Programació en C

Lluís Alsedà

Departament de Matemàtiques
Universitat Autònoma de Barcelona
http://www.mat.uab.cat/~alseda

Versió 5 (7 d'abril de 2021)

Continguts

El primer programa

Índex

Un programa??

Un programa és un fitxer de text (*font* — creat amb un *editor*) que, compilant-lo, ens dona un *executable*. L'executable conté instruccions de processador, que s'executen quan el Sistema Operatiu les hi envia.

Bondia.c ⟶ `gcc␣-Wall␣-O3␣-o␣Bondia.exe␣Bondia.c` ⟶ Bondia.exe

- Bondia.c: És el fitxer *font*. Es crea amb un editor qualsevol.
- `gcc␣-Wall␣-O3␣-o␣Bondia.exe␣Bondia.c`: És la instrucció de compilació.
- Bondia.exe: És el programa (executable). S'executa com qualsevol altra comanda: Passant el seu nom al Sistema Operatiu.

Compilar??

```
gcc␣-Wall␣-O3␣-o␣Bondia.exe␣Bondia.c
```

- `gcc`: El programa que realment fa la compilació (traducció de codi font a codi màquina).
- `-Wall`: Diem que volem saber *tots* (*all*) els *avisos* (*W*arnings).
- `-O3`: És el nivell d'optimització.
- `-o Bondia.exe`: És el nom que volem donar al programa que es crea. Si no s'especifica obtenim a.out (sempre el mateix; per tant no sabem quin programa és i diversos programes es matxuquen entre ells).
- `Bondia.c`: és el nom del fitxer a compilar

Com s'executa el programa:

```
./Bondia.exe
```

Com entrar el codi font?

Exemple: el programa Bondia.c

```
// Programa bondia.c
#include <stdio.h>

int main ()
{
  printf ("Bon dia a tothom\n");
  return 0;
}
```

- Ull als errors tipogràfics

- *Totes les instruccions de **C** s'han d'acabar en punt i coma ;*

Ara cal "jugar" una mica amb el compilador ...

Estructura d'un programa

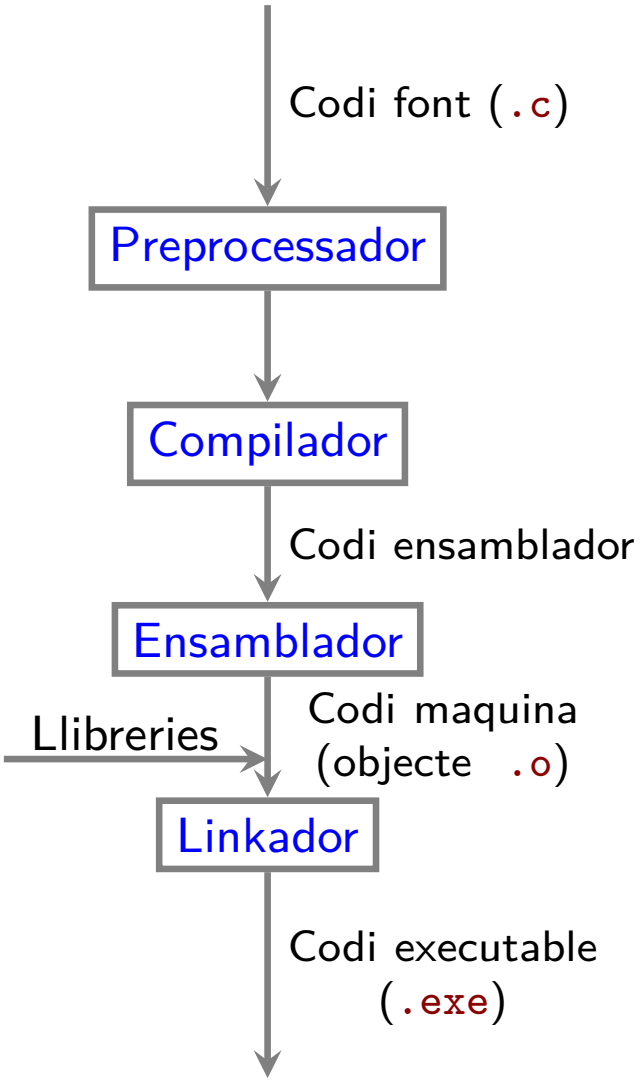
```
/*  Capçalera d'identificació del programa i l'autor */

/* Fitxer:     nom.c
   Contingut:  Exemple d'estructura d'un programa en C
   Autor:      nom de l'autor
   Revisió:    preliminar; data de la darrera modificació     */

/*  Comandes del preprocessador. Són de dos tipus             */
/*  Tipus 1: inclusions de fitxers de capçalera. Per exemple: */

#include <stdio.h>

/*  Tipus 2: definició de constants simbòliques. Per exemple: */

#define PI 3.14159265358979323846264338327950288419 7

int main()  /* Funció principal. Hi poden haver paràmetres    */
            /* El programa es comença a executar per aquí     */
{
/* Declaracions. Per exemple:                                 */
   double r;

     /* cos de la funció principal; instruccions del programa */

}  /* Fi del main */

/* FUNCIONS AUXILIARS                                         */

   . . .
```

Un exemple que ja fa càlculs: el programa area.c

El següent programa calcula l'àrea d'un cercle de radi *r*
El radi es llegeix del teclat

```
// Programa area.c
#include <stdio.h>

#define PI 3.14159265358979323846264338327950288419 7

int main ()
{
  double r;

  printf("Entra el radi del cercle: ");
  scanf ("%lf", &r);

  printf("\nL'àrea és: %14.6f\n\n", PI*r*r);

  return 0;
}
```

Entenent el procés de compilació: Que passa quan executem la instrucció `gcc`?



A la figura es mostren les diferents etapes que realitza el `gcc` per a obtenir el codi executable.

- **Preprocessador**: Accepta el codi font com a entrada, treu els comentaris i interpreta les directives del preprocessador que s'inicien amb `#` (concretament les instruccions `#include` i `#define` que ja hem vist).
- **Compilador**: Tradueix el codi font que rep del preprocessador a llenguatge ensamblador.
- **Ensamblador**: Crea el programa en codi màquina (objecte) a partir del programa en ensamblador.
- **Linkador**: Quan el programa fa referencia a funcions d'una biblioteca, el linkador afegeix el codi màquina d'aquestes funcions al programa principal per a crear un fitxer executable autònom (que inclou el codi de totes les funcions utilitzades — per exemple s'incorpora el codi de la funció `printf`).

La llibreria matemàtica

Quan dins d'un programa volem usar alguna de les funcions matemàtiques del **C**:

```
acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs,
floor, fmod, frexp, ldexp, log, log10, modf, pow, sin,
sinh, sqrt, tan, tanh
```

cal que fem que el nostre programa conegui com estan definides aquestes funcions, quin tipus de valor retornen, quants paràmetres tenen, de quins tipus són, ...

Això es fa, de manera similar a com ho hem fet per les funcions `printf` i `scanf`, afegint el fitxer de capçalera `math.h`.

Exemple: calculem l'arrel quadrada d'un nombre:

```
#include <stdio.h>
#include <math.h>
int main () { double r;
  printf("Entra el nombre r: "); scanf ("%lf", &r);
  printf("\nL'arrel quadrada de r és: %14.6f\n\n", sqrt(r));
  return 0;
}
```

La llibreria matemàtica: Compilació

A diferència del que passa amb les funcions de la llibreria `stdio`, el codi de les funcions de la llibreria matemàtica no s'afegeix automàticament al nostre programa quan linkem. Aquest procés cal fer-lo explícitament am la comanda `gcc` afegint l'opció `-lm`:

```
gcc␣-Wall␣-O3␣-o␣Bondia.exe␣Bondia.c␣-lm
```

L'opció `-l` del `gcc`

Quan afegim l'opció `-lllibreria` ò `-l␣llibreria` a la comanda de compilació, el `gcc` busca la llibreria anomenada `libllibreria.a` i l'afegeix al nostre programa (és a dir permet que el linkador busqui dins de `libllibreria.a` — a més de les altres llibreries que es carreguen automàticament — el codi de les funcions que cridem al nostre programa, afegint aquest codi a l'executable).

Donat que la llibreria matemàtica es diu `libm.a`, l'opció `-l␣m` afegida a la comanda de compilació, posa a disposició del linkador la llibreria matemàtica.

Índex

Els tipus de dades més comuns i la seva precedència

Especificació	# bytes (8 bits)	Rang	Precisió
unsigned char	1	0 — 255	
(signed) char	1	-128 — 127	
unsigned short (int)	2	0 — 65535	
(signed) short (int)	2	-32768 — 32767	
unsigned (int)	4	0 — 4294967295	
(signed) int	4	-2147483648 — 2147483647	
unsigned long (int)	8	0 — 18446744073709551615	
(signed) long (int)	8	-9223372036854775808 — 9223372036854775807	
(signed) long long (int)	8	-9223372036854775808 — 9223372036854775807	
unsigned long long (int)	8	0 —18446744073709551615	
float	4	1.175494×10^{-38} — $3.402823 \times 10^{+38}$	7 dígits
double	8	$2.225074 \times 10^{-308}$ — $1.797693 \times 10^{+308}$	15 dígits
long double	16	$3.362103 \times 10^{-4932}$ — $1.189731 \times 10^{+4932}$	15 dígits

Nota

La funció `sizeof( )` torna el número de bytes d'un tipus de dada.

Exemple

`sizeof(unsigned long);` torna el valor `8`.

Per més detalls veieu el programa de la pàgina ▸ 115, que és el que ha generat aquesta taula.

Constants

Constants	Tipus de dades compatibles
'A' '3' '\t'	char
3	char short int long unsigned
-27	char short int long
-100000	int long
4294967200	unsigned int unsigned long
3. 3.0	float double long double
-0.03 -3.0e-2	float double long double

Declaració de variables

Qualsevol declaració de variable consistirà en una especificació de tipus: `char`, `int`, `float`, ...
precedida d'un modificador: `unsigned`, `signed`, `short`, `long`
a menys que aquest últim ja indiqui clarament el tipus de la variable.

Exemple

```
unsigned short natural;  /* La variable 'natural' es declara */
                         /* com un enter positiu             */
int        i, j, k = 1;  /* Les variables 'i' 'j' 'k'        */
                         /* seran enters (amb signe).        */
```

Aquí s'han fet *dues* operacions diferents simultàniament:

- Declaració de variables (que consisteix en la reserva de memòria que ocuparà la variable i l'etiquetatge del "calaix" de memòria que ocupa).
- Inicialització ($k = 1$).

Els operadors i la seva precedència

operadors aritmètics:	%		mòdul (només per enters)
	*	/	producte i divisió
	+	−	suma i resta
operadors relacionals:	>	>=	més gran (o igual) que
	<	<=	més petit (o igual) que
	==	!=	igual i diferent
operadors lògics:	!		*no* (proposició lògica)
	&&		*i* lògic
	||		*o* lògica

El tipus del resultat d'una operació

En operar dues variables o constants el tipus del resultat coincideix amb el tipus superior dels operands.

Nota: L'ordre de precedència dels tipus de dades està determinat per la taula de la Pàgina 11 Un tipus és superior a un altre sempre que estigui mes avall a la taula.

Exemples

Suposem que tenim:	El resultat de:
`int a = 2, b = 3;`	`a*b` és de tipus `int`
`float c = 5.0;`	`b*c` és de tipus `float`
`double d = 7.5;`	`d*a` és de tipus `double`
	`d*a` és de tipus `double`

Uiii!!: això té conseqüències sorprenents

Als exemples anteriors `c = c / a;` modifica correctament el valor de `c` a `2.5`.

En canvi, `c = b / a;` modifica erròniament el valor de `c` a `1.0`.

Que ha passat?: El resultat de `b / a;` és de tipus `int` i s'obté per truncament. Per tant el vertader valor de `b / a = 1.5` es trunca a `1`. Llavors, com que `c` és de tipus `float` l'`int 1` es converteix automàticament a `float 1.0`.

La coerció de tipus de dades

Una manera d'arreglar el problema de la pàgina anterior és fer

`c = (a*1.0) / b`.

Això és molt ineficient i té el problema que no es pot decidir si el resultat és `float` o `double`. La manera correcta de fer-ho és *forçant el canvi de tipus*.

La instrucció general és:

El forçament de tipus (`cast`)

```
( especificacio_tipus ) operand
```

Exemples

- `(double) n` dona com a resultat el valor de `n` convertit a `double`.
- `(int) a` dona com a resultat el valor de `a` truncat a `int`. És a dir, torna la part entera de `a` (si `a` és positiu).

Precedència dels operadors en C

Operador	Significat
`( )`	Crida a funció
`[ ]`	Accés a component de vector
`. ->`	Accés a camp d'estructura
`!`	Negació lògica
`++ --`	Autoincrement i autodecrement
`+ -`	Indicació i canvi de signe
`* \&`	Indirecció i adreça
`(tipus)`	Coerció de tipus
`sizeof(tipus)`	Número de *bytes*
`* / \%`	Producte, divisió i mòdul
`+ -`	Suma i resta
`< <= > >=`	Comparació de nombres
`== !=`	Igualtat i diferència lògica
`\&\&`	*i* lògica
`||`	*o* lògica
`? :`	Expressió condicional
`=`	Assignació
`+= -=`	Assignació amb operació
`*= /= \%=`	Assignació amb operació
`,`	Concatenació

A la taula es poden observar els operadors (ordenats de més a menys prioritat, amb els grups de prioritat separats per línies horitzontals) i el seu significat. Evidentment, els parèntesis es poden usar per trencar la prioritat d'un determinat conjunt d'operadors en una expressió. És molt recomanable emprar parèntesis en els casos en que l'ordre d'avaluació no quedi clar sense consultar la taula de precedència d'operadors.

Índex

Condicionament: l'if

```
if ( condició )  instrucció;
else             instrucció;
```

```
if ( condició ) {
                 instrucció;
                   . . .
                 instrucció;
} else {
                 instrucció;
                   . . .
                 instrucció;
} /* fi de l'if */
```

Les solucions de l'equació de segon grau

Suposem que tenim $ax^2 + bx + c$.

```
disc = b*b - 4*a*c;
if(disc < 0) printf("Discriminant negatiu. No hi ha arrels reals.\n");
else {
  disc = sqrt(disc); a = 2*a;
  printf("Les arrels són: %f i %f\n", (-b+disc)/a,(-b-disc)/a);
}
```

Una condició més complicada

```
if(((a > 0) && (d*c <= a)) || (d > 0 && d < 4))
     printf("Les variables a, d i c són correctes\n");
```

Iteració: bucles

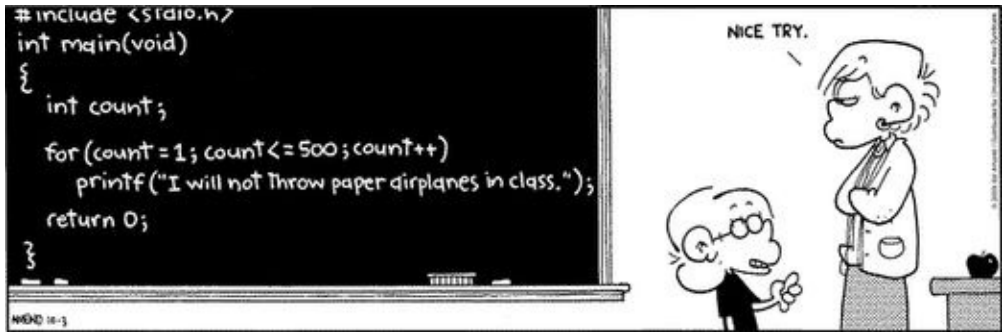
```
for (
      inicialització; /* Per ex.: i = 0 */
      condició;       /* Per ex.: i < LIMIT */
      continuació;    /* Per ex.: i = i + 1 */
    ) {
          instrucció;
             . . .
          instrucció;
}  /* fi del for */
```

```
while ( condició )  {
          instrucció;
             . . .
          instrucció;
}  /* fi del while */
```

```
do {
     instrucció;
        . . .
     instrucció;
} while ( condició );
```

Exemple

```
pot = 1;
for (i = 0; i <= n; i++){
   printf ("3^%d = %d\n", i, pot);
   pot = pot * 3;
}
```



Equivalència entre el bucle `for` i el `while`

```
for ( inicialització;
      condició;
      continuació;
    ) {
          instrucció;
             . . .
          instrucció;
}  /* for */
```

```
inicialització;
while( condició )
{
     instrucció;
        . . .
     instrucció;
     continuació;
}  /* while */
```

Un exemple

```
pot = 1;
for (i = 0; i <= n; i++){
   printf ("3^%d = %d\n", i, pot);
   pot = pot * 3;
}
```

```
i=0; pot=1;
while(i <= n){
  printf ("3^%d = %d\n", i, pot);
  pot = pot * 3; i = i + 1;
}
```

Els `for` es poden compactar molt: l'exemple anterior en forma compacta

```
for (i = 0, pot=1; i <= n; i++, pot *= 3)
                  printf ("3^%d = %d\n", i, pot);
```

Control de flux dels bucles: `break;` i `continue;`

El **C**, incorpora instruccions que ens permeten programar una sortida forçada de bucle (`break;`) i la continuació forçada del bucle (`continue;`).

