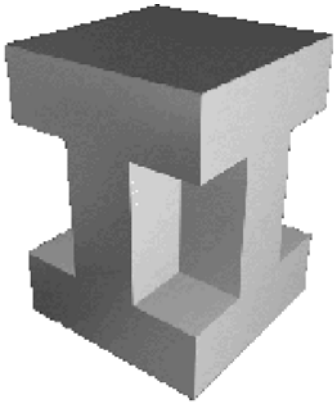
GUÍA DE ESTILO DE C++
(REDUCIDA)



Ingeniería Informática

Seminario de Programación
Departamento de Informática

GUÍA DE ESTILO DE C++

1. Introducción

Este documento tiene como finalidad proporcionar un conjunto de reglas que nos ayuden a escribir programas en C++ con un "buen estilo". Un código escrito con buen estilo es aquel que tiene las siguientes propiedades:

- Está organizado.
- Es fácil de leer.
- Es fácil de mantener.
- Es fácil detectar errores en él.
- Es eficiente.

Hay muchos estilos que cumplen estas características. En este documento simplemente vamos a dar uno de ellos. Este documento es una versión resumida y adaptada de la guía de estilo para C de Juan José Moreno Moll.

2. Lectura y mantenimiento del código

Los principios de esta sección sirven para aumentar la legibilidad del código y para facilitar su mantenimiento. Éstos son:

- Organizar programas utilizando técnicas para encapsular y ocultar información.
- Aumentar la legibilidad usando los espacios y líneas en blanco.

- Añadir comentarios para ayudar a otras personas a entender el programa.
- Utilizar identificadores que ayuden a entender el programa.

2.1. Encapsulación y ocultación de información

La encapsulación y ocultación de información ayudan a organizar mejor el código y evitan el acoplamiento entre funciones del código.

La **encapsulación** permite agrupar elementos afines del programa. Los subprogramas afines se agrupan en ficheros (unidades), y los datos en grupos lógicos (estructuras de datos).

Ocultación de información: Un subprograma *no necesita* saber lo siguiente:

- La fuente de los parámetros que se le pasan como entrada.
- Para que servirán sus salidas.
- Qué subprogramas se activaron antes que él.
- Qué subprogramas se activarán después que él.
- Cómo están implementados internamente otros subprogramas.

Para conseguir esto se deben seguir las siguientes reglas:

- No hacer referencia o modificar variables globales (evitar efectos laterales).
- Declarar las variables y tipos como locales a los subprogramas que los utilizan.
- Si queremos evitar cambios indeseados en parámetros, pasarlos por valor.
- Un procedimiento sólo debe modificar los parámetros pasados en su llamada.

2.2. Uso de comentarios

Los comentarios dan información sobre lo que hace el código en el caso que no sea fácil comprenderlo con una lectura rápida. Se usan para **añadir información** o **para aclarar secciones de código**. No se usan para describir el programa. Por lo tanto no se deben poner comentarios a una sola instrucción. Los comentarios se añaden en los niveles siguientes:

- **Comentario a ficheros**: Al comienzo de cada fichero se añade un *prólogo del fichero* que explica el propósito del fichero y da otras informaciones.
- **Comentarios a subprogramas**: Al comienzo de cada subprograma se añade un *prólogo del subprograma* que explica el propósito del subprograma.
- **Comentarios dentro del código**: Estos comentarios se añaden junto a la definición de algunas variables (las más importantes), para explicar su propósito, y al comienzo de algunas secciones de código, especialmente complicadas, para explicar que hacen.

Los comentarios se pueden escribir en diferentes estilos dependiendo de su longitud y su propósito.
En cualquier caso seguiremos las siguientes **reglas generales**:

- Los **comentarios** en general se escriben **en líneas que no contienen código** y antes del código que queremos clarificar. Esta regla se aplica siempre si el comentario tiene más de una línea.
- **Sólo** en dos casos se permite **poner en la misma línea** un comentario y una instrucción: **comentarios a una definición de variable**, que explica la finalidad de esta variable. Y un **comentario** para indicar **final de una estructura del lenguaje**.

Aquí vamos a describir como hacer **comentarios dentro del código**. Dentro de este tipo de comentarios se pueden distinguir:

- Comentarios en cajas: Usados para prólogos o para separar secciones.
- Separador de secciones: Son líneas que sirven para separar secciones, funciones, etc.
- Comentarios para bloques grandes: Se usan al comienzo de bloques de código grandes para describir esa porción de código.
- Comentarios cortos: Se usan para describir datos y casi siempre se escriben en la misma línea donde se define el dato. También se usan para indicar el final de una estructura.
- Comentarios para bloques pequeños: Se escriben para comentar bloques de instrucciones pequeños. Se colocan antes del bloque que comentan y a la misma altura de sangrado.

