PROGRAMACIÓN 1 2023/2024

Guía de estilo y diseño de programación

1. Estilo de programación

El código de un programa debe poder ser leído por: a) el programador que lo ha escrito, b) otros programadores (durante el desarrollo y tal vez un tiempo después) y c) el computador (mediante un programa compilador o intérprete). Para el computador no es significativo lo bien escrito que esté, siempre que sea correcto sintácticamente, para los programadores sí lo es. En un programa bien escrito es más fácil encontrar errores, hacer modificaciones, mejoras, etc. No hay que confundir estilo de programación con la "decoración" del código. En general, debe seguirse un estilo de programación encaminado a una buena legibilidad del código [1]. Es importante que dicho estilo sea consistente, por este motivo a nivel profesional se siguen las guías o convenios establecidos para el lenguaje empleado. Por ejemplo:

https://google.github.io/styleguide/cppguide.html
https://gcc.gnu.org/codingconventions.html
https://kernelnewbies.org/New_Kernel_Hacking_HOWTO/Kernel_Programming_Style_Guidelines
https://google.github.io/styleguide/pyguide.html

En esta asignatura es **OBLIGATORIO** seguir la siguiente guía de estilo:

1.1 Identificadores de variables, constantes simbólicas, funciones, etc.

a. Usar identificadores pronunciables y expresivos escritos en minúsculas (salvo en el caso de las constantes simbólicas).

Ejemplo 1

```
int determ = 0;          ¿determinante, determinado, ...?     MAL
int cript = 0;           ¿?
int contador = 0;        ¿Qué cuenta?                         MAL
```

b. La longitud de un identificador no es una virtud, lo es su legibilidad. Por ejemplo, en bucles es más claro usar `i`, `j`, `k`, `n` (de *i-ésimo*, *j-ésimo*, etc.).

Ejemplo 2

```
for (indice = 0; indice < MAX_PRECIPITACIONES; indice++) {     MAL
  cout << precipitaciones[indice] << endl;
}

for (i = 0; i < MAX_PRECIPITACIONES; i++) {                    BIEN
  cout << precipitaciones[i] << endl;
}
```

c. Si el identificador lo forman varias palabras deben separarse usando `'_'` (ver Ejemplo 3).
d. Para los identificadores de variables y constantes simbólicas usar sustantivos, salvo para las booleanas donde deben usarse participios.
e. Para los identificadores de funciones que devuelvan algún valor usar sustantivos, salvo si se devuelve un booleano donde deben usarse participios o preguntas sí/no. Para funciones que no devuelvan ningún valor usar infinitivos.
f. No usar abreviaturas, salvo que formen parte de la jerga o dominio del problema (p. e. IP, DNS) o sean de uso general (p. e. dcha, num).

Ejemplo 3

```
int p_der = 0;                MAL
int posicion_dcha = 0;        BIEN
```

PROGRAMACIÓN 1 2023/2024

1.2 Constantes simbólicas

a. Usar constantes simbólicas para valores arbitrarios sin significado claro o con múltiples usos (i. e. "números mágicos" [2]).

Ejemplo 4

```
if (limite < 1345) {                          MAL
  ...
}

const int LIMITE_MAX_DCHA = 1345;             BIEN
...
if (limite < LIMITE_MAX_DCHA) {
  ...
}
```

b. Sus identificadores deben ser expresivos (identificar su "misión" no su valor).

Ejemplo 5

```
const char BARRA = "/";                                              MAL
...
cout << dia << BARRA << mes << BARRA << año << endl;

const char SEPARADOR_FECHA = "/";                                    BIEN
...
cout << dia << SEPARADOR_FECHA << mes << SEPARADOR_FECHA << año << endl;
```

c. Sus identificadores se escribirán totalmente en mayúsculas, separando las palabras como en las variables con `'_'` (ver Ejemplo 4).

1.3 Espaciado horizontal y vertical

a. Usar adecuadamente espacios en blanco en las expresiones.

Ejemplo 6

```
distancia=sqrt(pow(x2-x1,2)+pow(x2-x1,2));                MAL
distancia = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));      BIEN
```

b. Usar líneas en blanco para separar las diferentes secciones de código dentro de una función (ver Ejemplo 7).

c. Usar márgenes adentrados (i. e. sangrías) para reflejar los niveles de anidamiento. No dejar demasiado margen (mejor 2 o 4 espacios), si hay muchos anidamientos puede "agotarse" rápidamente la anchura horizontal disponible y tener que partir muchas líneas (ver principio 1.5). Se recomienda usar espacios individuales en vez de tabuladores (que pueden dar problemas si se usan distintos editores en un mismo documento).

