UOC
Universitat Oberta de Catalunya
www.uoc.edu

guía de estilo

Programación en C

Profesorado FP
Última actualización: Septiembre, 2009

Índice

1 Introducción

El objetivo de la presente guía es fijar una serie de pautas de programación en C para la asignatura Fundamentos de Programación.

Cabe destacar que actualmente no existe un “estándar” en estilo de programación en C. Sin embargo, consideramos que es necesario establecer unas normas comunes a seguir en la escritura de programas ya que éstas:

- Ayudarán a los estudiantes a adquirir un buen estilo inicial de programación, evitando de esta forma coger malos hábitos.
- Reducirán el número de errores. Recordar que es preferible detectar errores en tiempo de compilación/link-edición que durante la ejecución.
- Generarán un código más legible, tanto para el propio autor del código como para otras personas que no sean las autoras (por ejemplo, los profesores).

Es importante aclarar que es posible que los profesores de la asignatura tengan adoptadas normas de programación ligeramente diferentes (en parte o en su totalidad) a las aquí presentadas. Esto se puede apreciar en las soluciones propuestas de ejercicios de examen o de prácticas. En cualquier caso, a efectos de la corrección, las normas que se tendrán en cuenta serán las de este documento.

2 Consideraciones generales

2.1 Fichero fuente

Los programas escritos en C tienen extensión **`.c`**.

2.2 Estructura

Un fichero de código C sigue la siguiente estructura general:

1. Cabecera. Todos los ficheros deben comenzar con un comentario que contenga: el nombre del fichero, el autor, la fecha de creación y una breve descripción del contenido.
2. Includes. Ficheros de cabecera (**`.h`**) precedidos por la directiva **`#include`**. Primero se sitúan los del sistema y luego los propios de la aplicación (si los hubiera).
3. Definición de constantes simbólicas (**`#define`**) y tipos definidos por el usuario (**`typedef`**).
4. Pre-declaración. Definición de prototipos de acciones y funciones locales.
5. Main. En caso de existir, la implementación de la función **`main`**.
6. Implementación de funciones/acciones. Dentro de cualquier función/acción (incluida la función `main`), en primer lugar se deben declarar las variables locales y, a continuación, el cuerpo de la función/acción.

Ejemplo 1:

```
/*
** File:          Example.c
** Author:        Nombre Apellido
** Date:          01-08-2009
** Description:   Saludo universal
*/

/* System header files */
#include <stdio.h>

/* Application header files */
#include “example.h”

/* Symbolic constants */
#define  MAX  10

/* User defined types */

/* Pre-declaration of actions and functions */

/* Main function */
int main() {
    printf(“\n Hola, mundo!”);
    return 0;
}

/* Implementation of actions and functions */
```

3 Definición de constantes

En general, los valores numéricos no deben ser usados directamente en el código, ya que dan lugar a lo que se conoce como “números mágicos”. Valores como *3.14159* o *42* no son autodescriptivos y son además difíciles de mantener. Por este motivo, es más adecuado utilizar en constantes con nombres simbólicos.

Para la definición de las constantes, se recomienda utilizar **`const`** en lugar de **`define`**. Si bien **`const`** hace que la constante ocupe un espacio a diferencia de **`define`**, el compilador realizará chequeo de tipos minimizando la posibilidad de error, mientras que **`define`** no lo hará.

Ejemplo 2:

```
#define  PI  3.14159
#define  ANSWER 42
```

o

```
const float PI = 3.14159;
const int ANSWER = 42;
```

```
if (n > PI/2)
    printf(“\n Correct”);
```

es mejor que:

```
if (n > 3.14159/2)
    printf(“\n Correct”);
```

4 Nombres

Para identificar los elementos del código (constantes, tipos, funciones, acciones y variables) se recomienda escoger nombres que sean auto-descriptivos. Para el caso de las variables, los nombres cortos aumentan su legibilidad y facilidad de uso.

Si un nombre está formado por varias palabras, éstas se han de separar utilizando el símbolo “_”. El resto son mayúsculas, minúsculas o números con las siguientes convenciones:

- Los nombres de **constantes** se escriben exclusivamente con letras mayúsculas.

  Ejemplo 3:

  ```
  #define MAX_SUBJECTS 30
  const int MAX = 1000;
  ```

- Los nombres de los **tipos definidos por el usuario** comienzan habitualmente con la letra “t”, y van seguidos inmediatamente de las palabras que describen el tipo.

  Ejemplo 4:

  ```
  typedef int tVector[MAX];
  typedef struct {
      int idSubject;
      int numberOfStudents;
      int room;
  } tSubject;
  typedef tSubject tSubjectsSet [MAX_ SUBJECTS];
  ```

- Los nombres de **funciones**, **acciones** y **variables** se escriben exclusivamente con letras minúsculas.

  Ejemplo 5:

  ```
  void deleteSubject(int id, tSubjectsSet subjects);
  ```

Se recomienda evitar el uso de nombres negados para las **variables** de tipo lógico (`bool`).

