MANUAL DE ESTILO C / C++

Oscar Valtueña García

http://www.geocities.com/webmeyer/prog/estilocpp/

Manual de estilo C/C++

http://www.geocities.com/webmeyer/prog/estilocpp/

Versión: 1.1.0, Enero de 2005
(c) 2005 Oscar Valtueña García

oscarvalt@gmail.com

INDICE

1. Introducción

A la hora de programar es muy importante seguir un estilo.
Este nos permitirá revisar, mantener y actualizar el código de una manera más sencilla y ordenada.
Seguir estas normas también evitará que caigamos en errores y vicios que dificultan la comprensión de las líneas de código y que pueden ocultar posibles "efectos colaterales" que podrían hacerse visibles en cualquier momento (para desgracia y desesperación del programador).

Históricamente, el primer objetivo de todo esto fue tener mi propio manual de estilo, para aplicarlo en mis desarrollos. Para ello recopilé indicaciones y consejos de multitud de manuales, de grandes expertos en programación y mi propia experiencia.
Cuando fue creciendo el manual consideré adecuado ponerlo en la Web, para que pudiera ser consultado por cualquiera que se encontrara en mi misma situación.

Seguramente vaya actualizando y aumentando el manual de manera paulatina, y seguramente haya fallos, errores u omisiones que hay que corregir... (si ves alguno de estos, o quieres hacer algún comentario, madame un correo a oscarvalt@gmail.com).

Todas las normas y consejos que siguen se pueden resumir de una manera sencilla en:

Hay que evitar técnicas que atenten contra la modularidad y la encapsulación del código.
Hay que conseguir un código sencillo, legible y fácil de mantener.

2. Sobre el código

2.1 Líneas

* Es aconsejable no utilizar más de una instrucción por línea de código. Para conservar la legibilidad del mismo

Por ejemplo, la siguiente línea no es muy clara:

```
nValor=nA*2; nValor2=nA+12; nValor3=nValor+nValor2;
// esto puede producir errores de interpretación.
```

Queda más claro de la siguiente manera:

```
nValor  = nA*2;
nValor2 = nA+12;
nValor3 = nValor+nValor2;
// ahora están más claras las diferentes asignaciones.
```

* Las líneas de código no deberían ser superiores a 80 caracteres, que es el tamaño de línea habitual en editores de texto plano de sistemas operativos como Unix y Linux.

Muchas emulaciones y algunos editores no soportan más de 80 caracteres por línea. Líneas mayores provocarían incomodidad al revisar el código (ya que las líneas aparecerían partidas), pudiendo dar lugar a posibles errores en la interpretación.

2.2 Indentación o sangrado

* Es aconsejable que el código esté indentado, de una forma adecuada y consistente. La indentación (o sangrado) ayuda a ver la estructura del programa y facilita la identificación de posibles errores

Existen multitud de editores y formateadores de código que ayudan en esta tarea, pero es aconsejable ir haciendo la indentación sobre la marcha.

Al indentar (o sangrar) el código se presenta de una manera clara la agrupación en bloques del programa, consiguiendo que sea más sencillo relacionar las llaves de cierre con sus correspondientes llaves de apertura.

El siguiente trozo de código puede dar problemas en su comprensión:

```
for (i=0; i<nMaximo; i++)
{
if (bValor)
{
hazOperacionParcial();
ponResultadoParcial ();
if (i > nMedio)
{
```

```
hazOperacionMayorMedio ();
ponResultadoMayorMedio ();
}
}
ponSiguienteResultado ();
}
// Al no haber sangrado parece que sobran llaves de cierre.
// o no se sabe muy bien cuando se ejecutan las funciones.
```

Queda más claro de la siguiente manera:

```
for (i=0; i<nMaximo; i++)
   {
     if (bValor)
       {
         hazOperacionParcial();
         ponResultadoParcial ();
         if (i > nMedio)
           {
             hazOperacionMayorMedio ();
             ponResultadoMayorMedio ();
           }
       }
     ponSiguienteResultado ();
   }
// Con el sangrado queda muy clara la agrupación en
// bloques del código.
```

2.3 Bloques

* Deben utilizarse llaves para delimitar todos los bloques de sentencias de control y bucles.

Las llaves deben estar en la misma columna (y preferentemente solos, sin código ejecutable).

* Habitualmente puede resultar útil señalar mediante comentarios el principio y el fin de bloques grandes o muy importantes dentro de la estructura del programa.

Por ejemplo el siguiente caso:

```
if (bInsertar)
  { // ********* Inicio de insercción de datos *********
    ...
    ...
    // Aquí iría el código del bloque de insercción de datos
    // un montón de líneas, con bloques internos, etc...
    ...
    ...
  } // ********* Fin de insercción de datos ************
```

2.4 Paréntesis

* Hay que hacer uso de los paréntesis para hacer más claro el orden de evaluación de operadores en una expresión, incluso en situaciones en las que puede no ser obligatorio utilizarlos

Por ejemplo, la siguiente expresión, sin paréntesis, es correcta, pero poco legible:

```
int i = a>=b && c<d && e+f <= g+h; // ¿Cual es el orden de evaluacion?
```

De la siguiente manera es igual de correcta, pero más comprensible:

```
int j = (a>=b) && (c<d) && ( (e+f) <= (g+h) ); // Expresion clara
```

* Pese a todo, hay que tener cuidado y limitarse en el uso de los paréntesis, que pueden complicar la lectura del código.

Si se tienen más de dos niveles de paréntesis en una sola expresión habría que dividirla en varias expresiones más sencillas (con cuatro niveles de paréntesis la lectura ya es sumamente complicada).

De esta manera conseguimos un código resultante que puede ser igual de compacto, y resultará mucho más legible.

Por ejemplo, en vez de usar:

```
// demasiados paréntesis que hacen ilegible el código
if (((4+(i*6))==x) && ((((x/5)-j)>=y) || ((6*x) != k))) hazAlgo();
```

Es más legible utilizar lo siguiente:

```
bool test1 = (4+(i*6) == x);
bool test2 = ((x/5)-j >= y);
bool test3 = (6*x != k);
// conseguimos una condición más comprensible
if (test1 && (test2 || test3)) hazAlgo();
```

2.5 Espacios en blanco

* Debe hacerse un uso generoso de líneas y espacios en blanco en pro de hacer más claro y comprensible el código.

Las funciones y los bloques de código deben aislarse unos de otros, usando varias líneas en blanco para facilitar su lectura y poner de manifiesto su caracter de unidad de programación.

Así conseguiremos tener un código de facil lectura (si fuera excesivamente compacto sería poco legible), y en el que sea sencillo localizar funciones y bloques de código.

* Se pueden utilizar espacios en blanco para alinear la parte interna de sentencias o estrucutras, para así facilitar su lectura.

Por ejemplo, en el siguiente código se han "colocado" partes de código para facilitar su lectura utilizando espacios en blanco:

```
int   nContador = 0;
int   vVar      = 0
char* psAux     = NULL;
...
...
incrementaVariable (nContador,  2);
incrementaVariable (nVar,      32);
...
```

2.6 Comentarios

* Los comentarios deben ser minimizados (utilizar comentarios muy largos puede entorpecer la lectura), deben facilitar la comprensión del código y aportar información útil sobre las sentencias, bloques, variables, funciones, etc... que afectan.

En ningún momento hay que poner demasiados comentarios (comentar todas las instrucciones utilizadas, por ejemplo, no aporta nada y entorpecen la lectura) que puedan dificultar la comprensión del programa o que no aporten ninguna información útil sobre el mismo.

* Es aconsejable comentar los ficheros, funciones y bloques de código, para así ofrecer información útil sobre su funcionalidad o su uso, pero sin extenderse de forma innecesaria.

En otros apartados se comenta esto mismo, indicando la estructura de estos comentarios explicativos.

* Se recomienda comentar las declaraciones de datos no auto explicativas, y la omisión intencionada de código.

* Hay mantener y actualizar los comentarios a la vez que se mantiene y actualiza el código.

Unos comentarios no actualizados pueden llevar a error al revisar el fuente.

* En C los comentarios solamente se construyen con /* ..... */.

```
/* Estilo de comentario en C */

/* Comentario que se escribiria en C
 * dividido en varias lineas. */
```

En C hay que tener mucho cuidado al utilizar los comentarios que abarquen varias líneas.

Algunos editores marcan con /* la primera linea del comentario y con * las siguientes, la última terminaría en */ (como en el ejemplo mostrado).

* En C++ los comentarios tendrían que ser solamente los construidos con //.

```
      // Estilo de comentario en C++

      // Comentario que se escribiria en C++
      // dividido en varias lineas.
```

* En caso de trabajar en C++ se aconseja utilizar solamente los comentarios propios del C++. Los comentarios de C pueden llevar a errores ocultos en caso de tener comentarios anidados (que en unos compiladores producirían errores, y en otros no).

