Headers and Source Files: C++ Code Organization

This reading explains a few details to keep in mind about how C++ programs are structured in code. Because a C++ program's source code exists across several separate files, you need to know how the compiler will pull those pieces together.

If you look at the example source code provided so far, you'll see that C++ code files often come with two different file extensions:

.h files are "header files". These usually have definitions of objects and declarations of global functions. Recently, some people name header files with a ".hpp" suffix instead.

.cpp files are often called the "implementation files," or simply the "source files". This is where most function definitions and main program logic go.

In general, the header files contain *declarations* (where classes and custom types are listed by name and type, and function prototypes give functions a type signature) while the source files contain *implementation* (where function bodies are actually defined, and some constant values are initialized). It becomes easier to understand this organizational separation if you know more about how the code is compiled into a program you can run. Let's look at the **cpp-class** example from the provided code.

The Cube.h header file has this content:

```
/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

// All header (.h) files start with "#pragma once":
#pragma once

// A class is defined with the `class` keyword, the name
// of the class, curly braces, and a required semicolon
// at the end:
class Cube {
  public:  // Public members:
    double getVolume();
    double getSurfaceArea();
    void setLength(double length);

  private: // Private members:
    double length_;
};

```

Note the **#pragma once** at the beginning. Instructions beginning with **#** are special commands for the compiler, called preprocessor directives. This instruction prevents the header file from being automatically included multiple times in a complex project, which would cause errors.

This header file also includes the declaration of the Cube class, including listing its members, but the definition doesn't give the full source code of the functions here. For example, the body of the setLength function is not written here! Only the signature of the function is given, to declare its input argument types and return type. Instead, the function will be defined in the Cube.cpp source code file.

Here is the Cube.cpp source code file:

```cpp
/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include "Cube.h"

double Cube::getVolume() {
  return length_ * length_ * length_;
}

double Cube::getSurfaceArea() {
  return 6 * length_ * length_;
}

void Cube::setLength(double length) {
  length_ = length;
}
```

Notice that the first thing it does is **#include "Cube.h"** to include all the text from the Cube.h file in the same directory. Because the filename is specified in quotes, as "Cube.h", the compiler expects it to be in the same directory as the current file, Cube.cpp. Below we'll see a different way to refer to filenames using **< >** instead.

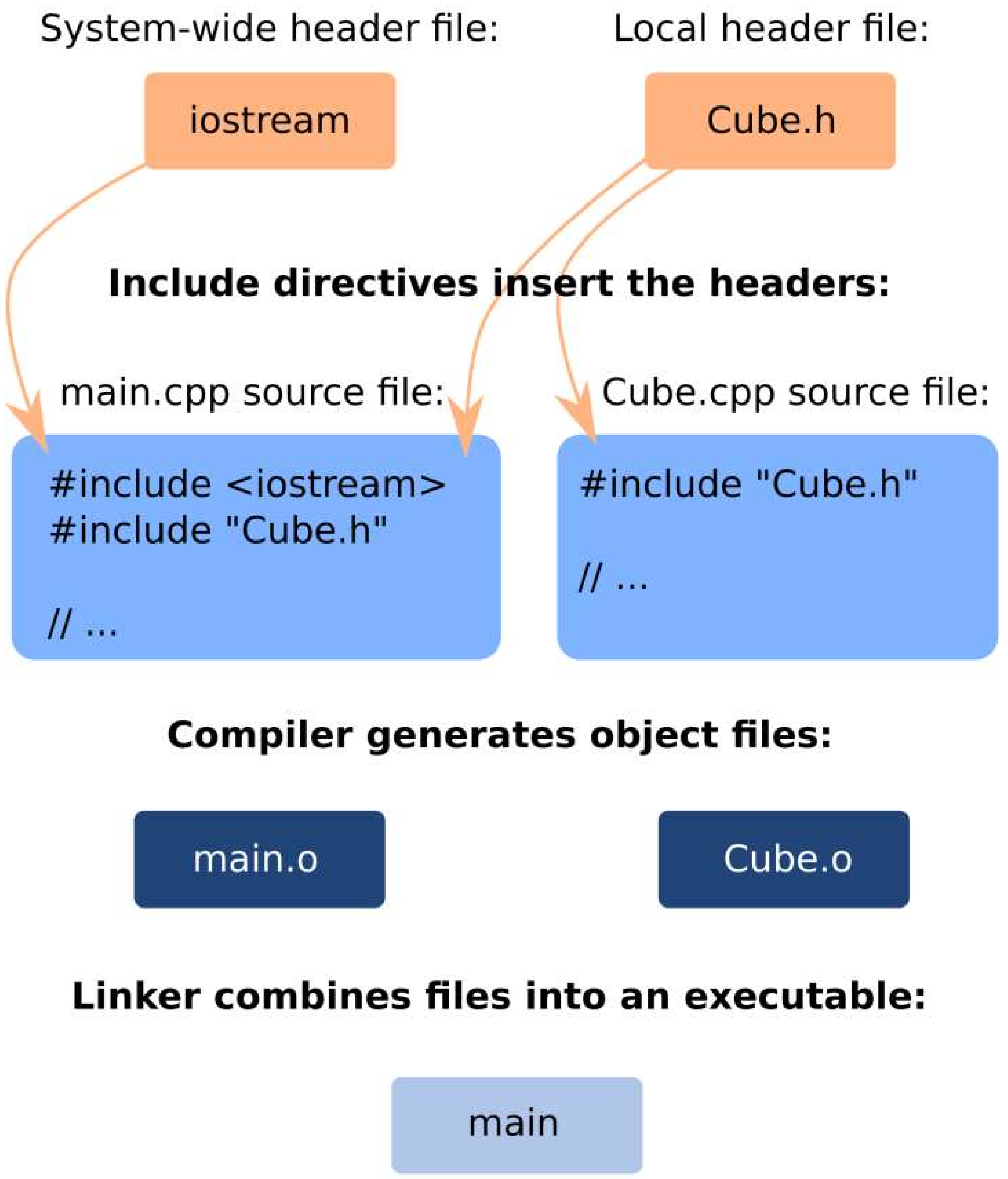
Then, the file gives the definitions for the functions from the class definition before. These full source code listings for the functions that were declared previously are called the implementation. (It is common that function bodies will be implemented in the cpp file and *not* in the h file, but sometimes *short* class function bodies will be defined directly in the h file where they are declared inside the class declaration. The compiler handles that situation in a special way automatically so that it doesn't cause problems in the linking stage, which is described below.)