Nota

També existeix la instrucció de salt incondicional `goto`, que, per a conservar la programació estructurada, no s'hauria d'utilitzar mai.

Problema d'exemple

Per n variant de 1 a 20 i a enter positiu calcula $n/(n^2 - a * n + 375)$; sempre que el denominador sigui positiu.

Problema d'exemple (variant)

Fer el càlcul solament fins la primera vegada que el denominador no és positiu.

Una solució del problema d'exemple

El programa bucles.c

```
// Programa bucles.c
#include <stdio.h>

void main () {
  int a, n;

  printf("Entra el paràmetre a: "); scanf("%d",&a);

  for (n = 1; n <= 20; n++){ int denom;
     denom = n*n -a*n + 375;
     if(denom > 0)
       printf ("Formula per n=%d: %lf\n", n, ((double) n)/denom);
  }
}
```

Una solució del problema d'exemple

El programa bucles.c versió 2

```
// Programa bucles2.c
#include <stdio.h>

void main () {
  int a, n;

  printf("Entra el paràmetre a: "); scanf("%d",&a);

  for (n = 1; n <= 20; n++){ int denom;
     denom = n*n -a*n + 375;
     if (denom <= 0) continue;
     printf ("Formula per n=%d: %lf\n", n, ((double) n)/denom);
  }
}
```

Una solució del problema d'exemple

El programa bucles.c versió 3

```
// Programa bucles3.c
#include <stdio.h>

void main () {
  int a, n;

  printf("Entra el paràmetre a: "); scanf("%d",&a);

  for (n = 1; n <= 20; n++){ int denom;
     denom = n*n -a*n + 375;
     if (denom <= 0) break;
     printf ("Formula per n=%d: %lf\n", n, ((double) n)/denom);
  }
}
```

Índex

El preprocessador

Ja hem vist que el preprocessador permet d'incloure els fitxers de capçalera de les funcions i definir constants simbòliques.

```
#include <fitxer-de-capçalera.h>
#include "fitxer-de-capçalera-propi.h"
#define símbol expressió_constant
```

Exemples

```
#include <stdio.h>
#define DIMENSIO 2
#define PI 3.14159265358979323846264338327950288419 7
#define TOL 1.0e-6
```

El preprocessador funciona com un editor. Substitueix les definicions pel seu valor en tot el text. Per exemple substitueix `PI` per `3.14159265358979323846264338327950288419 7` en tot el text com si el programador ho hagués escrit explícitament així.

Macros

A més, el preprocessador permet algunes substitucions amb paràmetres (les “macros"), en contraposició a les substitucions literals de les definicions constants vistes fins ara.

```
#define nom_de_macro(arguments) expressio_amb_arguments
```

L'ús de les macros ajuda a aclarir petites parts de codi emprant una “espècie de funció"amb algun nom significatiu de manera que la lectura en sigui més fàcil i no impliqui una crida real a una funció amb un cos de codi que no la justifiqui com a tal. En aquestes construccions acostuma a ser útil

l'*if aritmètic*

```
(condició) ? valor-si-veritat : valor-si-fals
xx = (x < 0 ? -x : x) ; // xx és el valor absolut d'x
```

Exemples

```
#define abs(x) (x < 0 ? -x : x)
#define round(x) ((int) ((double) x + 0.5 ))
#define trunc(x) ((int) (x))
#define MAX(x,y) ((x<y) ? y : x)
#define PI 3.14159265358979323846264338327950288419 7
#define quadrat(x) x*x
#define area_cercle(r) PI*quadrat(r)
```

Cal tenir en compte que el nom de la macro i el parèntesi esquerre no poden anar separats i que la continuació d'una línia es fa col · locant una barra inversa just abans del caràcter de salt de línia (en cas que la comanda de preprocessador sigui massa llarga).

En el cas de les macros, el preprocessador també funciona com un editor substituint les definicions pel seu valor en tot el text. En aquest cas, però, cal que ho faci de manera intel · ligent tenint en compte, en cada cas, els paràmetres de la macro.

Exemple: `MAX(a,3)` es substitueix per: `((a<3) ? 3 : a)`.

Índex

Utilitat de les funcions

Eficiència:

Si cal fer el mateix procediment a llocs diferents del programa, el podem codificar una única vegada en una funció i utilitzar als llocs que calgui cridant la funció

Estructura—Modularitat:

La creació de funcions permet dividir el programa en parts per modularitzar-lo. Aquesta descentralització millora la comprensió i fa més fàcil el disseny del programa i el seu depurat d'errors.

Seguretat:

Cada funció es pot provar i consolidar per separat. En cas d'error podem descartar les funcions consolidades.

Estructura de les funcions

```
tipus nomfuncio ( llista-parametres )
{
// Declaracions de variables locals

// Instruccions de la funció

   return (expressió);
} /* Fi de la funció */
```

El valor de retorn

Les funcions retornen un valor del `tipus` especificat a la declaració de la funció. El valor concret el retorna la instrucció `return` i és el resultat de l'avaluació de l'expressió associada.

Exemple : La funció que calcula $\frac{1}{1+x^2}$

Torna l'avaluació d'aquesta funció com a `double`

```
double f( double x )
{
  return 1.0/(1.0 + x*x);
} /* f */
```

Exemple complet: el programa grafica.c

Treu per pantalla una taula de la funció $\frac{1}{1+x^2}$ amb la x de 0 a 10 en intervals de 0.01

```
// Programa grafica.c
#include <stdio.h>

/* Prototipus de la funció f com als fitxers de capçalera ('.h')
   Serveix perquè el compilador pugui comprovar que utilitzem
   la funció correctament                                    */
double f( double );

void main ()
{
  double x;

  for(x=0.0; x<= 10.0 ; x+= 0.01) printf("%f %f\n",x,f(x));
}

double f( double x )
{
  return 1.0/(1.0 + x*x);
} /* f */
```

Les variables del C: variables locals i globals

En el programa anterior, el fet que la variable es digui x al programa principal i a la funció no és important.

No són la mateixa variable ni ocupen les mateixes posicions de memòria ja que *les variables en C són locals del mòdul (estructura* `{ ... }`*) on viuen!!* (i per tant solament són conegudes i utilitzables dins d'aquest mòdul).

A més, en **C**, els paràmetres de les funcions es passen *per valor*. Això vol dir que, quan es crida la funció, el valor de la variable x del programa principal es copia al contingut de la variable x de la funció. En particular:

- La variable x del programa principal *és diferent* de la variable x de la funció.
- Si modifiquem la variable x a dins de la funció, la variable x al programa principal no s'altera.

Exemple sobre variables locals i globals: el programa grafica2.c

```
// Programa grafica2.c
#include <stdio.h>

double f( double );
double uno=1.0; // Aquesta variable es global de tot el programa

void main ()
{ double x = 2.0;

  printf("Uno: %lf\n", uno); // Correcte: variable coneguda
  printf("Dos: %lf\n", dos); // Error de compilat: la variable 'dos' solament es coneix mes avall
  printf("Aux: %lf\n", aux); // Error de compilat: 'aux' solament es coneix dins del bloc següent
  printf("z: %lf\n", z); // Error de compilat: 'z' solament es coneix dins la funció f

  { double aux=2*x;
    printf("Aux: %lf\n", aux); // Correcte: variable coneguda
// Observem que "x abans" = "x després"
    printf("x abans = %f; Funció = %f; x després = %f\n",x,f(x),x);
  }
  printf("Aux: %lf\n", aux); // Error de compilat: 'aux' solament es coneix dins el bloc anterior
}

double dos=2.0;  // Aquesta variable solament és coneguda a partir d'aquí

double f( double z )
{
  z = dos/(uno + z*z); // Correcte aquí coneixem uno i dos. Redefinim la z.
  return z;
} /* f */
```

Conseqüències importants del fet que els paràmetres es passen per valor

Com ja hem dit, en **C**, els paràmetres de les funcions es passen per valor. És a dir, quan es crida la funció, el valor de cada paràmetre del programa principal es copia al contingut del paràmetre corresponent de la funció, i la variable corresponent del programa principal no s'altera. A l'exemple anterior:

```
double f( double z )
{
  z = 2.0/(1.0 + z*z); // Redefinim la z.
  return z;
} /* f */

printf("x abans = %f; Funció = %f; x després = %f\n",x,f(x),x);
```

la `x` abans de cridar `f(x)` coincideix amb el valor de `x` després de cridar la funció malgrat que el valor del paràmetre corresponent (`z`) es modifica dins de la funció.

Funcions que tornen més d'un valor — Motivació

Una limitació òbvia de les funcions tal com es defineixen és que tenen un únic *valor de retorn* i per tant, *en principi*, solament poden modificar una *única variable* del mòdul que les crida (programa principal o una altra funció).

Això és una limitació important i, en alguns cassos, és una limitació inassumible.

Donat que els paràmetres d'una funció no es poden modificar ja que en **C** es passen per valor, aquest problema es pot resoldre de dues maneres:

- usar paràmetres que permetin modificar les seves variables associades, i
- passar com a paràmetres objectes multidimensionals indexats com són *vectors i matrius*.

La tecnologia per a les dues solucions anteriors passa pel mateix lloc: l'ús d'*apuntadors*. En el primer cas els apuntadors proporcionen una drecera per modificar diverses variables al mòdul que crida la funció sense canviar el fet que els paràmetres no es modifiquen degut a que es passen per valor.

Índex

Definició

Els vectors i matrius són *tires* de variables que permeten emmagatzemar conjunts de dades en nombre variable.

Exemple: Els coeficients d'un polinomi, els resultats d'un experiment per fer-ne estadística, ...

Nota

Les matrius $m \times n$ es representen com a vectors de llargada $m \cdot n$ escrits per files.

Nota

*En **C** els vectors i matrius sempre comencen per l'índex 0*. Per tant, si declarem un vector `v` de 5 components, aquestes són `v[0]`, `v[1]`, `v[2]`, `v[3]` i `v[4]`.

Inicialització i declaració de vectors i matrius

Exemples

```
double v[5] = { 1.0, 1.1, 1.2, 1.3, 1.4 };
int matriu[3][3] = { 00, 01, 02, 10, 11, 12, 20, 21, 22 };
int m[3][3] = {                    /* o bé, més recomanable ... */
                { 00, 01, 02 },
                { 10, 11, 12 },
                { 20, 21, 22 }
              };
/*           Inicialització incompleta (permesa)              */
int  v[30]  = { 20, 21, 22, 23, 24, 25, 26 };
```

Els vectors i matrius es representen en estructures de dades consecutives a la memòria:

1506

v[0]	v[1]	v[2]	v[3]	v[4]
1.0	1.1	1.2	1.3	1.4

00c0702d

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]	m[2][0]	m[2][1]	m[2][2]
00	01	02	10	11	12	20	21	22

Crida dels elements de vectors i matrius

Per fer referència a un element d'un vector o matriu, només cal indicar el nom i la posició de l'element en particular al qual volem accedir:

```
v[i]
matriu[i_0][i_1]...[i_n]
```

la `i` i les `i_k` han de ser variables, constants o expressions el resultat de les quals siguin enters.

Nota

El C no comprova que al referenciar un element d'un vector o matriu no ens passem del rang de dimensionament.

Exemple: La instrucció `v[27] = 74;` és perfectament vàlida independentment de la dimensió de `v`.
El programa, de fet, còpia la constant `74` a la zona de memòria que és 27 llocs més enllà de l'inici del vector, independentment de si aquesta zona de memòria correspon a `v` o no. Això, clarament, pot comprometre l'execució del nostre programa i, probablement, el funcionament del sistema operatiu.

Exemples senzills: omplir i escriure vectors i matrius

El programa matrius.c

```
#include <stdio.h>
#define DIMENSIO 2

int main()
{ int a[DIMENSIO][DIMENSIO];
  int i,j;
  puts("Entrem la matriu");
  for (i=0; i < DIMENSIO ; i++){
     for (j=0; j < DIMENSIO ; j++){
        printf("a(%d,%d) = ? ",i+1,j+1);
        scanf("%d",&(a[i][j]));
     }
  }
  puts("Ara imprimim la matriu:");
  for (i=0; i < DIMENSIO ; i++){
     for (j=0; j < DIMENSIO ; j++){
        printf("%d ",a[i][j]);
     } ; printf("\n");
  }
 return 0;
}
```

El programa vectorsf.c
L'escriptura la fa una funció

```
#include <stdio.h>
#define DIMENSIO 2
void imprimeixvector (float [], int);

int main() { int i; float v[DIMENSIO];
  puts("Entrem el vector");
  for (i=0; i < DIMENSIO ; i++){
        printf("v(%d) = ? ",i+1);
        scanf("%f",&(v[i]));
  }
  puts("Ara l'imprimim:");
  imprimeixvector (v, DIMENSIO);
  return 0;
}

void imprimeixvector (float vect[], int dim){ int i;
  for(i=0; i < dim ; i++) printf("%f ",vect[i]);
  printf("\n");
  return;
 }
```

Els vectors com a paràmetres de funcions

Els vectors es passen directament com a paràmetres de funcions indicant que són objectes indexats. El vector (la seva adreça de memòria) no és modificable per la funció; els seus elements si que ho són.

El programa `matrius.c` revisitat amb l'escriptura feta per una funció

El programa `matrius.c` revisitat

```
#include <stdio.h>
#define DIMENSIO 20

void imprimeixmatriu (float [][DIMENSIO], int);

int main() { int i,j, dim;
  float a[DIMENSIO][DIMENSIO];

  printf("Entra la dimensió <= %d\n",DIMENSIO);
  scanf("%d", &dim);
  if(dim > DIMENSIO) {
      puts("Error: dimensió massa gran");
      return 1;
  }

  puts("Entrem la matriu");
  for (i=0; i < dim ; i++){
     for (j=0; j < dim ; j++){
        printf("a(%d,%d)=?",i+1,j+1);
        scanf("%f",&(a[i][j]));
     }
  }

  puts("Ara imprimim la matriu:");
  imprimeixmatriu (a, dim);
  return 0;
} /* Fi del main */
```

La funció `imprimeixmatriu`

```
void imprimeixmatriu (float mat[][DIMENSIO],
                      int dim) { int i, j;
   for (i=0; i < dim ; i++){
     for (j=0; j < dim ; j++){
        printf("%f ",mat[i][j]);
     } ; printf("\n");
  }
  return;
}
```

Les matrius com a paràmetres de funcions

Les matrius es passen directament com a paràmetres de funcions indicant que són objectes indexats i *especificant la segona dimensió de la matriu* tal com ha estat declarada (això es justificarà més endavant en estudiar l'aritmètica d'apuntadors).
La matriu (la seva adreça de memòria) no és modificable per la funció; els seus elements si que ho són.

Índex

L'habitat de les variables: La memòria

En programar en **C** la memòria es pot visualitzar com una tira (lineal) llarguiiiiiiiiiiiiiissima (això depèn de l'ordinador ☺) de posicions binàries anomenades *bits* i agrupades en blocs de 8 bits anomenats *bytes*.

Una visió bàsica de les variables

Una variable és de fet un nom o etiqueta per una zona de la memòria. L'associació entre una variable i la seva zona de memòria assignada es produeix en el moment de *declarar* la variable.

Quan es declara una variable se n'especifica un *tipus* que determina la mida (en bytes) de la zona de memòria assignada i com s'interpreta el seu contingut (que sempre està escrit en binari).

