C++ Best Practice – Designing Header Files

by Alan Griffiths

C++, and its close relatives C and C++/CLI, have a compilation model that was invented way back in the middle of the last century – a directive to insert text into the translation unit from (at least in the typical case) a header file. This has the advantage of simplicity – and to run on 1970's computers compilers had to be simpler than they do today. However, there are other considerations and it is not clear that the same trade-off would be made if inventing a language today (certainly more recently designed languages like Java and C# import a compiled version of referenced code).

A consequence of this language design decision is that the responsibility for dealing with these "other considerations" shifts from the language and compiler writer to the developers using the language. Naturally, experienced C++ developers do not think about all these as they work, any more than an experienced driver thinks about where she places her hands and feet when turning a corner. But just as many drivers get into bad habits (and cut corners), many programmers fail to observe best practice at all times, so it is worth reviewing these factors occasionally.

Considerations for Designing C++ Header Files

1. The effect of including a header file should be deterministic (and not be dependent upon the context in which it is included).
2. Including a header file should be idempotent (including it several times should have the same effect as including it once).
3. A header file should have a coherent purpose (and not have unnecessary or surprising effects).
4. A header file should have low coupling (and not introduce excessive dependencies on other headers).

It should come as no surprise that this list is analogous to the design criteria for other elements of software – functions, classes, components, subsystems, etc. It includes well known techniques for making design elements easier to use, reuse and reason about. There are exceptions: for example, functions that may produce a different result on each invocation – **`std::strtok`** is one – but not all design is good design. (Not even in the standard library.)

If all header files met these criteria, then it would make the life of developers easier. However, because the language doesn't forbid ignoring these principles there are header files that don't satisfy them. One needs to look no further than the standard library and **`<cassert>`** to find an example of a header file that breaks the first two criteria. Not only does the effect of this header depend upon whether the preprocessor macro NDEBUG has been defined[1], but it is also designed to be included multiple times in a translation unit:

```
#undef NDEBUG
#include <cassert>
void assert_tested_here() { assert(true); }

#define NDEBUG
#include <cassert>
void assert_untested_here() { assert(false); }
```

1 It is also worth noting that one should be very wary of code using **`assert`** in header files as it may expand differently in different translation units. (Which would break the "One Definition Rule".)

Other examples exist too: Boost.preprocessor [1] makes use of a technique referred to as "File Iteration" (repeatedly `#including` a file under the control of preprocessor macros to generate families of macro expansions). Although this technique is undeniably useful, the circumstances where it is appropriate are rare – and the vast majority of headers do meet (or should meet) the conditions given above.

Writing Deterministic, Idempotent Headers

Most header files however are designed to be independent of when or how many times they are included. Lacking direct language support, developers have come up with a range of idioms and conventions[2] to achieve this. These methods relate only to the structure of the header which means that they are simple, effective and easy to follow:

- Only include header files at file scope. Including header files within a function or namespace scope is possible but it isn't idiomatic (and won't work unless the header is designed to make this possible).

```
// Do this...
#include "someheader.hpp"

// Not this
namespace bad_idea
{
    #include "someheader.hpp"
}
```

- The *Include Guard Idiom* is to surround the body of the include file by an **`#ifndef`** block and define the corresponding macro in the body. The body of the header, therefore, has no effect after the the first inclusion (but this does not avoid the file being reopened and parsed by the compiler's preprocessor[3]).

```
// Like this
#ifndef INCLUDED_EXAMPLE_HPP_ARG_20060303
#define INCLUDED_EXAMPLE_HPP_ARG_20060303
...
#endif
```

  Note that one needs some mechanism for ensuring the header guards are unique. (Some development environments will generate guaranteed unique guards for you – here I've just used a combination of the header name, my initials and the date.)
- Create *Self Contained Headers* that do not rely on the presence of macros, declarations or definitions being available at the point of inclusion. Another way to look at this is that they should be compilable on their own. One common way to ensure headers

2 This is not as effective as support within the language: not only does it require developers to know the idioms but also compilers can't be optimised on this asumption as they must support headers for which it is not true.