Finally, let's look at main.cpp:

```cpp
/**
 * C++ code for creating a Cube of length 2.4 units.
 * - See ../cpp-std/main.cpp for a similar program with print statements.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>
#include "Cube.h"

int main() {
  Cube c;

  c.setLength(3.48);
  double volume = c.getVolume();
  std::cout << "Volume: " << volume << std::endl;

  return 0;
}

```

This time there are two #include directives! First, it includes a standard library header from the system directory. This is shown by the use of **< >**. When you write **#include <iostream>**, the compiler will look for the **iostream** header file in a system-wide library path that is located outside of your current directory.

Next, it does **#include "Cube.h"** just like in the Cube.cpp file. You have to include the necessary headers in every cpp file where they are needed. However, you shouldn't use #include to literally include one cpp file in another! **There is no need to write #include "Cube.cpp"** because the function definitions in the Cube.cpp file will be compiled separately and then *linked* to the code from the main.cpp file. You don't need to know all the details of how this works, but let's look at a diagram that shows the general idea:



```
#include <iostream>
#include "Cube.h"

// ...
```

```
#include "Cube.h"

// ...
```

The Cube.cpp files and main.cpp files make requests to include various header files. (The compiler might automatically skip some requests because of **#pragma once** to avoid including multiple times in the same file.) The contents of the requested header files will be temporarily copied into the cpp source code file where they are included. Then, the cpp file with all of its extra included content will be compiled into something called an **object file**. (Our provided examples keep the object files hidden in a subdirectory, so you don't need to bother with them. But, if you see a file that has a **.o** extension, that is an object file.) Each cpp file is separately compiled into an object file. So, in this case Cube.cpp will be compiled into Cube.o, and main.cpp will be compiled into main.o.

Although each cpp file needs the appropriate headers included for compilation, that has to do with checking type information and declarations. The compiled object files are allowed to rely on *definitions* that appear in the other object files. That's why it's okay that main.cpp doesn't have the Cube function definitions included: as long as main.cpp does know about the type information from the Cube function signatures in Cube.h, the main.o file will be "**linked against"** the compiled definitions in the Cube.o file. (I don't know why we say "link against" instead of "link with" when we talk about this, but that is the usual terminology!) The linker program will also link against system-wide object files, such as for **iostream**. After the compiler and linker programs finish processing your code, you will get an executable file as a result. In this case, that file is simply named **main**.

Fortunately, you won't have to configure the compiler manually in this course. We will provide a **Makefile** to you for each project, which is a kind of script that tells the compiler how to build your program. We'll talk more about that in the next reading lesson.