UD4

INTRODUCCIÓ A LA PROGRAMACIÓ ESTRUCTURADA EN C

Autor: Lluís València López
`http://www.xtec.cat/~lvalenci/cfgs.htm`

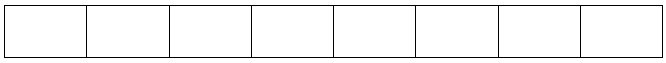
4.1 Vectors o matrius

Fins ara, hem fet programes que utilitzaven únicament dades simples, variables declarades com d'un dels tipus elementals: char, short, int o long long si eren enteres o float, double o long double si eren nombres amb decimals.

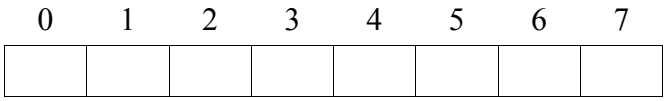
Els vectors són les primeres dades estructurades que estudiarem.

Són variables que permeten emmagatzemar diverses dades d'**un mateix tipus**.

Va bé pensar els vectors així:



Com una sèrie de caselles ocupades per dades del mateix tipus elemental. Les caselles es numeren començant per zero:



Per tant, què ens caldrà per definir un vector?

- Un tipus de dada elemental.
- Un nom de variable
- Una quantitat de "caselles", d'elements, de posicions.

Vegem alguns exemples triant valors sense cap pretensió més enllà de mostrar com es declara una variable de tipus vector:

Tipus elemental	Nom de la variable	Quants elements	Declaració en C	Índexos vàlids
int	var1	5	`int var1[5]`	0,1,2,3,4
char	Lletres	7	`char lletres[7]`	0,1,2,3,4,5,6
long double	Frac	2	`long double frac[2]`	0,1

Acabem de veure que el lloc que ocupa un element en el vector, aquest lloc que comença per 0, s'anomena **índex** d'aquell element.
A més, el nombre d'elements d'un vector es denomina **longitud** del vector.

Per tant, la declaració serà sempre:

tipus_elemental nom_vector[longitud]

Com fem assignacions o comparacions amb elements d'un vector?

Fent anar l'índex corresponent. En el cas de l'exemple `int var1[5]`, podríem fer, per exemple, `var1[3]=-44` que seria una assignació vàlida ja que els elements del vector són enters i –44 és un enter.

Si, posant un altre exemple, volem preguntar si l'element de la casella 0 de `lletres` és una minúscula, farem:

```
if ((lletres[0]>='a')&&(lletres[0]<='z'))
```

Ara bé, si un vector s'ha declarat de longitud N, **MAI** no farem cap assignació que utilitzi N com a índex. És a dir, en els exemples anteriors (recordem les declaracions dels vectors var1 i lletres) no convé fer `var1[5]=99` ni `lletres[7]='S'` perquè estem utilitzant un índex invàlid.

També podem assignar elements a un vector en el moment de declarar-lo. En aquest cas, no cal posar la longitud del vector:

```
int var1[]={23, 0, -12,- 44, 1056};
```

Quan es declara un vector, què succeeix a nivell de la taula de variables/adreces i la memòria lògica?

Prenem com exemple el primer vector que hem declarat

```
int var1[5]
```

A nivell de la memòria, suposem que aquest vector se li assigna la posició 0x1000. Com que el vector conté 5 `int` i `sizeof(int)` és 4, es reservaran 5*4=20 bytes consecutius. Altrament dit, a la posició 0x1000 es trobarà `var1[0]`, a la 0x1004 `var1[1]`, a la 0x1008 `var1[2]`, a la 0x100C `var1[3]` i a la 0x1010 `var1[4]`(Vegeu la part dibuixada en negre de l'esquema de la següent pàgina).

Val la pena saber que SEMPRE que es declara un vector, el NOM DEL VECTOR és un sinònim de l'adreça del seu primer element. En els nostres exemples:

`var1` és sinònim de `&var1[0]`
`lletres` és sinònim de `&lletres[0]`
`frac` és sinònim de `&frac[0]`

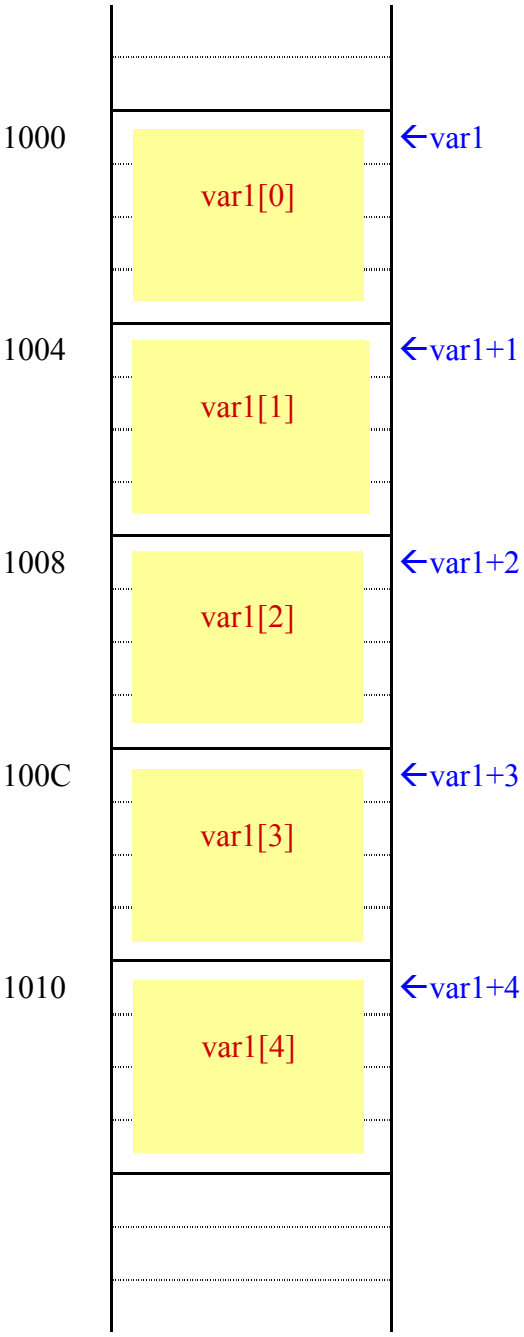
Així, com es té `var1 ≡ &var1[0]`, l'aritmètica d'apuntadors farà que les adreces de la resta d'elements del vector siguin:

```
var1+1 ≡ &var1[1]
var1+2 ≡ &var1[2]
var1+3 ≡ &var1[3]
var1+4 ≡ &var1[4]
```

A nivell de la taula de variables-adreces que havíem estudiat a la UD3, els vectors s'emmagatzemen així:

variable	adreça	tipus de dada elemental
`var1`	`0x1000`	`int`

Adonem-nos d'un canvi respecte les dades de tipus elementals: ara el nom de la variable no conté cap dada sinó l'adreça del vector.



Si volem copiar un vector no podem fer una assignació directa. Pensem que tenim les declaracions:

```
int vectA[5];
int vectB[5];
```

En aquest cas, no és correcte fer l'assignació:

```
vectB = vectA;
```

Perquè? (penseu una estona abans de passar de pàgina...)