Les variables i el seu habitat: Exemple

Declaracions

Les variables es declaren amb un tipus associat:

```
modificador tipus nom
```

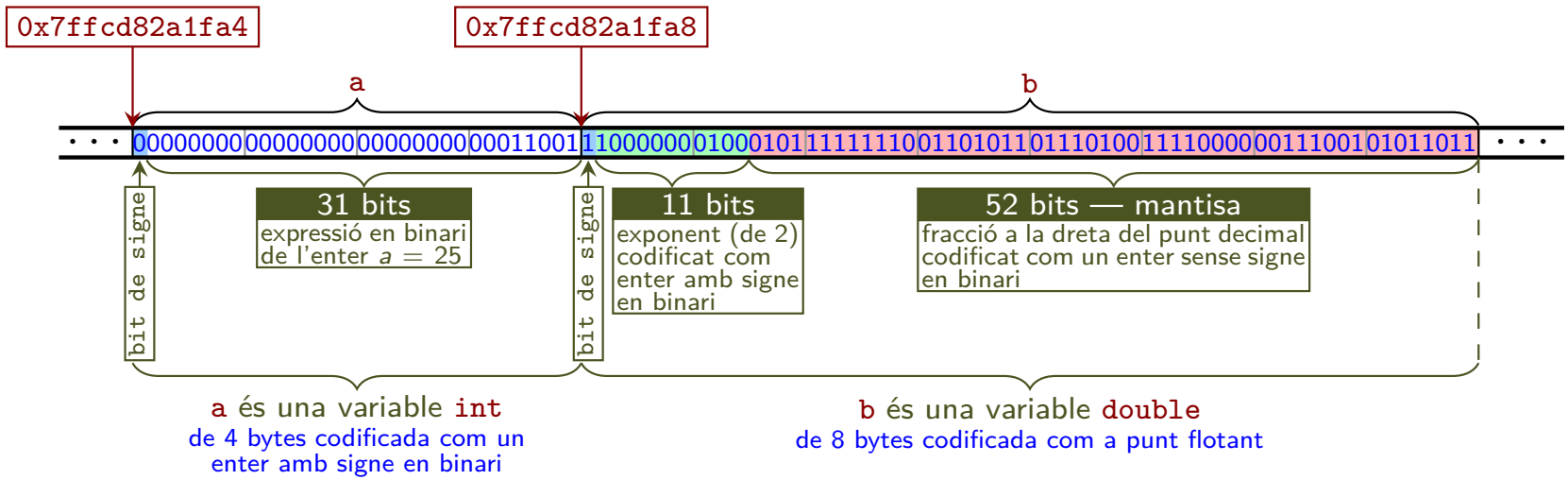
El tipus d'una variable determina la mida (en bytes) de la zona de memòria que ocupa i com es codifica el seu *contingut*, que sempre està escrit en binari.

Exemples

```
unsigned long ull; // Declaració amb modificador
int a = 25;        // Declaració i assignació
double b = -43.987654321; // Declaració i assignació
```

La memòria

s'organitza linealment i s'indexa per enters que, normalment, s'escriuen en hexadecimal.



Les variables i el seu habitat: Definicions

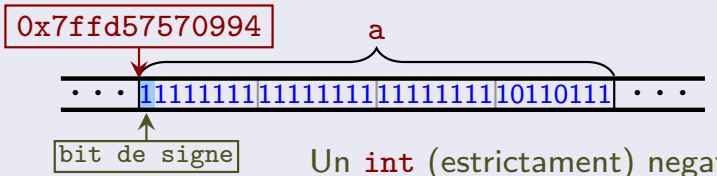
Terminologia: *contingut*

- El *contingut* d'una variable és el valor emmagatzemat a la zona de memòria que ocupa.
- La *mida* d'aquesta zona de memòria (el seu nombre de bytes) està totalment determinada pel *tipus* de la variable (`sizeof(tipus)`).

  Exemple: A la figura, la mida de la variable `a`, que ha estat *declarada* com `int`, és `sizeof(a) = sizeof(int) = 4` bytes. Anàlogament, la mida de la variable `double b` és `sizeof(b) = sizeof(double) = 8` bytes.
- La *codificació/descodificació* del codi binari emmagatzemat a la zona de memòria que ocupa també està determinat pel *tipus* de la variable.

Exemple: Codificació d'un `int`

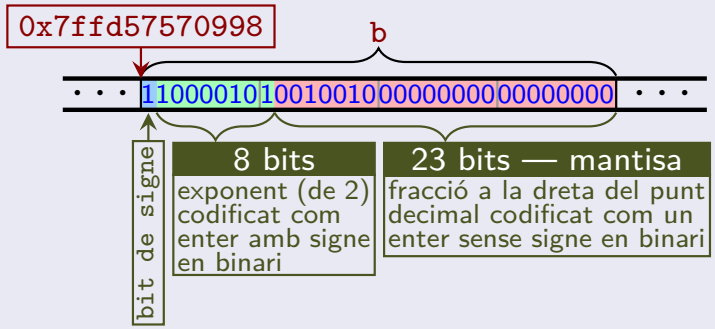
```
int a = -73;
```



Un `int` (estrictament) negatiu es codifica en binari com el *complement a 2*† de $-a$.

Exemple: Codificació d'un `float`

```
float b = -73.0;
```



†El *complement a 2* d'un enter $x > 0$ es defineix com $2^{(8 \texttt{ * sizeof(x)})} - x$ i es calcula més fàcilment invertint (negant) els bits de l'expressió en binari d'x: `~x` (equivalentment, fent les substitucions $0 \mapsto 1$ i $1 \mapsto 0$ a cada bit — *complement a 1*) i sumant 1 al resultat. **Comprovació:** 73 en binari és $00\cdots001001001$.

Les variables i el seu habitat: Definicions

L'operador de *direcció* (o d'*adreça*) &

La *direcció* d'una variable és l'índex del lloc de memòria on comença la zona reservada d'emmagatzemament de la variable en temps d'execució.
La direcció d'una variable `var` es denota (i s'obté) amb: `&var`.

Nota insistent

- A diferència de la mida i la codificació d'una variable, la direcció no depèn del seu tipus. Depèn solament de la situació i organització de la memòria de l'ordinador en executar el programa.
- L'operador `&` solament està ben definit quan s'aplica a una variable.
- En principi no es poden `&`-titzar expressions aritmètiques encara que involucrin operadors.

Exemple

A les figures de la pàgina anterior tenim:
direcció(a) = `&a = 0x7ffd57570994`;
direcció(b) = `&b = 0x7ffd57570998`;

Observació: Incidentalment, `&b = &a + 4`, que és un fet normal quan hi ha memòria de sobres, `b` es declara immediatament després d'`a` i `sizeof(a) = 4`.

Què és un apuntador?

És una variable que està definida (declarada) per a contenir les direccions on comencen altres variables.

També es consideren apuntadors les expressions aritmètiques que involucren apuntadors i, potser, altres variables o constants (aritmètica d'apuntadors).

Definició: *valor* d'un apuntador

El *valor* d'un apuntador és l'adreça de memòria que especifica.

Declaració d'apuntadors

Declaració d'apuntadors

Una variable apuntador es declara com totes les altres variables però el seu identificador (nom) va precedit d'un asterisc `*`:

```
modificador tipus * nom_apuntador
```

Opcionalment es pot fer la declaració i la inicialització de l'apuntador simultàniament:

```
modificador tipus * nom_apuntador = expressió;
```

Definició: tipus d'un apuntador — Els apuntadors tenen el seu caràcter

Un apuntador declarat com en el block anterior es diu que té *tipus* `modificador tipus *` o, equivalentment, que és un *apuntador a* `modificador tipus`.

Exemples

```
int * punt = &x;      // Apuntador a int inicialitzat a &x
char * car;           // Apuntador a caràcter
unsigned int * uu;    // Apuntador a unsigned int
float * f;            // Apuntador a float
double * num = NULL; // Apuntador a double inicialitzat a NULL
```

Apuntadors: definicions

Definicions: Els apuntadors apunten

Un apuntador es diu que *apunta* a la variable que comença a la direcció de memòria especificada pel valor de l'apuntador.

Els apuntadors mantenen el tipus

La variable *apuntada* per un apuntador de tipus `modificador tipus *` és, per definició, de *tipus* `modificador tipus`.

Més precisament: la zona d'emmagatzematge en memòria de la variable *apuntada* per un apuntador de *tipus* `modificador tipus *` ocupa els `sizeof(modificador tipus)` bytes que hi ha a continuació del *valor (direcció)* de l'apuntador, i la seva codificació en binari està determinada pel tipus `modificador tipus` de la variable apuntada (veure les pàgines 47 i 48).

Exemple: `float * f;`

L'apuntador `f` apunta a una variable `float` (per tant de `sizeof(float)` = 4 bytes), que comença a la direcció continguda a `f` (valor d'`f`), i que està codificada com a la pàgina 48.

Exemple: `unsigned int * uu;`

L'apuntador `uu` apunta a una variable `unsigned int` (per tant de `sizeof(unsigned int)` = 4 bytes), que comença a la direcció continguda a `uu` (valor de `uu`), i que està codificada en binari.

Declaració d'apuntadors

Observacions

- De la definició de tipus d'un apuntador es dedueix que *hi ha tants tipus d'apuntadors com tipus de dades*.
- El tipus d'un apuntador `modificador tipus *` és un tipus de dada.
- Per tant `(modificador tipus *) *`, que és el tipus *apuntador a apuntadors a* `modificador tipus` és un nou tipus de dades.
- Recursivament hi ha infinits tipus de dades i infinits tipus d'apuntadors.

Funcionament i operadors dels apuntadors

L'operador de *contingut* o *indirecció* *

L'operador `*` aplicat a un apuntador indica el contingut de la variable apuntada, que és del tipus determinat per l'apuntador.

L'operador indirecció es pot usar tant per a llegir el valor contingut a la variable apuntada com per assignar-n'hi un de nou.

Exemple: Si `punt` és el nom d'un apuntador, `*punt` representa el contingut de la variable apuntada per `punt`.
Podem fer tant: `printf("%d\n", *punt);`
com: `*punt = 27;` ò `*punt = a*27 - 34;`

Nota

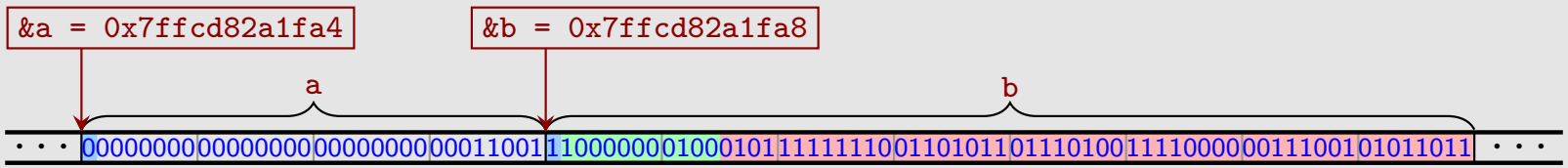
Observeu el significat diferent dels dos `*` a l'expressió `*punt = a*27 - 34;`
El primer representa la indirecció de l'apuntador `punt`. El segon és l'operador producte de `a` per `27`. Els dos cassos es diferencien sense ambigüitat pel context. Si cal ens hem d'ajudar de `( )` per a clarificar les expressions i treure ambigüitats aparents: en lloc d'escriure `b = a * *punt;` és molt més clar si escrivim `b = a * (*punt);`.

Exemples detallats:
Que es pot i no es pot fer amb els operadors & i *

```
int a = 25;
```

`a` ⟶ 25 `&a` ⟶ 0x7ffcd82a1fa4 `*(&a)` ⟶ 25

`*a` ⟶ **ERROR**: `a` no és un apuntador



`&a`=0x7ffcd82a1fa4 **ERROR**: `&a` *no* és una variable; és una constant (no ocupa cap espai de memòria, de fet, és l'adreça d'un espai de memòria). Per tant no l'hi podem assignar cap valor

Exemples detallats:
Que es pot i no es pot fer amb els operadors & i *

```
int a = 25;
```

```
int * p = &a;
```
p ⟶ 0x7ffcd82a1fa4 *p ⟶ 25
&p ⟶ valor de l'adreça de memòria que ocupa p

```
*p = 73;
```
*p ⟶ 73 a ⟶ 73
&a, p i &p queden inalterats

```
p = NULL;
```
hem "desapuntat" p

Tractament avançat de vectors: Els vectors són apuntadors

Suposem que tenim la següent declaració d'un vector:

```
double v[5] = { 1.0, 1.1, 1.2, 1.3, 1.4 };
```

0x7ffcd82a1fa4

v[0]	v[1]	v[2]	v[3]	v[4]
1.0	1.1	1.2	1.3	1.4

`v` és, a tots els efectes, un apuntador a `double` (com `double * v`) que apunta a `v[0]`.

Conseqüències

`v` $\longrightarrow$ `0x7ffcd82a1fa4`	`&v[0]` $\longrightarrow$ `0x7ffcd82a1fa4`	`v` i `&v[0]` coincideixen
`*v` $\longrightarrow$ `1.0`	`v[0]` $\longrightarrow$ `1.0`	`*v` i `v[0]` coincideixen

La declaració d'un apuntador no l'hi assigna memòria (no fa que apunti enlloc).

La "càrrega" d'un apuntador cal fer-la explícitament amb la fórmula "Declaració + Assignació" (o usant memòria dinàmica):

```
int x = 25;
int * p = &x;
```

```
double v[5] = { 1.0, 1.1, 1.2, 1.3, 1.4 };
double *vp = v;
```

Fets bàsics sobre els apuntadors — "Axiomàtica d'apuntadors"

Si `ap` és un apuntador a `modificador tipus` ja carregat:

- `*ap = ap[0]` és l'objecte apuntat,
- `*ap` i `ap[0]` tenen tipus `modificador tipus` i, per tant,
- `sizeof(*ap) = sizeof(ap[0]) = sizeof(modificador tipus)` i
- `ap` $\rightarrowtail$ `&(*ap) = &(ap[0])`.

Nota: `sizeof(*ap)` està definit encara que `ap` no estigui "carregat" (no és el cas amb `sizeof(ap[0])` ja que `ap[0]` només existeix quan `ap` apunta a alguna cosa — és a dir, quan està carregat).

Aritmètica d'apuntadors: Com funcionen els vectors

Notació: valor

En aquesta presentació, el *valor* d'un apuntador el denotarem per valor($\cdot$). Així mateix, si `ap` és un apuntador escriurem `ap` $\rightsquigarrow$ `val` per denotar que `val` = valor(`ap`) és el valor d'`ap`.

Aritmètica d'apuntadors

Si tenim un apuntador `modificador tipus * ap;` i `k` és un enter, llavors `ap + k` *és un apuntador del mateix tipus que* `ap` (és a dir és un apuntador a `modificador tipus`) que té valor (apunta a la direcció)

valor(`ap`) + `k*sizeof(modificador tipus)` = valor(`ap`) + `k*sizeof(*ap)`

(és a dir, `k*sizeof(*ap)` bytes més enllà de valor(`ap`)).

Nota

`k` es pot substituir (ad ridiculum) per qualsevol expressió que doni un enter.
Exemple: `ap + ((int) sqrt(10)) = ap + ((int) 3.1415)`.

Aritmètica d'apuntadors: Justificació de la notació

Justificació de la notació `valor` i $\rightsquigarrow$

Normalment quan una certa expressió `expr` té valor `val` escrivim `expr = val`.

En el cas de l'aritmètica d'apuntadors cal una distinció més explícita i que no usi el signe $=$ per a evitar (i clarificar!!) fórmules com

```
ap + k = ap + k*sizeof(*ap)
```

que iguala objectes diferents i, a més, no té cap sentit matemàtic.

Aritmètica d'apuntadors: Exemples

La visió dels vectors des de l'aritmètica d'apuntadors

v $\rightsquigarrow$ &v[0]	= 0x7ffcd82a1fa4	*v = v[0]
v+1 $\rightsquigarrow$ &v[1]	= 0x7ffcd82a1fa4 + sizeof(double)	*(v+1) = v[1]
v+2 $\rightsquigarrow$ &v[2]	= 0x7ffcd82a1fa4 + 2×8	*(v+2) = v[2]
v+3 $\rightsquigarrow$ &v[3]	= 0x7ffcd82a1fa4 + 3×8	*(v+3) = v[3]
v+4 $\rightsquigarrow$ &v[4]	= 0x7ffcd82a1fa4 + 4×8	*(v+4) = v[4]
v+5 $\rightsquigarrow$ &v[5]	= 0x7ffcd82a1fa4 + 5×8	*(v+5) = v[5]
v+i $\rightsquigarrow$	0x7ffcd82a1fa4 + i ×8	*(v+i) = fora de rang; resultat indeterminat ja que $i > 5$

Una nova interpretació de les declaracions de vectors

`modificador tipus nom[MIDA];`

declara una variable anomenada `nom`, de tipus

`modificador tipus [MIDA]`

Exemple: `unsigned long v[5]` declara una variable `v` de tipus `unsigned long [5]`.