3 There is a technique for avoiding this compile-time cost – *External Include Guards*: each **`#include`** can be surrounded by a **`#ifndef`**...**`#define`**...**`#endif`** block akin to the include guard idiom mentioned below. It does work (I've reduced a 12 hour build cycle to 4 hours this way) but it is painful to use because it is verbose, and it is necessary to ensure that the guard macros are chosen consistently. Use it only in extreme need and, before using it, check your compiler for optimisations that would make it unnecessary.

are self-contained is to make them the first include in the corresponding implementation file.

- *Don't use using directives – especially in the global namespace (and particularly* `using namespace std;`*).* Users of the header file may be surprised and inconvenienced if names are introduced into scopes they think they control – this is particularly problematic with using directives at file scope as the set of names introduced depends upon which other headers are included, and name lookup in client code always has the potential to search the global namespace.

Compiler suppliers also like to add value and, on the basis that almost all header files are deterministic and idempotent, have used this assumption in attempts to reduce compilation time:

- There are compilers (like gcc) that recognise the Include Guard Idiom and don't process such header files (again) if the corresponding macro is defined. As the cost of opening files and parsing them is often a significant component of the compile time the saving can be significant.
- There are compilers (like Microsoft's Visual C++) that implement a language extension – `#pragma once` – that instructs the compiler to only process a header file the first time it is encountered. As the cost of opening files and parsing them is often a significant component of the compile time the saving can be significant.
- Some compilers (like Microsoft's Visual C++) allow headers to be "pre-compiled" into an intermediate form that allows the compiler to quickly reload the state it reaches after processing a series of headers. This may be effective if there is a common series of headers at the start of multiple translation units – opinions differ about the frequency of this occurring (if it is infrequent there may be a net cost of pre-compiled headers).
- There have also been "C++" compilers (such as IBM's VisualAge C++) that have abandoned the inclusion model of compilation – but that makes them significantly non-compliant and, to the best of my knowledge, this approach has been abandoned.

With the appropriate co-operation from the developer any, or all, of these can be effective at reducing build times. However, only the first of these is a portable approach in that it doesn't have the potential to change the meaning of the code. (If the code in a header file does have a different effect if included a second time then `#pragma once` changes its meaning; if files sharing a "precompiled header" actually have different sequences of includes the meaning is changed; and, with no separate translation units, VisualAge C++ interpreted `static` definitions and the anonymous namespace in non-standard ways.)

Writing Coherent Headers

It is much harder to meet this objective by rote mechanical techniques than the two considered above. This is because a single design element may map to several C++ language elements (for example, the `<map>` header provides not only the `std::map<>` class template, but also a number of template functions such as the corresponding `operator==` and a class template for `std::map<>::iterator`. It takes all of these elements to support the single "ordered associative container" concept embodied in this header. (Of course, this header also includes a parallel family of templates implementing "multimap" – it is less clear that this is part of the same concept.)

There have been various attempts to give simple rules for the design of coherent headers:

- "*Put each class in its own header file*." I frequently encounter this rule in "coding standards" but while classes and headers are both units of design they are not at the same level of abstraction. The example given above of a container and its iterators shows that this rule is simplistic. However, the weaker (but harder to explain) rule "if you have two classes in the same header then question whether they support the same abstraction" is probably useful.
- "*Can the purpose of the header be summarised in a simple phrase without using 'and' or 'or'?*" This is an application of a common test for coherence in a design. The problem with this test is that one needs to "get it" before it is useful: is "represent the household pets" a single concept? Or is "represents the household cats, dogs and budgerigars" multiple concepts? Given an appropriate context either might be true.

Of course, there are common mistakes that can be avoided:

- *Never create a header called* `utils.hpp` *(or variations like* `utility.h`*).* It is too easy for a later developer to add things to this header instead of crafting a more appropriate one for their purpose. (On one project I worked on `utility.h` got so cluttered that one of my colleagues created a `bits-and-bobs.h` header "to make it easier to find things". It didn't work.)