Raonem al voltant de quina seria la taula variables-adreces:

variable	Adreça	tipus de dada elemental
`vectA`	`0x1000`	`Int`
`vectB`	`0x1014`	`Int`

És a dir, `vectB` (recordem que `vectB` és l'adreça del primer element de `vectB`) val 0x1014 i no pot passar a valer cap altra cosa, ja que aquesta és una dada que ens ve assignada en el moment de fer la declaració, no una dada que podem canviar nosaltres.

Si el que volem és copiar el contingut de les "caselles" de vectA (més ben dit seria *copiar els elements de vectA*) "dins" vectB, caldrà fer una assignació per a cada element:

```
vectB[0] = vectA[0];
vectB[1] = vectA[1];
vectB[2] = vectA[2];
vectB[3] = vectA[3];
vectB[4] = vectA[4];
```

En canvi, si la situació és aquesta:

```
int vectA[5];
int vectB[5];
int * apint;
```

Sí que podem fer, posem per cas:

```
apint = vectA;
```

Observem que, un cop tenim sinònims per les adreces dels elements del vector, també podem demanar què hi ha en aquestes adreces que no serà altra cosa que els propis elements. És a dir, es tindrà que:

```
* vectA    = vectA[0];
*(vectA+1) = vectA[1];
*(vectA+2) = vectA[2];
*(vectA+3) = vectA[3];
*(vectA+4) = vectA[4];
```

Com a exemple final, declararem el vector lletres[4], recollirem els seus elements (4 caràcters) del teclat i els mostrarem per pantalla.

```
#include <stdio.h>

main()
{
char lletres[4];
scanf("%c%c%c%c",lletres, lletres+1, lletres+2, lletres+3);
printf("%c%c%c%c\n", lletres[0], lletres[1], lletres[2],\
       lletres[3]);
}
```

Cal posar especial atenció a dos detalls:

- Dins l'scanf continua havent &, el que passa que "amagats" sota els seus sinònims.
- Al `printf` només poden anar els elements del vector, és a dir coses de la forma `lletres[i]`, on `i` ha de ser un índex vàlid.

4.2 Strings

Què és un string?

Des d'un punt de vista informal, per acostar-nos al concepte, podríem dit que els strings són variables que contenen "frases": les tirallongues de caràcters. La traducció més estesa d'string és ***cadena de caràcters***.

La primera aproximació seriosa que farem al concepte d'string és:

Un string és un vector de dades de tipus char

Però això no és una definició ja que qualsevol vector de `char` no és un string. Per ser-ho, convé *marcar* el final de la cadena de caràcters (de la frase) amb un caràcter especial: un byte tot de zeros.
Aquest caràcter, el que té com a codi ASCII el zero s'anomena caràcter nul i es pot expressar entre apòstrofs així: `'\0'`

Ara sí que donem la primera definició d'string:

***Un string és un vector de dades de tipus* `char` *que marca el final de frase amb* `'\0'`**

Per exemple, podem definir el següent vector:

```
char cad[20];
```

La variable `cad` podrà contenir qualsevol frase de com a molt 19 caràcters (cal 1 caràcter per marcar el final de frase amb `'\0'`)

És a dir, cad pot contenir la frase *Em dic Genaro*, que consta de 13 caràcters, o bé la frase *Avui toca strings*, que consta de 17, o bé *Hola*, de 4 caràcters, però no podrà contenir *Avui és el dia que sabrem finalment què es un string*, perquè passa dels 19 caràcters.

En un segon exemple, considerem el vector:

```
char frase[10];
```

Hores d'ara, ja ha de quedar clar que com a molt podrà contenir strings de, com a molt, 9 caràcters.
Si podem assignar a aquest vector la frase *Hola Joan*, el contingut del vector serà:

'H'	'o'	'l'	'a'	' '	'J'	'o'	'a'	'n'	'\0'

Mentre que si contingués el nom *Shakira*, el seu contingut, com a vector fora:

'S'	'h'	'a'	'k'	'i'	'r'	'a'	'\0'	?	?

Hem posat un interrogant a les dues darreres caselles perquè no sabem què hi haurà (tampoc no ens importa gaire).

Assignació de valors a un string

Quan es declara un string, es pot inicialitzar amb una frase escrita entre comentes:

```
char cadena[30]="Muntanyes del Canigó";
```

Que omplirà les primeres 21 caselles de `cadena` amb la frase que veiem i posarà `'\0'` a la casella 22. Les caselles 23 a 29 que hem reservat, en l'exemple no s'utilitzen.

Si volem assignar valor a un string un cop declarat, podem fer servir la funció **`strcpy`** de la biblioteca `string.h`, que s'utilitza així:

```
strcpy(cadena, "Muntanyes del Canigó");
```

Observem que tant si inicialitzem l'string en declarar-lo com si fem ús de les funcions de la biblioteca string.h, no cal que ens preocupem d'escriure el caràcter nul al final de la frase.

També es pot fer servir una funció de la biblioteca `stdio.h` que és idèntica a `printf` excepte que no imprimeix en pantalla sinó en l'string que li passem com a primer paràmetre. Es tracta de la funció `sprintf`, de la qual aquí tenim un exemple:

```
int n=105;
char frase[40];
sprintf(frase, "El valor d'n és %d", n);
```

En aquest cas, el contingut del vector frase serà :

'E'	'l'	' '	'v'	'a'	'l'	'o'	'r'	' '	'd'	'\''	'n'	'é'	's'	' '	'1'	'0'	'5'	'\0'	

Ara, ja podem donar la segona definició d'string:

Un string és una cadena de text entre cometes dobles.

Que determina el concepte en funció de les constants que assignarem a un string.

No ens podem permetre de confondre caràcters amb strings, tot i que un string podria tenir un únic caràcter (a part del '\0'). Fixem-nos en el següent exemple:

```
char var1='a';
char var2[ ]="a";
```

I adonem-nos que var1 és una variable bàsica, un `char`, que pren valor 'a', mentre que var2 és un vector que contindrà dos valors de tipus `char`: una 'a' i el caràcter nul '\0'.

Lectura i escriptura d'strings

Per mostrar per pantalla un string, s'utilitza **`printf`** amb el modificador `%s`. Si fem

```
char cadena[30]="Muntanyes del Canigó";
printf("%s\n", cadena);
```

S'imprimirà *Muntanyes del Canigó* per pantalla.

Per llegir un string que s'introdueixi des del teclat , podem fer servir **`scanf`** amb l'opció **`%s`**, però això té el problema que no recull espais en blanc. Podríem utilitzar aquesta possibilitat en l'exemple:

```
char nom[20];
printf("Escriu el teu nom (no accepto espais en blanc): ");
scanf ("%s", nom);
printf ("Hola, %s\n", nom);
```

L'exemple funcionarà bé amb *Marta* o *Jaume*, però no amb *Joan Anton* o amb *Maria Josep*.

Tot i que el compilador donarà sempre un avís, podem fer servir la funció **`gets`** per recollir frases amb espais en blanc:

```
char nom[20];
printf("Escriu el teu nom: ");
gets(nom);
printf ("Hola, %s\n", nom);
```

Ara, provem amb noms compostos que no haurà cap problema.

