Metodología de la programación

Ciclo de vida del software

Fases del ciclo de vida: Análisis, diseño, implementación...

Programación estructurada

Cuestiones de estilo

Identificadores
Constantes
Expresiones
Declaraciones
Estructuras de control
Comentarios
Convenciones

Bibliografía

- ✓ Steve McConnell: “Code Complete”.
  Estados Unidos: Microsoft Press, 1994. ISBN 1-55615-484-4.

- ✓ Brian W. Kernighan & Rob Pike: “La práctica de la programación”.
  México: Pearson Educación, 2000. ISBN 968-444-418-4.

- ✓ Francisco J. Cortijo, Juan Carlos Cubero & Olga Pons: “Metodología de la Programación”. Granada: Proyecto Sur, 1993. ISBN 84-604-7652-9.

- ✓ “C Style Guide”.
  NASA Software Engineering Laboratory, August 1994. SEL-94-003

Ciclo de vida del software

El ciclo de vida de una aplicación comprende las siguientes etapas:

- ✓ Planificación: ámbito del proyecto, estudio de viabilidad, análisis de riesgos, planificación temporal, asignación de recursos.
- ✓ Análisis (¿qué?): elicitación de requerimientos
- ✓ Diseño (¿cómo?): estudio de alternativas, diseño arquitectónico
- ✓ Implementación: adquisición, creación e integración de los recursos necesarios para que el sistema funcione.
- ✓ Pruebas: pruebas de unidad, pruebas de integración, pruebas alfa, pruebas beta, test de aceptación.
- ✓ Mantenimiento (correctivo y adaptativo)

Programación estructurada

IDEA CENTRAL: Las estructuras de control de un programa sólo deben tener un punto de entrada y un punto de salida.

La programación estructurada...
mejora la productividad de los programadores.
mejora la legibilidad del código resultante.

La ejecución de un programa estructurado progresa disciplinadamente, en vez de saltar de un sitio a otro de forma impredecible

En programación estructurada sólo se emplean tres construcciones:

- ✓ Secuencia
  Conjunto de sentencias que se ejecutan en orden
  (asignaciones y llamadas a rutinas)

- ✓ Selección
  Estructura de control condicional
  (if-then-else, case/switch)

- ✓ Iteración
  Estructura de control repetitiva
  (bucles: while, do...while, for)

Teorema de Böhm y Jacopini (1966)

Cualquier programa de ordenador puede diseñarse e implementarse utilizando únicamente las tres construcciones estructuradas (secuencia, selección e iteración; esto es, sin sentencias `goto`).

Böhm, C. & Jacopini, G.: "Flow diagrams, Turing machines, and languages only with two formation rules". Communications of the ACM, 1966, Vol. 9, No. 5, pp. 366-371

Dijkstra, E.W.: "Goto statement considered harmful". Communications of the ACM, 1968, Vol. 11, No. 3, pp. 147-148

Cuestiones de estilo

IDEA

Escribimos código para que lo puedan leer otras personas, no sólo para que lo traduzca el compilador (si no fuese así, podríamos seguir escribiendo nuestros programas en binario).

Identificadores

- Selección de los identificadores: Los identificadores deben ser **descriptivos** (reflejar su significado).

  ✗ `p, i, s...`

  ✓ `precio, izquierda, suma...`

- Uso de minúsculas y mayúsculas: Generalmente, resulta más cómodo leer texto en minúsculas, por lo que usaremos siempre identificadores en minúsculas, salvo:
  - para construir identificadores compuestos (p.ej. `otraVariable`) y
  - para definir constantes simbólicas y macros (con `#define`).

- En ocasiones, se permite el uso de nombres cortos para variables locales cuyo significado es evidente (p.ej. bucles controlados por contador)

  ✓
  ```
  for (elemento=0; elemento<N; elemento++)
      ...
  ```

  ✓
  ```
  for (i=0; i<N; i++)
      ...
  ```

Constantes

- Se considera una mala costumbre incluir literales de tipo numérico (**"números mágicos"**) en medio del código. Se prefiere la definición de constantes simbólicas (con `#define` o, mejor aún, con `enum`).

  ✗ `for (i=0; i<79; i++)...`

  ✓ `for (i=0; i<columnas-1; i++)...`

Expresiones

- Uso de paréntesis: Aunque las normas de precedencia de los operadores están definidas por el estándar de C, no abusaremos de ellas. Siempre resulta más fácil interpretar una expresión si ésta tiene los paréntesis apropiados. Además, éstos eliminan cualquier tipo de ambigüedad.

- Uso de espacios en blanco: Resulta más fácil leer una expresión con espacios que separen los distintos operadores y operandos involucrados en la expresión.

  ✗
  ```
  a%x*c/b-1
  ```

  ✓
  ```
  ( (a%x) * c ) / b - 1
  ```

- Expresiones booleanas:
  Es aconsejable escribirlas como se dirían en voz alta.

