Universidad Miguel Hernández
ESCUELA POLITÉCNICA SUPERIOR DE ELCHE
INGENIERIA DE SISTEMAS Y AUTOMATICA

Estándar de Normalización y Documentación del Software

Estándar de programación en C / C++

OSCAR REINOSO GARCÍA, LUIS MIGUEL JIMÉNEZ GARCÍA

Resumen: En este documento se recogen unas guías para la programación y documentación de proyectos software para uso tanto en proyectos de investigación como en proyectos docentes que así lo requieran. Este documento tiene carácter interno al área de *Ingeniería de Sistemas y Automática*, y únicamente pretende ser una normalización de los trabajos efectuados dentro de este área de conocimiento.

Palabras Clave: Documentación, Normalización, Estándar de Programación, C, C++.

25 Septiembre 1998

ISA-UMH © R-98-001v2.0

EPSE ♦ Avda. Ferrocarril s/n ♦ E-03202 Elche Alicante ♦ ESPAÑA ♦ Tel: + 34 (6) 665 86 55 ♦ Fax: + 34 (6) 665 87 98 ♦ http://www.umh.es

ESTA PÁGINA HA SIDO DEJADA EN BLANCO INTENCIONADAMENTE

Indice de Contenidos

ESTA PÁGINA HA SIDO DEJADA EN BLANCO INTENCIONADAMENTE

1 INTRODUCCIÓN

En este documento se presenta una propuesta de estándar de escritura de programas tanto en lenguaje C como C++. El objetivo es facilitar una documentación uniforme de manera que se posibilite el desarrollo en grupos (dirigidos por un director de proyecto o programador principal) de una manera sistemática. El mantenimiento de un software uniforme permite reutilizar código por otras personas integrantes del mismo proyecto. Otro de los objetivos fundamentales es facilitar y promover una documentación estándar final del proyecto. Sin duda el mantenimiento de los programas se ve facilitado al hacer uso de un código totalmente uniforme. Para ello se establecen una serie de reglas que favorecen la claridad del código. El presente documento abarca exclusivamente la etapa de desarrollo (posterior al diseño) incluyendo la organización de los archivos de código, la sintaxis, la nomenclatura y la documentación interna de los programas.

El documento pretende ser una base para el desarrollo de cualquier programa escrito en lenguaje C/C++, estando sujeto a cualquier tipo de modificación, ampliación, mejora, etc. A lo largo del documento se hará uso del término proyecto para referirse a un programa o aplicación completa específica. A su vez el programa o aplicación podrá estar dividido en diferentes módulos con entidad independiente. A priori no se puede establecer una diferencia entre el proyecto en general y los módulos de los que debe constar puesto que esto depende fuertemente del tipo de aplicación a realizar. En función de la aplicación será el director y los programadores del proyecto los que decidirán el número y la estructuración de los módulos que debe contener.

Por otra parte se recomienda la creación y difusión de librerías de propósito específico que faciliten la codificación y de esta forma fomenten la reutilización del software. Por último es preciso señalar que el estándar de documentación que se propone pretende ser independiente de la plataforma de desarrollo, es decir, no debe depender (en principio) de la máquina, sistema operativo, o compilador utilizado.

2 TERMINOLOGÍA

Todos los archivos dentro de un proyecto software poseen o deben poseer una extensión en su nombre. Éstos se usan para dos cosas. Primero para identificar el tipo de archivo vista al usuario o programador, basta con ver su extensión para saber qué tipo de archivo. Segundo, para ser reconocidos por el entorno y usar el compilador o editor correspondiente a dicho archivo. Las extensiones generales más utilizadas son:

Tabla I: Ficheros comunes en C		
Extensión DOS	**Extensión UNIX**	**Descripción**
.c	.c	Códigos fuente en lenguaje C
.h	.h	Archivos cabecera en lenguaje C
.cpp	.cc	Códigos fuente en lenguaje C++
.hpp	.hpp	Archivos cabecera en lenguaje C++
.obj	.o	Archivo objeto, generado tras compilación
.dll		Librerías dinámicas de Windows
.exe		Archivo ejecutable
.lib	.a ó .l	Librerías

De igual forma existen una serie de extensiones comunes dentro del entorno de desarrollo de Visual C++.

