3. Programación en C

Fundamentos de Informática

Especialidad de Electrónica – 2009-2010

Ismael Etxeberria Agiriano

07/10/2009

Escuela Universitaria de Ingeniería Vitoria-Gasteiz
Ingeniaritzako Unibertsitate Eskola Vitoria-Gasteiz

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Índice

3. Programación en C

1. Introducción

- **Problema**
  - El **analista** estudia el problema y produce las especificaciones o pliego de condiciones
- **Pliego de condiciones**
  - El **diseñador** estudia las especificaciones de qué debe hacer el programa y produce los algoritmos, por ejemplo, en forma de diagramas de flujo
- **Algoritmos**
  - El **programador**, partiendo de los algoritmos (que son los "planos del programa") codificará el programa
- **Programa**
  - El **usuario final** utilizará el programa tantas veces como desee, aportando datos y obteniendo resultados

Codificación en un lenguaje de programación

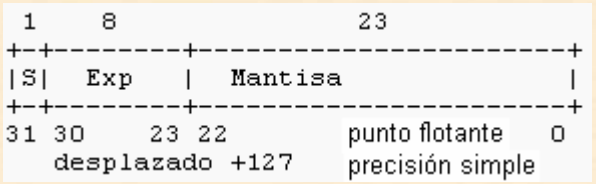
- **Léxico**
  - Normas para construir las palabras correctamente
- **Sintaxis**
  - Normas para escribir las frases correctamente
- **Semántica**
  - Significado de las frases o sentencias
- **Estilo**
  - Prácticas que no afectan al programa final pero sí a la programación y el mantenimiento de programas

2. Léxico

- Literales
- Palabras reservadas
- Identificadores
- Comentarios
- Operadores
- Separadores y terminadores
- Directivas del preprocesador

- **Literales**
  - Los literales se utilizan para expresar **valores constantes**
  - Un literal expresa directamente un valor de un *tipo* dado
    - Entero
    - Real
    - Carácter
    - Cadena de caracteres
  - La codificación a la hora de guardar un dato depende del tipo de un dato
  - Los literales tienen un tipo implícito aunque pueden sufrir una trasformación inmediata en una asignación a una variable de otro tipo

- **Literales enteros**
  - **Base 10**: positivos y negativos
    - `9210 -32767`
  - **Base 8**: anteponemos el cero **0**
    - `040 073`
  - **Base 16**: anteponemos **0x** ó **0X**
    - `0x1a40 0xFF7F`
- **Literales reales**
  - Con **punto** (coma) decimal
    - `1.  .32  3.14159265358979`
  - **Notación científica**, *mantisa* E *exponente* ≡ *mantisa*·$10^{exponente}$
    - `1.9E-39  6.02214179e23` (norm.: primer dígito significativo y luego punto)



- **Caracteres literales**
  - Entre **comillas simples**
    - `'H'  ':'  ' '  '3'`
  - Formatos especiales: códigos de **control**, letras en **octal** y en **hexadecimal**
    - `'\n'  '\016'  '\x7f'`
- **Cadenas literales**
  - Entre **comillas dobles**
    - `"Hora: "  "\n\t1\n\t2"  "%d: %s"`

- **Palabras reservadas**
  - Son las “conocidas”, las que tienen un significado especial para el compilador
  - Las mostraremos en negrita
  - En papel las subrayaremos
- **Ejemplos**

```
int return char void const while if break
  double case continue default do else float
  for long short sizeof switch unsigned
```

- **Más ejemplos**

```
auto enum extern goto register signed static
  struct typedef union volatile
```

- **Identificadores**
  - Nombres utilizados para datos (constantes y variables), tipos de datos definidos, funciones, etc.
- **Formación**
  - Empiezan obligatoriamente por un carácter alfabético inglés `a-z A-Z` o por el carácter `_`
  - Secuencias compuestas por caracteres alfabéticos ingleses `a-z A-Z` y/o numéricos `0-9` y/o el carácter `_`
  - No pueden ser palabras reservadas
  - Distingue entre mayúsculas y minúsculas
- **Ejemplo**

```
res ≠ Res ≠ RES ≠ _RES
```