Writing Decoupled Headers

Occasionally one encounters adherents of a "pretend C++ is Java" style of header writing – just put the implementation code inline in the class definition. They will cite advantages of this techniques: you only code interfaces once (no copying of function signatures to the implementation file), there are fewer files to keep track of, and it is immediately apparent how a function works. (There is a further advantage that I've never seen mentioned – it is impossible to write headers with circular dependencies in this style[4].) Listing 1 shows a header written in this style.

In the listing I've highlighted the pieces of code that are of no value to translation units that use the header file. This is of little account when there are few users of the `telephone_list` class (for example, when the header is first being written and there is only the test harness to consider[5]). However, having implementation code in a header distracts from defining the contract with the client code.

Further, most C++ projects are considerably larger than this and, if this approach were applied to the entire codebase, the cost of compiling everything in one big translation unit would be prohibitive. Even on a smaller scale exposing implementation code can also have an adverse effect on the responsiveness of the build process: compiling the implementation of a function or member-function in every translation unit that includes the header is wasteful; in addition, implementation code often requires more context than interfaces (e.g. the definitions of classes being used rather than declarations of them); and, finally, the implementation

4 This is not true in Java, but C++ is not Java.

5 I suspect that I'm not the only one to apply "test driven development" to C++ by initially writing some pieces of new code in a header. If you decide to try this approach remember: refactoring the code to separate out the implementation afterwards is part of the work you need to do. This is not a case where "you ain't gonna need it" applies

Listing 1

```
// allinline.hpp - implementation hiding
// example.
#ifndef INCLUDED_ALLINLINE_HPP_ARG_20060303
#define INCLUDED_ALLINLINE_HPP_ARG_20060303
#include <string>
#include <utility>
#include <map>
#include <algorithm>
#include <ctype.h>

namespace allinline
{
  /** Example of implementing a telephone list
    * using an inline implementation.
  */
  class telephone_list
  {
  public:
    /** Create a telephone list.
    * @param    name    The name of the list.
    */
    telephone_list(const std::string& name)
     : name(name), dictionary() {}
    /** Get the list's name.
    * @return   the list's name.
    */
    std::string get_name()
     const { return name; }
    /** Get a person's phone number.
    * @param    person  The person's name
    * (must be an exact match)
    * @return   pair of success flag and (if
    * success) number.
    */
    std::pair<bool, std::string> get_number(
     const std::string& person) const {
     dictionary_t::const_iterator
     i = dictionary.find(person);
     return(i != dictionary.end()) ?
        std::make_pair(true, (*i).second) :
        std::make_pair(false, std::string());
    }

    /** Add an entry. If an entry already
    * exists for this person it is
    * overwritten.
    *   @param  name    The person's name
    *   @param  number  The person's number
    */
    telephone_list& add_entry(
     const std::string& name,
     const std::string& number) {
      std::string nn(name);
      std::transform(nn.begin(), nn.end(),
       nn.begin(), &tolower);
      dictionary[nn] = number;
      return *this;
    }
```

```
  private:
    typedef std::map<std::string,
            std::string> dictionary_t;
            std::string  name;
            dictionary_t dictionary;

    telephone_list(const telephone_list& rhs);
    telephone_list& operator=(
     const telephone_list& rhs);
  };
}
#endif
```

is more likely to change than the interface and trigger a mass recompilation of the client code.

Hence:

- *Don't put implementation code in headers*. If we apply this rule to our example we arrive at something like Listing 2[6]. Once again, I've highlighted stuff that is of no interest to the client code.
- *Don't include unnecessary headers*. It may seem obvious, but the easiest thing that developers can do to reduce the number of header files being included is not to include them when they are not needed. It takes only one unnecessary header in each file to have an enormous effect on compilation times: not only do the original files each add an extra file, but so do each of the added files, and the files they add, and... – the only reason that this doesn't go on forever is that eventually some of these includes are duplicated and, because of their include guards, these duplicates do not include anything the second or subsequent times they are included.

When is a Header File Include "Necessary"?