El siguiente ejemplo de comentario puede no compilar en algunos compiladores

```
/* Una primera linea de error
   /* Un comentario anidado */
   El comentario anidado puede provocar que no se compile.
   en determinados compiladores */
```

2.7 Código

* Nunca hay que poner rutas absolutas en el código. Ya que con ello conseguimos que el ejecutable dependa de la ruta absoluta indicada en la compilación, y tanto el código fuente como el ejecutable serían no portables.

Si se necesita acceder a ficheros, directorios, etc... hay que usar otros métodos para llegar a ellos, como argumentos, variables de entorno, ficheros de configuración, etc...

* Si fuera necesaria la portabilidad del código entre diferentes plataformas, se separarán las líneas propias de cada plataforma mediante sentencias de precompilación

#if defined() ... #else ... #endif

(usando defines que estarán bien documentados en la cabecera del fichero).

Por ejemplo el siguiente caso:

```
// Para windows se definirará WIN32 en la compilación
#if defined (WIN32)
  struct _stat stbuf;
  resultado = _stat (path_evaluar, &stbuf);
#else
  struct stat stbuf;
  resultado = stat (path_evaluar, &stbuf);
#endif
```

2.8 Compilación

* Se aconseja compilar usando el nivel máximo de warnings disponible en el compilador y eliminar todos los avisos que aparezcan.

Generalmente estos warnings hacen referencia a líneas de código problemáticas que es conveniente (aunque no sea imprescindible) solucionar.

* Como se ha comentado en un punto anterior, hay que utilizar defines de precompilación para separar el código dependiente de cada plataforma cuando se busque la portabilidad de los ficheros fuente.

* En la cabecera del fichero fuente hay que indicar los defines de precompilación que pueden utilizar y el motivo para utilizarlos.

Conviene no abusar de los defines y sentencias de precompilación, ya que añaden complejidad al código.

3. Variables

3.1 Nomenclatura de las variables

* Los nombres utilizados para las variables tienen que ser autoexplicativos. De manera que en el propio nombre esté la indicación del uso o finalidad de las variables.

* Para componer los nombres de las variables se utilizarán principalmente sustantivos, pudiendo calificarse con adjetivos.

Esta norma se establece para mejorar la identificación de los distintos componentes de un sistema.

* Una práctica muy habitual es utilizar una serie de prefijos para poder identificar el tipo de variable de una forma sencilla y rápida. Este tipo de nomenclaturas se denomina notación húngara (ideada por Charles Simonyi, arquitecto jefe de Microsoft, [2]).

Estos prefijos se añaden al nombre de la variable, sin separarlos de ninguna forma, como se muestra en los ejemplos de la tabla.

Tipo	Prefijo	Ejemplo
`void`	`v`	`void vVariableVacía;`
`bool`	`b`	`bool bOperacionValida;`
`char`	`c`	`char cTeclaPulsada;`
`int`	`n`	`int nNumeroCeldas;`
`long`	`l`	`long lTotalUnidades;`
`float`	`f`	`float fPrecioUnitario;`
`double`	`d`	`double dAnguloMinimo;`
`* (puntero)`	`p`	`int * pnMatrizValores;`
`& (referencia (C++))`	`r`	`float & rfMaximoAngulo;`
`[] (array)`	`a`	`double afRangosMaximos[3];`
`enum (enumeración)`	`e`	`EBoole eResultado;`

Como se ve en los mismos ejemplos indicados en esta tabla, se pueden utilizar varios de estos prefijos en una misma definición.

Para el tema de cadenas de caracteres (`strings`, no confundir con la clase `String`), podríamos pensar que la forma correcta de definirlas es, por ejemplo: `char acNombrefichero[256+1];`. Pero para poder identificarlas, ya que se usan de manera extendida, podríamos usar el prefijo `s`. Por ejemplo:

```
// definición de una cadena de caracteres
char sNombreFichero [256+1];
```

De forma adicional, podríamos diferenciar la definición de una cadena de caracteres mediante el uso de `[] (corchetes)` (como en el ejemplo anterior) de la definición de un puntero a carácter `(char *)` que contendría la dirección de memoria donde empieza esta cadena de caracteres.

```
// definición de un puntero a una zona de memoria que contiene una
cadena
char *psPunteroAlNombre;
```

Estas dos definiciones se usan para esos casos, y quizás contradigan en parte la anterior norma de nomenclatura, pero pueden ayudar a diferenciar las variables "cadena de caracteres" definidas por [] (corchetes) de las definidas mediante char * (puntero a carácter), ya que pueden ser tratadas de manera muy diferente por los compiladores, y la forma de utilizarlas por el programador es diferente.

Para otros tipos de variable no especificados en la tabla anterior no se establece prefijo, pero se puede utilizar algún otro definido por otra norma o por el propio usuario.

Por ejemplo los siguientes:

Tipo	Prefijo
byte o unsigned char (uchar)	by
word o unsigned int (uint)	w
unsigned long (dword)	dw
handle (manejador, puntero de objetos de windows)	h

* Para el contador de bucle for se pueden utilizar variables numéricas llamada i, j... siempre que esto no perjudique en la comprensión del código (y estas variables se utilicen solamente dentro del bucle del for).

Para estos casos (C++) se recomienda declarar la variable i en el propio bucle.

```
Ejemplo:
for (int i=0; i<strlen(sCadenaPrueba); i++) { ... }
```

3.2 Utilización de variables

* Todas las variables hay que declararlas en el ámbito más restringido posible.

Esto lleva a una utilización más eficiente de la pila y a una reducción de los posibles errores por efectos "colaterales".

* En general se aconseja limitar al máximo posible la visibilidad de las variables. Como norma general de protección de datos.

* Se debe evitar la conversión explícita de tipos de datos, ya que atenta contra la comprobación de tipos de datos que hace el compilador.

Se aconseja utilizar la conversión implícita de tipos de datos cuando sea posible.

* Se aconseja que cada variable se declare de forma separada.

Así se desaconseja una definición como la siguiente:

```
// Multiple definición de variables, desaconsejada
int nContador, nNumeroFilas, nNumeroColumnas, nTotalElementos;
```

Mientras que lo correcto sería lo siguiente:

```
// Definición correcta de variables.
int nContador;
int nNumeroFilas;
int nNumeroColumnas;
int nTotalElementos;
```

* Usar unsigned para variables que se sepa con seguridad que nunca van a tomar valores negativos.

* Es aconsejable que todas las variables sean inicializadas en la propia declaración de las mismas.
Por ejemplo:

```
// Definición más correcta de variables, con inicialización.
int nContador       = 0;
int nNumeroFilas    = 0;
int nNumeroColumnas = 0;
int nTotalElementos = 0;
```

3.3 Variables locales

* Las variables locales se deberían definir solamente en el bloque en el que se vayan a utilizar (en C++).
Así mejoraríamos el uso de la pila.

* Las variables locales se deberían definir justo antes de su utilización (en C++).
De esta manera se evita el error habitual de tener variables definidas que luego no se utilizan.

3.4 Variables globales

* En general: NO UTILIZAR VARIABLES GLOBALES salvo en caso totalmente inevitable.
La existencia de variables globales atenta contra la comprensión del código y su encapsulamiento, además de que puede provocar efectos "colaterales" inesperados (si una función varía el valor de la variable global de forma no controlada) que desembocan en errores muy difíciles de identificar.

* En el caso completamente inevitable de tener que utilizar variables globales, documentar ampliamente en los ficheros y funciones que variables globales se van a utilizar, y cuales se van a modificar.

3.5 Variables estáticas

* Se recomienda minimizar el uso de variables estáticas, y solamente para casos en los que se necesite "recordar" el estado de una función entre llamadas consecutivas a la misma.
No se deberían utilizar las variables estáticas como variables globales ocultas.

* Las variables estáticas se tienen que inicializar en el momento en que se declaran, de manera obligatoria.

3.6 Inicialización de variables

* Todas las variables hay que inicializarlas explícitamente antes de ser utilizadas, en especial las que se alojan en la pila y las que obtienen espacio de almacenamiento de forma dinámica durante la ejecución.
De esta manera evitamos la ejecución de código con valores desconocidos de las variables, que provoca el funcionamiento aleatorio del programa.

* Las variables estáticas se tienen que inicializar en el momento en que se declaran.

3.7 Asignación de memoria

* Se aconseja el uso de new y delete para C++, en lugar de malloc, calloc y free (propios de C), ya que los operadores new y delete están especialmente diseñados para trabajar con clases y llamar a los constructores y destructores (respectivamente) de las mismas.

* Siempre que se alloque memoria de manera dinámica (tanto con malloc y calloc (propios de C) como con new (propio de C++)) hay que asegurarse de que el espacio requerido fué, efectivamente, servido por el sistema operativo. Nunca se debería suponer que una asignación de memoria ha terminado correctamente.
Si no se comprueba si se ha reservado correctamente la memoria se pueden obtener errores fatales, difíciles de detectar en pruebas y difíciles de localizar.