Ejemplo 7

```
void seleccion(int v[], int longitud) {
    int i_minimo = 0;

    for (int i = 0; i < longitud - 1; i++) {
        i_minimo = i;
        for(int j = i + 1; j < longitud; j++) {
            if (v[j] < v[i_minimo]) {
                i_minimo = j;
            }
        }
        intercambiar(v[i], v[i_minimo]);
    }
}
```

1.4 Instrucciones o sentencias

a. Poner una sola instrucción por línea.

b. Poner siempre los delimitadores de instrucciones compuestas `{}` aunque haya una única instrucción anidada (ver Ejemplo 7).

PROGRAMACIÓN 1 2023/2024

1.5 Longitud y cortado de las líneas largas

a. El espacio horizontal debe tener una longitud máxima de 80 o 100 caracteres. Las líneas deben partirse adecuadamente dentro de ese espacio para favorecer la legibilidad.

Ejemplo 8

```
num_baldosas = (largo_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA *     MAL
(ancho_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA;

num_baldosas = (largo_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA *     BIEN
               (ancho_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA;

cout << "El cuadrado de " << numero << " es "                             MAL
<< numero * numero << endl;

cout << "El cuadrado de " << numero << " es "                             BIEN
     << numero * numero << endl;
```

1.6 Comentarios

a. Deben aportar información adicional relevante que el código no dé por sí mismo, evitando aquellos superfluos o que expliquen obviedades. En general, el código debe ser auto-comentado mediante una buena elección de identificadores.

Ejemplo 9

```
const int LADO_BALDOSA = 30;  // iniciamos lado baldosa en 30     MAL
const int LADO_BALDOSA = 30;  // medida en centímetros            BIEN
```

b. Los programas deben tener un comentario de cabecera de fichero que incluya, al menos: nombre del fichero fuente, autor y fecha de cada versión.

Ejemplo 10

```
/*
 * baldosas.cpp
 *
 * Versión 1.0 8/11/2023
 * Adolfo Domínguez
 */
```

c. Las funciones deben tener un comentario de cabecera que incluya una muy breve descripción de lo que hacen y, si es el caso, del valor devuelto.

Ejemplo 11

```
/*
 * Guarda notas en un fichero.
 *
 * Devuelve true si ha tenido éxito y false en caso contrario.
 */
bool guardar_notas(const nota notas[], int num_notas) {
  ...
}
```

1.7 Tipo o fuente de letra

a. Para facilitar la lectura del código en los listados emplear tipos de letra de anchura fija (p. e. `Courier New`) frente a las de anchura variable (p. e. Arial).

Ejemplo 12

```
int cuenta_digitos(int numero) {
  int digitos = 1;
  numero = numero / 10;
  while(numero > 0) {
    digitos++;
    numero = numero / 10;
  }
  return digitos;
}
```

Ejemplo 13

```
int cuenta_digitos(int numero) {
  int digitos = 1;
  numero = numero / 10;
  while(numero > 0) {
    digitos++;
    numero = numero / 10;
  }
  return digitos;
}
```

PROGRAMACIÓN 1 2023/2024

2. Diseño

El objetivo principal debe ser desarrollar código fácil de mantener, reutilizar y extender. Un principio muy general de diseño en ingeniería es el denominado KISS (*Keep It Simple, Stupid* o *Keep It Small and Simple*) [3]. Una aplicación de este principio al diseño de código es la filosofía UNIX representada, entre otros, por los principios de: modularidad, claridad, transparencia, economía, robustez y extensibilidad [4].

De manera general, deben seguirse los principios del "Pensamiento Computacional" [5]:

- **Abstraer** los datos y las acciones no relevantes en cada nivel de detalle de la solución.
- **Descomponer** los problemas en otros más pequeños y manejables.
- **Reconocer** problemas resueltos previamente.
- **Escribir** los algoritmos que indiquen los pasos precisos de la solución.

En esta asignatura deberemos seguir los siguientes principios de diseño:

2.1 Resolver primero el problema: análisis y diseño (método y algoritmo). Solo después codificar la solución.

2.2 Entre legibilidad y eficiencia es preferible lo primero.

2.3 No optimizar el código prematuramente. El resultado puede empeorar la legibilidad (ver principio 2.2) y, en caso de optimización con bajo nivel de detalle, producir mejoras poco relevantes.

2.4 No repetir código mediante el método de "Copiar & Pegar", dicho código será más propenso a errores y difícil de mantener. Es mejor escribir funciones (con los argumentos adecuados) que resuelvan un mismo problema una sola vez.

