8.9 — Class code and header files

BY ALEX ON SEPTEMBER 11TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 13TH, 2018

Defining member functions outside the class definition

All of the classes that we have written so far have been simple enough that we have been able to implement the member functions directly inside the class definition itself. For example, here's our ubiquitous Date class:

```cpp
class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};
```

However, as classes get longer and more complicated, having all the member function definitions inside the class can make the class harder to manage and work with. Using an already-written class only requires understanding its public interface (the public member functions), not how the class works underneath the hood. The member function implementation details just get in the way.

Fortunately, C++ provides a way to separate the "declaration" portion of the class from the "implementation" portion. This is done by defining the class member functions outside of the class definition. To do so, simply define the member functions of the class as if they were normal functions, but prefix the class name to the function using the scope resolution operator (::) (same as for a namespace).

Here is our Date class with the Date constructor and setDate() function defined outside of the class definition. Note that the prototypes for these functions still exist inside the class definition, but the actual implementation has been moved outside:

```cpp
class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};

// Date constructor
Date::Date(int year, int month, int day)
```

```
{
    SetDate(year, month, day);
}

// Date member function
void Date::SetDate(int year, int month, int day)
{
    m_month = month;
    m_day = day;
    m_year = year;
}
```

This is pretty straightforward. Because access functions are often only one line, they are typically left in the class definition, even though they could be moved outside.

Here is another example that includes an externally defined constructor with a member initialization list:

```
class Calc
{
private:
    int m_value = 0;

public:
    Calc(int value=0): m_value(value) {}

    Calc& add(int value) { m_value  += value;  return *this; }
    Calc& sub(int value) { m_value -= value;  return *this; }
    Calc& mult(int value) { m_value *= value;  return *this; }

    int getValue() { return m_value ; }
};
```

becomes:

```
class Calc
{
private:
    int m_value = 0;

public:
    Calc(int value=0);

    Calc& add(int value);
    Calc& sub(int value);
    Calc& mult(int value);

    int getValue() { return m_value; }
};

Calc::Calc(int value): m_value(value)
{
}

Calc& Calc::add(int value)
{
    m_value += value;
    return *this;
}

Calc& Calc::sub(int value)
{
    m_value -= value;
    return *this;
}

Calc& Calc::mult(int value)
{
    m_value *= value;
    return *this;
}
```

Putting class definitions in a header file

In the lesson on **header files**, you learned that you can put function declarations inside header files in order to use those functions in multiple files or even multiple projects. Classes are no different. Class definitions can be put in header files in order to facilitate reuse in multiple files or multiple projects. Traditionally, the class definition is put in a header file of the same name as the class, and the member functions defined outside of the class are put in a .cpp file of the same name as the class.

Here's our Date class again, broken into a .cpp and .h file:

Date.h:

```cpp
#ifndef DATE_H
#define DATE_H

class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};

#endif
```

Date.cpp:

```cpp
#include "Date.h"

// Date constructor
Date::Date(int year, int month, int day)
{
    SetDate(year, month, day);
}

// Date member function
void Date::SetDate(int year, int month, int day)
{
    m_month = month;
    m_day = day;
    m_year = year;
}
```

Now any other header or code file that wants to use the Date class can simply `#include "Date.h"`. Note that Date.cpp also needs to be compiled into any project that uses Date.h so the linker knows how Date is implemented.

Doesn't defining a class in a header file violate the one-definition rule?

It shouldn't. If your header file has proper header guards, it shouldn't be possible to include the class definition more than once into the same file.

Types (which include classes), are exempt from the part of the one-definition rule that says you can only have one definition per program. Therefore, there isn't an issue #including class definitions into multiple code files (if there was, classes wouldn't be of much use).

Doesn't defining member functions in the header violate the one-definition rule?

It depends. Member functions defined inside the class definition are considered implicitly inline. Inline functions are exempt from the one definition per program part of the one-definition rule. This means there is no problem defining trivial member functions (such as access functions) inside the class definition itself.

Member functions defined outside the class definition are treated like normal functions, and are subject to the one definition per program part of the one-definition rule. Therefore, those functions should be defined in a code file, not inside the header. The one exception for this is for template functions, which we'll cover in a future chapter.

So what should I define in the header file vs the cpp file, and what inside the class definition vs outside?

You might be tempted to put all of your member function definitions into the header file, inside the class. While this will compile, there are a couple of downsides to doing so. First, as mentioned above, this clutters up your class definition. Second, functions defined inside the class are implicitly inline. For larger functions that are called from many places, this can bloat your code. Third, if you change anything about the code in the header, then you'll need to recompile every file that includes that header. This can have a ripple effect, where one minor change causes the entire program to need to recompile (which can be slow). If you change the code in a .cpp file, only that .cpp file needs to be recompiled!

Therefore, we recommend the following:

- For classes used in only one file that aren't generally reusable, define them directly in the single .cpp file they're used in.
- For classes used in multiple files, or intended for general reuse, define them in a .h file that has the same name as the class.
- Trivial member functions (trivial constructors or destructors, access functions, etc…) can be defined inside the class.
- Non-trivial member functions should be defined in a .cpp file that has the same name as the class.

In future lessons, most of our classes will be defined in the .cpp file, with all the functions implemented directly in the class definition. This is just for convenience and to keep the examples short. In real projects, it is much more common for classes to be put in their own code and header files, and you should get used to doing so.

Default parameters

Default parameters for member functions should be declared in the class definition (in the header file), where they can be seen by whomever #includes the header.

Libraries

Separating the class definition and class implementation is very common for libraries that you can use to extend your program. Throughout your programs, you've #included headers that belong to the standard library, such as iostream, string, vector, array, and other. Notice that you haven't needed to add iostream.cpp, string.cpp, vector.cpp, or array.cpp into your projects. Your program needs the declarations from the header files in order for the compiler to validate you're writing programs that are syntactically correct. However, the implementations for the classes that belong to the C++ standard library is contained in a precompiled file that is linked in at the link stage. You never see the code.

Outside of some open source software (where both .h and .cpp files are provided), most 3rd party libraries provide only header files, along with a precompiled library file. There are several reasons for this: 1) It's faster to link a precompiled library than to recompile it every time you need it, 2) a precompiled library can be distributed once, whereas compiled code gets compiled into every executable that uses it (inflating file sizes), and 3) intellectual property reasons (you don't want people stealing your code).

Having your own files separated into declaration (header) and implementation (code file) is not only good form, it also makes creating your own custom libraries easier. Creating your own libraries is beyond the scope of these tutorials, but separating your declaration and implementation is a prerequisite to doing so.

8.10 -- Const class objects and member functions

Index