$\Longrightarrow$ `modificador tipus [MIDA]` és un tipus de dades (és el tipus de dades vectors `modificador tipus` de `MIDA` components).

$\Longrightarrow$ Com que existeixen apuntadors a qualsevol tipus de dades, existeixen apuntadors a `modificador tipus [MIDA]`.

Apuntadors a vectors sencers (II)

Declaració

```
modificador tipus (* nom)[MIDA];
```

declara un apuntador anomenat `nom` a vectors sencers de tipus `modificador tipus` i de mida `MIDA`. És a dir, a dades del tipus

```
modificador tipus [MIDA]
```

Exemple

`unsigned int (*p)[17]` declara un apuntador `p` a vectors `unsigned int` de `17` components.

Notes:

1. Com que un vector és un apuntador; un apuntador a vectors sencers és un apuntador a un apuntador i, per tant, és un apuntador doble.
2. Com que els apuntadors també són tipus de dades, `modificador tipus (*)[MIDA]` és un nou tipus de dada.

Apuntadors a vectors sencers (III)

Recordem que la simple declaració d'un apuntador no l'hi assigna memòria (no fa que apunti enlloc). La "càrrega" d'un apuntador cal fer-la explícitament amb la fórmula "Declaració + Assignació" o usant memòria dinàmica.

Declaració + Assignació

```
int V[5]={0,1,2,3,4};

int (*VP)[5] =
      (int (*)[5]) V;
```

Nota sobre el forçament de tipus `(int (*)[5])`

És necessari perquè `V` és un apuntador a `int` que cal convertir en un apuntador a `int [5]` com `VP`.

Declaració + Memòria Dinàmica

```
int (*VP)[5];
VP = (int (*)[5]) malloc( sizeof(*VP) );
if(VP == NULL){
   fprintf (stderr,
        "\nERROR en ''carregar'' VP ...\n\n"
   ); return 1; }
```

Observació

`sizeof(*VP)` funciona malgrat que l'apuntador `VP` encara no està inicialitzat.

Nota sobre el forçament de tipus `(int (*)[5])`

És necessari perquè `malloc` torna un apuntador `void`.

Apuntadors a vectors sencers. Funcionament

Suposem que tenim un apuntador a vectors sencers

```
modificador tipus (* VP)[MIDA]
```

que ja està "carregat".

Fets bàsics sobre els apuntadors a vectors sencers

Compareu amb la pàgina 58

- L'objecte apuntat per `VP` (que s'anomena `*VP = VP[0]`) és un apuntador té tipus `modificador tipus [MIDA]` i, per tant, és un vector de tipus `modificador tipus` i que té `MIDA` components:

```
*VP = VP[0] = {(*VP)[0] (*VP)[1]     ··· (*VP)[MIDA-1]}
            = {*(*VP)   *((*VP) + 1)··· *((*VP) + MIDA-1)}
            = {VP[0][0] VP[0][1]     ··· VP[0][MIDA-1]}
            = {*(VP[0]) *(VP[0] + 1)··· *(VP[0] + MIDA-1)}.
```

- `sizeof(*VP) = MIDA * sizeof((*VP)[0])`
  `= MIDA * sizeof(*(*VP)) =`
  `sizeof(VP[0]) = MIDA * sizeof(VP[0][0])`
  `= MIDA * sizeof(modificador tipus)`.
- `VP` $\rightarrowtail$ `&(*VP) = &((*VP)[0]) = &(*(*VP))`
  `= &(VP[0]) = &(VP[0][0]) = &(*(VP[0]))`.

Aquesta expressió suggereix que `VP` és, de fet, una matriu d'una fila.

Apuntadors a vectors sencers. Funcionament (II)

"La prueba del algodón"

Exemple

```
float V[5] = {0.0,1.0,2.0,3.0,4.0};
float (*VP)[5] = (float (*)[5]) V;
```

i	V[i]	(*VP)[i]	VP[0][i]	*((*VP) + i)	*(VP[0] + i)
0	0.0	0.0	0.0	0.0	0.0
1	1.0	1.0	1.0	1.0	1.0
2	2.0	2.0	2.0	2.0	2.0
3	3.0	3.0	3.0	3.0	3.0
4	4.0	4.0	4.0	4.0	4.0

La taula de l'esquerra mostra el vector `V` escrit de 5 maneres diferents.

La primera (columna `V[i]` — per comprovació) mostra el vector imprès de manera completament estàndard.

Les darreres 4 columnes mostren el vector `V` a través del seu apuntador sinònim (a vectors sencers) `VP`. Les 4 maneres són les descrites a la pàgina anterior. En les dues primeres s'escriu el vector `*VP` en components i en les dues darreres es fa servir l'aritmètica d'apuntadors.

Tractament avançat de matrius: Les matrius són apuntadors a vectors sencers

Una matriu

```
tipus Mat[RowNum][ColNum];
```

es pot visualitzar com un vector de `RowNum` vectors fila, cada un d'ells, de tipus `tipus [ColNum]`.
Equivalentment, com un apuntador a vectors fila sencers de `ColNum` components com

```
tipus (*) [ColNum]
```

que ja està carregat amb `RowNum` d'aquests vectors fila sencers.

Exemple

```
int m[2][4];
```

La matriu `m` es pot visualitzar com un vector format per `2` vectors fila de tipus `int [4]`.
Equivalentment, com un apuntador de tipus `int (*) [4]` a vectors fila de tipus `int [4]` que ja està carregat amb `2` d'aquests vectors fila.

Tractament avançat de matrius: Les matrius són apuntadors a vectors sencers (II)

Per definir una matriu a partir d'un apuntador a vectors sencers cal declarar-lo i carregar-lo explícitament amb la fórmula "Declaració + Assignació" o usant memòria dinàmica.

Declaració + Assignació

```
int m[2][4] = { { 00, 01, 02, 03 },
                { 10, 11, 12, 13 } };
int (*mp)[4] =  m;
```

Declaració + Memòria Dinàmica

```
int (*mp)[4];
if((mp = (int (*)[4]) malloc(2 * sizeof(*mp))) == NULL) {
        fprintf (stderr, "\nERROR en ''carregar'' mp ...\n\n");
        return 1;
}
for(i=0; i < 4; i++){for(j=0; j < 4; j++){
        mp[i][j] = i*10 + j;
}}
```

Tractament avançat de matrius: Aritmètica d'apuntadors a vectors sencers

Una matriu com apuntador a vectors sencers

Sigui `Mat` una matriu declarada com un apuntador a vectors sencers:

`modificador tipus (* Mat)[ColNum]`

i "carregada" (veure els exemples de la pàgina anterior).

Apuntadors a files senceres d'una matriu

Les expressions `Mat + i` definides per tot $i \geq 0$ són apuntadors a

`modificador tipus [ColNum]`

ja que l'apuntador original `Mat` ho és. Per tant, `Mat[i] = *(Mat + i)` (que són els objectes apuntats per `Mat + i`) són els vectors (sencers)

`{Mat[i][0] Mat[i][1] ··· Mat[i][ColNum-1]}`

que tenen tipus `modificador tipus [ColNum]`. Per tant,

$$\begin{aligned} \texttt{Mat + i} \leadsto \texttt{\&(Mat[i])} &= \texttt{\&(Mat[i][0])} \\ &= \text{valor}(\texttt{Mat}) + \texttt{i * sizeof(Mat[i])} \\ &= \text{valor}(\texttt{Mat}) + \texttt{i * sizeof(*Mat)} \\ &= \text{valor}(\texttt{Mat}) + \texttt{i * ColNum * sizeof(Mat[0][0])} \\ &= \text{valor}(\texttt{Mat}) + \texttt{i * ColNum * sizeof(modificador tipus)}. \end{aligned}$$

De fet, són el vector (sencer) fila `i` de la matriu `Mat` (suposant que hagi estat carregada, al menys, amb $i + 1$ files).

Veure la pàgina 65

Tractament avançat de matrius: Aritmètica d'apuntadors a vectors sencers (II)

Aritmètica d'apuntadors: els vectors fila com apuntadors

Com hem vist, els objectes `Mat[i] = *(Mat + i)` són vectors (sencers) de tipus `modificador tipus [ColNum]`. Per tant, en ser vectors, són apuntadors a `modificador tipus` que apunten a `Mat[i][0]`:

```
Mat[i] = *(Mat + i) ↬ &(Mat[i][0]), i
*(Mat[i]) = *(*(Mat + i)) = Mat[i][0].
```

Per altra banda, usant l'aritmètica estàndard d'apuntadors a vectors, obtenim diverses expressions per l'element a la columna `j` de la fila `i` de `Mat`:

```
Mat[i][j] = *(Mat + i)[j]
          = *(Mat[i] + j) = *( *(Mat + i) + j)
```

Per un exemple d'ús d'aquesta aritmètica d'apuntadors veure la pàgina ▸ 124 on s'aplica a emmagatzemar un fitxer amb un nombre indeterminat de línies.

Índex

Funcions que tornen més d'un valor: trencant la immodificabilitat dels paràmetres

Objectiu — Motivació

En aquest apartat volem aprendre a definir i usar paràmetres de funcions que permetin modificar les seves variables associades amb la finalitat de tenir funcions que puguin retornar més d'un valor.

Més precisament: volem crear funcions que, mitjançant la modificació (internament — dins el cos de la funció) de variables associades als seus paràmetres, puguin traspassar aquestes modificacions a variables del mòdul que crida la funció (òbviament excloent l'ús del *valor de retorn de la funció*).

Com hem dit, això passa per l'ús d'*apuntadors com a paràmetres*.

Funcions que tornen més d'un valor: trencant la immodificabilitat dels paràmetres (II)

La màgia dels apuntadors: trencant la immodificabilitat dels paràmetres

Volem modificar una variable anomenada `x` mitjançant una funció.

Part 1: En comptes de passar com a paràmetre la variable `x` a la funció, l'hi passem un apuntador `p = &x` que apunti a `x` (així, `p` és el paràmetre *immodificable* que passem per valor).

Part 2: Dins de la funció modifiquem `*p` (no el paràmetre `p`!!). Això fa el que volem ja que `*p = x` és, de fet, la variable que volem modificar.

Nota: Això és *legal* ja que no alterem `p`.
El que modifiquem és `*p`, que està permès.

Nota: Aquest "truc" ja l'hem usat amb la funció `scanf`:

```
scanf ("%lf", &r);
```

Exemple: donats dos nombres x i y volem calcular $x + y$, $x - y$, $x * y$ i x/y usant una única funció

```
// Programa calculs.c
#include <stdio.h>

// Prototipus
void calculs( double, double, double *, double *, double *, double *);

void main ()
{
  double x,y;
  double suma, resta, p, q;

  printf("Entra x i y separats per una coma: "); scanf("%lf,%lf", &x,&y);

  calculs(x, y, &suma, &resta, &p, &q);

  printf("La suma és: %f\n", suma);
  printf("La resta és: %f\n", resta);
  printf("El producte és: %f\n", p);
  printf("El quocient és: %f\n", q);
}

void calculs( double x, double y, double *s, double *r, double *p, double *q)
{
    *s = x+y;
    *r = x-y;
    *p = x*y; // No confongueu els dos *. Són diferents!!
    *q = x/y; // Error. Pot donar problemes si y és zero.
} /* Fi de calculs */
```

Funcions que tornen més d'un valor: Usant vectors

Vectors com paràmetres

Els vectors, en ser apuntadors, es passen així com a paràmetres de funcions:

`float vect[]` o `tipus *nom`

L'apuntador (com a paràmetre) *no és modificable* però els elements del vector (als que s'accedeix per indirecció usant l'aritmètica d'apuntadors) si que ho són.

Nota: Com que un vector (apuntador) no sap la seva dimensió hi poden haver problemes greus d'accessos fora de rang (és a dir fora de la memòria vàlida del vector). Veure la pàgina 61.

Exemple: el programa `vectorsf.c`

```
#include <stdio.h>
#define DimVec 2

void imprimeixvector (float [], int);

int main() { int i;
  float v[DimVec];

  puts("Entrem el vector");
  for (i=0; i < DimVec ; i++){
        printf("v(%d) = ? ",i+1);
        scanf("%f",&(v[i]));
  }

  puts("Ara l'imprimim:");
  imprimeixvector (v, DimVec);
  return 0;
}

void imprimeixvector (float vect[], int dim)
{ int i;
  for(i=0; i < dim ; i++) printf("%f ",vect[i]);
  printf("\n");
  return;
 }
```

Funcions que tornen més d'un valor: Usant matrius

Matrius com paràmetres

Una matriu de `ColNum` columnes és un apuntador a vectors fila sencers de `ColNum` components (veure la pàgina 67). Per tant, en passar-la com a paràmetre a una funció cal fer-ho així (és a dir, com apuntador a vectors fila sencers de `ColNum` components):

`float mat[][DiMat]`, o
`tipus *nom[ColNum]`

L'apuntador (com a paràmetre) *no és modificable* però els elements de la matriu (als que s'accedeix per indirecció usant l'aritmètica doble d'apuntadors) si que ho són.

Nota: Com en el cas dels vectors hi poden haver problemes greus d'accessos fora de rang.

Exemple: el programa `matrius.c`

```
#include <stdio.h>
#define DiMat 20
void imprimeixmatriu (float [][DiMat], int);

int main() { int i,j, dim; float a[DiMat][DiMat];
  printf("Entra la dimensió <= %d\n",DiMat);
  scanf("%d", &dim);
  if(dim > DiMat) {
    puts("Error: dimensió massa gran"); return 1;
  }
  puts("Entrem la matriu");
  for(i=0 ; i<dim ; i++){ for(j=0 ; j<dim ; j++){
        printf("a(%d,%d)=?",i+1,j+1);
        scanf("%f",&(a[i][j]));
  }}
  puts("Ara imprimim la matriu:");
  imprimeixmatriu (a, dim);
  return 0;
} /* Fi del main */

void imprimeixmatriu (float mat[][DiMat], int dim)
{ int i, j;
  for(i=0 ; i<dim ; i++){ for(j=0 ; j<dim ; j++){
        printf("%f ",mat[i][j]);
  } ; printf("\n"); }
  return;
}
```

Aplicacions de l'aritmètica d'apuntadors als vectors

Usant l'aritmètica d'apuntadors

podem fer més eficients els processos de manipulació dels vectors ja que, de fet, són apuntadors.

Exemple: la funció `imprimeixvector`

sense aritmètica d'apuntadors

```
void imprimeixvector (float vect[], int dim)
{ int i;

  for(i=0 ; i < dim ; i++)
           printf("%f ",vect[i]);
  printf("\n");

  return;
 }
```

amb aritmètica d'apuntadors

```
void imprimeixvector (float *vect, int dim)
{ int i;

  for(i=0 ; i < dim ; i++, vect++)
                printf("%f ", *vect );
  printf("\n");

  return;
 }
```

Nota

Com a exemple addicional veure els codis de les funcions de *cadenes de caràcters* `str***`, començant per la funció `strcpy` (pàgina ▸ 90).

Les matrius com apuntadors a vectors: Forçant el tipus

Les matrius tenen un estructura lineal en memòria

```
int m[2][4] = { { 00, 01, 02, 03 },
                { 10, 11, 12, 13 } };
```

0x7ffcd82a1fa4

m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[1][0]	m[1][1]	m[1][2]	m[1][3]
00	01	02	03	10	11	12	13

L'estructura lineal en memòria de les matrius permet tractar-les com vectors forçant el seu tipus:

```
(tipus-de-dada *) apuntador-matriu
```

Exemple: la matriu `m` com vector

```
int *mv = (int *) m;
```

Nota sobre el forçament de tipus `(int *)`

És necessari perquè `m`, en ser una matriu de 4 columnes, és un apuntador de tipus `int (*)[4]` que cal convertir en un apuntador a `int` com `mv`.

