Class with separate header file and implementation file

This document is a description of the following source code files from the **int_array_sep_files** directory that was posted today. I wanted to spend more time on this in class.

This directory contains the following files.

- IntArray.h
- IntArray.cpp
- main.cpp

C++ class definitions/file structure can take a different format from what you are used to in Java. Java combines the ***interface*** (method signatures) and the ***implementation*** (method definitions) into the same the source code file. You can do the same thing in C++ and most of the examples in Runestone follow this format.

The more common structure is to segment a class definition into two separate files. One file containing the interface and a separate file containing the implementation. The file containing the interface specification is called a header file and is stored in a file with a **.h** extension. You can see this in the file **IntArray.h**. Notice that this file is similar to a "table of contents" and simply contains variable definitions and method signatures. Programmers use these to communicate to the compiler, the things that it may "see" during its task.

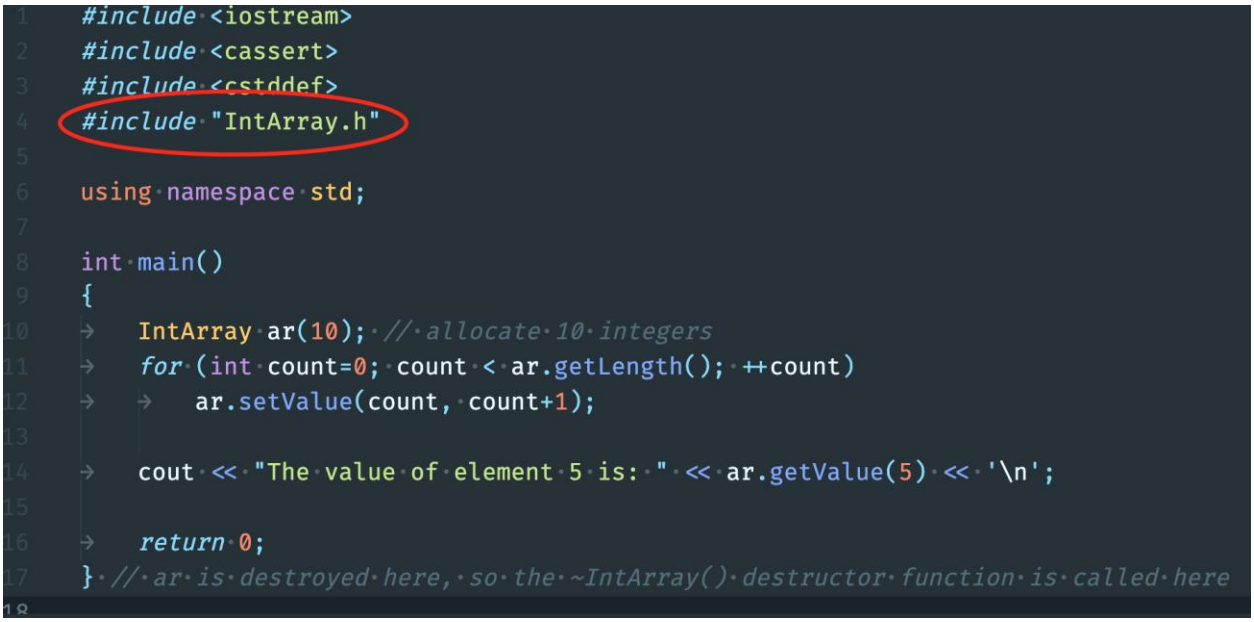
```
#ifndef INT_ARRAY_H
#define INT_ARRAY_H

class IntArray{
    private:
        int *m_array;
        int m_length;

    public:
        IntArray(int length);
        ~IntArray();
        void setValue(int index, int value);
        int getValue(int index);
        int getLength();
};
#endif
```

The **#ifndef, #define, #endif** directives are called "header guards". These directives protect against duplicate definitions of features that may occur when a header file is included into an

application more than once.

See Here: https://www.learncpp.com/cpp-tutorial/header-guards/
https://en.wikipedia.org/wiki/Include_guard

Notice that this header file can be included into other source code files for this exact purpose. Inspect **main.cpp** to see this



This separation of interface from implementation offers a lightweight way to communicate what is in a particular class. The **implementation** of the method signatures defined in a header appear in a file with a **.cpp** extension. Check out **IntArray.cpp** to see this in action. Notice that we must also include the appropriate header file to communicate what we are implementing. Notice that we have to use the **scope resolution operator ::** to further define the connection between the **names** of the items we are implementing and the header in which they are defined. The identifiers that we are providing definitions for are in the “IntArray” namespace.

```cpp
#include <iostream>
#include "IntArray.h"

IntArray :: IntArray(int length) {
    std::cout << "Constructor called" << std::endl;
    assert(length > 0);

    m_array = new int[static_cast<std::size_t>(length)];
    m_length = length;
}

IntArray :: ~IntArray(){
    std::cout << "Destructor automatically called" << std::endl;
    // Dynamically delete the array we allocated earlier
    delete[] m_array;
}

void IntArray :: setValue(int index, int value) { m_array[index] = value; }
int IntArray :: getValue(int index) { return m_array[index]; }
int IntArray :: getLength() { return m_length; }
```

