Chapter 50: Create and include header files

In modern C, header files are crucial tools that must be designed and used correctly. They allow the compiler to cross-check independently compiled parts of a program.

Headers declare types, functions, macros etc that are needed by the consumers of a set of facilities. All the code that uses any of those facilities includes the header. All the code that defines those facilities includes the header. This allows the compiler to check that the uses and definitions match.

Section 50.1: Introduction

There are a number of guidelines to follow when creating and using header files in a C project:

- Idemopotence

  If a header file is included multiple times in a translation unit (TU), it should not break builds.

- Self-containment

  If you need the facilities declared in a header file, you should not have to include any other headers explicitly.

- Minimality

  You should not be able to remove any information from a header without causing builds to fail.

- Include What You Use (IWYU)

  Of more concern to C++ than C, but nevertheless important in C too. If the code in a TU (call it `code.c`) directly uses the features declared by a header (call it `"headerA.h"`), then `code.c` should `#include "headerA.h"` directly, even if the TU includes another header (call it `"headerB.h"`) that happens, at the moment, to include `"headerA.h"`.

Occasionally, there might be good enough reasons to break one or more of these guidelines, but you should both be aware that you are breaking the rule and be aware of the consequences of doing so before you break it.

Section 50.2: Self-containment

Modern headers should be self-contained, which means that a program that needs to use the facilities defined by `header.h` can include that header (`#include "header.h"`) and not worry about whether other headers need to be included first.

Recommendation: Header files should be self-contained.

Historical rules

Historically, this has been a mildly contentious subject.

Once upon another millennium, the AT&T Indian Hill C Style and Coding Standards stated:

> Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be `#include`d for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common `#include`s in one include file.

This is the antithesis of self-containment.

Modern rules

However, since then, opinion has tended in the opposite direction. If a source file needs to use the facilities declared by a header `header.h`, the programmer should be able to write:

```
#include "header.h"
```

and (subject only to having the correct search paths set on the command line), any necessary pre-requisite headers will be included by `header.h` without needing any further headers added to the source file.

This provides better modularity for the source code. It also protects the source from the "guess why this header was added" conundrum that arises after the code has been modified and hacked for a decade or two.

The NASA Goddard Space Flight Center (GSFC) coding standards for C is one of the more modern standards — but is now a little hard to track down. It states that headers should be self-contained. It also provides a simple way to ensure that headers are self-contained: the implementation file for the header should include the header as the first header. If it is not self-contained, that code will not compile.

The rationale given by GSFC includes:

> §2.1.1 Header include rationale

> This standard requires a unit's header to contain `#include` statements for all other headers required by the unit header. Placing `#include` for the unit header first in the unit body allows the compiler to verify that the header contains all required `#include` statements.

> An alternate design, not permitted by this standard, allows no `#include` statements in headers; all `#includes` are done in the body files. Unit header files then must contain #ifdef statements that check that the required headers are included in the proper order.

> One advantage of the alternate design is that the `#include` list in the body file is exactly the dependency list needed in a makefile, and this list is checked by the compiler. With the standard design, a tool must be used to generate the dependency list. However, all of the branch recommended development environments provide such a tool.

> A major disadvantage of the alternate design is that if a unit's required header list changes, each file that uses that unit must be edited to update the `#include` statement list. Also, the required header list for a

> compiler library unit may be different on different targets.

> Another disadvantage of the alternate design is that compiler library header files, and other third party files, must be modified to add the required `#ifdef` statements.

Thus, self-containment means that:

- If a header `header.h` needs a new nested header `extra.h`, you do not have to check every source file that uses `header.h` to see whether you need to add `extra.h`.
- If a header `header.h` no longer needs to include a specific header `notneeded.h`, you do not have to check every source file that uses `header.h` to see whether you can safely remove `notneeded.h` (but see Include what you use.
- You do not have to establish the correct sequence for including the pre-requisite headers (which requires a topological sort to do the job properly).

Checking self-containment

See Linking against a static library for a script `chkhdr` that can be used to test idempotence and self-containment of a header file.

Section 50.3: Minimality

Headers are a crucial consistency checking mechanism, but they should be as small as possible. In particular, that means that a header should not include other headers just because the implementation file will need the other headers. A header should contain only those headers necessary for a consumer of the services described.

For example, a project header should not include **`<stdio.h>`** unless one of the function interfaces uses the type `FILE *` (or one of the other types defined solely in **`<stdio.h>`**). If an interface uses `size_t`, the smallest header that suffices is **`<stddef.h>`**. Obviously, if another header that defines `size_t` is included, there is no need to include **`<stddef.h>`** too.

If the headers are minimal, then it keeps the compilation time to a minimum too.

It is possible to devise headers whose sole purpose is to include a lot of other headers. These seldom turn out to be a good idea in the long run because few source files will actually need all the facilities described by all the headers. For example, a **`<standard-c.h>`** could be devised that includes all the standard C headers — with care since some headers are not always present. However, very few programs actually use the facilities of **`<locale.h>`** or **`<tgmath.h>`**.

- See also How to link multiple implementation files in C?

Section 50.4: Notation and Miscellany

The C standard says that there is very little difference between the `#include <header.h>` and `#include "header.h"` notations.

