

Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2019

Departamento de Computación - FCEyN - UBA

Buenas prácticas de programación

1

Buenas prácticas de programación

We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such detail that a machine can execute them is programming. Such a specification is code.

(...) code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision –so there will always be code.

Robert Martin, 2009.

2

Buenas prácticas de programación

I know of one company that, in the late 80s, wrote a killer app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased (...) The company went out of business a short time after that.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. It was the bad code that brought the company down.

Robert Martin, 2009.

3

Utilizar nombres declarativos

- ▶ Usar nombres **que revelen la intención** de los elementos nombrados. El nombre de una variable/función debería decir todo lo que hay que saber sobre la variable/función!
 1. Los nombres deben referirse a **conceptos** del dominio del problema.
 2. Una excepción suelen ser las variables con scopes pequeños. Es habitual usar **i, j y k** para las variables de control de los ciclos.
 3. Si es complicado decidirse por un nombre o un nombre no parece natural, quizás es porque esa variable o función no representa un concepto claro del problema a resolver.
- ▶ Evitar la **desinformación!**
 1. Llamar **hp** a la “hipotenusa” es una mala idea.
 2. Tener variables llamadas **cantidadDeElementosEnEIEjeXSinMarcar** y **cantidadDeElementosEnEIEjeYSinMarcar** no es buena idea.

4

Utilizar nombres declarativos

- ▶ Usar nombres **pronunciables**! No es buena idea tener una variable llamada **cdcpdtdc** para representar la "cantidad de cuentas por tipo de cliente".
- ▶ Se debe tener **un nombre por concepto**. No tener funciones llamadas **grabar**, **guardar** y **registrar**.
- ▶ Los nombres de las funciones deben representar el **concepto** calculado, o la **acción** realizada en el caso de las funciones **void**.
- ▶ No hacer **chistes**!

5

No hacer chistes

- ▶ Why does man print "gimme gimme gimme" at 00:30?
- ▶ *We've noticed that some of our automatic tests fail when they run at 00:30 but work fine the rest of the day. They fail with the message "gimme gimme gimme" in stderr, which wasn't expected. Why are we getting this output?*
- ▶ Source:
<https://unix.stackexchange.com/questions/405783/why-does-man-print-gimme-gimme-gimme-at-0030>
- ▶ La canción de Abba: "Gimme gimme gimme a man after midnight"

6

Utilizar nombres declarativos

```
int x = 0;
vector<double> y;
...
for(int i=0; i<=4; i=i+1) {
    x = x + y[i];
}

int totalAdeudado = 0;
vector<double> deudas;
...
for(int i=0; i<=conceptos; i=i+1) {
    totalAdeudado = totalAdeudado + deudas[i];
}
```

7

Indentación (sangrado)

```
int main () {
    int i, j;
    for (i = 0; i <= 10; i++){
        for (j = 0; j <= 10; j++){
            cout << i << " x " << j << " = " << i*j;
        }
    }
    return 0;
}

int main () {
    int i, j;
    for (i = 0; i <= 10; i++){
        for (j = 0; j <= 10; j++){
            cout << i << " x " << j << " = " << i*j;
        }
    }
    return 0;
}
```

8

Code styles

- ▶ Existen distintos estilos de código para C++
- ▶ Ejemplos:
 - ▶ Google
 - ▶ GNU
 - ▶ Stroustup
 - ▶ LLVM
 - ▶ ...etc
- ▶ Se puede configurar la IDE para que re-formatee respetando el estilo elegido.
- ▶ En CLION: Settings - Editor - Code Style - C/C++ - Set from

9

Distintos estilos

GNU C++ Code Style

```
int sumIndexes (int n)
{
    int s = 0;
    int i = 1;
    while (i ≤ n)
    {
        s += i;
        i++;
    }
    return s;
}
```

Google C++ Code Style

```
int sumIndexes(int n) {
    int s = 0;
    int i = 1;
    while (i ≤ n) {
        s += i;
        i++;
    }
    return s;
}
```

10

Comentarios

- ▶ Además de escribir comandos, los lenguajes de programación permiten escribir **comentarios**.
- ▶ Tipos de comentarios en C++:
 - ▶ Comentario de línea: `// ...`
 - ▶ Comentario de bloque: `/* ... */`
- ▶ Es importante hacer un uso adecuado de los comentarios, para que no oscurezcan el código.