Apuntadors i strings

En els exemples d'strings vistos fins ara, quina mena de variable són `cad`, `frase`, `cadena`, `var2` o `nom`? Doncs com a noms de vectors de `char`, són apuntadors a `char`. Aleshores podem dir, sense que es tracti d'una definició, que:

Un string és un apuntador a char

No es tracta d'una definició perquè no tots els apuntadors a `char` seran strings: cal que apuntin a una cadena de caràcters que marca el seu final amb '\0'.

Quan en un programa escrivim una cadena de caràcters entre cometes (cometes dobles), aquesta es situa en una posició de memòria que tot programa té reservada per als strings, de manera que és vàlid fer assignacions com la del següent exemple:

```
char * aps; //declarem un apuntador a char
...
aps = "Una frase adjudicada a mig programa";
```

I podrem tractar aps com un string qualsevol, com ara per imprimir-lo amb `%s` o utilitzar-lo en funcions de la biblioteca string.

La biblioteca string

La biblioteca string ens facilita funcions per al tractament d'strings.
Per fer-la servir, només caldrà posar `#include <string.h>` a l'encapçalament dels nostres programes.
D'entre les seves funcions, és primordial conèixer l'ús de `strlen`, `strcpy` i `strcmp` que tenen els següents perfils:

```
size_t strlen(const char *s);
char *strcpy(char *desti, const char *origen);
char *strcmp(char *cad1, const char *cad2);
```

La funció `strlen` ens retorna la mida d'un string (en realitat `size_t` vol dir `unsigned char`), mentre que la funció `strcpy` copia la cadena origen en la cadena destí, incloent el '\0', és clar. De moment, no posarem atenció al valor de retorn d'aquesta funció.

La funció `strcmp` ens servirà de moment per veure si dos strings contenen la mateixa frase, en el qual cas retorna 0.
Així, per saber si l'string s1 i l's2 contenen el mateix text, preguntem:

```
strcmp(s1,s2)==0
```

Exemple:

```
#include <stdio.h>
#include <string.h>

main()
{
     char frase[80];
     char copia[80];
     printf("Escriu una frase: ");
     gets(frase);
     printf ("La teva frase té %d caràcters\n",\
             strlen(frase));
     strcpy(copia, frase);
     printf ("I és la frase: \n%s\n", copia);
     if (strcmp(copia, frase)==0)
          printf("Les dues frases són iguals\n");
     else
          printf("Les dues frases són diferents\n");
}
```

Pel que fa a la funció `strcpy`, adonem-nos que ens facilita les coses donat que els strings són vectors i ja havíem vist que per copiar un vector cal fer-ho component a component.

FAQ'S

Pregunta: Tot vector de `char` es un string?

Resposta: No. Cal que tingui un caràcter "marca de final de frase" (caràcter '\0').

P: Si `s` està declarat `char s[10]`, podem escriure `s="Hola";` ?

R: No. `s` no pot canviar, és l'adreça que ens han assignat pel primer element del vector. Només ho podem fer en el moment de declarar: `char s[10]="Hola";` (o bé `char *s="Hola";`)

P: Si es declara `char s[10];` , a quin tipus de variable li podem assignar el valor d'`s`?

R: `char * var;`

P: Si tenim `char s[10];` , podem utilitzar a `s` les funcions d'strings?

R: Només les que inicialitzen `s`. Per exemple, sí podríem fer: `strcpy(s,"Hola");`

P: Si tenim:
```
char s[10]="Hola";
char t[10]="Hola";
```
La comparació `s==t` es cert(1) o fals(0) ?

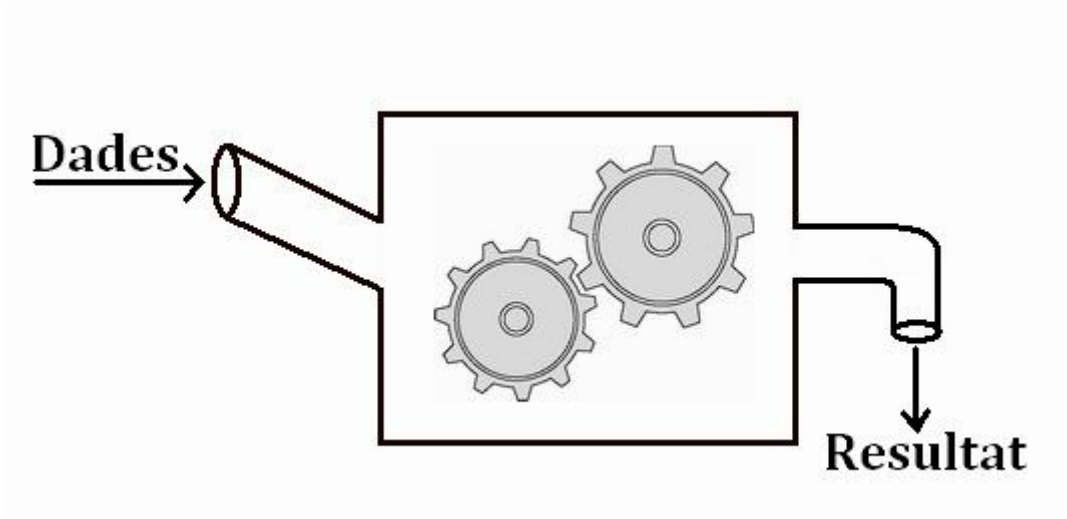
R: Fals. `s` és l'adreça del 1er element d'`s` (p. ex. 0x1000) i `t` la del primer de `t`, per exemple 0x100A, mai no poden ser iguals.

4.3 Introducció a les funcions

Conceptes:
funció, paràmetres d'una funció, valor de retorn i crida d'una funció.

A la propera unitat, la UD5, tractarem a fons les funcions en C i desvelarem totes les qüestions que quedin pendents en aquesta petita introducció.

Encara que tractant-se de C hi ha un munt d'excepcions, m'agrada presentar el concepte de funció així:
Pensem que una funció és una caixa amb un *"funcionament"* determinat per certa *"maquinària"* interna:



Les dades que se li *passen* a una funció en C s'anomenen **paràmetres** de la funció i la dada que retorna la funció **valor de retorn**.

Així, el símil del mecanisme es pot expressar: a iguals paràmetres, igual valor de retorn.

Per especificar bé una funció en C ens cal saber:

- Quants paràmetres té.
- Quin és el tipus de dada de cada paràmetre.
- Quin és el tipus del seu valor de retorn.

De manera que el ***prototipus***, ***declaració*** o ***perfil*** d'una funció consisteix a escriure:

```
tipus_valor_de_retorn nom_funció (llista_paràmetres);
```

On `llista_paràmetres` vol dir una sèrie de declaracions de variables separades per comes.

Per exemple, una funció que rebi com a dades dos nombres enters i retorni la seva potència entera (el primer elevat al segon) es declararia així:

```
int potencia(int base, int exponent);
```

De fet, quan es declara una funció només cal dir de quin tipus seran els seus arguments i no és obligatori posar-los-hi nom. Fixem-nos: la declaració acaba amb punt i coma.

Suposem que algú escriu el codi de la funció que hem posat com a exemple i que ens assegura que només funciona per bases i exponents positius i que dóna el valor de la potència quan aquest es pot calcular, mentre que retorna -1 quan el càlcul està fora del rang dels enters o quan es fa servir amb algun argument negatiu.