Un ejemplo correcto es el siguiente:

```
char *psCadena = new char[24];
// se comprueba si se ha obtenido memoria realmente
if (0 == psCadena) return ERROR_RESERVA_MEMORIA;
...
```

* Hay que liberar toda la memoria que se haya reservado de manera dinámica. No hay que esperar que "algo" libere la memoria, hay que hacerlo explícitamente.

* Si una variable ya tiene asignada memoria de manera dinámica, hay que liberarla antes de cambiar y asignarle nueva memoria.

* Cuando se libere memoria con delete (C++) reservada para un array, hay que indicar el operador [] (corchetes).

Esto se hace por legibilidad, y es obligatorio para algunos compiladores.

Así sería incorrecto indicar...

```
char *psCadena = new char[24];
...
delete psCadena; // Uso incorrecto de delete para un array
```

En cambio lo correcto seria indicar...

```
char *psCadena = new char[24];
...
delete [] psCadena; // Uso correcto de delete para un array
```

* Cuando se libera memoria asignada dinámicamente (tanto con free (propio de C) como con delete (propio de C++)) siempre se asignará a 0 el puntero superviviente.

De esta manera se minimizan los problemas que surgen de una mala gestión de memoria.

Por ejemplo:

```
delete pAuxiliar; // Liberamos memoria
pAuxiliar = 0; // asignamos a 0 el puntero resultante.
```

* En el uso de punteros se desaconseja la comparación con NULL o la asignación a NULL. Es preferible realizar la comparación con 0 o la asignación a 0.

NULL es un estándar ANSI-C definido como (void*)0 o 0. Si NULL esta definido de tipo (void *) no se puede asignar arbitrariamente a otro tipo de puntero sin utilizar una conversión explícita. Por esta razón es preferible utilizar 0 en asignaciones y comparaciones con punteros.

Ejemplo:

```
char *psCadena = 0;
... // unas cuantas líneas de código
psCadena = new char[24];
if (0 == psCadena) return ERROR_RESERVA_MEMORIA;
... // muchas líneas de código más adelante
delete psCadena;
psCadena = 0;
```

* Se desaconseja el uso de punteros a punteros. Estas técnicas complican mucho la sintaxis y la comprensión del código, pudiendo ser sustituidas utilizando otras técnicas.

3.8 Cadenas de caracteres

* Como se ha dicho anteriormente (notación húngara), en la definición de cadenas de caracteres se puede utilizar el prefijo s o el prefijo ps, dependiendo del caso, para identificar el tipo de variable, como por ejemplo:

```
char sNombreFichero [256+1]; // cadena con memoria ya reservada
char *sPunteroAlNombre=0;    // apuntará a una cadena en memoria
```

* La memoria necesaria para almacenar una cadena de N caracteres se definirá como una zona de memoria para N+1 caracteres.

La posición número N+1 es la del caracter "\0" o de fin de cadena (hay que tener en cuenta que la primera posición es la número 0).

Esta regla resultará muy util para evitar confusiones por el caracter "extra" que supone el caracter fin de cadena.

Por ejemplo, para el siguiente código:

```
char sMiCadena[4+1];
sprintf (sMiCadena, "hola");
```

la variable sMiCadena contendría lo siguiente:

carácter	h	o	l	a	\0
posición	0	1	2	3	4

4. Sentencias de control

4.1 Comparaciones

* Cuando se utilicen comparaciones entre una variable y una constante, hay que indicar como primer miembro de la comparación la constante.

Esto se hace así para prevenir posibles errores (muy difíciles de identificar) por indicar una asignación en vez de una comparación.

Así la siguiente sentencia sería poco correcta:

```
if (nDato == 10) { ... }
// si por error se pusiera if (nDato = 10) esto no daría un error de
// compilación, pero provoca malfuncionamiento (por una alteración
// del valor de nDato).
```

En cambio lo correcto sería realizarlo así:

```
if (10 == nDato) { ... }
// si por error se pusiera if (10 = nDato) daría un error de
// compilación
```

* Se aconseja no hacer comparaciones de igualdad con constantes numéricas de punto flotante.

Es mejor sustituir estas comparaciones de igualdad por desigualdades y utilizar los valores mínimos definidos para tipo `DBL_EPSILON` y `FLT_EPSILON`.

Por ejemplo, la siguiente sentencia:

```
// Comparación errónea de variable en punto flotante.
if (0.1 == fVariable) { ... }
```

Es mejor sustituirlo por

```
// Comparación correcta con una variable en punto flotantes.
if (FLT_EPSILON > fabs(fVariable-0.1)) { ... }
```

4.2 Asignaciones

* En las asignaciones se debe evitar la conversión explícita de tipos de datos. Se aconseja utilizar la conversión implícita de los datos siempre que sea posible.

* Se aconseja no hacer asignaciones múltiples.

Las asignaciones múltiples pueden dar lugar a actuaciones erróneas, se puede entener (de manera errónea) que ciertas variables son equivalentes (en cuyo caso sobrarían) y no da legibilidad al código.

Por ejemplo, el código siguiente:

```
// Asignación múltiple (desaconsejada)
nXSuperior = nYSuperior = (nAnchoPantalla – nAnchoVentana)/2;
```

Quedaría más sencillo y comprensible como:

```
// Varias asignaciones, mucho más sencillo,
nXSuperior = (nAnchoPantalla – nAnchoVentana)/2;
nYSuperior = nXSuperior;
```

* Se debe evitar el uso de expresiones de asignación dentro de expresiones condicionales, ya que dificulta la depuración y legibilidad del código.

Por ejemplo, el siguiente código:

```
// asignación en una comparación (desaconsejada)
if (0.5*PI <= (fAngulo=asin(fRadio))) {...}
```

Sería más correcto como sigue:

```
fAngulo = asin(fRadio);      // Primero asignación
if (0.5*PI <= fAngulo) {...} // Después la comparación
```

4.3 Operador ternario ?

* Es aconsejable utilizar sentencias if en vez del operador ternario ?.

Este operador ternario ?, propio de C, no presenta ventajas frente a la sentencia if, que es más conocida, de uso extendido y da lugar a un código más legible.

El siguiente código:

```
// Uso del operador ternario (desaconsejado)
nValorRetorno = (bStatusCorrecto ? hacerUnaCosa() : hacerOtra());
```

Quedaría más sencillo para su lectura como:

```
// Uso de sentencia if... else..., más correcta
if (bStatusCorrecto) nValorRetorno = hacerUnaCosa();
else nValorRetorno = hacerOtra();
```

4.4 Sentencia for

* La sentencia for se usará para definir un bucle en el que una variable se incrementa de manera constante en cada iteración y la finalización del bucle se determina mediante una expresión constante.

* Como contador for se utilizarán preferiblemente variables de un solo carácter como i, j, k... y se declarará (C++) este contador en el propio bucle for.
En el bucle del for no se modificará el valor de esta variable contador.
Por ejemplo:

```
for (int i=0; i<nMaximoVueltas; i++)
   {
     // Sentencias que NO modifican el valor de i
   }
```

4.5 Sentencias while y do while

* La sentencia while se usará para definir un bucle en el que la condición de terminación se evalúa al principio del bucle.

* La sentencia do...while se usará para definir un bucle en el que la condición de terminación se evaluará al final del bucle.

* Al comenzar un bucle while o do...while la expresión de control debe tener un valor claramente definido, para impedir posibles indeterminaciones o errores de funcionamiento

Un caso de posible indeterminación sería el siguiente:

```
int nVariableCount;
...
// código que no modifica nVariableCount
...
// No podríamos saber si se iniciará el bucle
while (MAXIMO_CONTADOR < nVariableCount) { ... }
```

En cambio sería correcto el siguiente código:

```
int nVariableCount;
...
// código que no modifica nVariableCount
...
nVariableCount = 0; // Inicializamos el contador del bucle
while (MAXIMO_CONTADOR < nVariableCount) { ... }
```

4.6 Sentencia Switch

* En las sentencias switch hay que incluir siempre la opción default y el break en todas las ramas.

Un esqueleto de una sentencia switch:

```
// Ejemplo de uso del switch
switch (nCondicion)
  {
```

```
    case 1:
      ...
      break;

    case 2:
      ...
      break;

    ...

    default:  // opción por defecto
      ...
  }
```

4.7 Sentencia goto

* En general NO USAR SENTENCIAS GOTO.

Las sentencias goto y otras sentencias de salto similares que rompen la estructura del código provocan una alteración no lineal en el flujo del programa. Atentan seriamente contra la integridad del código. Aparentemente pueden suponer una solución rápida a problemas puntuales, pero conllevan muchos posibles problemas colaterales, imprevisibles y de difícil determinación.

4.8 Sentencias break y continue

* Se aconseja minimizar el uso de las sentencias break y continue (o no usarlas), a favor de utilizar sentencias de tipo if … else.