11

Comentarios

- ▶ Los comentarios no **arreglan** código de mala calidad! En lugar de comentar el código, hay que clarificarlo.
- ▶ Es importante **expresar las ideas en el código**, y no en los comentarios.
- ▶ Los mejores comentarios son los conceptos que **no se pueden escribir** en el lenguaje de programación utilizado:
 1. Explicar la intención del programador.
 2. Explicitar precondiciones o suposiciones.
 3. Clarificar código que a primera vista puede no ser claro.

12

Comentarios: Malos usos

► Ejemplo 1:

```
/**
 * Funcion que toma dos enteros y devuelve un float.
 */
float calculateModule(int x, int y) {...}
```

► Ejemplo 2:

```
while (x<y) { // itero hasta que x es mayor o igual que y
    ..
}
```

► Ejemplo 3:

```
cout << y; // Imprimo el valor de y
```

13

Comentarios: Buenos usos

► Ejemplo 1:

```
/**
 * Computa el modulo del vector de a partir de
 * dos coordenadas.
 * Las coordenadas deben ser numeros no negativos.
 * Si la precondition no se cumple, retorna -1.
 */
float calculateModule(int x, int y) {...}
```

► Ejemplo 2:

```
// guardamos los Ids en un vector porque no cambian
// durante el algoritmo
vector<s> clientIds;
```

► Ejemplo 3:

```
// el llamado a f se hace siempre con y>0
x = f(y);
```

14

Variables: Inicialización

- En C/C++ podemos tener variables **declaradas pero no inicializadas**, y el valor que contienen en ese caso es impredecible (decimos que contienen “basura”).

- Para evitar esta situación, es recomendable **inicializar siempre** las variables.

► Falta inicializar:

```
int i;
i = 10; // mal
```

► Bien inicializado:

```
int i = 10; // bien
```

15

Variables: Scope de declaración

- Usar el scope más **pequeño** posible!
- Ejemplo: Definición fuera de scope más pequeño posible:

```
int main() { // scope 1
    int t = 0;
    while (...) { // scope 2
        while (...) { // scope 3
            t = 0 ..
        }
    }
}
```

- Ejemplo: Definición en scope más pequeño posible:

```
int main() { // scope 1
    while (...) { // scope 2
        while (...) { // scope 3
            int t = 0 ..
        }
    }
}
```

16

Uso de constantes

- ▶ Si hay un valor que es importante, debe ser declarado como una constante
- ▶ Se utiliza para documentar el sentido del valor
- ▶ Ejemplo: Definición en scope más pequeño posible:

```
int index = find(...);  
if (index==1) {...}
```

- ▶ En realidad, el valor `-1` representa que el valor no ha sido encontrado.

```
#define INDEX_NOT_FOUND -1  
...  
int index = find(...);  
if (index==INDEX_NOT_FOUND) {...}
```

17

Error codes

- ▶ En C++, hay funciones que entre sus parámetros retorna un *error code*
- ▶ Este valor indica como se ejecutó la función
 - ▶ Si el resultado fue correcto
 - ▶ Si el resultado es incorrecto (no se pudo leer el archivo, etc.)
- ▶ Es una buena práctica chequear el resultado en lugar de darlo por correcto
- ▶ En caso de existir un resultado incorrecto, hay que poder recuperar el programa (ej: reintentar lectura)

18

Funciones

- ▶ Las funciones deben ser **pequeñas**! Una función con demasiado código es difícil de entender y mantener.
 1. **Encapsular** comportamiento dentro de funciones auxiliares!
- ▶ **Regla fundamental.** Cada función debe ...
 1. ... hacer **sólo una cosa**,
 2. ... hacerla **bien**, y
 3. ... ser el **único** componente del programa encargado de esa tarea particular.
- ▶ Las funciones no deben tener **efectos colaterales**! En particular, el uso de **variables globales** debe ser particularmente cuidado.

19

Cyclomatic Complexity

- ▶ Es una métrica que se utiliza para aproximar la dificultad para testear un programa
- ▶ Se calcula a partir del control-flow graph
- ▶ $E - N + 2$ donde E es la cantidad de arcos y N es la cantidad de nodos
- ▶ Hay herramientas que la calculan automáticamente
- ▶ Se puede fijar un máximo por proyecto que todos deben respetar
- ▶ La cyclomatic complexity se puede computar automáticamente a partir del control-flow graph, existen herramientas integradas que lo analizan y emiten *warnings* si se ha roto alguna regla.