Ara `mv` és també la matriu `m` però organitzada linealment (com un vector) seguint l'ordre per files (veure la figura anterior). De fet, `mv` apunta al mateix lloc que `m` però "veu" aquesta zona de memòria com un vector `int` de 8 components.

La matriu `m` com a vector
Una visió a través de l'aritmètica d'apuntadors

mv	$\rightsquigarrow$	&m[0][0] = 0x7ffcd82a1fa4	*mv	= mv[0] = m[0][0]
mv+1	$\rightsquigarrow$	&m[0][1] = 0x7ffcd82a1fa4 + sizeof(int)	*(mv+1)	= mv[1] = m[0][1]
mv+2	$\rightsquigarrow$	&m[0][2] = 0x7ffcd82a1fa4 + 2×4	*(mv+2)	= mv[2] = m[0][2]
mv+3	$\rightsquigarrow$	&m[0][3] = 0x7ffcd82a1fa4 + 3×4	*(mv+3)	= mv[3] = m[0][3]
mv+4	$\rightsquigarrow$	&m[1][0] = 0x7ffcd82a1fa4 + 4×4	*(mv+4)	= mv[4] = m[1][0]
mv+5	$\rightsquigarrow$	&m[1][1] = 0x7ffcd82a1fa4 + 5×4	*(mv+5)	= mv[5] = m[1][1]
mv+6	$\rightsquigarrow$	&m[1][2] = 0x7ffcd82a1fa4 + 6×4	*(mv+6)	= mv[6] = m[1][2]
mv+7	$\rightsquigarrow$	&m[1][3] = 0x7ffcd82a1fa4 + 7×4	*(mv+7)	= mv[7] = m[1][3]
mv+i	$\rightsquigarrow$	0x7ffcd82a1fa4 + i ×4	*(mv+i)	=

Fora de rang; resultat indeterminat (ja que $i > 7$)

Exemple: les matrius com apuntadors a vectors

Imprimint una matriu accedint-hi com a vector: La funció `printMat4Vect`

Canvis al programa `matrius.c` per a poder cridar la funció `printMat4Vect` correctament

`matrius.c`: veure la pàgina 44 o, equivalentment, la pàgina 76

Essencialment, cal forçar el tipus de la matriu en cridar la funció.

```
void imprimeixmatriu (float [][DiMat], int);
imprimeixmatriu (a, dim);
```

⇒

```
printMat4Vect (float *, int, int);
printMat4Vect ((float *) a, dim, DiMat);
```

La funció `printMat4Vect` bàsica sense aritmètica d'apuntadors

```
void printMat4Vect ( float *mat, int dim,
                     int Col_Num_Declar )
{ int i, j;

  for(i=0 ; i < dim ; i++){
    for(j=0 ; j < dim; j++)
      printf("%f ", mat[i*Col_Num_Declar + j]);
    printf("\n");
  }

  return;
}
```

La funció `printMat4Vect` eficient amb aritmètica d'apuntadors

```
void printMat4Vect ( float *mat, int dim,
                     int Col_Num_Declar )
{ int i, j;

  for(i=0 ; i < dim ; i++, mat += Col_Num_Declar){
    for(j=0 ; j < dim ; j++)
      printf("%f ", *(mat + j));
    printf("\n");
  }

  return;
}
```

Exemple: les matrius com apuntadors a vectors

La funció `printMat4Vect` senzilla per matrius plenes no quadrades

```
void printMat4Vect (float *mat, int dim_files, int dim_cols)
{  register int i;

   dim_files *= dim_cols;

   for(i=0; i < dim_files ; i++, mat++){
     if( !(i % dim_cols) ) printf("\n");
     printf ( "%8.4f", *mat );
   } ; printf("\n\n");

   return;
}
```

`dim_files` ara és la dimensió de la matriu com a vector.
Pas per valor: `dim_files` no es modifica al mòdul que crida la funció. La modificació és fa solament en l'àmbit de la funció

Índex

Un tipus especial de vectors: Les cadenes de caràcters són vectors `char`

Definició de cadena de caràcters

Les cadenes de caràcters són vectors de tipus `char` *amb l'enter 0* al final com a marca de final de cadena.

Nota

Una cadena de caràcters *sempre* usa una posició més de les que necessita per a contenir la cadena que l'hi està destinada (el 0 com a marca de final de cadena).

Exemple: `"Hola"` necessita un vector `char` de 5 o més posicions per a poder ser escrita correctament en una cadena de caràcters.

Les variables char: Un gran desconegut

Les variables del (mal anomenat tipus) `char` són variables enteres d'un byte (8 bits) de llargada. Per tant poden contenir enters en el rang -128 — 127 en la versió `signed`, i 0 — 255 en la versió `unsigned`.

El contingut de les variables `char` es pot imprimir en qualsevol format compatible (per exemple `%d` o `%u`) però estan pensades per a ser impreses com *caràcters* amb el format `%c` (o `%s` si és una cadena (vector) de caràcters). En fer això es produeix una *traducció* de l'enter contingut a la variable `char` al seu equivalent com a caràcter segons la taula `ASCII`.

Comandes:

```
man ascii
man iso_8859-1 o bé man latin1
```

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (such as ISO 8859-1, the Linux default character set) contain ASCII as their lower half.The international counterpart of ASCII is known as ISO 646.
The ASCII standard was published by the United States of America Standards Institute (USASI) in 1968.

La codificació dels char: la taula ASCII

ascii - the ASCII character set encoded in octal, decimal, and hexadecimal

ASCII control characters

Dec	Hex	Oct	Char	Meaning & C esc. seq.
0	00	000	NUL	'\0' (null character)
1	01	001	SOH	(start of heading)
2	02	002	STX	(start of text)
3	03	003	ETX	(end of text)
4	04	004	EOT	(end of transmission)
5	05	005	ENQ	(enquiry)
6	06	006	ACK	(acknowledge)
7	07	007	BEL	'\a' (bell)
8	08	010	BS	'\b' (backspace)
9	09	011	HT	'\t' (horizontal tab)
10	0A	012	LF	'\n' (new line)
11	0B	013	VT	'\v' (vertical tab)
12	0C	014	FF	'\f' (form feed)
13	0D	015	CR	'\r' (carriage ret)
14	0E	016	SO	(shift out)
15	0F	017	SI	(shift in)
16	10	020	DLE	(data link escape)
17	11	021	DC1	(device control 1)
18	12	022	DC2	(device control 2)
19	13	023	DC3	(device control 3)
20	14	024	DC4	(device control 4)
21	15	025	NAK	(negative ack.)
22	16	026	SYN	(synchronous idle)
23	17	027	ETB	(end of trans. blk)
24	18	030	CAN	(cancel)
25	19	031	EM	(end of medium)
26	1A	032	SUB	(substitute)
27	1B	033	ESC	(escape)
28	1C	034	FS	(file separator)
29	1D	035	GS	(group separator)
30	1E	036	RS	(record separator)
31	1F	037	US	(unit separator)
127	7F	177	DEL	

ASCII printable characters

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
32	20	040	SPACE	64	40	100	@	96	60	140	‘
33	21	041	!	65	41	101	A	97	61	141	a
34	22	042	"	66	42	102	B	98	62	142	b
35	23	043	#	67	43	103	C	99	63	143	c
36	24	044	$	68	44	104	D	100	64	144	d
37	25	045	%	69	45	105	E	101	65	145	e
38	26	046	&	70	46	106	F	102	66	146	f
39	27	047	’	71	47	107	G	103	67	147	g
40	28	050	(	72	48	110	H	104	68	150	h
41	29	051	)	73	49	111	I	105	69	151	i
42	2A	052	*	74	4A	112	J	106	6A	152	j
43	2B	053	+	75	4B	113	K	107	6B	153	k
44	2C	054	,	76	4C	114	L	108	6C	154	l
45	2D	055	-	77	4D	115	M	109	6D	155	m
46	2E	056	.	78	4E	116	N	110	6E	156	n
47	2F	057	/	79	4F	117	O	111	6F	157	o
48	30	060	0	80	50	120	P	112	70	160	p
49	31	061	1	81	51	121	Q	113	71	161	q
50	32	062	2	82	52	122	R	114	72	162	r
51	33	063	3	83	53	123	S	115	73	163	s
52	34	064	4	84	54	124	T	116	74	164	t
53	35	065	5	85	55	125	U	117	75	165	u
54	36	066	6	86	56	126	V	118	76	166	v
55	37	067	7	87	57	127	W	119	77	167	w
56	38	070	8	88	58	130	X	120	78	170	x
57	39	071	9	89	59	131	Y	121	79	171	y
58	3A	072	:	90	5A	132	Z	122	7A	172	z
59	3B	073	;	91	5B	133	[	123	7B	173	{
60	3C	074	<	92	5C	134	\ '\\'	124	7C	174	|
61	3D	075	=	93	5D	135	]	125	7D	175	}
62	3E	076	>	94	5E	136	^	126	7E	176	~
63	3F	077	?	95	5F	137	_				

Latin-1 Extended ASCII characters

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
160	A0	240	NBSP	192	C0	300	À	224	E0	340	à
161	A1	241	¡	193	C1	301	Á	225	E1	341	á
162	A2	242	¢	194	C2	302	Â	226	E2	342	â
163	A3	243	£	195	C3	303	Ã	227	E3	343	ã
164	A4	244	¤	196	C4	304	Ä	228	E4	344	ä
165	A5	245	¥	197	C5	305	Å	229	E5	345	å
166	A6	246	¦	198	C6	306	Æ	230	E6	346	æ
167	A7	247	§	199	C7	307	Ç	231	E7	347	ç
168	A8	250	¨	200	C8	310	È	232	E8	350	è
169	A9	251	©	201	C9	311	É	233	E9	351	é
170	AA	252	ª	202	CA	312	Ê	234	EA	352	ê
171	AB	253	«	203	CB	313	Ë	235	EB	353	ë
172	AC	254	¬	204	CC	314	Ì	236	EC	354	ì
173	AD	255	SHYP	205	CD	315	Í	237	ED	355	í
174	AE	256	®	206	CE	316	Î	238	EE	356	î
175	AF	257	¯	207	CF	317	Ï	239	EF	357	ï
176	B0	260	°	208	D0	320	Ð	240	F0	360	ð
177	B1	261	±	209	D1	321	Ñ	241	F1	361	ñ
178	B2	262	²	210	D2	322	Ò	242	F2	362	ò
179	B3	263	³	211	D3	323	Ó	243	F3	363	ó
180	B4	264	´	212	D4	324	Ô	244	F4	364	ô
181	B5	265	µ	213	D5	325	Õ	245	F5	365	õ
182	B6	266	¶	214	D6	326	Ö	246	F6	366	ö
183	B7	267	·	215	D7	327	×	247	F7	367	÷
184	B8	270	¸	216	D8	330	Ø	248	F8	370	ø
185	B9	271	¹	217	D9	331	Ù	249	F9	371	ù
186	BA	272	º	218	DA	332	Ú	250	FA	372	ú
187	BB	273	»	219	DB	333	Û	251	FB	373	û
188	BC	274	¼	220	DC	334	Ü	252	FC	374	ü
189	BD	275	½	221	DD	335	Ý	253	FD	375	ý
190	BE	276	¾	222	DE	336	Þ	254	FE	376	þ
191	BF	277	¿	223	DF	337	ß	255	FF	377	ÿ

Comentaris sobre els codis ASCII i la conversió a caràcters

Observació

El caràcter `'0'` té codi ASCII `48` (en decimal) mentre que no hi ha cap caràcter que tingui el `0` com a codi ASCII. Això fa natural l'elecció de l'enter `0` com a marca de final de cadena.

Exemple:
Un codi senzill que escriu els caràcters imprimibles de la taula ASCII

```
char codi;
for(codi=33 ; codi < 127 ; codi++) printf("%3d <--> %c\n", codi, codi);
```

Inicialització i declaració de cadenes

Exemples

```
char cadena[21] = { 'H', 'o', 'l', 'a', 0 };
char cadena[21] = "Hola";            /* o bé, més còmodament ... */
char missatge[] = "Hola a tothom";
```

Nota

Les cadenes inicialitzades amb un text entre cometes (per exemple `char cadena[21] = "Hola";`) ja porten implícit l'enter `0` d'acabament i no cal incloure'l explícitament entre les cometes.

Funcions estàndard per a cadenes de caràcters

En **C** no es poden fer assignacions, comparacions ni cap altra operació directa amb cadenes *completes* ja que són vectors. Solament es poden fer caràcter a caràcter.

És per això que totes les operacions de manipulació de cadenes són complexes i, per tant, s'han d'implementar com a funcions de llibreria.

Els prototipus de les funcions de cadenes de caràcters es carreguen usant el fitxer de capçalera:

Els prototipus d'algunes funcions `string`

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

Algunes funcions `string` i el seu ús

Un resum d'algunes funcions `string`

```
int   strlen  ( char *string );
char *strcpy  ( char *destination, char *source );
char *strncpy ( char *destination, char *source, int num-de-chars );
char *strcat  ( char *destination, char *source );
char *strncat ( char *destination, char *source, int num-de-chars );
char *strdup  ( char *source );
int  *strcmp  ( char *string_1, char *string_2 );
int  *strncmp ( char *string_1, char *string_2, int num-de-chars );
char *strchr  ( char *string, char character );
char *strrchr ( char *string, char character );
```

`strlen`

La longitud efectiva d'una cadena de caràcters `kdna` en un moment determinat es pot obtenir a través de la funció `strlen(kdna)`.

Exemple: Amb la declaració `char cadena[21] = "Hola";` tenim `strlen(cadena)` $\longrightarrow$ `4`.

Algunes funcions `string` i el seu ús (II)

`strcpy` — `strncpy`

Per a copiar el contingut de la cadena de caràcters `kdna` a `kdna9` es pot usar la funció `strcpy(kdna9, kdna)`.

Nota

La funció `strncpy` fa el mateix que la seva germana `strcpy` limitant el nombre de caràcters a copiar a `num-de-chars`. Això és útil, per exemple, quan la cadena font és més llarga que la capacitat de la cadena destinació: la instrucció
`strncpy(kdna9, kdna, num-de-chars)` amb `num-de-chars < LONGKDNA9`
còpia parcialment la cadena font limitant el nombre de caràcters copiats (recordem que cal reservar el darrer caràcter de la còpia (o de `kdna9`) per situar-hi el `0` de marca de final de cadena).

Per entendre-ho millor: una possible implementació de la funció `strcpy`
Per eficiència usem el fet que els vectors són apuntadors i l'aritmètica d'apuntadors

```
void strcpy(char *cd, const char *co)
{
   while(*co) { *cd = *co; cd++; co++; }
   *cd = 0;
}
```

▸ Tornar a la pàgina 77

Algunes funcions string i el seu ús (III)

strcmp — strncmp

La comparació de cadenes es fa caràcter a caràcter, començant pel primer caràcter de les dues cadenes a comparar i passant als següents mentre la diferència dels codis ASCII corresponents sigui 0. La funció strcmp retorna el valor de l'última diferència (la dels primers caràcters que difereixen). És a dir: negatiu si la segona cadena és alfabèticament més gran que la primera; positiu en el cas oposat i 0 si són iguals.

Nota: Si una cadena és una subcadena de l'altra és automàticament més petita.

Per entendre-ho millor:
una possible implementació de la funció strcmp

```
int strcmp( char *cad1, char *cad2 )
{
  while( ( *cad1 && *cad2 ) && (*cad1 == *cad2) ) {
     cad1++; cad2++;
  }
  return *cad1 - *cad2;
} /* strcmp */
```

La funció strncmp fa el mateix que la seva germana strcmp limitant el nombre de caràcters a comprovar a num-de-chars.

strchr — strrchr

Aquestes funcions busquen un caràcter dins d'una cadena. Si el troben retornen la subcadena de caràcters que comença al caràcter buscat dins la cadena. En cas contrari retornen NULL.
La primera funció inspecciona la cadena de principi a final i la segona ho fa de final a principi.

Per entendre-ho millor: una possible implementació de la funció strchr

```
char * strchr( char *cadena, char caracter )
{
  while( *cadena && *cadena != caracter ) cadena++;
  if (*cadena == 0) return NULL;
  return cadena;
} /* strchr */
```

Ordenació en C: la funció qsort
Aprofitant que sabem apuntadors i usar la funció strcmp

Índex

Manual de la funció qsort

La funció qsort té capçalera

```
void qsort (void * base, size_t num, size_t size,
            int (*compara)(const void *, const void *))
```

Observem que el quart paràmetre és una funció (compara) que, per definició, retorna un int i té exactament dos paràmetres que són apuntadors (a void).