Tabla II: Ficheros incorporados por Visual C++	
Extensión	**Descripción**
.dsp	Fichero general del proyecto
.dsw	Configuración del entorno de trabajo para el proyecto
.rc	Recursos de Visual C++ incorporados al proyecto

3 ESTRUCTURA DE DIRECTORIOS

En este apartado se establece la organización de los archivos que componen el proyecto durante su desarrollo. Se propone la siguiente organización de los subdirectorios que componen el proyecto. En el ejemplo que se propone a continuación se establece un proyecto (aplicación) que no presenta más que un módulo.

\proyecto Contiene todos los subdirectorios del proyecto o aplicación.

- **\inc** Contiene todos los archivos .h y .hpp producidos en la generación del módulo.
- **\src** Contiene todos los archivos .c y .cpp producidos en la generación del módulo.
- **\lib** Contiene las librerias utilizadas por el módulo.
- **\obj** Contiene todos los archivos de código objeto generados en el módulo.
- **\exe** Contiene todos los archivos ejecutables generados en el módulo.
- **\inp** Contiene los ficheros de entrada que necesite el módulo.
- **\out** Contiene los ficheros de salida que genere el módulo.
- **\doc** Contiene los ficheros de documentación del módulo.
- **\hlp** Contiene los ficheros de ayuda generados en el módulo.
- **\rsc** Contiene los ficheros de recursos

Para el caso de proyectos que tengan diferentes módulos la estructura es completamente análoga, teniendo una subestructura por cada módulo.

\proyecto	Contiene todos los subdirectorios del proyecto o aplicación.
\modulo1	Contiene los subdirectorios dentro del módulo 1
\inc	Contiene todos los archivos .h y .hpp producidos en la generación del módulo 1.
\src	Contiene todos los archivos .c y .cpp producidos en la generación del módulo 1.
\lib	Contiene las librerias utilizadas por el módulo 1.
\obj	Contiene todos los archivos de código objeto generados en el módulo 1.
\exe	Contiene todos los archivos ejecutables generados en el módulo 1.
\inp	Contiene los ficheros de entrada que necesite el módulo 1.
\out	Contiene los ficheros de salida que genere el módulo 1.
\doc	Contiene los ficheros de documentación del módulo 1.
\hlp	Contiene los ficheros de ayuda generados en el módulo 1.
\rsc	Contiene los ficheros de recursos del módulo 1
\modulo2	Contiene los subdirectorios dentro del módulo 1
\inc	Contiene todos los archivos .h y .hpp producidos en la generación del módulo 2.
\src	Contiene todos los archivos .c y .cpp producidos en la generación del módulo 2.
\lib	Contiene las librerias utilizadas por el módulo 2.
\obj	Contiene todos los archivos de código objeto generados en el módulo 2.
\exe	Contiene todos los archivos ejecutables generados en el módulo 2.
\inp	Contiene los ficheros de entrada que necesite el módulo 2.
\out	Contiene los ficheros de salida que genere el módulo 2.
\doc	Contiene los ficheros de documentación del módulo 2.
\hlp	Contiene los ficheros de ayuda generados en el módulo 2.
\rsc	Contiene los ficheros de recursos del módulo 2.
\modulo3	
...	

4 VARIABLES GLOBALES

El uso de variables globales en los proyectos debe seguir las siguientes normas:

- Utilizar el menor número de variables globales. Sólo se deben utilizar en aquellos casos en los que sea estrictamente necesario. La utilización de las variables globales debe ser controlada por el director del proyecto (ó coordinador de la programación).

- Reducir al mínimo el alcance de cada variable global (por ejemplo un módulo, unos módulos, una librería, el proyecto, ...).

- Declarar las variables globales como externas en el archivo cabecera de cada fichero fuente según la estructura aportada en el capítulo 6.

- Seguir la sintaxis indicada en el capítulo 10 a la hora de elegir el nombre.

5 ESTRUCTURA DEL CÓDIGO FUENTE

Los proyectos extensos deben estar separados en módulos cada uno de los cuales debe tener una entidad independiente. Los módulos se entienden como entidades autónomas que proveen una serie de funciones públicas al programa principal o a otros módulos. Los módulos a su vez estarán formados por varios ficheros fuente y sus correspondientes cabeceras. Cada fichero fuente debe llevar asociado su correspondiente fichero de cabecera. Cada módulo dispondrá de un fichero cabecera de módulo que debe incluir únicamente las cabeceras de los archivos para la utilización del mismo.

