Buenas prácticas de programación

Algoritmos y Estructuras de Datos I

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.

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.

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 **cantidadDeElementosEnElEjeXSinMarcar** y **cantidadDeElementosEnElEjeYSinMarcar** no es buena idea.

Utilizar nombres declarativos

- Usar nombres pronunciables! No es buena idea tener una variable llamada **cdcptdc** 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!

Ejemplo: Utilizar nombres declarativos

```
1 int x = 0;
2 vector<double> y;
3 ...
4 for(int i=0;i<=4;i=i+1) {
5  x = x + y[i];
6 }
```

```
1 int totalAdeudado = 0;
2 vector<double> deudas;
3 ...
4 for(int i=0;i<=conceptos;i=i+1) {
5  totalAdeudado = totalAdeudado + deudas[i];
6 }
```


Indentación (sangrado)

```
1 int main () {
2 int i, j;
3 for (i = 0; i <= 10; i++){
4 for (j = 0; j <= 10; j++){
5 cout << i << " x  " <<  j << " = " << i*j;
6 }
7 }
8 return 0;
9 }
```

```
1 int main () {
2   int i, j;
3   for (i = 0; i <= 10; i++){
4     for (j = 0; j <= 10; j++){
5       cout << i << " x  " <<  j << " = " << i*j;
6     }
7   }
8   return 0;
9 }
```


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.

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.

Comentarios: Malos usos

- Ejemplo 1:

```
1 /**
2   * Funcion que toma dos enteros y devuelve un float.
3   */
4 float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

```
1 while (x<y) { // itero hasta que x es mayor o igual que y
2   o ..
3 }
```

- Ejemplo 3:

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


Comentarios: Buenos usos

- Ejemplo 1:

```
1 /**
2   * Computa el modulo del vector de a partir de dos
        coordenadas.
3   * Las coordenadas deben ser numeros no negativos.
4   * Si la precondicion no se cumple, retorna -1.
5   */
6 float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

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

- Ejemplo 3:

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


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:

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

- Bien inicializado:

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

Variables: Scope de declaración

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

```
1 int main() { // scope 1
2   int t = 0;
3   while (...) { // scope 2
4     while (...) { // scope 3
5       t =○ ..
6     }
7   }
```

- Ejemplo: Definición en scope más pequeño posible:

```
1 int main() { // scope 1
2   while (...) { // scope 2
3     while (...) { // scope 3
4       int t =○ ..
5     }
6   }
```


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.

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.

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.

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!

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.

Bibliografía

- Robert Martin, *Clean code*. Prentice-Hall, 2009.
- Steve McConnell, *Code complete*. Microsoft Press, 1993.