Estas sentencias break y continue hacen más ilegible el código.

4.9 Sentencias exit y return

* La sentencia exit finaliza la ejecución de un proceso de manera inmediata, forzando la vuelta al sistema operativo.

Se desaconseja su utilización en cualquier parte del código, siendo preferible controlar el flujo en el proceso mediante bucles condicionales y devolver el control mediante la sentencia return.

* La sentencia return se utiliza para salir de una función o procedimiento, volviendo al punto en el cual se llamó a dicha función o procedimiento.

En el código hay que minimizar la utilización de return, sólo tendría que aparecer una vez en cada función o procedimiento, al final del mismo, de manera que se tenga un sólo punto de entrada a la función y un solo punto de salida de la misma.

4.10 Bloques

* Las sentencias if, else, for, while y do while tienen que estar seguidas siempre por un bloque, aunque este bloque esté vacío.

Por ejemplo, el siguiente código sería erróneo:

```
// while sin bloque, desaconsejable
while( /* Lo que sea */ );
```

Sería correcto ponerlo como sigue:

```
// While con bloque vacío, correcto
while( /* Lo que sea */ )
   {
     // vacio
   }
```

5. Funciones

5.1 Nomenclatura de las funciones

* Los nombres de las funciones tienen que ser auto explicativos. Tienen que dar una idea clara de cual es su finalidad.

* Los nombres de las funciones deberían empezar por una letra minúscula (por contra los métodos de las clases tienen que empezar por una letra mayúscula).
Así seguimos una nomenclatura ampliamente difundida y utilizada.

De manera que sería poco correcto llamar a una función:

```
// Definición poco correcta de una función
int RealizarCopia (char *psDestino, char *psOrigen);
```

Mientras que sería correcto llamarla:

```
// Definición correcta de una función
int realizarCopia (char *psDestino, char *psOrigen);
```

* Para componer los nombres de las funciones se aconseja utilizar formas verbales (infinitivos) pudiéndose acompañar (o no) con sustantivos para precisar la acción.
De esta manera mejoramos la identificación de los distintos componentes de un sistema.

Por ejemplo, una función que hace una carga de datos en un array si la llamamos `datos()` no queda clara su finalidad, en cambio si la llamamos `cargarDatosIniciales()` está muy claro su uso.

5.2 Utilización de funciones

* En general se aconseja limitar el máximo posible la visibilidad de las funciones. Como norma general de protección de datos.

5.3 Argumentos de las funciones

* Los primeros argumentos deberían ser los argumentos de salida (o destino) y los últimos argumentos los de entrada (u origen).
Así seguiríamos la forma de las funciones standard habituales, como memcpy, strcat, etc...
Además es raro que un argumento de salida sea un argumento opcional, normalmente los argumentos opcionales son los argumentos de entrada.

Una definición de ejemplo:

```
// Función de copia, pon un argumento origen (psOrigen) y
// otro destino (psDestino)
int realizarCopia (char *psDestino, char *psOrigen);
```

Esta regla también se aplicará a los métodos de las clases C++.

* Se debe comprobar la validez de los argumentos de entrada (tanto en las funciones como en los métodos) de la función siempre que puedan tomar valores anómalos o inadecuados.

De esta manera evitaríamos posibles errores de funcionamiento debido a valores inesperados de argumentos de entrada (por ejemplo si no han sido debidamente inicializados).

* Se deben especificar los nombres de los argumentos formales de las funciones (y los métodos de las clases C++) en la definición de las mismas.

Esta suele ser una practica habitual, que ayuda a la legibilidad del código y reduce el riesgo de que cambios en el código afecten a argumentos distintos de los que pensábamos.

Por ejemplo, el siguiente código:

```
// Definición poco correcta de miFuncion
void miFuncion (*pdDatoFinal, dDatoMinimo, dDatoMaximo, nIntervalo)
double *pdDatoFinal;
double dDatoMinimo;
double dDatoMaximo;
int nIntervalo;
{ ... }
```

Debería ser (para ser más correcto):

```
// Definición correcta de miFuncion.
void miFuncion (double *pdDatoFinal, double dDatoMinimo,
                double dDatoMaximo, int nIntervalo)
{ ... }
```

* Sería aconsejable que las funciones (a los métodos C++ les pasaría lo mismo) no tuvieran un número grande de argumentos (preferentemente menos de 7).

Cuantos más argumentos tenga una función es más probable que se pueda hacer un uso equivocado de la función, al igual que se incrementa la dificultad de la lectura.

* En general es aconsejable minimizar el uso de argumentos por defecto (tanto en las funciones como en los métodos de las clases C++), o no usarlos, en especial en las funciones sobrecargadas, para favorecer la comprensión del código.

En algunas ocasiones, al revisar el código se puede tener la duda de si algún argumento ha sido omitido por error o intencionadamente, este error se agrava en funciones sobrecargadas, pudiendo dar lugar a indeterminaciones.

* En la definición de las funciones se aconseja poner en líneas separadas los argumentos de la función, sobre todo si son muchos estos argumentos.

Por ejemplo:

```
// La siguiente definición de "miFuncion" sería más correcta que
// la anterior:
void miFuncion (double *pdDatoFinal,
                double dDatoMinimo,
                double dDatoMaximo,
                int nIntervalo)
{ ... }
```

5.4 Retorno de la función

* Muy habitualmente el retorno de las funciones es un indicador de estado.

Hay que acostumbrarse a comprobar siempre el estado devuelto por una función antes de continuar la ejecución normal del programa.

Incluso es deseable que exista un registro de condiciones anómalas o inesperadas que recojan posibles errores obtenidos mediante ese indicador de estado.

Esta regla también se aplicará tanto a las funciones como a los métodos de las clases C++.

* Cuando una función (o un método de una clase C++) devuelva un indicador de estado, este indicador debe tomar un valor de los posibles que se encontrarán en un fichero include designado a tal caso.

De esta manera se puede localizar claramente el motivo del error, si es que se produce alguno.

* Una función no devolverá nunca un puntero a una variable local. Ya que la memoria donde se guardan los calores las variables locales de las funciones es "desalocada" al salir de la función. El compilador puede o no avisar de este problema.

5.5 Referencias (C++)

* En general debería limitarse (tanto en las funciones como en los métodos) mucho el uso de referencias (no constantes).

Si deben usarse referencias constantes en vez de llamadas por valor.

Con referencias se economiza el uso de la pila (y mejor si son constantes, ya que su uso tiene la misma sintaxis que si fueran por valor), excepto para tipos predefinidos por el compilador (por que no se gana nada) y punteros (por que no tiene sentido).

Al usar referencias en una función (o en un método C++) se puede modificar el contenido del objeto referenciado, este caso que puede ser fuente de errores, se soluciona al pasar una referencia constante a la función.

Las siguientes serían definiciones erróneas:

```
void maximoValor (CDato const &rObjeto); // error de codificación
void maximoValor (CDato const objeto);   // pasa el objeto por valor
```

La forma correcta de definir esta función sería:

```
void maximoValor (const CDato &rObjeto);
// uso correcto de referencias
```

5.6 Sobrecarga de funciones (C++)

* La sobrecarga de funciones (y métodos) debe utilizarse con mucho cuidado y de forma uniforme.

Se recomienda limitar su uso a operaciones definidas en el ámbito del problema, para mejorar la claridad del código y su facilidad de comprensión.

* Las funciones (y métodos) sobrecargadas tienen que estar muy relacionadas entre sí. Sería absurdo que las funciones sobrecargadas tuvieran usos diferentes.

5.7 Funciones inline

* Solo se deberían utilizar funciones (y/o métodos) inline en las implementaciones (en el desarrollo de las funciones miembro específicas que constituyen la lógica de funcionamiento de la clase (C++)).

* Normalmente las funciones (y métodos) inline tendrían que declararse en un fichero aparte.

Esto se hace así para poder solucionar los problemas de algunos debugger con estas funciones. Declarándolas en un fichero aparte, podemos incluir este fichero bien en el fichero de cabecera, bien en el fichero de código, de acuerdo a nuestro interés.

* Es preferible usar funciones (o métodos) inline en lugar de macros creadas utilizando #define.

* Las funciones (y métodos) inline han de tener una sola línea de código.

5.8 Cabeceras de las funciones

* Las definiciones de todas las funciones tienen que incluir una cabecera descriptiva que indique los siguientes campos.

1. Nombre de la función.
2. Finalidad o uso de la función.
3. Argumentos de la función, indicando si son de entrada o salida, y la finalidad o uso de cada uno de ellos.
4. Retorno de la función. Indicando los posibles valores de retorno y el significado de los mismos.
5. Variables globales afectadas (si las hay).
6. Nombre del autor y fecha de última modificación.
7. Historial de moficiaciones, con fecha, motivo y nombre del autor.

