ALL CODE CRAFT

Top 10 C++ header file mistakes and how to fix them

May 22, 2016 Deb Haldar 6 Comments C++

C++ header files is a rather mundane topic by most standards. Talking about header files is not as interesting as discussing complex search algorithms or debating design patterns . It's not an academically stimulating subject to teach, so most CS programs do not emphasize header file design in their courses.

However, not having the correct header file design decisions can have significant ramifications on your project in terms of increases build times, compilation fiascos, code maintainability issues and

Search

Popular Posts

System Design Interview Concepts – Database Sharding

Top 20 C++ multithreading mistakes and how to avoid them

Top 10 dumb mistakes to avoid with C++ 11 smart pointers

Top 10 C++ header file mistakes and how to fix them

System Design Interview Concepts - Consistent Hashing

Top 15 C++ Exception handling mistakes and how to avoid them.

6 Tips to supercharge C++11 vector performance

System Design Interview Concepts – Eventual Consistency

Top 20 C pointer mistakes and how to fix them

plain information leakage. The larges your C++ project is, the more important this becomes.

Here's a short description of the top 10 header file issues that can crop up if you're not careful and how to avoid them.

Mistake # 1: Not using "include guards" in a header file.

When the preprocessor sees a #include, it replaces the #include with the contents of the specified header. Using a include guard , you can prevent a header file being included multiple times during the compilation process. The most common way to define an include guard is as follows:

```
//File: Aircraft.h
#ifndef AIRCRAFT_H
#define AIRCRAFT_H

\\the entire file

#endif
```

You usually name your #include guard the same as the name of your header file.

There are two main problems that #include guards help solve.

1. It can help prevent danger circular references between header files which can cause weird compilation failures.

Consider the following example where main.cpp includes both Airbus.h and Boeing.h:

```
//File: Airbus.h
#include "Boeing.h"
namespace Airbus
{
        class Carrier
        {
                Carrier();
                ~Carrier();
        };
}
```

C++ 11 Auto: How to use and avoid abuse

RECENT POSTS

System Design Interview Concepts – Database Sharding

System Design Interview Concepts – Eventual Consistency

System Design Interview Concepts – Load Balancing

System Design Interview Concepts – CAP Theorem

Top 20 C pointer mistakes and how to fix them

35 things I learnt at Game Developer Conference (GDC) 2018

System Design Interview Concepts – Consistent Hashing

Archives

August 2018 (2)

July 2018 (2)

May 2018 (1)

April 2018 (1)

October 2017 (1)

August 2017 (2)

January 2017 (1)

November 2016 (1)

October 2016 (2)

August 2016 (1)

June 2016 (2)

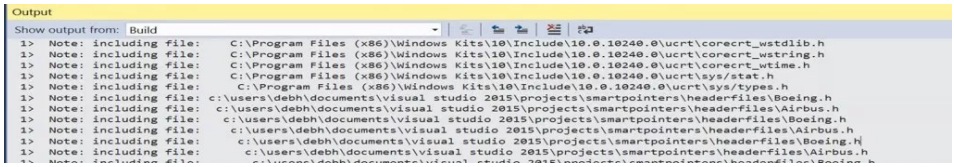
May 2016 (4)

```
//File: Boeing.h
#include "Airbus.h"
namespace Boeing
{
        class Carrier
        {
                Carrier();
                ~Carrier();
        };
}

// main.cpp : Defines the entry point for the
console application.
#include "stdafx.h"
#include "Boeing.h"
#include "Airbus.h"

int main()
{
    return 0;
}
```

Compiling the code above gives the following error:

1>c:\users\user\documents\visual studio 2015\projects\smartpointers\headerfiles\airbus.h(2): fatal error C1014: too many include files: depth = 1024

If you're in a large project with hundreds of include files, it might take some digging to find it out. If you're using VS2015, you're in luck because there's an option to show the include order : Right Click the Project -> Properties -> C/C++ -> Advanced -> Show Includes. If you turn this on, you'll see the following in the output window:



Looking at this you can easily tell that there's a circular reference between Boeing.h and Airbus.h. Fortunately, include guards can help fix the problem. The revised piece of code is below.

```
//File: Airbus.h
#ifndef AIRBUS_H
#define AIRBUS_H

#include "Boeing.h"
```

April 2016 (2)
March 2016 (1)
February 2016 (3)
January 2016 (3)

```
namespace Airbus
{
        class Carrier
        {
                Carrier();
                ~Carrier();
        };
}
#endif

//File: Boeing.h
#ifndef BOEING_H
#define BOEING_H

#include "Airbus.h"
namespace Boeing
{
        class Carrier
        {
                Carrier();
                ~Carrier();
        };
}
#endif

// main.cpp : Defines the entry point for the
console application.
//

#include "stdafx.h"
#include "Boeing.h"
#include "Airbus.h"

int main()
{
    return 0;

}
```

2. In the absence of an include guard, a file will need to be processed multiple times and can cause significant build delays in large systems.