Ejemplo 6:

```
  bool found = FALSE;
```

es mejor que:

```
  bool notFound = TRUE;
```

Para las **funciones** o **acciones**, se aconseja seguir las siguientes convenciones:

- Incluyen verbos en forma imperativa.

  Ejemplo 7:

```
void sort(tVector v);
void deleteRepeatedElements(tVector v);
```

- Las funciones de tipo lógico (`bool`) comienzan habitualmente por "`is`".

  Ejemplo 8:

```
bool isOdd(int n);
bool isEmpty(tVector v);
```

  Existen también otras posibilidades:

```
bool belongs(int a, tVector v);
```

5 Consideraciones del código

A continuación se describen las normas y recomendaciones para organizar correctamente el código:

- Cada línea de código contiene una única instrucción básica:

  Ejemplo 9:

```
i = 0;
j = 1;
```

  es mejor que:

```
i = 0;  j = 1;
```

- Si alguna de las tres partes de la construcción de un **`for`** está vacía, o si es necesario modificar la variable de control de un **`for`** en el cuerpo del bucle, entonces es mejor usar un **`while`**.

  Ejemplo 10:

```
/* Returns the first position of v that contains x or -1 if v does not
   contain x */
int searchPosition(int x, tVector v) {
    bool found = FALSE;
    int i = 0;
    while (!found && i < MAX) {
```

```
        if (v[i] == x)
            found = TRUE;
         else
            i++;
    }
    if (found)
        return i;
    else
        return -1;
}
```

es mejor que:

```
int searchPosition(int x, tVector v) {
    bool found = FALSE;
    int i = 0;
    for (; (!found && i < MAX); ) {
        if (v[i] == x)
            found = TRUE;
         else
            i++;
    }
    if (found)
        return i;
    else
        return -1;
}
```

- Evitar escribir códigos de funciones o acciones muy largos. En cuyo caso, se aconseja crear otra función o acciones con parte del código, aumentando de ésta forma su legibilidad. Por otro lado, se recomienda evitar demasiadas llamadas anidadas.

- Evitar comparaciones de igualdad o desigualdad entre números reales, especialmente si éstos son resultado de una serie de operaciones. Esto es debido a que pequeños errores de redondeo podrían hacer fallar el programa.

- Una variable de tipo lógico, no se debe comparar con *verdadero* o *falso* directamente.

  Ejemplo 11:

  ```
  if (found) …
  if (!found) …
  ```

  es mejor que :

  ```
  if (found == TRUE) …
  if (!found == FALSE) …
  ```

- En esta asignatura, consideraremos que las **variables globales** están prohibidas

- Los **parámetros** de una acción se deben colocar en éste orden: entrada, entrada/salida, salida.

- Sentencia **`switch`**: La sentencia **`break`** debe incluirse salvo que se desee comportamiento “en cascada”, el cual debe comentarse claramente.

  Ejemplo 12:

```
switch (condition) {
case VALUE_1:
case VALUE_2: /* Equal to VALUE_1 */
               statement;
               break;
case VALUE_3:
              statement;
              break;
}
```

6 Líneas en blanco

Es conveniente separar con líneas en blanco las diferentes partes (ficheros de cabecera, constantes, tipos, acciones y funciones) de un fichero C (ver Ejemplo 1). Dentro de cada parte es conveniente a su vez separar cada elemento con el mismo número de líneas en blanco, salvo los tipos definidos por el usuario que se escriben sin separación (ver Ejemplo global, apartado 12).

En los casos en que el trozo de código es largo, se recomienda colocar una línea en blanco que separe bloques conceptuales.

7 Indentación

Se aconseja indentar con cuatro espacios por nivel. Evitar el uso de tabuladores, configurar el editor para desactivarlos.

Las líneas con más de 78 caracteres se recomienda cortarlas dado que la impresión de un código con líneas más largas no se verá correctamente.

Si un trozo de código tiene demasiadas indentaciones, es aconsejable utilizar una función/acción auxiliar.

8 Espacios

Como norma general, se aconseja dejar un espacio de separación entre palabras clave, instrucciones, literales, etc. Las excepciones más importantes son:

- Los punto y coma deben estar enganchados a lo que tengan a la izquierda (ver Ejemplo 13).
- La parte interna de un paréntesis no se separa de aquello que la contenga (ver Ejemplo 14).
- El paréntesis izquierdo de una función/acción se escribe enganchado a su nombre (ver Ejemplo 14).
- Las llaves para acceder a una posición de un vector o `string` no contienen espacios (ver Ejemplo 14).

- Los operadores binarios *, /, %, - (cambio de signo) se escriben sin espacio para enfatizar su precedencia.

  Ejemplo 13:

```
int a;
a = x + y*z;
```

  es mejor que:

```
a = x + y * z;
```

  y que:

```
a = x+y*z;
```

  También:

```
a = -b;
```

  es mejor que:

```
a = - b;
```

- Los operadores ++ (incremento) y –- (decremento) no llevan espacio.

- El carácter & (para indicar paso por referencia) se escribe inmediatamente a la derecha del tipo al cual está asociado (ver Ejemplo global, apartado 12).

  Ejemplo 14: Este ejemplo incluye varios puntos antes mencionados:

```
int anotherFunction(tVector v) {
    int i;
    int j = 0;
    int a;
    for (i = MAX – 5; i >= -4; i--) {
        a = i*i + factorial(a) + v[j];
        j++;
    }
   return a;
}
```

9 Llaves

La variedad de formas de colocación de las llaves es muy amplia. En ésta asignatura se recomienda la siguiente forma:

```
if (condition1) {
    …
}

if (condition2) {
    …
}
else {
    …
}

while (condition3) {
    …
}

for (inicialization;  condition;  “increase”) {
    …
}
```

Como se puede observar, no se coloca nada a la derecha de una llave, ya sea la que abre como la que cierra.

En el caso de **`if`**'s anidados, donde las condiciones son excluyentes entre si (es decir, que será necesario evaluarlas una tras la otra (estructura del tipo: *if else if else if ..*)) se recomienda escribirlo de la siguiente forma:

```
if (condition1) {
    …
}
else if (condition2) {
    …
}
else if (condition3) {
   …
}
…
else {
      …
}
```

Ejemplo 15:

```
char input[MAX];
…
/* c corresponds to each char of the input */
if (c >= ‘a’ && c <= ‘z’) {
    printf(“\n lower case letter”);
```

```
}
else if (c >= ‘A’ && c <= ‘Z’) {
    printf(“\n capital letter”);
}
else if (c >= ‘0’ && c <= ‘9’) {
    printf(“\n digit”);
}
else {
    printf(“\n unknown”);
}
```

10 Paréntesis

Tal como se ha comentado anteriormente:

- Las palabras clave se separan de los paréntesis.

  Ejemplo 16:

  ```
  if (condition) ...
  ```

- La parte interna de un paréntesis no se separa de aquello que contenga.

  Ejemplo 17:

  ```
  for (i = 0; i <= MAX; i++) …
  ```

- El paréntesis izquierdo de una función o acción se escribe a continuación de su nombre.

  Ejemplo 18:

  ```
  int anotherFunction(tVector v) …
  ```

- No se colocan paréntesis alrededor de lo que devuelve una función.

  Ejemplo 19:

  ```
  return found;
  ```

  es mejor que :

  ```
  return (found);
  ```

11 Comentarios

Es necesario documentar el código en la justa medida. En concreto, es necesario explicar en pocas líneas lo que hace cada función/acción. En los casos en los que no sea trivial como se hace, se aconseja agregar un

comentario breve explicándolo.

Ejemplo 20:

```
/* Return n! */
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n*factorial(n – 1);
}
```

Colocar comentarios redundantes puede ser tan pernicioso como omitir comentarios importantes tal como se muestra en el ejemplo siguiente.

Ejemplo 21:

```
i++ /* Increments the value of i in one unit */
```

12 Ejemplo global

A continuación se muestra un ejemplo que ilustra gran parte de las normas vistas a lo largo de este documento:

```
/*
** File:            Example.c
** Author:          Profesorado FP
** Date:            01-08-2009
** Description:     Ejemplo global
*/

/* System header files */
#include <stdio.h>
#include <stdlib.h>

/* Symbolic constants */
#define END_SEQ    -1

/* User defined types */
typedef enum {FALSE, TRUE} bool;
typedef int tNatural;

/* Pre-declaration of actions and functions */
bool isEvenNumber(tNatural num);

/* Main function */
/* Pre:The standard input contains a sequence of natural numbers ending with -1 */
int main() {
    tNatural num;
    tNatural sum = 0;
```

```
    /* Read natural number */
    scanf("%d",&num);

    while (num != END_SEQ) {
        if (isEvenNumber(num))
            sum = sum + num;
        scanf("%d",&num);
    }
    printf("%d ", sum);
    return 0;
}
/* Pos: The standard output shows the sum of the even numbers */
/* Implementation of actions and functions */
bool isEvenNumber(tNatural num) {
    if (num%2 == 0)
        return TRUE;
    else
        return FALSE;
}
```

13 Referencias

- *Consejos y normas de estilo para programar en C*. http://trajano.us.es/docencia/FundamentosDeLaProgramacion/04-05/documentos/guiaEstilo.pdf

- Kernighan, B., Ritchie, D. *The C programming Language*. http://www.e-booksdirectory.com/details.php?ebook=2367

- Mañas, José A.. *Estilo de programación*. http://www.lab.dit.upm.es/~lprg/material/apuntes/doc/estilo.htm

- Roura, S. *Normes de programació de P1* (2007). http://www.lsi.upc.edu/~prap/prap/normesP1.pdf