6. Clases (C++)

6.1 Nomenclatura de las clases

* Los nombres de las clases tienen que ser auto explicativos. El nombre de una clase tiene que dar una idea clara de su uso o finalidad.

* Para componer los nombres de las clases se utilizarán principalmente sustantivos, pudiendo calificarse con adjetivos.
Esto se realizará así para mejorar la identificación de los distintos componentes de un sistema.

* El nombre de la clase debería empezar por una C mayúscula.
De esta manera se identificaría claramente su utilización.

Un ejemplo de definición de una clase:

```
class CMiClase { ... };
```

6.2 Secciones en la declaración de clases

* En la declaración de clases se deberían organizar las variables y funciones (métodos) que la componen en las siguientes secciones:

1. Constructores: Los constructores de la clase, además de los métodos que realicen tareas de inicialización, como, por ejemplo, Create().
2. Destructores: el destructor de la clase, así como los métodos que realicen tareas de finalización, como por ejemplo, Finalize().
3. Atributos: lugar donde se declaran las variables miembro y las funciones que las manipulan.
4. Operaciones: Funciones miembros (métodos) de la clase, que otras clases pueden llamar para realizar operaciones (son de acceso público o protegido). Si el número de operaciones es grande se puede subdividir en otros grupos (por ejemplo: operaciones - entramado, operaciones - funciones gráficas, ...)
5. Sobreescribibles: Funciones miembros (métodos) que pueden ser sobreescritas en otras clases. Es el lugar donde colocar las funciones miembro (métodos) virtual.
6. Implementación: Funciones miembro (métodos) específicas de esta clase que realizan operaciones internas, que constituyen la lógica de funcionamiento de la clase.

El motivo de esta separación no es otro que seguir convenciones utilizadas ampliamente.

6.3 Datos miembros de clase

* Los datos miembros de una clase tienen que tener un nombre que sea auto explicativo de su utilidad. Como sucedía con las variables.

* Los nombres de los datos miembros de una clase empezarán por un prefijo "m_" que permitirá identificarlos claramente como datos miembro.

* Para componer los nombres de los datos miembros de una clase se seguirán criterios similares a las variables. Con la diferencia del prefijo "m_" que permite identificar a las variables miembro.

Ejemplo:

```
Class CMiClase
  {
    private:
    int m_nNumeroEntradas;
    double m_dValorMaximo;
    char * m_psNombreIdentificador;
    ...
  }
```

* Los datos miembros de una clase tienen que ser privados (private), no públicos (public) ni protegidos (protected)
De esta manera aseguramos la encapsulación de la clase.

* Se recomienda el uso de métodos sencillos (preferiblemente métodos inline) para poder obtener el valor de las variables miembros interesantes. Es aconsejable que estos métodos se declaren como constantes.
Por ejemplo:

```
Class CMiClase
  {
    public:
    int GetNumeroEntradas () const;
    ...

    private:
    int m_nNumeroEntradas;
    ...
  }
```

* Se recomienda lo siguiente: "no usar datos globales" y "no usar datos miembros públicos" según dice Stroustrup, creador de C++ [1].

6.4 Métodos (funciones miembro) de clase

* Los nombres de las funciones miembro (métodos) de una clase tiene que ser auto explicativos de la finalidad de dichos métodos. Como ocurría con las funciones.

* Los nombres de los métodos de una clase deberían empezar por mayúscula.

Así damos mayor claridad al código, diferenciando el uso de un método de la clase del uso de una función global.

Un ejemplo de definición de un método miembro de clase:

```
int CMiClase::MiFuncion(int nNumero) {...}
```

* Los nombres de las métodos de una clase seguirán una nomenclatura similar a las funciones normales de C (salvo por la primera letra en mayúscula).
Así para componer los nombres de las funciones miembro (métodos) se usarán formas verbales (infinitivos) acompañadas de sustantivos que maticen su acción.

* Deben evitarse la utilización de las funciones (métodos) friend, para asegurar la encapsulación de la clase.

* Las funciones miembros (métodos) de una clase que no modifiquen los datos miembros (ni llamen a funciones que modifiquen los datos miembros) de la misma deberían declararse como const.
De otra forma los objetos creados como constantes no tendrían acceso a las funciones miembro de la clase.

* Se recomienda lo siguiente: "no usar funciones (no miembros) globales" según dice Stroustrup, creador de C++ [1].

* En general se aconseja limitar el máximo posible la visibilidad de las funciones miembro. Como norma general de protección de datos.

* Es aconsejable que todas las clases tengan un constructor de copia, cuando el constructor de copia por defecto no sea satisfactorio (por ejemplo cuando haya variables miembro que sean punteros).

* Si una clase es usada como clase base y tiene funciones virtuales, es necesario indicarle un destructor virtual, para evitar "sorpresas" en el uso de instancias.

* En caso de que una función miembro devuelva una referencia o un puntero a un dato miembro, tiene que devolver una referencia constante o un puntero constante (para evitar modificaciones o alteraciones de este dato miembro desde fuera de la clase).

6.5 Sobrecarga de operadores

* La sobrecarga de operadores debe utilizarse con mucho cuidado y de forma uniforme.
Se recomienda limitar su uso a operaciones definidas en el ámbito del problema, ya que los operadores son descriptivos de operaciones ya conocidas.

* Los operadores sobrecargados deben mantener el significado de la operación que realizan.
No sería lógico definir, por ejemplo, un operador + sobrecargado que no tuviera el significado del original (la suma).

* En el caso de que haya dos operadores opuestos (como es el caso de = y !=) es aconsejable sobrecargar los dos, aunque solamente se quiera sobrecargar uno de ellos.

6.6 Instancias

* Cuando se crea un objeto instancia de una clase con parámetros, se debe utilizar una llamada al constructor de la clase con esos argumentos.

De esta forma evitamos malfuncionamientos del constructor y una posible copia adicional del objeto de la clase.

Por ejemplo, estaría mal indicado lo siguiente:

```
CMiClase miObjeto = 3; // ¿nuevo objeto o objeto copia?
```

En cambio sería correcto instanciar el objeto de la siguiente manera:

```
CMiClase miObjeto(3);
// Creación correcta de una instancia de una clase con parámetros.
```

Esta norma no implica que no se definan y empleen constructores de copia.

* En general se aconseja limitar el máximo posible la visibilidad de las instancias. Como norma general de protección de datos.

6.7 Cabeceras de las clases y los métodos

* Todas las clases tienen que incluir una cabecera descriptiva que indique los siguientes campos.

1. Nombre de la clase.
2. Finalidad o uso de la clase.
3. Variables globales afectadas (si las hay).
4. Nombre del autor y fecha de última modificación.
5. Historial de modificaciones, con fecha, motivo y nombre del autor.

* Las definiciones de todos los métodos tienen que incluir una cabecera descriptiva que indique los siguientes campos.

1. Nombre del método.
2. Nombre de la clase a la que pertenece.
3. Visibilidad dentro de la clase (público, privado, protegido).
4. Finalidad o uso del método.
5. Argumentos del método. Indicando si son de entrada o de salida y la finalidad de cada uno de ellos.
6. Variables globales afectadas (si las hay).
7. Variables miembros modificadas en esta clase.

8. Nombre del autor y fecha de última modificación.
9. Historial de modificaciones, con fecha, motivo y nombre del autor.

6.8 Objetos

* Los nombres de los objetos tienen que empezar por una letra minúscula.
Los punteros a objetos tienen que empezar por el prefijo p.
Por ejemplo:

```
CMiClase  unObjeto(3);  // Declaración de un objeto
CMiClase *pOtroObjeto;  // Declaración de un puntero a un objeto
```

* Nunca convertir punteros a objeto de una clase derivada a punteros a objeto de una clase base virtual.

7. Defines, macros y constantes

7.1 Nomenclatura de defines, macros, constantes

* Los nombres de defines, macros, constantes, etc... tienen que ser auto explicativos. En sí mismos tienen que contener una explicación de su uso o finalidad.

* Para componer los nombres de defines, constantes, etc.. se utilizarán principalmente sustantivos, pudiendo calificarse con adjetivos.

Para las macros, se utilizarán formas verbales (infinitivos), acompañados opcionalmente de sustantivos que califiquen la acción.

Esto se realizará así para mejorar la identificación de los distintos componentes de un sistema.

* Las constantes se tienen que definir en mayúsculas (tanto las definidas por #define como las definidas utilizando const).

Si se componen de varias palabras se deberían separan por el carácter "_".

La constante se conformará mediante un sustantivo (o conjunto de sustantivos) con posibles adjetivos.

De esta forma seguimos una norma ampliamente utilizada por los programadores.

Ejemplos de definición de constantes:

```
#define PI                  3.14
#define DIAS_DE_LA_SEMANA   7
```

7.2 Utilización de defines, macros, constantes

* Se deben utilizar constantes (const) o enumeraciones (enum) para definir constantes, evitando el uso de constantes con #define en lo posible.