Ejemplo 14

¿Hay errores?

```
cambios = cantidad_introducida - importe;

monedas50cts = cambios / 50;
cambios = cambios - monedas50cts * 50;
cout << monedas50cts << " monedas de 50 cts." << endl;
monedas20cts = cambios / 20;
cambios = cambios - monedas20cts * 20;
cout << monedas20cts << " monedas de 20 cts." << endl;
monedas10cts = cambios / 10;
cambios = cambios - monedas10cts * 10;
cout << monedas10cts << " monedas de 10 cts." << endl;
monedas50cts = cambios / 5;
cambios = cambios - monedas5cts * 5;
cout << monedas5cts << " monedas de 5 cts." << endl;
```

2.5 La mayoría de algoritmos (todos en esta asignatura) están compuestos, en cada nivel de detalle (o abstracción), por unas pocas acciones. Por lo que, las funciones deben poder "verse" completamente en un solo "golpe de vista". Así, con una instrucción por línea (ver principio 1.4.a) pueden ser razonables funciones con hasta 20 líneas. Salvo muy contadas excepciones muchas más líneas indican un mal diseño o código repetido (ver principio 2.4).

2.6 Cada función debe realizar una sola acción (de un nivel de detalle o abstracción) y de la manera más general posible. El identificador de la función debe nombrar de manera clara dicha acción. Una manera de detectar este error es cuando no se sabe qué identificador usar (seguramente porque realiza varias acciones del mismo nivel de detalle). Habitualmente, el mal programador lo resuelve poniendo el nombre compuesto por dichas acciones o, en el peor de los casos, el de una sola de ellas.

2.7 Ocultar los detalles de implementación en funciones (principio de la caja negra).

2.8 No usar variables globales.

2.9 Inicializar siempre las variables al declararlas (ver Ejemplo 7).

PROGRAMACIÓN 1 2023/2024

2.10 Las funciones deber tener un número de argumentos pequeño. Un valor mayor de cuatro implica, normalmente, un mal diseño de los datos (ver principio 2.11).

2.11 Usar vectores y estructuras de manera adecuada para agrupar datos que están relacionados.

Ejemplo 15

MAL

```
void mostrar_tiempo(const int dia, const mes, const año,
                    const hora, const minuto, const int segundo) {
  cout << dia  << SEPARADOR_FECHA << mes << SEPARADOR_FECHA << año << " "
       << hora << SEPARADOR_HORA << mes << SEPARADOR_HORA << segundo << endl;
}
```

MAL

```
void mostrar_tiempo(const int fecha[3], const int hora[3]) {
  cout << fecha[0] << SEPARADOR_FECHA << fecha[1] << SEPARADOR_FECHA << fecha[2] << " "
       << hora[0] << SEPARADOR_HORA << hora [1] << SEPARADOR_HORA << hora[2] << endl;
}
```

BIEN

```
struct fecha {
  int dia;
  int mes;
  int año;
};

struct hora {
  int hora;
  int minuto;
  int segundo;
};

struct tiempo {
  fecha fecha;
  hora hora;
};
...
void mostrar_tiempo(const tiempo tiempo) {
  cout << tiempo.fecha.dia << SEPARADOR_FECHA << tiempo.fecha.mes << SEPARADOR_FECHA
       << tiempo.fecha.año << " "
       << tiempo.hora.hora << SEPARADOR_HORA  << tiempo.hora.minuto << SEPARADOR_HORA
       << tiempo.hora.segundo << endl;
}
```

2.12 No deben realizarse comparaciones con valores `true` o `false` (por ser redundantes).

Ejemplo 16

MAL

```
if (quedan_fondos_disponibles(cuenta) == true) {
  ...
}
```

MAL

```
if (quedan_fondos_disponibles(cuenta) == false) {
  ...
}
```

BIEN

```
if (quedan_fondos_disponibles(cuenta)) {
  ...
}
```

BIEN

```
if ( ! quedan_fondos_disponibles(cuenta)) {
  ...
}
```

2.13 No usar condicional en las funciones que retornen el resultado de una expresión condicional.

Ejemplo 17

MAL

```
if (limite >= LIMITE_IZDA) && (limite <= LIMITE_DCHA) {
  return true;
} else {
  return false;
}
```

BIEN

```
return (limite >= LIMITE_IZDA) && (limite <= LIMITE_DCHA);
```

PROGRAMACIÓN 1 2023/2024

Referencias

[1] Robert C. Martin. *Clean Code. A Handbook of Agile Software Craftsmanship*. Prentice-Hall.
[2] https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants
[3] https://en.wikipedia.org/wiki/KISS_principle
[4] Eric S. Raymond. *The Art of UNIX Programming.* Addison-Wesley Professional Computing Series.
[5] https://en.wikipedia.org/wiki/Computational_thinking
[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.