointer (or reference) to that type of object (reference 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. Finally, 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.

Hiding information with incomplete declarations

Incomplete declarations can be used to hide information about the contents of a class or struct, and thus provide encapsulation, decoupling, and insulation - keeping different modules of code out of each other's way, and limiting the propagation of changes through the code in a large project. The idea is that you can freely refer to pointers or references to an incompletely declared class or struct as long as your code doesn't require knowing how big the pointed-to object is, or the names of any of its members. So for example, if the code just has to store and retrieve pointers to objects, but not access the information in the objects, it doesn't have to know anything about the nature of the pointed-to objects. Thus the code that maintains the container of pointers can be kept separate from the code that works with the content of the objects - we can change one module with absolutely no effect on the other - not even a recompilation (if we have set up the modules properly - see the handouts on *Header File Guidelines*). Since C does not have access controls like "private," incomplete declarations provide a good encapsulation technique for C programs.

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 almost worst-case example - only a couple of member functions are shown, but having other 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 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 - what in the world is a "B"? It hasn't seen the 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 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. However, function foo takes an argument of type B. To compile this function, the compiler needs to know how much space on the function-call stack a B-type argument will require. Since it hasn't seen the complete declaration of B yet, it has no way of knowing. So it will complain, saying something like "Error: illegal use of incomplete class."

Likewise, 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 another compiler error. So just providing the name of to-be-declared class in a forward declaration is not enough in this case.

Reference or pointer parameters, and a bit of re-arranging

Here's how we solve this problem. First, we use only reference or pointer type parameters for the forward-declared class. Second, 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: Prototype with a reference parameter of the other class type

        /* rest omitted to save space */
};
```

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

// the function definition appears later, or in a separate .cpp file from the class declaration
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 it knows how to generate a call to function foo, because *reference parameters are always the same size!* A similar situation holds for pointer-type parameters — it doesn't matter what a B object is like, or how big it is, if we use a pointer or reference parameter for it. Also, 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.

Idioms for reference parameters. In C++, the use of a reference type parameter carries a strong suggestion that the function will modify the caller's argument variable - this is an idiomatic way of returning an additional value from a function. However, if the only reason for the reference parameter is to allow the classes to be declared properly, letting the reference parameter stand as-is is misleading. The solution for this is to use a const reference parameter; the above example lines become:

```
void foo(const B& b);  // Ex. 1: Prototype with a const reference parameter of other class

void A::foo(const B& b)// Ex. 1B: define A's foo function after the B declaration
```

The compiler will not allow the caller's argument to be modified, and any experience programmer will recognize this idiom as meaning that the reference is being used for some other reason.

Pointer parameters. The same logic works with pointer type parameters. For example, the following will also compile successfully:

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

class A {
public:
      void foo(B * b); // Ex. 1: Prototype with a pointer parameter of the other class type

      /* rest omitted to save space */
};

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

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
}
```

Because pointers always have the same size, so the compiler can generate a call using a pointer parameter without needing to see the full declaration of the class first. An address is an address, regardless of the type of object at that address. Why do references work? Because the C++ definition says they must, but it is easy to see why because a good way to implement references is with pointers "under the hood."

Member variables in related classes: A lot messier

The above story is about function 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:

```
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 subobject 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 uncommitted (or be changed). Now, 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?"