La funció qsort que fa?

Ordena num blocs (anomenats *tokens*) del vector base, cada un de mida size bytes, de més petit a més gran fent servir l'algorisme *quicksort*.

Tots els algorismes d'ordenació es basen en *comparar* certs parells de tokens i, si no estan en l'ordre correcte, *permutar*-los.
Quins tokens es comparen i en quin ordre depèn de l'algorisme i en condiciona la seva eficiència.

La funció qsort no coneix la naturalesa ni el contingut dels tokens i, per tant, no els pot comparar. *La comparació de tokens es descentralitza a la funció* compara, que funciona com la strcmp quant al valor de retorn.

Nota: Per permutar dos tokens no cal conèixer el seu contingut. El més eficient és fer-ho byte per byte (sempre que coneixem la seva mida).

Com hem dit, la funció qsort ha de descentralitzar la comparació de tokens degut a que no coneix ni la seva naturalesa ni el seu contingut.

La funció compara vista des de qsort

Quan qsort vol comparar dos tokens A i B ho fa cridant:

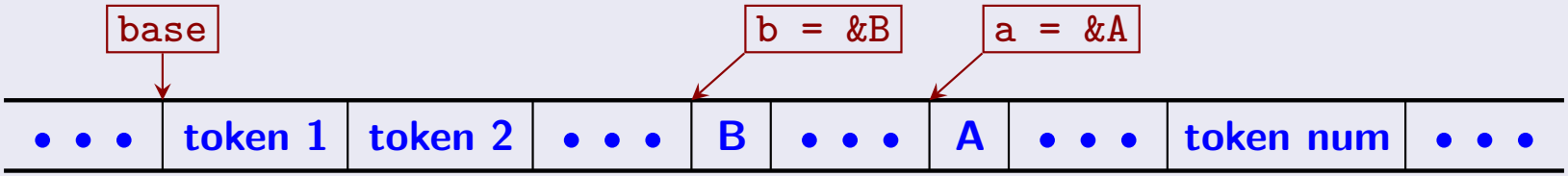
compara(&A, &B) que *ha de retornar* un valor $\begin{cases} \text{negatiu} & \text{si } A > B, \\ 0 & \text{si } A = B, \text{ i} \\ \text{positiu} & \text{si } A > B; \end{cases}$

on els símbols $<$ i $>$ s'interpreten en funció del tipus dels tokens.
És a dir si els tokens són nombres, els símbols $<$ i $>$ usualment adopten el sentit estàndard; si són *strings* llavors $<$ i $>$ denoten "més petit" i "més gran" en l'ordre lexicogràfic (el del diccionari); ....

La funció qsort deixa al programador la feina de codificar la funció compara de manera que compleixi els requeriments especificats, ja que és el programador qui sap de qui tipus són els tokens, com accedir a la informació que contenen, i com comparar-los.

La funció qsort (i, per extensió, compara) veu el vector base com

un desplegament de num *tokens* consecutius a la memòria, cada un d'ells de tipus void i mida size bytes. No coneix el seu tipus ni contingut.



Les interioritats de la funció compara — la sala de màquines

La funció compara rep apuntadors void a i b als tokens A i B, respectivament.

La primera cosa que hem de fer per a poder usar els tokens A i B, accedir a les seves dades i fer la comparació és convertir els dos apuntadors void a apuntadors "legals" als tokens. Això és fa amb un *forçament de tipus*. Si el vector base ha estat declarat de tipus tipus (per exemple, int, float, ...),

`(tipus *) a` i `(tipus *) b`

converteixen a i b en apuntadors a tipus.

Finalment, `*((tipus *) a)` i `*((tipus *) b)`

es refereix al *contingut* dels tokens A i B que és el que (o ens permet accedir al que) hem de comparar.

Ordenació en C: L'exemple més senzill possible

Ordenació d'un vector de dades numèriques de tipus int

```
#include <stdio.h>
#include <stdlib.h> //

#define NumDades 20

int compara_ints(const void *a, const void *b) {
                    return ( *((int *) a) - *((int *) b) );
}

int main() {
    register int i;
    int dades[NumDades] = { 111, 58, 67, 84, 195, 96, 168, 45,
                            48, 53, 140, 180, 136, 190, 79,
                            132, 19, 32, 178, 110  };

    qsort(dades, NumDades, sizeof(int), compara_ints);
    for(i=0; i < NumDades; i++)
        printf("dades[%2.2d] = %d\n",i, dades[i]);
    return 0;
}
```

`*((int *) a)` i `*((int *) b)` accedeixen al *contingut* dels tokens apuntats per `a` i `b` (veure la pàgina anterior). Són els dos `int`'s que cal comparar.

Resultat:

```
dades[00] = 19
dades[01] = 32
dades[02] = 45
dades[03] = 48
dades[04] = 53
dades[05] = 58
dades[06] = 67
dades[07] = 79
dades[08] = 84
dades[09] = 96
dades[10] = 110
dades[11] = 111
dades[12] = 132
dades[13] = 136
dades[14] = 140
dades[15] = 168
dades[16] = 178
dades[17] = 180
dades[18] = 190
dades[19] = 195
```

`*((int *) a) - *((int *) b)`

és la manera més fàcil de comparar dos nombres amb la finalitat que el resultat compleixi les especificacions de la pàgina 95.

Ordenació en C: Un exemple senzill

Ordenació d'un vector de dades numèriques en punt flotant

```
#include <stdio.h>
#include <stdlib.h> // Per qsort
#define TOL 1.e-7
#define NumDades 30
int compara_floats(const void *a, const void *b) {
    float diff = *((float *) a) - *((float *) b);
    return ((diff < -TOL) ? -1 : ((diff > TOL) ? 1 : 0));
}

int main() { register int i;
    float dades[NumDades] = {
        0.1798854, 0.1176260, 0.3074593, 0.7795518, 0.6183642,
        0.0738382, 0.3337906, 0.1887642, 0.4191682, 0.3355493,
        0.1751513, 0.8525568, 0.6881627, 0.3511742, 0.3422691,
        0.5398925, 0.2191889, 0.5840292, 0.6594841, 0.8408323,
        0.0004115, 0.4917996, 0.6445988, 0.1403347, 0.2485667,
        0.9294325, 0.1099306, 0.3420534, 0.4186507, 0.6168580 };

    qsort(dades, NumDades, sizeof(float), compara_floats);
    for(i=0; i < NumDades; i++)
        printf("dades[%2.2d] = %g\n",i, dades[i]);
    return 0;
}
```

En punt flotant les comparacions d'igualtat no funcionen

degut al errors d'arrodoniment: la comparació d'igualtat s'ha de realitzar com `fabs(diff) < TOL`.
La funció `compara_floats` implementa aquesta comparació, juntament amb les de negativitat i positivitat alhora, amb l'ajut de dos *if's aritmètics*.

Resultat:

```
dades[00] = 0.0004115
dades[01] = 0.0738382
dades[02] = 0.109931
dades[03] = 0.117626
dades[04] = 0.140335
dades[05] = 0.175151
dades[06] = 0.179885
dades[07] = 0.188764
dades[08] = 0.219189
dades[09] = 0.248567
dades[10] = 0.307459
dades[11] = 0.333791
dades[12] = 0.335549
dades[13] = 0.342053
dades[14] = 0.342269
dades[15] = 0.351174
dades[16] = 0.418651
dades[17] = 0.419168
dades[18] = 0.4918
dades[19] = 0.539892
dades[20] = 0.584029
dades[21] = 0.616858
dades[22] = 0.618364
dades[23] = 0.644599
dades[24] = 0.659484
dades[25] = 0.688163
dades[26] = 0.779552
dades[27] = 0.840832
dades[28] = 0.852557
dades[29] = 0.929433
```

Ordenació en C: Un exemple amb més estructura

Ordenació de les files d'una matriu de dades numèriques respecte de la tercera columna

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

#define TOL 1.e-7
#define NumDades 6
#define NumCamps 5

void imprimeixmatriu (float *, int, int);
int compara_files(const void *a, const void *b) {
    float diff = ((float *) a)[2] - ((float *) b)[2];
    return ((diff < -TOL) ? -1 : ((diff > TOL) ? 1 : 0));
}

int main() {
  float dades[NumDades][NumCamps] = {
    { 0.1798854, 0.1176260, 0.3074593, 0.7795518, 0.6183642 },
    { 0.0738382, 0.3337906, 0.1887642, 0.4191682, 0.3355493 },
    { 0.1751513, 0.8525568, 0.6881627, 0.3511742, 0.3422691 },
    { 0.5398925, 0.2191889, 0.5840292, 0.6594841, 0.8408323 },
    { 0.0004115, 0.4917996, 0.6445988, 0.1403347, 0.2485667 },
    { 0.9294325, 0.1099306, 0.3420534, 0.4186507, 0.6168580 } };

  qsort(dades, NumDades, NumCamps * sizeof(float), compara_files);
  printMat4Vect ((float *) dades, NumDades, NumCamps);
  return 0;
}
```

La matriu ordenada (3a columna)

0.0738	0.3338	0.1888	0.4192	0.3355
0.1799	0.1176	0.3075	0.7796	0.6184
0.9294	0.1099	0.3421	0.4187	0.6169
0.5399	0.2192	0.5840	0.6595	0.8408
0.0004	0.4918	0.6446	0.1403	0.2486
0.1752	0.8526	0.6882	0.3512	0.3423

Els tokens són files senceres (de mida `NumCamps * sizeof(float)`) de la matriu de `dades`.
Per tant, inicialment, `a` és un apuntador `void` a un token que es converteix en un apuntador a `float` mitjançant el *forçament de tipus* `(float *) a`.
Com que els vectors són apuntadors, `a` es converteix en un vector de `float`'s que és una fila sencera de la matriu `dades` (i per tant té `NumCamps` components).
Resumint: `((float *) a)[2]` (respectivament `((float *) b)[2]`) es refereix a la columna 2 de la fila de la matriu que correspon al token `a` (respectivament `b`).
Nota: La tercera columna en notació "humana" és la segona en C (on es comptа a partir de zero)

Aquí usem alguna de les funcions `printMat4Vect` vistes abans (preferiblement la de la pàgina 81).

Ordenació en C: Un exemple realista

Ordenació de strings amb gestió òptima de memòria

```
#include <stdio.h>
#include <stdlib.h> //Per qsort
#include <string.h>

#define NumNoms 5

int compara_Noms(const void *a, const void *b) {
          return strcmp(*((char **) a), *((char **) b));
}

int main() { register int i;
  char AreaDeMemoriaPerGuardarTotsElsNomsArrenglerats[104];
  char * Nom[NumNoms];
  Nom[0] = AreaDeMemoriaPerGuardarTotsElsNomsArrenglerats;

  strcpy(Nom[0], "Garcia Garcia, Garcia");
  Nom[1] = Nom[0] + strlen(Nom[0]) + 1;
  strcpy(Nom[1], "Boix Serra, Ramon");
  Nom[2] = Nom[1] + strlen(Nom[1]) + 1;
  strcpy(Nom[2], "Goma Goma, Gumersindo");
  Nom[3] = Nom[2] + strlen(Nom[2]) + 1;
  strcpy(Nom[3], "Aliseda de Soto, Luis");
  Nom[4] = Nom[3] + strlen(Nom[3]) + 1;
  strcpy(Nom[4], "Carlos Primero, Don");

  printf("\n\nAra els noms ordenats\n");
  qsort(Nom, NumNoms, sizeof(char *), compara_Noms);
  for(i=0; i < NumNoms; i++)
      printf("Nom[%d] = %s\n",i, Nom[i]);
  return 0;
}
```

Sobre l'origen dels strings

En un programa realista els noms es llegeixen d'un fitxer i es faria servir memòria dinàmica. També hi hauria un procediment "a priori" per a determinar la llargada total del vector de noms arrenglerats incloent els `NumNoms` `0`'s com a marques de final d'string (és a dir el misteriós `104` del programa).

Assignació recursiva dels apuntadors `Nom[i]` a l'inici de cada un dels string's emmagatzemats correlativament a l'`AreaDeMemoriaPerGuardarTotsElsNomsArrenglerats` (veure la figura a la pàgina següent).

A cada pas *i*, `Nom[i]` ha d'apuntar a la primera posició lliure de l'AreaDeMemoria. En inicialitzar — pas 0 — fem que `Num[0]` apunti al principi de l'AreaDeMemoria.

L'operació bàsica al pas *i* és copiar (amb `strcpy`) un string a `Nom[i]`, que és la primera posició lliure de l'AreaDeMemoria. Això deixa `Nom[i]` i algunes de les posicions següents ocupades per l'string *i*.

Nota: `strcpy` afegeix el `0` de marca de final d'string.

La nova primera posició lliure de l'AreaDeMemoria és `Nom[i] + strlen(Nom[i]) + 1`. L'assignació

```
Nom[i+1] = Nom[i] + strlen(Nom[i]) + 1
```

fa que, `Nom[i+1]` apunti a la nova primera posició lliure d'AreaDeMemoria.

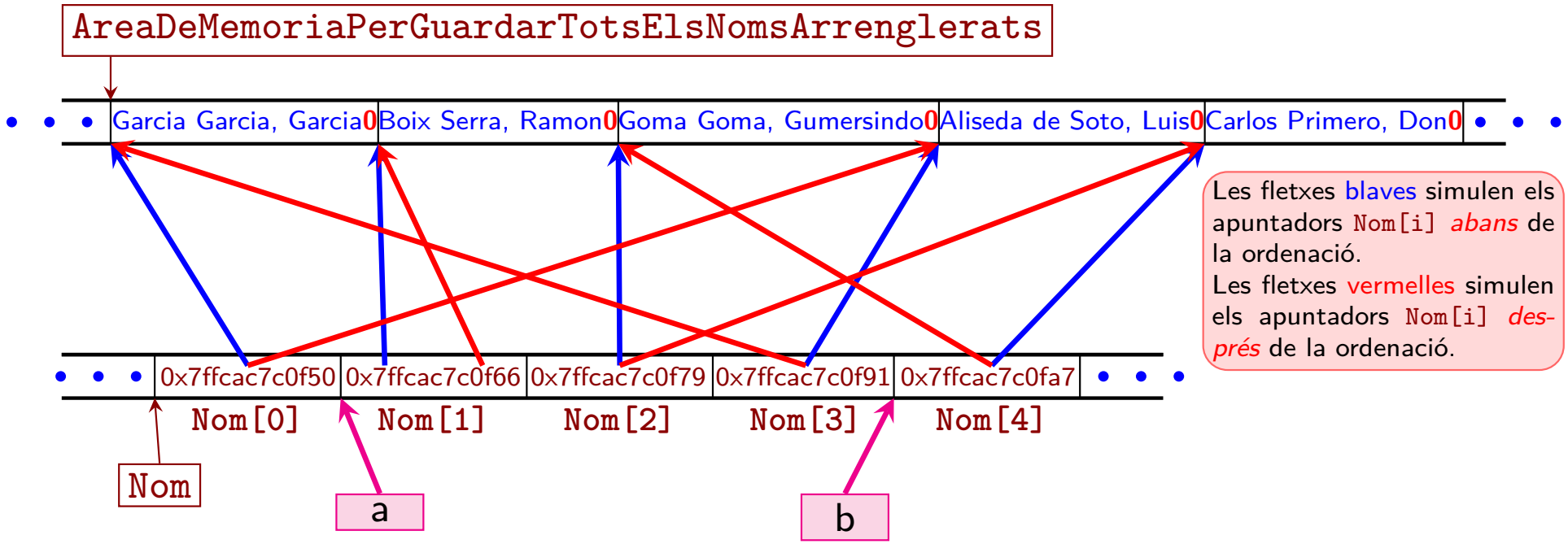
Aquesta ordenació és molt eficient ja que els noms no es mouen mai de lloc. Només es modifiquen els apuntadors `Nom[i]`.

Tots el noms ordenats

