15.2 — Classes and header files

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 a simple `Date` class where all member functions are defined inside the `Date` class definition:

```
#include <iostream>

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

public:
    Date(int year, int month, int day)
        : m_year { year }
        , m_month { month }
        , m_day { day}
    {
    }

    void print() const { std::cout << "Date(" << m_year << ", " << m_month << ", " <<
m_day << ")\n"; };

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

int main()
{
    Date d { 2015, 10, 14 };
    d.print();

    return 0;
}
```

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 implementations clutter up the public interface with details that aren't relevant to actually using the class.

To help address this, C++ allows us to separate the “declaration” portion of the class from the “implementation” portion by defining member functions outside of the class definition.

Here is the same `Date` class as above, with the constructor and `print()` member functions defined outside the class definition. Note that the prototypes for these member functions still exist inside the class definition (as these functions need to be declared as part of the class type definition), but the actual implementation has been moved outside:

```
#include <iostream>

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

public:
    Date(int year, int month, int day); // constructor declaration

    void print() const; // print function declaration

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

Date::Date(int year, int month, int day) // constructor definition
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

void Date::print() const // print function definition
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day << ")\n";
};

int main()
{
    const Date d{ 2015, 10, 14 };
    d.print();

    return 0;
}
```

Member functions can be defined outside the class definition just like non-member functions. The only difference is that we must prefix the member function names with the name of the class type (in this case, `Date::`) so the compiler knows we’re defining a member of that class type rather than a non-member.

Note that we left the access functions defined inside the class definition. Because access functions are typically only one line, defining these functions inside the class definition adds minimal clutter, whereas moving them outside the class definition would result in many extra lines of code. For this reason, the definitions of access functions (and other trivial, one-line functions) are often left inside the class definition.

Putting class definitions in a header file

If you define a class inside a source (.cpp) file, that class is only usable within that particular source file. In larger programs, it’s common that we’ll want to use the classes we write in multiple source files.

In lesson 2.11 -- Header files, you learned that you can put function declarations in a header files. Then you can #include those functions declarations into multiple code files (or even multiple projects). Classes are no different. A class definitions can be put in a header files, and then #included into any other files that want to use the class type.

Unlike functions, which only need a forward declaration to be used, the compiler typically needs to see the full definition of a class (or any program-defined type) in order for the type to be used. This is because the compiler needs to understand how members are declared in order to ensure they are used properly, and it needs to be able to calculate how large objects of that type are in order to instantiate them. So our header files usually contain the full definition of a class rather than just a forward declaration of the class.

Naming your class header and code files

Most often, classes are defined in header files of the same name as the class, and any 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:

```
#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 print() const;

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

#endif
```

Date.cpp:

```
#include "Date.h"

Date::Date(int year, int month, int day) // constructor definition
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

void Date::print() const // print function definition
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day << ")\n";
};
```

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 that the linker can connect calls to the member functions to their definitions.

Best practice

Prefer to put your class definitions in a header file with the same name as the class. Trivial member functions (such as access functions, constructors with empty bodies, etc…) can be defined inside the class definition.

Prefer to define non-trivial member functions in a source file with the same name as the class.

Doesn’t defining a class in a header file violate the one-definition rule if the header is #included more than once?

Types are exempt from the part of the one-definition rule (ODR) that says you can only have one definition per program. Therefore, there isn’t an issue #including class definitions into multiple translation units. If there was, classes wouldn’t be of much use.

Including a class definition more than once into a single translation unit is still an ODR violation. However, header guards (or `#pragma once`) will prevent this from happening.

Inline member functions

Member functions are not exempt from the ODR, so you may be wondering how we avoid ODR violations when member functions are defined in a header file (that may then be included into more than one translation unit).

Member functions defined *inside* the class definition are implicitly inline. Inline functions are exempt from the one definition per program part of the one-definition rule.

Member functions defined *outside* the class definition are not implicitly inline (and thus are subject to the one definition per program part of the one-definition rule). This is why such functions are usually defined in a code file (where they will only have one definition across the whole program).

Alternatively, member functions defined outside the class definition can be left in the header file if they are made inline (using the `inline` keyword). Here’s our *Date.h* header again, with the member functions defined outside the class marked as `inline`:

Date.h:

```
#ifndef DATE_H
#define DATE_H

#include <iostream>

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

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

    void print() const;

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

inline Date::Date(int year, int month, int day) // now inline
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

inline void Date::print() const // now inline
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day << ")\n";
};

#endif
```

This *Date.h* can be included into multiple translation units without issue.

Key insight

Functions defined inside the class definition are implicitly inline, which allows them to be #included into multiple code files without violating the ODR.

Functions defined outside the class definition are not implicitly inline. They can be made inline by using the `inline` keyword.

Inline expansion of member functions

The compiler must be able to see a full definition of a function in order to perform inline expansion. Most often, such functions (e.g. access functions) are defined inside the class definition. However, if you want to define a member function outside the class definition, but still want it to be eligible for inline expansion, you can define it as an inline function just below the class definition (in the same header file). That way the definition of the function is accessible to anybody who #includes the header.