- **Comentarios**
  - Texto que se introduce en el código fuente para explicar el programa
  - Son ignorados por el compilador
  - No generan código ejecutable
  - Empiezan por `/*` y terminan por `*/`
  - Cuidado con los anidamientos
  - En **C++** se permiten comentarios “para el resto de la línea”, que comienzan por `//`. En C no existen pero muchos compiladores lo permiten.
- **Ejemplos**

```
/* Solo tener en cuenta
 * balances positivos */
int pos;  // Contador de posición
```

- **Operadores**
  - Son los que expresan las operaciones como cambio de signo, suma, resta, multiplicación, división, comparaciones, desplazamientos, …
  - El más importante es la asignación
  - Según el número de operandos pueden ser unarios, binarios, ternarios, …
- **Ejemplos**

```
+   -   *   /   >   >=   ==   !=   <   <=  ?:
!   &  &&  |  ||  ^   ~   >>  <<
=   +=   *=   /=   >>=   &=   |=   ^=
```

- **Separadores y terminadores**
  - Sirven para separar las unidades léxicas y para hacer los programas más legibles
- **Ejemplos**
  - Punto y coma
  - Espacio
  - Salto de línea
  - Tabulador

- **Directivas del preprocesador**
  - El preprocesador es un proceso previo al compilador
  - El compilador ve el código tras las transformaciones del preprocesador
  - Comienzan por el símbolo `#` (almohadilla)
- **Ejemplos**
  - Macrodefiniciones
    - `#define N_Avogadro 6.02214179e23`
  - Inclusiones de ficheros
    - `#include <stdio.h>`

3. Sintaxis

- Un programa se compone de frases que serán declaraciones e instrucciones
- Las instrucciones y declaraciones terminan siempre en ;
- Un espacio fuera de un literal (por ejemplo, que no esté dentro de una cadena) es intercambiable por cualquier combinación de uno o más espacios, saltos de línea y tabuladores (llamados “caracteres blancos”)
- Construcciones básicas
  - Declaraciones de constantes y variables
  - Expresiones: prioridad y asociatividad
  - Asignación
  - Instrucciones de entrada/salida

- **Declaraciones de constantes y variables**
  - Un **dato** es una celda de memoria que ocupa uno o más bytes
  - Un dato tiene **tres características**:
    - **Nombre**: identificador que lo distingue en su ámbito
    - **Tipo**: define las propiedades del dato
      - Ocupación de memoria
      - Rango de valores posibles
      - Operaciones
    - **Valor**
  - Un dato puede ser:
    - **Constante**: se le asigna un valor en la declaración y no cambia (ni puede cambiar) a lo largo de la ejecución
    - **Variable**: el valor del dato puede cambiar mediante una o más asignaciones

- **Declaración de constantes**
  - Se pueden utilizar macro-definiciones para definir constantes literales, lo cual será equivalente a utilizar el valor literal cada vez
  - Ejemplo:

```
#define IVA_1 0.07
```

  - El estándar ANSI de C introduce, entre otras, la palabra clave **`const`** para indicar y proteger un dato que no debe ser modificado a lo largo del programa. En ese sentido es "una variable que no se puede modificar".
  - Ejemplo:

```
const double cero_abs -271.0;
```

- **Declaración de variables**
  - Se especifica primero el **tipo** y luego una o más **nombres** de variables separadas por comas
  - Opcionalmente pueden **inicializarse** en el momento de la declaración
  - El compilador no va a dar ningún valor por defecto a las variables por lo que es incorrecto evaluar una variable que no ha recibido ningún valor
    - Muchos compiladores lo detectan
  - Ejemplos:

```
int i, n;
int j=1, k;
double nom=0, itr=0, tip;
```