Recommendation: Always use an include guard as shown above to optimize build times and avoid weird build errors. If your compiler supports and optimized #pragma once as an include guard mechanism, you should use that because it is usually more performant and less error prone than using an explicit include guard. For example, a lot of our internal code uses the following

convention for public header files. Notice that if we're on a MS compiler where _MSC_VER is defined, we'll use the #pragma directive which is supported and optimized by the compiler.

```
#ifndef HEADER_FILE
#define HEADER_FILE

#ifdef _MSC_VER
#pragma once
#endif  // _MSC_VER

// Contents of the header file here

#endif  // HEADER_FILE
```

MISTAKE # 2: Incorporating "using namespace" statements at top level in a header file

Headers should define only the names that are part of the interface, not names used in its own implementation. However, a using directive at top level in a header file injects names into every file that includes the header.

This can cause multiple issues:

1. It is not possible for a consumer of your header file to undo the namespace include – thus they are forced to live with your namespace using decision, which is undesirable.
2. It dramatically increases the chance of naming collissions that namespaces were meant to solve in the first place.
3. It is possible that a working version of the program will fail to compile when a new version of the library is introduced. This happens if the new version introduces a name that conflicts with a name that the application is using from another library.
4. The “using namespace” part of the code takes effect from the point where it appears in the code that included your header, meaning that any code appearing before that might get treated differently from any code appearing after that point.

Recommendations:

1. Try to avoid putting any using namespace declarations in your header files. If you absolutely need some namespace objects to

get your headers to compile, please use the fully qualified names (Eg. std::cout , std::string)in the header files.

```
//File:MyHeader.h:
class MyClass
{
private:
    Microsoft::WRL::ComPtr _parent;
    Microsoft::WRL::ComPtr _child;
}
```

2. If recommendation #1 above causes too much code clutter – restrict your “using namespace” usage to within the class or namespace defined in the header file. Another option is using scoped aliases in your header files as shown below.

```
//File:MyHeader.h:

class MyClass
{
namespace wrl = Microsoft::WRL; // note the
aliasing here !
private:
    wrl::ComPtr _parent;
    wrl::ComPtr _child;
}
```

MISTAKE # 3 : Having multiple unrelated functionality grouped together in a single header file (and cpp file)

I’ve seen multiple cases where a header file becomes a dumping ground for all miscellaneous functionality added at a late phase in the project. Recently. I came across a codebase that lumped a logging functionality and HTTP Get/Post API in a single header file. This fundamentally violates the concept of Single Responsibility Principle in a module. Even worse, when I first started reading the code, I thought it was some sort of logger specific to networking/http – but it turned out that it was just a general purpose file logger which happened to share some helper functions from the http library in the same

module !!! There is no way I can pull out either the HTTP or FileLogger for use in another project without significant rework.

Recommendation: Each header file, which basically provides an interface for your client software, should provide ***one clearly identifiable piece of functionality***. (The same goes for your cpp files).

MISTAKE # 4: Not making the header file compliable by itself

A header file should have everything it needs to compile by itself , i.e., it should explicitly #include or forward declare the types/ structs it needs to compile. If a header file does not have everything it needs to compile but the program incorporating the header file compiles, it indicates that somehow the header file is getting what it needs because of an include order dependency. This typically happens because another header file gets included in the compile chain before this incompilable header file which provides the missing functionality. If the include order/build order dependency changes, then the whole program might break in unexpected ways. The C++ compiler is notorious for misleading error messages and it might not be easy to locate the error at that point.

Recommendation: Check your header filies by compiling them in isolation via a testMain.cpp that includes nothing but the header file under test. If it produces a compilation error, then something either needs to get included in the header file or forward declared. The process should be repeated for all header files in the project using a bottoms-up approach. This'll help prevent random build break as the codebase grows larger and code blocks are moved around.

MISTAKE 5.a : Including non-required header files in your header – for example, including files that only the .cpp file code needs .

A common example of un-needed header files in your header file is <cmath> and <algorithm>.