An examination of Listing 2 shows that I've highlighted three includes – two of these can be removed without preventing the header compiling while the third is not used in the interface to client code – it is an "implementation detail" that has leaked. There are also two headers (not highlighted) that are needed to compile the header itself and also include definitions used in the interface to client code. The two sidebars ("Cheshire Cat" and "Interface Class") show two alternative design techniques that permit removing these includes from the header file. Of course, they are still required by the implementation file and need to be moved there.

Deciding whether it is necessary to include a header file isn't quite as simple as removing it and asking "does it compile?" – a trial and error approach to eliminating header files can generate both false positives and false negatives: positives where a header appears unnecessary incorrectly (because it is also being included indirectly by another header – where it may be unnecessary), and negatives where a header is incorrectly being considered necessary (because it indirectly includes the header that is actually needed). There really is no shortcut that avoids understanding which definitions and declarations a header introduces and which of these are being used – and this is easier when headers have a coherent function.

6 This example may look familiar to some of you – in Overload 66 Mark Radford and I used a similar example in our article "Separating Interface and Implementation in C++". I've stolen it (and the "Cheshire Cat" and "Interface Class" sidebars) because it addresses the current theme.

Cheshire Cat

- A private "representation" class is written that embodies the same functionality and interface as the naïve class – however, unlike the naïve version, this is defined and implemented entirely within the implementation file. The public interface of the class published in the header is unchanged, but the private implementation details are reduced to a single member variable that points to an instance of the "representation" class – each of its member functions forwards to the corresponding function of the "representation" class.
- The term "Cheshire Cat" is an old one, coined by John Carollan over a decade ago [2]. Sadly it seems to have disappeared from use in contemporary C++ literature. It appears described as a special case of the "Bridge" pattern in *Design Patterns* [3], but the name "Cheshire Cat" is not mentioned. Herb Sutter [4] discusses it under the name "Pimpl idiom", but considers it only from the perspective of its use in reducing physical dependencies. It has also been called "Compilation Firewall".
- Cheshire Cat requires "boilerplate" code in the form of forwarding functions that are tedious to write and (if the compiler fails to optimise them away) can introduce a slight performance hit. It also requires care with the copy semantics (although it is possible to factor this out into a smart pointer [5]). As the relationship between the public and implementation classes is not explicit it can cause maintenance issues.

```
// cheshire_cat.hpp Cheshire Cat -
// implementation hiding example
#ifndef INCLUDED_CHESHIRE_CAT_HPP_ARG_20060303
#define INCLUDED_CHESHIRE_CAT_HPP_ARG_20060303
#include <string>
#include <utility>
namespace cheshire_cat
{
  class telephone_list
  {

  public:
    telephone_list(const std::string& name);

    ~telephone_list();
    std::string get_name() const;
    std::pair<bool, std::string>
    get_number(const std::string& person)
               const;
    telephone_list&
    add_entry(const std::string& name,
              const std::string& number);

  private:
    class telephone_list_implementation;
    telephone_list_implementation* rep;
    telephone_list(const telephone_list& rhs);
    telephone_list& operator=
                  (const telephone_list& rhs);
  };
}
#endif
```

Listing 2

```
// naive.hpp - implementation hiding
// example.
#ifndef INCLUDED_NAIVE_HPP_ARG_20060303
#define INCLUDED_NAIVE_HPP_ARG_20060303
#include <string>
#include <utility>
#include <map>
#include <algorithm>
#include <ctype.h>

namespace naive
{
  /** Telephone list. Example of implementing a
  *telephone list using a naive implementation.
  */

  class telephone_list
  {
  public:
    /** Create a telephone list.
    * @param    name    The name of the list.
    */
    telephone_list(const std::string& name);
    /** Get the list's name.
    * @return   the list's name.
    */
    std::string get_name() const;
    /** Get a person's phone number.
     * @param    person  The person's name
     * (must be an exact match)
     * @return   pair of success flag and (if
     * success) number.
     */
    std::pair<bool, std::string>
    get_number(const std::string& person)
     const;
    /** Add an entry. If an entry already
    * exists for this person it is overwritten.
    *
    *   @param  name    The person's name
    *   @param  number  The person's number
    */
    telephone_list&
    add_entry(const std::string& name,
              const std::string& number);

  private:
    typedef std::map<std::string,
                     std::string> dictionary_t;
    std::string  name;
    dictionary_t dictionary;
    telephone_list(const telephone_list& rhs);
    telephone_list& operator=
     (const telephone_list& rhs);
  };
}
#endif
```

