Separate Compilation / File Organization in C++

Lee S. Koh

- C++ has facilities for dividing a program into parts that are kept in separate files, compiled separately, and then linked together to give a single *executable file*. Typically, there are 2 facets to using these facilities:
  - A program typically uses components. The components and the program itself are placed in separate files.
    - (The components may have been created by other parties not directly associated with the program.)
    - The file for the program is called the *application file* (or *client file* or *driver file*).
  - Each component (built using `class`, `struct`, *etc.*) is defined in a pair of files so that the specification of <u>what</u> the component does (how it is to be used) is separate from <u>how</u> the component is implemented.
    - The file for the specification part (which describes the component's interface) is called the *interface file*.
    - The file for the implementation part is called, well, the *implementation file*.
    - Such an "interface-implementation" pair of files is often referred to as a *unit* or *module*. It may be compiled separately (thus the phrase "C/C++ support for ***separate compilation***"), the result of which is an *object file*.

  Why all this fuss? Because doing so befits ***information hiding*** and ***modular programming***, which can bring about various advantages, including:

  - ***Reusability:*** A library of components (modules) can be built; many programs can use the same components.
  - ***Maintainability:*** A major part of maintenance arises from the need for change due to bugs, requirements, *etc.* Mitigating the impact of change is thus key to tackling maintenance. *Information hiding* and *modular programming* both aid in reducing such impact through *isolation/localization*: when a module need change, the impact should ideally be localized to that module; *interface tends to change less often than implementation* so it makes sense to isolate the implementation (and hide it from whoever need not to know). In rather oversimplified terms, with separate files, if certain module needs fixing, we only need to change and recompile one implementation file; the other files, including the application file, need not be changed or even recompiled.
  - ***Manageability:*** Modularly structuring a program facilitates division, testing, *etc.* of work/responsibilities.

  Some further notes:

  - ***Source file:*** A more inclusive term that can refer to either an implementation file or an application file.

    Header file: A more exclusive (to C/C++) term for interface file.
    - Any file having ties to a component must `#include` that component's interface file.
      - The implementation file implementing the component must do so.
      - An application file making use of the component must do so.
      - An interface file making references (in some function prototypes, say) to the component must do so.
    - A file having ties to a component would typically `#include` the component's interface file at the top (head), thus the use of the term header file (in C/C++) to refer to the interface file.
  - In general, an application might have ties to several components and each component might be kept in a separate (interface-implementation) pair of files.

    Furthermore, the components and the application itself might have intertwined ties. For instance, suppose application **a1** uses components **c1** and **c2**; it may turn out that **c2** also uses **c1** (or **c1** also uses **c2**).

    A good possibility therefore exists (especially for programs with some complexity) for the *multiple inclusion* of a component's header file in another (header or source) file, and a compilation error results if that happens.

    To avoid the problem, C/C++ provides a way (called a *macro guard* or *inclusion guard*) for attaching a section of code with an "if it's already included once before, don't include it again" tag as exemplified below:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
... // header file proper
#endif
```

In this case, the header file is named **`MyHeader.h`**, and the **`MY_HEADER_H`** seen used is derived from that name following a common convention (which should be quite self-evident).

- Basic guidelines on: (NOTE: There are more details still to be accounted for in more involved situations.)
  - What typically are included in the ***interface*** (***header***) ***file***:
    - Documentation comments that are for consumption by *users*.
      - Including *preconditions*, *postconditions*, whether safe to use *value semantics*, and additional usage notes.
        - To be "safe to use value semantics", value-copying (deep copying) and not pointer-copying (shallow copying) must apply when copying component objects (via copy construction or copy assignment).
    - Macro (or inclusion) guard.
    - Relevant header file(s).
      - Such as header file(s) of other component(s) referenced in this header file.
      - (A header file is usually irrelevant if excluding it won't cause any error when compiling the module.)
    - Relevant globals.
      - Such as global constant(s) and `typedef`(s).
      - (A global is usually irrelevant if excluding it won't lead to any error when compiling the module.)
    - `class` (and/or `struct`) definition(s).
      - Including component-level constant(s) and `typedef`(s), and member variables.
      - Including declarations (prototypes) of functions specifying operations for the component.
        - They may be member functions, `friend` functions, ordinary functions and `operator` functions.
    - (`using namespace std;` directive should usually not be included. Why?)
  - What typically are included in the ***implementation file***:
    - Documentation comments that are for consumption by *implementers*.
      - Including preconditions and postconditions (for functions meant only for *implementers*), and *invariant*.
        - Invariant = set of rules that remain uncompromised as a component object is being put to use.
    - Relevant header file(s).
      - (A header file is usually irrelevant if excluding it won't cause any error when compiling the module.)
    - Definitions of functions specified in the matching interface (header) file.
      - (This usually makes up the bulk of the implementation file.)
  - What typically are included in the ***application file***:
    - Relevant header file(s).
      - (A header file is usually irrelevant if excluding it won't cause any error when compiling the application.)
    - Prototypes of functions that logically belong to the application.
      - Such as a menu-displaying function.
    - Relevant globals.
      - Such as constant(s) and `typedef`(s).
      - (A global is usually irrelevant if excluding it won't lead to any error when compiling the application.)
    - The `main` function.
    - Definitions of functions that logically belong to the application.
- Shortcoming of C++ `class` in supporting "separation of interface from implementation"
  - The non-`public` part of a `class` (*e.g.*: `private` member variables and functions) is actually part of the implementation (and not the interface). But C++ won't allow us to split the `class` definition across 2 files.
  - The usual weighed compromise is to place the entire `class` definition in the interface file, as is done above. (To get into how to overcome the shortcoming will be too distractive for our purpose.)