- **Tipos básicos en C**
  - **Enteros**
    - Tamaño básico (depende del compilador/modo): **`int`**
    - Corto: **`short int`** o simplemente **`short`** (2 bytes)
    - Largo: **`long int`** o simplemente **`long`** (4 bytes)
    - Con signo (por defecto): **`signed int`**
    - Sin signo: **`unsigned int`**
  - Por defecto se sobreentienden **`signed`** e **`int`**
  - **Ejemplos**

```
short i, n;
unsigned long j=1, k;
```

  - **Ejercicios**
    - ¿Cuál es el mayor **`short`** en decimal? ¿Y el menor?
    - ¿Cuál es el mayor **`long`** en decimal? ¿Y el menor?
    - ¿Cuál es el mayor **`unsigned short`** en decimal?
    - ¿Cuál es el mayor **`unsigned long`** en decimal?

- **Tipos básicos en C**
  - **Reales**
    - Norma IEEE 754 (1985)
    - Tamaño básico (4 bytes): **`float`**
      - 1 bit: signo
      - 8 bits: exponente (desplazado +127)
      - 23 bits: mantisa
      - Rango de un float X: `1.18e-38` ≤ `|X|` ≤ `3.40e38`
    - Tamaño doble (8 bytes): **`double`**
      - 1 bit: signo
      - 11 bits: exponente (desplazado +1023)
      - 52 bits: mantisa
      - Rango de un double X: `2.23e-308` ≤ `|X|` ≤ `1.79e308`
  - **Ejemplos**

```
float nota0, nota1;
double valx;
```

- **Tipos básicos en C**
  - **Caracteres**
    - Son enteros cortos de 1 byte: **`char`**
    - Por defecto con signo
    - Podemos especificar sin signo: **`unsigned char`**
    - Constantes literales entre comillas simples
    - Representación más extendida:
      - **ASCII** (7 bits): `'a'  'N'  ';'  '\n'  '\x2f'`
      - ASCII **extendido** (8 bits): `'á'  'ñ'  'ª'  '¿'  '╠'`
    - Otras representaciones: EBCDIC, Unicode
    - A veces utilizaremos un entero para albergar un carácter
  - **Ejemplos**

```
char c = '\0';
char c1, c2 = 0x40;
```

- **Tipos básicos en C**
  - **Cadenas de caracteres**
    - Ocupan un byte por carácter
    - Cuando están bien formadas utilizan un byte adicional para un carácter terminador, el carácter nulo `'\0'`
    - Son vectores o arrays de caracteres (ya veremos más adelante)
    - Podemos hacer referencia a ellas mediante punteros
  - **Ejemplos**

```
char *str;
char siglas[]="UPV/EHU";
char *dias[] = {
  "lu", "ma", "mi", "ju", "vi", "sa", "do"};
```

- **Expresiones**
  - Son fórmulas que calculan un valor
  - Una expresión puede ser un dato (variable o constante) u operaciones entre expresiones
  - Un operador relaciona uno o más operandos
    - Un operando puede ser una expresión que se calcula antes
- **Prioridad de operadores**
  - Para saber en qué orden se efectúan las operaciones se mirará a la **precedencia** o prioridad de los operadores
    - Algunos operadores tienen más prioridad que otros, por ejemplo, el producto tiene más prioridad que la suma
    - Los paréntesis hacen que se rompan prioridades y fuerzan el orden de ejecución
    - Se pueden utilizar siempre los paréntesis pero el exceso de paréntesis hace que el programa sea más difícil de leer
- **Ejemplo**