So why not put everything in a header file?

You might be tempted to put all of your member function definitions into the header file, either inside the class definition, or as inline functions below the class definition. While this will compile, there are a couple of downsides to doing so.

First, as mentioned above, defining members inside the class definition clutters up your class definition.

Second, if you change any of 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. The cost of recompilation can vary significantly: a small project may only take a minute or less to build, whereas a large commercial project can take hours.

Conversely, if you change the code in a .cpp file, only that .cpp file needs to be recompiled. Therefore, given the choice, it's generally better to put non-trivial code in a .cpp file when you can.

There are a few cases where it might make sense to violate the best practice of putting the class definition in a header and non-trivial member functions in a code file.

First, for a small class that is used in only one code file and not intended for general reuse, you may prefer to define the class (and all member functions) directly in the single .cpp file it is used in. This helps make it clear that the class is only used within that single file, and is not intended for wider use. You can always move the class to a separate header/code file later if you later find you want to use it in more than one file, or are finding that the class and member function definitions are cluttering your source file.

Second, if a class only has a small number of non-trivial member functions that are unlikely to change, creating a .cpp file that contains only one or two definitions may not be worth the effort (as it clutters your project). In such cases, it may be preferable to make the member functions `inline` and place them beneath the class definition in the header.

Third, in modern C++, classes or libraries are increasingly being distributed as "header-only", meaning all of the code for the class or library is placed in a header file. This is done primarily to make distributing and using such files easier, as a header only needs to be

#included, whereas a code file needs to be explicitly added to every project that uses it, so that it can be compiled. If intentionally creating a header-only class or library for distribution, all non-trivial member functions can be made `inline` and placed in the header file beneath the class definition.

Finally, for template classes, template member functions defined outside the class are almost always defined inside the header file, beneath the class definition. Just like non-member template functions, the compiler needs to see the full template definition in order to instantiate it. We cover template member functions in lesson 15.5 -- Class templates with member functions.

Author's note

In future lessons, most of our classes will be defined in a single .cpp file, with all the functions implemented directly in the class definition. This is done to keep the examples concise and easy to compile yourself. 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 arguments for member functions

In lesson 11.5 -- Default arguments, we discussed the best practice for default arguments of non-member functions: "If the function has a forward declaration (especially one in a header file), put the default argument there. Otherwise, put the default argument in the function definition."

Because member functions are always declared (or defined) as part of the class definition, the best practice for member functions is actually simpler: always put the default argument inside the class definition.

Best practice

Put any default arguments for member functions inside the class definition.

Libraries

Throughout your programs, you've used classes that are part of the standard library, such as `std::string`. To use these classes, you simply need to #include the relevant header (such as `#include <string>`). Note that you haven't needed to add any code files (such as `string.cpp` or `iostream.cpp`) into your projects.

The header files provide the declarations that the compiler requires in order to validate that the programs you're writing are syntactically correct. However, the implementations for the classes that belong to the C++ standard library are contained in a precompiled file that is linked in automatically at the link stage. You never see the code.

Many open source software packages provide both .h and .cpp files for you to compile into your program. However, most commercial libraries provide only .h files and 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 single copy of a precompiled library can be shared by many applications, 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).

We discuss how to include 3rd party precompiled libraries into your projects in the appendix.

While you probably won't be creating and distributing your own libraries for a while, separating your classes into header files and source files 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 if you want to distribute precompiled binaries.

Quiz time

Question #1

h/t to reader "learnccp lesson reviewer" for these quiz questions.

What is the purpose of defining member functions outside the class definition?
a) To make the class definition shorter and easier to manage.
b) To separate the public interface from the implementation details.
c) When defined in a source file, to minimize recompilation times when an implementation detail changes.
d) All of the above.

Show Solution

How do you define a member function outside the class definition?
a) Simply define the function as a normal function without any class prefix.
b) Define the function with the class name prefixed using the scope resolution operator (::).
c) Declare the function inside the class definition and define it outside using the friend keyword.
d) None of the above.

Show Solution

When should trivial member functions be defined inside the class definition?
a) Always, to improve performance.
b) When the functions have a single line of code.
c) When the functions are called frequently.
d) It is not recommended to define any member functions inside the class definition.

Show Solution

Where should the class definition be placed to facilitate reuse in multiple files or projects?
a) In a .cpp file with the same name as the class.
b) In a separate header file with the same name as the class.
c) In a .cpp file that includes the header file.
d) Anywhere in the code, as long as the functions are defined outside the class.

Show Solution

Which of the following is true about the one-definition rule for classes and member functions?
a) It prohibits defining a class in a header file.
b) It allows including the class definition multiple times in the same file.
c) Member functions defined inside the class are exempt from the one-definition rule.
d) Non-trivial member functions should always be defined in the header file.

Show Solution