#define es una directiva para el preprocesador, la cual se aplica con carácter global antes de hacer ninguna comprobación por parte del compilador. Sus problemas son los consabidos efectos colaterales.

Por ejemplo sería desaconsejable utilizar lo siguiente:

```
// Definimos (desaconsejado) valores para datos y constantes
// con #define
typedef int BOOLE  // el tipo de dato para los resultados
#define FALSO  0   // definición de resultado falso
#define CIERTO 1   // definición de resultado cierto

// El numero de días de una semana
#define DIAS_DE_LA_SEMANA 7
```

Sería más correcto utilizar lo siguiente:

```
enum EBoole // Definimos el tipo de dato EBoole para los resultados
  {
    FALSO,  // 0 por defecto
    CIERTO  // Verdad = 1
  };

// La constante numero de días de la semana
const int nDIAS_SEMANA = 7;
```

* Se aconseja el uso de defines para sustituir el uso de valores numéricos en el código.

* Es preferible usar funciones inline en lugar de macros creadas utilizando #define.
Ya que #define es una directiva de precompilación y puede provocar efectos colaterales.

* En general se aconseja limitar el máximo posible la visibilidad de defines, macros, constantes... Como norma general de protección de datos.

8. Excepciones (C++)

8.1 Utilización de excepciones

* Hay que asegurarse de que se capturan las excepciones cuando se transfieren al manejador de excepciones de la forma más sencilla posible.

* Comprobar los códigos de error de las excepciones que pueden ser recibidas desde funciones de librería, aún cuando estas funciones parezcan sencillas y que no fallan.

9. Tipos de datos y enumeraciones

9.1 Nomenclatura para tipos de datos y enumeraciones

* Los nombres de los tipos de datos y enumeraciones tienen que ser auto explicativos, descriptivos de su finalidad.

* Para componer los nombres de los tipos de datos se utilizarán principalmente sustantivos, pudiendo calificarse con adjetivos.
Esto se realizará así para mejorar la identificación de los distintos componentes de un sistema.

* Los nombres de los tipos de datos se identifican con un prefijo T. Y el nombre se compone en minúsculas, pero con la primera letra de cada palabra en mayúsculas, sin separadores entre palabras.
En vez del prefijo T se podría indicar un prefijo Tipo (o un sufijo Type si codificamos en inglés).
Esta regla permite una clara identificación de las definiciones de tipos de datos.
Por ejemplo, un tipo de dato para un flag carácter:

```
typedef char TFlagFunc;
```

* Las nombres de las enumeraciones se pueden identificar con un prefijo E. Y el nombre se compone en minúsculas, pero con la primera letra de cada palabra en mayúsculas, sin separadores entre palabras.
En vez del prefijo E se podría indicar un prefijo Enum.
Esta es otra regla pensada para la identificación clara de las enumeraciones frente a otros tipos de datos.
Por ejemplo, se podría definir la siguiente enumeración para las notas de un examen:

```
enum ENotaExamen {SUSPENSO, APROBADO, NOTABLE, SOBRESALIENTE};
```

9.2 Utilización de tipos de datos y enumeraciones

*Es aconsejable usar tipos definidos por el usuario en vez de tipos predefinidos.
De esta manera mejoraremos en la claridad y comprensión del código.

Por ejemplo, el siguiente tipo EBoole es más descriptivo que el tipo int como retorno de funciones:

```
enum EBoole {FALSO, CIERTO};
```

*No deben utilizarse tipos anónimos, ya que hacen el código ilegible y poco comprensivo.

Por ejemplo, en vez de...

```
// Uso (desaconsejado) de tipos anónimos de datos.
enum {SUSPENSO, APROBADO, NOTABLE, SOBRESALIENTE} eNotaAlumno;
```

Estaría mejor definir...

```
// Definimos un tipo de dato ENotaExamen.
enum ENotaExamen {SUSPENSO, APROBADO, NOTABLE, SOBRESALIENTE};

// Definimos la variable para la nota
ENotaExamen eNotaAlumno = APROBADO;
```

*Se aconseja utilizar tipos de datos definidos con typedef para simplificar la síntaxis cuando se utilicen punteros a funciones.

Por ejemplo el siguiente caso:

```
#include <math.h>
// Normalmente una forma de declarar punteros a funciones es
// la siguiente.
// double ( *mathFunc ) ( double ) = sqrt;
// Utilizando typedef simplificamos la síntaxis
typedef double TMathFunc ( double );

TMathFunc *mathFunc = sqrt;
void main ()
{
    // Podemos invocar la función de dos formas:

    // 1. Forma sencilla
    double dValue1 = mathFunc( 23.0 );

    // 2. Forma complicada, desaconsejada
    double dValue2 = ( *mathFunc )( 23.0 );
}
```

* Es preferible usar enumeraciones en vez de constantes #define para variables que pueden tomar valores en un rango muy limitado y conocido.

9.3 Conversión de tipos de datos

* Se deben evitar la conversión explícita de tipos de datos, ya que atenta contra la comprobación de de tipos de datos de los compiladores C y C++. Pudiéndose tener problemas de ejecución.

Lo aconsejable es utilizar implícitamente la conversión de tipos de datos, siempre que esto sea posible.

* Nunca realizar la conversión de tipos const a tipos no-const. Esta práctica errónea puede dar errores por que obliga al compilador a alojar constantes en la zona de las variables.

10. Ficheros

10.1 Tipos de ficheros

* Los ficheros de cabecera del proyecto se llamarán fichero.h (si trabajamos en C) o fichero.hh (si trabajamos en C++) (también sería correcto el uso de fichero.hpp para C++)

* Los ficheros con el código de las funciones no inline se llamará fichero.c (si trabajamos en C) o fichero.cc (si trabajamos en C++) (también sería correcto el uso de fichero.cpp para C++)

* Los ficheros con el código de las funciones inline se llamará fichero.ic (si trabajamos en C) o fichero.icc (si trabajamos en C++) (también sería correcto el uso de fichero.ipp)

El separar las funciones inline (funciones definidas en una sola línea) de las demás funciones se debe a que algunos debuggers (y más raramente algunos compiladores) no manejan bien las funciones inline.

Separando las funciones inline en un fichero aparte, podemos conseguir que se incluyan en los ficheros de cabecera o en los ficheros fuente, según nosotros deseemos de acuerdo a directivas de compilación.

A continuación se muestra en un ejemplo como se haría...

```
// A incluir en fichero.hh
#if !defined(DEBUG_MODE)
#include <fichero.icc>
#endif

// A incluir en fichero.cc
#if defined(DEBUG_MODE)
#include <fichero.icc>
#endif
```

Con este ejemplo conseguimos que si se compila en modo debug (definiendo DEBUG_MODE) las funciones inline (funcion.icc) se incluirán en el fichero.hh, mientras que en el modo normal se incluirían en el fichero.cc

10.2 Nombres de los ficheros

* Para C en cada fichero de una librería tiene que haber funciones relacionadas con el mismo fin, y el nombre sería el apropiado para poder identificar el fin de estas funciones (por ejemplo funciones Fecha.c para recopilar funciones de manejo de fechas).

* Para C++ en cada fichero de una librería tiene que haber solamente una clase, y el nombre del fichero sería el mismo que el de la clase (por ejemplo CMiClase.cc para la implementación de la clase CMiClase).

10.3 Estructura de los ficheros

* Los ficheros de cabecera (fichero.h, fichero.hh, fichero.hpp) tendrían que tener una estructura como la siguiente:

1. Cabecera del fichero.
2. Includes si es necesario alguno (se aconseja que en los ficheros de cabecera no vaya ningún fichero include o el mínimo número imprescindible de ellos).
   - o Primero los includes del sistema (si hace falta alguno)
   - o Después los includes propios del proyecto (si hace falta alguno)
3. Constantes simbólicas y definiciones de macros que se vayan a utilizar en otros módulos que incluyan este.
4. Definición de tipos que se vayan a utilizar en otros módulos que incluyan este.
5. Prototipos de las funciones del módulo que puedan ser utilizadas desde otro módulo.
6. Declaración de las clases (C++) del módulo que puedan ser utilizadas desde otro módulo.

* Los ficheros fuente (fichero.c, fichero.cc, fichero.cpp) tendrían que tener una estructura como la siguiente:

1. Cabecera del fichero.
2. Includes necesarios para el módulo.
   - o Primero los includes del sistema.
   - o Después los includes propios del proyecto.
3. Constantes simbólicas y dfiniciones de macros que solamente vayan a utilizarse en este módulo.
4. Definición de tipos que se vayan a utilizar solamente en este módulo.
5. Variables globales usadas en el módulo.
   - o Primero las declaradas en otros módulos distintos, con la palabra reservada extern.
   - o Después las declaradas en este módulo.
6. Prototipos de las funciones del módulo que sólo se usen en este módulo.
7. Declaración de las clases (C++) del módulo que sólo se usen en este módulo.
8. Implementación de las funciones del módulo.
   - La función principal (main()) tiene que ser la primera, las demás estarán ordenadas por orden de aparición en la función principal o por orden de llamada, u organizándolas por su uso o finalidad.
   - Es aconsejable que las implementaciones estén en el mismo orden que sus prototipos.
9. Implementación de los métodos de las clases (C++).
   - Es aconsejable que las implementaciones de los métodos estén en el mismo orden en que aparecen en las declaraciones de las clases.