  `65+2*3` es equivalente a `65+(2*3)` y distinto de `(65+2)*3`

- **Asociatividad de operadores**
  - Cuando dos operadores tienen la **misma prioridad** se mirará a la **asociatividad** para conocer el orden de ejecución
    - Por ejemplo el producto y la división tienen la misma prioridad o cualquier operador consigo mismo, por ejemplo `3+2+5`
  - Los operadores se asocian a izquierdas o a derechas
  - Un operador se asocia a **izquierdas** cuando realiza primero la operación de la izquierda y utiliza ese resultado para seguir operando
    - Habitualmente los operadores se asocian a izquierdas
    - La suma, la resta, la multiplicación, la división... se asocian a izquierdas
    - **Ejemplo**: `20/4*3` es equivalente a `(20/4)*3` y vale `15`
  - Un operador se asocia a **derechas** cuando cede la evaluación al operando que está a su derecha
    - La **asignación** es un ejemplo de operador que se asocia a derechas
    - **Ejemplo**: `a=b+3` primero se calcula `b+3` y luego se asigna

- **Tabla de prioridad y asociatividad**

Nivel	Operadores	Descripción	Asoci.
1	`() [] -> .`	Acceso a un elemento de un vector y paréntesis	Izquierdas

2	`+ - ! ~`
	`* &`
	`++ --`
	`(cast) sizeof`
		Signo (unario), negación lógica, negación bit a bit
		Acceso a un elemento (unarios): puntero y dirección
		Incremento y decremento (pre y post)
		Conversión de tipo (*casting*) y tamaño de un elemento
			Derechas

3	`* / %`	Producto, división, módulo (resto)	Izquierdas
4	`+ -`	Suma y resta	Izquierdas
5	`>> <<`	Desplazamientos	Izquierdas
6	`< <= >= >`	Comparaciones de superioridad e inferioridad	Izquierdas
7	`== !=`	Comparaciones de igualdad	Izquierdas
8	`&`	Y (*And*) bit a bit (binario)	Izquierdas
9	`^`	O-exclusivo (*Exclusive-Or*) (binario)	Izquierdas
10	`|`	O (*Or*) bit a bit (binario)	Izquierdas
11	`&&`	Y (*And*) lógico	Izquierdas
12	`||`	O (*Or*) lógico	Izquierdas
13	`?:`	Condicional	**Derechas**
14	`= *= /= %= += -= >>= <<= &= ^= |=`	Asignaciones	**Derechas**
15	`,`	Coma	Izquierdas

- **Tipos de operadores**
  - **Unarios**: afectan a un solo operando
    - Generalmente suelen tener mayor prioridad
    - **Ejemplos**
      - Cambio de signo: `-num`
      - Preincremento y predecremento: `++num --num`
      - Postincremento y postdecremento: `num++ num--`
  - **Binarios**: afectan a dos operandos
    - **Ejemplo**
      - Resta: `num1-num2`
  - **Ternarios**: afectan a tres operandos
    - Generalmente suelen tener menor prioridad
    - **Ejemplo**
      - Operador condicional: `n<0?-n:n`

- **Tipos de operaciones**
  - **Asignaciones**
    - Las asignaciones son operadores especiales
    - Valen lo que vale su parte derecha
  - Operaciones **aritméticas o numéricas**
    - Afectan a operandos numéricos
    - El resultado es del tipo de los operandos con las debidas conversiones
  - Operaciones **relacionales**
    - Comparan magnitudes
    - El resultado es lógico: cierto o falso, `0` ó `1`
  - Operaciones **lógicas**
    - Relacionan operandos *tomados como* lógicos (`0` ó *no nulo*)
    - El resultado es lógico: cierto o falso, `0` ó `1`
  - Operaciones **bit a bit**
    - Relacionan los bits de los operandos
    - El resultado es otro entero

- **Operaciones aritméticas o numéricas**
  - Son las operaciones básicas
    - `+`: Suma
    - `-` : Resta
    - `*` : Producto
    - `/` : Cociente (entero o real, según operandos)
    - `%`: Resto de la división entera
  - Algunas implican asignación de incremento o decremento
    - `++c`: Preincremento de `c`
    - `c++`: Postincremento de `c`
    - `--c`: Predecremento de `c`
    - `c--`: Postdecremento de `c`
  - Hay formatos de operación *op* y asignación compacta
    - `+=  -=  *=  /=  %=`
    - `c` *op*`=` *expr* equivale a `c = c` *op* *expr*
    - `c += 2` equivale a `c = c + 2`