```
Nom[0] = Aliseda de Soto, Luis
Nom[1] = Boix Serra, Ramon
Nom[2] = Carlos Primero, Don
Nom[3] = Garcia Garcia, Garcia
Nom[4] = Goma Goma, Gumersindo
```

Ordenació en C: Un exemple realista

Ordenació de strings amb gestió òptima de memòria: Una imatge



Les fletxes blaves simulen els apuntadors `Nom[i]` *abans* de la ordenació.
Les fletxes vermelles simulen els apuntadors `Nom[i]` *després* de la ordenació.

On és el que cal ordenar?

Ara, els tokens són cadenes de caràcters o, el que és el mateix, apuntadors a `char` (ja que el seu `tipus` és `char *`).

`a` i `b` són apuntadors `void` als token's, que s'han de reconvertir en apuntadors a `char *`. Això és fa amb un *forçament de tipus*:

`(char **) a` i `(char **) b`

Resumint, `(char **) a` i `(char **) b` són els apuntadors `a` i `b` convertits en apuntadors a `char *`; és a dir, en apuntadors a cadenes de caràcters.

Per tant, `*((char **) a)` i `*((char **) b)` són el *contingut* de les variables apuntades per `a` i `b`. És a dir, són les cadenes de caràcters que cal comparar.

`Nom` és una *matriu no quadrada*

que no desaprofita memòria.
La manera més simple de declarar `Nom` és: `Nom[NumNoms][L]` amb
`L = strlen(string més llarg) + 1`
Però en aquest cas, totes les files corresponents a string's curts estarien desaprofitant memòria.

Índex

Motivació

En programar sovint trobem situacions on hem de tractar amb dades que són dinàmiques per naturalesa o bé el nombre de dades pot canviar durant l'execució del programa (per exemple, el nombre de clients en una cua pot augmentar o disminuir durant el procés) o bé simplement no sabem de quina dimensió necessitarem un vector o matriu durant l'execució del programa.

Això ens porta, de manera natural, a usar tècniques dinàmiques de reserva de memòria en temps d'execució; optimitzant així l'ús de la memòria.

El procés que assigna memòria en temps d'execució es coneix com *assignació dinàmica de memòria*.

Alguns llenguatges tenen l'habilitat de calcular l'espai de memòria que necessita el programa *durant* la seva execució, permetent especificar la mida de vectors i matrius quan es necessiten. El **C** no té implementada aquesta possibilitat (a nivell de llenguatge) però disposa de quatre funcions de gestió de memòria, que permeten assignar i alliberar memòria durant l'execució del programa: `malloc`, `calloc`, `free` i `realloc`

Funcions d'assignació dinàmica de memòria

Assignació d'un bloc de memòria: `void *malloc(size_t nombre-de-bytes)`

Podem assignar un bloc de memòria de mida `nombre-de-bytes` usant la funció `malloc`. Aquesta funció torna un apuntador de tipus `void` que, usant un *cast*, pot ser assignat a qualsevol tipus d'apuntador:

```
ptr = (cast-tipus *) malloc(nombre-de-bytes);
```

Si la instrucció s'executa amb èxit, `ptr` és un apuntador del tipus especificat que apunta a l'inici d'un àrea de memòria de la mida especificada. En cas contrari, `ptr` conté un `NULL`.

Exemples

```
int * x;
x=(int *) malloc(100*sizeof(int));
```

```
int (*d)[2];
d=(int (*)[2]) malloc(200*sizeof(int));
```

Si les instruccions s'executen amb èxit, `x` i `d` són apuntadors a zones de memòria de mida 100 i 200 `int`'s, respectivament. És a dir, el codi anterior és equivalent a les declaracions:

```
int x[100];
```

i

```
int d[100][2];
```

Aquesta funció s'utilitza per a reservar `n` blocs de memòria, cada un de la mateixa mida, inicialitzant tots els bytes que s'han reservat a zero. La forma general de `calloc` és:

```
ptr = (cast-tipus *) calloc(n, mida-element-en-bytes);
```

Com amb la funció `malloc`, si la instrucció s'executa amb èxit, `ptr` és un apuntador del tipus especificat que apunta a l'inici d'un àrea de memòria del nombre d'elements i mida especificada. En cas contrari, `ptr = NULL`.

En l'assignació dinàmica de memòria, és la nostra responsabilitat alliberar la memòria reservada quan no es necessita més. Això és important quan anem mancats d'espai de memòria per al nostre programa.

Quan ja no necessitem les dades que emmagatzemàvem en un bloc de memòria i no pretenem utilitzar aquell bloc per emmagatzemar qualsevol altra informació, podem alliberar-lo per a ús futur.

Això és fa amb la funció `free`:

```
free(ptr);
```

on `ptr` és un apuntador que s'ha creat utilitzant `malloc` o `calloc`.

Funcions d'assignació dinàmica de memòria

Canviant la mida de memòria assignada:

`void *realloc(void *pointer, size_t nombre-de-bytes)`

Cas que la memòria assignada utilitzant `malloc` o `calloc` sigui inadequada (sigui insuficient o en sobri), podem ajustar la mida de memòria ja assignada usant la funció `realloc`.

Aquest procés s'anomena *reassignació de memòria*.

La instrucció general de redistribució de memòria és:

```
ptrnew = realloc(ptrold, newsize);
```

Assigna un espai de memòria de mida `newsize` a `ptrnew`. Aquest espai conté les mateixes dades que hi havia a la regió apuntada per `ptrold` (truncada al mínim de la mida de `ptrold` i `newsize`).

Si `realloc` no és capaç de redimensionar la memòria inicial (és a dir de trobar un bloc de memòria *consecutiva* de mida `newsize`) al lloc apuntat per `ptrold`, busca un nou espai, copia les dades, i allibera l'apuntador anterior (`ptrold`).

Si aquesta assignació fracassa, `realloc` deixa inalterat `ptrold` i torna el valor `NULL`, que s'assigna a `ptrnew`.

Exemple d'assignació dinàmica de memòria i reassignació

```c
#include <stdio.h>
#include <stdlib.h>

main(){
   char *buffer, *aux;

/* Reservant memòria. Comprovació imprescindible */
  if((buffer = (char *) malloc(10)) == NULL){
      printf("malloc ha fallat\n"); exit(1);
  }
  strcpy(buffer,"Bangalore");
  printf("\nLa cadena de caràcters conté: %s\n", buffer);

/* Reassignació. Comprovació imprescindible */
  if((aux = (char *) realloc(buffer,30)) == NULL){
      printf("Ha fallat la reassignació.\n"); exit(1);
  } buffer = aux;
  printf("\nMida del Buffer modificada.\n");
  printf("\nLa cadena de caràcters encara conté: %s\n", buffer);
  strcpy(buffer, "Santa Perpetua de Mogoda");
  printf("\nLa cadena de caràcters ara conté: %s\n", buffer);

/* Alliberant memòria */
  free(buffer);
  buffer = NULL; /* Evita la reutilització d'un apuntador alliberat */
}
```

Veure també l'exemple de la pàgina ▸ 124 on s'utilitza l'assignació dinàmica de memòria per a emmagatzemar un fitxer amb un nombre indeterminat de línies.

Índex

La instrucció base és:

```
printf("format d'escriptura", [llista de variables a imprimir]);
```

Exemples

```
printf("L'import de la factura número %5d", NumFact);
printf("del Sr. %s puja a %d Euros.\n", NomClient, Import);
```

El format pot contenir *caràcters especials* (`\n`) i *especificadors de camp* (`%5d`).

Si s'especifiquen un nombre diferent de camps a imprimir i variables a la llista d'objectes imprimir, els resultats són imprevisibles (en general, veurem caràcters estranys a la pantalla).

Caràcters especials

`\n`	*newline*
`\f`	*form feed*
`\b`	*backspace*
`\t`	*tab*
`\nnn`	`ASCII` *nnn*
`\0nnn`	`ASCII` *nnn* (octal)
`\0xnn`	`ASCII` *nn* (hexadecimal)
`\\`	*backslash*
`%%`	*el caràcter* %

Especificadors de camp

La forma general dels especificadors de camp és:

```
%[-][0][amplada[.precisió]][l][u]tipus-dada
```

Llistat dels possibles `tipus-dada`

`d`	enter en base decimal
`i`	enter amb signe (igual a `d`)
`u`	enter no signat
`o`	enter en base vuit (octal)
`x`	enter en hexadecimal
`c`	caràcter
`s`	cadena de caràcters
`f`	real en punt flotant
`e`	real en format exponencial
`g`	real en format `e`, `f` o `d`

Modificadors dels especificadors de camp

- `amplada` delimita el nombre de caràcters mínim que es reservarà per a la impressió del camp corresponent.
  Exemple: Si volem imprimir `12`, el format `%5d` imprimeix `␣␣␣12`.
- El signe `-` indica que el camp s'alinea a l'esquerra.
  Exemple: Si volem imprimir `12`, `%-5d` imprimeix `12␣␣␣`.
- El modificador `0` indica que el camp s'ha d'omplir amb zeros a l'esquerra.
  Exemple: Si volem imprimir `12`, `%05d` imprimeix `00012`.
- El modificador `l` indica `long tipus-dada` en lloc de `tipus-dada`.
  Exemple: Cal usar `%ld` per imprimir variables del tipus `long int` i `%lf` per imprimir variables del tipus `double`.
- El modificador `u` indica `unsigned tipus-dada` en lloc de `tipus-dada`.
  Exemple: Cal usar `%ud` per imprimir variables del tipus `unsigned int` i `%uc` per imprimir variables del tipus `unsigned char`.
- En el cas dels nombres en punt flotant, es pot especificar el nombre de decimals que es desitja imprimir amb `precisió`.
  Exemple: Si volem imprimir `12.43`, el format `%10.5f` imprimeix `␣␣␣12.4300`.

Exemple: fabriquem la taula ASCII

El programa fesascii.c

```
// Programa fesascii.c
#include <stdio.h>

void main ()
{
  char codi;
  for(codi=33 ; codi < 127 ; codi++) printf("%3d <--> %c\n", codi, codi);
}
```

Sortida (parcial)

```
48 <--> 0     58 <--> :
49 <--> 1     59 <--> ;
50 <--> 2     60 <--> <
51 <--> 3     61 <--> =
52 <--> 4     62 <--> >
53 <--> 5     63 <--> ?
54 <--> 6     64 <--> @
55 <--> 7     65 <--> A
56 <--> 8     66 <--> B
57 <--> 9     67 <--> C
```

Notem que cada enter entre 33 i 126 (els caràcter visibles) l'imprimim de dues maneres diferents: com a enter %d (codi numèric) i com a caràcter %c (que tradueix el codi numèric a caràcter).

Exemple: programació de la taula *Els tipus de dades més comuns*

Programa verificacio-de-tipus.c

```
#include <stdio.h>
#include <limits.h> // Límits dels tipus de dades enters
#include <float.h> //  Límits dels tipus de dades de punt flotant

int main () {
  puts ("Verificació dels tipus de dades\n");

  printf("%-14s %6s  %s  %s\n", "Type", "length", "                  Min", "Max");
  printf("%-14s %6s  %s  %s\n", "==============", "======",
                                "====================", "====================");
  printf("%-14s %6lu  %20d  %-23u\n", "unsigned char", sizeof(unsigned char), 0, UCHAR_MAX );
  printf("%-14s %6lu  %20d  %-23d\n", "char", sizeof(char), SCHAR_MIN, SCHAR_MAX );
  printf("%-14s %6lu  %20d  %-23u\n", "unsigned short", sizeof(unsigned short), 0, USHRT_MAX);
  printf("%-14s %6lu  %20d  %-23d\n", "short", sizeof(short), SHRT_MIN, SHRT_MAX );
  printf("%-14s %6lu  %20d  %-23u\n", "unsigned", sizeof(unsigned), 0, UINT_MAX);
  printf("%-14s %6lu  %20d  %-23d\n", "int", sizeof(int), INT_MIN, INT_MAX );
  printf("%-14s %6lu  %20d  %-23lu\n", "unsigned long", sizeof(unsigned long), 0, ULONG_MAX);
  printf("%-14s %6lu  %20ld  %-23ld\n", "long", sizeof(long), LONG_MIN, LONG_MAX );
  printf("%-14s %6lu  %20Ld  %-23Ld\n", "long long", sizeof(long long), LLONG_MIN, LLONG_MAX );
  printf("%-14s %2lu  %20d  %-23Lu\n", "unsigned long long", sizeof(long long), 0, ULLONG_MAX );
  printf("-------------------------------------------------------------------\n");
  printf("%-14s %6lu  %20e  %-23e\n", "float", sizeof(float), FLT_MIN, FLT_MAX);
  printf("%-14s %6lu  %20e  %-23e\n", "double", sizeof(double), DBL_MIN, DBL_MAX );
  printf("%-14s %6lu  %20Le  %-23Le\n", "long double", sizeof(long double), LDBL_MIN, LDBL_MAX );
  return 0;
}
```

▸ Tornar a la pàgina 11

La instrucció base és:

```
scanf("format de lectura", [llista de variables a llegir amb &]);
```

La funció `scanf( );` llegeix els caràcters teclejats per convertir-los en dades del tipus especificat pels especificadors de camp del format. Les dades que s'obtenen d'aquesta operació es situen a la variable d'adreça indicada al seu argument (és a dir, a `&variable`). Aquesta lectura de caràcters del teclat s'atura quan el caràcter llegit ja no es correspon amb un possible caràcter del format especificat. La resta de caràcters queden al *buffer* del teclat en espera d'alguna altra lectura.

Els especificadors de camp, tipus de dades i modificadors són els mateixos que els del la funció `printf( );` excepte el modificador `*`.

Exemples

```
scanf( "%d %f %c", &enter, &real, &caracter);
3 2.4 a
```

```
scanf( "%d; %f, %c: %lf", &enter, &real, &caracter, &doble);
3; 2.4, a: 0.37
```

El modificador *

Llegeix *i descarta* una dada del tipus `tipus-dada`.

Exemple

```
scanf( "%*d %f", &doble);
```

descarta un `int` i un espai en blanc i llegeix la variable anomenada `doble`.

Funcions específiques d'entrada/sortida de caràcters

- `putchar( character )`: Mostra `character` per pantalla.
  Exemple: `putchar('\n');`
- `puts( string )`: Mostra la cadena de caràcters `string` per pantalla.
  Exemple: `puts( "Hola a tothom!" );`
- `getchar( )`: Retorna un caràcter llegit del teclat.
  Exemple: `Opcio = getchar( );`
- `gets( string )`: Es llegeixen caràcters de teclat (fins que es troba un `EOF` o un `'\n'`) i es guarden a la variable `string`.

Més entrada/sortida: Fitxers

La connexió del nostre programa amb fitxers de dades fitxers es fa usant *nanses* a fitxers:

```
FILE *nom-de-nansa
```

De fet una "nansa" és una variable del tipus *apuntador a fitxer* que s'inicialitza al obrir el fitxer. És el nom del nostre canal de comunicació (dins del programa) amb el fitxer. Aquest tipus de variables es defineixen al fitxer de capçalera `stdio.h`.

Hi ha tres *nanses* que estan definides sempre:

variable	significat	dispositiu
`stdin`	entrada estàndard	teclat
`stdout`	sortida de dades estàndard	pantalla
`stderr`	sortida d'errors estàndard	pantalla

Obrir i tancar: les funcions `fopen` i `fclose`

La funció base és `fopen`: crea un canal de comunicació entre un fitxer i el programa en el mode especificat pel *mode d'obertura*.