* Los ficheros que contienen las funciones inline (fichero.ic, fichero.icc, fichero.ipp) solamente contendrán el código de las funciones inline y tendrán una estructura como la siguiente:

1. Cabecera del fichero
2. Código de las funciones inline.
   Se aconseja que estén en el mismo orden en que se declaran en el fichero de cabecera.

10.4 Cabeceras de los ficheros

* Todos los ficheros tienen que incluir una cabecera que indique los siguientes campos.

1. Nombre del fichero
2. Finalidad o uso del módulo (incluyendo los argumentos si en este fichero se encuentra la función main).
3. Variables globales afectadas (si las hay).
4. Nombre del autor y fecha de última modificación.
5. Historial de modificaciones, con fecha, motivo y nombre del autor.

10.5 Ficheros de cabecera (*.h)

* Todos los ficheros de cabecera tienen que tener un mecanismo para impedir que sean incluidos más de una vez (lo mismo les pasaría a los ficheros de las funciones inline).

Por ejemplo, el siguiente método valdría:

```
#ifndef __FICHERO_H__
   #define __FICHERO_H__
   // __FICHERO_H__ sería un identificador propio de este
   // fichero (fichero.h)
   // ya que lo he construido con el nombre del fichero

   ...
   // Aquí iría todo el contenido del fichero
   ...

#endif
```

* Todos los ficheros de cabecera incluidos en los ficheros deben serlo exclusivamente por que se usan.

Hay que evitar la inclusión de ficheros de cabecera que no se usan, por legibilidad, puede complicar la compilación y el linkado innecesariamente.

* Los includes de librerías propios del sistema se realiza indicando el nombre entre caracteres < y >. Por ejemplo:

```
#include <stdio.h>
```

Los includes propios del proyecto se realizarán indicando el nombre entre caracteres ". Por ejemplo:

```
#include "CMiClase.hh.h"
```

A. Bibliografía

* [1] The C++ Programming Language, Bjarne Stroustrup, ed. Addison-Wesley

* [2] Hungarian Notation (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsgen/html/hunganotat.asp), Charles Simonyi, Microsoft Corporation.

* [3] Normas de codificación C++ (http://www.procuno.com/fci-stde/Archivo/Normas%20codificaci%C3%B3n%20V14.rtf), Fernando López Murube, Procedimientos-Uno.

* [4] Normas de estilo para programación en lenguaje C (http://polaris.lcc.uma.es/docencia/ITInd/finformatica/normasC.html), Escuela Universitaria Politecnica, Universidad de Málaga.

* [5] C++ Style Guide, Ellemtel Telecommunication Systems Laboratories

* [6] Programar con estilo (http://idam.ladei.com.ar/articulo.php?archivo=ProgramarConEstilo&nombre=Programar%2Bcon%2Bestilo&tipo=simple), Fernando José Serrano García, IDAM

* [7] C++ Programming Style, Tom Cargill, ed. Addison Wesley Professional.

* [8] Normas de estilo en C++ (http://geneura.ugr.es/~jmerelo/c++-avanzado.htm), J. J. Merelo, GeNeura

* [9] C++ Programming Style (http://www.spelman.edu/~anderson/teaching/resources/style/), Scott D. Anderson

* [10] C# Codign Standard, Juval Lowy, www.idesign.net

* [11] C++ coding standards (http://www.catalase.com/codestd.htm), James Crook

* [12] General C++ Codign Standard at the NFRA (http://www.nfra.nl/~seg/cppStdDoc.html), Ger van Diepen, NFRA

ANEXOS

Traducción obtenida de la página de traducciones de licencias GNU (http://gugs.sindominio.net/gnu-gpl/).
(Ver el original (http://gugs.sindominio.net/gnu-gpl/fdl-es.html) del Grupo de Usuarios de Sindominio, ver el texto original (en inglés) de GNU (http://www.gnu.org/copyleft/fdl.html)).

Licencia de Documentación Libre GNU (traducción)

Versión 1.1, Marzo de 2000

Esta es la GNU Free Document License (GFDL), versión 1.1 (de marzo de 2.000), que cubre manuales y documentación para el software de la Free Software Foundation, con posibilidades en otros campos. La traducción[1] no tiene ningún valor legal, ni ha sido comprobada de acuerdo a la legislación de ningún país en particular. Vea el original (http://www.gnu.org/copyleft/fdl.html)

Los autores de esta traducción son:

* Igor Támara <ikks@bigfoot.com>

* Pablo Reyes <reyes_pablo@hotmail.com>

* Revisión : Vladimir Támara P. <vtamara@gnu.org>

0. Preámbulo

El propósito de esta licencia es permitir que un manual, libro de texto, u otro documento escrito sea "libre" en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera comercial o no. En segundo término, esta licencia preserva para el autor o para quien publica una manera de obtener reconocimiento por su trabajo, al tiempo que no se consideran responsables de las modificaciones realizadas por terceros.

Esta licencia es una especie de "copyleft" que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Esto complementa la Licencia Pública General GNU, que es una licencia de copyleft diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: Un programa libre debe venir con los manuales que ofrezcan las mismas libertades que da el software. Pero esta licencia no se limita a manuales de software; puede ser usada para cualquier trabajo textual, sin tener

en cuenta su temática o si se publica como libro impreso. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

1. Aplicabilidad y definiciones

Esta Licencia se aplica a cualquier manual u otro documento que contenga una nota del propietario de los derechos que indique que puede ser distribuido bajo los términos de la Licencia. El "Documento", en adelante, se refiere a cualquiera de dichos manuales o trabajos. Cualquier miembro del público es un licenciatario, y será denominado como "Usted".

Una "Versión Modificada" del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones y/o traducciones a otro idioma.

Una "Sección Secundaria" es un apéndice titulado o una sección preliminar al prólogo del Documento que tiene que ver exclusivamente con la relación de quien publica o, los autores del Documento o, el tema general del Documento(o asuntos relacionados) y cuyo contenido no entra directamente en este tema general. (Por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar matemáticas.) La relación puede ser un asunto de conexión histórica, o de posición legal, comercial, filosófica, ética o política con el tema o la materia del texto.

Las "Secciones Invariantes" son ciertas Secciones Secundarias cuyos títulos son denominados como Secciones Invariantes, en la nota que indica que el documento es liberado bajo esta licencia.

Los "Textos de Cubierta" son ciertos pasajes cortos de texto que se listan, como Textos de Portada o Textos de Contra Portada, en la nota que indica que el documento es liberado bajo esta Licencia.

Una copia "Transparente" del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público general, cuyos contenidos pueden ser vistos y editados directamente con editores de texto genéricos o (para imágenees compuestas por pixeles) de programas genéricos de dibujo o (para dibujos) algún editor gráfico ampliamente disponible, y que sea adecuado para exportar a formateadores de texto o para traducción automática a una variedad de formatos adecuados para ingresar a formateadores de texto. Una copia hecha en un formato de un archivo que no sea Transparente, cuyo formato ha sido diseñado para impedir o dificultar subsecuentes modificaciones posteriores por parte de los lectores no es Transparente. Una copia que no es "Transparente" es llamada "Opaca".

Como ejemplos de formatos adecuados para copias Transparentes están el ASCII plano sin formato, formato de Texinfo, formato de LaTeX, SGML o XML usando un DTD disponible ampliamente, y HTML simple que sigue los estándares, diseñado para modificaciones humanas. Los formatos Opacos incluyen PostScript, PDF, formatos propietarios que pueden ser leídos y editados unicamente en procesadores de palabras propietarios, SGML o XML para los cuáles los DTD y/o herramientas de procesamiento

no están disponibles generalmente, y el HTML generado por máquinas producto de algún procesador de palabras solo para propósitos de salida.

La "Portada" en un libro impreso significa, la portada misma, más las páginas siguientes necesarias para mantener la legibilidad del material, que esta Licencia requiere que aparezca en la portada. Para trabajos en formatos que no tienen Portada como tal, "Portada" significa el texto cerca a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del trabajo.