Recommendation: Do not bloat your header files with unnecessary #includes.

Mistake # 5.b : Putting too much information in a header file and causing information leakage.

This is really important if you're creating and distributing DLLs. Each DLL is packaged with a header file that acts as a public interface of the functionality provided by the DLL. So If you're developing a Protocol Handler to send AMQP Network Traffic, you'd not want to expose what implementation engine you're using underneath the scenes.

Recommendation: Only expose functionality that the client of your library needs in a header file.

Mistake # 6: Not explicitly including all STL headers required by your cpp code file.

The standard does not specify which STL header files will be included by which other STL headers. So if you forget to include STL headers explicitly required by your code, it may work because the dependency is brought in via some other header file you included. However, any change / removal of dependencies can break the build in unexpected ways.

Recommendation: Always explicitly include the STL functionality used by your cpp files . For example, if you use <algorithms>, include that header explicitly in your cpp file.

Mistake # 7: Not making judicious use of forward declarations in header files

Forward declaration is an interesting technique often employed in C++ used to

- **Reduce compile times:** If your header needs a type declared in another header to compile, you have two options : either include the dependent header in your header file or forward declare the types in your header file. If the dependent

header file is very large and you only need to use say 5% of the types in the dependent header, it's much better to use forward declaration to make those types known in your header file than to bring in the full dependent header. If your header file is included by multiple projects in a very large solution , it can shave off hours from the build time.

- **Break cyclic dependency between code:** Imagine a situation where you have an Aircraft class and an Airport class. An Aircraft has reference to an Airport as it's home base and an Airport has a fleet of Aircrafts. In this situation , the Aircraft class needs to know the declaration of Airport exists and vice-versa. If you make both header file include each other, we'll end up in a never ending cyclic dependency. Consider the followiing piece of code:

```
#pragma once
//File: Aircraft.h
#include "Airport.h"

class Aircraft
{
        Airport* m_HomeBase;
};

#pragma once
//File: Airport.h
#include
#include "Aircraft.h"

class Airport
{
        std::vector m_Fleet;
};

// ForwardDeclaration.cpp : Defines the entry point
for the console application.

#include "stdafx.h"
#include "Airport.h"

int main()
{
    return 0;
}
```

The code above fails to compile with the following arcane errors :

1> Note: including file: c:\users\debh\documents\visual studio 2015\projects\smartpointers\forwarddeclaration\Aircraft.h

1>c:\users\debh\documents\visual studio 2015\projects\smartpointers\forwarddeclaration\aircraft.h(7): error C2143: syntax error: missing ';' before ''*
1>c:\users\debh\documents\visual studio 2015\projects\smartpointers\forwarddeclaration\aircraft.h(7): error C4430: missing type specifier – int assumed. Note: C++ does not support default-int
1>c:\users\debh\documents\visual studio 2015\projects\smartpointers\forwarddeclaration\aircraft.h(7): error C2238: unexpected token(s) preceding ';'

This is what happened:

1. Main included “Airport.h”
2. The first thing “Airport.h” included is “Aircraft.h”
3. While trying to include “Aircraft.h”, the compiler does not know a definition of “Airport” which is used in “Aircraft.h” header. At this point, it fails compilation.

The fix is easy: Just forward declare the class Airport in “Aircraft.h

```
#pragma once
//File: Aircraft.h
#include "Airport.h"

class Airport; //Forward Declare Airport!

class Aircraft
{
        Airport* m_HomeBase;
};
```

Recommendation: If you have cyclic dependencies between header file objects or just using < 10% of header file functionality, consider using forward declarations.

Mistake # 8: Including a cpp file in a header file.

This sometimes happens because people want to share a bunch of code between cpp files for maintainability reasons. This is a bad idea – it can confuse the programmer , some IDE navigational features and even some build engines. Also, if this is a public API, people expect to get a set of header files to use your DLL or LIB.

Getting a cpp file , they might think that something went wrong in the packaging/installation of the product.

Recommendation: Please put all shared code in an internal header file.

Mistake # 9: Declaring functions shared between multiple cpp files in separate header files/ code files.

When multiple files compile against a single function, the declaration for that function must be in a single header file. This allows maintainers to update the function declaration in a single place and detect any errors at compile time. This also makes it impossible to declare the function using the wrong parameter types, since there's an authoritative declaration.