So now we have three source code files

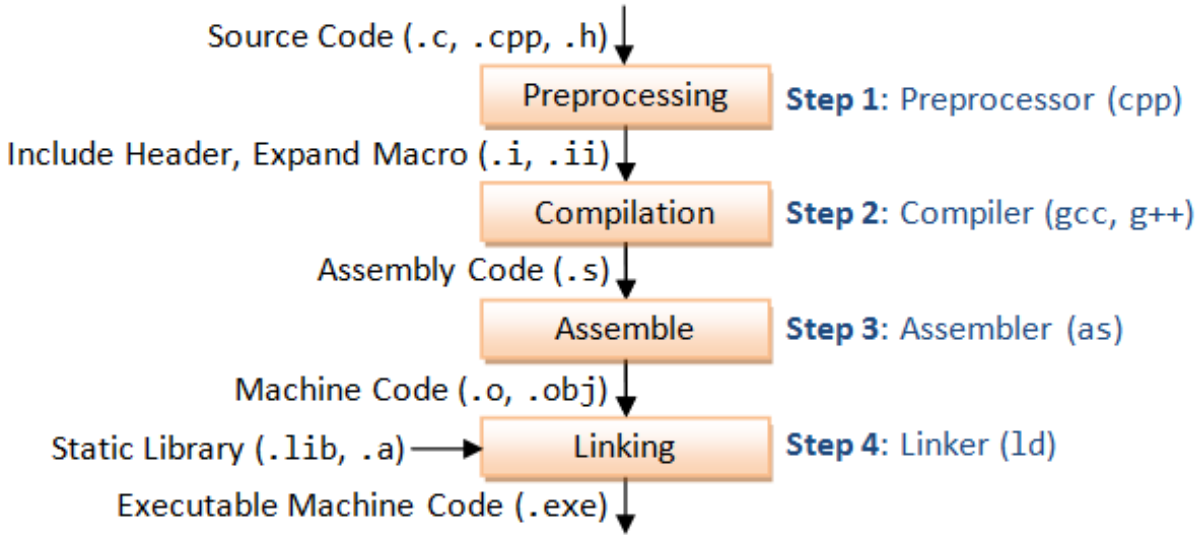
1. The header file: **IntArray.h**
2. The implementation file: **IntArray.cpp**
3. The application file, containing our main function: **main.cpp**

Compilation

The default build task that VS Code defines in **tasks.json** is only configured to compile **the active file** (singular), meaning the file that is currently being shown in the editor. You can in fact configure this build task to compile everything, but I'd like to show this explicitly as it is educational. I would also like you to become comfortable typing compiler options from the terminal.

Concept: main.cpp and IntArray.cpp need to be **compiled and linked** into a single executable. This is a two-step process executed by two different tools. Notice that the header file does not **need** to be compiled because it is used by the preprocessor step. (It's actually a 4-step process but we can discuss this later). There can be infinite source code files that comprise the single executable.

The C++ Build Process



Here is my directory structure

```
❯ ls -l
total 24
-rw-r--r--@ 1 Admin  staff  577 Sep  3 07:37 IntArray.cpp
-rw-r--r--@ 1 Admin  staff  243 Sep  3 07:37 IntArray.h
-rw-r--r--@ 1 Admin  staff  401 Sep  3 07:37 main.cpp
```

Here is my build syntax:

> g++ -Wall –pedantic -o executable IntArray.cpp main.cpp

This is telling the g++ compiler to preprocess, compile, assemble and link the files **IntArray.cpp and main.cpp** into an executable named **executable**. Remember, the **–Wall** option tells the compiler to show all warnings and the **–pedantic** option tells the compiler to strictly adhere to standards. The **-o** flag allows us to specify an executable name **other than a.out (or a.exe on Windows).** This name can be anything that is a valid identifier on your system.

You can determine the executable's format by using the **file** command. You can research your executable format to discover what types of meta data are included in the finished product. This will obviously be different on your system. Unfortunate for you Windows users, you don't get this incredibly useful command. Powershell may have something similar. Git Bash also.

```
❯ file executable
executable: Mach-O 64-bit executable x86_64
```

Now you can run the program by typing **./executable**

```
❯ ./executable
Constructor called
The value of element 5 is: 6
Destructor automatically called
```

Windows folks should see the **.exe** extension added to the end of the object file.