Els nostres programes podran fer ús d'aquesta funció sempre la declarin abans, bé directament, bé mitjançant la clàusula del preprocessador `#include`.

Aleshores, el nostre programa, podrà fer que la variable `cub` sigui la tercera potència de la variable `n` amb l'assignació següent: `cub = potencia(n,3);`

Hem fet ús de la funció (es diu: hem cridat la funció) amb **paràmetres actuals `n`** i **`3`**, encara que qui ha programat la funció hagi posat nom `base` i `exponent`, posem per cas, als **paràmetres formals** de la funció.

Codificació de funcions

Fem un pas més: mirem com podem nosaltres no tan sols cridar funcions sinó també crear-les: com ha de ser el codi d'una funció.
Per poder fer un exemple complert, escriurem una funció que sumi dos nombres de tipus **int** i doni com a resultat un nombre de tipus **long long**.

El codi d'una funció comença amb el seu prototipus, ara obligatòriament amb nom de variable als paràmetres. Després és com el que havíem vist fins ara dins la funció `main` fins arribar a la darrera línia de la funció abans de tancar la clau que començarà per `return` (també això té excepcions, però ja en parlarem...).

Així, la funció del nostre exemple tindrà un codi ben curt:

```
long long suma(int a, int b)
{
     return (long long)a+b;
}
```

Un programa que la faci servir podria ser aquest:

```
#include <stdio.h>
long long suma(int, int);

main()
{
     int i,j;
     printf("Introdueix dos nombres enters: ");
     scanf ("%d %d", &i, &j);
     printf ("%d + %d = %lld\n", i,j,suma(i,j));
}
```

Fixem-nos com el programa declara la funció abans del seu ús.

Tot i les diferents possibilitats que estudiarem per dur a terme programes estructurats (amb funcions), la metodologia que seguirem per començar serà:

```
// #directives del preprocessador
// Declaració de funcions
main()
{
     //programa principal
}
// Codificació de les funcions
```

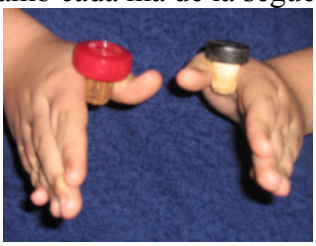
De manera que el programa font que compilarem per provar l'exemple serà:

```
#include <stdio.h>

long long suma(int, int);

main()
{
     int i,j;
     printf("Introdueix dos nombres enters: ");
     scanf ("%d %d", &i, &j);
     printf ("%d + %d = %lld\n", i,j,suma(i,j));
}

/*************************************************
*/
long long suma(int a, int b)
{
     return (long long)a+b;
}
```

El que hem vist fins ara és com són les funcions pròpiament dites, perquè hi ha certes *"funcions"* que no retornen cap valor (caldria anomenar-les procediments) i fins i tot hi ha que modifiquen el valor de les dades d'entrada.
De moment seguim la regla següent: no posarem la crida d'una funció allà on no tingui sentit escriure una variable o constant del mateix tipus que el que retorna la funció.

4.4 Algorismes amb ordinogrames

Una altra debilitat que tinc és explicar què és un algorisme amb un exemple que consisteix a agafar dos taps, un amb cada ma de la següent manera:



Es tracta de fer el que suggereix la següent seqüència de fotografies:

Quan mostro aquest "truquet" a classe, demano un voluntari que no l'hagi vist mai que el faci i el que ha passat fins ara és que no se'n surt sense separar els dits un cop ha agafat amb cada mà el tap de l'altra (fins ara no havia hagut fotos...)
Aleshores dono un algorisme: mireu de posar el dit gros a la part de sota del tap (amb taps amb cap com aquests és fàcil distingir la part de sota) i amb qualsevol dels altres dits agafeu l'altre extrem del tap i... *voilà*! els taps canvien de mà!

El que vull mostrar és que un algorisme és una metodologia força sistemàtica per resoldre un problema que no requereix més que seguir-lo fil per randa per arribar a la solució.
Posem per cas que volem saber quina és la part entera del logaritme en base dos d'un nombre. Tranquils, que és més fàcil del que sembla: donat un nombre N, esbrinar quina potència de dos és la d'exponent més alt que no és més gran que N.
Per exemple, si volem saber la part entera del logaritme en base 2 de 346, començarem a fer potències de dos fins arribar a la vuitena, que val 256, perquè la novena ja val 512, que és més gran que 346. De manera que la part entera del logaritme en base 2 de 346 és 8.

L'algorisme que seguirem per calcular la part entera del logaritme en base 2 d'un nombre N enter i positiu, pot ser el següent:

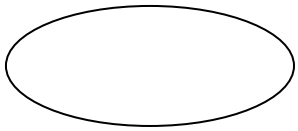
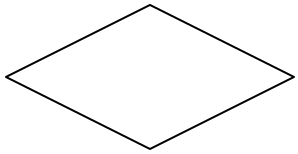
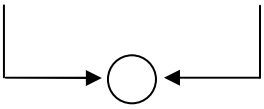
1) Inicialitzem E (la part entera del logaritme en base 2) a 0
2) Si N és més gran que 1 anem al punt 3), sinó al 6)
3) Sumem 1 a E
4) Fem N=N div 2 (div voldrà dir divisió entera)
5) Anem al punt 2)
6) Mostrem el valor d'E

Si li donem voltes a l'algorisme, ens adonarem que és equivalent al que hem fet abans d'anar calculant potències de 2.

La qüestió és: com puc representar l l'algorisme anterior?
Una manera de fer-ho és mitjançant ordinogrames.

Els ordinogrames són símbols que representen accions, preguntes, impressió de resultats...

No hi ha dos manuals que expliquin els ordinogrames exactament igual. Mirarem de fer servir símbols que estan força acceptats però posarem algunes regles que potser no són tan globals.

