Forward Declarations in C++

C++ is unique in the languages that we are studying in VGDD because a class is usually declared in one file (the header file -- *.h) and defined in another (implementation file -- *.cpp)[1]. The other languages we look at merely just define the class in code and the run time environment resolves the links to other code modules seamlessly. Because C++ is a fully compiled language, sometimes it is up to the developers to maintain a proper relationship between files so the compiler can fully make sense of what we write. There is no runtime environment to handle this for us.

Let's review a few basic points:

1. Only .cpp files (implementation files) are compiled into object code.
2. Header files are not compiled, but the contents of that file are inserted into the memory copy of any implementation file during the compilation process.
3. This is achieved by using the #include preprocessor directive.

Let's suppose that we are writing a game when one class has a data member that is a pointer to an object of another class. That is straight forward enough, but let's say that the object of the second type has a pointer back to the first. This can be quite confusing to the compiler – about as confusing as the verbal description, so let's build a concrete example.

Chess

We are writing a computer chess game and we are define our classes. Two of these are a `Square` class that holds a pointer to a `ChessPiece` object, it being a base class for all the different types of pieces that there are in chess. If the pointer is `NULL`, that represents an empty square.

[1] Note that there are exceptions to this: small trivial classes are often implemented in header files and template classes must be. Most non-trivial classes will have a separate implementation file.

The file square.h, including the generic compiler guard, will look something like this[2]:

```
#if !defined(SQUARE_H__)
#define SQUARE_H__
#include "chesspiece.h"

class Square
{
//...a bunch of stuff that we can safely omit for this example.
private:
    ChessPiece* _pPiece;
};
#endif
```

The file "chesspiece.h" starts out looking like this:

```
#if !defined(CHESS_PIECE_H__)
#define CHESS_PIECE_H__
//The base class for the specific types of pieces

class ChessPiece
{
//...a bunch of stuff that we can safely omit for this example.
};

#endif
```

The code is found in listing_0.

[2] If you are more used to the Visual Studio specific compiler guards, remove the lines starting `#if`, `#define` and `#endif` and insert `#pragma once` at the top of the file.

As we polish the look of our game, we discover that an object of class `ChessPiece` has to have a reference to a `Square` object in the `Draw(…)` virtual function. We modify our `ChessPiece` class like this:

```
#if !defined(CHESS_PIECE_H__)

#define CHESS_PIECE_H__

#include “square.h”


Class ChessPiece

{

public:

    virtual void Draw(const Square& sqr);

};

#endif
```

…and previously unseen compilation errors jumps up on your screen. Visual Studio gives you two errors in chesspiece.h:

- `chesspiece.h(9) : error C4430: missing type specifier - int assumed. Note: C++ does not support default-int`
- `chesspiece.h(9) : error C2143: syntax error : missing ',' before '&'`

You think “Stupid Language! I **have** included square.h and it’s telling me it doesn’t know what that class is!”

What is going on? From a reading of the code, the problem does not appear. To find the roots of the issue, you need to look into square.h. It hold the following line:

```
#include "chesspiece.h"
```

When the compiler compiles square.h, it reads and inserts the contents of chesspiece.h into its memory copy. It also triggers that files compiler guard so when the compiler attempts to read chesspiece.h again, the result of the include of square.h is nothing. The `ChessPiece` class knows nothing of the `Square` class and the compiler complains.

A Solution?

At first, you might try to rearrange the order of your includes. That might fix this particular problem but is will show up elsewhere when the compiler complains that it doesn't know that a `ChessPiece` is. But you might get lucky and find just the right combination and order of includes. It works…until you throw in another class and its header file. Besides, who has the time to worry about the order of includes or whether a header file is included one, twice, a hundred times. This is not a productive use of your time.

How about removing compiler guards? Now you get errors about having multiple definitions of the same class. "Stupid ***&^$#%*** Language!"

The Real Solution: Forward Declaration

Simply put, the problem is that the includes in your header files have cyclical references: chesspiece.h includes square.h and square.h includes chesspiece.h. When you include compiler guards, then something will be missed.

To fix this, what you need to do is:

- Identify one relationship where one class refers to the other by C++ reference or pointer.
- Declare that class through a forward declaration and remove the include in the header file.
- For the class that is referenced by the forward declaration, move anything except an assignment or a function parameter of that class to the implementation file (.cpp) and include the header file in the .cpp.

listing_2:

square.h:

```
#if !defined(SQUARE_H__)

#define SQUARE_H__

//include removed

//#include "chesspiece.h"

//forward declaration added

class ChessPiece;


class Square

{

     void SetChessPiece(ChessPiece* p);

//...a bunch of stuff that we can safely omit for this example.

private:

    ChessPiece* _pPiece;

};

#endif
```

square.cpp:

```
#include "square.h"

#include "chesspiece.h"


void Square::SetChessPiece(ChessPiece* p)

{

    _pPiece = p;

}
```

A forward declaration tells the compiler that there is a class, which might not yet be compiled, with the name that you have declared. Since a pointer is just a memory address the compiler does not need any functions or data members of the class to be resolved until the implementation file. In this way, it acts for classes much the same as function prototypes act for functions.

Best Practices

One goal of good project design (the way your files are organized) is to remove any sort of dependency between header files so that they can be called in any order you see fit. This is best done by using forward declarations instead of includes in header files whenever you can. It is not always possible to do this, but in many cases this can be done. In the previous example we forward declared `ChessPiece` in square.h. We can also do the reverse: forward declare `Square` in chesspiece.h

Listing_3:

chesspiece.h

```
#if !defined(CHESS_PIECE_H__)

#define CHESS_PIECE_H__

//remove the include

//#include "square.h"

//use a forward declaration

class Square;


class ChessPiece

{

public:

    virtual void Draw(const Square& sqr);

};

#endif
```

chesspiece.cpp:

```
#include "chesspiece.h"

#include "square.h"


void ChessPiece::Draw(const Square& sqr)

{

//....base drawing code

}
```

The Catch

So, why not use forward declarations all the time? The short answer is, you can't – that would be far too simple, wouldn't it? In reality, there are very good technical reasons for this. Remember that C++ is a fully compiled language that does not execute in a runtime environment that can resolve these issues at compile time.

You have to use includes in two common situations:

1. If you are declaring an object in a class declaration, you need to include the header file where that object's class is declared.
2. If you are interacting with and object through its functions or data members in the header file the compiler has to resolve the function calls or data members of that object.

When designed you classes consider using forward declarations when you can. It will reduce the coupling between your classes (a very good thing) and you can spend more time working on your game's cool features instead of struggling with the compiler.