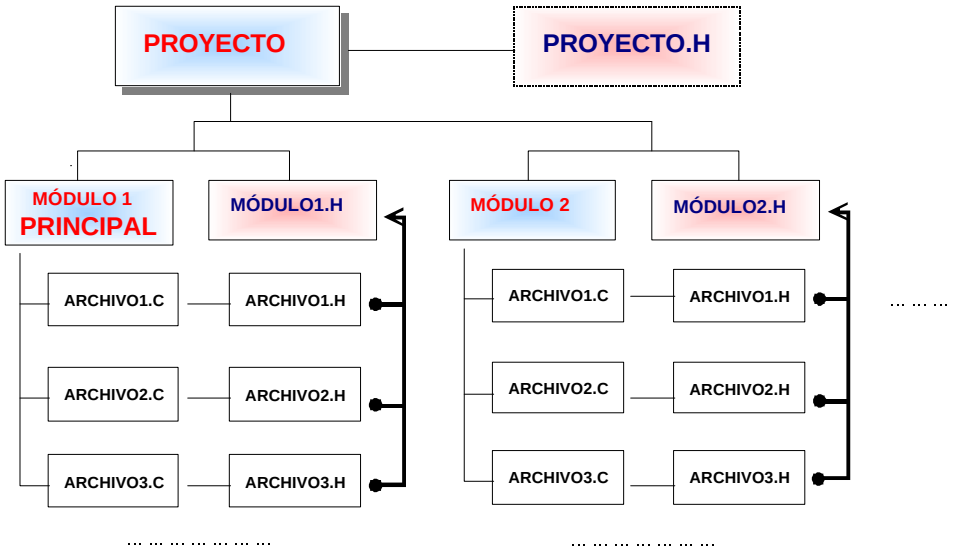

Figura 1 : Arquitectura modular

En caso necesario puede existir un archivo de cabecera general del proyecto (*proyecto.h*) que contenga vinculaciones a cabeceras de los diferentes módulos que se crean necesarios.

Es muy importante estructurar los programas de forma que las funciones se descompongan en tareas concretas y definidas, evitando funciones con código demasiado extenso que impidan su legibilidad.

Cuando se realiza un proyecto en C++ mediante clases, se debe utilizar un archivo independiente para cada clase, definiendo la clase en la cabecera e implementando los métodos en el archivo de código (.CPP).

En el siguiente gráfico se presenta la estructura de los módulos dentro del proyecto.

6 ESTRUCTURA INTERNA DE LOS ARCHIVOS EN C

6.1 Archivos de Encabezado (.H)

Los archivos de encabezados correspondientes a un archivo de código fuente deben seguir una estructura que evite errores de compilación por redefinición de identificadores. Para ello, el contenido de los archivos de encabezamiento debe estar contenido en un bloque de directivas condicionales.

Así mismo, dentro del contenido, se codificarán tres bloques diferenciados:

- **Definiciones comunes**: relación de tipos de datos, clases, variables globales y prototipos de funciones utilizadas tanto en el archivo de código fuente(interno) como exportadas a otros archivos o módulos (externo).
- **Definiciones externas**: relación de variables globales exportadas a otros archivos o módulos.
- **Definiciones internas** : relación de tipos de datos, y prototipos de las funciones utilizadas únicamente en el archivo de código fuente.

Tanto las definiciones comunes, externas como internas deben contener los siguientes elementos:

- Definición de Constantes (mediante la directiva '#define' o 'const')
- Definición de Macros (mediante la directiva '#define').
- Definición de estructuras de datos.

La estructura del archivo de encabezado se concreta en el siguiente ejemplo:

```
//*****************************************************************
//
//    NOMBRE DEL ARCHIVO:      prueba.h
//    MÓDULO:
//    FECHA:
//    AUTOR:
//    DESCRIPCIÓN:
//
```

```
//*********************************************************************

#ifndef  __PRUEBA_H__
#define  __PRUEBA_H__

//---------------------------------------------------------------------
// Bloque de definiciones  comunes
//---------------------------------------------------------------------
// Definición de constantes
#define  CONSTANTE1      1000L
const int CONSTANTE2=23;

// Definición de macros
#define CUADRADO(x)((x) *(x))

//Definición de tipos de datos y clases
typedef struct {
      float real;
      float img;
} complejo;

// Prototipos de funciones globales
int Funcion1( );

//---------------------------------------------------------------------
#ifndef     __PRUEBA_C__
//---------------------------------------------------------------------
// Bloque de definiciones externas

// Variables Globales externas
extern int global1;

//---------------------------------------------------------------------
#else
//---------------------------------------------------------------------
// Bloque de definiciones internas

// Prototipos de funciones internas
int Funcion2( );

//---------------------------------------------------------------------
#endif       // Fin bloque de definiciones internas
#endif       // Fin bloque de la cabecera
```