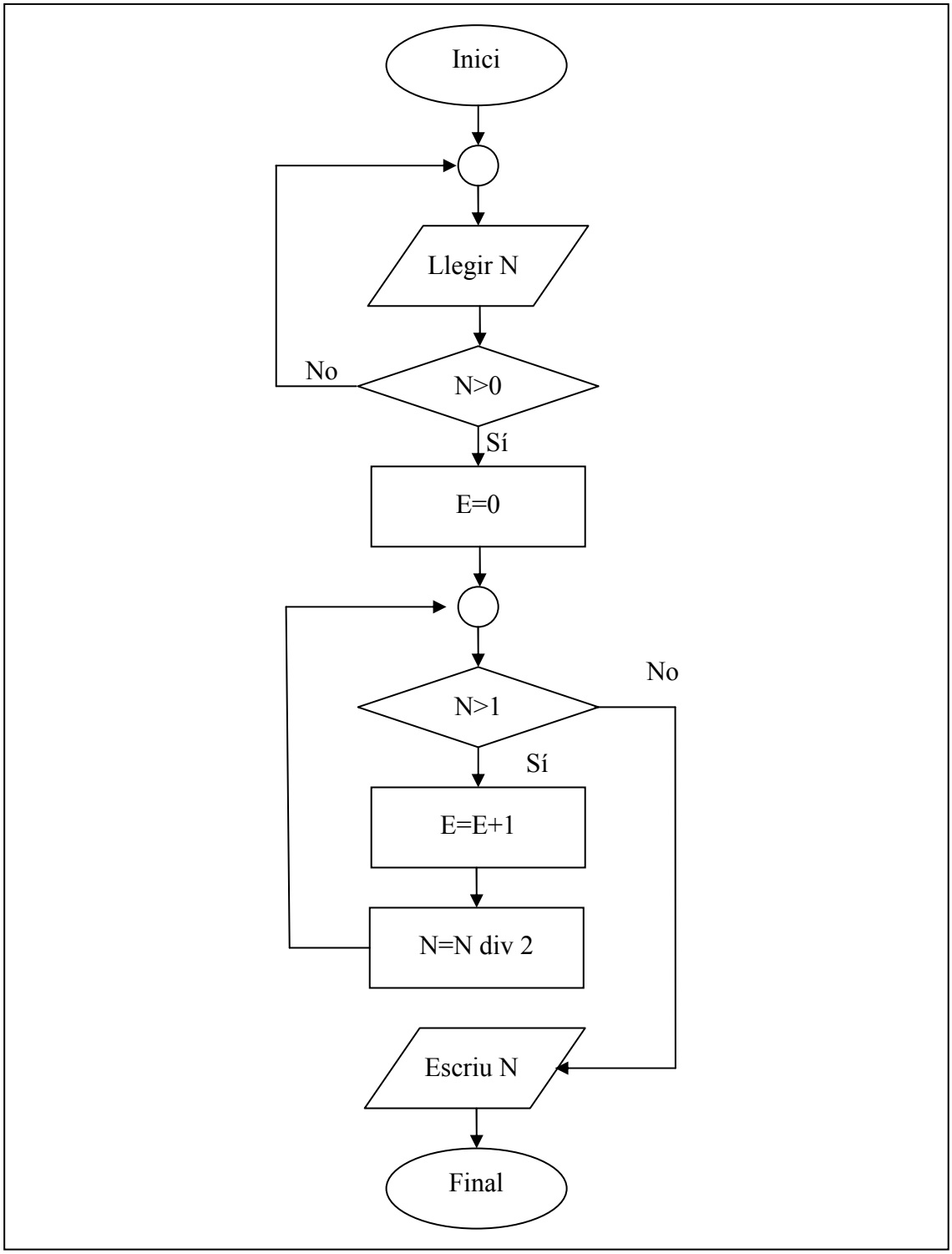
	El·lipse: indica l'inici i el final de l'algorisme
	Paral·lelogram oblic: indica entrada o sortida de dades
	Rectangle: dins haurà alguna acció a realitzar. Les més bàsiques seran les assignacions.
	Rombe: hi posarem condicions que podran ser certes o falses
	Rodonetes: seran connectors de flux, ja que les fletxes indiquen el flux de dades.

Hi ha qui afegeix símbols per indicar si l'entrada de dades és des del teclat o la sortida per pantalla o per la impressora, però nosaltres no farem aquestes distincions.

Seguirem algunes normes per elaborar ordinogrames:

- El flux ens ha de portar, triem el camí que triem, al final del programa.
- No pot sortir més d'una fletxa de flux de cada figura, excepte dels rombes, que en sortiran dues.
- No farem arribar més d'una fletxa a una figura, excepte als connectors de flux.
- Mirarem de no fer dibuixos que no s'entenguin i per això evitarem de creuar fluxos o de fer diagrames massa grans (podem resumir una part d'un diagrama en un rectangle: equivaldria a fer servir una funció).
- Normalment, farem que l'algorisme es llegeixi de dalt a baix.
- Els algorismes no són C, per tant cal desvincular-los del llenguatge (en principi han de servir per qualsevol llenguatge estructurat). Així, les accions que hi posem han de ser en el llenguatge matemàtic usual.

Bé, crec que ja podem "pintar" l'ordinograma del nostre exemple, el càlcul de la part entera del logaritme en base 2 d'un nombre enter positiu:

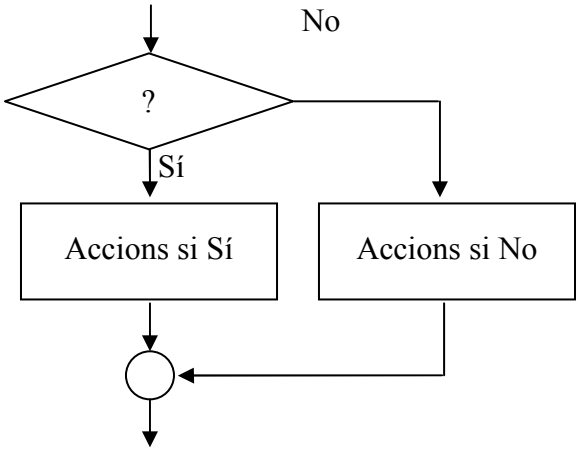


4.5 Estructures de control alternatives

Les estructures de flux alternatives de què gaudeix C les podem classificar d'entrada com alternativa simple i alternatives múltiples

Alternativa simple

Es tracta de l'estructura que amb un ordinograma pintaríem així:



En C la pregunta sempre s'expressarà, com ja hem vist a la UD3 com una condició (comparació o combinació lògica de comparacions) i la sintaxi depèn de si hi ha accions a fer en cas que la resposta sigui No o no les hi ha. Si no s'ha de fer res en cas de No, serà:

```
if (condició)
     //bloc d'accions en cas que la condició sigui certa
```

En canvi, si en el cas de No sí que s'han de fer accions:

```
if (condició)
     //bloc d'accions en cas que la condició sigui certa
else
     //bloc d'accions en cas que la condició sigui falsa
```

Hem de tenir present que un bloc d'accions és una instrucció de C o diverses instruccions **limitades per claus**.

Exemple: Programa que llegeixi un nombre decimal N i una lletra L. Suposarem que l'usuari només pot introduir la lletra 'e' o la lletra 'd'. Si introdueix la 'e' voldrà dir que la quantitat que ha introduït són euros i els ha de traduir a dòlars, mentre que si la lletra que ha posat és la 'd' farà just el contrari. Consulteu la cotització de l'euro en dòlars i poseu-la on hi ha X.XXXX. (Deixem l'elaboració de l'ordinograma per al lector)