Listing 3

```
#ifndef INCLUDED_LISTING3_HPP_ARG_20060303
#define INCLUDED_LISTING3_HPP_ARG_20060303

#include "location.hpp"
#include <ostream>

namespace listing3
{
  class traveller
  {
  public:
    ...
    location current_location() const;
    void list_itinary(std::ostream& out) const;
  };
}
#endif
```

There are also other ways to avoid including headers. Frequently, a header file will bring in a class (or class template) definition when all that is needed is a declaration. Consider Listing 3 where the header file `location.hpp` is included whereas all that is needed is the statement "class location;". A similar approach can be taken with **`std::ostream`** – but with a **`typedef`** in the standard library one cannot do it oneself: one cannot say **`typedef basic_ostream <char, char_traits<char> > ostream;`** without first declaring **`template<...> class basic_ostream;`** and **`template<...> class char_traits;`** and doing this is problematic because the bit I've shown as "..." is implementation defined.

The designers of the standard library did realise that people would want to declare the input and output stream classes without including the definitions, and allowed for this by providing a header

[continued at bottom of next page]

Listing 4

```
#ifndef INCLUDED_LISTING4_HPP_ARG_20060303
#define INCLUDED_LISTING4_HPP_ARG_20060303

#include <iosfwd>

namespace listing4
{
  class location;

  class traveller
  {
  public:
    ...
    location current_location() const;
    void list_itinary(std::ostream& out) const;
  };
}
#endif
```

Interface Class

All member data is removed from the naïve class and all member functions are made pure virtual. In the implementation file a derived class is defined and implements these member functions. The derived class is not used directly by client code, which sees only a pointer to the public class.

This is described in some detail in Mark Radford's "C++ Interface Classes – An Introduction" [6].

Conceptually the Interface Class idiom is the simplest of those we consider. However, it may be necessary to provide an additional component and interface in order to create instances. Interface Classes, being abstract, can not be instantiated by the client. If a derived "implementation" class implements the pure virtual member functions of the Interface Class, then the client can create instances of that class. (But making the implementation class publicly visible re-introduces noise.) Alternatively, if the implementation class is provided with the Interface Class and (presumably) buried in an implementation file, then provision of an additional instantiation mechanism – e.g. a factory function – is necessary. This is shown as a static **`create`** function in the corresponding sidebar.

As objects are dynamically allocated and accessed via pointers this solution requires the client code to manage the object lifetime. This is not a handicap where the domain understanding implies objects are to be managed by a smart pointer (or handle) but it may be significant in some cases.

```
// interface_class.hpp - implementation
// hiding example.
#ifndef INC_INTERFACE_CLASS_HPP_ARG_20060303
#define INC_INTERFACE_CLASS_HPP_ARG_20060303
#include <string>
#include <utility>

namespace interface_class
{
  class telephone_list
  {
  public:
    static telephone_list* create(
     const std::string& name);
    virtual ~telephone_list()   {}
    virtual std::string get_name() const = 0;
    virtual std::pair<bool, std::string>
    get_number(const std::string& person)
               const = 0;
    virtual telephone_list&
    add_entry(const std::string& name,
              const std::string& number) = 0;
  protected:
    telephone_list()   {}
    telephone_list(const telephone_list& rhs)
      {}
  private:
    telephone_list& operator=
     (const telephone_list& rhs);
  };
}
#endif
```

Visiting Alice

by Phil Bass

"The time has come," the Walrus said,
"To talk of many things:
Of tuples, trees and composites;
Of visitors and kings."[1]

Welcome

"Good morning, everyone, and welcome to the Wonderland Social Club annual treasure hunt. I am the Walrus." (*coo-coo coo-choo*) "Well, not a walrus, but I am quite long in the tooth." (*groan*)

"This year the clues are all in trees. On each clue sheet there's a clue to an item of treasure. Some clue sheets also contain two further clues, which lead to more clue sheets. With each treasure clue there is an indication of the value of the treasure at that location."