  ✗
  ```
  if ( !(bloque<actual) ) ...
  ```

  ✓
  ```
  if ( bloque >= actual ) ...
  ```

- Expresiones complejas:
  Es aconsejable dividirlas para mejorar su legibilidad (p.ej. operador `?:`).

  ✗
  ```
  x += ( xp = ( 2*k < (n-m) ? c+k: d-k ));
  ```

  ✓
  ```
  if ( 2*k < n-m )
     xp = c+k;
  else
     xp = d-k;

  x += xp;
  ```

  ✓
  ```
  max = (a > b) ? a : b;
  ```

- **Claridad**:
  Siempre buscaremos la forma más simple de escribir una expresión.

  ✗
  ```
  key = key >> ( bits – ((bits>>3)<<3));
  ```

  ✓
  ```
  key >>= bits & 0x7;
  ```

- Conversiones de tipo (castings):
  Evitaremos las conversiones implícitas de tipo. Cuando queramos realizar una conversión de tipo, lo indicaremos explícitamente.

  ✓ `i = (int) f;`

Declaraciones

- Usualmente, declararemos una única variable por línea.

- Nunca mezclaremos en una misma línea la declaración de variables que sean de distintos tipos o que se utilicen en el programa para distintos fines.

  ✗ `int i, datos[100], v[3], ok;`

Estructuras de control

- Sangrías: Conviene utilizar espacios en blanco o separadores para delimitar el ámbito de las estructuras de control de nuestros programas.

- Líneas en blanco: Para delimitar claramente los distintos bloques de código en nuestros programas dejaremos líneas en blanco entre ellos.

Comentarios

- Comentarios descriptivos: Los comentarios deben comunicar algo. Nunca han de limitarse a decir en lenguaje natural lo que ya está escrito en el código. Jamás se utilizarán para "parafrasear" el código y repetir lo que es obvio.

  ✗
  ```
  i++;    /* Incrementa el contador */
  ```

  ✓
  ```
  /* Recorrido secuencial de los datos */
  for (i=0; i<N; i++) ...
  ```

  ✗
  ```
  int mes; /* Mes */
  ```

  ✓
  ```
  int mes; /* Mes del año (1..12) */
  ```

- Comentarios de prólogo: Al principio de cada módulo han de incluirse comentarios que resuman la tarea que realiza el módulo y su interfaz (parámetros...).

```
// Cálculo del MCD
// utilizando el algoritmo de Euclides
//
// Parámetros: Los números ‘a’ y ‘b’
// Resultado: Máximo común divisor de ‘a’ y ‘b’
```

- Usualmente, también se incluyen comentarios que aclaren los algoritmos y técnicas utilizados en la implementación del programa.

- Al comienzo de cada fichero de código, se suele incluir un comentario en el que aparece el autor del módulo y una lista de las revisiones que ha sufrido a lo largo de su vida.

```
// Revisor ortográfico
// © Fernando Berzal, 2003
//
// Revisiones:
// v2.0 Nov’03 Análisis de sufijos
// v1.1 Oct’03 Diccionario mejorado
// v1.0 Oct’03 Versión inicial
```

- No comente el código “malo” (uso de construcciones extrañas, expresiones confusas, sentencias poco legibles...): Reescríbalo.

- No contradiga al código: Los comentarios suelen coincidir con el código cuando se escriben, pero a medida que se corrigen errores y el programa evoluciona, los comentarios suelen dejarse en su forma original y aparecen discrepancias. Si cambia el código, asegúrese de que los comentarios sigan siendo correctos.

- Los comentarios han de aclarar; esto es, ayudar al lector en las partes difíciles (y no confundirle). Si es posible, escriba código fácil de entender por sí mismo: cuanto mejor lo haga, menos comentarios necesitará.

✘
```
ax = 0x723;   /* RIP L.v.B. */
```

NB: Beethoven murió en 1827 (0x723 en hexadecimal).

Convenciones

- Siempre se han de evitar los efectos colaterales (modificaciones no deseadas que pueden afectar a la ejecución del programa):

  - ```
    #define MAX(a,b) (((a)>(b))?(a):(b))
    ...
    k = MAX(i++,j++);
    ```

  - ```
    scanf ("%d %d", &pos, &v[pos]);
    ```

- Salvo en la cabecera de los bucles `for`, sólo incluiremos una sentencia por línea de código.

- Sean cuales sean las convenciones utilizadas al escribir código (p.ej. uso de sangrías y llaves), hay que ser consistente en su utilización.

```
while (...) {
   ...
}

for (...) {
   ...
}

if (...) {
   ...
}
```

```
while (...)
{
   ...
}

for (...)
{
   ...
}

if (...)
{
   ...
}
```

El código bien escrito es más fácil de leer, entender y mantener (además, seguramente tiene menos errores)