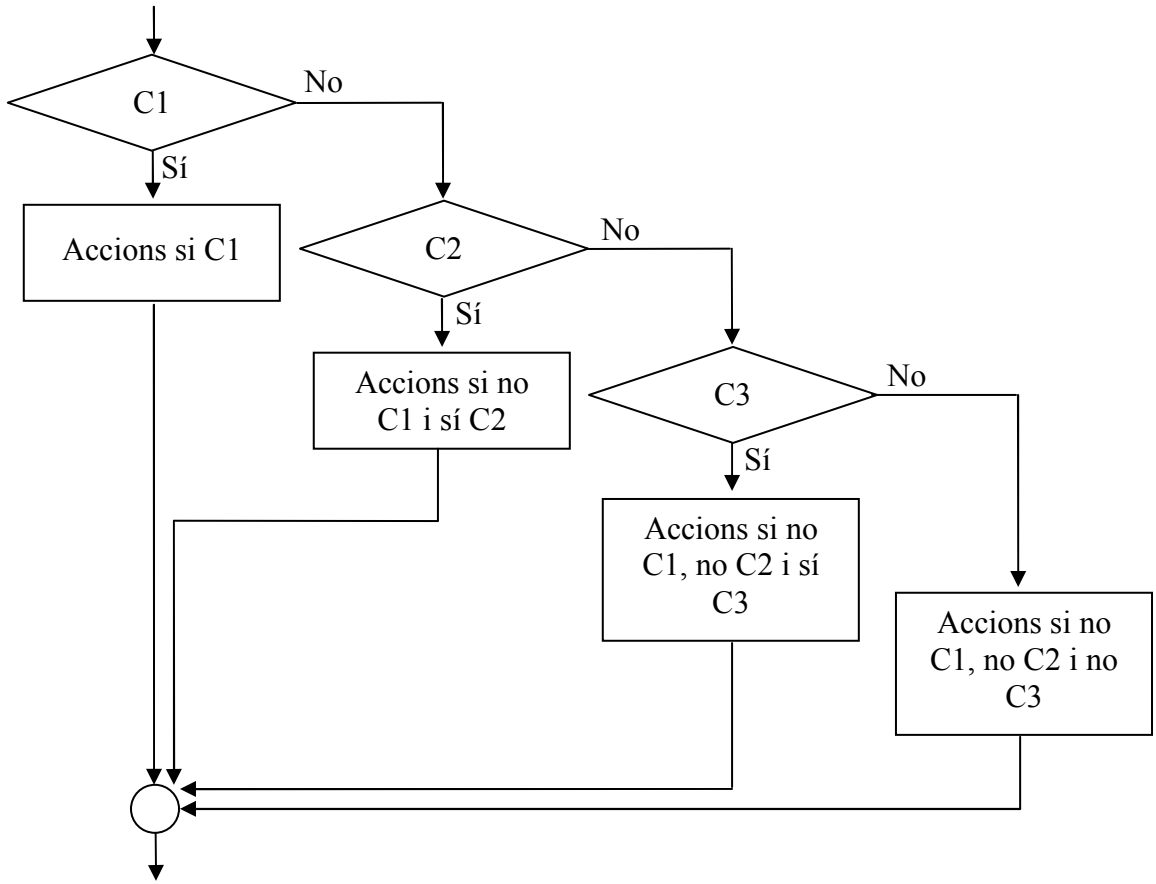
```
#include <stdio.h>
#define TAXA X.XXXX

main()
{
float quantitat, euros, dolars;
```

```
char moneda;
printf (“Escriu un quantitat seguida d’e o d: “);
scanf(“%f %c”,&quantitat, &moneda);
if (moneda==’e’)
     {
     euros=quantitat;
     dolars=euros*TAXA;
     printf (“%g euros equivalen a %g dòlars\n”, euros,\
               dolars);
     }
else
     printf (“%g dòlars equivalen a %g euros\n”, \
               quantitat, quantitat/TAXA);
printf (“Final\n”);
}
```

Tot i que podríem haver-nos estalviat de declarar les variables `euros` i `dolars`, d’aquesta manera hem pogut mostrar un bloc amb més d’una instrucció i un altre amb una de sola.
Tampoc era necessària la darrera línia, però així queda constància de que tant un camí com l’altre *s’uneixen* aquí.

Alternatives múltiples

En C disposem de dues estructures alternatives múltiples. La més general correspondria a l’ordinograma:



Hem posat un exemple amb tres preguntes (4 alternatives) quan bé podria haver estat amb més.

La sintaxi en C seria:

```
if (Condició1)
     //bloc accions en cas que la condició 1 sigui certa
else if (Condició2)
     //bloc d'accions amb C1 falsa i C2 certa
else if (Condició3)
     //bloc amb C1 i C2 falses i C3 certa
else
     //bloc amb C1, C2 i C3 falses
```

Com a exemple, posarem un programa que reculli un caràcter i digui si es tracta d'una lletra majúscula, una lletra minúscula, un dígit o un cap de les tres coses (considerem lletres les de l'alfabet anglès, és a dir, 'ç' es considerarà que no és cap de les tres coses).

```
#include <stdio.h>

main()
{
char caracter;
printf ("Escriu un caràcter qualsevol: ");
scanf("%c",&caracter);
if ((caracter>='A')&& (caracter<='Z'))
     printf ("%c és una lletra majúscula\n", caracter);
else if ((caracter>='a')&& (caracter<='z'))
     printf ("%c és una lletra minúscula\n", caracter);
else if ((caracter>='0')&& (caracter<='9'))
     printf ("%c és un dígit\n", caracter);
else
     printf ("%c és un caràcter especial\n", caracter);
printf ("Final\n");
}
```

L'altre esquema alternatiu múltiple que ens ofereix C seguiria el mateix ordinograma però en el cas concret que C1 sigui la pregunta var=valor1, C2 var=valor2... fins arribar al cas que no és dels valors anteriors.

A més, *var* **ha de ser**, en C, una variable entera (char, short, int, long long).

La sintaxi és la següent:

```
switch (variable)
     {
     case valor1:
                 instruccions;
                 break;
     case valor2:
                 instruccions;
                 break;
     ...
     case valorN:
                 instruccions;
                 break;
     default:
                 instruccions;
                 break;
     }
```

Notem que després de cada `case` no hi van claus, sinó les instruccions una darrera l'altra i al final una sentència *break*. Si no es posés aquesta sentència (`switch` funciona enviant el flux al `case` que correspon) allà on faltés, s'executarien les instruccions i es continuarien executant instruccions fins al final o fins a trobar un altre *break*, circumstància normalment indesitjada.
Cada any algun alumne pregunta *i el darrer break, és necessari?* I la resposta és sempre la mateixa: no es necessari però és qüestió d'estil i metodologia perquè el que sí és de debò greu és deixar-se'n algun, de *break*, allà on cal posar-lo.

Exemple: Programa que reculli una paraula de cinc lletres (no es poden posar accents) i compti quantes són vocals i quantes consonants.

```
#include <stdio.h>

main()
{
char paraula[6];
int cons=0, voc=0;
printf ("Escriu una paraula de 5 lletres minúscules: ");
scanf("%s", paraula);
//passem les lletres a minúscules
int i=0;
if ((paraula[i]>='A')&& (paraula[i]<='Z'))
     paraula[i]= paraula[i]-'A'+'a';
if ((paraula[i]>='a')&& (paraula[i]<='z'))
     {
     switch(paraula[i])
          {
          case 'a':
          case 'e':
          case 'i':
          case 'o':
          case 'u':
                    voc++;
```

```
                    break;
          default:
                    cons++;
                    break;
          }
     }
i=i+1;
/*repetim les instruccions que hi ha des del primer if fins
aquesta els 4 cops que falta (quines ganes d'arribar a les
estructures iteratives!)*/
...
printf("Hi ha %d vocals i %d consonants\n",voc,cons);
}
```

Excepcionalment, en el C que podem trobar als sistemes Linux, podem utilitzar rangs dins un *switch*, tal com mostra aquest altre exemple:

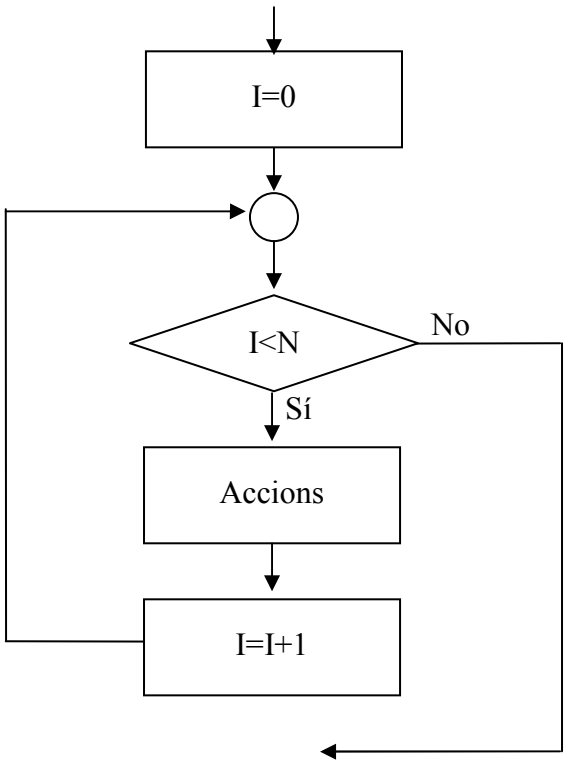
```
#include <stdio.h>
#include <limits.h>

main()
{
int nombre;
printf ("Escriu una nombre enter: ");
scanf("%d", &nombre);
switch(nombre)
     {
     case INT_MIN ... -1:
               printf("negatiu\n");
               break;
     case 0:
               printf("zero\n");
               break;
     default:
               printf("positiu\n");
               break;
     }
}
```

4.6 Estructures de control iteratives

Feia dies que estàvem preparats per entendre les estructura de control de flux iteratives i, per tant, ja teníem ganes d'arribar a aquest punt.
Les estructures de control iteratives són les que treuen el màxim profit de la capacitat de càlcul d'un ordinador ja que plantegen la repetició, canviant algunes condicions, d'una sèrie d'accions.
Per exemple, utilitzant aquestes estructures de control, també anomenades **bucles**, podríem sumar tots els termes de la progressió n^2+n+1 per n des de 0 fins 10000 sense conèixer cap fórmula per calcular directament aquesta suma i l'execució serà gairebé instantània.

Estructura for

Explicarem per començar la versió més utilitzada d'aquesta estructura, sovint anomenada bucle incondicional, perquè recorre tots els valors que pot prendre certa variable entera entre un de mínim i un de màxim.
La seva correspondència amb ordinogrames seria, poc o molt:



I la seva sintaxi:

```
for (i=0;i<N;i++)
     //Accions
```

Fem notar que *i* i *N* han d'haver estat declarades com a enteres.

Per exemple, l'enunciat anterior: sumem els termes de la progressió n^2+n+1 per n des de 0 fins 10000.

```
#include <stdio.h>
#define N 10001

main()
{
int i;
long long suma=0;
for (i=0;i<N;i++)
        {
        suma+=i*i+i+1;
        }
printf("La suma es %lld\n",suma);
}
```

Comproveu com l'execució és molt ràpida, tot i que la iteració provoca que es realitzin desenes de milers d'operacions.

Ja hem dit que aquesta és la versió més utilitzada del bucle *for*, però és molt simple comparada amb la versatilitat que en realitat té aquest bucle. En general:

`for (`*inicialitzacions*`;` *condicions per continuar*`;` *canvis entre iteracions*`)`

Per exemple, un bucle for vàlid podria ser:

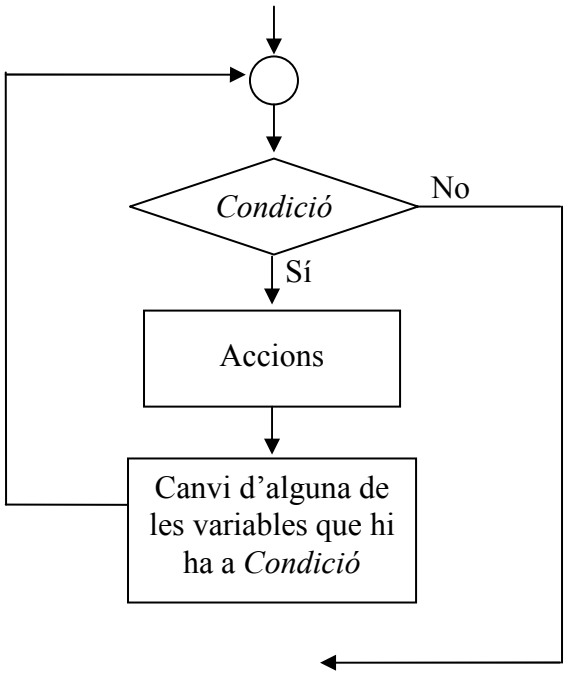
```
for (i=10, j=23; i>0, j<=92; i=i-2, j=j*2)
     printf("i: %d; j: %d\n",i,j);
```

L'execució del qual produirà la sortida:

```
i: 10   j: 23
i: 8    j: 46
i: 6    j: 92
```

Estructura while

Aquesta estructura de control executa una sèrie d'instruccions mentre es doni certa condició.



La seva sintaxi en C és força simple:

```
while (condició)
     //bloc d'instruccions
```

És important entendre que si s'entra a executar un bucle *while* és perquè es dóna la condició que el gestiona i que per tant **CAL** alterar dins les seves instruccions alguna de les variables que apareixen a la condició perquè l'execució de bucle no sigui infinita.

Exemple: Fem un programa que utilitzi la funció *longitud* per veure la longitud d'un string. No val fer anar cap funció de *string.h*.

```
#include <stdio.h>

unsigned char longitud(char *);

main()
{
char cadena[81];
printf("Escriu una frase: ");
gets(cadena);
printf ("La teva frase te %u caracters\n", \
        longitud(cadena));
}

unsigned char longitud(char * s)
{
unsigned char llarg=0;
```

```
int i=0;
while (s[i]!='\0')
        {
        llarg++;
        i++;
        }
return llarg;
}
```

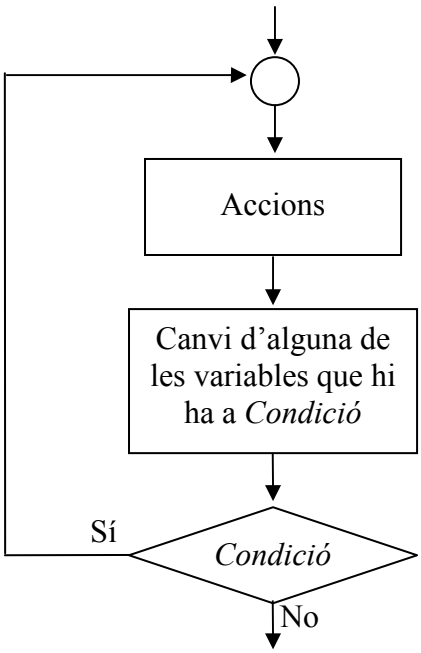
En aquest cas, la variable de la condició que fem canviar de valor dins el bucle és i, tot i que també haguéssim pogut escriure'l, el bucle, així:

```
while (s[i++]!='\0')
        llarg++;
```

Vigilem: una de les errades més freqüents i difícils de detectar quan es programa en C és deixar descuidat un punt i coma (;) al final d'una sentència de control.