20

Cognitive Complexity

- ▶ Es una métrica que se utiliza para aproximar la dificultad para que un ser humano entienda un programa
- ▶ Idea:
 - ▶ Code is not considered more complex when it uses shorthand that the language provides for collapsing multiple statements into one
 - ▶ Code is considered more complex for each “break in the linear flow of the code”
 - ▶ Code is considered more complex when “flow breaking structures are nested”

21

Naming Conventions

- ▶ Indican que reglas hay que seguir para nombrar variables, constantes, funciones, archivos, tipos de datos, etc.
- ▶ Idea: evitar uso inconsistente
 - ▶ `firstindexwithvalue`
 - ▶ `first_index_with_value`
 - ▶ `firstIndexWithValue`
 - ▶ `FIRST_INDEX_WITH_VALUE`
 - ▶ `FirstIndexWithValue`
- ▶ Google C++ variable naming convention: “The names of variables (including function parameters) are all lowercase, with underscores between words.”

22

Formato vertical

- ▶ Los **archivos** del proyecto no deben ser demasiado grandes!
- ▶ Conceptos relacionados deben aparecer verticalmente cerca en los archivos.
- ▶ Las funciones más importantes deben estar en la parte superior, seguidas de las funciones auxiliares.
 1. Siempre la **función llamada** debe estar debajo de la **función llamadora**.
 2. Las funciones menos importantes deben estar en la parte inferior del archivo.
- ▶ Se suele equiparar un archivo de código con un **artículo periodístico**. Debemos poder interrumpir la lectura en cualquier momento, teniendo información acorde con la lectura realizada.

23

Modularización

- ▶ Cada archivo del proyecto debe tener **código homogéneo**, y relacionado con un concepto o grupo de conceptos coherentes.
- ▶ Los **nombres de los archivos** también deben ser representativos!
- ▶ Muchos lenguajes de programación dan facilidades para organizar archivos en **paquetes** (o conceptos similares), para tener una organización en más de un nivel.

24

Modularización

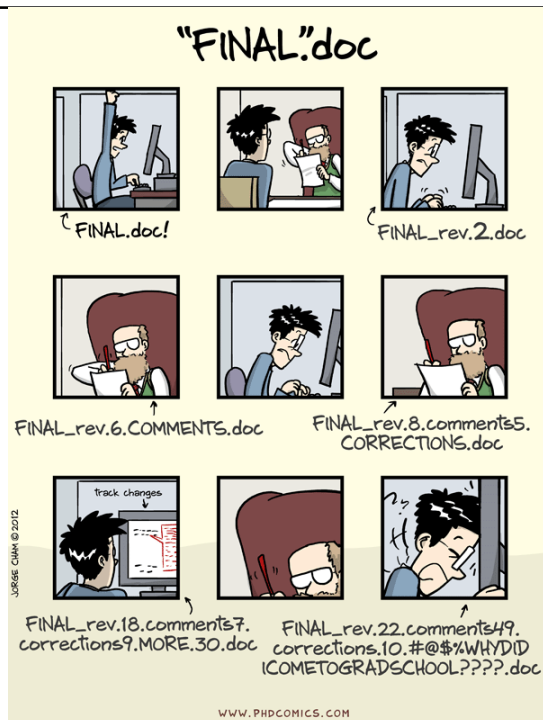
- **Single responsibility principle:** Cada función debe tener un **único motivo de cambio**.
 1. Interfaz de usuario.
 2. Tecnología para la interfaz de usuario.
 3. Lógica de negocio.
 4. Consideraciones sobre los algoritmos.
 5. Almacenamiento permanente.
 6. ...
- El código responsable de cada uno de estos aspectos debe estar **separado** del resto!

25

Al momento de escribir código ...

- **Desarrollo incremental:** Escribir **bloques pequeños y testeables** de funcionalidad, teniendo en todo momento una aplicación (parcialmente) funcional.
 1. Esto implica conocer en todo momento el objetivo del código que estamos escribiendo!
 2. Si se usa un repositorio de control de versiones, el **próximo commit** debe estar bien definido.
- No **incorporar expectativas en las estimaciones**. Si vemos que los plazos no se van a poder cumplir, reaccionar rápidamente.
- Para el código más crítico, recurrir a **pair programming**: dos personas juntas sobre una única computadora.
- **Descansar!** Es importante destinar tiempo de descanso entre las sesiones de código.

26



Version Control Systems (CVSs)

- Permite organizar el trabajo en equipo
- Guarda un historial de versiones de los distintos archivos que se usaron
- Existen distintas aplicaciones: svn, cvs, hg, git

28