Comentarios en cajas:

Ejemplo: comentario tipo prólogo en una caja:

```
/************************************************************/
/*      AUTOR: Nombre                                       */
/*                                                          */
/*      PROPÓSITO:                                          */
/*                                                          */
/************************************************************/
```

Separador de secciones:

Ejemplo: Separador de secciones.

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

Comentarios para bloques grandes:

Ejemplo: comentarios para bloques grandes de código.

```
/*
 *     Usar para comentarios a más de un bloque de
 *     sentencias.
 */
```

Comentarios cortos:

En caso de utilizar este tipo de comentario, seguir las siguientes reglas:

- Utilizar uno o más tabuladores para separar la instrucción y el comentario.

- Si aparece más de un comentario en un bloque de código o bloque de datos, todos comienzan y terminan a la misma altura de tabulación.

Ejemplo: Comentarios en un bloque de datos.

```
int alu_teoria;       // Número de alumnos por grupo teoría
int alu_practicas;    // Número de alumnos por grupo prácticas
```

Ejemplo: Comentarios al final de una estructura.

```
for(i = 1; i <= FINAL; i++)
{
    while ( !correcto )
    {
        correcto = false;
```

```
            cout << "\n Introduce dato:";
            cin >> dato;
            if (dato < 100)
                correcto = true;
        }; // WHILE
        cout << "\n"
    }; // FOR
```

Comentarios para bloques pequeños:

Ejemplo:

```
// Hallamos la nota media de todos los examenes
for(i = 0; i < NUM_ALUMNOS; i++)
    suma = suma + nota[i];
media = suma / NUM_ALUMNOS;
```

2.3. Uso de espacios y líneas en blanco

Los espacios en blanco facilitan la lectura y el mantenimiento de los programas. Los espacios en blanco que podemos utilizar son: líneas en blanco, carácter espacio, sangrado.

Línea en blanco:

Se utiliza para separar "párrafos" o secciones del código. Cuando leemos un programa entendemos que un fragmento de código entre dos líneas en blanco forma un conjunto con una cierta relación lógica.

Veamos como separar secciones o párrafos en el programa:

- Las secciones que forman un programa se separan con al menos una línea en blanco (declaración de constantes, declaración de variables, programa principal, …).
- Dentro de un subprograma se separan con una línea en blanco las secciones de declaraciones y el código del subprograma.
- Dentro de un subprograma se separan con una línea en blanco los fragmentos de instrucciones muy relacionadas entre sí (por ejemplo, conjunto de instrucciones que realizan una operación).

Espacio en blanco:

Los espacios en blanco sirven para facilitar la lectura de los elementos que forman una expresión. Los espacios en blanco se utilizan en los casos siguientes:

- Las variables y los operadores de una expresión deben estar separados por un elemento en blanco.
- Las lista de definición de variables, y las listas de parámetros de una función se deben separar por un espacio en blanco.

Ejemplos:

Espaciado correcto: `media = suma / cuenta;`

Espaciado incorrecto: `media=suma/cuenta;`

Espaciado dentro de una lista de parámetros: `concat(s1, s2)`

Sangrado:

El sangrado se utiliza para mostrar la estructura lógica del código. El sangrado óptimo es el formado por **cuatro espacios**. Es un compromiso entre una estructuración legible y la posibilidad de que alguna línea (con varios sangrados) del código supere el ancho de una línea de una hoja de papel o del monitor.

2.4. Elección adecuada de identificadores

Los identificadores que dan nombre a subprogramas, constantes, tipos o variables han de colaborar a la autodocumentación del programa, aportando información sobre el cometido que llevan a cabo. Para elegir nombre se deben seguir las siguientes recomendaciones generales:

- Elegir nombres comprensibles y en relación con la tarea que corresponda al objeto nombrado.
- Seguir un criterio uniforme con las abreviaturas de nombres. Elegir abreviaturas que sugieran el nombre completo.
- Utilizar prefijos y sufijos cuando sea necesario.
- Uso del carácter '_' o de una letra mayúscula para distinguir las palabras que forman un identificador.

Nombres habituales:

Hay algunos nombres cortos que se usan muy habitualmente. El uso de estos nombres (y sólo de estos) es aceptable.

Ejemplo: nombres cortos aceptables.

```
c, ch        caracteres
i, j, k      índices
n            contadores
p, q         punteros
s, Cad       cadenas
```

Uso de mayúsculas y minúsculas:

Para reconocer fácilmente la clase de un identificador se utilizan las siguientes normas de utilización de mayúsculas y minúsculas cuando se construyen nombres:

- **Variables**: los nombres de las variables se construyen con palabras (o abreviaturas) minúsculas separadas por el carácter '_'.