2. Copia literal

Puede copiar y distribuir el Documento en cualquier medio, sea en forma comercial o no, siempre y cuando esta Licencia, las notas de derecho de autor, y la nota de licencia que indica que esta Licencia se aplica al Documento se reproduzca en todas las copias, y que usted no adicione ninguna otra condición a las expuestas en en esta Licencia. No puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

También puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

3. Copiado en cantidades

Si publica copias impresas del Documento que sobrepasen las 100, y la nota de Licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible, todos esos textos de Cubierta: Textos Frontales en la cubierta frontal, y Textos Posteriores de Cubierta en la Cubierta Posterior. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como quien publica tales copias. La Cubierta Frontal debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede adicionar otro material en la cubierta. Las copias con cambios limitados en las cubiertas, siempre que preserven el título del Documento y satisfagan estas condiciones, puede considerarse como copia literal.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la cubierta real, y continuar el resto en páginas adyacentes.

Si publica o distribuye copias Opacas del Documento cuya cantidad exceda las 100, debe incluir una copia Transparente que pueda ser leída por una máquina con cada copia Opaca, o entregar en o con cada copia Opaca una dirección en red de computador públicamente-accesible conteniendo una copia completa Transparente del Documento, sin material adicional, a la cual el público en general de la red pueda acceder a bajar anónimamente sin cargo usando protocolos de standard público. Si usted hace uso de la última opción, deberá tomar medidas necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio por lo menos un año después de su última distribución de copias

Opacas (directamente o a través de sus agentes o distribuidores) de esa edición al público.

Se solicita, aunque no es requisito, que contacte a los autores del Documento antes de redistribuir cualquier gran número de copias, para permitirle la oportunidad de que le provean una versión del Documento.

4. Modificaciones

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto licenciando la distribución y modificación de la Versión Modificada a quienquiera que posea una copia de este. En adición, debe hacer lo siguiente en la Versión Modificada:

* A. Uso en la Portada (y en las cubiertas, si hay alguna) de un título distinto al del Documento, y de versiones anteriores (que deberían, si hay alguna, estar listados en la sección de Historia del Documento). Puede usar el mismo título que versiones anteriores al original siempre que quién publicó la primera versión lo permita.

* B. Listar en la Portada, como autores, una o más personas o entidades responsables por la autoría o las modificaciones en la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (Todos sus autores principales, si hay menos de cinco).

* C. Estado en la Portada del nombre de quién publica la Versión Modificada, como quien publica.

* D. Preservar todas las notas de derechos de autor del Documento.

* E. Adicionar una nota de derecho de autor apropiada a sus modificaciones adyacentes a las otras notas de derecho de autor.

* F. Incluir, inmediatamente después de la nota de derecho de autor, una nota de licencia dando el permiso público para usar la Versión Modificada bajo los términos de esta Licencia, de la forma mostrada en la Adición (LEGAL)abajo.

* G. Preservar en esa nota de licencia el listado completo de Secciones Invariantes y en los Textos de las Cubiertas que sean requeridos como se especifique en la nota de Licencia del Documento

* H. Incluir una copia sin modificación de esta Licencia.

* I. Preservar la sección llamada "Historia", y su título, y adicionar a esta una sección estableciendo al menos el título, el año, los nuevos autores, y quién publicó la Versión Modificada como reza en la Portada. Si no hay una sección titulada "Historia" en el Documento, crear una estableciendo el título, el año, los autores y quien publicó el Documento como reza en la Portada, añadiendo además un artículo describiendo la Versión Modificada como se estableció en el punto anterior.

* J. Preservar la localización en red, si hay , dada en la Documentación para acceder públicamente a una copia Transparente del Documento, tanto como las otras direcciones de red dadas en el Documento para versiones anteriores en las cuáles estuviese basado. Estas pueden ubicarse en la sección "Historia". Se puede omitir la ubicación en red para un trabajo que sea publicado por lo menos 4 años antes que el mismo Documento, o si quien publica originalmente la versión da permiso explícitamente.

* K. En cualquier sección titulada "Agradecimientos" o "Dedicatorias", preservar el título de la sección, y preservar en la sección toda la sustancia y el tono de los agradecimientos y/o dedicatorias de cada contribuyente que estén incluidas.

* L. Preservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección. M. Borrar cualquier sección titulada "Aprobaciones". Tales secciones no pueden estar incluidas en las Versiones Modificadas.

* M. Borrar cualquier sección titulada "Aprobaciones". Tales secciones no pueden estar incluidas en las Versiones Modificadas.

* N. No retitular ninguna sección existente como "Aprobaciones" o conflictuar con título con alguna Sección Invariante.

Si la Versión Modificada incluye secciones o apéndices nuevos o preliminares al prólogo que califican como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, adicione sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede adicionar una sección titulada "Aprobaciones", siempre que contenga únicamente aprobaciones de su Versión Modificada por varias fuentes--por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como un standard.

Puede adicionar un pasaje de hasta cinco palabras como un Texto de Cubierta Frontal, y un pasaje de hasta 25 palabras como un texto de Cubierta Posterior, al final de la lista de Textos de Cubierta en la Versión Modificada. Solamente un pasaje de Texto de Cubierta Frontal y un Texto de Cubierta Posterior puede ser adicionado por (o a manera de arreglos hechos por) una entidad. Si el Documento ya incluye un texto de cubierta para la misma cubierta, previamente adicionado por usted o por arreglo hecho por la misma entidad, a nombre de la cual está actuando, no puede adicionar otra; pero puede reemplazar la anterior, con permiso explícito de quien publicó anteriormente tal cubierta.

El(los) autor(es) y quien(es) publica(n) el Documento no dan con esta Licencia permiso para usar sus nombres para publicidad o para asegurar o implicar aprobación de cualquier Versión Modificada.

5. Combinando documentos

Puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, y listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y múltiples Secciones Invariantes Idénticas pueden ser reemplazadas por una sola copia. Si hay múltiples Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único adicionándole al final de este, en paréntesis, el nombre del autor o de quien publicó originalmente esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes en la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección titulada "Historia" de los varios documentos originales, formando una sección titulada "Historia"; de la misma forma combine cualquier sección titulada "Agradecimientos", y cualquier sección titulada "Dedicatorias". Debe borrar todas las secciones tituladas "Aprobaciones."

6. Colecciones de documentos

Puede hacer una colección consistente del Documento y otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en los varios documentos con una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para una copia literal de cada uno de los documentos en cualquiera de todos los aspectos.

Puede extraer un solo documento de una de tales colecciones, y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraído, y siga esta Licencia en todos los otros aspectos concernientes a la copia literal de tal documento.

7. Agregación con trabajos independientes

Una recopilación del Documento o de sus derivados con otros documentos o trabajos separados o independientes, en cualquier tipo de distribución o medio de almacenamiento, no como un todo, cuenta como una Versión Modificada del Documento, teniendo en cuenta que ninguna compilación de derechos de autor sea clamada por la recopilación. Tal recopilación es llamada un "agregado", y esta Licencia no aplica a los otros trabajos auto-contenidos y por lo tanto compilados con el Documento, o a cuenta de haber sido compilados, si no son ellos mismos trabajos derivados del Documento.

Si el requerimiento de la sección 3 del Texto de la Cubierta es aplicable a estas copias del Documento, entonces si el Documento es menor que un cuarto del agregado entero, Los Textos de la Cubierta del Documento pueden ser colocados en cubiertas que

enmarquen solamente el Documento entre el agregado. De otra forma deben aparecer en cubiertas enmarcando todo el agregado.

8. Traducción

La Traducción es considerada como una clase de modificación, Así que puede distribuir traducciones del Documento bajo los términos de la sección 4. Reemplazar las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero puede incluir traducciones de algunas o todas las Secciones Invariantes adicionalmente a las versiones originales de las Secciones Invariantes. Puede incluir una traducción de esta Licencia siempre que incluya también la versión Inglesa de esta Licencia. En caso de un desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la versión original en Inglés prevalecerá.

9. Terminación

No se puede copiar, modificar, sublicenciar, o distribuir el Documento excepto por lo permitido expresamente bajo esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y serán automáticamente terminados sus derechos bajo esa licencia. De todas maneras, los terceros que hayan recibido copias, o derechos, de su parte bajo esta Licencia no tendrán por terminadas sus licencias siempre que tales personas o entidades se encuentren en total conformidad con la licencia original.

10. Futuras revisiones de esta licencia

La Free Software Foundation puede publicar nuevas, revisadas versiones de la Licencia de Documentación Libre GNU de tiempo en tiempo. Tales nuevas versiones serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar problemas o intereses. Vea http://www.gnu.org/copyleft/.

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que una versión numerada particularmente de esta licencia o "cualquier versión posterior" se aplica a esta, tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que ha sido publicada (no como un borrador)por la Free Software Foundation. Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada(no como un borrador) por la Free Software Foundation.

Addendum

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente derecho de autor y nota de licencia justo después del título de la página:

Derecho de Autor © Año Su Nombre.

Notas

[1] N. del T. Derechos Reservados en el sentido de GNU
http://www.gnu.org/copyleft/copyleft.es.html

MANUAL DE ESTILO C / C++

http://www.geocities.com/webmeyer/prog/estilocpp/

Oscar Valtueña García

oscarvalt@gmail.com