Update + Resumen de "C Header File Guidelines" por David Kieras, EECS Dept / U Michigan – 2012

R.O. 10-2016 / rev 11-2017

A) RESUMEN DE REGLAS:

Regla#1: Cada módulo con su (*.h; *.c) debería corresponder a una funcionalidad claramente definida.

Regla#2: Siempre utilice "include guards" o protectores de inclusión, en un *.h.

Regla#3: Todas las declaraciones necesarias para usar un módulo deben estar en el *.h correspondiente, y este archivo siempre se utiliza para acceder a ese módulo.

Regla#4: El header / *.h sólo contiene declaraciones, y es incluido por el archivo .c del módulo.

Regla#5: Ingrese las variables que son globales a todo el programa con un extern declaration en el .h, y una declaración de definición en el .c de ese módulo.

Regla#6: Mantenga las declaraciones internas de un módulo fuera del archivo header (.h).

Regla#7: Cada header A.h debería #incluir cualquier otro header que A.h requiera para compilar correctamente, pero no mas.

Regla#8: Si sirve hacer solamente una declaración incompleta de la estructura tipo X, usela en lugar de #incluir el X.h completo.

Regla#9: El contenido de un header debería compilar correctamente por si solo.

Regla#10: Un archivo A.c primero debería #incluir su A.h, y después cualquier otro header que requiera.

Regla#11: Nunca #incluir un .c por ninguna razón!

B) EXPLICACIÓN

1. **¿Qué debe ir en los archivos de encabezado (*header* o *.h) de un proyecto complejo?**

Los programas de C y C++ toman habitualmente la forma de un conjunto de módulos compilados separadamente. Gracias a esta separación, durante el desarrollo de un proyecto grande, un nuevo ejecutable puede ser ensamblado rápidamente si solamente los módulos alterados necesitan recompilación.

Cada módulo consta de variables globales, funcionas y structs. Las funciones generalmente están definidas en un archivo .c (*source*), y (salvo el módulo principal) cada .c tiene un .h asociado. La idea es que cada módulo pueda acceder a la funcionalidad de un módulo X simplemente con #include X.h para el header, y el

linker hace el resto. El código en X.c sólo necesita ser compilado una vez, o cuando se realicen modificaciones. Esto permite a la utilidad Make de Unix o al IDE realizar un trabajo eficiente de armado de nuevas versiones.
Un programa en C bien organizado tiene una buena elección de módulos, y *headers* bien construidos que facilitan la comprensión y acceso a las funcionalidades de cada módulo. También ayuda a asegurar que el programa utiliza las mismas declaraciones y definiciones a lo largo de todos los componentes. Esto es importante para auxiliar a los compiladores en mantener la regla básica de "Una sola Definición" o *One Definition Rule (ODR)* [1] definida para C++ pero válida en C también. Por último, los *headers* bien diseñados tienen mínimo "acoplamiento" en el sentido de reducir lo más posible la cantidad de `#include` que cada módulo agrega – esto reduce el tiempo de compilación y facilita mucho la organización y depuración del código.

Las siguientes reglas sintetizan las mejores prácticas para armar las duplas de header y source (*.h; *.c) para mayor claridad y efectividad de compilación:

Regla#1: Cada módulo con su (*.h; *.c) debería corresponder a una funcionalidad claramente definida. Conceptualmente, un modulo es un grupo de declaraciones y funciones que puede ser desarrollado y mantenido separadamente de otros módulos, y reutilizado de ser necesario. No conviene forzar juntas cosas que serán usadas o mantenidas por separado, y separar elementos que siempre se trabajan juntos. Dentro de las librerías estándar, los módulos math.h y string.h son claros ejemplos.

Regla#2: Siempre utilice "include guards" o protectores de inclusión, en un *.h. La forma más compacta de estos protectores utiliza el #ifndef. Utlice un símbolo de guarda basado en el nombre del header, dado que estos símbolos son sencillos y generalmente los nombres de archivos son únicos en cada proyecto. Por ejemplo para geo_base.h, se comienza con:

```
#ifndef GEO_BASE_H
#define GEO_BASE_H
```

y finaliza con:

```
#endif
```

(no comenzar los símbolos de guard con _ : estos son reservados para uso interno de los compiladores, y conducen a errores muy extraños..)

Regla#3: Todas las declaraciones necesarias para usar un módulo deben estar en el *.h correspondiente, y este archivo siempre se utiliza para acceder a ese módulo . El uso y la inclusión de este header provee toda la información requerida para usar el módulo, compilarlo y linkearlo correctamente. Mas aún, si el módulo A necesita utilizar la funcionalidad del módulo X, debería hacer un `#include X.h` y nunca contener declaraciones propias de módulos o funciones que aparezcan en X. Porqué? Porque si el módulo X se cambia, y uno olvida modificar las declaraciones "caseras" en el módulo A, podría conducir a extraños errores de run-time que pasarían sin detectar por el compilador y linker. Esto es una violación de la ODR que los compiladores / linkers de C no pueden detectar. Siempre accediendo a un módulo a través de su header asegura que un único conjunto de declaraciones necesita ser mantenido, y ayuda a cumplir la ODR.