```
FILE * fopen( const char *nom-de-fitxer, const char *mode-d-obertura );
              mode-d-obertura = "r" lectura      "+" lectura/escriptura
                                "w" escriptura   "b" fitxer binary
                                "a" afegir
int fclose( FILE *nom-de-nansa );
```

Si s'aconsegueix crear el canal el fitxer es "connecta" al nostre programa i `nansa` esdevé el nom lògic del canal de connexió amb el fitxer. Si el procés falla `nansa = NULL`.

Modes d'obertura

- `r`: El fitxer ha d'existir i hem de tenir permisos de lectura.
- `w`: El fitxer es crea (si ja existia s'esborra *irrecuperablement* la versió anterior).
- `a`: Si el fitxer no existeix es crea. En cas contrari s'obre i el que hi escrivim s'afegeix a final de fitxer.
- `r+`: Obrim per lectura i escriptura. El fitxer ha d'existir. El cursor es posiciona a l'inici del fitxer.
- `w+`: Obrim per lectura i escriptura. El fitxer es crea (si ja existia s'esborra *irrecuperablement* la versió anterior). El cursor es posiciona a l'inici del fitxer.
- `a+`: Obrim per lectura i escriptura. Si el fitxer no existeix es crea. El cursor es posiciona al final del fitxer.

Lectura/Escriptura d'un fitxer

Les funcions a usar son:

```
int fprintf( FILE *nom-de-nansa, const char *format, ... );
int fscanf ( FILE *nom-de-nansa, const char *format, ... );
```

totalment anàlogues a les funcions `printf` i `scanf` excepte que el primer paràmetre és la `nansa` d'on cal llegir o escriure dades.

Nota

```
printf(format [, llista_var])  ≡ fprintf(stdout, format [, llista_var])
scanf (format [, llista_&var]) ≡ fscanf (stdin, format [, llista_&var])
```

Esquema base per a llegir/escriure dades d'un fitxer

```
#include <stdio.h>
    ....
int main()
{
   FILE *nansa;
```

```
   nansa = fopen(.....);  // Inicialització
   if(nansa == NULL){ // OBLIGATORI comprovar que s'ha obert el fitxer
      puts(... treure un missatge d'error ...);
     return 1;
   }
```

```
   while(fscanf(nansa, ... ) != EOF) {//Recorregut seqüencial del fitxer
          instrucció;
             . . .
   } /* Fi del while */
```

```
   fclose( nansa ); // Tanquem el fitxer
   return 0;
}
```

Exemple: Lectura estàndard de fitxers

Llegim les dades d'un fitxer que té dues columnes enteres i calculem una mitjana ponderada de cada línia: $0.3 \cdot a + 0.7 \cdot b$

```
#include <stdio.h>

int main ()
{
    int a,b;
    float w1=0.3, w2=0.7;
    char NomIn[]="duescolumnes.dat", NomOut[]="duescolumnes.res";
    FILE *fin, *fo;


    if ((fin = fopen (NomIn, "r")) == NULL)  {
        fprintf (stderr, "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n", NomIn);
        return 1;
    }

    if ((fo = fopen (NomOut, "w")) == NULL)  {
        fprintf (stderr, "\nERROR: El fitxer '%s' no es pot crear...\n\n", NomOut);
        return 2;
    }

    while(fscanf(fin, "%d %d\n",&a,&b) != EOF) fprintf(fo,"%d %d -> %f\n", a, b, w1*a+w2*b);

    fclose (fin);  fclose(fout);
    return 0;
}
```

Exemple: Lectura estàndard d'un fitxer i memòria dinàmica

Usem una matriu dinàmica en funció del nombre de línies

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

int main () {   int i, nlin=0;
    char c;
    FILE *fin;
    int (*dades)[2]; // Apuntador a vectors de dues components: dades[i] apunta a dades[i][0]

    if ((fin = fopen ("dades.dat", "r")) == NULL)  {
        fprintf (stderr,"\n ERROR: El fitxer 'dades.dat' no existeix o no es pot obrir...\n\n");
        return 1; }

    while((c = fgetc(fin)) != EOF){ if(c == '\n') nlin++; } // Comptem el número de línies
    rewind(fin);

    if((dades = (int (*)[2]) malloc(2*nlin*sizeof(int))) == NULL){
       fprintf (stderr, "\nERROR: No es possible assignar la memòria necessària...\n\n");
       return 1; }

    for(i=0; i < nlin ; i++) fscanf(fin, "%d %d\n",dades[i] , dades[i]+1) ;
// {int (*aux)[2] = dades; for(i=0;i<nlin;i++,aux++) fscanf(fin,"%d %d\n", *aux, (*aux)+1);}//Equivalent
    fclose (fin);

    for(i=0; i < nlin ; i++) printf("%d %d\n", dades[i][0], dades[i][1]);
//  for(i=0; i < nlin ; i++, dades++) printf("%d %d\n", *(*dades), *((*dades)+1)); // Equivalent
    return 0;
}
```

▸ Tornar a les pàgines 70 ▸ 108

Taula de les funcions d'entrada/sortida a fitxers

Funcions d'accés i control de fitxer

```
FILE * fopen( const char *nom-fitxer, const char *mode );
int fclose ( FILE *nom-de-nansa );
void rewind( FILE *nom-de-nansa );
```

Funcions de lectura

```
int fgetc( FILE *nom-de-nansa );
char * fgets( char *string, int maxchar, FILE *nom-de-nansa );
int fscanf( FILE *nom-de-nansa, const char *format, ... );
```

Funcions d'escriptura

```
int fputc( int character, FILE *nom-de-nansa );
int fputs( const char *string, FILE *nom-de-nansa );
int fprintf( FILE *nom-de-nansa, const char *format, ... );
```

Funcions d'accés directe

```
long ftell( FILE *nom-de-nansa );
int  fseek( FILE *nom-de-nansa, long int salt, int direccio );
    direccio  = SEEK_SET endavant a partir de l'inici de fitxer
                SEEK_CUR endavant a partir de la posició actual
                SEEK_END endarrere a partir del final de fitxer
```

Nota

La manera més ràpida de llegir un fitxer és caràcter a caràcter usant la funció `fgetc`.

Per tant,

la manera més ràpida d'anar al final de la línia actual és:

```
while(fgetc(fin) != '\n'){}
```

Exemple 1: Lectura d'una columna d'un fitxer

Tenim un fitxer que té diverses columnes i en volem llegir una (NCOL)

```
#include <stdio.h>
/* Columna que volem extreure
   Ull: El programa solament funciona si el fitxer té al menys NCOL columnes
        Això s'hauria de comprovar */
#define NCOL 4

int main ()
{
    int x,n;
    char NomIn[]="noucolumnes.dat";
    FILE *fin, *fout=stdout;


    if ((fin = fopen (NomIn, "r")) == NULL)  {
        fprintf (stderr, "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n", NomIn);
        return 1;
    }

/* Comencem llegint una dada per a comprovar que no hem arribat a final de fitxer
   El 'for' llegeix fins la columna NCOL. Ara x és la dada que volem
   (si NCOL=1 aquest 'for' no s'executa)
*/
    while(fscanf (fin, "%d", &x) != EOF){
        for(n=2;n<=NCOL;n++) fscanf (fin, "%d", &x);
        fprintf(fout, "%d\n", x);
        while(fgetc(fin) != '\n'){} // Llegim fins a final de línia sense guardar
    }
    fclose (fin);  fclose(fout);
    return 0;
}
```

Exemple 2: Transposa les columnes d'un fitxer

Volem escriure un fitxer que té TOTCOLS columnes de manera que cada columna passi a ser una fila (usem el codi de l'Exemple 1)

```
#include <stdio.h>
#define TOTCOLS 9
void EscriuUnaColumnaEnFila(FILE *, FILE *, int);
int main () { int col;
              char NomIn[]="noucolumnes.dat";
              FILE *fin, *fout=stdout;

    if ((fin = fopen (NomIn, "r")) == NULL)  {
        fprintf (stderr, "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n", NomIn);
        return 1;
    }

   for(col=1; col <= TOTCOLS; col++){
      EscriuUnaColumnaEnFila(fin, fout, col);
      rewind(fin); //EscriuUnaColumnaEnFila arriba a final de 'fin'. "Rebobinem" a cada iteració
   }
   fclose (fin);  fclose(fout); return 0;
} /* Fi del main */
// fem servir una funció per facilitat i claredat
void EscriuUnaColumnaEnFila(FILE *fin, FILE *fout, int col)
{   int x, n;
    while(fscanf (fin, "%d", &x) != EOF){
      for(n=2;n<=col;n++) fscanf (fin, "%d", &x);
      fprintf(fout, "%d ", x);
      while(fgetc(fin) != '\n'){} // Llegim fins a final de línia sense guardar
    }
    fputc('\n', fout);
} /* Fi de la funció EscriuUnaColumnaEnFila */
```

Exemple 3: Lectura eficient amb moltes columnes

Volem llegir una columna evitant la lectura completa de cada fila (accés directe)

```
#include <stdio.h>
#define NCOL 4 /* Columna que volem extreure
Ull: El programa solament funciona si el fitxer: té al menys una línia; té al menys NCOL columnes
     i cada columna té l'amplada fixada en totes les línies del fitxer (no cal que cada columna
     tingui la mateixa amplada)                                                              */

int main () { int x,n;
              char NomIn[]="noucolumnes.dat";
              FILE *fin, *fout=stdout;
              long M, P;

    if ((fin = fopen (NomIn, "r")) == NULL)  {
        fprintf (stderr, "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n", NomIn);
        return 1;
    }

// Inicialització
    for(n=1;n<NCOL;n++) fscanf (fin, "%*d"); P = ftell(fin);
    fscanf (fin, "%*d"); M = ftell(fin) - P;
    while(fgetc(fin) != '\n'){}
    M = ftell(fin) - M;

// Procés de lectura
    fseek(fin, P, SEEK_SET); // Anem al principi de la columna NCOL de la primera línia
    while(fscanf (fin, "%d", &x) !=EOF){
          fprintf(fout, "%d\n", x);
          fseek(fin, M, SEEK_CUR);
    }
    fclose (fin);  fclose(fout); return 0;
}
```

Lectura eficient d'un fitxer amb moltes columnes: comentari detallat de la inicialització

Calculem on és l'inici de la columna NCOL

```
for(n=1;n<NCOL;n++) fscanf (fin, "%*d");
P = ftell(fin);
```

Calculem l'amplada de la columna NCOL

```
fscanf (fin, "%*d");
M = ftell(fin) - P;
```

Trobem la mida d'una línia descomptant l'amplada de la columna NCOL. Observem que això és el que cal saltar per anar del final de la columna NCOL d'una línia al principi de de la columna NCOL de la línia següent

```
while(fgetc(fin) != '\n'){}
M = ftell(fin) - M;
```

Entrada sortida a cadenes de caràcters

Apart de les nombroses operacions d'strings que es poden fer amb les funcions de la secció *Cadenes de caràcters*, pot resultar molt útil usar un string de *buffer* com si fos una sortida o una entrada de dades. Amb això es pot

- fabricar una cadena complicada a partir d'altres cadenes, variables numèriques i resultats d'expressions
- omplir diverses cadenes i/o variables numèriques extraient informació total o parcial d'una cadena de caràcters complicada.

Per a això existeixen les versions per a cadenes de les funcions `printf( )` i `scanf( )`:

```
sprintf( cadena, format [, llista_variables ] );
sscanf ( cadena, format [, llista_&variables ] );
```

En tots dos casos, `cadena` es refereix a la cadena de caràcters sobre la qual es farà l'operació.

Exemples

```
float a=3.73, b=4.5, c=6.7;
char Kad[31]="hola";
char CadCom[81];
sprintf ( CadCom, "%4.2f %s %4.2f %5.3f" , a, Kad, b, c);
```

Resultat: `CadCom` $\longrightarrow$ `3.73 hola 4.50 6.700`

```
char Nns[31]="3.73 hola 4.5 6.7";
float a;
sscanf ( Nns, "%*f %*s %f" , &a );
```

Resultat: `a` $\longrightarrow$ `4.5`

Paràmetres i valors de retorn del `main`

Índex

Paràmetres del `main`

El programa `parametres.c`

Il · lustra com es defineixen i s'utilitzen els paràmetres del `main`

```
// Programa parametres.c
#include <stdio.h>
int main (int argc, char *argv[])
{
  int n;

  printf("Número de paràmetres: %d\n", argc);

  if(argc == 1){
    printf("Error: No hi ha paràmetres\n\n");
    return 1;
  }

  for(n=0; n<argc; n++){
    printf("Paràmetre %d: %s\n",n,argv[n]);
  }
  return 0;
}
```

Entrada de paràmetres numèrics al `main`

Es fa usant les funcions "`ato`":

- `atof` converteix strings a `double`;
- `atoi` converteix strings a `int`;
- `atol` converteix strings a `long`.

Els seus prototipus són a `stdlib.h`.

Exemple

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

int main (int argc, char *argv[])
{
  double sigma;
  unsigned long trans, iter;

// Testing parameters and open output file
  if (argc != 4)
    {
      printf ("\nÚs: %s sigma trans niter\n\n", argv[0]);
      return 1;
    }

  sigma = atof (argv[1]);
  trans = (unsigned long) atol (argv[2]);
  iter = (unsigned long) atol (argv[3]);
```

Nota

Les funcions "`ato`" intenten convertir strings a nombres. Paren la conversió quan troben un caràcter il · legal (segons el tipus de conversió demanada) i retornen el valor computat fins llavors. Per tant, *SEMPRE cal controlar el valor de retorn de les funcions "ato"*.

Exemple:

`atoi(3.24)` $\longrightarrow$ `3`

`atoi(a=3.24)` $\longrightarrow$ `0`

`atof(3.24g4536)` $\longrightarrow$ `3.24`

`atof(a=3.24)` $\longrightarrow$ `0.0`

Exercici (paràmetres del `main` i cadenes de caràcters)

Volem escriure un programa que rebi `<nom>.ext` com un dels paràmetres del `main` i obri aquest fitxer per lectura. També volem obrir per escriptura un fitxer que es digui `<nom>.res`.

```
#include <stdio.h>
#include <string.h>
#define MAXLENNOM 251

void MissErr(const char * nomftx, const char *miss){
      fprintf (stderr, "\nERROR al obrir '%s': %s.\nParem...\n\n", nomftx, miss);
}

int main (int argc, char *argv[])
{   char nomsortida[MAXLENNOM];
    FILE *fin, *fout;

    if ( argc < 2 || strrchr(argv[1], '.') == NULL ) { // argv[1] ha de tenir contingut i un punt
        fprintf (stderr, "\nUs: %s <nomfitxer>.ext\n\n", argv[0]); return 1;
    }

    if(strlen(argv[1])+3 >= MAXLENNOM) {
          MissErr(argv[1], "El nom del fitxer és massa llarg"); return 11; }

    if ((fin = fopen (argv[1], "r")) == NULL) {
            MissErr(argv[1], "El fitxer no existeix o no es pot obrir");  return 12; }

    strcpy(nomsortida, argv[1]);  strcpy(strrchr(nomsortida, '.'), ".res");
    if ((fout = fopen (nomsortida, "w")) == NULL) {
             MissErr(nomsortida, "No es pot obrir el fitxer de sortida");  return 13; }
```

El programa anterior com funciona?

Funcionament de `strrchr(kdna, 'c')`

Aquesta funció busca la primera ocurrència del caràcter `c` a `kdna` començant per darrere, i torna un apuntador al substring de `kdna` que va d'aquest lloc al final del de `kdna`. Si no troba el caràcter `c` torna `NULL`.

- La condició `strrchr(argv[1], '.') == NULL` és certa quan `argv[1]` no conté cap `'.'`. Això és erroni perquè hem demanat que el fitxer sigui de la forma `<nom>.ext` (que ha d'incloure un `'.'`).
- `strcpy(nomsortida, argv[1]);` copia `argv[1]` a `nomsortida`.
- `strcpy(strrchr(nomsortida, '.'), ".res");` Notem que `strrchr(nomsortida, '.')` és un apuntador al substring de `nomsortida` que va del darrer punt fins al final (és a dir, és un apuntador a l'extensió del nom del fitxer). La instrucció `strcpy(strrchr(nomsortida, '.'), ".res");` copia (matxucant) `".res"` sobre l'extensió (donada per `strrchr(nomsortida, '.')`). Així obtenim `<nom>.res`.

Valors de retorn del `main`

El següent fitxer és un *script* de `bash` (la consola de Linux) que mostra com els valors de retorn del `main` passen al sistema operatiu i permeten decidir què passa a continuació de l'execució d'un programa (en funció de si ha acabat correctament o amb errors).
En aquest codi, `$@` dóna els paràmetres de la línia de comandes que ha iniciat l'execució de l'"script" mentre que `$?` és el codi de retorn de la darrera instrucció executada (en aquest cas del programa `parametres`).

L'script `controla-parametres`

```
#!/bin/bash
# Script controla-parametres

clear # Neteja la pantalla
# 'echo' escriu coses a pantalla. 'echo -e' admet caràcters de control com '\n'
echo "Anem a executar el programa parametres amb els paràmetres:"
echo $@
echo

./parametres $@
if test $? -ne 0 ; then
    echo -e "El programa no ha funcionat\nParem el procés..."
    exit 1
fi

echo -e "El programa ha funcionat\nContinuem treballant...."
exit 0
```

Uff que la programació enganxa!! Bona sort!!