> [`#include <header.h>`] searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

> [`#include "header.h"`] causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"..."` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read [`#include <header.h>`] …

So, the double quoted form may look in more places than the angle-bracketed form. The standard specifies by example that the standard headers should be included in angle-brackets, even though the compilation works if you use double quotes instead. Similarly, standards such as POSIX use the angle-bracketed format — and you should too. Reserve double-quoted headers for headers defined by the project. For externally-defined headers (including headers from other projects your project relies on), the angle-bracket notation is most appropriate.

Note that there should be a space between `#include` and the header, even though the compilers will accept no space there. Spaces are cheap.

A number of projects use a notation such as:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

You should consider whether to use that namespace control in your project (it is quite probably a good idea). You should steer clear of the names used by existing projects (in particular, both `sys` and `linux` would be bad choices).

If you use this, your code should be careful and consistent in the use of the notation.

Do not use `#include "../include/header.h"` notation.

Header files should seldom if ever define variables. Although you will keep global variables to a minimum, if you need a global variable, you will declare it in a header, and define it in one suitable source file, and that source file will include the header to cross-check the declaration and definition, and all source files that use the variable will use the header to declare it.

Corollary: you will not declare global variables in a source file — a source file will only contain definitions.

Header files should seldom declare `static` functions, with the notable exception of `static inline` functions which will be defined in headers if the function is needed in more than one source file.

- Source files define global variables, and global functions.
- Source files do not declare the existence of global variables or functions; they include the header that declares the variable or function.
- Header files declare global variable and functions (and types and other supporting material).
- Header files do not define variables or any functions except (`static`) **`inline`** functions.

Cross-references

- Where to document functions in C?
- List of standard header files in C and C++
- Is **`inline`** without `static` or **`extern`** ever useful in C99?
- How do I use **`extern`** to share variables between source files?
- What are the benefits of a relative path such as `"../include/header.h"` for a header?
- Header inclusion optimization
- Should I include every header?

Section 50.5: Idempotence

If a particular header file is included more than once in a translation unit (TU), there should not be any compilation problems. This is termed 'idempotence'; your headers should be idempotent. Think how difficult life would be if you had to ensure that `#include <stdio.h>` was only included once.

There are two ways to achieve idempotence: header guards and the `#pragma once` directive.

Header guards

Header guards are simple and reliable and conform to the C standard. The first non-comment lines in a header file should be of the form:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

The last non-comment line should be `#endif`, optionally with a comment after it:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

All the operational code, including other `#include` directives, should be between these lines.

Each name must be unique. Often, a name scheme such as `HEADER_H_INCLUDED` is used. Some older code uses a symbol defined as the header guard (e.g. `#ifndef BUFSIZ` in **`<stdio.h>`**), but it is not as reliable as a unique name.

One option would be to use a generated MD5 (or other) hash for the header guard name. You should avoid emulating the schemes used by system headers which frequently use names reserved to the implementation — names starting with an underscore followed by either another underscore or an upper-case letter.

The `#pragma once` Directive

Alternatively, some compilers support the `#pragma once` directive which has the same effect as the three lines shown for header guards.

```
#pragma once
```

The compilers which support `#pragma once` include MS Visual Studio and GCC and Clang. However, if portability is a concern, it is better to use header guards, or use both. Modern compilers (those supporting C89 or later) are required to ignore, without comment, pragmas that they do not recognize ('Any such pragma that is not recognized by the implementation is ignored') but old versions of GCC were not so indulgent.

Section 50.6: Include What You Use (IWYU)

Google's Include What You Use project, or IWYU, ensures source files include all headers used in the code.

Suppose a source file `source.c` includes a header `arbitrary.h` which in turn coincidentally includes `freeloader.h`, but the source file also explicitly and independently uses the facilities from `freeloader.h`. All is well to start with. Then one day `arbitrary.h` is changed so its clients no longer need the facilities of `freeloader.h`. Suddenly, `source.c` stops compiling — because it didn't meet the IWYU criteria. Because the code in `source.c` explicitly used the facilities of `freeloader.h`, it should have included what it uses — there should have been an explicit `#include "freeloader.h"` in the source too. (Idempotency would have ensured there wasn't a problem.)

The IWYU philosophy maximizes the probability that code continues to compile even with reasonable changes made to interfaces. Clearly, if your code calls a function that is subsequently removed from the published interface,

no amount of preparation can prevent changes becoming necessary. This is why changes to APIs are avoided when possible, and why there are deprecation cycles over multiple releases, etc.

This is a particular problem in C++ because standard headers are allowed to include each other. Source file `file.cpp` could include one header `header1.h` that on one platform includes another header `header2.h`. `file.cpp` might turn out to use the facilities of `header2.h` as well. This wouldn't be a problem initially - the code would compile because `header1.h` includes `header2.h`. On another platform, or an upgrade of the current platform, `header1.h` could be revised so it no longer includes `header2.h`, and then`file.cpp` would stop compiling as a result.

IWYU would spot the problem and recommend that `header2.h` be included directly in `file.cpp`. This would ensure it continues to compile. Analogous considerations apply to C code too.