Regla#4: El header / *.h sólo contiene declaraciones, y es incluido por el archivo .c del módulo . Solamente inserte en el .h declaraciones de los tipos:

- estructuras
- prototipos de funciones
- variables tipo extern global

Solamente inserte en el .c lo siguiente:

- definición de funciones
- definición de variables globales

El archivo .c debe #include el archivo .h del módulo. El compilador puede entonces detectar discrepancias entre ambos, y asegurar la consistencia.

Regla#5: Ingrese las variables que son globales a todo el programa con un extern declaration en el .h, y una declaración de definición en el .c de ese módulo . Para variables globales que son vistas desde todo punto del programa, ponga una declaración con extern en el archivo X.h, como en:

```
extern int g_numero_items;
```

Los otros módulos A, B,.. sólo #incluyen el X.h – el archivo X.c debe #incluir el X.h, y cerca del comienzo del archivo debe aparecer una declaración que defina e inicialice la variable global, aquí:

```
int g_numero_items = 0;
```

Por supuesto, podría definirse otro valor pero una definición a 0 es de práctica habitual para indicar una *definición* – esto debería ser un punto único. Esto es la metodología aceptada en C++ y es trasladable a C para asegurar el cumplimiento de la ODR.

Regla#6: Mantenga las declaraciones internas de un módulo fuera del archivo header (.h) . Algunas veces, un módulo utiliza componentes internos que no deberían poder ser accesibles desde otros puntos del programa. Ponga estas variables, estructuras y funciones cerca del inicio del archivo .c, y no las mencione en el .h. Mas aún, declare estas variables y funciones como `static` en el comienzo del archivo .c para que tengan un linkeado interno a ese archivo – de esta manera, los otros módulos no pueden saber de estas declaraciones que son internas a este módulo.

Regla#7: Cada header A.h debería #incluir cualquier otro header que A.h requiera para compilar correctamente, pero no mas. Si A.h necesita de una estructura de X, entonces #incluir X.h en el A.h, de modo que el compilador sepa de qué tamaño es esa estructura. No incluir en A.h encabezados que sólo el A.c necesita. Por ejemplo, math.h sólo se incluiría en A.c porque habitualmente es requerido por las definiciones de función.

Regla#8: Si sirve hacer solamente una declaración incompleta de la estructura tipo X, usela en lugar de #incluir el X.h completo. Si la estructura del tipo X sólo aparece como tipo puntero en otra struct, o en una función, y el código en el header no intenta acceder a variables miembros de X, entonces no #incluir el X.h sinó realizar una "forward declaration" de X.

Ejemplo: La estructura Cosa se refiere a X a través de un puntero:

```
struct X;   // forward declaration

struct Cosa{
      int I;
```

```
struct X* x_ptr;
};
```

El compilador no protestará con esto porque los punteros siempre tienen el mismo tamaño y comportamiento, sin importar el tamaño del objeto al que apuntan. Tipicamente, sólo el códico en el .c necesita acceder a los miembros de X, por lo cual el .c sí deberá #incluir X.h. Esto tiende a encapsular módulos y desacoplarlos de otros.

Regla#9: El contenido de un header debería compilar correctamente por si solo. Un archivo header debería #incluir o declarar "forward" todo lo que necesita para compilar. Si esta regla no se observa, pueden aparecer errores extraños cuando se cambian otros encabezados o sus #includes.
Como verificación, realice e intente compilar un test.c que sólo contenga #include A.h. No debería producir errores de compilación. Si lo hace, algo requiere declaración forward o #inclusion. Pruebe todos los .h de un proyecto empezando por la parte inferior de la jerarquía, y trabaje hacia arriba, esto contribuirá a eliminar dependencias entre archivos header.

Regla#10: Un archivo A.c primero debería #incluir su A.h, y después cualquier otro header que requiera. Siempre haga esta inclusión del propio header al inicio, para evitar "esconder" cualquier elemento que falta, y es #incluido por otros .h. Después, si la implementación del código de A utiliza X, entonces realice la #inclusión de X.h en A.c, para que A.c no dependa de que X.h sea accidentalmente #incluido en otra parte.
No hay consenso claro respecto de si A.C debería #incluir archivos que A.h ya ha explícitamente #incluido. Dos sugerencias:

- Si el X.h es un requerimiento indispensable para que A.h compile, entonces #incluirlo en A.c es redundante.
- Siempre #incluir X.h en A.c es una manera de dejarle claro al lector que se está usando X, y ayuda a forzar que las declaraciones de X estén aunque A.h cambie por alteraciones de diseño futuras. Entonces está OK incluir X.h aunque sea redundante (siempre que esté en uso!)

Regla#11: Nunca #incluir un .c por ninguna razón! Puede que haya código que no consista de declaraciones normales y definiciones, entonces uno está tentado de usar un #include "pepe.c" – evitar esto por errores difíciles de detectar. Si es imprescindible, hacer un "pepe.inc" u otra notación.

[1] ISO C++ Standard (ISO/IEC 14882) 2003,