  Ejemplo:
  ```
  num_alumnos
  ```

- **Nombres de Funciones y tipos**: los nombres de las funciones y los tipos se pueden escribir en dos estilos:

  a) Una o más palabras en minúsculas separadas por '_'. Sólo si una de las palabras es un nombre propio se admite que la primera letra de una palabra esté en mayúscula.

  Ejemplo:
  ```
  insertar_alumno
  ```

  b) Una o más palabras en minúsculas excepto la primera letra, que es mayúscula (las demás no). No se usa el carácter '_'. La primera letra mayúscula de cada palabra indica la separación entre palabras.

  Ejemplo:
  ```
  InsertarAlumno
  ```

- **Constantes e identificadores del preprocesador** : usan nombres construidos con palabras en mayúsculas separadas por el carácter '_'.

  Ejemplo:
  ```
  MAX_ALUMNOS
  ```

3. Organización interna

En este apartado vamos a describir la forma correcta de escribir ciertas estructuras del lenguaje.

3.1. Declaraciones:

Las diferentes secciones de declaraciones deberán estar claramente diferenciadas y seguir el siguiente orden:

- Directivas *#include*
- Declaración de constantes (*#define* o *const*).
- Declaración de tipos (*typedef*).
- Declaración de variables.
- Prototipos de funciones

Después de los prototipos irá el código del programa principal (*main()*) y a continuación el código del resto de las funciones.

Las declaraciones deben seguir las siguientes reglas:

- Alinear los nombres de las variables de forma que la primera letra del nombre de cada variable esté en la misma columna.
- Definir una variable por línea junto con su comentario (si éste es necesario).

  Como excepción a esta regla podemos agrupar en la misma línea variables auxiliares, contadores, etc.
- Si un grupo de subprogramas utiliza un parámetro o una variable interna para una labor semejante llamar con el mismo nombre a esa variables en todas las funciones.
- No utilizar nombres para variables internas que sean iguales a nombres de variables globales.

3.2. Pares de llaves: { }

La forma que tiene C++ de agrupar instrucciones en bloques es utilizar las llaves { }. Su colocación se debe hacer en líneas reservadas para cada una de ellas, sin ninguna otra instrucción en esa línea. Ambas deben ir en la misma columna que la instrucción de la que dependen.

Ejemplo:

```
for(i = 0; i < N; i++)
{
    vect[i] = vect2[i];
    suma = suma + vect[i];
};
```

Sentencia if:

La sentencia o sentencias que corresponden a la sección *then* o *else* van siempre en una nueva línea sangrada:

```
if (expresión)
    sentencia;
else
    otra_sentencia;
```

```
if (expresión)
{
    sentencia1;
    sentencia2;
}
else
{
    otra_sentencia1;
```

```
        otra_sentencia2;
};
```

Sentencia switch:

Los valores asociados a la sentencia *switch* irán en línea aparte sangrada. El bloque de sentencias asociado comenzará en otra línea aparte y también sangrada. Todos los bloques de sentencias pertenecientes al CASE comenzarán en la misma columna.

```
switch (expresión)
{
    case 1:
        sentencia1;
        sentencia2;
        break;
    case 2:
        otra_sentencia;
        break;
    default:
        otra_mas;
};
```

Sentencias de repetición:

La sentencia o sentencias pertenecientes a la estructura de repetición (***for***, ***while*** o ***do ... while***) van siempre en una nueva línea sangrada:

```
while (expresión)
    sentencia;
```

```
while (expresión)
{
    sentencia1;
    sentencia2;
};
```

4. Portabilidad y eficiencia

Un código bien escrito debe poder ejecutarse en otras máquinas o plataformas haciendo un mínimo de cambios. Nuestro código debe ser lo más portable posible.

4.1. Reglas para mejorar la portabilidad:

- Usar ANSI C++ en la medida de lo posible.
- Escribir código portable en principio. Considerar detalles de optimización sólo cuando sea necesario. Muchas veces la optimización es diferente según la plataforma o la máquina utilizada. En todo caso:
  a) documentar estos detalles para poder modificarlos si necesitamos cambiar el código de plataforma.
  b) aislar la optimación de otras partes del código.
- Algunos subprogramas son no portables de forma inherente (por ejemplo '*drivers'* de dispositivos suelen ser distintos en sistemas operativos distintos). Aislar este código del resto.
- Organizar el código en ficheros de forma que el código portable esté en ficheros distintos del código no portable.
- Computadoras distintas pueden tener un tamaño de tipos diferente. Utilizar el operador *sizeof()* para asegurar la portabilidad en este caso.
- Evitar las multiplicaciones y divisiones hechas con desplazamientos de bits.