"You have until 6 o'clock this evening to find as much treasure as you can. The team with the most valuable hoard of treasure will be the winner. The first clue is outside in the garden. See you back here in the Carpenter's Arms at 6 o'clock. Good luck everybody!"

Planning the Route

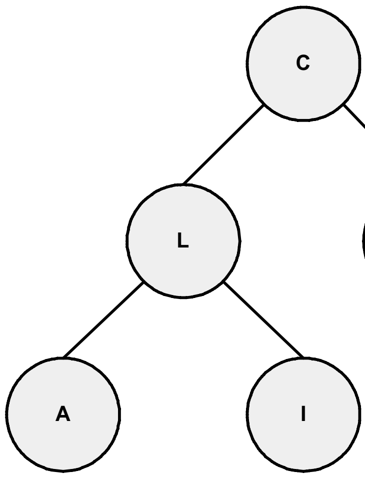
There were three teams: four trainee nurses called the Pre-Tenders, three employees of the Royal Mail called the Post Men and two publicans called the Inn Keepers. The Pre-Tenders decided to do the easy clues first; the Post Men chose to visit the nearest places first; and the Inn Keepers settled for finding the most valuable treasure first.

Overload readers will have spotted immediately that the treasure hunters' problem involves the traversal of an abstract binary tree. The Walrus had drawn the tree on a sheet of paper so that he could refer to it when he was adding up the scores at the end of the day. And, as it turned out, the Pre-Tenders would visit the treasure locations in pre-order sequence, the Post Men in post-order sequence and the Inn Keepers in in-order sequence.

Encoding the Problem

Bill, a nerdy-looking youth with thick oyster-shell glasses had spotted this, too. He was a C# programmer and was often to be seen in the corner of the bar with a beer and a lap-top. To him the treasure hunters' tree seemed to be a set of dynamically constructed, garbage collected, polymorphic objects. "It's a binary tree", he said to his best mate, Ben. "Yes", agreed Ben, but Ben's mental imagery was very different. "Nested structs", said Ben.

Bill looked at him blankly for a moment, decided Ben must have been joking and replied with an ironic, "Yeah, right". But Ben was a C++ programmer and he wasn't joking. "No, really" said Ben, pulling out his own lap-top, "Look, I'll show you what I mean".



```
struct L
{
  char a, i;
};

struct C
{
  L    l;
  char e;
};
```

Listing 1 - A simple tree as nested structs

1 After Lewis Caroll's "The Walrus and the Carpenter". By the way, he lied about the kings.

[continued from previous page]

file for the purpose - `<iosfwd>`. While this doesn't avoid including any header it is much simpler than `<ostream>` and with its aid we can write Listing 4.

Conclusion

Designing C++ header files is like many tasks in developing computer software – if done badly it can cause major problems. I hope that I've shown that, given the right techniques and idioms plus an understanding of the issues, doing it well isn't so terribly hard.

Alan Griffiths
<alan@octopull.demon.co.uk>

References

1. "An Introduction to Preprocessor Metaprogramming", David Abrahams and Aleksey Gurtovoy, `http://boost-consulting.com/tmpbook/preprocessor.html`
2 "Constructing bullet-proof classes", John Carrolan, Proceeding C++ at Work - SIGS
3 Gamma, Helm, Johnson & Vlissides. "Design Patterns", Addison-Wesley, 1995
4 Herb Sutter. Exceptional C++, Addison-Wesley, 2000
5 "To Grin Again", Alan Griffiths, *Overload 72*.
6 "C++ Interface Classes – An Introduction", Mark Radford, *Overload 62*

See also "Separating Interface and Implementation in C++", Alan Griffiths and Mark Radford, Overload 66