Algunas buenas prácticas de estilo de programación y diseño de software

1. Estilo

Debe usarse un estilo encaminado a una buena legibilidad del código. Es importante que dicho estilo sea consistente, por este motivo deben seguirse alguna de las guías o convenios establecidos para el lenguaje empleado. Ejemplos de estos son:

https://google.github.io/styleguide/javaguide.html
https://gcc.gnu.org/codingconventions.html
https://google.github.io/styleguide/
https://source.android.com/setup/code-style
https://www.kernel.org/doc/html/v4.10/process/coding-style.html
https://www.python.org/dev/peps/pep-0008/

Principios importantes son:

1.1 Identificadores de variables, constantes simbólicas, funciones/métodos, clases, etc.

a. Usar identificadores pronunciables y expresivos.
b. La longitud de un identificador no es una virtud, lo es su legibilidad.

Ejemplo 1

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

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

c. Si el identificador lo forman varias palabras deben separarse siguiendo la guía de estilo establecida para el lenguaje (C/C++: `'_'`, Java: primera letra en mayúsculas salvo en primera palabra, etc.). Ver Ejemplo 1.
d. Para los identificadores de variables y constantes simbólicas usar sustantivos, salvo para las booleanas donde deben usarse participios.
e. Para los nombres de las clases usar sustantivos.
f. Para los nombres de las funciones/métodos que no devuelvan ningún valor usar infinitivos.
g. No usar abreviaturas, salvo que formen parte de la jerga del problema (p. e. IP, DNS) o sean de uso general.

Ejemplo 2

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

1.2 Constantes simbólicas

a. Usar constantes simbólicas para valores arbitrarios sin significado claro o con múltiples usos (i.e. "números mágicos" [1]).

Ejemplo 3

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

Ejemplo 4

```
#define LIMITE_MAX_DCHA 1345            BIEN
...
if (limite < LIMITE_MAX_DCHA) {
  ...
}
```

b. Para sus identificadores seguir las guías de estilo establecidas por el lenguaje (p. e. C/C++, Java: todo en mayúsculas), incluyendo la separación entre palabras. Ver Ejemplo 4.

1.3 Márgenes adentrados y espacios en blanco

a. Usar adecuadamente espacios en blanco en las expresiones.

Ejemplo 5

```
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/método. Ver Ejemplo 6.

c. Usar márgenes adentrados (4 espacios) para reflejar claramente la estructura del código (i.e. los diferentes niveles de anidamiento).

Ejemplo 6

```
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 Longitud y cortado de las líneas largas

a. Las líneas deben tener una longitud máxima de 80 o 100 caracteres. Las líneas más largas deben partirse adecuadamente.

Ejemplo 7

```
while ((peso == MAX_PESO) && (suma_total < valor_necesario) &&
(valor_necesario <= MAX_PERMITIDO)) {                          MAL
  ...
}
while ((peso == MAX_PESO) &&                                   BIEN
       (suma_total < valor_necesario) &&
       (valor_necesario <= MAX_PERMITIDO)) {
  ...
}
...                                                            BIEN
num_baldosas = (largo_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA *
               (ancho_habitacion + LADO_BALDOSA - 1) / LADO_BALDOSA;
```

1.5 Instrucciones o sentencias

a. Poner una sola instrucción por línea.

b. Poner siempre los delimitadores de instrucciones compuestas (p. e. { } en C/C++, Java) aunque haya una única sentencia anidada. Ver Ejemplo 6.

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 8

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

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

Ejemplo 9

```
/*
 * baldosas.cpp
 *
 * Versión 1.0 8/11/2017
 * Adolfo Domínguez
 */
```

c. Las clases/funciones/métodos 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 10

```
/*
 * 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) {
  ...
}
```

2. Diseño

El objetivo principal debe ser desarrollar código fácil de mantener, reutilizar y extender.

En ingeniería un principio muy general de diseño es el denominado KISS ("*Keep It Simple, Stupid*" o *"Keep It Small and Simple"*) [2]. Una aplicación de este principio al diseño de software es la filosofía UNIX representada, entre otros, por los principios de: modularidad, claridad, transparencia, economía, robustez y extensibilidad [3].

Otros principios generales son los definidos por el denominado "Pensamiento Computacional [4]":

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

Principios más específicos son:

2.1. Resolver primero el problema y luego 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 (principio 2.2) y, en caso de optimización con bajo nivel de detalle, producir mejoras poco relevantes.

2.4. No repetir código mediante *Copy&Paste*, el cual será propenso a errores y difícil de mantener. Escribir librerías/clases o funciones/métodos que resuelvan un mismo problema una sola vez.

2.5. Hacer unidades de código con pocas instrucciones. Con una instrucción por línea (principio 1.5) son razonables funciones/métodos con un máximo de 15 o 20 líneas y librerías/clases de hasta 400 o 500 líneas. Salvo contadas excepciones valores mayores indican una descomposición deficiente y/o código repetido (principio 2.4).

2.6. Cada función/método debe hacer una sola cosa en su nivel de detalle o abstracción, el identificador debe nominar de manera clara dicha acción.

2.7. Las funciones/métodos deber tener una cantidad de argumentos pequeña. Un valor mayor de cuatro implica, normalmente, un mal diseño de datos/objetos (principio 2.11).

2.8. Ocultar los detalles de implementación en librerías/clases y funciones/métodos (principio de la caja negra).

2.9. No usar variables globales, salvo decisión de diseño muy justificada.

2.10. Inicializar siempre las variables al declararlas. Ver Ejemplo 6.

2.11. Usar las estructuras de datos adecuadas (vectores, *structs*, colecciones, etc.) para agrupar datos que están relacionados (principio 2.7).

2.12. No usar excepciones como mecanismo de implementación de la lógica de control. Es poco eficiente y hace el código menos legible.

Ejemplo 11

MAL

```
try {
  usuarios.put(nombre, usuario);
} catch(NullPointerException e){
  usuarios = new Map<String, Usuario>();
  usuarios.put(nombre, usuario);
}
```

BIEN

```
if (usuarios == null) {
  usuarios = new Map<String, Usuario>();
}
usuarios.put(nombre, usuario);
```

De manera específica para la programación orientada a objetos:

2.13. Cada clase debe tener una única responsabilidad.

2.14. Evitar dependencias innecesarias entre clases haciendo que cada una conozca tan poco como sea posible acerca de la estructura y propiedades del resto (principio 2.8).

2.15. Usar polimorfismo en vez de condicionar acciones según la clase de un objeto.

2.16. Entre herencia y composición es preferible lo segundo.

2.17. Usar programación genérica (principio 2.4).

2.18. Identificar y resolver problemas recurrentes mediante patrones de diseño [5].

2.19. Desacoplar las clases siguiendo el patrón arquitectónico MVC (*Model View Controller*).

2.20. Definir, por defecto, como privadas las propiedades o atributos de las clases (principio 2.14).

2.21. Limitar el uso de métodos *getters*/*setters*, los cuales deben ser resultado de decisiones de diseño justificadas sobre qué propiedades deben tener acceso de lectura y/o escritura (principio 2.14).

Referencias

[1] https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants

[2] https://en.wikipedia.org/wiki/KISS_principle

[3] Eric S. Raymond. *The Art of UNIX Programming.* Addison-Wesley Professional Computing Series.

[4] https://en.wikipedia.org/wiki/Computational_thinking

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.