Estructura do-while

Aquesta estructura de control executa un bloc d'instruccions i després avalua una condició. Si es compleix la condició es tornen a executar el bloc.
L'ordinograma podria ser:



Pel que fa a la sintaxi:

```
do{
     //instruccions
     }while (Condició);
```

Podem identificar aquest esquema a l'exemple d'ordinograma que hi ha a l'apartat 4.4, el de calcular la part entera dels logaritmes en base 2:

```
#include <stdio.h>
main()
{
int i, n, e;
do
        {
        printf ("Escriu un enter: ");
        scanf("%d",&n);
        }while (n<=0);
e=0;
i=n; //copiem n per poder-lo imprimir al final
while (i>1)
        {
        e++;
        i=i/2;
        }
printf("La part entera de log2(%d) es %d\n",n,e);
}
```

4.7 Ús de les estructures iteratives en el tractament de vectors

En aquest darrer capítol d'aquesta unitat, veurem dos esquemes clàssics de tractament de vectors: l'esquema de recorregut (volem fer alguna cosa amb tots i cada un dels elements d'un vector) i el de recerca (cerquem un determinat element d'un vector).

Recorreguts en vectors

Sovint volem tractar d'alguna manera tots els elements d'un vector. Partirem d'una declaració genèrica:

```
#define N dddd
...
int i;
tipus vect[N];
```

On *dddd* és un determinat nombre enter i *tipus* un dels tipus de dades elementals que coneixem.

Podrem fer un recorregut d'aquest vector si:

1) Sabem quants elements té el vector (en el nostre cas suposarem que N)
2) Hi ha un element E declarar `tipus E;` que marca el final del vector. Aquest element normalment s'haurà d'excloure del tractament.

En el primer cas, l'esquema que proposem és:

```
for (i=0;i<N;i++)
     //tractament de cada vect[i]
```

Mentre que per al segon cas, si *E* marca el final del vector i és un element especial que no cal tractar:

```
i=0;
while (vect[i]!=E)
     //tractament de cada vect[i]
```

Notem que en aquest darrer esquema és essencial que hagi un element que marqui el final, altrament no sortirem del bucle.

Hem vist un exemple de tractament amb marca de final en la funció *longitud* de l'exemple que hem posat en explicar els bucles *while*.

En els exercicis que proposarem segur que trobarem aplicacions de recorregut seguint el primer esquema.

Recerca en vectors

El problema en general és el següent: volem saber quina és la posició que ocupa l'element R en un determinat vector. Suposem que tenim les declaracions genèriques:

```
#define N dddd
...
int i;
tipus vect[N];
```

Proposem el següent esquema per fer la recerca:

```
i=0;
while ((vect[i]!=R)&&(i<N-1))
     i++;
if (vect[i]==R)
     //La primera aparició d'R a vect és a la posició i
else
     //No havia cap element de vect igual a R
```

Com a exemple, fem una funció que digui en quina posició d'un determinat text (s'entrarà el text com un string) es troba un determinat caràcter. La funció retornarà 0 si el caràcter no és al text.

```
#include <stdio.h>
#include <string.h>

unsigned char cerca(char *, char);

main()
{
char frase[81];
char lletra;
int pos;
printf("Escriu una frase: ");
gets(frase);
printf("Escriu un caracter: ");
```

```
scanf(" %c",&lletra);
pos=cerca(frase,lletra);
if (pos>0)
        printf ("La primera aparicio de %c en la frase es\
                    a la posicio %u\n", lletra, pos);
else
        printf ("El caracter %c no es a la \
                    frase\n",lletra);
}

/**********
unsigned char cerca(char * s, char c)
{
int i=0;
while ((s[i]!=c)&&(i<strlen(s)-1))
        i++;
if (s[i]==c)
        return i+1;
else
        return 0;
}
```

EXERCICIS ORIGINALS PROPOSATS A CLASSE

1. Feu un programa que us reculli nom i data de naixement i contesti quina edat teniu en anys. *(Cal fer servir la biblioteca time per saber quin dia és avui)*
2. Fes un programa que reculli nom d'usuari i password i si són iguals contesti "estàs violant una norma bàsica de seguretat".
3. Feu un programa que reculli una frase i contesti si és o no un palíndrom.
4. Feu un programa que reculli 10 enters, els guardi en un vector i després digui quins són els seus quadrats.
5. Feu un programa que reculli enters fins que se n'introdueixi un zero. Com a l'exercici anterior, els enters es posaran en un vector i després es mostraran els seus quadrats.
6. Feu un programa que reculli edats de persones en anys fins que s'introdueixi un -1. Ha de filtrar que no s'introdueixi cap valor inferior a –1 ni superior a, posem per cas, 120. Al final ha de mostrar un informe de quantes persones tenen entre 0 i 2 anys, quantes entre 3 i 5, quantes entre 6 i 11, quantes entre 12 i 15 i quantes 16 o més.
7. Feu un programa que reculli dues cadenes i en formi una de nova juxtaposant-les. No val fer ús de cap funció de la biblioteca string.
8. Sabem que per moure N discos en les torres de Hanoi d'un pal a un segon pal, primer n'hem de moure N-1 al tercer pal, per poder moure el disc més gran al segon i després hem de moure un altre cop N-1 al segon. Per tant, el nombre de mínim de moviments m_N que calen per moure N discos és $m_{N-1}+1+ m_{N-1}$ és a dir:
   $m_N = 2*m_{N-1}+1$. És fàcil veure que $m_1=1$ (per moure 1 disc només cal 1 moviment). Fes la taula "discos-nombre de moviments" per a un nombre de discos des d'1 fins a 30 (Primer, fes-la en paper fins a 4 discs, per entendre millor el problema).
9. Feu un programa que calcularà el màxim comú divisor de dos nombres així:
   a. Ordenareu els dos nombres en *menor* i *major*.
   b. Emplenareu un vector amb tots els divisors de *menor*, des de l'1 fins al propi *menor*. Marcareu el final dels divisors amb un zero.
   c. Emplenareu un altre vector amb tots els divisors de *major*. Com abans, marcareu el final amb un zero.
   d. Per a cada divisor de *menor*, començant des del final del vector, mirareu si hi és entre els divisors de *major*, moment en què haureu trobat el màxim comú divisor.

   Exemple:
   - L'usuari introdueix 60 i 48 i el nostre programa acaba assignant *menor*=48 i *major*=60.
   - S'omplen els vectors divisors_major: [1,2,3,4,5,6,10,12,15,20,30,60,0,…] i divisors_menor: [1,2,3,4,6,8,12,16,24,48,0,…}
   - Comencem per 48 i mirem si és a divisors_major, que NO hi és, després seguim amb 24, que tampoc no hi és, després 16 i finalment 12, que sí hi és.
   - El programa contestarà: el màxim comú divisor de 48 i 60 és 12.
10. Feu un programa que reculli dues cadenes i digui si la primera és subcadena de la segona i, si ho és, la posició que ocupa.