Consider the following bad example of multiple declaration followed by a correct one:

BAD:

```
lib\Square.h
         int Square(int a);
lib\Square.cpp
         int Square(int a) { return a*a; }
myProgram\main.cpp
         int Square(int a);           // declare
Square() a second time - Bad !!!
         void DoStuff() { Square(33); }    // use
Square()
```

Correct Way:

```
lib\Square.h
         int Square(int a);
lib\Square.cpp
         int Square(int a) { return a*a; }
myProgram\main.cpp
         #include
         void DoStuff() { Square(33); }    // use
Square()
```

Recommendation: Shared functions between cpp files should be defined just once in a single header file.

Mistake # 10: Putting your project's header files into the precompiled header file.

Using the precompiled headers can significantly speed up your build time. One of the ways to screw it up is to include your own header files into the precompiled header file (pch.h or stdafx.h) . If you do, anytime those header files change, it'll trigger a re-build of your project. The ideal candidates for inclusion in precompiled header are large header files that you don't expect to change and is used by many of your cpp files– like windows.h, STL headers and header only implementations like rapid json.

Recommendation: Put only headers that'll not change in your precompiled headers.

Note: Read this excellent paper by Bruce Dawson for an indepth treatment of the subject matter.

So, What's Next ?

If you want to delve more into good physical design practices for C++, the following books are a good place to start:

- C++ Coding Standards: 101 Rules, Guidelines, and Best Practices by Herb Sutter and Andrei Alexandrescu
- Large-Scale C++ Software Design By John Lakos – it's a little dated but good read none the less

Did I miss any header file issues that should be called out? Please let me know via comments and I'll roll it back into the article.

Please ***share*** if you liked the article 🙂

← Top 10 dumb mistakes to avoid with C++ 11 smart pointers

[The Worst Programming Interview Question I've Ever Seen ! →]

Paul • 7 months ago

Reduce to an absolute minimum, #ifdef statements in the header file.
For example, the following is asking for trouble:

```
#ifdef MACRO
const u_int32 value_c = VALUE1;
#else
const u_int32 value_c = VALUE2;
#endif

class LinkCapabilityMgnt : public LinkCapabilityMgnt_ske
{
...
ClassDefinition classArray_m[value_c];
}
```

The problem is that the source files that #include the header file may or may not have the MACRO defined. This can be the case if several source files include this header file, but for example, are in different namespaces, libraries, etc with different macros defined.

Javene Mcgowan • 7 months ago

I have two points

1. Tell OpenGL that they must make their header files compliable. Mistake no 4.

2. I don't understand Mistake number 10

Oleksandr Malyushytskyy • 2 years ago

You are right that #pragma once is more optimized then include guards

on compilers where it is supported.
But using #pragma once and #include guards together is totally inefficient.
If you can guarantee that your code will always be compiled with compiler which understand #pragma once,
you use it, otherwise you use include guards. There is on only one reason to have code like below and it has nothing to do with efficiency. In fact it introduces inefficiency in order to still use #pragma once on the compilers which support it. And this is done cause often IDE tools (mostly IntellySense) can't rely on macro processing (since macro can be defined in other files) and use #pragma once which is limited to a single file.

As for efficiency on the compiler which supports pragma once your code does 3 checks instead of one (2 by #ifdef, 1 by pragma)
on other compilers it does one unneeded check - #ifdef _MSC_VER

```
#ifndef HEADER_FILE

#define HEADER_FILE

#ifdef _MSC_VER

#pragma once

#endif // _MSC_VER

// Contents of the header file here

#endif // HEADER_FILE
```

▲ ▼ • Reply • Share ›

Deb Haldar Mod ➚ Oleksandr Malyushytskyy • 2 years ago

You're right Oleksandr - putting in a check like that when the compiler supports #pragma once is a waste.

> "If you can guarantee that your code will always be compiled with compiler which understand #pragma once,you use it, otherwise you use include guards."
> The guidance is for public header files/ APIs that are distributed to external parties - we don't know if their compiler supports #pragma once or not - so the include guard is necessary.
> If you know your partner's compiler supports #pragma once, putting an header guard is not advisable like you mentioned.

▲ ▼ • Reply • Share ›

Oleksandr Malyushytskyy ➚ Deb Haldar • 2 years ago

My point was single include guards are better than attempt to use combination..

▲ ▼ • Reply • Share ›

Deb Haldar Mod ➚ Oleksandr Malyushytskyy • 2 years ago

I agree :)

▲ ▼ • Reply • Share ›

ALSO ON **A CODERS JOURNEY**

Transient Errors Are Evil – How

Event based synchronization of