- **Operaciones relacionales**
  - Comparan magnitudes y tienen resultado lógico `0` ó `1`
  - Los operandos relacionales son 6
    - `>` Mayor
    - `>=` Mayor o igual, no sirve =>
    - `==` Igual, no confundir con la asignación `=`
    - `!=` Distinto, no confundir con otros lenguajes `<>`
    - `<` Menor
    - `<=` Menor o igual
  - Hay que tener cuidado porque no se pueden formar expresiones matemáticas de acotamiento:
    - La expresión `3 <= x <= 7` siempre será cierta ya que se evalúa de izquierda a derecha y `3 <= x` se evaluará a `0` ó a `1`, que siempre será inferior o igual a `7`
    - Tendremos que expresar la conjunción lógica "`3 <= x` **y** `x <= 7`"

- **Operaciones lógicas**
  - Relacionan expresiones lógicas
  - Una expresión lógica (operando *tomado como* **lógico**)
    - Es Falso si vale `0`
    - Es Cierto para cualquier otro valor
  - El resultado es de tipo lógico (booleano): `0` ó `1`
  - Los operandos lógicos son 3:
    - `!a` Negación (No, ¬ ó *Not*)
      - `1` si el operando es falso
      - `0` en caso contrario (si es cierto)
    - `a&&b` Conjunción (Y, ∧ ó *And*)
      - `1` si el ambos operandos son ciertos
      - `0` en caso contrario (si alguno o ambos son falsos)
    - `a||b` Disyunción (O, ∨ ó *Or*)
      - `1` si alguno de los operandos es ciertos
      - `0` en caso contrario (si ambos son falsos)

- **Operandos lógicos**
  - El resultado se expresa mediante **tablas de verdad**

		Negación	Conjunción	Disyunción
		No a	a y b	a ó b
a	b	!a	a && b	a || b
0	0	1	0	0
0	Cierto	1	0	1
Cierto	0	0	0	1
Cierto	Cierto	0	1	1

  - Equivalencias:
    - `!(A && B)` ≡ `!A || !B`
    - `!(A || B)` ≡ `!A && !B`
    - `!(A == B)` ≡ `A != B`
    - `!(A < B)` ≡ `A >= B`

    ¡Compruébalo!

- **Operaciones lógicas bit a bit**
  - Relacionan dos expresiones enteras bit a bit
  - Los operadores lógicos **bit a bit** son 4:
    - `~a` Complemento (negación, No, ¬ ó *Not*)
    - `a&b` Conjunción (Y, ∧ ó *And*)
    - `a|b` Disyunción (O, ∨ ó *Or*)
    - `a^b` Disyunción exclusiva (O-exclusivo, ⊕ ó *Exclusive-Or*)

		Complemento	Conjunción	Disyunción	Disyunción exclusiva
a	**b**	**~a**	**a & b**	**a | b**	**a ^ b**
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

- **Otras operaciones unarias bit a bit**
  - Además de las operaciones bit a bit vistas en C existen los desplazamientos
  - Los operandos de desplazamiento son 2:
    - `<<n` Desplazar `n` bits a la izda ingresando ceros por la dcha
    - `>>n` Desplazar `n` bits a la dcha ingresando ceros por la izda
  - Ejemplo (suponiendo palabras de 3 bits):

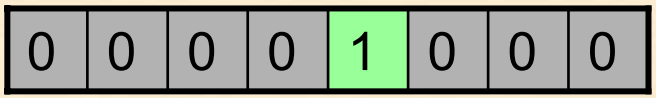
	Complemento	Desplazar izquierda	Desplazar derecha	Desplazar Izquierda	Desplazar derecha
a	~a	a << 1	a >> 1	a << 2	a >> 2
000	111	000	000	000	000
001	110	010	000	100	000
010	101	100	001	000	000
011	100	110	001	100	000
100	011	000	110	000	111