6.2 Archivos de Encabezado en C++

Los archivos de encabezados definidos anteriormente son válidos tanto en C como en C++, pero al ser la cabecera el lugar donde se ubica la definición de las clases es preciso especificar con más detalle su estructura presentando un ejemplo.

La clase se declara en el bloque de definiciones comunes presentando en primer lugar sus componentes privadas y posteriormente las componentes públicas. Se recomienda que los constructores y el destructor vayan en primer lugar evitando la implementación de dichos métodos dentro de la cabecera. En todo caso se recomienda mantener el mismo orden en la declaración de métodos de la clase y en su implementación en el fichero de código .CPP.

La estructura del archivo de encabezado se concreta en el siguiente ejemplo:

```
//*********************************************************************
//
//    NOMBRE DEL ARCHIVO:       PILA.h
//    MÓDULO:
//    FECHA:
//    AUTOR:
//    DESCRIPCIÓN:
//
//*********************************************************************
#ifndef  __PILA_H__
#define  __PILA_H__

//---------------------------------------------------------------------
// Bloque de definiciones  comunes
//---------------------------------------------------------------------
// Definición de constantes
#define     LIMITE 100

// Definición de macros
#define CUADRADO(x)((x) *(x))

//Definición de tipos de datos y clases
class CPila {

private:
      int tam;     // tamaño de la pila
      char *ppila; // puntero a la pila
      int top;     // última posición ocupada +1

public:
      CPila(const int t=LIMITE);// constructor
      CPila(const CPila &pila_orig);// constructor copia
      ~CPila();                 // destructor

      // operador asignación
      CPila& operator = (const CPila & pila_orig);

      void Meter( const char c);// introduce un carácter en la pila
      char Sacar(void);         // extare un caracter de la pila
      int Vacia(void);          // Comprueba si la pila está vacía
      int Llena(void);          // Comprueba si la pila está llena

};    // fin de la clase CPila

// Prototipos de funciones globales


//---------------------------------------------------------------------
#ifndef     __PILA_CPP__
//---------------------------------------------------------------------
// Bloque de definiciones externas

// Variables Globales externas

//---------------------------------------------------------------------
#else
//---------------------------------------------------------------------
// Bloque de definiciones internas

// Prototipos de funciones internas

//---------------------------------------------------------------------
#endif      // Fin bloque de definiciones internas
#endif      // Fin bloque de la cabecera
```

6.3 Archivos Código Fuente (.C)

Los archivos de código fuente, tanto para proyectos cortos que consten de un solo archivo, como en componentes de un módulo, deben seguir la siguiente estructura:

- Cajetín de archivo.
- Directiva '#define __NOMBREFICHERO_C__' (para identificar el archivo en los ficheros de encabezamiento)
- Inclusión de los archivos de encabezamiento de las librerías estándar (mediante la directiva '#include')
- Archivos de encabezamiento de otras librerías usadas
- Archivos de encabezamiento del módulo o proyecto (cuando proceda)
- Inclusión del archivo de encabezamiento del propio fichero.
- Declaración e inicialización de variables globales externas.
- Declaración e inicialización de variables globales internas al archivo (declaradas como static).
- Declaración de funciones precedida por un cajetín con la información necesaria.

Ejemplo:

```
//*********************************************************************
//
//    NOMBRE DEL ARCHIVO:       prueba.c
//    MÓDULO:
//    FECHA:
//    AUTOR:
//    DESCRIPCIÓN:
//
//*********************************************************************

#define  __PRUEBA_C__

// Librerías Estándar
#include  <stdio.h>

// Otras librerías
#include "..\inc\matriz.h"

// Encabezado Módulo-Proyecto

// Encabezado del archivo
#include "..\inc\prueba.h"


// Variables Globales externas
int global1=0;

// Variables Globales internas
static int global2=0;

//*********************************************************************
//    Nombre:
//    Variables globales:
//    Variables de salida:
//    Comentarios (parámetros):
//
//*********************************************************************
int Funcion1( char *par1)
{

}
```

6.4 Archivos Código Fuente en C++ (.CPP)

Los archivos de código fuente en C++ utilizarán la misma estructura que los archivos de C comentados en el apartado anterior. Se debe utilizar un archivo por cada clase definida implementando cada uno de los métodos declarados en la cabecera.

Ejemplo:

```
//*********************************************************************
//
//    NOMBRE DEL ARCHIVO:       Pila.cpp
//    MÓDULO:
//    FECHA:
//    AUTOR:
//    DESCRIPCIÓN:
//
//*********************************************************************

#define  __PRUEBA_CPP__

// Librerías Estándar
#include  <stdio.h>

// Otras librerías

// Encabezado Módulo-Proyecto

// Encabezado del archivo
#include "..\inc\pila.h"

// Variables Globales externas

// Variables Globales internas

//*********************************************************************
//    Nombre:
//    Variables globales:
//    Variables de salida:
//    Comentarios (parámetros):
//
//*********************************************************************
int CPila::Cpila()
{

}
```

7 SINTAXIS DE LAS FUNCIONES

Tanto en C como en C++ una función debe ser declarada (prototipo) antes de ser usada. En un caso por imperativos del compilador (C++), y en el otro para evitar errores en la programación. Esto es así por dos razones:

- No tener una función declarada implica que no se comprueba el tipo de los parámetros en las llamadas a la función antes de la declaración.
- Tener todos los prototipos agrupados aporta claridad a los programas.

7.1 Colocación de las funciones en los ficheros cabecera

Los prototipos se encontrarán situados en la cabecera asociada al archivo fuente donde se define. Todos estos prototipos deben situarse dentro del bloque común excepto aquellas que sean de uso exclusivamente interno.

7.2 Nomenclatura de las funciones

- No deben usarse signos de puntuación en el interior de los nombres. Tampoco deben dejarse espacios en blanco (‘ ‘) ni usar el símbolo de subrayado (‘_’) excepto en la separación de la identificación del módulo ó en la identificación de versiones de una misma función. Ejemplo:

```
VIS_AbrirImagen_2()
```

- En caso necesario se insertará un prefijo de módulo que estará formado por tres letras en mayúsculas seguido del símbolo de subrayado (‘_’). Este identificador permite señalar a qué módulo pertenece la función dentro de un proyecto complejo formado por varios módulos. Evidentemente en aquellos casos en los que no se utilicen módulos dentro de un proyecto, esto no será necesario.

- El cuerpo del nombre se escribirá en minúsculas excepto la primera letra de cada palabra que irá en mayúsculas, no existiendo espacio entre las diferentes palabras que compongan el nombre.

- Se debe usar como nombre verbos en tiempo infinitivo que describan la acción que realiza esta función así como el objeto al que afecta dicha acción.

Ejemplos:

```
FIC_RecogerEnteroFichero(FILE *fp, int *nombre)
FIC_CerrarFichero(FILE fp)
```

7.3 Comentarios

La definición de la función debe ir precedida por el cajetín normalizado presentado previamente. En caso de funciones muy cortas en las que el cajetín no sea necesario dicha función deberá ir enmarcada por dos líneas de asteríscos.

Ejemplos:

```
//*********************************************************************
//    Nombre:
//    Variables globales:
//    Variables de salida:
//    Comentarios (parámetros):
//
//*********************************************************************
int Funcion1()
{
      ...
```

```
      ...
}

//*********************************************************************
int Funcion2()
{
      ...
}
//*********************************************************************
```

8 CONSTANTES Y MACROS

Las constantes y macros deben estar siempre definidas en el fichero de cabecera correspondiente. Se escribirán siempre en mayúsculas y en caso necesario con el prefijo del módulo al que pertenecen. Se puede utilizar la directiva *'#define'* o el modificador *'const'*. En la definición de las macros debe utilizarse los paréntesis para evitar la precedencia de otros posibles operadores. Ejemplo:

```
#define IMG_TAMANOIMAGEN 512

#define IMG_CUADRADOPIXEL(x)    ((x)*(x))

const int IMG_LIMITE=100;
```

9 SINTAXIS GENERAL DE LOS PROGRAMAS

9.1 Comentarios

Es extremadamente útil la utilización de comentarios en aquellas partes del código que permitan explicar sentencias que puedan ser no muy claras. Ejemplos:

```
A=elem[13].b;            // Comentarios

typedef struct
{
      long val[10];      // Comentarios
      int *p;            // Comentarios
      char *nombre;      // Comentarios
} tabla;
```

Es recomendable el uso de comentarios del tipo / / en lugar del tipo /* .... */. Conviene que estos comentarios se separen con un tabulador de la sentencia para mayor claridad. Asimismo si se ponen varios seguidos queda más claro si comienzan a la misma altura. No es preciso ni recomendable poner comentarios donde el código sea 'suficientemente' claro.

Además existen los comentarios de línea, comentarios que describen una parte de código que se encuentra a continuación. Estos comentarios deben estar alineados con el código escrito. Ejemplo:

```
// Busqueda de elemento minimo
minimo = MAXINT;
for (i=0; i< MAX_TABLA_1; i++)
      minimo = min(minimo, TABLA_1[i]);
for (i=0; i< MAX_TABLA_2; i++)
      minimo = min(minimo, TABLA_2[i]);
```

La premisa fundamental al escribir los comentarios, es que debe quedar claro de un simple vistazo qué es código y qué son comentarios.

9.2 Edición de programas

En este apartado se recogen algunas normas básicas que deben seguirse en la programación de las diferentes sentencias.

- El ancho de una línea no debe superar nunca los 80 caracteres Si en una línea en la que hemos abierto un paréntesis que todavía no ha sido cerrado tenemos la necesidad de incluir más de 80 columnas debemos escribir las sucesivas líneas en la posición siguiente a dicho paréntesis. Ejemplo:

```
FunciónPrimera(variable1, variable2, variable3, FuncionSegunda(
                variable1, variable2, variable3));
```

- Si se trata de una línea en la que no hay paréntesis, los símbolos como '=' deben estar colocados en la línea superior y el sangrado debe hacerse como sigue:

```
LocalA = globalB + globalC +
       otraVariable;
```

- Debe haber un espacio blanco antes y después de cada operador binario. Ejemplo:

```
   a = b;
```

- No debe haber un espacio blanco entre un operador unario y el operando.
- Después de cada signo de puntuación ',' ó ';' debe haber un espacio en blanco pero nunca antes.
- El nivel de sangrado o de indentación debe ser de 4 espacios.
- No debe dejarse un espacio después de abrir un paréntesis ni antes de cerrarlo.
- Se debe dejar siempre una línea en blanco después de la declaración de las variables.

- No debe haber una línea en blanco entre el bloque de declaraciones pasadas en una función y el código del mismo.

- Las llaves '{' que abren el cuerpo de un bucle, ya sea 'if', 'while', 'for' ó 'do-while', se situarán en la siguiente línea, a la misma altura de sangrado que la condición. Ejemplo:

```
if (a && b)
{
        a = b + d;
        a = b;
}
else
{
        a = c;

}
```

- La llave que cierra el cuerpo de un bucle se pone al mismo nivel de sangrado que la llave de apertura.

- La estructura de una sentencia 'switch' sería:

```
switch (a)
{
                    printf("....");
                    break;
      case 'D':
      case 'E':
                    break;
      default:      printf("Error");

}
```

- Al concatenar un 'if' con el código 'else' de un 'if' anterior, puede optarse por una e as dos a continuación

```
{
      C = D;
      B = D;
}
      if (C == D)
      {

            A = D;
      }
      else

            F = M;
            G = N;
      }
```

```
if (A == B)
{
      C = D;
      B = D;

else  if (C == D)



}
else
{
      F =

}
```

- Entre el nombre de una función y el paréntesis que lo sigue no debe haber espacio en blanco.

- El código de una función debe empezar a escribirse a un nivel de sangrado superior que la llave de apertura, ya que cada vez que se abre una llave se inicia un bloque

lógico, por lo que hay que añadir un nivel de sangrado. Se puede observar que siempre que se abren llaves se inicia un bloque lógico, por lo que hay que añadir un nivel de sangrado.

10 SINTAXIS DE LAS VARIABLES

Las variables siempre se declararán en minúscula. En la medida de lo posible el nombre de la variable debe hacer referencia al uso al que ésta se encuentre destinada.

No deben usarse signos de puntuación en el interior de nombres de variables. Tampoco puede usarse el símbolo '_' ni dejar espacios en blanco.

Cuando una variable contenga más de una palabra, comenzarán siempre en minúscula, y se identificará cada comienzo de palabra por la correspondiente letra mayúscula. Ejemplo:

```
int  pintaVentana;
```

El nombre de la variable debe ser un sustantivo que se ajuste al contenido de la variable y al uso que se va a hacer de la misma.

Las variables utilizadas como contadores comunes con una única letra se escribirán en minúscula, y se utilizarán como nombres habituales 'i', 'j', 'k', etc.

Las variables que se declaren de tipo puntero deberán comenzar por p (que denote que es puntero) antes del nombre del puntero que evidentemente comenzará por mayúscula. Ejemplo:

```
FILE *pFichero;
```

11 SINTAXIS DE LAS CLASES EN C++

Cuando se defina una clase, su nombre debe comenzar con la letra 'C' en mayúsculas, seguido del nombre descriptivo que también empezará en mayúsculas. Se procurará utilizar nombres cortos sin signos de puntuación, símbolo '_' o espacios en blanco.

```
class CVector {
private:

public:


};
```

12 DOCUMENTACIÓN

Los documentos generados en las diferentes etapas del ciclo de vida software son los que componen la documentación final del proyecto. Dentro de ésta, lógicamente está incluida la documentación del código fuente. La estructura de la documentación debería ser la siguiente:

12.1 Especificación de requisitos

En este apartado se especifican todas las funcionalidades del sistema a desarrollar así como también sus interfases, características de rendimiento, tipo de hardware y restricciones. Es un documento generado tanto por el usuario como por el grupo de desarrollo.

12.2 Análisis

Dependiendo del tipo de metodología usada se generan documentos equivalentes que reflejan todo el proceso de análisis. En este caso podríamos tener:

Documentos:
- DFD, Diagrama de Flujo de Datos
- Esquema de clases

12.3 Diseño

El documento más importante al finalizar el proceso de diseño es el pseudocódigo de todos los métodos o funciones usados en el sistema. Estos pseudocódigos son derivados de los procesos o de las clases identificadas en la etapa de análisis. Otra representación paralela al pseudocódigo son los diagramas de bloques. Estos diagramas dan una idea general del proceso o módulo que se trata de representar, deben ser sencillos y sólo se deben presentar las partes importantes que reflejan la estructura del módulo o función.

Documentos:
- Pseudocódigo
- Diagrama de Bloques

12.4 Código Fuente

Este es uno de los documentos más importantes. Una buena documentación de código facilita su mantenimiento. Una forma de estructurar este documento es la siguiente:

12.4.1 Introducción

Es una breve descripción del código. Ésta debe hacer referencia al proyecto o al sistema que contiene a este código.

12.4.2 Técnicas de programación

En este apartado se describen las técnicas no comunes usadas en este código. Estas técnicas son esos detalles o peculiaridades producto de la creatividad del programador. También se especifican detalles para su uso.

12.4.3 Variables globales

Se debe realizar una tabla con las variables globales indicando su tipo y su descripción. Las variables globales no deben ser prácticamente usadas.

12.4.4 Bases de datos

Cuando un proyecto utiliza bases de datos o archivos de almacenamiento se deben documentar sus estructuras y tipo de datos. Esto facilita las entradas y salidas de datos así como posibles errores ajenos al código.

12.4.5 Diagrama de bloques y/o comunicación entre clases

Los diagramas de bloques y de comunicación entre clases representan la dinámica del sistema. Deben incluirse en aquellas partes críticas del código. Así mismo debe incluirse la jerarquía de clases.

12.4.6 Código fuente (comentado)

Finalmente se incluye el código fuente comentado según los apartados anteriores.

ESTA PÁGINA HA SIDO DEJADA EN BLANCO INTENCIONADAMENTE

ISA-UMH

INGENIERIA DE SISTEMAS Y AUTOMATICA

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

Universidad Miguel Hernández

Avda. Ferrocarril s/n
03202 Elche – Alicante
ESPAÑA

Tel: +34 (6) 665 86 55
Fax: +34 (6) 665 86 55
http://www.umh.es