- No reemplazar los subprogramas estándar con subprogramas realizados por nosotros. Si no nos gusta una implementación de un subprograma estándar realizar un subprograma semejante pero con otro nombre.
- Utilizar las directivas de compilación condicional para mejorar la portabilidad del código.

4.2. Reglas para mejorar la eficiencia:

- Recordar que el código debe ser mantenido.
- Si la eficiencia del programa no es crítica sustituir instrucciones rápidas por instrucciones comprensibles.
- Si la eficiencia es importante (sistemas en tiempo real) y es necesario utilizar expresiones no comprensibles y muy complicadas pero rápidas, añadir comentarios que ayuden a comprender el código.
- Reducir al mínimo las operaciones de entrada/salida.
- Usar los operadores: `++`, `--`, `+=`, `-=`, etc...
- Liberar memoria tan pronto como sea posible.
- Cuando se pasan estructuras grandes a un subprograma, hacerlo por referencia. Esto evita manipular datos sobre la pila y hace la ejecución más rápida.

5. Programa ejemplo

```
/*****************************************************/
/* Programa Notas                                    */
/* Autor: Fernando Barber                            */
/* Proposito: Manejo de notas de alumnos en un array */
/*****************************************************/

#include <iostream.h>
#include <string>

const int MAX_ALU = 200;

struct Alumno{
        string nombre;
        int nota;
        int curso;
        };

typedef Alumno V_Alumnos[MAX_ALU];

void IntroducirAlumno(Alumno & alu);
void EscribirAlumno(Alumno alu);
void Introducir(V_Alumnos v_alu, int & num_alu);
int PosAlu(string nom_alu, const V_Alumnos v_alu, int num_alu);
void EncontrarAlumno(V_Alumnos v_alu, int num_alu);

/*
 * Programa Principal
 */
int main()
{
    V_Alumnos alumnos;
    int num_alu;
    int opcion;

    num_alu = 0;
    do
    {
        cout << "Menu:\n";
        cout << " 1-Introducir alumnos\n";
        cout << " 2-Encontrar un alumno\n";
        cout << " 3-Salir\n";
        cout << "Introducir opcion:";
        cin >> opcion;
        // Borra el salto de linea del buffer
        cin.ignore();

        switch (opcion)
        {
            case 1 :
                Introducir(alumnos, num_alu);
                break;
            case 2 :
                EncontrarAlumno(alumnos, num_alu);
        };
    }
    while (opcion != 3);
    return 0;
};
```

```
/*
 * Funcion para introducir un alumno
 */
void IntroducirAlumno(Alumno & alu)
{
    cout << "\n Nombre:";
    getline(cin, alu.nombre);
    cout << "\n Nota:";
    cin >> alu.nota;
    cout << "\n Curso:";
    cin >> alu.curso;
    // Borra el salto de linea del buffer
    cin.ignore();
};

/*
 * Funcion para escribir un alumno
 */
void EscribirAlumno(Alumno alu)
{
    cout << alu.nombre << endl;
    cout << alu.nota << endl;
    cout << alu.curso << endl;
};

/*
 * Funcion que va pidiendo los alumnos hasta que se introduce un
 * nombre vacio.
 */
void Introducir(V_Alumnos v_alu, int & num_alu)
{
    Alumno alu;

    cout << "Introduccion del alumno " << (num_alu + 1);
    cout << "\n Pon un nombre vacio para salir";
    IntroducirAlumno(alu);
    while ( alu.nombre != "" )
    {
        num_alu ++;
        v_alu[num_alu] = alu;
        cout << "Introduccion del alumno " << (num_alu + 1);
        cout << "\n Pon un nombre vacio para salir";
        IntroducirAlumno(alu);
    }
};

/*
 * Funcion que devuelve la posicion de un alumno.
 * Devuelve -1 en caso de no encontrarlo.
 */
int PosAlu(string nom_alu, const V_Alumnos v_alu, int num_alu)
{
    int i;
    int posicion;

    i = 0;
    while((i < num_alu) && (v_alu[i].nombre != nom_alu))
        i++;
    if (v_alu[i].nombre != nom_alu )
        posicion = -1;
    else
        posicion = i;
    return posicion;
```

```
};
```

```
/*
 * Funcion que pide el alumno a buscar y lo muestra en pantalla
 */
void EncontrarAlumno(V_Alumnos v_alu, int num_alu)
{
    int posicion;
    string nombre;

    cout << "\n Que alumno quieres encontrar? \n";
    getline(cin,nombre);
    posicion = PosAlu(nombre, v_alu, num_alu);
    if (posicion == -1)
        cout << "Alumno no encontrado \n";
    else
        EscribirAlumno(v_alu[posicion]);
};
```