- **Ejemplos de operaciones bit a bit**
  - A continuación se muestran operaciones bit a bit entre constantes literales suponiendo que un entero ocupa un byte.
  - Los operandos se muestran en hexadecimal para que puedan obtenerse fácilmente los patrones binarios
  - Se puede comprobar la corrección tanto analíticamente como probándolas en un programa (adaptando el tamaño)

Expresión	Resultado	Expresión	Resultado	Expresión	Resultado
`0x25&0x69`	**0x21**	`0x2f&0x67`	**0x27**	`0xda&0x69`	**0x48**
`0x25|0x69`	**0x6d**	`0x43|0x81`	**0xc3**	`0xda|0x69`	**0xfb**
`0x25^0x69`	**0x4c**	`0x92^0x71`	**0xe3**	`0xda^0x69`	**0xb3**
`~0x01`	**0xfe**	`~0xa2`	**0x5d**	`~0xe5`	**0x1a**
`0x04<<1`	**0x08**	`0x1a << 3`	**0xd0**	`0x80**<<1`	**0x00**
`0x05>>1`	**0x02**	`0x91 >> 4`	**0xf9**	`0x01>>1`	**0x00**

** Para probar `0x80<<1` con una palabra de 16 bits habrá que poner `0x8000` y con una palabra de 32 bits habrá que poner `0x80000000<<1`

- **Máscaras: ejemplo de uso de operaciones bit a bit:**
  - El uso de máscaras permite empaquetar datos lógicos (booleanos)
  - Cada bit almacena un valor lógico
  - La máscara contiene todo ceros salvo el bit correspondiente:
    - `0x1 0x2 0x4 0x8 0x10 0x20 0x40 0x80 0x100 0x200 …`
  - Ejemplo: máscara 0x08



- Operaciones con máscaras (`msc` es la máscara)
  - Dado un `bloque` de bits que almacenan valores lógicos

Acción	Operación	Descripción
Activar	`bloque |= msc`	Poner a 1 el bit de la máscara
Desactivar	`bloque &= ~msc`	Poner a 0 el bit de la máscara
Alternar	`bloque ^= msc`	Cambiar de 0 a 1 ó de 1 a 0 el bit de la máscara
Verificar	`bloque & msc`	Mirar si está activo (a 1) el bit de la máscara

4. Estilo de programación

- Aparte de por sus autores la mayoría de **los programas son leídos** por otros programadores.
- Cada uno de los lectores debe poder identificar fácilmente el funcionamiento del programa e incluso localizar y corregir posibles errores o **modificar el código** original.
- La lectura de un programa escrito con un **estilo inapropiado resulta desagradable** incluso para su autor.
- Se recomienda la utilización de **comentarios** para explicar el funcionamiento de cada porción de código.
- Deben seleccionarse **identificadores** lo más **descriptivos** posibles y atenerse siempre a los mismos criterios (uso de mayúsculas o minúsculas '_' etc.).

- **Recomendaciones de estilo**
  - El estilo no afecta a la ejecución del programa
    - Pero “no es lo mismo”. Este programa es válido:

```
main(){int d;printf("\Dame: ");scanf("%d",&d);printf("%d",d);}
```

  - Es bueno utilizar unas normas “del lugar”
  - Cada maestrillo tiene su librillo
  - Mejor dos espacios que un tabulador
    - Los tabuladores no se distinguen
    - Cambian de tamaño de un editor a otro
  - Un salto de línea entre la declaración de variables y el código
  - Agrupar declaraciones de variables de conceptos similares

Escuela Universitaria de Ingeniería Vitoria-Gasteiz

Ingeniaritzako Unibertsitate Eskola Vitoria-Gasteiz

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea