Curs de C

Eines Informàtiques per a les Matemàtiques

Departament de Matemàtiques UAB*
Edició: Joaquim Roé

Versió: 26 de febrer de 2024

Índex

1 Començant a programar amb C **3**
1.1 Estructura del programa .c 3
1.2 Fem els primers càlculs. Declaració de variables 4
1.3 Control de flux: alternatives 7
1.4 Control de flux: iteració 9

2 Programació estructurada **13**
2.1 Funcions 13
2.2 Variables locals i globals 16

3 Arrays I: vectors i matrius **27**
3.1 Declaració i ús d'arrays 27
3.2 Adreces de variables i arrays dins la memòria 29

4 Arrays II: Apuntadors **38**
4.1 Variables apuntador 38
4.2 Els vectors com apuntadors 42

5 Arrays III: Cadenes de caràcters **49**
5.1 Variables que contenen text 49
5.2 Operacions en cadenes 51
5.3 Paràmetres de la funció main 61

6 Assignació dinàmica **63**
6.1 Les instruccions malloc, calloc i free 64
6.2 Assignació dinàmica de matrius 67

7 Fitxers **73**
7.1 Obrir i tancar fitxers, nanses 73
7.2 Entrada/sortida de caràcters 76
7.3 Fitxers binaris 82

*Aquestes notes són la plasmació, com a obra col·lectiva, de l'experiència docent de molts professors del Departament de Matemàtiques de la UAB, principalment Aureli Alabert, Lluís Alsedà, Jaume Coll, Gregori Guasp, Josep Maria Mondelo, Joaquim Roé i Albert Ruiz. Joaquim Roé ha fet l'edició de la versió actual en els cursos 2021-2023.

8 Tipus de dades **86**
8.1 Estructures . . . 87
8.2 Apuntadors a estructures . . . 89

9 Estructures enllaçades **95**
9.1 Llistes enllaçades . . . 95
9.2 Variants . . . 100

10 Biblioteques i fitxers de capçalera **109**
10.1 Biblioteques definides per l'estàndard del C . . . 109
10.2 Biblioteques matemàtiques de codi obert . . . 115

Preliminars. Què és el C?

El C és un llenguatge de programació de mig nivell amb algunes característiques de baix nivell (és a dir, és més proper al codi binari utilitzat internament per l'ordinador que no pas, per exemple, el llenguatge Python que heu utilitzat per programar dins de SageMath, el qual és d'alt nivell i per tant més proper als llenguatges humans).

El llenguatge C va ser creat per Dennis Ritchie i Ken Thompson als Laboratoris Bell d'AT&T, a principis de la dècada dels 70, partint d'un llenguatge anterior, que s'havia anomenat B. El van introduir per a poder desenvolupar el sistema operatiu Unix en un llenguatge que fos més flexible que l'assemblador (que és de baix nivell). Actualment, C és el llenguatge més utilitzat per a desenvolupar sistemes operatius i altres tipus de programari bàsic, i també és molt utilitzat per programar aplicacions en general i software científic en particular quan aquest requereix un ús eficient dels recursos (temps de processador i memòria).

Per què C?

El llenguatge C és molt apreciat per l'eficiència del codi que produeix. En el cas de les matemàtiques, resulta avantatjós quan cal realitzar càlculs costosos, ja sigui en temps o memòria. Dominar-lo implica assolir uns coneixements significatius de la representació interna de les dades (per exemple, numèriques) en els ordinadors, que us seran molt útils per entendre el funcionament i programar algoritmes en qualsevol llenguatge.

Exemple 0.1

Podem apreciar la diferència entre Python/SageMath (alt nivell) i C (nivell mitjà/baix) comparant dues versions de l'algoritme que calcula una aproximació de $\pi = 4\int_0^1 \sqrt{1-x^2}dx$ com la suma de Riemann que resulta de dividir l'interval $[0,1]$ en 10^9 intervals, programades en Python/SageMath i C respectivament.

```
import math
INTERVALS = 1000000
d = 1/INTERVALS
print ( 4 * d * sum(math.sqrt(1 - (d*x) ** 2) for x in range(INTERVALS)))
```

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

int main()
{
   const unsigned int INTERVALS = 1e6;
```

```
    double pi = 0, d = 1. / INTERVALS, x = 0;
    for(unsigned int n = 0; n < INTERVALS; n++, x += d)
       pi += d * sqrt(1 - x * x);
    pi = 4 * pi;
    printf("%lf\n", pi);
    return 0;
}
```

Ja a primer cop d'ull s'observen clares diferències entre els dos programes, ja que el segon (en C) necessita més línies de codi i la seva sintaxi (paraules clau i símbols especials) és més complexa. El C té un nombre relativament petit d'instruccions i construccions sintàctiques, i per exemple no té una funció `sum` per fer un sumatori com Python/SageMath sinó que cal programar-la (ja practicarem com fer-ho). No obstant, en executar-los, el resultat és el mateix: una aproximació del nombre π, i —en les implementacions més estàndard— l'execució en Python triga significativament més temps i necessita més memòria que el programa en C (tot i que hi ha tècniques per compilar el programa en Python i aconseguir una eficiència similar al C).

Com funciona?

El llenguatge C és (gairebé sempre) *compilat*. Això significa que, per a poder executar el programa que hem escrit en llenguatge C, primer cal *traduir-lo* al llenguatge binari de la màquina mitjançant un *compilador*.

Bondia.c ⟶ `gcc` [fitxers auxiliars] ⟶ Bondia

A partir del fitxer font, que és un fitxer de text creat amb un editor, el compilador `gcc` genera un fitxer executable, capaç de ser cridat i fer la feina, possiblement crent fitxers auxiliars en el procés.

1 Començant a programar amb C

1.1 Estructura del programa .c

Exemple 1.1

Introduïu aquestes línies en un editor de text (Kate, Code::Blocks, Eclipse, Notepad++,...) i deseu el fitxer font amb el nom `bondia.c`.

```
// Programa bondia.c
// Autor: Professors Dpt Matemàtiques
// Data: Gener 2020
// Descripció: Programa que escriu "Bon dia a tothom" en pantalla

#include <stdio.h>

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

Podeu compilar el programa des de la consola amb la instrucció

- `gcc -o Bondia Bondia.c` (en Linux/MacOS)

- `gcc -o Bondia.exe Bondia.c` (en Windows)

(si feu servir un *entorn integrat* com Code::Blocks pot ser suficient fer clic a la icona adequada). Per executar-lo, tant si heu fet la compilació des de la línia de comandes com dins d'un entorn integrat, localitzeu el directori on es troba i a continuació:

- A la consola, escriviu `./Bondia` (en Linux/MacOS) o `Bondia.exe` (en Windows).

Analitzant el codi anterior podem distingir tres blocs: en el primer, totes les línies comencen amb `//` i el contingut és **documentació** sobre el programa; el segon en aquest cas només consta d'una instrucció `#include`; el tercer és la **funció** anomenada `main`.

En general, el contingut del programa previ a la funció `main` s'anomena **preàmbul**, i consta d'algunes o totes les següents parts:

- **Documentació** útil per als autors i usuaris del codi. El compilador ignora aquestes línies.
- Instruccions `#include` per usar **biblioteques** de funcions predefinides. Més endavant aprofundirem en el seu ús.
- Instruccions `#define` per introduir **constants** i **macros**.
- **Declaracions globals** de funcions (prototipus) i variables. A la segona pràctica aprofundirem en el seu ús. En programes que usen diverses funcions, trobarem la seva definició *després* de la funció `main`.

Important: Totes les instruccions de C s'han d'acabar en punt i coma ;

Exercici 1.1

Esbrineu què fan els símbols `\n` en el programa `bondia.c`

1.2 Fem els primers càlculs. Declaració de variables

Una diferència important entre el llenguatge C i d'altres com Python o SageMath és que abans de poder usar una variable numèrica, aquesta s'ha de *declarar*, assignant-li un *tipus*, que serà diferent segons si els valors que li hem d'assignar seran enters o reals.

Exemple 1.2

El programa següent permet calcular l'àrea d'un cercle de radi donat per l'usuari. Copieu-lo, compileu-lo i executeu-lo:

```c
// Programa area.c
// Autor: Professors Dpt Matemàtiques
// Data: 2010/2021
// El programa demana a l'usuari un radi i imprimeix l'àrea del cercle
   corresponent

#include <stdio.h>

const double PI = 3.1415926535897932;

int main()
{
```

```
    double r;

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

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

    return 0;
}
```

Analitzem els components del programa.

- `const double PI = 3.1415926535897932;`
  Si necessitem usar alguna constant com el nombre π, l'hem de declarar. En aquest cas s'ha fet al preàmbul, fent-la *global*.
- `double r;`
  Declarem la variable `r` com a tipus `double`. Aquesta és una especificació adequada per a treballar amb nombres reals. La constant `PI` també és del tipus `double`, però amb el modificador `const` informem el compilador que el seu valor no canviarà.
- `scanf`, `printf`
  Les funcions `printf` i `scanf` estan definides a la biblioteca `stdio.h`, i serveixen per escriure en pantalla i llegir de teclat, respectivament. La instrucció `scanf ("%lf", &r);` llegeix un nombre amb decimals del teclat i el guarda a la variable `r`.
- `%lf`
  Es refereix al format de lectura d'una variable real. Els caràcters `lf` fan referència a *long* `float` (tot i que en l'actualitat no s'usa gaire, `float` és una especificació adequada per a nombres reals "curts", amb menor rang i precisió dels nombres).
- `%14.6lf`
  Es refereix al format d'escriptura d'una variable real. La sortida ocuparà 14 caràcters, dels quals 6 reservats per als decimals.

Exercici 1.2

En una mateixa instrucció `printf` es pot combinar text amb la impressió de diverses variables en el format adequat. Canvieu el segon `printf` del programa anterior per

```
    printf("\nL'àrea d'un cercle de radi %lg és: %lg\n\n", r, PI * r * r);
```

En aquest cas, el valor de la variable `r` apareixerà en el lloc del primer codi de format `%lg`, i el valor πr^2 en el lloc del segon.

L'especificador de format `%lg`, com `%lf`, és adequat per a nombres reals, però no sempre presenta el resultat en la mateixa forma; experimenteu amb *valors molt grans* del radi per descobrir la diferència.

Exercici 1.3

Escriurem un programa que ensenyarà per pantalla el producte de dos nombres.

(a) Creeu un programa nou amb el nom `producte.c`. Introduïu el preàmbul adequat i una funció `main` (definint les variables que calguin) que contingui les línies següents:

```
    printf("Valor de x\n");
    scanf("%lf", &x);
    printf("Valor de y\n");
    scanf("%lf", &y);

    printf("El producte de %lg per %lg és %lg\n", x, y, x * y);
```

Compileu-lo i executeu-lo per a calcular el següent producte: $43 \cdot 21$.

(b) Podríem haver llegit els dos nombres de la manera següent:

```
    printf("Valors de x i de y\n");
    scanf("%lf %lf", &x, &y);
```

Comproveu-ho efectuant els canvis adequats en el programa. A l'hora d'introduir els nombres, `x` i `y`, caldrà separar-los amb un espai o bé introduir el primer nombre, `x`, prémer `Retorn` i després introduir el segon nombre, `y`.

Usant la versió modificada del programa, calculeu els següents productes: $-14.25 \cdot 53$, $-1425 \cdot (-2430)$.

Les variables que han de contenir valors numèrics reals es declaren de tipus `float`, `double`, o `long double`. Les que han de contenir valors enters, es declaren de tipus `int`, `long int` o `long long int`, amb el modificador `unsigned` si no cal admetre valors negatius. Quan es declara la variable, el compilador li reserva uns quants bytes de memòria, que depenen del tipus. En la mateixa instrucció de declaració es pot donar un valor a la variable (*inicialització*).

Exemple 1.3

```
// Programa divisio.c
// Autor: Professors Dpt Matemàtiques
// Data: Gener 2020
// El programa demana a l'usuari un enter i calcula el quocient i residu
// de la divisió entre 5

#include<stdio.h>

int main()
{
    int a, b = 5, q, r;

    printf("Quin nombre vols dividir entre 5? ");
    scanf("%d", &a);

    q = a / b;
    r = a % b;

    printf("El quocient de la divisió és %d, i el residu és %d.\n", q, r);

    return 0;
}
```

- Observeu com en la mateixa instrucció hem declarat quatre variables, `a`, `b`, `q`, `r`, i hem inicialitzat la variable `b` amb el valor 5.
- L'operador `/`, aplicat a nombres enters, dona un valor enter (per exemple, `1/2` dona `0`).

- L'operador `%` (mòdul) calcula el residu d'una divisió entre dos enters.
- L'especificador de format `%d` és adequat per a nombres enters en notació decimal. També es pot usar `%i` (de integer).

Exercici 1.4

Modifiqueu el programa anterior perquè calculi el quocient i el residu entre dos nombres introduïts per l'usuari.

1.3 Control de flux: alternatives

Quan cal que el programa decideixi entre dues accions alternatives segons si es compleix o no una condició, s'usa la sintaxi de sota. La condició a avaluar pot involucrar fórmules matemàtiques, potser combinades amb connectors lògics. Els operadors disponibles en C són els de la taula de la dreta, ordenats per ordre de precedència (els de més amunt s'avaluen primer). La precedència es pot cancel·lar usant parèntesis.

```
if(condicio)
{
   instruccio;
   . . .
   instruccio;
}
else
{
   instruccio;
   . . .
   instruccio;
}
```

Operador	Significat
`!`	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

Exemple 1.4

Escriurem un programa que mostri per pantalla l'arrel quadrada d'un nombre real positiu o zero. Escriviu el següent codi amb l'editor i guardeu–lo amb el nom de `arrel.c`.

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

int main()
{
   double x;

   printf("Introdueix el valor de x\n");
   scanf("%lf", &x);

   if(x < 0) {
      printf("L'arrel quadrada de %lf no es real\n", x);
      return 1;
```

```
    }
    else {
        printf("L'arrel quadrada de %lf es %lf\n", x, sqrt(x));
        return 0;
    }
}
```

- `if`
  Aquesta instrucció avalua una condició; si se satisfà, aleshores executa el que hi ha entre claus. La part `else` és opcional, i s'executarà en cas que no es satisfaci la condició del `if`. En aquest cas, `else` s'executarà si el valor de `x` és positiu o zero.

- `sqrt`
  És una funció que calcula l'arrel quadrada d'un nombre `double`. Per a poder fer servir aquesta funció o qualsevol altre funció matemàtica cal fer el següent:

  – Al codi del programa s'ha d'incloure el fitxer de capçalera `math.h`.
  – En Linux, si ho compilem des de la línia de comandes, cal afegir (cada vegada que compilem) l'opció `-lm`:

    ```
    gcc -Wall -o arrel arrel.c -lm
    ```

    per carregar la biblioteca que conté les funcions matemàtiques bàsiques. Si ho compilem des de **Code::Blocks**, per carregar la biblioteca matemàtica cal anar al menú `Settings`, obrir `Compiler Settings`, i a la pestanya `Linker settings` afegir `libm` a la finestra blanca de l'esquerra (prement el botó `Add`). Això només cal fer-ho una vegada.

Exercici 1.5

Creeu un programa (haureu de declarar les variables necessàries, i demanar amb `scanf` els valors de a, b, c) per trobar les solucions de l'equació de segon grau $ax^2 + bx + c = 0$ basat en les línies següents.

```
    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: %lf i %lf\n", (-b + disc) / a, (-b - disc)
    / a);
    }
```

- En el cas que un bloc d'instruccions d'un `if` consta d'una sola instrucció, es poden ometre les claus:

  ```
      if(disc < 0)
          printf("Discriminant negatiu. No hi ha arrels reals.\n");
  ```

- L'expressió de la condició que determina quines instruccions s'executaran és ella mateixa una instrucció que pot *fer coses* com assignar valors. Les dues primeres instruccions es poden combinar així:

```
  if((disc = b * b - 4 * a * c) < 0)
     printf("Discriminant negatiu. No hi ha arrels reals.\n");
```

En el fons cada comparació retorna 1 o 0 depenent de si es compleix o no, i el que fa la instrucció `if` és mirar si el resultat de la comparació és diferent de zero.

- Per calcular b^2 fem `b * b`. L'operador `^` existeix però no serveix per calcular potències (és l'operador `XOR` bit a bit).

1.4 Control de flux: iteració

Quan cal *repetir* una acció s'usa una de les instruccions `for`, `while` o `do–while`:

```
for(
   inicialitzacio; // Per exemple: i = 0
   condicio;       // Per exemple: i < LIMIT
   continuacio;    // Per exemple: i = i + 1
)

{
   instruccio;
   . . .
   instruccio;
}
```

```
while(condicio)
{
   instruccio;
   . . .
   instruccio;
}
```

```
do
{
   instruccio;
   . . .
   instruccio;
}
while(condicio);
```

Exemple 1.5

Escriurem un programa que calcula potències de 3 utilitzant el `for`.

```
// Programa potencies.c
// Autor: Professors Dpt Matemàtiques
// Data: 2010-2013
// Càlcul de potències de 3

#include <stdio.h>

int main()
{
   int n, i, pot = 1;

   printf("Calcul de les potencies de 3\n");
   printf("Quantes en vols calcular?\n");
   scanf("%d", &n);

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

```
    return 0;
}
```

El programa executarà repetidament les instruccions entre les claus que segueixen el `for`, variant els valors de `i` des de `1` (la inicialització és el primer argument), mentre es compleixi la condició `i <= n` (el segon argument) i d'un en un (el tercer argument, `i++`, significa el mateix que `i = i + 1`).

Exercici 1.6

(a) Deseu el programa anterior amb el nom de `potencies.c` i compileu–lo. Executeu-lo per a calcular les 5 primeres potències de 3.

(b) Torneu a executar-lo per a calcular les 20 primeres potències de 3. Comproveu que el resultat no és correcte. Per a poder calcular 3^{20} de manera correcta, declareu la variable `pot` com `unsigned int`. També haureu de canviar el format `%d` a `%u` (que és el format per escriure o llegir dades `unsigned int`) quan s'escrigui la variable `pot`. Noteu, però, que no calcula bé 3^{21}. (*Nota*: Per a una variable de tipus `int` (que és equivalent a `long int`) s'utilitzen 4 bytes, la mateixa quantitat que per a una variable de tipus `unsigned int`. La diferència entre aquests dos tipus de variable és que `unsigned int` és un enter sense signe, i, per tant, es pot representar un rang doble d'enters positius que amb una variable de tipus `int`).

Exercici 1.7

Escriviu un programa en C que calculi la suma de n nombres reals. Per fer aquesta tasca només necessiteu tres (o quatre) variables: `n` (entera), el nombre de sumands; `x` (real) per introduir cada sumand (es pot reaprofitar la mateixa variable cada cop, ja que no cal "recordar" els sumands individualment); `acumulat` (real) la variable on s'aniran sumant els valors i contindrà el resultat final; i opcionalment, `i` (entera), l'índex del bucle.

El primer de tot que ha de fer el programa és demanar la quantitat de nombres que volem sumar, `n`, i introduir-lo pel teclat. Caldrà usar la funció `scanf` amb el format adequat. Cadascun dels n nombres s'ha de llegir del teclat, i per tant, també caldrà que useu la funció `scanf` amb el format adequat. Caldrà un bucle `for` o `while` per fer-ho. El resultat de la suma caldrà escriure'l per pantalla. Per a fer això cal la funció `printf`. Deseu aquest programa amb el nom `suma.c`.

Alguns detalls avançats

El codi següent és *erroni* perquè no fa allò desitjat, però es compilarà sense errors (potser surti un warning):

```
if(a = 1) b = 1;
```

El problema està en l'operador `a = 1`, que és una assignació, quan la intenció era incloure l'operador de comparació `a == 1`. Vegem per parts com s'interpretarà aquesta sintaxi en compilar, i què farà el programa.

Comprovacions: cert si no és zero

Un operador de comparació (<, >, ==, !=, <=, >=) produeix un *valor numèric*, igual que els operadors aritmètics. Aquest valor numèric és zero si l'enunciat de la comparació és fals, i un valor diferent de zero (normalment, 1) si és cert. En conseqüència, el paràmetre `condicio` que reben les instruccions `if`, `while` i `for` és de fet un enter, i s'interpretarà com a `cert` excepte quan valgui zero.
Això vol dir, per exemple, que tota comparació del tipus `(n != 0)` és equivalent a `(n)`, que és més breu (però gens recomanable). Per tant, el bucle següent és sintàcticament correcte i calcula el factorial del nombre n:

```
for (fact = 1; n; n = n - 1)
    fact = fact * n;
```

La condició que es comprova és `n`; com hem dit, seria equivalent (i més clar) escriure `n != 0`, i per tant es calcula el producte de `n`, `n-1`, ..., `1`.

Exercici opcional 1.8

Explica què fa el següent codi, és a dir, quant val `m` al final del bucle. Si ho necessites, pots fer un programa on els valors de `a` i `b` es preguntin amb un `scanf`, `m` s'imprimeixi amb `printf`, i provar amb diferents valors. Tingues en compte que `m % b` és el residu de la divisió de `m` per `b`.

```
  for (m = a; m % b; m = m + a);
```

Novament, seria equivalent (i preferible) haver posat la condició com `m % b != 0`.

Valors i assignacions

En una instrucció d'assignació com aquesta:

```
x = 1.0 + cos(0.5);
```

primer *s'avalua* la part dreta del signe `=` (que en aquest cas significa cridar la funció `cos` amb el paràmetre `0.5` i el valor retornat sumar-lo a `1.0`) i a continuació s'assigna el resultat de l'expressió a la variable de la part esquerra del signe `=`. Ara bé, l'assignació en si mateixa és també una expressió, amb valor igual al valor assignat. Això es pot usar tant per fer noves assignacions com comparacions. Per exemple,

```
x = y = 1.0;
```

fa exactament allò esperat. Un exemple més sofisticat seria el següent. Si estem resolent una equació de segon grau, el codi:

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

és equivalent al codi que hem vist a l'exercici 1.5. Fixeu-vos que el `gcc` fa el càlcul del segon valor dins el `printf` *abans* que el del primer, i per això en calcular el primer, les variables `disc` i `a` ja tenen el valor actualitzat. Aquesta forma de programar no és gens recomanable, ja que els programes resultants són difícils de llegir i de mantenir, i poden tenir problemes

de portabilitat. Per exemple, el fet que el segon valor es calculi abans del primer en aquest codi *no està establert* en l'estàndard del C, i un altre compilador podria donar un resultat diferent.
Entendre com funciona aquesta sintaxi ens pot ser útil, més que per utilitzar-la (que no és recomanable) per entendre i evitar certs errors, com el de la instrucció

```
if(a = 1) b = 1;
```

Com que el valor d'una assignació és igual al valor assignat, i en una condició qualsevol cosa diferent de zero s'interpreta com a cert, la sintaxi és acceptada i el que passarà és:

- S'assignarà el valor 1 a la variable `a`.
- La instrucció `if` interpretarà aquest valor, que és diferent de zero, com a cert, i per tant executarà la instruccó `b = 1`. (En canvi, si la condició-assignació fos `a = 0`, no s'executaria el `b = 1`.)

Exercici opcional 1.9

Estudia el bucle següent, que correspon a un "joc" on l'usuari ha d'escriure els nombres de la forma n^2 i el joc continua mentre les respostes siguin correctes:

```
   n=1;
   do{
      n++;
      printf("Quant val %u^2? ", n);
      scanf("%u", &q);
   }
   while((q == n * n) && (n < max));
```

Escriu un programa que contingui aquestes línies, i prova'l. A continuació, modifica la línia on es comprova la condició per:

```
   while((q = n * n) && (n < max));
```

Comprova que el compilador no hi posa cap pega, i observa què fa el programa ara. Explica-ho.

Els operadors &, &&, | i ||

Un altre error relativament freqüent és confondre els operadors `&` i `|` amb `&&` i `||`. Quan avaluem condicions, l'operador `&&` és la conjunció lògica *i* i l'operador `||` és la disjunció lògica *o*. Per tant, la condició de la instrucció següent:

```
if ((a>5) || ((a==3) && (b>0)))
```

serà certa si $a > 5$ o bé $a = 3$ i $b > 0$. En canvi, `&` i `|` són operadors de manipulació de bits. El valor de `a & b` és aquell que, en la representació binària, té iguals a 1 els bits que tant `a` com `b` tenen iguals a 1. El valor de `a | b` és aquell que, en la representació binària, té iguals a 1 els bits que `a` o `b` (o tots dos) tenen iguals a 1. Així, per exemple, com que la representació binària de 7 és 00000111, i la de 10 és 00001010, la de `7 & 10` és 00000010, és a dir `7 & 10` val 2. De manera similar, `7 | 10` val 15.
L'ús de `&` i `|` en la condició d'un `if` o un bucle és molt sovint un error.

2 Programació estructurada

S'anomena *programació estructurada* un conjunt de tècniques de programació que tenen per objectiu millorar la claredat, qualitat i temps de desevolupament dels programes, mitjançant estructures simples organitzades de forma jeràrquica per controlar el flux d'execució del programa. Concretament, en aquest paradigma les estructures usades són, exclusivament, les construccions *alternativa* (`if`/`then`/`else` o `switch`) *iterativa* (`for`/`do`/`while`) i les *funcions*. El llenguatge C està concebut expressament per a la programació estructurada; en la secció anterior vam veure com s'hi implementen les construccions alternativa i iterativa, ara ens centrarem en les *funcions*.

2.1 Funcions

Si cal fer el mateix procediment a llocs diferents d'un programa, no és necessari ni convenient repetir el codi: el podem escriure una única vegada en una *funció* i utilitzar als llocs que calgui *cridant* la funció. Els avantatges de fer-ho així són múltiples:

- 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. En el disseny "de dalt a baix" de programes, el programador es concentra primer en l'algorisme global del problema, assumint que les funcions compliran el seu paper, i és després que escriu el codi de cada funció.
- Cada funció es pot provar i consolidar per separat. En cas d'error podem descartar que es trobi en les funcions consolidades, i localitzar-lo més fàcilment.
- Reutilització: funcions ja provades i consolidades poden resultar d'utilitat en programes futurs. Les millors funcions serveixen en multitud de contextos i no només per a un programa que les hagi pogut necessitar; aquestes es poden col·leccionar en biblioteques de funcions.

Tota funció de C té la següent estructura:

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

// Instruccions de la funció

   return (expressio);
}
```

Cada funció retorna 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 2.1

Considerem la funció (matemàtica) $f : \mathbb{R} \to \mathbb{R}$ definida per

$$f(x) = \frac{\sin x + 10}{x^2 + 1} \, .$$

La funció (de C) següent calcula, donat un valor real x, el valor $f(x)$, i el retorna com a `double`:

```
double f(double x)
{
   return (sin(x) + 10.0) / (x * x + 1.0);
}
```

En qualsevol lloc del programa podem *cridar* la funció `f` i realitzarà aquest càlcul, per exemple, escrivint

```
    y=f(0);
```

s'assignaria a la variable `y` el valor `10`. Ara escriurem un programa complet on es defineix i utilitza aquesta funció.

```
// Programa avaluafuncio.c
// Autor: Professors Dpt Matemàtiques
// Data: 2010-2013
// Avalua una funció i treu el resultat per pantalla

#include <stdio.h>
#include <math.h>

double f(double);

int main()
{
   double x;

   printf("Introdueix el valor de x\n");
   scanf("%lf", &x);

   printf("El valor de f(%lg) és %lg\n", x, f(x));
   return 0;
}

double f(double x)
{
   return (sin(x) + 10.0) / (x * x + 1.0);
}
```

El programa calcula el valor de $f(x)$ en un punt `x` introduït per pantalla. Analitzem els nous elements:

- `double f(double);`

  En aquesta sentència estem informant del *prototipus* de la funció que calcula $f(x)$: estem dient que a aquesta funció cal passar–li un paràmetre `double`, i que retorna un valor `double`. El nombre de paràmetres de cada funció és arbitrari (pot ser zero) però fixat, i els seus tipus també estan fixats. El valor que retorna és únic i també de tipus fixat; aquest pot ser `void`, el que significa que no es retorna cap valor.

  Per afavorir la llegibilitat del programa, la primera funció que escrivim en el fitxer és la funció `main`. Ara bé, perquè aquesta pugui *cridar* les altres funcions, com és `f` en el cas de l'exemple, cal haver-ne declarat el prototipus abans de ser usades. El que fem doncs, és incloure el prototipus de cada funció (excepte `main`) en el preàmbul.

- `f(x)`

  Això és la *crida a la funció* des de dins del programa principal `main()`, i amb la qual li passem com argument el valor de `x`, introduït anteriorment per pantalla.

- `return (sin(x) + 10.0) / (x * x + 1.0);`

  La funció ens retorna el valor `(sin(x) + 10.0) / (x * x + 1.0)`, és a dir, el valor de $f(x)$.

Deseu el programa amb el nom `avaluafuncio.c` i compileu–lo. Comproveu-lo calculant $f(3)$ i $f(-2)$.

Formalment, `main` és una funció que retorna un enter. En els programes d'aquest curs, farem que `main` sempre retorni `0` (aquest valor pren sentit quan uns programes en *criden* uns altres, cosa que no farem).

Exercici 2.1

Un cop comprovat el funcionament del programa anterior, hem *validat* que la funció `f` està correctament programada. Ara podeu elaborar un programa diferent que la faci servir, concretament en aquest exercici heu d'escriure un programa per calcular una taula de la funció $f(x)$ per valors de x entre -30 i 30 en intervals de 0.01, i imprimir-los en pantalla.

Deseu una còpia del fitxer anterior amb el nom `taulafuncio.c`. Modifiqueu el programa introduint un bucle per imprimir la taula. D'acord amb la filosofia de la programació estructurada, el codi hauria de ser el màxim de reutilitzable; amb aquesta finalitat, no hauríeu de fixar els valors extrems en el bucle dins la funció `main`, sinó posar-los en variables `const` en el preàmbul, per poder-los canviar en el futur si cal:

```
const double step = 1.e-2;
const double max = 30.;
```

Amb aquestes instruccions estem definint `step` i `max` com a constants amb valors de 0.01 i 30, respectivament. D'aquesta manera podreu codificar el bucle així:

```
    for(x = -max; x <= max; x = x + step)
        printf("%lg\t %lg\n", x, f(x));
```

Deseu, compileu i executeu el programa. Fixeu-vos en la última línia de la sortida. Quin problema observeu? Aquest problema es pot resoldre substituint les línies anteriors per:

```
const double step = 1.e-2;
const double max = 30.;
const double zero = 1.e-6;
```

```
    for(x = -max; x <= max + zero; x = x + step)
        printf("%lg\t %lg\n", x, f(x));
```

Podeu explicar el que passa?

Observeu que, malgrat que el programa calculi tota la taula de valors demanada, a la finestra on s'executa només hi podem veure les últimes línies. Més endavant veurem com accedir a tota la sortida i treure'n profit.

Exercici 2.2

Farem ara un programa que escrigui a la pantalla les taules de multiplicar del 1 al 10. Primer creeu una funció que es digui `taula` i que tingui per prototipus `void taula (int n);`. Aquesta funció escriurà a la pantalla *una* taula de multiplicar (la del nombre enter `n` que té com a paràmetre), però no retornarà res (per això la declarem `void`).

Seguidament, feu que el programa principal cridi la funció 10 vegades consecutives per $n = 1, 2, \ldots, 10$, per tal que s'escriguin *totes* les taules de multiplicar. Deseu aquest programa amb el nom de `taules.c`.

Exercici 2.3

Feu un programa que donats dos nombres enters m i n, amb $m \geq n$, calculi el nombre combinatori $\binom{m}{n}$. Recordeu que

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!},$$

on $m! = m \cdot (m-1) \cdots 2 \cdot 1$. Caldrà que tingueu en compte les següents consideracions:

- Per definició, $0! = 1$.
- Si $m \leq 2n$, aleshores usem que

  $$\binom{m}{n} = \frac{m \cdot (m-1) \cdot \cdots \cdot (n+1)}{(m-n) \cdot (m-n-1) \cdot \cdots \cdot 2 \cdot 1}$$

  Per exemple, si volem calcular $\binom{6}{4}$ ho faríem així:

  $$\binom{6}{4} = \frac{6!}{4!\ 2!} = \frac{6 \cdot 5}{2 \cdot 1}$$

  i només cal fer 6×5, 2×1 i la divisió.
- Si $m > 2n$, aleshores usem la propietat $\binom{m}{n} = \binom{m}{m-n}$, i calculem aquest últim nombre combinatori com en el cas anterior.

2.2 Variables locals i globals

En el programa de l'exemple 2.1 i l'exercici 2.1, el fet que la variable es digui x al programa principal i a la funció no és important. De fet no són la mateixa variable i ocupen posicions diferents a la memòria. Tant els paràmetres de la funció com les variables declarades dins de la funció són locals de la funció i no es poden 'veure' des de fora; només són conegudes i utilitzables dins la funció. En general les variables declarades dins d'un grup $\{\ldots\}$ són locals del grup. De vegades es recomana, per evitar errors de programació deguts a reutilitzar una variable quan el seu valor ja ha perdut el sentit, que l'índex d'un bucle sigui local del bucle; per exemple, al programa de l'Exemple 1.5 s'hauria pogut posar:

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

En C, els arguments de les funcions sempre 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.

Per poder modificar el valor d'un argument cal usar *apuntadors*, com veurem més endavant.

Exemple 2.2

```
// Programa d'exemple sobre variables locals i globals

#include <stdio.h>

double f(double);
double var1 = 1.0; // Aquesta variable és global de tot el programa

int main()
{
   double x = 2.0;

   printf("Var1: %lf\n", var1); // Correcte: variable coneguda
   printf("Var2: %lf\n", var2); // Error de compilació:
   // la variable 'var2' només es coneix més avall
   printf("Aux: %lf\n", aux);   // Error de compilació:
   // 'aux' només es coneix dins del bloc següent
   printf("z: %lf\n", z);       // Error de compilació:
   // 'z' només 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 = %lf; Funció = %lf; x despres = %lf\n", x, f(x), x
   );
   }
   printf("Aux: %lf\n", aux);   // Error de compilació:
   //'aux' només es coneix dins el bloc anterior
   return 0;
}

double var2 = 2.0; // Aquesta variable només és coneguda a partir d'aqui

double f(double z)
{
   z = var2 / (var1 + z * z);
   // Correcte, aquí coneixem var1 i var2. Redefinim la z.
   return z;
}
```

Recursivitat

Una funció pot ser cridada des de main o des de qualsevol altra funció, en particular una funció es pot cridar a si mateixa. Això s'anomena recursivitat. És una eina potent però que té inconvenients, ja que cada vegada que es crida, la funció crea noves variables locals. Això incrementa els requeriments en temps d'execució i memòria utilitzada. En el pitjor dels casos, molts nivells de recursivitat poden arribar a exhaurir la memòria del sistema. Sempre que sigui possible, **és preferible evitar la recursivitat** i utilitzar la construcció iterativa de bucles en el seu lloc, per exemple, en el càlcul del factorial, o en casos com el de l'exercici següent:

Exercici 2.4

La successió de Fibonacci es defineix de forma recursiva com

$$F_0 = 0, F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, n > 2.$$

Podem escriure un programa recursiu per calcular el terme F_n definint una funció com

```
unsigned int Fibonacci(unsigned int n)
{
    if(n < 2)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Fes, amb aquesta funció, un programa que demani a l'usuari el nombre n i imprimeixi en pantalla F_n. Desa'l amb el nom `Fibonacci_recursiu.c`, compila'l i prova'l amb uns quants valors grans de n.

Ara pensa com fer la mateixa feina evitant la recursivitat. El truc és usar un bucle dins la funció per calcular tots els termes F_k per $k = 1, \ldots, n$, tenint tres variables (per exemple `Fk`, `Fanterior` i `F2anterior`) on hi haurà els valors de F_k, F_{k-1} i F_{k-2} en cada iteració. Observa que un cop calculat F_k, el valor de F_{k-2} ja no serà més necessari, i podem reassignar els valors per trobar el següent terme:

```
        F2anterior = Fanterior;
        Fanterior = Fk;
        Fk = Fanterior + F2anterior;
```

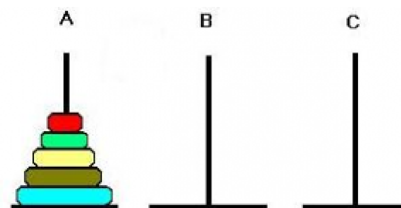
Desa aquest nou programa amb el nom `Fibonacci_iteratiu.c`, compila'l i compara la velocitat d'execució amb l'anterior.

Finalment, cal tenir en compte que a vegades un estudi previ de les matemàtiques involucrades poden simplificar encara més el càlcul. Si saps la fórmula explícita per als nombres de Fibonacci pots crear un programa que (calculant potències de nombres reals) troba el terme n-èsim sense necessitat de calcular els anteriors.

Exemple 2.3

En alguns casos, no obstant, la simplicitat de programació de l'algoritme recursiu el fa adequat, ja que escriure un algoritme no recursiu per la mateixa funció implica una quantitat important de feina.

Farem un programa per resoldre el trencaclosques conegut com *les Torres de Hanoi*. Aquest consisteix en tres varetes verticals i un nombre indeterminat n de discs de mides diferents escalonades que poden inserir-se a les varetes lliscant-hi lliurement.



A l'inici, els discs estan col·locats de més gran a més petit en la primera vareta formant una estructura cònica. El joc consisteix a passar tots els discs a la tercera vareta tenint en compte les regles següents:

- Cada moviment consisteix a agafar el disc superior d'una de les torres i situar-lo a una de les altres torres.
- Cap disc pot trobar-se sobre d'un de més petit.

El joc es pot resoldre de forma recursiva: del que es tracta és en primer lloc de passar la pila de $n-1$ discs superior a la vareta del mig usant la tercera vareta com auxiliar; seguidament passar el disc més gran a la tercera vareta; i finalment passar la pila de $n-1$ discs que teníem al mig a la tercera vareta. El primer i darrer passos són equivalents a un problema de Hanoi amb un disc menys. Vegem com fer-ho en C:

```
// Programa hanoi.c
// Autor: Professors Dpt Matemàtiques
// Data: 2020
// Programa per resoldre recursivament el joc Torres de Hanoi

#include <stdio.h>

int passos = 0;

void torre_hanoi(int n, char origen, char desti, char aux);

int main()
{
   unsigned int n;

   printf("Quants discs té la teva Torre de Hanoi?\n");
   scanf("%u", &n);

   torre_hanoi(n, 'A', 'C', 'B');
   printf("\n\nHe resolt el problema en %d passos.\n", passos);
   return 0;
}

void torre_hanoi(int n, char origen, char desti, char aux)
{
   if(n == 1) {
      passos = passos + 1;
      printf("\nPas %d: mou el disc 1 de %c a %c", passos, origen, desti)
   ;
      return;
   }
   torre_hanoi(n - 1, origen, aux, desti);
   passos = passos + 1;
   printf("\nPas %d: mou el disc %d de %c a %c", passos, n, origen, desti
   );
   torre_hanoi(n - 1, aux, desti, origen);
}
```

- Observem que hem usat variables de tipus `char` per als caràcters `'A'`, `'B'` i `'C'` que designen les tres varetes.
- La funció `torre_hanoi` té quatre paràmetres: el nombre de discs a moure, i els noms de les tres varetes (`origen`, la vareta de la qual hem de treure els discs; `desti`, la vareta on han d'arribar; i `aux`, la vareta auxiliar).
- La variable `passos` és global. Per això es pot accedir al seu valor tant des de la funció `main` com dins de `torre_hanoi`. En canvi, les variables `origen`, `desti` i `aux` són diferents en cada *instància* que es crida la funció.

Deseu el programa anterior en un fitxer anomenat hanoi.c. Compileu-lo, i executeu-lo amb uns quants valors petits de n, fins estar segurs que n'enteneu el funcionament.

Podeu dir quants cops s'ha cridat la funció torre_hanoi (i quantes variables origen existeixen) en el moment d'imprimir-se en pantalla el Pas 1?

Exercici 2.5

En l'exemple anterior s'ha utilitzat una variable global, passos, per comptar el nombre de passos realitzats. Modifiqueu el programa per evitar l'ús de la variable global.
Indicació: Si volem que totes les variables siguin locals, la única manera de passar informació entre funcions és mitjançant els paràmetres (la funció rep dades) i el valor de retorn (la informació dona dades). Una manera de fer-ho en aquest cas és que la funció torre_hanoi tingui un paràmetre més, enter, en el qual se li diu el nombre de passos prèviament realitzats, i retorni un valor enter igual al nombre de passos total després d'executar-se; en tot cas necessitareu declarar una variable passos local dins la funció main.

Treballant a la consola

Tots els sistemes operatius incorporen un programa *intèrpret d'ordres*, també anomenat *emulador de terminal* o *consola*, que té la capacitat de comunicar ordres dels usuaris (introduïdes per teclat) al sistema operatiu i al conjunt de programes instal·lats en un ordinador, i presentar en forma de text en pantalla les respostes del sistema operatiu i altres programes. És una forma de treball interactiva, en què usuari i màquina es comuniquen en forma successiva.
En el món dels ordinadors personals hi ha actualment dos tipus d'intèrprets majoritaris: d'una banda, els basats en l'estàndard POSIX, com les diverses formes de Unix, MacOS, o fins i tot Android, i de l'altra, els produïts per Microsoft, basats en l'antic MS-DOS i distribuïts en tots els sistemes Windows.

POSIX (Linux, MacOS i UNIX)

En aquesta secció suposarem que el sistema utilitzat és un Linux com els que hi ha disponibles als ordinadors de la facultat, però la sintaxi utilitzada igualment a altres formes de Unix, MacOS, i Android.
Si obriu una consola[a] us apareixerà alguna cosa com

```
usuari01@ordinador01:~$
```

D'això se'n diu "prompt", i significa que el sistema està esperant que li entreu una instrucció. Si escriviu alguna cosa i premeu l'Intro, la primera paraula del que hagueu escrit s'interpretarà com el nom d'un programa a executar, i la resta s'interpretarà com arguments, que es passaran al programa en qüestió. Així, si escriviu ls s'executarà el programa que mostra la llista dels fitxers del directori de treball (probablement el vostre directori home), i si escriviu ls -l s'executa el programa amb l'opció -l, i, per tant, a més dels noms dels fitxers del directori de treball, veureu informació addicional com els seus permisos, mida o data de darrera modificació.

Navegar per l'arbre de directoris

En POSIX no hi ha "lletres d'unitat" (a:,c:,...) com al Windows, sinó que hi ha un únic *arbre* de directoris (o carpetes). El directori arrel es representa per /, i aquest mateix caràcter fa

de separador de directoris. Així, `/bin` és el subdirectori `bin` del directori arrel, i `/bin/ls` és el fitxer `ls` del directori `/bin`.

Exercici 2.6

Per a canviar de directori s'utilitza la instrucció `cd` i per a veure el contingut d'un directori la instrucció `ls`.

(a) Obriu una consola. Executeu `pwd` ("print working directory") per veure a quin directori esteu.

(b) Aneu al directori `Documents`, mitjançant la instrucció `cd Documents`. Creeu un altre directori, que es digui `altredir`, amb la instrucció `mkdir altredir` i un fitxer nou, amb `touch altrefitxer.txt`. Feu `ls -l` per veure que efectivament heu creat el fitxer i el directori (observeu que la línia corresponent a `altredir` comença amb una `d`).

(c) Obriu el directori `Documents` en una finestra del navegador d'arxius (`dolphin` en els Linux de la facultat), i comproveu que efectivament s'han creat el directori i el fitxer (si cal s'actualitza prement `F5`).

(d) Esborreu tant el fitxer com la carpeta, mitjançant les intruccions `rm altrefitxer.txt` i `rmdir altredir`.

Tanqueu la consola que teniu oberta i obriu-ne una altra. Executeu, successivament:

```
pwd
cd .
pwd
cd Documents
pwd
cd ..
pwd
cd Documents/..
pwd
cd /usr/share/codeblocks ; pwd
cd /usr/share/codeblocks/../.. ; pwd
cd ~ ; pwd
cd /home ; pwd
cd ; pwd
cd /usr; pwd
cd $HOME ; pwd
cd -; pwd
```

Notes:

- La segona instrucció acaba en "`.`".
- El "fitxer" **.** (punt) denota el directori actual, sigui quin sigui, i el "fitxer" **..** denota el directori del nivell immediatament superior.
- El caràcter $\sim$ equival sempre a *Home* i s'obté prement simultàniament les tecles `Alt Gr` i `Ñ`).

- Podeu obrir una consola al directori que esteu veient amb el `dolphin` mitjançant la tecla `F4`.

Exercici 2.7

(a) Què passa quan es posen dues (o més) instruccions a la mateixa línia separades per "`;`" com al final del bloc d'instruccions anterior?

(b) Perquè la primera instrucció (`cd .`) i la vuitena (`cd Documents/..`) no fan res?

(c) Trobeu les tres instruccions `cd` del bloc anterior que són equivalents, *independentment* del directori on estigueu situats. Què fan?

Com ja hem vist, per a crear directoris i fitxers en blanc mitjançant la consola s'utilitzen les intruccions `mkdir` i `touch` respectivament. Per a copiar fitxers i directoris s'utilitza la instrucció `cp`. Podeu consultar el seu funcionament escrivint `man cp` o `info cp` a la consola. En tots dos casos, sortiu de l'ajuda prement `q`.

Exercici 2.8

Navegueu amb instruccions `cd` fins al directori on teniu els programes en **C** dels apartats anteriors. Alternativament, obriu el directori en `dolphin` i premeu `F4` per obrir-hi una consola. Creeu-hi un directori nou que es digui `progs`, i copieu el programa `taules.c` dins aquest directori, amb la instrucció `cp taules.c progs`. Entreu a aquest directori. Executeu la instrucció `pwd` per a comprovar que hi sou a dins, i la instrucció `ls` per comprovar que el fitxer s'hi ha copiat correctament.
Torneu al directori que conté tots els programes i executeu la instrucció `cp *.c progs`. Esbrineu què ha passat.

Dues altres instruccions bàsiques són `mv` (moure o canviar el nom) i `rm` (esborrar). Com abans, podeu usar les instruccions `man` o `info` per aprendre'n el funcionament.
Observació important: La instrucció `rm` esborra de manera permanent el que li diguem. No ho envia a cap part de l'ordinador en que es pugui recuperar. A més, segons com estigui configurat no demana confirmació abans d'esborrar.

Compilar i executar des de consola

Podem executar un programa compilat per nosaltres escrivint el seu nom en la consola.

Exercici 2.9

Situeu-vos en el directori on heu compilat els programes anteriors, i executeu la instrucció `./taules`. La part `./` de la instrucció senzillament li diu al programa de la consola que el fitxer executable està en el directori actual (`.`). Si el directori on esteu és `Documents`, podríeu executar-ho igualment fent `~/Documents/taules`; o des del vostre *Home*, fent `Documents/taules`.

Per compilar un programa en C des de la consola, usem el compilador gcc. És important saber que quan compilem un programa des d'un entorn integrat com Code::Blocks, el que fa aquest és exactament executar el compliador com ho farem ara des de la consola.

Exemple 2.4

Entreu al directori progs que heu creat abans, on hi hauria d'haver tots els fitxers font dels programes que heu escrit (amb l'extensió .c). Comproveu amb ls que no hi ha els fitxers executables corresponents. Ara escriviu la instrucció

```
gcc -Wall -o taules taules.c
```

Comprovareu que s'ha creat el fitxer executable corresponent; executeu-lo amb ./taules

- 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).
- -o taules: És el nom que volem donar al programa que es crea.
- taules.c: és el nom del fitxer a compilar.

Exercici 2.10

Compileu (d'un en un) tots els fitxers .c del directori progs. Observareu que teniu problemes amb aquells, com taulafuncio.c, que usen el fitxer de capçalera math.h. (Aquesta dificultat no es dona quan compilem dins el sistema operatiu macOS). Executeu la instrucció

```
gcc -Wall -o taulafuncio taulafuncio.c -lm
```

La última opció -lm li diu al compilador (més precisament, a l'enllaçador o *linker*, -l) que ha d'enllaçar codi de la biblioteca matemàtica. Aquesta opció caldrà incloure-la sempre que necessitem math.h, i cal posar-la sempre en últim lloc.

Microsoft (Windows/MS-DOS)

La consola de Windows té una sintaxi una mica diferent a la de Linux/macOS/UNIX). Cal observar que segons la versió de Windows que tenim instal·lada, podem tenir accés a consoles lleugerament diferents; en aquest document utilitzem la consola que està present en *tots* els sistemes Windows: el programa cmd.exe. És possible que en el teu Windows també hi hagi la PowerShell, que disposa d'algunes instruccions habituals en la consola de Linux que no existeixen en cmd.exe (per exemple, ls/cp/mv, cat o tee).
Veureu, en tot cas, que els conceptes bàsics són els mateixos que en la consola de Linux o MacOS. Si obriu una consola[b] us apareixerà alguna cosa com

```
C:\Users\Nom Usuari>
```

D'això se'n diu "prompt", i significa que el sistema està esperant que li entreu una instrucció. Si escriviu alguna cosa i premeu l'Intro, la primera paraula del que hagueu escrit s'interpretarà com el nom d'un programa a executar, i la resta s'interpretarà com arguments, que es passaran al programa en qüestió. Així, si escriviu dir s'executarà el programa que mostra la llista dels fitxers del directori de treball (probablement el vostre directori home), i si

escriviu `dir /w` s'executa el programa amb l'opció `/w` (de `w`ide) i per tant, el llistat apareixerà amb diversos fitxers en la mateixa línia però sense tanta informació de cada fitxer.

Navegar per l'arbre de directoris

En Windows, cada unitat d'emmagatzematge, ja sigui un disc dur (més precisament, una partició) intern o extern, una clau USB, un CD, etc., té associada una lletra. Típicament `C:` correspon al disc dur principal (intern) on hi ha els arxius del sistema operatiu i els directoris dels usuaris. El caràcter `\` fa de separador de directoris. Així, `C:\Windows` és el subdirectori `Windows` de la unitat `C:`, i `C:\Windows\System.ini` és el fitxer `System.ini` del directori `C:\Windows`.

Exercici 2.11

Per a canviar de directori s'utilitza la instrucció `cd` i per a veure el contingut d'un directori la instrucció `dir`.

(a) Obriu una consola. Executeu `cd` ("current directory") per veure a quin directori esteu. Típicament, el prompt del sistema també ens diu en quin directori estem (com en l'exemple de dalt.)

(b) Aneu al directori `Documents`, mitjançant la instrucció `cd Documents` (ara, `cd`="change directory"). Creeu un altre directori, que es digui `altredir`, amb la instrucció `mkdir altredir` Feu `dir` per veure que efectivament heu creat el directori (observeu que la línia corresponent a `altredir` conté l'expressió `<DIR>`).

(c) Obriu el directori `Documents` en una finestra de l'`Explorador de fitxers` (és possible que en l'Explorador, aquesta carpeta es vegi amb el nom `"Mis Documentos"`), i comproveu que efectivament s'ha creat el directori `altredir`.

(d) Esborreu la carpeta, mitjançant la intrucció `rmdir altredir`. Es pot esborrar una carpeta que té fitxers dins, junt amb el seu contingut, amb l'opció `/s`: `rmdir /s altredir`.

Tanqueu la consola que teniu oberta i obriu-ne una altra. Executeu, successivament:

```
cd
cd .
cd Documents
cd ..
cd Documents\..
cd C:\Windows\Temp
cd C:\Windows\Temp\..\..
cd %userprofile%
cd C:\Users
cd C:\Users\%username%
```

Notes:

- La segona instrucció acaba en "`.`".
- El "fitxer" **.** (punt) denota el directori actual, sigui quin sigui, i el "fitxer" **..** denota el directori del nivell immediatament superior.

- `%username%` és una variable d'entorn que té valor igual al nom de l'usuari; `%userprofile%` té valor igual a l'adreça del directori propi de l'usuari.
- En algunes versions de Windows podeu obrir una consola al directori que esteu veient en l'`Explorador d'Arxius` anant al menú `Fitxer`→`Obrir Consola`, on `Consola` pot ser el `Símbolo del Sistema` o el `PowerShell` depenent de la configuració.

Exercici 2.12

(a) Perquè la segona instrucció (`cd .`) i la cinquena (`cd Documents\..`) no fan res?

(b) Trobeu les dues instruccions `cd` del bloc anterior que són equivalents, *independentment* del directori on estigueu situats. Què fan?

Com ja hem vist, per a crear i esborrar directoris mitjançant la consola s'utilitzen les intruccions `mkdir` i `rmdir` respectivament. Per a copiar fitxers i directoris s'utilitzen la instruccions `copy` i `Xcopy`. Podeu consultar el seu funcionament escrivint `help copy` o `help Xcopy` a la consola.

Exercici 2.13

Navegueu amb instruccions `cd` fins al directori on teniu els programes en **C** dels apartats anteriors. Alternativament, obriu el directori a l'`Explorador de Fitxers` i obriu-hi una consola usant el menú. Creeu-hi un directori nou que es digui `progs`, i copieu el programa `taules.c` dins aquest directori, amb la instrucció `copy taules.c progs`. Entreu a aquest directori, i comproveu en el prompt que hi sou a dins. Executeu la instrucció `dir` per comprovar que el fitxer s'hi ha copiat correctament.
Torneu al directori que conté tots els programes i executeu la instrucció `copy *.c progs`. Esbrineu què ha passat.

Dues altres instruccions bàsiques són `move` (moure o canviar el nom) i `del` (esborrar). Com abans, podeu usar la instrucció `help` per aprendre'n el funcionament.
Observació important: Les instruccions `rmdir` i `del` esborren de manera permanent el que li diguem. No ho envien a cap part de l'ordinador d'on es puguin recuperar. A més, segons com estigui configurat el sistema no demanen confirmació abans d'esborrar.

Compilar i executar des de consola

Podem executar un programa compilat per nosaltres escrivint el seu nom en la consola.

Exercici 2.14

Situeu-vos en el directori on heu compilat els programes de les pràctiques, i executeu la instrucció `.\taules`. La part `.\` de la instrucció senzillament li diu al programa de la consola que el fitxer executable està en el directori actual (`.`). Si el directori del fitxer és `Documents`, podríeu executar-ho igualment amb la instrucció `%userprofile%\Documents\taules`; o des del vostre *Home*, fent `Documents\taules`.

Per compilar un programa en C des de la consola, usem el compilador `gcc`. És important saber que quan compilem un programa des d'un entorn integrat com Code::Blocks, el que fa aquest és exactament executar el compliador com ho farem ara des de la consola.

Exemple 2.5

Entreu al directori `progs` que heu creat abans, on hi hauria d'haver tots els fitxers font dels programes que heu escrit (amb l'extensió `.c`). Comproveu amb `dir` que no hi ha els fitxers executables corresponents. Ara escriviu la instrucció

```
gcc -Wall -o taules.exe taules.c
```

Comprovareu que s'ha creat el fitxer executable corresponent; executeu-lo escrivint `taules` (o `.\taules`) a la consola.

- `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).
- `-o taules`: És el nom que volem donar al programa que es crea.
- `taules.c`: és el nom del fitxer a compilar.

El *path* del sistema Si la instrucció anterior per a compilar no et funciona, i en canvi compiles sense problemes des de l'IDE (Code::Blocks, per exemple), possiblement el *path del sistema* del teu ordinador no inclou el directori on està instal·lat el `gcc`. Al document Instal·lació del compilador (final de la secció 2) s'explica com modificar el path del sistema per resoldre aquest problema.
En la consola de Windows, el path del sistema inclou el directori actual (`.`) i per això es pot executar un programa escrivint simplement el seu nom; en canvi, en sistemes POSIX aquesta pràctica es desaconsella per seguretat. Per això en Linux és obligatori escriure `./taules` mentre que en Windows el `.\` és opcional.

[a] En un sistema Kubuntu com els de la facultat, podeu fer doble-clic a la icona "Konsole" de l'escriptori, o bé, a l'esquerra de la barra inferior, cliqueu al logo de KDE [la lletra K], llavors "Sistema", "Konsole".
[b] A la barra del Windows, on hi ha la lupa, escriviu `cmd` o `Símbolo del sistema`.

3 Arrays I: vectors i matrius

Els *vectors*, *matrius* i *cadenes de caràcters* són *tires* de variables que permeten emmagatzemar conjunts de dades amb un nom comú, i índexos per referir-nos a cada dada individual. En C aquesta mena de variables es coneixen amb el nom d'`array`.

3.1 Declaració i ús d'arrays

Les dimensions d'un array en declarar-lo, així com les coordenades de cada component, es denoten amb `[·]`. Les matrius $m \times n$ es poden pensar com vectors de llargada $m \cdot n$ escrits per files. En C els índexs de vectors i matrius sempre comencen per 0. Per tant, si declarem un vector `v` de 5 components, aquests són `v[0]`, `v[1]`, `v[2]`, `v[3]` i `v[4]`.

La declaració i inicialització de vectors i matrius segueix la sintaxi que podeu veure a continuació:

```
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  u[30]  = { 20, 21, 22, 23, 24, 25, 26 };
```

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[3]      // Valor: 1.3
m[1][2]   // Valor: 12
```

Els índexs poden ser variables, constants o expressions el resultat de les quals siguin enters.

Exemple 3.1

```
// Programa matriu.c
// Autor: Professors Dpt Matemàtiques
// Data: 2010-2013
// S'introdueix una matriu i s'imprimeix per pantalla

#include <stdio.h>

#define DIMENSIO 2

int main()
{
   double a[DIMENSIO][DIMENSIO];

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

   puts("Ara imprimim la matriu:");
   for(int i = 0; i < DIMENSIO ; i++) {
```

```
        for(int j = 0; j < DIMENSIO ; j++) {
            printf("%5.2lf\t", a[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

- `#define DIMENSIO 2`
  Podem usar la instrucció de preprocessador `#define` per constants enteres com la dimensió; si més endavant necessitem modificar el programa per usar matrius d'una altra mida només haurem de tocar aquesta línia.

- En la declaració d'una matriu, el nombre dins el primer claudàtor indica la quantitat de files de la matriu, mentre que el nombre dins del segon claudàtor indica la quantitat de columnes. En aquest cas coincideixen.

- Per introduir (`scanf`) o per imprimir (`printf`) arrays hem construït bucles (`for`). No és possible fer-ho amb una sola instrucció (podeu comprovar que `printf("%i\n",a);` no imprimeix pas la matriu, i canviar `%i` per un altre codi de format no ho arreglaria). Igualment, no és possible fer una assignació com `b=a;` si `a` i `b` són de tipus `array`; cal fer un bucle.

- Com que els índexs comencen en 0, però volem que de cara a l'usuari comencin en 1, el `printf` previ a l'`scanf` imprimeix `i+1`, `j+1`.

- La instrucció `puts` imprimeix un text per la sortida estàndard (pantalla). A diferència de `printf`, no admet codis de format per imprimir valors de variables.

Exercici 3.1

En aquest exercici escriurem un programa per calcular el determinant d'una matriu 2×2. Deseu una còpia del programa de l'exemple anterior amb el nom de `det2.c`, i afegiu en el lloc oportú, la línia següent:

```
    printf("El determinant és \t %lg \n", a[0][0]*a[1][1] - a[0][1]*a
    [1][0]);
```

Compileu-lo i executeu-lo per calcular el determinant de la matriu $A = \begin{pmatrix} 1 & -2 \\ 3 & -1 \end{pmatrix}$.

Exercici 3.2

Feu un programa que llegeixi dues matrius de dimensions 2×3 i escrigui per pantalla la matriu suma de les matrius. Deseu-lo amb el nom `sumamatrius.c`.

Indicació: El format de sortida pot ser com en el programa `matriu.c` de l'exemple 3.1, sense parèntesis al voltant.

Exercici 3.3

Escriviu un programa en C que calculi el determinant d'una matriu 3×3 i, en cas que aquest sigui positiu, el seu logaritme neperià. Les entrades de la matriu cal introduir–les per pantalla. En C, el logaritme neperià d'un nombre de tipus `float`, `double` o `long double` es calcula usant la funció `log`. Compte amb no confondre-la amb el logaritme decimal (funció `log10`) i el logaritme en base 2 (funció `log2`). Aquesta funció està disponible a la biblioteca matemàtica, i per tant cal incloure el fitxer de capçalera `math.h` per usar-la.

El programa ha d'escriure per pantalla el determinant de la matriu i, en cas que sigui positiu, el seu logaritme neperià. Deseu–lo amb el nom de `det3.c`.

Indicació: No es demana usar un algoritme que pugui servir per trobar determinants $n \times n$; en comptes d'això, aprofiteu la regla de Sarrus.

Exercici 3.4

Els *arrays* no només serveixen per emmagatzemar matrius, també poden ser útils si necessitem desar una llista llarga de nombres.

La successió de Hofstadter-Conway dels 10000 dòlars `https://oeis.org/A004001` es defineix de forma recursiva com

$$a_1 = a_2 = 1,$$
$$a_n = a_{a_{n-1}} + a_{n-a_{n-1}}, n > 2.$$

Observa que per estar ben definida la recursió, cal que $a_{n-1} \leq n-1$, ja que altrament $a_{n-a_{n-1}}$ no existiria, i a més necessitem conèixer $a_{a_{n-1}}$ per calcular a_n. Comença per demostrar (a mà) per inducció que $a_n \leq n$ per tot n i que per tant tenim una successió ben definida.

Escriu un programa recursiu per calcular el terme a_n. Hauràs de crear una funció `int hc(int n)` que retorni 1 si $n = 1$ o $n = 2$, i en cas contrari es cridi a si mateixa per calcular el terme a_n com `hc(hc(n-1))+hc(n-hc(n-1))`. Desa aquest programa amb el nom `hc_recursiu.c`, compila'l i prova'l amb uns quants valors grans de n.

Ara pensa com fer la mateixa feina evitant la recursivitat. El truc és declarar dins la funció un vector `int a[n]` on anar posant tots els termes anteriors de la successió. Llavors pots fer un bucle per calcular a_k on $k = 1, \ldots, n$, en què per calcular a_k no crides la funció sinó que consultes els valors calculats. Desa aquest nou programa amb el nom `hc_iteratiu.c`, compila'l i compara la velocitat d'execució amb l'anterior.

3.2 Adreces de variables i arrays dins la memòria

La memòria d'un ordinador es pot concebre com una llarga seqüència de cel·les d'emmagatzematge numerades, on cada una pot contenir un byte d'informació. Els vectors i matrius es guarden en cel·les consecutives a la memòria. Així doncs, les declaracions

```
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};
```

podem visualitzar-les de la forma següent:

	1502	1510	1518			
	v[0] 1.0	v[1] 1.1	v[2] 1.2	v[3] 1.3	v[4] 1.4	...

	7023	7027	7031									
	m[0][0] 00	m[0][1] 01	m[0][2] 02	m[1][0] 10	m[1][1] 11	m[1][2] 12	m[2][0] 20	m[2][1] 21	m[2][2] 22	...	...	

Cada variable `double` ocupa 8 bytes, per tant l'adreça de `v[1]` és 8 bytes posterior a la de `v[0]`, i així successivament. El mateix s'aplica a les components de la matriu `m`, tenint en compte que cada `int` ocupa 4 bytes. Les adreces precises de memòria on es troben les variables no les podem saber a priori (s'estableixen en executar el programa). Els nombres 1502, 7023, usats en la representació, són ficticis. L'operador `&` (*adreça*) serveix per obtenir l'adreça on es troba una variable en el moment d'executar el programa:

Exemple 3.2

```
// Programa adreces.c
// Data: Febrer 2020

#include<stdio.h>

int main()
{
   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};
   for(unsigned int i = 0; i < 5; i++) {
      printf("v[%u]\n", i);
      printf("\t Es troba a l'adreça %p\n", &v[i]);
      printf("\t Ocupa %lu bytes\n", sizeof(v[i]));
      printf("\t El seu valor és %lf\n", v[i]);
   }
   for(unsigned int i = 0; i < 3; i++)
      for(unsigned int j = 0; j < 3; j++) {
         printf("matriu[%u][%u]\n", i, j);
         printf("\t Es troba a l'adreça %p\n", &matriu[i][j]);
         printf("\t Ocupa %lu bytes\n", sizeof(matriu[i][j]));
         printf("\t El seu valor és %i\n", matriu[i][j]);
      }
   printf("El vector v ocupa %lu bytes\n", sizeof(v));
   printf("La matriu matriu ocupa %lu bytes\n", sizeof(matriu));
   return 0;
}
```

- El codi de format més apropiat per una adreça és `%p`, que escriurà l'adreça en hexadecimal (base 16). Podeu usar també `%lli` en el seu lloc, imprimint l'adreça en base 10.
- La funció `sizeof( )` torna el número de bytes d'un tipus de dada. Tant es pot aplicar a un tipus de variable (per exemple, `sizeof(int)`) com a la variable concreta (per exemple, `sizeof(matriu[0][0])`). En el cas dels arrays, torna el nombre total de bytes usats.
- En referenciar un element d'un vector o matriu, el C **no comprova** que no ens passem de la seva dimensió. La instrucció `v[27] = 74;` seria acceptada independentment de la dimensió de `v`; el programa copiaria la constant `74` a la zona de memòria (de tamany `double`) que és 27 llocs més enllà de l'inici del vector, independentment de si aquesta zona de memòria correspon a `v` o no. Similarment, `matriu[11][15]` es refereix a la posició de memòria $11 \times 3 + 15$ llocs més enllà de l'inici del vector (ja que 3 és la longitud de les files de `matriu`).

  Canvieu els extrems dels bucles anteriors, per exemple, posant:

```
      for (unsigned int i=0; i<50; i++)
```

Veureu que el programa es compila i s'executa sense problemes, i ens mostra el contingut de posicions de memòria *que no corresponen al vector* `v` interpretades com a nombres `double`. Aquestes posicions poden correspondre a altres variables del nostre programa, al propi codi del programa, o cel·les de memòria no assignades. Si en comptes de llegir-les hi escrivíssim, això, clarament, podria comprometre l'execució del nostre programa.

És doncs responsabilitat nostra en fer el programa assegurar-nos que no accedim a posicions dels `arrays` fora dels índexs declarats.

Exemple 3.3

En aquest exemple veiem com es pot fer la escriptura d'un vector mitjançant una funció. Copia i desa el programa següent amb el nom `vectorfuncio.c`.

```
// Passar un vector a una funció
// Autors: Professors Dpt Matemàtiques
// Data: 2010-2020

#include <stdio.h>
#define DIMENSIO 2

void imprimeixvector(int, double []);

int main()
{
   double v[DIMENSIO];
   int i;

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

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

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

- Observem que hem passat la mida del vector a la funció com un paràmetre `dim`, quan podríem haver usat la constant global `DIMENSIO`. Ho hem fet així seguint la filosofia d'usar variables locals sempre que és possible, i perquè d'aquesta manera podrem reutilitzar la funció en qualsevol programa que usi vectors (sense dependre que hi hagi una constant global anomenada `DIMENSIO`).

- En el prototipus de la funció, el vector que es passa està declarat com un `array` de longitud indeterminada. Això és suficient perquè el compilador sàpiga que `vect[i]` fa referència al valor `double` situat `i` llocs més enllà de l'inici de `vect`. S'hauria pogut també definir la funció amb el prototipus

  ```
  void imprimeixvector (int dim, double vect[DIMENSIO])
  ```

  però seria totalment equivalent.

- En el cas de passar una matriu `M[files][cols]` com a paràmetre, en canvi, el nombre de columnes s'ha d'incloure necessàriament en el prototipus, per exemple es podria definir

  ```
  void imprimeixmatriu( int files, int columnes, double M[][DIMENSIO])
  ```

  o alternativament

  ```
  void imprimeixmatriu( int files, int columnes, double M[][columnes])
  ```

  Això és perquè el compilador farà correspondre `M[i][j]` a la posició $i \times \text{DIMENSIO} + j$ llocs més enllà de l'inici de la matriu, i per a fer això necessita conèixer la llargada de les files de la matriu (A diferència dels valors `files` i `columnes`, que usarà el nostre codi per saber els límits de la matriu, però no el compilador).

- Una funció pot tenir paràmetres de tipus `array`, però no pot retornar com a valor un `array`.

Exercici 3.5

A diferència de les variables ordinàries, en pasar un vector o matriu com a paràmetre, no se'n fa una còpia com a variable local de la funció sinó que la funció rep *l'adreça* del seu primer element. Com a conseqüència, si modifiquem un `array` passat a la funció com a paràmetre, queda modificat l'`array` de la funció que fa la crida. Podem aprofitar aquesta característica per fer, per exemple, una funció que *llegeix* un vector:

```
void llegeixvector(int dim, double vect[])
{
   int i;
   for(i = 0; i < dim ; i++) {
      printf("v(%d) = ? ", i + 1);
      scanf("%lf", &(vect[i]));
   }
   return;
}
```

Afegiu aquesta funció al programa `vectorfuncio.c`, de manera que main cridi les dues funcions `llegeixvector` i `imprimeixvector` per fer la feina.

Exercici 3.6

Construirem una funció que calculi el producte d'una matriu quadrada per un vector. Començarem fent els càlculs a la funció `main()`, dins un fitxer `matriuxvector.c`:

```c
#include<stdio.h>

#define DIMENSIO 3

void imprimeixvector(int, double []);

int main()
{
   double vect[DIMENSIO] = {1.1, 2.2, 3.3};
   double mat[DIMENSIO][DIMENSIO] = { 1., 1., 0.,
                                      0., 1., 1.,
                                      1., 0., 1.
                                    }; //matriu 3x3
   imprimeixvector(DIMENSIO, vect);
   return 0;
```

(a) Completeu aquest programa amb la definició de la funció `imprimeixvector` de l'exemple 3.3.

(b) Creeu un vector `resul`. Modifiqueu la funció `main` perquè s'hi calculi el producte de `mat` per `vect` i el resultat es guardi en `resul`. Feu-lo utilitzant 2 bucles. En el bucle interior calculeu el producte de la fila `i` de la matriu pel vector `vect`. El resultat es guardarà a `resul[i]`. Amb el bucle exterior feu variar el comptador `i` de 0 fins a 2. Utilitzeu la funció `imprimeixvector` perquè el programa imprimeixi el resultat.

(c) Compileu el programa `matriuxvector.c` i comproveu que funciona correctament.

(d) Ara cal crear la funció `matriuxvector`: els dos bucles que inicialment estaven a la funció `main` aniran dins la nova funció. Volem que aquesta funció serveixi per a matrius de qualsevol dimensió. Recordeu el que hem dit a la pàgina 32 sobre el pas de matrius a funcions.

En el **C** original les dimensions de les matrius havien de quedar fixades en el moment de compilar, i en molta bibliografia existent trobareu aquesta restricció. L'estàndard **C**99 va establir la possibilitat que la dimensió es fixi segons el valor d'una variable, i això ens permetrà modificar el programa perquè pugui treballar efectivament amb matrius de mida arbitrària. Per exemple, podem fer les següents declaracions:

```c
   int dim;

   printf("De quina dimensió són la matriu i el vector? ");
   scanf("%d", &dim);

   double vect[dim];
```

Important: La matriu i el vector s'han de declarar *després* de donar el valor a la variable `dim`. En tot cas, *la dimensió d'un array és fixada*. Encara que posteriorment a la declaració `double vect[dim];` canviés el valor de la variable `dim`, l'espai reservat per al vector no canviaria.

Aquest tipus d'array s'anomena `VLA` (Variable Length Array) i l'ús de matrius VLA com a paràmetres de funcions té una restricció important: en aquest cas és *obligatori* que la dimensió de les files de la matriu es passi com un paràmetre, declarat abans que la matriu, com podria ser

```c
void imprimeixmatriu (int dim, double m[][dim]);
```

Això és pel mateix motiu que, fins i tot si la dimensió de les files és una constant, s'ha d'incloure en el prototipus (com hem explicat abans). Tingueu-ho present en els apartats següents.

(e) Definiu una variable `dim` (local dins `main`) de tipus `int` i inicialitzeu-la amb el valor 3. `dim` serà la dimensió de les matrius i vectors del programa. Modifiqueu el programa (declaracions i crides a funcions) per tal que els càlculs es facin en funció de `dim`.

(f) Modifiqueu el programa per tal que demani (`scanf`) un valor per a `dim`, de manera que pugui funcionar amb matrius de dimensió arbitrària. Podeu reutilitzar la funció `llegeixvector` d'un exercici anterior per fer que demani els coeficients de `mat` i `vect`, els quals ara seran de dimensió `dim`×`dim` i `dim` respectivament. El programa imprimirà per pantalla el resultat de calcular `mat` per `vect`. Deseu el programa amb el nom de `matriuxvector.c`.

Exercici 3.7

Escriviu un programa en C que demani 2 vectors i calculi el seu producte escalar. Recordeu que el producte escalar de vectors es defineix com

$$\begin{pmatrix} u_1 & u_2 & \dots & u_n \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = u_1 \cdot v_1 + u_2 \cdot v_2 + \cdots + u_n \cdot v_n$$

Errors i warnings

Abans de produir la versió definitiva d'un programa ens hem d'assegurar que estigui lliure d'errors. En el moment de la compilació, el compilador ens pot avisar d'alguns errors de sintaxi, confusió de tipus, etc. Els errors que pot detectar el compilador s'anomenen errors de compilació, i és important aprendre a identificar-los. Habitualment no és difícil localitzar-los en el codi, ja que el propi compilador ens indica la línia on es troba l'error (tot i que de vegades, l'error és a la línia anterior a aquella on el compilador detecta el problema, i una única equivocació pot generar diversos missatges d'error).
El compilador ens pot donar dos tipus de missatges: els errors i els advertiments (warnings). Els advertiments (warnings) son problemes lleus o alertes sobre possibles problemes. Encara que el compilador acabarà creant el programa executable és important corregir aquests errors que originen els advertiments, perquè segur que seran l'origen de futurs problemes més greus.
Segons quines opcions de compilació apliquem, el compilador analitzarà de manera més o menys exhaustiva el codi, i ens farà més o menys warnings. Com que els warnings habitualment corresponen a errors de programació, és recomanable compilar sempre amb l'opció `-Wall` (`W`arnings=all=tots). Compilant des de la consola:

```
gcc -Wall -o nomprograma nomprograma.c
```

Compilant dins d'un IDE, normalment l'opció `-Wall` s'estableix en la configuració; per exemple, en **Code::Blocks** en el menú `Settings→Compiler Settings→Global Compiler Settings`, de la pestanya `Compiler Flags`, s'ha d'activar l'opció "Enable all common compiler warnings".
Per assegurar-nos que el nostre codi s'ajusta a l'estàndard oficial del C (útil per assegurar la *portabilitat*, és a dir, que el programa es compilarà correctament en qualsevol compilador de C) pot ser útil activar també l'opció `-pedantic`, que genera alguns warnings addicionals.

Exercici 3.8

Copieu i compileu el codi següent.

```
// Programa warnings.c
// Autor: Professors Dpt Matemàtiques
```

```
// Data: Març 2020
// Programa que suma dos nombres, il.lustració dels warnings de
   compilador

int main()
{
   double a, b, c;
   printf("Introdueix un nombre: ");
   scanf("%d", &a);
   printf("Introdueix un altre nombre: ");
   scanf("%d", &b);
   c = sumados(a, b);
   printf("La suma de %d i %d és %d.\n", a, b, c);
   return 0;
}

int sumados(int m, int n)
{
   return m + n;
}
```

El compilador produeix un programa executable, però ens mostra 10 warnings. Executant el programa veiem que la suma no s'executa correctament (en una prova en Linux, compilat amb gcc, demanant la suma de 1 i 2 s'ha obtingut la sortida `La suma de 1 i 745772880 és 16`.)

(a) Abans de seguir llegint, intenteu eliminar tots els warnings, fent els canvis adients al programa. Llegiu bé els warnings, fixant-vos en les línies del programa que els generen.

Els warnings que genera el programa anterior són de tres tipus:

- `implicit declaration of function 'printf'`
  Aquest warning ens avisa que hem oblidat donar el prototipus de la funció de què es tracti; tant en el cas de `printf` o `scanf` com en el de `sumados`. La conseqüència és que el compilador *assumeix* en aquest punt quin hauria de ser el prototipus, en funció dels paràmetres que se li passen i de com s'utilitza el seu valor de retorn. Això pot deixar indetectats altres errors.

- `format '%d' expects argument of type 'int *', but argument 2 has type 'double *'`
  Aquest warning ens avisa que hem usat el codi de format `%d` en un `scanf` que ha d'assignar valor a una variable `double`. En aquest cas ens hem equivocat perquè volíem posar `%lf`.

  La conseqüència d'aquest error és que el valor que entrem per pantalla no s'assignarà correctament a la variable `a` o `b`.

- `format '%d' expects argument of type 'int', but argument 2 has type 'double'`
  És un warning similar a l'anterior, però en aquest cas referit al `printf`. Els valors no s'interpretaran correctament a l'imprimir.

Així doncs, si canviem tots els `%d` per `%lf` i posem al preàmbul del programa les línies

```
#include<stdio.h>
int sumados(int m, int n);
```

(la línia `#include<stdio.h>` és per tenir els prototipus de `printf` i `scanf`) ara la compilació no produeix cap warning. No obstant, l'execució segueix sent problemàtica. Si introduïm els nombres 1.5 i 2.5 la resposta ara és `La suma de 1.500000 i 2.500000 és 3.000000`. Això és degut a què la funció `sumados` pren paràmetres enters, i per tant, quan és cridada amb valors decimals, aquests es trunquen a enters; la suma que ha fet el programa és internament 1+2. El compilador no avisa d'aquestes conversions, ni tan sols amb l'opció `-Wall`. Ara bé, si afegim addicionalment l'opció `-Wconversion`, obtindrem l'avís:

- `conversion to 'int' from 'double' may alter its value`

(b) Canvieu el prototipus i la declaració de la funció `sumados` per tal d'eliminar aquest últim warning i comproveu que ara el programa funciona correctament.

A diferència d'un warning, un error impedeix la construcció de l'executable. El compilador ens informa del tipus d'error i la línia en la qual es produeix l'error. S'ha de tenir en compte que no tots els errors són independents, això vol dir que és probable que un mateix error produeixi més d'un missatge d'error.

Exercici 3.9

Copieu i compileu el codi següent.

```
// Programa errors.c
// Autors: Professors Dpt Matemàtiques
// Data: Març 2020
// Programa que no es compila, que pretén calcular el factorial

#include<stdio.h>

int main()
{
   int n;

   printf("Digues un nombre: ");
   scanf("%d", &n);
   fact = 1
          for(i = 2, i <= n, i++)
             fact = fact * i;
   printf("El factorial de %d és %d.\n", n, fact);

   return 0;
}
```

D'entrada, el compilador troba dos errors:

- `'fact' undeclared (first use in this function)`
  Hem intentat usar una variable sense haver-la declarat prèviament. Si ens hi fixem, ens adonarem que hi ha una altra variable no declarada, l'índex `i` del bucle.

- `expected ';' before 'for'`
  Hem oblidat el `;` després de la instrucció `fact=1`. Aquest missatge d'error típicament mostra el nombre de línia *posterior* a la línia on manca el `;`

Corregiu aquests dos errors i torneu a compilar. Veureu que ara apareixen dos errors i un warning, que abans no hi eren!

- `warning: right-hand operand of comma expression has no effect`
- `error: expected ';' before ')' token`
- `error: expected expression before ')' token`

En realitat tots tres missatges fan referència al mateix error, en la sintaxi del `for`. Trobeu quin és l'error i corregiu-lo.

4 Arrays II: Apuntadors

4.1 Variables apuntador

Així com els valors enters es desen en variables de tipus `int` (i els seus derivats, com `unsigned int`, `long long int`, etcètera) i els valors amb decimals es desen en variables de tipus `double` (i derivats), les *adreces de memòria* es desen en variables de tipus *apuntador*, que es declaren amb el símbol `*`. Tot apuntador va associat a una variable d'un tipus concret, és a dir, un apuntador declarat com `int *` apuntarà a un valor enter; un apuntador declarat com `double *` apuntarà a un valor en punt flotant, etcètera. Per visualitzar un exemple considerem el codi següent:

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

Com ja vam veure en la pràctica anterior, l'operador adreça `&` proporciona l'adreça en memòria de la variable corresponent. La instrucció `int * p = &x;` declara un apuntador a enters anomenat `p`, i li dona com a valor l'adreça de la variable `x`. La situació en memòria serà similar a la següent:

		1502	1506	1510	1518	
	...	**25**	...	***1502***	...	...

- El *contingut* de la variable `x` és: `25`.
- L'*adreça* on comencen els 4 bytes de la variable `x` és: `&x` que dóna com a resultat (posem per cas) `1502`.
- El *contingut* de la variable `p` és: l'adreça `1502`.
- L'*adreça* on comencen els 8 bytes de la variable `p` és: `&p` que dóna com a resultat `1510`.

El símbol `*` també funciona com un *operador invers* de `&`, l'operador 'contingut'.

- El *contingut* `*(&x)` és: `25`.
- El *contingut* `*p` és: `25`.
- És erroni usar `*x`, ja que `x` no és un apuntador, i no conté una adreça.

Exemple 4.1

Vegem, seguint amb l'exemple anterior, què es pot fer i què no amb els operadors `&` i `*`. Creeu un programa `apuntadors.c` que tingui a la funció `main` les línies següents.

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

printf("La variable x és a l'adreça %p i té el valor %i.\n", &x, x);
printf("La variable p és a l'adreça %p i té el valor %p; el contingut
apuntat és %i.\n", &p, p, *p);

*p = 73;
printf("La variable x és a l'adreça %p i té el valor %i.\n", &x, x);

{
    int y = 9;
    p = &y;
```

```
        printf("La variable y és a l'adreça %p i té el valor %i.\n", &y, y)
    ;
        printf("La variable p és a l'adreça %p i té el valor %p; el
    contingut apuntat és %i.\n", &p, p, *p);
        &x = &y;    // Aquesta línia donaria error.
        printf("La variable x segueix estant a l'adreça %p i té el valor %i
    .\n", &x, x);
    }

    printf("La variable y val %i.\n", y); // Aquesta línia donaria error
    printf("La variable p segueix estant a l'adreça %p i té el valor %p;
    el contingut apuntat és %i.\n", &p, p, *p);
    printf("Això és perillós, perquè la variable y ja no està reservada i
    l'adreça no ens pertany.\n");
    p = NULL;
```

- Tal com està, el programa produirà dos errors de compilació. El primer en la instrucció `&x=&y;`. L'adreça `&x` *no* és una variable; és una constant. Per tant no li podem assignar cap valor. L'altre error és en la línia que pretén utilitzar la variable `y` *fora del bloc* `{...}` on està definida. Un cop hàgiu entès el motiu dels errors, esborreu les dues línies del codi, compileu i executeu el programa.

- `*p = 73;` Aquesta instrucció posa el valor 73 a la posició de memòria apuntada per `p`. Com que aquesta és l'adreça de la variable `x`, el valor de `x` canvia. Comproveu-ho mirant el resultat del `printf` que hi ha a continuació.

- Dins el bloc `{...}` on es defineix `y`, s'assigna a `p` l'adreça d'aquesta nova variable. Això no afecta la variable `x`, però ara no es pot usar `p` per accedir a `x`. Un cop el programa surt d'aquest bloc, `y` deixa d'estar disponible. No obstant, l'apuntador `p` segueix contenint l'adreça on s'havia allotjat `y`. Això és perillós. Quan una variable deixa d'estar disponible, cal *desapuntar* qualsevol apuntador que contingués la seva adreça.

- `p = NULL;` Aquesta instrucció desapunta `p`. D'aquesta manera s'evita que una instrucció posterior del tipus `*p = valor;` modifiqui per error una posició de memòria que no ens pertany

Els apuntadors permeten canviar el valor d'una variable cridant a una funció, passant aquesta variable com un argument. Com podem modificar els arguments d'una funció si sempre es passen per valor? La manera d'aconseguir-ho és: en comptes de passar la variable `x` a la funció com a argument, li passem un apuntador `p = &x` que apunti a `x` (`p` és l'argument *no modificable* que passem per valor). Llavors, dins de la funció podem modificar `*p` (no el paràmetre `p`). Notem que, `*p = x` és, de fet, la variable que volem modificar.

Exemple 4.2

Si compileu i executeu el programa veureu que el resultat és 16, és a dir, la funció modifica el valor de la variable.

```
#include<stdio.h>

void eleva_al_quadrat(int *);

int main()
{
```

```
    int a = 4;
    eleva_al_quadrat(&a);
    printf("%i\n", a);
    return 0;
}

void eleva_al_quadrat(int * var)
{
    *var = (*var) * (*var);
}
```

- La crida `eleva_al_quadrat(&a)` li passa a la funció l'adreça de la variable `a`; d'aquesta manera, la funció pot modificar-ne el contingut.
- Dins la funció `eleva_al_quadrat`, el paràmetre `var` conté aquesta adreça. `(*var)` és el valor que hi ha a la variable. Tot i que s'utilitza el mateix símbol `*` per a l'operador contingut i per al producte, cal no confondre'ls. Al davant d'una adreça com `var`, es tracta de l'operador contingut; entre dos nombres, es tracta de l'operador multiplicació. En l'expressió `(*var) * (*var)` només el del mig representa una multiplicació. El resultat de la multiplicació es torna a guardar en la mateixa adreça.

Recordeu que, quan fem servir la fucnió `scanf`, li passem l'*adreça* de la variable que estem llegint, amb l'operador `&`. Per això `scanf` pot escriure el valor en aquesta variable.

Sabem que cada funció té un únic valor de retorn (o cap si és de tipus `void`). En canvi, no hi ha límit al nombre de paràmetres. Així doncs, quan desitgem que una funció "retorni" més d'un valor, podem passar-li a la funció les adreces de les variables on volem que posi els "valors de retorn". Aquestes adreces seran paràmetres de la funció, de tipus apuntador.

Exercici 4.1

Creeu un programa `calculs.c` amb una funció anomenada `calculs` que calculi la suma, resta, producte i quocient de dos nombres `double`, de manera que amb la funció `main` següent apareguin tots els resultats per pantalla.

```
int main ()
{
  double x,y;
  double suma, resta, prod, quoc;

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

  calculs(x, y, &suma, &resta, &prod, &quoc);

  printf("La suma és: %lf\n", suma);
  printf("La resta és: %lf\n", resta);
  printf("El producte és: %lf\n", prod);
  printf("El quocient és: %lf\n", quoc);

  return 0;
}
```

El prototipus de la funció haurà de ser

```
void calculs( double, double, double *, double *, double *, double *);
```

Exercici 4.2

El mètode de Newton per calcular els zeros d'una funció $f(x)$ és un procediment iteratiu que a partir d'un valor inicial x_0 va calculant aproximacions cada cop millors d'alguna solució de $f(x) = 0$. És un mètode que convergeix molt de pressa quan partim d'un punt inicial proper a l'arrel. (Però la convergència no està garantida.)

Si a la iteració k-èsima estem en el valor aproximat x_k, l'aproximació a la iteració següent es calcula així:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Aquesta fórmula es justifica a partir de polinomis de Taylor.

Anem a fer un programa que apliqui el mètode de Newton a calcular l'únic zero de la funció $f(x) = x - \cos x$. Dit d'altra manera, que trobi la solució de l'equació $x = \cos x$. Que la solució és única es pot justificar teòricament o fer-ho evident amb una representació gràfica. El valor inicial el llegirem de la pantalla.

Declarem les variables que usarem:

```
int k = 0;  //Control de les iteracions
double x, fx, dfx, d; //Contindran x, f(x), f'(x), d=f(x)/f'(x)
//de les successives iteracions.
```

Definim una funció de C que, donat un valor x, avaluï $f(x)$ i $f'(x)$. Com que volem que ens "retorni" dos valors, haurem d'usar apuntadors:

```
void aval (double x, double *pfx, double *pdfx)
{
  *pfx = x - cos (x);
  *pdfx = 1. + sin (x);
}
```

En la funció `main()`, escriurem un bloc que s'haurà d'anar repetint fins que l'aproximació sigui prou bona, o fins que hàgim fet un nombre màxim d'iteracions. Per exemple,

```
do {
  aval (x, &fx, &dfx);
  d = fx / dfx;
  x -= d;    // Equival a x = x - d;
  k++;       // Equival a k = k + 1;
}
while ((k <= kmax) && (fabs (d) >= tol)); // Fi del proces iteratiu
```

Observeu que passem a la funció `aval` les adreces de les variables que a la funció estan declarades com apuntadors. Observeu també que s'aniran fent iteracions mentre es compleixi que no hem arribat a un cert nombre màxim donat pel valor de `kmax` i que el valor absolut (calculat per la funció `fabs()`) de `d` sigui més gran que una tolerància màxima `tol`. Declareu aquests valors com a constants globals; per exemple, poseu `const double tol = 1.e-6;`

Ens cal afegir també instruccions per a la lectura del valor inicial per pantalla, per exemple,

```
printf ("Valor de x (llavor inicial): ");
scanf ("%lf", &x);
```

i per a l'escriptura dels resultats:

```
if (k > kmax)
{
  printf ("Problemes de convergència:\n");
  printf ("El mètode no ha convergit en %d iteracions amb tolerancia %lf\
   n", kmax, tol);
}
```

```
else
{
  printf("La solució es: %lf.\nEl mètode ha convergit en %d iteracions
   amb tolerancia %le\n", x, k, fabs (d));
}
```

(a) Poseu tots aquests ingredients junts en un fitxer `newton.c` i compileu i executeu el programa, provant diversos valors inicials. No oblideu incloure el fitxer de capçalera `math.h` per poder usar les funcions trigonomètriques i `fabs()`, i escriure el prototipus de `aval`.

(b) Modifiqueu la sortida per tal d'anar veient el valor de `x` de totes les iteracions.

(c) Modifiqueu el programa per calcular zeros d'alguna altra funció. Per exemple, $f(x) = e^{-x} - \sqrt{x}$.

4.2 Els vectors com apuntadors

Si tenim un apuntador declarat com `tipus * ap;` i `k` és un enter, `ap + k` és un apuntador a `tipus` que apunta a l'adreça de memòria `ap + k*sizeof(tipus)` (és a dir, `k*sizeof(tipus)` bytes més enllà de `ap`).

Quan definim un vector (per exemple `double v[3]`), la variable `v` fa referència a l'adreça de memòria on hi ha la component `v[0]`. Per això quan passem un vector com argument a una funció, en realitat estem passant la posició de memòria on comença el vector; en conseqüència, les modificacions que fem als valors d'un vector a dins d'una funció es conserven quan l'execució torna al programa principal, idènticament al que passa quan el paràmetre de la funció és un apuntador.

De fet, en molts sentits un vector `v` es comporta exactament com un apuntador. Si hem declarat un vector `double v[3]` i un apuntador `double * p` podem assignar `p=v` i llavors:

el vector `v` és com l'apuntador `p`	`v` i `p` són diferents
• L'adreça de `v[i]` es pot designar equivalentment com `v+i`, `&v[i]`, `p+i`, o `&p[i]`.	• La declaració `double v[3];` reserva espai per a 3 variables `double`. La declaració `double * p;` no reserva espai per a cap `double`.
• El valor `v[i]` es pot designar equivalentment com `v[i]`, `*(v+i)`, `p[i]`, o `*(p+i)`.	• `sizeof(v)` retorna $24 = 3 \times 8$ ja que `v` té tres components double, i `sizeof(double)` és 8; en canvi `sizeof(p)` retorna 8, que és el nombre de bytes de qualsevol apuntador.
• En el prototipus d'una funció, un paràmetre vector es pot anomenar `double v[]` o `double * v` i és equivalent.	

Exemple 4.3

L'equivalència entre vectors i apuntadors permet fer molt eficients les manipulacions de vectors. La funció `imprimeixvector` de l'exemple 3.3 es pot programar de la següent manera:

```
void imprimeixvector (int dim, double *vect)
{
  int i;
  for(i=0; i < dim; i++){
    printf("%lf ", *vect);
    vect++;
```

```
    }
  printf("\n");
  return;
}
```

- El valor de `i` serveix per comptar quantes components del vector imprimim, però no és necessari per accedir al vector: l'apuntador `vect` apunta en cada moment a la component que cal imprimir.

Exercici 4.3

Quan cridem la funció `scanf`, podem usar la sintaxi `v+i` en comptes de `&v[i]`. Reescriu la funció `llegeixvector` de l'exercici 3.5 canviant les línies adequades per

```
  scanf("%lf",vect);
  vect++;
```

Comprova que segueixi funcionant correctament.

Exercici 4.4

Estudia el codi següent fins que tinguis clar l'ús de l'apuntador `*p` en la funció `llegeix_matriu`. Després crea una nova funció `imprimeix_matriu` que imprimeixi per pantalla una matriu amb el nombre de files i columnes donat. Completa-ho escrivint una funció `main()` que cridi successivament les funcions `llegeix_matriu` i `imprimeix_matriu`.

```
void llegeix_matriu(int files, int cols, double matriu[files][cols])
{
    double *p;
    p = &matriu[0][0];
    printf("Introdueix les entrades de la matriu\n");
    for(unsigned int i = 1; i <= files; i++)
    {
        for(unsigned int j = 1; j <= cols; j++)
        {
            printf("a(%d,%d) = ", i, j);
            scanf("%lf", p);
            p++;
        }
    }
}
```

Les matrius $n \times m$ són vectors que tenen per components n vectors de m posicions. Un apuntador a un vector sencer de 2 posicions `double` es pot declarar amb

```
double (* apvec) [2];
```

Llavors, si `mat` és una matriu de dues columnes, podem fer `apvec = mat`. Ara `apvec` apunta a la primera fila de `mat`. Més en general, `mat[i]` és el vector sencer igual a la fila i-èsima de `mat`, que podem identificar amb l'apuntador `apvec+i`.

Cal no confondre l'anterior amb

```
double * (vecap [2]);
```

que és un vector de dos apuntadors a double. És a dir, són dos apuntadors a double allotjats en posicions consecutives de memòria però que poden apuntar a variables double totalment independents.

Alguns detalls avançats

Validació de dades

La funció scanf retorna un valor enter, que fins ara no hem usat, però que és recomanable usar per comprovar la validesa de les dades entrades. Concretament, retorna la quantitat d'arguments llegits. Per tant, si per algun motiu scanf falla i no pot llegir el que se li demana, retornarà un 0. Podem aprofitar el fet que el C interpreta un valor enter diferent de zero com a sinònim del valor lògic CERT i l'enter zero com a sinònim de FALS per *validar* que s'ha introduït alguna cosa, amb la sintaxi `if (!scanf(  ))`.
En general, és sempre una bona pràctica comprovar que les dades introduïdes per l'usuari a través d'un scanf són adequades al context. Per exemple, en els exercicis següents, anem a fer una modificació al programa de l'exercici 4.2 per aplicar-lo a calcular zeros de polinomis; òbviament, això només serà d'interès per a polinomis de grau ≥ 2.

Exercici opcional 4.5

Feu un programa polinomi.c que demani a l'usuari els coeficients d'un polinomi i posteriorment imprimeixi per pantalla aquest polinomi.
Necessitareu un vector per guardar els coeficients del polinomi. La dimensió d'aquest vector s'haurà de demanar a l'usuari amb un scanf en la funció main:

```
printf("Entra el grau del polinomi (grau >= 2): ");
if(!scanf("%d", &grau) || grau < 2)
{
   printf("ERROR: grau il.legal!\n");
   return 1;
}
double coef[grau+1];
```

Observeu la expressió `!scanf` dins de l'`if`: equival a preguntar si no s'ha llegit res. Això ho combinem mitjançant la disjunció `||` amb la comprovació de si el grau entrat està fora del rang permès. Si passa alguna de les dues coses, imprimim un missatge d'error i acabem el programa amb el return.
Després d'aquestes comprovacions, declarem el vector (tingueu present que un polinomi $a_0 + a_1x + \cdots + a_dx^d$ té $d+1$ coeficients). Quan demaneu aquests coeficients amb scanf (que aniran a les components `coef[i]` del vector) per la consola, feu que es comprovi de manera similar si s'han llegit correctament. Finalment, és convenient comprovar també que el coeficient de grau més alt no és zero (altrament, aquell no seria el grau del polinomi), és a dir, que el seu valor absolut no és més petit que un cert llindar:

```
if(fabs(coef[grau]) < 1.e-16)
```

La sortida del programa ha de ser de la forma

```
El polinomi que has introduït és: 1.00 +2.15 x -3.00 x^2 +0.71 x^3
```

Indicació: `%+6.2lf`

La "regla de Horner" per avaluar polinomis

El següent ingredient per al programa que volem fer de trobar zeros de polinomis és una funció que avaluï polinomis i les seves derivades.
Es coneix en computació amb el nom de *regla de Horner*[a] l'algoritme per avaluar polinomis que es dedueix de la igualtat

$$P(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_d\alpha^d = a_0 + \alpha\left(a_1 + \alpha\left(a_2 + \cdots + \alpha(a_{n-1} + \alpha a_n)\right)\right).$$

Es calculen de forma recurrent els nombres

$$\begin{aligned} b_{n-1} &= a_n, \\ b_j &= \alpha \cdot b_{j+1} + a_{j+1}, \quad j = n-2, \ldots, -1. \end{aligned}$$

D'aquesta manera el valor final b_{-1} és igual a $P(\alpha)$. (Fixeu-vos que de fet els b_j són els coeficients del polinomi quocient de P per $(x - \alpha)$, segons l'algoritme "de Ruffini", però per avaluar el polinomi no cal guardar-los, i es pot usar una sola variable `b` per posar-hi aquests valors).
Una recurrència similar ens dona la regla de Horner per a les derivades:

$$\begin{aligned} c_{n-1} &= n \cdot a_n, \\ c_j &= \alpha \cdot c_{j+1} + (j+1)a_{j+1}, \quad j = n-2, \ldots, 0. \end{aligned}$$

El resultat c_0 serà l'avaluació en α del polinomi derivada $P'(x)$ (demostreu-ho!).

Exercici opcional 4.6

Feu una còpia del programa anterior, anomenada `polinomi_avalua.c` afegint una funció

```
aval(coef, grau, x, &fx, &dfx)
```

que avaluï $P(x)$ i $P'(x)$ en el valor `x` i els retorni en les variables apuntades per `&fx` i `&dfx`, com en l'exercici 4.2.
Feu també que la funció `main` demani a l'usuari un valor de x i imprimeixi per pantalla els dos valors calculats per `aval`.

Exercici opcional 4.7

Copieu el programa de l'exercici anterior en un nou fitxer `newtonpol.c`, i copieu-hi, dins la funció `main`, la part rellevant del programa de l'exercici 4.2, per tal que trobi un zero del polinomi introduït. El nucli de les iteracions és el mateix, excepte que la crida a la funció `aval` s'ha de modificar lleugerament.
Utilitzeu el programa per a trobar una aproximació del "nombre auri" que és arrel del polinomi $1 + x - x^2$.

Exercici opcional 4.8

En l'exercici 4.4 hem vist com fer que un apuntador vagi indicant tots els coeficients d'una matriu, per files, recorrent la memòria de manera seqüencial. Òbviament podem fer el mateix per anar indicant els coeficients *per columnes*, el que ens permetrà imprimir la matriu transposada, encara que el codi serà una mica més complex.

Completa el programa de l'exercici 4.4 creant una funció `imprimeix_transposada` que (usant apuntadors) imprimeixi per pantalla la matriu transposada de la que hi ha guardada.

[a] El nom fa honor a William George Horner, qui va usar aquesta "regla" el 1819 en un algoritme per trobar zeros de funcions; Horner no havia llegit el treball de Paolo Ruffini de 1804, i l'atribueix a Lagrange (1736-1813) tot i que Newton (1642-1726) ja l'havia feta servir, i de fet es considera que diversos matemàtics xinesos i perses dels segles II-XIV coneixien el mètode.

Depuració d'errors

Els errors de programació realment difícils de localitzar són aquells que porten a tenir un programa sintàcticament correcte, de manera que per al compilador no hi ha cap error, però erroni des del punt de vista de la intenció del programador. És a dir un programa que es pot compilar i executar sense problemes, però que es comporta de manera diferent a la desitjada.

Exemple 4.4

Ara veurem algunes estratègies per depurar programes. Baixeu-vos del Campus Virtual el fitxer anomenat `matriuxvector_erroni.c`. És una versió errònia del programa fet en l'exercici 3.6. Compileu-lo i executeu-lo, per fer el producte de la matriu identitat 2×2 pel vector $(1, 2)$. Veureu que la compilació i l'execució es fan sense problemes, però el resultat és incorrecte. Per què?

Imprimir valors intermedis D'entrada, l'error podria estar en qualsevol lloc, en les funcions que llegeixen el vector o la matriu, en la que fa el producte, o en la que imprimeix el resultat. Si hem anat desenvolupant el programa per parts, com aconsellàvem a la secció sobre programació estructurada, probablement ja sabem que les funcions `llegeixvector`, `llegeixmatriu` i `imprimeixvector` funcionen correctament. Però de fet, en aquest programa els hem modificat lleugerament, així que, qui sap? Una opció raonable seria, doncs, afegir (de manera provisional, per estudiar què passa) la instrucció `imprimeixvector(dim, vect);` just després de `llegeixvector`. Veurem d'aquesta manera que el vector es llegeix i escriu correctament, és a dir, en principi descartarem que l'error estigui ni en la funció `llegeixvector` ni en `llegeixvector`. De manera similar podríem comprovar que `llegeixmatriu` funciona correctament, per exemple imprimint `mat[0]` i `mat[1]` (que són vectors) amb la funció `imprimeixvector`. D'aquesta manera podem arribar la conclusió que l'error es troba en la funció `matriuxvector`, i estudiar el seu codi.
Aquesta estratègia també funciona molt bé quan un programa es "penja", per descobrir en quin punt del programa hi ha el problema.

Debugger És possible que ja hàgiu trobat l'error en la funció `matriuxvector`, però també pot ser que no l'hàgiu vist. Què fem ara? Una cosa que funcionaria seria incloure, dins dels dos bucles de la funció, instruccions per imprimir els valors de la matriu i el vector que s'estan multiplicant. Però anar fent això de forma sistemàtica acaba sent poc pràctic.
Per ajudar a identificar on falla un programa, resulta de gran utilitat una eina que ens permeti executar els nostre programa de forma controlada, pas a pas, veient potser els valors de les variables durant el procés. L'eina que utilitzarem serà el *depurador* (**debugger**) `gdb`. Per poder analitzar un programa mitjançant un depurador, el fitxer executable haurà de contenir certa informació que l'associï al codi font del qual és derivat. Perquè s'inclogui

aquesta informació hem de compilar el programa amb l'opció -g. Obriu una consola, compileu el programa i executeu el depurador amb les instruccions

```
gcc -Wall -g -o matriuxvector matriuxvector_erroni.c
gdb matriuxvector
```

Us apareixeran unes quantes línies amb informació del programa gdb, i un 'prompt' que us dona l'opció d'introduir ordres. Introduïu:

```
break 42
run
```

Això farà que s'executi el nostre programa fins la línia 42, allà on comença la funció `matriuxvector`. Demanarà les dimensions, el vector i la matriu, i s'aturarà; el gdb presentarà per pantalla la línia següent (la 43, el primer `for`). Ara feu

```
step
step
print mat[i][j]
```

La instrucció `step` fa que s'executi una línia de codi i es presenti la següent. La instrucció `print` fa que aparegui per pantalla el valor actual de la variable que s'escrigui a continuació. Es pot fer servir per visualitzar un coeficient d'una matriu o vector, i podeu anar controlant el valor de qualsevol variable com `i`, `j`, etc. En aquest cas serà especialment interessant fer `print mat[0]`, el que imprimeix tot un vector (la primera fila de la matriu). Veieu què ha passat? Si no enteneu per què, repasseu el prototipus a la línia 40.
Algunes de les instruccions del depurador:

- `help` Dona accés al menú d'ajuda.
- `run` Engega l'execució del programa.
- `step` Executa una línia del programa. Si es tracta d'una crida a una funció, només executa la crida i es pararà en la primera línia de la funció.
- `next` Executa una línia del programa. Si aquesta és la crida a una funció l'executa completament.
- `break` Estableix punts de parada (o *ruptura*, **breakpoints**) en l'execució del programa.
- `print` Mostra el contingut d'una variable.
- `list` Mostra el llistat de la funció actual
- `quit` Acaba la sessió de depuració

Depuració dins l'entorn integrat Si heu editat el programa dins un entorn integrat com Code::Blocks, usualment també és possible fer la depuració de manera més visual en el propi entorn. En el cas de Code::Blocks això només és possible creant un "projecte" dins el qual hi hagi el nostre programa.
Des del menú `File`, creeu un projecte nou de tipus "Console application". Doneu-li el nom que vulgueu (per exemple, `matriuxvector.cbp`), escolliu el llenguatge C, i per la resta d'opcions, deixeu les que porta per defecte. Això crea un directori nou amb una col·lecció de fitxers auxiliars que poden ser molt útils per programes grans però que ara mateix no ens interessen especialment.

Un cop estigui creat el projecte, des del menú `Project` obriu el fitxer `matriuxvector_erroni.c` per incloure'l en el projecte, i (si cal) esborreu qualsevol altre fitxer `.c` que en formi part (potser el **Code::Blocks** haurà creat un `main.c` per defecte).
Ara, quan hàgiu compilat el programa, les instruccions del `dbg` es poden aplicar senzillament clicant sobre icones adequades (estan al costat de les icones de compilar i executar) o a través del menú `Debug`. El procés és força intuïtiu. Si voleu veure els valors de les variables, aneu al menú `Debug→Debugging Windows` i activeu `Watches`. Tindreu a la vista permanentment les variables que escolliu, mentre aneu avançant en l'execució del programa. La tria de variables a vigilar (**watch**) és editable.

Exercici 4.9

Executeu pas a pas el programa `hanoi.c` de l'exercici 2.3 amb el depurador, per a un problema amb 4 discs. Comproveu la vostra resposta a la pregunta feta, *quants cops s'ha cridat la funció `torre_hanoi` en el moment d'imprimir-se en pantalla el Pas 1?*

5 Arrays III: Cadenes de caràcters

5.1 Variables que contenen text

A la pràctica 2 (exemple 2.3) vam veure que les lletres (o més en general els caràcters) es poden desar en variables de tipus `char`. Per al text, farem servir *cadenes de caràcters*, que són vectors de tipus `char` (per tant tot el que sabem sobre vectors hi és aplicable) *amb el codi* ASCII 0 al final com a marca de final de cadena (notem que no hi ha cap caràcter que tingui el codi ASCII 0 — en particular el caràcter '0' té el codi ASCII 48). Per tant, una cadena de caràcters *sempre* usa una posició més de les que necessita per a contenir el text hi està destinat (el 0 com a marca de final de cadena). `"Hola"` necessita un vector `char` de 5 o més posicions per a poder ser escrita correctament. Les cadenes de caràcters es poden inicialitzar caràcter a caràcter o bé escrivint el text entre cometes.

Exercici 5.1

Escriviu un programa, anomenat `cadenes.c`, que contingui les següents línies i comproveu-ne el funcionament.

```
char cadena1[21] = { 'B', 'o', 'n', ' ', 'd', 'i', 'a', 0 };
char cadena2[21] = "Bon dia";
char missatge[]  = "Bon dia vida";
printf("%s\n",cadena1);
printf("%s\n",cadena2);
printf("%s\n",missatge);
```

Les cadenes inicialitzades amb un text entre cometes ja porten implícit el caràcter `'\0' = 0` d'acabament i no cal incloure'l explicitament entre les cometes.

Ara afegiu les línies següents al codi i expliqueu què passa:

```
missatge[10]=0;
printf("%s\n",missatge);
```

De fet, `char` és un tipus de variable entera d'un byte (8 bits) de llargada. Per tant, pot contenir enters entre -128 i 127 (o entre 0 i 255 en la versió `unsigned char`). Una variable `char` es pot imprimir en pantalla en format numèric, amb `%i` o `%u`, tot i que estan pensades per a ser impreses com a caràcters amb `%c`. 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.

Exercici 5.2

Escriviu i executeu un programa, anomenat `ascii.c`, que contingui les següents línies, per veure els caràcters imprimibles en la codificació ASCII.

```
 for(char c = 32; c <= 126; c++)
    printf("ASCII %i -> %c\n", c, c);
```

Compareu el resultat amb la taula 1.

Com podeu veure, aquesta codificació no conté caràcters accentuats. L'estàndard ASCII es va crear als Estats Units la dècada del 1960 pensant exclusivament en les necessitats de l'anglès. És una codificació (encoding) de 7 bits (conté 128 caràcters, incloent símbols i caràcters de control). Amb l'objectiu de codificar també caràcters accentuats i altres símbols es van desenvolupar diverses codificacions de 8 bits que contenen a la seva meitat inferior el codi ASCII i a la meitat superior els caràcters addicionals necessaris per a *certs* idiomes; són codificacions "regionals". En

Dec %d	Hex %x	Char %c	Dec %d	Hex %x	Char %c	Dec %d	Hex %x	Char %c
32	20	SPACE	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			

Taula 1: Els caràcters imprimibles del codi ASCII.

particular les codificacions ISO 8859-1/Windows 1252 contenen el conjunt de caràcters anomenat latin-1 que es considera apropiat per a la majoria d'idiomes de l'Europa Occidental. Veieu la taula 2.

A finals de la dècada del 1980 es va posar en marxa el projecte Unicode per unificar les diverses codificacions i permetre emmagatzemar de manera uniforme qualsevol mena d'escriptura que es faci servir. La codificació utf-8 d'Unicode ha anat reemplaçant progressivament les codificacions de 8 bits. En aquesta, els caràcters ASCII es representen amb 1 byte, però els que contenen signes diacrítics es representen per 2 o més bytes. En conseqüència, aquests caràcters *no caben* en una variable `char` i per tant *no es poden imprimir amb* `printf("%c",...)` però sí quan formen part de cadenes amb `printf("%s",...)`.

Exercici 5.3

Recordant que una cadena és un vector, feu un programa, amb un bucle `for` que escrigui per pantalla els codis ASCII de cadascun dels components `char` que formen la cadena inicialitzada amb la instrucció següent:

```
unsigned char cadena[30] = "L'endemà és un món nou.";
```

Quins codis representen les lletres accentuades **à**, **é** i **ó**? La sortida dependrà de la codificació que usi l'editor.

Feu dues còpies del programa, anomenades `utf8.c` i `win1252.c`, desant cadascuna en una codificació diferent (utf-8 / Windows 1252). Gairebé qualsevol editor modern us permetrà canviar la codificació o encoding; en el Code::Blocks haureu d'obrir el menú `Settings→Editor→Encoding settings`. Comproveu les diferències en la codificació executant les dues còpies del programa. En particular, fixeu-vos que les cadenes tenen longituds diferents segons la codificació.[1]

Quina seria la longitud mínima d'una cadena que contingui la frase `"La plaça és plena."` en utf-8?

5.2 Operacions en cadenes

Tal com passa amb els vectors en general, el C no permet fer assignacions, comparacions ni cap altra operació directa amb cadenes *completes*. Solament es poden fer caràcter a caràcter.

Exemple 5.1

L'operació de copiar una cadena origen (`co`) a una cadena destí (`cd`) es podria implementar en una funció de la manera següent:

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

[1]Independentment de la codificació en què estigui escrit el programa, en executar-lo veureu els caràcters impresos amb `%s` o `%c` traduïts d'acord amb la codificació que estigui *activa a la consola*. En Windows podeu forçar la consola a usar utf-8 posant la instrucció `system("chcp 65001 > nul");` a l'inici de la funció `main`, i a usar Windows 1252 amb la instrucció `system("chcp 1252 > nul");` (es necessita `#include<stdlib.h>`). En Linux i macOS es pot establir la codificació de la consola a través dels menús (en el cas d'Ubuntu, al menú `Visualitza` de la **Konsole**; en el cas de macOS, al menú `Terminal→Preferències→Perfils` de la **Terminal**, pestanya `Avançat`.

Dec %d	Hex %x	UTF-8 %x%x	Char %c	Dec %d	Hex %x	UTF-8 %x%x	Char %c	Dec %d	Hex %x	UTF-8 %x%x	Char %c
160	A0	C2·A0	NBSP	192	C0	C3·80	À	224	E0	C3·A0	à
161	A1	C2·A1	¡	193	C1	C3·81	Á	225	E1	C3·A1	á
162	A2	C2·A2	¢	194	C2	C3·82	Â	226	E2	C3·A2	â
163	A3	C2·A3	£	195	C3	C3·83	Ã	227	E3	C3·A3	ã
164	A4	C2·A4	¤	196	C4	C3·84	Ä	228	E4	C3·A4	ä
165	A5	C2·A5	¥	197	C5	C3·85	Å	229	E5	C3·A5	å
166	A6	C2·A6	¦	198	C6	C3·86	Æ	230	E6	C3·A6	æ
167	A7	C2·A7	§	199	C7	C3·87	Ç	231	E7	C3·A7	ç
168	A8	C2·A8	¨	200	C8	C3·88	È	232	E8	C3·A8	è
169	A9	C2·A9	©	201	C9	C3·89	É	233	E9	C3·A9	é
170	AA	C2·AA	ª	202	CA	C3·8A	Ê	234	EA	C3·AA	ê
171	AB	C2·AB	«	203	CB	C3·8B	Ë	235	EB	C3·AB	ë
172	AC	C2·AC	¬	204	CC	C3·8C	Ì	236	EC	C3·AC	ì
173	AD	C2·AD	SHY	205	CD	C3·8D	Í	237	ED	C3·AD	í
174	AE	C2·AE	®	206	CE	C3·8E	Î	238	EE	C3·AE	î
175	AF	C2·AF	¯	207	CF	C3·8F	Ï	239	EF	C3·AF	ï
176	B0	C2·B0	°	208	D0	C3·90	Ð	240	F0	C3·B0	ð
177	B1	C2·B1	±	209	D1	C3·91	Ñ	241	F1	C3·B1	ñ
178	B2	C2·B2	²	210	D2	C3·92	Ò	242	F2	C3·B2	ò
179	B3	C2·B3	³	211	D3	C3·93	Ó	243	F3	C3·B3	ó
180	B4	C2·B4	´	212	D4	C3·94	Ô	244	F4	C3·B4	ô
181	B5	C2·B5	µ	213	D5	C3·95	Õ	245	F5	C3·B5	õ
182	B6	C2·B6	¶	214	D6	C3·96	Ö	246	F6	C3·B6	ö
183	B7	C2·B7	·	215	D7	C3·97	×	247	F7	C3·B7	÷
184	B8	C2·B8	¸	216	D8	C3·98	Ø	248	F8	C3·B8	ø
185	B9	C2·B9	¹	217	D9	C3·99	Ù	249	F9	C3·B9	ù
186	BA	C2·BA	º	218	DA	C3·9A	Ú	250	FA	C3·BA	ú
187	BB	C2·BB	»	219	DB	C3·9B	Û	251	FB	C3·BB	û
188	BC	C2·BC	¼	220	DC	C3·9C	Ü	252	FC	C3·BC	ü
189	BD	C2·BD	½	221	DD	C3·9D	Ý	253	FD	C3·BD	ý
190	BE	C2·BE	¾	222	DE	C3·9E	Þ	254	FE	C3·BE	þ
191	BF	C2·BF	¿	223	DF	C3·9F	ß	255	FF	C3·BF	ÿ

Taula 2: Els caràcters del bloc regional `latin-1` (`NBSP` significa *non-breaking space*, és un espai entre paraules que no es poden separar en un final de línia; `SHY` significa *soft hyphen*, i serveix per marcar les separacions entre síl·labes). En les codificacions de 8 bits `Windows 1252`/`ISO 8859-1` aquests caràcters ocupen el rang 160–255, mentre que en `UTF-8` es codifiquen amb dos bytes (i per tant no són imprimibles amb `%c`). El rang 128–159 de les codificacions de 8 bits, no mostrat aquí, consta de codis de control en `ISO 8859-1` però són caràcters imprimibles en `Windows 1252`.

```
}
```

Observeu com s'usa el fet que els vectors són apuntadors i l'aritmètica d'apuntadors. En detall:

- `while(*co) ... co++`: El paràmetre `co` apunta inicialment al primer caràcter de la cadena origen, a copiar. Amb aquest bucle anem incrementant l'apuntador (`co++`), recorrent tots els caràcters de la cadena, mentre siguin diferents del caràcter 0. Quan s'arribi al caràcter 0, que marca el final del text, se surt del bucle.

- El paràmetre `cd` apunta inicialment al primer caràcter de la cadena de destí. La instrucció `*cd = *co` és la que copia el primer caràcter de `co` a `cd`. Dins el bucle, `cd` s'incrementa conjuntament amb (`co++`), copiant un a un tots els caràcters no nuls.

- Finalment, un cop s'ha sortit del bucle, cal posar el caràcter 0 per marcar també el final de la cadena `cd`. Observem que a causa de la instrucció d'increment `cd++` executada després de copiar l'últim caràcter no nul, la variable `cd` en aquest moment apunta *la següent posició* després de l'últim caràcter no nul.

- La funció `strcpy` no declara en cap moment les cadenes `co` i `cd`. Aquestes s'han d'haver declarat a la funció principal que cridi `strcpy`, i en particular és responsabilitat de la funció principal assegurar-se que `cd` és prou llarga perquè hi càpiga la cadena a copiar. La funció `strcpy` no comprova si la cadena de destí té prou components `char` per contenir la cadena origen.

Afortunadament no cal definir aquest tipus d'operacions de baix nivell cada vegada que necessitem usar cadenes de caràcters ja que hi ha tota una col·lecció de funcions pel tractament de cadenes, definides al fitxer de capçalera `string.h`. En els següents exercicis suposarem que el preàmbul conté la instrucció `#include<string.h>`.

Exercici 5.4

La funció `strlen` dona la longitud de la cadena que hi entrem com argument. Més precisament, retorna el nombre de caràcters entre la posició de memòria que li donem com a argument i el final de cadena `'\0'` que troba. Recupereu el fitxer `cadenes.c` de l'exercici 5.1 i feu-ne una còpia amb el nom `cadenes2.c`. Ara podem comprovar el funcionament de `strlen` fent les modificacions següents

(a) Carregueu a la capçalera el fitxer `string.h`.

(b) Imprimiu el valor de la funció `strlen` aplicada a les variables `cadena1`, `cadena2` i `cadena1+2`.

Si tot ha anat bé, veiem que la `cadena1+2` té dos caràcters menys que la `cadena1`. Sabries explicar per què?

(c) Quina és la diferència entre fer `sizeof(cadena)` i `strlen(cadena)`?

Exercici 5.5

Per a copiar cadenes tenim la funció `strcpy` de `string.h`. Com podem veure a l'exemple següent es pot fer servir tant per a copiar un text literal entre cometes a una variable de cadena com per a copiar el contingut d'una variable de cadena a una altra.

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

int main()
{
  char cadena1[10], cadena2[10];
  strcpy( cadena1, "Hola");
  printf( "%s\n", cadena1);
  strcpy( cadena2, cadena1);
  printf( "%s\n", cadena2);
  return 0;
}
```

Guardeu aquest programa amb el nom `copiar.c`, compileu-lo i executeu-lo.

La funció `strncpy` és molt semblant. Accepta un tercer argument que és el nombre màxim de caràcters que volem copiar. En aquest cas no es copia el caràcter de final de cadena; per tant, si és el cas, l'haurem d'afegir manualment.

Per exemple, podem modificar el segon `strcpy` del programa anterior per

```
strncpy(cadena2,cadena1,3); cadena2[3]='\0';
```

Compileu el programa modificat i executeu-lo.

La instrucció `strncpy` té un avantatge important: si coneixem la mida de la cadena de destí (per exemple, perquè l'hem declarat nosaltres) però no la de l'origen (pot provenir d'una funció no escrita per nosaltres), es podria donar el cas que la cadena a copiar no càpiga en el destí. Amb `strcpy`, s'escriuria a les posicions de memòria següents, amb els possibles errors que això pugui ocasionar.

Exercici 5.6

Com ho faríeu per a extreure un tros de cadena d'una cadena fixada? Per exemple, volem treure del cinquè al novè caràcter de la cadena

`Hola a tothom!`

Feu una funció en C anomenada `extreurecadena` que tingui 4 arguments:

```
void extreurecadena( char *cadena, int n, int m, char *result)
```

- L'argument `cadena` ha de ser una cadena que ja estigui definida.
- El segon i tercer arguments són enters `n` i `m` amb `n<m`.
- El quart argument és la cadena on posar els caràcters que hi ha entre les posicions n i m (incloses) del primer argument.

Podeu utilitzar les instruccions `strcpy` o `strncpy`.

Exercici 5.7

La funció `strcat` permet unir dues cadenes de caràcters en una. El que fa és eliminar el caràcter de final de cadena de la primera i copiar el contingut de la segona a continuació. Podem veure el funcionament a l'exemple següent:

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

int main()
{
  char cadena1[25]="Hola",cadena2[]=" a tothom";
  printf("%s!\n",cadena1);
  strcat(cadena1,cadena2);
  printf("%s!\n",cadena1);
  return 0;
}
```

Guardeu aquest programa amb el nom `concatenar.c`, compileu-lo i executeu-lo. Observeu que, com que la segona cadena s'afegeix a la primera, hem d'estar segurs que que hem reservat espai suficient.

Una altra opció és el `strncat`. El funcionament és el mateix que el de `strcat` afegint un tercer argument que és un enter positiu que especifica el nombre màxim de caràcters que volem afegir a la primera cadena.

Substituïu el segon `strcat` del programa `concatenar.c` per la línia de programa:

```
  strncat(cadena1,cadena2,6);
```

Compileu i executeu el programa. Modifiqueu el tercer argument de l' `strncat` per un nombre superior al de caràcters de la `cadena2` i mireu què passa quan torneu a compilar.

Exercici 5.8

La funció `strcmp` permet comparar cadenes. Si les dues cadenes són idèntiques, la funció retornarà el valor 0. Si les dues cadenes són diferents retornarà un nombre negatiu o positiu depenent de si l'ordre en què entrem els arguments és el lexicogràfic (ordre del diccionari) o no. A l'exemple següent podem veure les diferents sortides:

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

int main()
{
   char cadena1[] = "Hola", cadena2[] = "Hole", cadena3[] = "Hola";
   printf("Diferència entre %s i %s: %d\n", cadena1, cadena2, strcmp(
   cadena1,
          cadena2));
   printf("Diferència entre %s i %s: %d\n", cadena2, cadena1, strcmp(
   cadena2,
          cadena1));
   printf("Diferència entre %s i %s: %d\n", cadena1, cadena3, strcmp(
   cadena1,
          cadena3));
   return 0;
}
```

(a) Guardeu aquest programa amb el nom `comparar.c`, compileu-lo i executeu-lo.

(b) També en aquest cas hi ha una funció anàloga anomenada `strncmp`, on hi afegim un tercer argument que és el nombre de caràcters màxim fins on volem comparar. Modifiqueu el programa anterior per a veure que les tres cadenes coincideixen en els tres primers caràcters.

(c) Quan ens hem referit a l'ordre del diccionari, cal precisar que es tracta en realitat de l'ordre dels caràcters en l'encoding. Així, per exemple, els caràcters amb majúscules i minúscules es consideren diferents. Modifiqueu el programa per comparar les cadenes "Hola" i "hola". Això cal tenir-ho present també si les cadenes porten caràcters accentuats (compareu "Adeu" amb "Adéu").

Exercici 5.9

La funció strchr té dos paràmetres: una cadena de caràcters i un caràcter concret. Retorna l'adreça de memòria del primer caràcter de la cadena que coincideix amb el que entrem com a segon argument, si aquest existeix, i NULL altrament. Podem veure el funcionament a l'exemple següent:

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

int main()
{
  char cadena[]="Hola a tothom";
  printf("%s\n",cadena);
  printf("%s\n",strchr(cadena,'a'));
  printf("%s\n",strchr(cadena,' '));
  return 0;
}
```

(a) Guardeu el programa amb el nom caracterscadenes.c, compileu-lo i executeu-lo.

(b) La funció strrchr té els mateixos paràmetres però retorna l'última posició de memòria on apareix el caràcter que busquem. Afegiu una línia al programa caracterscadenes.c que imprimeixi el tros de cadena que hi ha entre la última a i el final de cadena.

(c) Observeu què passa canviant la a per b, un caràcter que no apareix a la cadena:

```
    printf("%s\n",strchr(cadena,'b'));
```

Modifiqueu el codi perquè, en el cas que strchr retorni NULL, s'imprimeixi el missatge "El caràcter no apareix en la cadena".

(d) Definiu una funció anomenada strnchr on hi entreu tres paràmetres:

```
char *strnchr(char *cadena, char c, int n)
```

on cadena és una cadena de caràcters, c és un caràcter i n és un enter positiu. La funció ha de retornar la posició de memòria de la cadena on es troba l'n-èssim caràcter c.

Comproveu el funcionament afegint la instrucció

```
printf("%s\n",strnchr(cadena,'o',2));
```

Alguns detalls avançats

Recordeu que en utf-8 les lletres amb accents no caben en variables `char`, sinó que es codifiquen amb dos o més `char`. Per tant, les funcions `strchr` i `strrchr` no són suficients per localitzar aquest tipus de símbols.

Exercici opcional 5.10

Busqueu informació sobre la funció `strstr` i feu-la servir per localitzar el "caràcter" `ó` a la cadena `"L'endemà és un món nou."`

Conversions entre nombres i cadenes

Exemple 5.2

Havent fet `#include<stdio.h>`, també podem usar la funció `sprintf`, amb una sintaxi molt semblant a `printf`, però amb un primer argument que és un apuntador a una cadena. En comptes d'imprimir en pantalla, `sprintf` col·loca el resultat (incloent-hi les variables convertides en el format que correspongui) a la cadena indicada. El valor de retorn és (com en el cas de `printf`!) la longitud de la cadena que s'ha creat:

```
   i = sprintf(cadena3, "Pi és aproximadament %lg", acos(-1));
   printf("%s (té longitud %d)\n", cadena3, i);
```

Així podem convertir un número (tant enter, com double, ...) en una cadena.

Exercici 5.11

Si en canvi volem convertir una cadena en el seu valor numèric el que podem fer és utilitzar les funcions `atof` (*ascii to float*), `atoi` (*ascii to integer*), que necessiten la inclusió del fitxer de capçalera `stdlib.h`.

La sintaxi és la mateixa per a les dues: com a argument entrem l'apuntador de la cadena (el nom) i assignem el valor que retorna a la variable numèrica. Ho podem veure a l'exemple següent:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char str1[] = "124z3yu87";
  char str2[] = "-3.14159";
  char str3[] = "1e24";
  printf("str1 com a enter: %d\n", atoi(str1));
  printf("str2 com a enter: %d\n", atoi(str2));
  printf("str3 com a enter: %d\n", atoi(str3));
  printf("str1 com a double: %lf\n", atof(str1));
  printf("str2 com a double: %lf\n", atof(str2));
  printf("str3 com a double: %lf\n", atof(str3));
  return 0;
}
```

(a) Guardeu el fitxer amb el nom `ato.c`, compileu-lo i executeu-lo.

(b) Observeu que la funció `atof` dóna com a resultat un `double` (tot i que del nom sembla que ha de retornar un `float`).

Treballant a la consola

Entrada i sortida

Els programes de consola sempre tenen una entrada estàndard (standard input – stdin), que està connectada al teclat, i una sortida estàndard (standard output – stdout), que està connectada a la pantalla.
Hi ha una forma comú de connectar la sortida d'un programa a l'entrada d'un altre, fent servir el connector "pipe": | (canonada). D'aquesta manera es formen unes "pipes" de dades, que passen per l'acció de diferents "motors" (programes). També es pot redirigir la sortida d'un programa a un fitxer, amb el connector >.
Treballarem un parell d'exemples perquè pugueu veure la utilitat dels dos tipus de redirecció. Com que les consoles de tipus POSIX i les de Microsoft tenen sintaxis lleugerament diferents, incloem dues versions d'aquest apartat.

POSIX (Linux i macOS)

Exemple 5.3

El programa taulafuncio que vau fer a l'exercici 2.1 té una sortida molt poc pràctica, ja que el que veiem per pantalla és només la part final (les últimes línies) de la taula de valors. Vegem com treure més profit d'una sortida amb moltes línies.

(a) Entreu al directori on teniu l'executable taulafuncio.

(b) Executeu

```
./taulafuncio | less
```

El programa less permet llegir una sortida amb moltes línies (o un fitxer de text llarg), desplaçant-nos per ell amb les tecles de cursor. Premeu q per sortir.

(c) Executeu

```
./taulafuncio | gnuplot -p -e "plot '-'"
```

El programa gnuplot és una eina molt potent per generar gràfics en dues i tres dimensions des de la consola. En la majoria d'instal·lacions Linux està disponible per defecte, però no és així en MacOS; si voleu generar gràfics de forma senzilla en un Mac a partir dels programes que fem en C, podeu instal·lar el gnuplot al vostre ordinador.[a] Nosaltres només en farem un ús limitat per visualitzar la sortida d'alguns programes. Podeu trobar-ne la documentació online a http://www.gnuplot.info/documentation.html . En aquest cas ha generat un gràfic de punts a partir de la taula creada pel programa taulafuncio. Proveu ara:

```
./taulafuncio | gnuplot -p -e "plot '-' lt rgb 'violet' with
lines"
```

Exercici 5.12

Executeu les següents instruccions i expliqueu què fa cadascuna. Si us cal, recordeu que amb man o info podeu obtenir ajuda sobre les instruccions.

```
./taulafuncio | nl
./taulafuncio | nl | tac
./taulafuncio | tac | nl
./taulafuncio | nl | tac | less
./taulafuncio | tac | nl | less
```

Un altre programa típic que es fa servir amb “pipes”, generalment al final, és el `wc` (de *word count*).

Exercici 5.13

Una altra forma de redirecció és la que deriva la sortida d’un programa cap a un fitxer, fent servir el connector “`>`” (o “`>>`” si volem afegir les dades al final del fitxer).

(a) Escriviu la taula de valors de la funció en un fitxer anomenat `resultat.txt` usant

```
./taulafuncio > resultat.txt
```

(b) Comproveu que s’ha generat un fitxer de text i obriu-lo amb un editor. També el podreu obrir amb un full de càlcul com LibreOffice, i usar els valors per a la finalitat que vulgueu, com fer un gràfic.

(c) Un cop desat el fitxer, el podem tornar a usar dins la consola, per exemple:

```
cat resultat.txt
less resultat.txt
gnuplot -p -e "plot 'resultat.txt'"
```

(d) Per desar la sortida de `gnuplot` en un fitxer, useu `set terminal` i `set output`:

```
./taulafuncio | gnuplot -p -e "set terminal png size 800,400;
set output 'grafic.png'; plot '-' with lines"
```

També es pot derivar el contingut d’un fitxer cap a l’entrada d’un programa (amb *`programa < fitxer`*), encara que és totalment equivalent a `cat` *`fitxer | programa`*. Una altra eina important quan estem fent servir “pipes”, és la “T”, que és un “canal secundari” pel qual podem tenir una còpia de les dades que estan passant per la “pipe”. Com sempre, el nom d’aquesta eina és senzill, no és més que el nom anglès de la T, `tee`. Fent

```
./taulafuncio | tee resultat.txt | gnuplot -p -e "plot '-'"
```

s’envia *amb una sola instrucció* la taula generada per `taulafuncio` al fitxer `resultat.txt` i al `gnuplot`.

Windows

Exemple 5.4

El programa `taulafuncio` que vau fer a l’exercici 2.1 té una sortida molt poc pràctica, ja que el que veiem per pantalla és només la part final (les últimes línies) de la taula de valors. Vegem com treure més profit d’una sortida amb moltes línies.

(a) Entreu al directori on teniu l’executable `taulafuncio`.

(b) Executeu

```
taulafuncio | more
```

El programa more permet llegir una sortida amb moltes línies (o un fitxer de text llarg), veient de manera successiva les seves línies en blocs de la mida de la finestra (el programa less de Linux és una evolució d'aquest que permet desplaçar-nos pel fitxer endavant i endarrere amb les tecles de cursor). Premeu q per sortir.

(c) (Opcional) El programa gnuplot és una eina molt potent per generar gràfics en dues i tres dimensions des de la consola. En la majoria d'instal·lacions Linux està disponible per defecte, però no és així en Windows. Si voleu generar gràfics de forma senzilla a partir dels programes que fem en C, podeu instal·lar el gnuplot al vostre ordinador.[b] Llavors podeu executar

```
taulafuncio | gnuplot -p -e "plot '-'"
```

Podeu trobar la documentació completa del programa a http://www.gnuplot.info/documentation.html. Nosaltres només en farem un ús limitat per visualitzar la sortida d'alguns programes.

En aquest cas ha generat un gràfic de punts a partir de la taula creada pel programa taulafuncio. Proveu ara:

```
taulafuncio | gnuplot -p -e "plot '-' lt rgb 'violet' with
lines"
```

Exercici 5.14

Executeu les següents instruccions i expliqueu què fa cadascuna. Si us cal, recordeu que amb help o info podeu obtenir ajuda sobre les instruccions.

```
taulafuncio | find "28."
taulafuncio | find /c "28."
taulafuncio | find /v "28."
taulafuncio | find /v /c ""
```

Exercici 5.15

Una altra forma de redirecció és la que deriva la sortida d'un programa cap a un fitxer, fent servir el connector ">" (o ">>" si volem afegir les dades al final del fitxer).

(a) Escriviu la taula de valors de la funció en un fitxer anomenat resultat.txt usant

```
taulafuncio > resultat.txt
```

(b) Comproveu des de l'Explorador d'arxius que s'ha generat un fitxer de text i obriu-lo amb un editor. També el podreu obrir amb un full de càlcul com LibreOffice, i usar els valors per a la finalitat que vulgueu, com fer un gràfic.

(c) Un cop desat el fitxer, el podem tornar a usar dins la consola, per exemple:

```
type resultat.txt
more resultat.txt
type resultat.txt | find "28."
```

(d) Si teniu el `gnuplot`, per desar la seva sortida en un fitxer, useu `set terminal` i `set output`:

```
taulafuncio | gnuplot -p -e "set terminal png size 800,400;
set output 'grafic.png'; plot '-' with lines"
```

Observeu que `type` treu per consola (`stdout`) el contingut del fitxer que se li passa com a paràmetre (en aquest cas `resultat.txt`), és a dir, en molts casos és equivalent a la instrucció `cat` de Linux.

[a] Per instal·lar el `gnuplot` en Mac, aneu a `https://sourceforge.net/projects/gnuplot/files/gnuplot/`, obriu la carpeta de la versió que us interessi i baixeu el fitxer comprimit `gpxxx-os2-emx.zip` (on xxx són els nombres de versió).

[b] Per instal·lar el `gnuplot` en Windows, aneu a `https://sourceforge.net/projects/gnuplot/files/gnuplot/`, obriu la carpeta de la versió que us interessi i baixeu l'instal·lador `gpxxx-win64-mingw.exe` (on xxx són els nombres de versió). Després d'executar l'instal·lador és possible que calgui canviar manualment el path tal com s'explica al document Instal·lació del compilador (final de la secció 2) per afegir el directori oportú, possiblement `C:\Program Files\gnuplot\bin\`.

5.3 Paràmetres de la funció main

En els exemples i exercicis anteriors hem vist que es poden escriure "paràmetres" a continuació del nom dels programes executats des de línia de comandes. Per exemple, cridem `gcc` amb el nom d'un fitxer `.c`, o `ls` amb el nom d'un directori, o `gnuplot` amb instruccions per fer un gràfic. Quan executem un programa escrit per nosaltres, aquesta possibilitat també hi és; el que hàgim escrit a la consola ho rep la funció `main` del nostre programa, com a paràmetres. Podem veure la forma de fer-ho a l'exemple següent.

Exemple 5.5

```
// Programa parametres.c
// Autor: Professors dpt Matemàtiques
// Data: 2003-2020

#include <stdio.h>

int main(int argc, char *argv[])
{
   int n;

   printf("Nombre 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;
}
```

- `int main (int argc, char* argv[])` Declarem `main` amb dos paràmetres: `argc` és un enter, i indica el nombre de "paraules" que hem posat a la instrucció, inclòs el nom del programa. En el cas de `ls directori`, el seu valor seria 2. El cas de l'`argv` és més complex, però molt més important. L'`argv` és un *vector de paraules*, o sigui que cada element `argv[0]`, `argv[1]`, ... és una cadena que conté una paraula de la instrucció donada. En el cas anterior, `argv[0]` és `ls`, mentre que `argv[1]` és el nom del directori.
- Els paràmetres no són *mai* nombres, són sempre paraules (si entrem un nombre és una paraula feta de xifres i no una quantitat).

Compileu el programa anterior, i executeu-lo des de la consola amb la instrucció `./parametre un 23 exemple`.

Exercici 5.16

Si necessitem tenir un paràmetre numèric, hem de pensar que inicialment s'interpretarà com una cadena (una de les paraules de `argv`) i que l'haurem de convertir a un nombre mitjançant una funció `atoi` o `atof`, de les que hem vist abans.

Feu un programa anomenat `factorial.c` que donat un enter n que entrem quan cridem el programa escrigui el seu factorial. Comproveu el programa executant-lo des de la consola, amb `./factorial 10`.

6 Assignació dinàmica

La memòria de l'ordinador és gestionada (o administrada) pel sistema operatiu, que proveeix porcions de la memòria als processos (o programes) a mesura que els necessiten, i les allibera a mesura que ja no la necessiten.

Quan s'executa un programa, el sistema operatiu li assigna un bloc de memòria de mida prefixada, la *pila* (o **stack**) del programa. Mentre el programa està corrent, cada vegada que s'executa una funció en la que declarem una variable o un vector (o més), una porció de la pila de la mida necessària és assignada a aquesta variable o vector. Quan acaba el bloc de codi (per exemple la funció) on s'ha declarat la variable, la porció de memòria corresponent queda alliberada. Aquestes porcions de memòria s'allotgen en posicions contigües dins de la pila; les últimes variables que s'han allotjat seran les primeres a ser desallotjades. És un mecanisme senzill i eficient, però amb alguns inconvenients:

- Hi ha un límit a la memòria que podem usar en la pràctica, donat per la mida de la pila, que està molt per sota de la memòria realment present a l'ordinador. Aquest límit dependrà del sistema operatiu i l'ordinador, i pot ser modificable. En Linux, podem saber aquesta mida executant la instrucció `ulimit -s` en la consola (una mida habitual en Linux actual són 8 Mb; suficient per a moltes coses, però no pas sempre).

- El fet que la memòria assignada a les variables es des-assigni sempre en l'ordre invers a l'assignació a vegades és ineficient. Depenent de l'estructura d'un programa, ens pot convenir "alliberar" memòria utilitzada per accions anteriors tot mantenint actives les últimes variables declarades. El funcionament de la pila no ho permet.

Exemple 6.1

Podem comprovar l'existència d'aquest límit mitjançant el programa `vectorsgrans.c` següent:

```
#include<stdio.h>

int main()
{
    long long unsigned int mida = 128;
    for(int i = 0; i < 20; i++) {
        long long int vector[mida];
        vector[0] = 1; // fem "alguna cosa" al vector
        printf("He creat i usat un vector de %Lu components.\n", mida);
        printf("Ocupa %Lu KB.\n", mida * sizeof(long long int) / 1024);
        mida = mida * 2;
    }
    return 0;
}
```

En un Linux on la mida de la pila són 8Mb, el programa *peta* en el moment que li toca declarar un vector d'aquesta mida. Quan *peta*, es perd qualsevol informació o càlcul que s'hagués pogut estar fent en el programa.

A continuació veurem com evitar aquests inconvenients reservant i alliberant la memòria explícitament dins el programa quan faci falta mitjançant les funcions `malloc`, `calloc` i `free`. El que fan les funcions `malloc` i `calloc` és comunicar al sistema operatiu la sol·licitud d'un bloc de memòria, i el sistema operatiu ens pot assignar un bloc en qualsevol espai de memòria del *munt* (o **heap**), el que a priori permet accedir a tota la memòria del sistema que no estigui essent utilitzada per altres programes. En el moment que la memòria assignada per aquest procediment deixa de ser necessària, això es comunica al sistema operatiu mitjançant la funció `free`.

Aquest mecanisme, més complex, s'anomena *assignació dinàmica* de memòria, perquè els blocs de memòria són assignats i des-assignats activament per instruccions del programa. Com veurem, també permet que el programador decideixi què fer en el cas que no hi hagi disponible la quantitat de memòria requerida.

6.1 Les instruccions malloc, calloc i free

Els prototipus de les funcions utilitzades per sol·licitar i alliberar memòria durant l'execució d'un programa es troben al fitxer de capçalera `stdlib.h`. Quan volem reservar memòria usarem `malloc` o `calloc` que, en cas d'èxit, ens indicaran la posició del bloc de memòria assignat mitjançant un *apuntador*.

Exemple 6.2

Podem veure el funcionament de `malloc` al programa següent:

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

int main(void)
{
    int mida = 0, i;
    int * v;

    printf("Entra la mida del vector: ");
    scanf("%d", &mida);
    // Sol.licitem l'espai.
    v = (int *) malloc(mida * sizeof(int));
    if(v == NULL) { // Comprovació (obligatòria)
       printf("No he pogut reservar l'espai\n");
       return 1;
    }
    for(i = 0; i < mida; i++) {
       printf("Entra v[%d] ", i + 1);
       scanf("%d", v + i); // v + i és el mateix que &v[i]
    }
    for(i = 0; i < mida; i++) {
       printf("v[%d]=%d\n", i+1, v[i]);
    }
    // Deixem l'espai lliure
    free(v);
    return 0;
}
```

- Hem usat el fitxer de capçalera `stdlib.h`.
- En lloc de declarar el vector com `v[mida]`, es declara un apuntador.
- La funció `malloc` té el prototipus

```
void * malloc(size_t nombre-de-bytes);
```

  El paràmetre `nombre-de-bytes` de la funció `malloc` estableix la mida del bloc de memòria reservat. El tipus `size_t` és un enter específic per emmagatzemar la mida (en bytes) d'un bloc de memòria; és el tipus retornat per `sizeof`. El valor retornat per la funció `malloc` és un apuntador a tipus `void` que pot ser assignat a qualsevol tipus d'apuntador usant un *cast* o conversió: `(int *)` converteix l'apuntador retornat en un apuntador a enters, que pot

ser assignat a la variable v. Si la instrucció s'executa amb èxit, aquest apuntador apunta a l'inici d'una àrea de memòria de la mida especificada.

- Si no es pot reservar aquesta memòria el valor retornat per malloc és NULL, i per tant en aquest cas no podem utilitzar-la. Cal **obligatòriament** comprovar sempre, després d'un malloc o calloc, si el vector retornat és NULL.
- A partir de llavors es pot tractar com un vector com els definits directament a la declaració de variables.
- Quan ja no ens fa falta, alliberem les posicions de memòria amb la funció free.

La funció calloc funciona de manera molt semblant. Ara bé, en comptes de dir la mida que volem reservar en bytes, primer cal dir-li la quantitat d'unitats i llavors la mida que ocupa cada unitat. Per exemple, al programa anterior podem modificar la línia que conté el malloc per la següent:

```
v = (int *) calloc(mida, sizeof(int));
```

i el resultat seria el mateix. Hi ha una diferència addicional entre malloc i calloc, que és que la segona inicialitza a zero totes les variables que formen el bloc de memòria allotjat.

Exercici 6.1

Modifiqueu el programa vectorsgrans.c de l'exemple 6.1 perquè usi assignació dinàmica de memòria. Caldrà fer els següents canvis:

(a) S'haurà de declarar un apuntador vector en comptes de vector[mida]; pot estar declarat dins del bucle for, o abans d'entrar al bucle.

(b) En el lloc del bucle on el programa original té la declaració del vector, cal fer la crida a malloc:

```
      vector = (long long int *) malloc(mida * sizeof(long long int));
      if(vector == NULL) {
         printf("No hi ha prou memòria per a un vector de %llu
   components.\n", mida);
         return 1;
      }
```

(c) La última instrucció al final del bucle ha de ser free(vector); per alliberar la memòria. En el programa original, la memòria s'allibera automàticament quan se surt del bloc {...} que conté la declaració (en aquest cas, el bucle for) però això no es dona en el cas de memòria assignada amb malloc o calloc.

Exercici 6.2

Feu un programa suma2.c que primer demani quants nombres volem sumar, després demani un a un els valors tipus double que volem sumar i finalment els sumi tots. La sortida per pantalla ha ser similar a

```
Nombres entrats: 1.1 12.1 3.14 33
Suma: 49.34
```

La instrucció realloc

En el cas que la memòria assignada utilitzant malloc o calloc esdevingui 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 reassignació de memòria és:

```
ptr-nou = (tipus *) realloc (ptr-vell, nou-nombre-de-bytes);
```

Assigna un espai de memòria de mida nou-nombre-de-bytes a ptr-nou. Aquest espai conté les mateixes dades que hi havia a la regió apuntada per ptr-vell (truncada al mínim de la mida de ptr-vell i nou-nombre-de-bytes).

Si realloc no és capaç de redimensionar la memòria inicial (és a dir de trobar un bloc de memòria *consecutiva* de mida nou-nombre-de-bytes) al lloc apuntat per ptr-vell, busca un nou espai, copia les dades, i allibera l'apuntador anterior (ptr-vell). Si aquesta assignació fracassa, realloc deixa inalterat ptr-vell i torna el valor NULL, que s'assigna a ptr-nou. Per aquest motiu, cal evitar la sintaxi ~~v = realloc (v, nou-nombre-de-bytes);~~ que pretén assignar directament la nova adreça al mateix apuntador; en el cas que la nova reserva no fos possible, aquesta instrucció faria perdre l'adreça de v, que tot i romandre activa (realloc només desallotja la memòria de ptr-vell en cas de tenir èxit en allotjar ptr-nou) seria ara inaccessible.

Exemple 6.3

Vegem un exemple de redimensionament.

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

int 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
    return 0;
}
```

6.2 Assignació dinàmica de matrius

Per treballar amb matrius la memòria de les quals s'ha assignat amb malloc o calloc hi ha dos procediments equivalents que podem escollir. En primer lloc, podem tractar una matriu $m \times n$ com un vector de $m \cdot n$ components:

Exemple 6.4

Si K és un cos arbitrari, aleshores el conjunt de les matrius $M_{m\times n}(K)$ és un $K-$ espai vectorial de dimensió $m \cdot n$, així doncs podem pensar qualsevol matriu A d'aquest conjunt com un vector de $m \cdot n$ components. En el següent exemple es mostra una manera de treballar amb matrius de nombres reals pensades com a vectors.

```
// Autor: Professors Dpt Matemàtiques
// Data: 2003-2020
// Matriu assignada dinàmicament i accedida amb #define

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

int main(void)
{
   double * a;
   int n, m, i, j;

   printf("m? n?:");
   scanf("%d %d", &m, &n);

   a = (double *) malloc(m * n * sizeof(double));
   if(a == NULL) {
      printf("Ha fallat l'assignació.\n");
      exit(1);
   }

#define A(i,j) a[n * (i) + (j)]
   printf("Ara entra els coeficients de la matriu:\n");
   for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
         scanf("%lf", &A(i, j));
   printf("Imprimim la matriu A:\n");
   for(i = 0; i < m; i++) {
      for(j = 0; j < n; j++)
         printf("A(%d,%d)= %lf\t", i + 1, j + 1, A(i, j));
      printf("\n");
   }
#undef A

   free(a);
   return 0;
}
```

- Com que hem declarat que l'apuntador a és de tipus double *, podem accedir al seu element i-èsim com un vector, a[i], però no té sentit fer-ho com a matriu a[i][j]. Ara bé, la intenció és que aquest vector contingui els coeficients d'una matriu A, en una fila llarga que és la successió de totes les files de A. Per tant, el coeficient $A_{i,j}$ de la matriu es trobarà a la posició $ni + j$ dins el vector.
- La instrucció #define A(i,j) a[n * (i) + (j)] ens evita haver d'escriure a[n * i + j] cada vegada que ens volem referir a l'element (i, j) de la matriu.

- Per seguretat, desactivem la *macro* `A(i,j)` amb `#undef` quan acaba el tros de programa on es fa servir.

Exercici 6.3

Les matrius de Hilbert $H = (h_{ij})$ són matrius quadrades que es defineixen per la següent expressió:

$$h_{ij} = \frac{1}{i+j-1}$$

Escriviu un programa que desi les matrius de Hilbert de dimensió 3^n, per n tan gran com sigui possible, i imprimeixi la seva traça.

La segona manera de pensar les matrius quan fem assignació dinàmica és usant **apuntadors a vectors sencers**, recordant la identificació entre matrius i vectors de vectors. Si un vector de `double` es pot allotjar mitjançant un apuntador a `double`, una matriu que té files de 3 `double` es podrà allotjar mitjançant un apuntador a `double[3]`, declarat com

```
double (*A) [3];
```

Òbviament, 3 es pot substituir per qualsevol valor (el nombre de columnes que tingui la matriu desitjada) també per una variable. L'exemple següent és completament anàleg a 6.4, però usant aquesta tècnica alternativa.

Exemple 6.5

```
// Autor: Professors Dpt Matemàtiques
// Data: Març 2020
// Matriu assignada dinàmicament per apuntador a fila

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

int main(void)
{
   int n, m, i, j;

   printf("m? n?:");
   scanf("%d %d", &m, &n);

   double (*A) [n];
   A = (double (*) [n]) malloc(m * n * sizeof(double));
   if(A == NULL) {
      printf("No hi ha prou espai\n\n");
      return 1;
   }

   printf("Ara entra els coeficients de la matriu:\n");
   for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
         scanf("%lf", &A[i][j]);
   printf("Imprimim la matriu A:\n");
   for(i = 0; i < m; i++) {
      for(j = 0; j < n; j++)
         printf("A(%d,%d) = %lf\t", i + 1, j + 1, A[i][j]);
```

```
        printf("\n");
    }

    free(A);
    return 0;
}
```

- En aquest cas l'apuntador `A` no es pot declarar fins que la variable `n` conté el nombre de columnes de la matriu, ja que el *tipus* de variable a què apunta `A` ha de ser: vectors (fila) de n components. Per això la conversió de l'apuntador aplicada al `malloc` és `(double (*)[n])`.
- L'avantatge és que ara podem usar la sintaxi habitual `A[i][j]` per accedir als components de la matriu, sense necessitat de `#define`.
- Cal tenir present que, un cop declarat l'apuntador `A`, el seu tipus no es pot canviar. Així, si s'apliqués un `realloc` a `A` es podria canviar el nombre de files que té la matriu, però no el nombre de columnes. Igualment, canviar el valor de n no afecta la mida de la matriu, ja que el nombre de columnes serà sempre el valor que tenia la variable `n` en el moment d'executar la declaració `double (*A) [n]`.

Exercici 6.4

Modifiqueu el programa `matriuxvector.c` de l'exercici 3.6 perquè usi assignació dinàmica de memòria per a les matrius.

Exercici 6.5

Modifiqueu el programa anterior creant dues funcions que retornin la suma i el producte de dues matrius.

Alguns detalls avançats

Tipus de dades

Els tipus de dades més comuns són el `char`, `int`, `float` i `double`; a continuació podeu veure totes les seves variants:

Especificació	**bytes** (8 bits)	**Rang**
`unsigned char`	1	0 — 255
`char`	1	−128 — 127
`unsigned short int`	2	0 — 65535
`short int`	2	−32768 — 32767
`unsigned int`	4	0 — 4294967295
`int`	4	−2147483648 — 2147483647
`unsigned long int`	4	0 — 4294967295
`long int`	4	−2147483648 — 2147483647
`unsigned long long int`	8	0 — 18446744073709551615
`long long int`	8	−9223372036854775808 — 9223372036854775807

Especificació	bytes	Rang	Precisió
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	$3.362103 \times 10^{-4932}$ — $1.189731 \times 10^{+4932}$	18 dígits

Podem declarar una variable dient el tipus (`char`, `int`,...) i, si cal, el modificador (`unsigned`, `long`,...).
Si fem operacions amb un tipus de dades, es mantindran en aquest tipus de dades: per exemple, si fem la divisió dels enters 4 dividit per 3 donarà l'enter 1. D'altra banda, si operem enters amb reals, l'operació es farà en els reals. Podem modificar el tipus de dades abans de fer les operacions, o bé forçar que es canviï el tipus fent intervenir un altre tipus entremig:

Exemple 6.6

```
// Programa CoercioDades.c
// Autor: Professors Dpt Matemàtiques UAB
// Data: 2020-2021
// Descripció: Mostra les operacions entre diferents tipus de daddes.

#include<stdio.h>
int main()
{
   int a;
   double x, y, z;
   x = 5 / 3;
   y = (double) 5 / 3;
   z = 5 * 1.0 / 3;
   a = z;
   printf("x=%lf, y=%lf, z=%lf i a=%d\n", x, y, z,a );
   return 0;
}
```

Té la sortida:

```
x=1.000000, y=1.666667, z=1.666667 i a=1
```

on veiem les diferències entre el valors que prenen `x`, `y` i `z`. També veiem que l'assignació `a = z;` ha convertit el nombre real 1.666 a enter, però per truncació, i no per aproximació a l'enter més proper.

Exercici opcional 6.6

La *coerció* o canvi de tipus té precedència sobre altres operadors. Canvia la línia rellevant de l'exemple anterior per `y = (double) (4 / 3);` Explica la diferència de comportament tenint en compte la precedència d'operadors.

Especificadors de camp en printf

El prototipus de la funció `printf` és

```
int printf(const char *format, ...)
```

Els `...` indiquen una llista de variables en un nombre indeterminat.

La cadena de caràcters apuntada per `format` pot contenir text amb *caràcters especials* i *especificadors de camp*, que seran *substituïts* a la pantalla pels valors de les variables de la llista. Si s'especifiquen un nombre diferent de camps a imprimir i variables a la llista `...` els resultats són imprevisibles (en general, veurem caràcters estranys a la pantalla).

Caràcter especial	Valor	Especificador	tipus-dada
`\n`	*newline*	`%d`	enter en base decimal
`\f`	*form feed*	`%i`	enter amb signe (igual a `%d`)
`\b`	*backspace*	`%u`	enter no signat
`\t`	*tab*	`%o`	enter en base vuit (octal)
`\v`	*tab* vertical	`%x`	enter en hexadecimal
`\nnn`	ASCII *nnn* (octal)	`%c`	caràcter
`\xnn`	ASCII *nn* (hexadecimal)	`%s`	cadena de caràcters
`\\`	*el caràcter* `\`	`%f`	real en punt flotant
`%%`	*el caràcter* `%`	`%e`	real en format exponencial
		`%g`	real en format `%e`, `%f` o `%d`

Podeu trobar tots els codis que s'utilitzen en la funció `printf` per especificar el lloc i el format de les variables que es volen imprimir anant a la pàgina `http://www.cplusplus.com/reference/clibrary/cstdio/printf/`. Són *tags* (etiquetes) que comencen per `%` i que tenen la forma general

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

on els `[ ]` assenyalen modificadors opcionals amb el significat següent:

- `amplada` delimita el nombre de caràcters mínim que es reservarà per a la impressió del camp corresponent.

  Exemple: Si el valor a imprimir és `12`, el format `%5d` imprimeix `␣␣␣12`.

- En el cas dels nombres en punt flotant, es pot especificar el nombre de decimals que es desitja imprimir amb `precisió`.

  Exemple: Si el valor a imprimir és `12.43`, el format `%10.5f` imprimeix `␣␣␣12.4300`.

- `-` El signe negatiu indica que el camp s'alinea a l'esquerra.

  Exemple: Si el valor a imprimir és `12`, `%-5d` imprimeix `12␣␣␣`.

- `+` El signe positiu indica que s'imprimeixi el signe, tant si és positiu com negatiu (si el valor és negatiu, el signe s'imprimeix en tot cas).

  Si el valor a imprimir és `12`, el format `%+5d` imprimeix `␣␣+12`.

- `0` indica que el camp s'ha d'omplir amb zeros a l'esquerra.

  Exemple: Si el valor a imprimir és `12`, `%05d` imprimeix `00012`.

- El modificador `L` (o `ll`) indica `long long tipus-dada` en lloc de `tipus-dada`. (En minúscules, el modificador `l` indica `long tipus-dada`, i s'aplica per defecte, ja que els `short int` i els `float` es converteixen a `int = long int` i a `double` respectivament abans d'imprimir-se.)

  Exemple: Cal usar `%Ld` per imprimir variables del tipus `long long int` i `%Lf` per imprimir variables del tipus `long 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`.

7 Fitxers

A la pràctica 5 vam veure com *redirigir* la sortida del nostre programa a un fitxer. Ara bé, la manipulació de fitxers es pot fer de manera molt més flexible si ho preveiem en el propi programa: es poden enviar missatges a la pantalla alhora que resultats a un fitxer, llegir dades d'un altre fitxer, usar diversos fitxers alhora...

7.1 Obrir i tancar fitxers, nanses

La connexió del nostre programa amb fitxers de dades es fa usant "nanses", que són variables del tipus *apuntador a fitxer*: `FILE * nom-de-nansa`. Quan obrim un fitxer des del nostre programa, s'inicialitza una *nansa*, la qual esdevé el nostre canal de comunicació amb el fitxer.

Per *obrir* i *tancar* fitxers usarem les funcions `fopen` i `fclose`, que tenen els prototipus següents:

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

La funció `fopen` crea un canal de comunicació entre un fitxer i el programa en el mode especificat pel *mode d'obertura*. Per exemple, per obrir un fitxer anomenat `dades.txt` en *mode lectura*, podríem fer:

```
FILE * nansa;
nansa = fopen ("dades.txt","r");
```

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

El mode d'obertura és una cadena contenint `r`, `w` o `a`, seguit opcionalment per `b`, seguit opcionalment per `+`.

- `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.
- `b`: El fitxer no és de text, sinó binari, i es llegirà per bytes o blocs de bytes, no com a dades escrites amb caràcters.

Exemple 7.1

Creeu i compileu un fitxer `escriu.c` que contingui el programa següent:

```
// Programa escriu.c
// Autor: Professors Dpt Matemàtiques
// Data: 2003-2020
// Escriu una paraula de la consola a un fitxer
```

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   FILE * meufitxer;
   meufitxer = fopen("rosalia.txt", "w");
   if(meufitxer == NULL) {
      fprintf(stderr, "ERROR: El fitxer 'rosalia.txt' no es pot obrir...\
   n\n");
      return 1;
   }
   fprintf(meufitxer, "%s\n", argv[1]);
   fclose(meufitxer);
   printf("Fet!\n");
   return 0;
}
```

Les novetats d'aquest programa són:

- `FILE* meufitxer;`
  defineix una variable, `meufitxer`, que és una "nansa" (un apuntador) a un fitxer.

- `meufitxer=fopen("rosalia.txt","w");`
  agafa un fitxer concret, que es diu `rosalia.txt` en aquest cas, i fa que `meufitxer` apunti a ell. El caràcter `"w"` és un indicador de **què** volem fer amb el fitxer. En aquest cas significa que volem escriure (write) dins del fitxer.

- `if(meufitxer==NULL)`
  controla que el fitxer s'hagi obert correctament. En cas contrari, la nansa tindrà el valor `NULL` i aturem el programa. Això passarà per exemple si volem escriure en un fitxer o directori on no tenim permis per escriure, o si el disc és ple o, el més habitual, si volem llegir un fitxer que no existeix.

- `fprintf(meufitxer, "%s\n", argv[1]);`
  és el mateix que `printf`, amb la diferència que en lloc d'escriure a pantalla s'escriu en el fitxer que s'especifica. En aquest cas concret, fem servir el format `%s` (string) que serveix per imprimir `argv[1]`, que és una cadena (recordeu els paràmetres de `main` explicats a la pràctica 5).

- `fclose(meufitxer);`
  retorna el control del fitxer al sistema operatiu. Mentre el tenim obert, el sistema no permetrà que cap altre procés el modifiqui.

Executeu per exemple la instrucció `./escriu hola` des de la consola i observeu que el fitxer `rosalia.txt` (que si no hi era s'ha creat ara) conté la paraula "hola".

Exercici 7.1

Modifiqueu el programa per a què utilitzi el primer argument (`argv[1]`) com a nom del fitxer de sortida en lloc del fitxer `rosalia.txt`, i el segon argument (`argv[2]`) com a paraula per escriure al fitxer. Guardeu el codi com `escriu-en.c` i comproveu que funciona.

Exemple 7.2

Per llegir d'un fitxer l'hem d'obrir amb el mode "r" (**r**ead): `meufitxer=fopen("UnNomDeFitxer.txt","r");` i en lloc de `scanf(...);` hem d'usar `fscanf(fitxer,...);`.

Baixeu-vos del Campus Virtual el fitxer `columnes.dat`. Obriu-lo amb un editor. Observeu que conté nombres, repartits en tres columnes. Si volem llegir les dades de tres en tres, ho podem fer dins d'un bucle de la manera següent:

```
// Programa d'exemple d'ús de fscanf
// Autor: Professors Dpt Matemàtiques
// Data: 2003-2020

#include <stdio.h>

int main()
{
   FILE * fitxer;
   double a, b, c;
   fitxer = fopen("columnes.dat", "r");
   if(fitxer == NULL) {
      printf("ERROR: El fitxer 'columnes.dat' no es pot obrir...\n\n");
      return 1;
   }
   while(EOF != fscanf(fitxer, "%lf %lf %lf", &a, &b, &c)) {
      // fes alguna cosa amb les variables a, b i c
   }
   fclose(fitxer);
   return 0;
}
```

Quant a la condició del `while`:

- `fscanf()` retorna el nombre de variables que s'han llegit.
- Si `fscanf()` no pot llegir cap variable perque hem arribat al final del fitxer, llavors retorna un valor impossible (-1), que és com està definida la constant `EOF` (**E**nd **O**f **F**ile).
- La funció `fscanf()` no entén de línies i columnes. Llegeix nombre rere nombre del fitxer. Si el programa espera quatre variables i a la línia actual només n'hi ha tres, continuarà llegint la fila següent.

Exercici 7.2

Deseu una còpia del programa anterior amb el nom `inversos.c`. Baixeu del Campus Virtual el fitxer `4columnes.dat`. Modifiqueu el codi per tal que escrigui a la pantalla l'invers de cada una de les **quatre** columnes. A cada línia de l'entrada li ha de correspondre una línia de sortida.

Per calcular l'invers d'un nombre real `x`, no cal fer res més que `1/x`.

Entrada i sortida estàndard

Hi ha unes "nanses" especials, que no apunten a cap fitxer pròpiament dit, sinó que són "fluxos de dades" (*data streams*) amb el sistema, les quals no cal obrir ni tancar, i s'anomenen *stdin*, *stdout* i *stderr*.

stdin: Entrada estàndard, apunta al teclat. La funció `scanf()`, de fet, llegeix de *stdin*. Es poden llegir fitxers, i també *stdin*, amb la funció `fscanf()`. De fet, és el mateix fer `fscanf(stdin, ...)` que `scanf(...)`

stdout: El `stdout` (sortida estàndard) apunta a la pantalla. És el mateix fer `fprintf(stdout,...)` que `printf(...)`.

stderr: El `stderr` també escriu per pantalla, però ho fa per un altre canal, que normalment es fa servir pels missatges d'error.

Exercici 7.3

Modifiqueu el codi `inversos.c` per a què llegeixi el `stdin` en lloc del fitxer `4columnes.dat`. Guardeu el programa amb el nom `inversos2.c` i compileu-lo. Executeu el programa donant-li com a dades el mateix fitxer d'abans, amb

```
cat 4columnes.dat | ./inversos2
```

(En la consola de Windows: `type 4columnes.dat | inversos2.exe`). El programa `cat` imprimeix un fitxer a la pantalla. Mitjançant la pipe `|`, l'estem donant com a entrada al programa `inversos2`, a través de la nansa `stdin`.

Nota: El que s'envia a `stderr` no segueix les pipes `|` i `>`. Tot i que, estrictament parlant, també és possible redirigir-ho, en principi sempre sortirà per pantalla. Això és útil per separar la sortida d'un programa dels missatges d'error.

Després executeu `./inversos2` directament, sense cap pipe ni paràmetre. Per exemple, el podeu executar des de Code::Blocks. El programa es quedarà quiet esperant que li inseriu les dades. Inseriu quatre nombres diferents de 0 (separats per espais) i premeu l'Intro. Inseriu vuit nombres i premeu l'Intro. Inseriu quatre nombres prement Intro després de cadascun. Premeu **Ctrl-D**, que equival a **EOF** (en Windows, **Ctrl-Z**).

Exercici 7.4

Modifiqueu el codi anterior per tal que el programa llegeixi el fitxer *sense imprimir res*, calculi les sumes dels elements de cada columna i, al final del programa, escrigui a pantalla la suma de cada una de les **quatre** columnes.

Guardeu el programa modificat amb el nom `suma4.c`.

7.2 Entrada/sortida de caràcters

Hi ha una col·lecció de funcions específiques per a l'entrada/sortida de caràcters. Aquestes són, de fet, més bàsiques que `printf` i `scanf`. No tenen tantes funcionalitats però en ocasions són preferibles.

- Funcions de lectura: `fgetc` retorna un caràcter llegit en un fitxer, `fgets` llegeix caràcters fins que troba un `EOF`, un `'\n'` ("Intro") o arriba al màxim de caràcters indicat, i es guarden a la `cadena` indicada:

```
int fgetc( FILE *nom-de-nansa );
char * fgets( char *cadena, int maxchar, FILE *nom-de-nansa );
```

  En el cas de llegir del teclat, disposem també de la funció `getchar( )`, que és equivalent a `fgetc(stdin)`.

- Funcions d'escriptura: `fputc` imprimeix un caràcter en un fitxer, `fputs` imprimeix una cadena:

```
int fputc( int caracter, FILE *nom-de-nansa );
int fputs( const char *cadena, FILE *nom-de-nansa );
```

  També en aquest cas hi ha les variants `putchar` i `puts` que imprimeixen a la pantalla (`stdout`).

La manera més ràpida de llegir un fitxer és caràcter a caràcter usant la funció `fgetc`. La manera més ràpida d'anar al final de la línia actual és:

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

on `fin` és el nom de la nansa associada al nostre fitxer.

Exemple 7.3

Ara farem servir `fgetc` per comptar el nombre de línies d'un fitxer consistent en dues columnes de nombres enters. A partir d'aquesta informació reservarem dinàmicament la memòria dinàmica necessària per contenir la matriu que formen:

```
// Programa lectura_dinamica.c
// Autor: Professors Dpt Matemàtiques
// Data: 2003-2020

#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) ;
//  Equivalentment:
```

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

Baixa't el fitxer `dades.dat` del Campus Virtual i comprova que el programa llegeix i imprimeix les dades correctament. Observem:

- `while((c = fgetc(fin)) != EOF)` posa el caràcter llegit a la variable `c` i comprova que no sigui `EOF`, és a dir, el bucle es repeteix fins arribar al `E`nd `O`f `F`ile (fi del fitxer). Dins el bucle, la variable `nlin` s'incrementa si hi ha una línia nova.

- `rewind(fin);` "rebobina" el fitxer, ja que havíem arribat al `EOF` i ara el volem llegir amb `fscanf`.

Exercici 7.5

Recorda que és possible redimensionar un bloc de memòria reservada dinàmicament amb la funció `realloc` (secció 6.1). Per tant, una manera alternativa de dur a terme la mateixa tasca de l'exemple anterior consisteix en anar augmentant la mida de la matriu dinàmica `dades` en el mateix bucle que va llegint el fitxer:

```
    for(nlin = 0; fscanf(fin, "%d %d\n", &a, &b) != EOF ; nlin++) {
        if((aux = (int (*)[2]) realloc(dades, 2 * (nlin+1) * sizeof(int)))
    == NULL) {
            fprintf(stderr,
                    "\nERROR: No es possible assignar la memòria necessària
    ...\n\n");
            break;
            }
```

D'aquesta manera s'evita haver de llegir el fitxer dues vegades. Modifica el programa de l'exemple anterior usant aquestes línies (o un codi similar) per anar incrementant la memòria reservada a mesura que es llegeix el fitxer.

Exercici 7.6

Copia el programa de l'exemple anterior amb el nom `filescolumnes.c` i modifica'l perquè escrigui els valors llegits en un segon fitxer, de manera que cada columna passi a ser una fila.

Fes també els canvis necessaris perquè els noms del fitxer d'entrada i sortida es donin des de la consola com a paràmetres (`argv[1]` i `argv[2]`).

La funció `fgets` llegeix una línia sencera fins al màxim de caràcters indicat (incloent el `'\0'` de fi de cadena). Si la lectura finalitza amb el caràcter de nova línia `'\n'`, llavors la cadena contindrà el caràcter de nova línia i després el finalitzador de cadena `'\0'`.

Exemple 7.4

Podem fer servir fgets per llegir cadenes que contenen espais, per exemple, com es pot veure en el programa següent:

```
// Programa fgets.c
// Autor: Professors Dpt Matemàtiques
// Data: Maig 2021
// Llegeix les capçaleres d'un fitxer separat per comes

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

#define MAXLIN 256

int main()
{
   FILE *fin;
   char linia[MAXLIN];
   char *inici, *coma;

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

   if(fgets(linia, MAXLIN, fin) == NULL) {
      fputs("\nERROR: El fitxer 'casos_sexe_municipi.csv' no es pot
   llegir...\n", stderr);
      return 1;
   }

   puts("Els encapçalaments de les columnes són:\n");
   inici = linia;
   do{
      coma = strchr(inici, ',');
      if(coma != NULL)
         *coma = '\0';
      puts(inici);
      inici = coma + 1;
   } while(coma != NULL);

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

Baixa't el fitxer casos_sexe_municipi.csv del Campus Virtual, i comprova que el programa llegeix la primera línia del fitxer, sencera, i que n'imprimeix les capçaleres (els noms dels diferents camps, que es troben a la primera línia, i que en aquest cas estan separats per comes i no per espais). Observem:

- Quan cridem fgets li passem la mida en bytes de la cadena, MAXLIN. Això ens assegura que, fins i tot si la primera línia del fitxer fos més llarga de l'esperat, no escrivim fora de la cadena declarada.
- Si té èxit, el valor de retorn de fgets és un apuntador a la cadena llegida; si hi ha algun error retorna NULL. Per això fem la comprovació `if(fgets(linia, MAXLIN, fin)) == NULL`.
- L'efecte de les línies

```
    coma = strchr(inici, ',');
    if(coma != NULL)
        *coma = '\0';
```

és substituir el caràcter `','` (que és on indica l'apuntador `coma` gràcies a `strchr`) pel codi `'\0'` de final de cadena. Per això quan el `puts` imprimeix en pantalla la cadena que comença a `inici`, s'atura al final d'aquell encapçalament. Com que a continuació establim `inici = coma + 1`, a la següent iteració del bucle, la funció `strchr` trobarà la següent coma de la cadena.

Exercici 7.7

Modifica el programa que acabem de veure per tal que imprimeixi en pantalla els encapçalaments *en ordre invers* al que tenen en el fitxer.

Indicació: Pot haver-hi diverses estratègies per a fer-ho; una opció és usar `strrchr`. Podeu també mantenir el bucle amb `strchr` i crear un vector d'apuntadors als inicis dels encapçalaments successius.

En el programa original, després d'executar tot el bucle `do-while`, la cadena `linia` ha quedat "trossejada" ja que les comes han estat substituïdes per caràcters nuls. Fes els canvis necessaris al programa perquè en acabar, la cadena `linia` quedi igual com era al principi.

Alguns detalls avançats

Trobar, en un fitxer constituït per columnes, els elements en una columna donada es pot fer de manera més eficient amb les instruccions d'*accés directe* a termes concrets d'un fitxer.

- `long ftell(FILE *nom-de-nansa)` retorna un *indicador de posició* dins el fitxer, igual al nombre de `char`s (bytes) des de l'inici del fitxer a la posició actual.
- `int fseek(FILE *nom-de-nansa, long int salt, int direccio)` ens mou `salt` bytes dins el fitxer. `direcció` pot ser:
  - `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.

Exemple 7.5

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 almenys una línia; té almenys 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[] = "7columnes.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;
}
```

- Inicialització: Calculem on és l'inici de la columna NCOL

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

  L'asterisc en el codi de format `%*d` fa que `scanf` descarti el valor llegit.

- Calculem l'amplada de la columna NCOL

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

- Trobem la mida d'una línia completa 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;
```

  Atenció: En `Windows`, els canvis de línia es representen amb *dos* caràcters, i la comprovació del `while` anterior en mode text pot fallar. Perquè us funcioni correctament el programa, caldrà obrir el fitxer en mode binari: `fopen(NomIn, "rb")`

Exercici opcional 7.8

Modifica el programa de l'exercici 7.6 perquè pugui tractar fitxers amb un nombre arbitrari de columnes separades per espais simples.
Indicació: Per determinar el nombre n de columnes, pots tenir en compte que es necessiten $n-1$ espais per separar-les, i comptar els espais que hi ha a la primera fila.

7.3 Fitxers binaris

L'escriptura de valors numèrics amb `printf`/`fprintf`, així com la seva lectura amb `scanf`/`fscanf`, impliquen una *conversió* entre el format intern de representació dels nombres (binari), ja siguin enters o en punt flotant, i el format (usualment decimal) d'escriptura (en el qual cada xifra decimal ocupa un `char`). Això pot ser un inconvenient en termes d'eficiència, que normalment es compensa per la facilitat de manipulació dels fitxers (que podem també llegir o modificar amb qualsevol editor de text). Ara bé, *en el cas que no hem d'obrir el fitxer amb un editor de text o un altre programa* tenim també la possibilitat d'obrir un fitxer en mode binari. En aquest cas hi escriurem amb la funció `fwrite`, que té prototipus

```
int fwrite(void *p, int mida, int nelem, FILE *nansa)
```

i en llegirem les dades amb la funció `fread`, que té prototipus

```
int fread(void *p, int mida, int nelem, FILE *nansa)
```

Aquestes funcions serveixen per escriure al (respectivament, llegir del) fitxer apuntat per `nansa` un seguit de `nelem` valors de `mida` bytes i posar-los a (respectivament, trets de) les posicions de memòria consecutives començant en `p`.

Exemple 7.6

El programa següent llegeix el fitxer `columnes.dat` que hem vist abans i n'escriu una còpia en format binari:

```
// Programa d'exemple d'ús de fwrite
// Autor: Professors Dpt Matemàtiques
// Data: 2022

#include <stdio.h>

int main()
{
    FILE * fitxer1, * fitxer2;
    double a[3];
    fitxer1 = fopen("columnes.dat", "r");
    fitxer2 = fopen("nombres.bin", "wb");

    if(fitxer1 == NULL) {
        fprintf(stderr, "ERROR: El fitxer 'columnes.dat' no es pot obrir
    per lectura...\n\n");
        fclose(fitxer2);
        return 1;
    }
    if(fitxer2 == NULL) {
        fprintf(stderr, "ERROR: El fitxer 'nombres.bin' no es pot obrir per
     escriptura...\n\n");
```

```
        fclose(fitxer1);
        return 1;
    }
    while(EOF != fscanf(fitxer1, "%lf %lf %lf", &a[0], &a[1], &a[2])) {
        if (fwrite(a, sizeof(double), 3, fitxer2) != 3){
            fprintf(stderr, "ERROR d'escriptura en el fitxer...\n\n");
            fclose(fitxer1);
            fclose(fitxer2);
        }
    }
    fclose(fitxer1);
    fclose(fitxer2);
    return 0;
}
```

- Quan cridem `fwrite` li passem com a primer paràmetre el vector `a`, que ja sabem que és equivalent a un apuntador; el segon paràmetre és la mida dels components de `a`, que són `double`; el tercer paràmetre és el nombre de components, és a dir la dimensió del vector, en aquest cas 3; l'últim és la nansa al fitxer.
- El valor de retorn és el nombre de valors escrits amb èxit; normalment serà igual al paràmetre `nelem`, però pot ser inferior en el cas que hi hagi un error d'escriptura.

Compila i executa el programa anterior; veuràs que apareix un fitxer, `nombres.bin`, amb les dades escrites en binari.

Exercici 7.9

Si proves d'obrir el fitxer `nombres.bin` directament, probablement no podràs veure'n el contingut, però sí que es pot llegir usant la funció `fread`. El valor de retorn d'aquesta funció és el nombre de valors llegits amb èxit; normalment si fas

```
fread(a, sizeof(double), 3, fitxer)
```

hauria de donar 3, però pot ser inferior. Concretament, serà inferior quan arribem al final del fitxer.

Escriu un programa, `imprimeix-binari.c`, que llegeixi el fitxer `nombres.bin` fins arribar al final (cosa que cal detectar usant el valor de la funció `fread`) i imprimeixi per pantalla els nombres llegits, de 3 en 3. Comprova que el resultat coincideix amb el contingut del fitxer `columnes.dat`.

La possibilitat d'escriure fitxers binaris ens pot ser també útil a l'hora de manipular formats estàndard, com formats gràfics, per exemple.

Exemple 7.7

El programa següent ens servirà per crear una imatge quadrada amb un gradient de color.

```
// Programa gradient.c
// Autor: Professors Dpt Matemàtiques
// Data: Abril 2022
// Crea un fitxer ppm amb un gradient de color

#include <stdio.h>
```

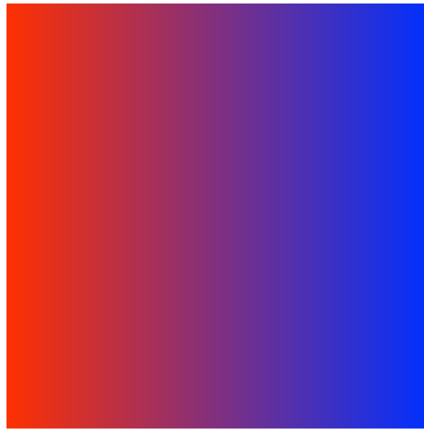
```
#define pixels 255

#define vermell 0
#define verd 1
#define blau 2

int main()
{
   unsigned char color[3] = {255,48,0}; // Vermell
   int x, y;

   char* nomfitxer = "gradient.ppm";
   FILE * fitxer = fopen(nomfitxer, "wb");

   if(fitxer == NULL) {
      fprintf(stderr, "ERROR: El fitxer 'gradient.ppm' no es pot obrir
   per escriptura...\n\n");
      return 1;
   }

   // Els fitxers ppm tenen una capçalera amb metadades
   // P6 identifica el tipus de fitxer
   // Una línia que comença per # és un comentari
   // Tres nombres determinen l'amplada en pixels, l'alçada en pixels
   // i el valor màxim d'intensitat en vermell, verd i blau
   fprintf(fitxer, "P6\n# Gradient vermell-blau\n%d\n%d\n%d\n",
           pixels, pixels, 255);

   for(y = 0; y < pixels; y++) {
      for(x = 0; x < pixels; x++) {
         color[vermell]--;
         color[blau]++;
         fwrite(color, 3, 1, fitxer); // Quan acabi el bucle, serà blau
      }
      color[vermell] = 255;
      color[blau] = 0;
   }

   fclose(fitxer);
   return 0;
}
```

- El resultat d'aquest programa es desa en un fitxer gràfic, `gradient.ppm`. El format gràfic `ppm` és molt poc eficient en termes de mida dels fitxer (per exemple, comparat amb un `png` o `jpg`), però molt fàcil de manipular per al programador: consta d'unes línies en text a l'inici on cal establir l'amplada i l'alçada en píxels de la imatge, i després, en format binari, tres nombres per a cada píxel que representen el color (el primer nombre indica la intensitat de vermell, el segon de verd, i el tercer de blau). Un cop executis el programa, obtindràs un fitxer `ppm`, que pots obrir amb un programa de manipulació de gràfics i tornar-lo a desar en format `png`, per exemple.

- Com que el format del fitxer és binari, l'obrim en mode `"wb"`, però les metadades s'escriuen en mode text amb `fprintf`. A continuació, la imatge pròpiament dita s'escriu amb `fwrite`, escrivient els 3 bytes que codifiquen el color de cada píxel.

- Els píxels estan ordenats per files (com una matriu) començant per dalt a l'esquerra. Per això hi ha els dos bucles que modifiquen les variables `y` i `x`. A l'interior dels bucles es modifica el color canviant el valor de la intensitat del vermell i el blau.

Un cop compilat i executat el programa, heu d'obtenir un fitxer `gradient.ppm`, que obert amb qualsevol programa gràfic té aquest aspecte:



Exercici 7.10

Modifica el programa anterior perquè el gradient de color vagi de dalt a baix (i no d'esquerra a dreta). Crea una nova versió on el gradient sigui en diagonal (vermell a la cantonada superior esquerra, blau a la inferior dreta).

8 Tipus de dades

Com hem anat veient, cada variable en C té assignat un tipus de dada, que ocupa un espai concret de memòria i accepta algunes operacions específiques. A més dels tipus definits per l'estàndard del C, si cal, es poden definir nous tipus de dades.

La instrucció `typedef` del llenguatge C permet la creació de **nous tipus de dades**. En el cas més simple, aquests són "sinònims" dels tipus predefinits (penseu en el tipus `size_t` que vam veure a la pràctica sobre assignació dinàmica).

Exemple 8.1

El codi següent defineix com a `natural` un tipus de variable entera sense signe i com a `Vector` un tipus de variable que és un vector amb 10 posicions `double`.

```c
// Programa DefTipus.c
// Autor: Professors Dpt Matemàtiques UAB
// Data: Abril 2020
// Descripció: Defineix nous tipus de variable.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef unsigned int natural;
typedef double Vector [10];

int main()
{
   natural i, j = 3;
   Vector v, m[9];

   printf("j=%u\n", j);

   srand(time(NULL));     //inicialitzacio

   for(i = 0; i < 10; i++) v[i] = (double)rand() / RAND_MAX;
   for(i = 0; i < 10; i++) printf("%lf\n", v[i]);

   for(i = 0; i < 9; i++) {
      for(j = 0; j < 10; j++) {
         m[i][j] = (double)rand() / RAND_MAX;
         printf("%lf ", m[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

- Veiem que `v` és un vector amb 10 posicions tipus `double`, mentre que `m` és una matriu amb 9 files (el 9 de la definició de `m`) i 10 columnes (el 10 de la definició de tipus `Vector`).
- Per omplir de nombres entre 0 i 1 l'espai de `v` i `m`, s'ha usat la funció `rand()` que genera nombres pseudoaleatoris. Aquesta funció requereix la inicialització d'una *llavor*, que es fa a la línia

```c
   srand(time(NULL));     //inicialitzacio
```

8.1 Estructures

Una estructura és una definició de tipus que permet agrupar altres tipus de dades. Pot contenir un conjunt d'elements heterogenis (anomenats camps) unificats sota un mateix nom, i agrupats en un mateix espai de memòria.

Podem declarar una estructura directament amb el nom de la variable, o bé associant-li un nom de tipus amb la instrucció `typedef`.

Exemple 8.2

Considerem el codi següent:

```
// Programa NotesAlum1.c
// Autor: Professors Dpt Matemàtiques UAB
// Data: 2020-2021
// Descripció: Exemples d'estructures en C

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

const char NomsQualif[4][12]={"Suspès","Aprovat","Notable","Excel·lent"};
const double Pes1 = 0.4;
const double Pes2 = 0.6;

struct nomalum {     // Estructura definida sense typedef
   char Nom[20], Cognom[20];
};

typedef struct {     // Estructura definida amb typedef
   double Ex1;
   double Ex2;
   char Qualificacio[12];
} qualif;
```

- S'han definit dues estructures. La primera, `struct nomalum` amb camps `Nom` i `Cognom` de tipus cadena `char[20]`. De la manera que s'ha definit (sense `typedef`), cada cop que vulguem declarar una variable d'aquest tipus necessitem escriure `struct nomalum`.
- La segona estructura s'anomena `qualif`, amb dos camps de tipus `double`, anomenats `Ex1` i `Ex2`, i un de tipus cadena de caràcters, `Qualificacio`. Com que s'ha declarat amb `typedef`, ara és un tipus de dades, i podem declarar directament les variables d'aquest tipus amb el nom que li hem assignat:

```
int main()
{
   struct nomalum Alumne;
   qualif Notes;
   double ponderat;
   int aux;

   printf("\tNOTES ALUMNE\n");
   printf("\tEntra nom i cognom: ");
   scanf("%s %s", Alumne.Nom, Alumne.Cognom);
   printf("\tEntra les dues notes: ");
   scanf("%lf %lf", &Notes.Ex1, &Notes.Ex2);

   ponderat = Pes1 * Notes.Ex1 + Pes2 * Notes.Ex2;
```

```
    aux = (ponderat - 3) / 2;
    if (aux < 0) aux = 0;
    strcpy(Notes.Qualificacio, NomsQualif[aux]);

    printf("\n\tEL RESULTAT HA ESTAT:\n");
    printf("\tNom: %s \tCognom: %s\n", Alumne.Nom, Alumne.Cognom);
    printf("\tQualificació: %s\n", Notes.Qualificacio);

    return 0;
}
```

- Observerm que, per accedir als camps d'una estructura s'usa la sintaxi `nomvariable.nomcamp` amb el símbol `.` entre el nom de la variable i el del camp:
  - Per exemple, `Alumne.Nom` fa referència al camp `Nom` de `Alumne`.
  - Podem accedir a l'adreça on es guarda la variable `Notes` mitjançant `&`, i es pot utilitzar aquesta sintaxi per a utilitzar `scanf`.
  - El codi demana per pantalla les notes numèriques, mentre que omple la cadena `Notes.Qualif` fent un càlcul.
- Cada instància d'una estructura conté tots els camps seguits:

	1502	1510	1518		
	Notes.Ex1 4.1	Notes.Ex2 6.2	Notes.Qualificacio "Aprovat"	. . .	Alumne.Nom "Sam"

Els valors de les variables de tipus estructura es poden incialitzar a l'estil dels vectors, per exemple:

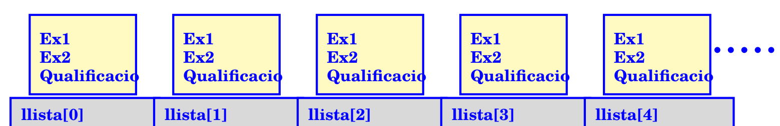
```
    struct nomalum Alumne = {"Àlex", "Bosch"};
```

assigna el valor `"Àlex"` a `Alumne.Nom` i el valor `"Bosch"` a `Alumne.Cognom`. En canvi, a diferència dels vectors i les matrius, es poden traspassar fent una assignació directa. Així, les instruccions

```
    struct nomalum Alumne = {"Àlex", "Bosch"};
    struct nomalum Alumne2;
    Alumne2 = Alumne;
```

copiarien les dues cadenes completes `Nom` i `Cognom` de l'estructura `Alumne` a l'estructura `Alumne2`.

Les estructures poden també funcionar "encaixades" com a camps en altres estructures més grans. De la mateixa manera, igual com es creen vectors de variables dels tipus bàsics, es poden crear vectors de tipus definits i vectors d'estructures. Podem pensar un vector d'estructures visualment de la següent manera:



Exemple 8.3

El programa anterior pren sentit quan introduïm un *vector* de variables de tipus `qualif` per guardar les notes de diversos alumnes. També té lògica incloure el nom i cognom en la pròpia estructura on posarem les qualificacions:

```c
// Programa NotesAlum2.c
// Autor: Professors Dpt Matemàtiques UAB
// Data: 2020-2021
// Descripció: Vectors d'estructures i estructures encaixades

#include <stdio.h>
#include <string.h>
#define NALUM 5

const char NomsQualif[4][12]={"Suspès","Aprovat","Notable","Excel·lent"};
const double Pes1 = 0.4;
const double Pes2 = 0.6;

struct nomalum {      // Estructura definida sense typedef
   char Nom[20], Cognom[20];
};

typedef struct {      // Estructura definida amb typedef
   struct nomalum Alumne;
   double Ex1;
   double Ex2;
   char Qualificacio[12];
} qualif;

int main()
{
   qualif llista[NALUM];
```

- L'estructura `nomalum` s'utilitza dins la definició de l'estructura `qualif`.
- A la funció `main` es declara un vector de dades tipus `qualif` que es diu `llista`.
- `llista[i]` (on `i` és un enter entre 0 i 4) és una estructura de tipus `qualif`, amb camps `llista[i].Alumne`, `llista[i].Ex1`, `llista[i].Ex2` i `llista[i].Qualificació`.
- `llista[i].Alumne` és una estructura de tipus `struct nomalum`, amb dos camps tipus cadena: `llista[i].Alumne.Nom` i `llista[i].Alumne.Cognom`.

Exercici 8.1

Desa el programa anterior amb el nom `NotesAlum2.c`, introduïnt els canvis de l'exemple 8.3. Fes els canvis necessaris al cos de la funció `main`, introduïnt un bucle per tal d'entrar noms i notes de 5 alumnes, i fer que les qualificacions es guardin al vector `llista`.

8.2 Apuntadors a estructures

Les estructures també accepten apuntadors i reserves de memòria dinàmica. La sintaxi per a reservar és exactament la mateixa que la descrita a la Pràctica 6, i obtindrem un apuntador a una estructura.

Exercici 8.2

Modificarem el programa de l'exercici anterior per tal de poder treballar amb un nombre qualsevol d'alumnes reservant la memòria corresponent amb `malloc`. Canvia les línies adequades per les següents i desa el programa amb el nom `NotesAlum3.c`. Comprova el seu funcionament i assegura't d'entendre'l. En particular, és important que la funció `sizeof` funciona correctament amb estructures (i tipus definits per nosaltres).

```
    qualif *llista;
    double ponderat;
    int nalum, aux;

    printf("\tNOTES ALUMNES\n");

    printf("\tQuants alumnes tens? ");
    scanf("%d",&nalum);

    // Reservem la memòria
    llista = (qualif *) malloc(nalum * sizeof(qualif));
    if(llista == NULL) {
       printf("No s'ha pogut reservar la memòria\n");
       return 1;
    }
```

Vectors i polinomis esparsos

És freqüent en aplicacions trobar-se amb matrius, vectors o polinomis que tenen gairebé tots els coeficients iguals a zero. El polinomi $P(x) = 2.14x^{45} - x^{99} + 100.52x^{131} + 7x^{97139}$ és un exemple de *polinomi espars*. Si guardem en memòria aquest polinomi en la forma d'un vector de coeficients `double a[97140]` que contingui $(a_0, a_1, \ldots, a_{97139})$, hi haurà 97136 d'aquests coeficients iguals a zero. Clarament no és la manera més eficient de procedir. En comptes d'això, podem guardar només els parells (k, a_k) on $a_k \neq 0$, en un vector `monomi a[4]` on l'estructura `monomi` està definida com

```
typedef struct {
unsigned int k;
double ak;
} monomi;
```

Exercici 8.3

Si tenim dos polinomis esparsos

$$P(x) = a_{k_1}x^{k_1} + a_{k_2}x^{k_2} + \cdots + a_{k_m}x^{k_m}$$
$$Q(x) = b_{\ell_1}x^{\ell_1} + b_{\ell_2}x^{\ell_2} + \cdots + b_{\ell_n}x^{\ell_n}$$

i anomenem $E_P = \{k_1, \ldots, k_m\}$, $E_Q = \{\ell_1, \ldots, \ell_n\}$ els conjunts d'exponents presents en cadascun d'ells, la seva suma $P(x)+Q(x)$ serà un polinomi espars, on el conjunt dels exponents presents és la unió $E = E_P \cup E_Q$; el coeficient de x^e en $P(x) + Q(x)$ per a cada $e \in E$ és

$$c_e = \begin{cases} a_e & \text{si } e \in E_P \setminus E_Q, \\ b_e & \text{si } e \in E_Q \setminus E_P, \\ a_e + b_e & \text{si } e \in E_P \cap E_Q. \end{cases}$$

Fes un programa per a sumar polinomis esparsos que utilitzi l'estructura `monomi` presentada anteriorment. Suposarem que els parells (k, a_k) que defineixen els sumands es troben en dos fitxers, `Pespars.dat` i `Qespars.dat`, que consten de dues columnes separades per espais, on la primera columna conté els exponents k i la segona els coeficients a_k. Podem suposar que aquests monomis estan ordenats per graus, és a dir, que l'exponent k és cada vegada més gran a mesura que avancem en el fitxer. El programa ha de comptar les línies en cada fitxer per determinar el nombre de monomis presents en cada polinomi, reservar la memòria necessària amb `malloc` (o `calloc`), i llegir els fitxers per desar els coeficients a la memòria reservada. Per aquesta part pots aprofitar codi de l'exemple 7.3.

A continuació caldrà reservar memòria per al polinomi espars suma; com que el cardinal $|E|$ de E és com a màxim igual a $|E_P| + |E_Q|$, serà suficient reservar aquest nombre de monomis. El programa ha de fer la suma, cosa que vol dir resseguir tots els coeficients de P i de Q per ordre de graus. El bucle següent pot servir d'inspiració (aquí suposem que P té m=`monomisP` monomis, Q té n =`monomisQ` monomis, i posarem a la variable `monomissuma` el nombre de monomis de la suma.)

```
  unsigned int j = 0;                           // Índex per als
  monomis de Q
  for(unsigned int i = 0; i < monomisP; i++){   // Recorrem els monomis
   de P
     while((j < monomisQ) && (b[j].k < a[i].k)){ // Graus de Q que no só
  n a P
        c[monomissuma] = b[j];                  // Es copia grau i
  coeficient
        monomissuma++;
        j++;                                    // Passar al monomi seg
  üent de Q
        }
     if((j < monomisQ) && (b[j].k == a[i].k)){   // Graus que són a P i
  Q
        if (fabs(b[j].ak + a[i].ak) > tol){     // si no es cancel·len
           c[monomissuma].k = a[i].k;           // es copia grau
           c[monomissuma].ak = a[i].ak + b[j].ak;// es sumen coeficients
           monomissuma++;
        }
        j++;                                    // Passar al monomi seg
  üent de Q
     }
     else{                                      // Graus de P que no só
  n a Q
        c[monomissuma] = a[i];                  // Es copia grau i
  coeficient
        monomissuma++;
     }
  }
```

Observa que l'índex `i` que recorre els monomis de P s'incrementa a cada iteració del bucle, i cal anar comparant els grau del monomi `j`-èsim de Q i l'`i`-èsim de P per incrementar `j` quan toqui.

Tingues present que si el grau de Q és més gran que el de P aquest bucle és insuficient, ja que no sumarà els últims termes de Q! Afegeix el codi que falta per acabar la suma.

Finalment caldrà escriure el resultat en un nou fitxer `PQespars.dat`.

Les estructures també es poden passar com a arguments a funcions. Com en el cas d'altres tipus de dades, es pot fer a partir dels valors (encara que tingui molts camps, es passa una còpia de tots els valors de l'estructura) o bé a partir d'un apuntador. D'altra banda, també es pot retornar el seu valor (tota una estructura) amb un `return`.

Exemple 8.4

El codi següent trasllada part del codi del programa NotesAlum2.c a una funció per a entrar les dades d'un alumne:

```
qualif entranotes(void)
{
   qualif Element;
   double ponderat;
   int aux;

   printf("\tEntra nom, cognom i dues notes: ");
   scanf("%s %s %lf %lf", Element.Alumne.Nom, Element.Alumne.Cognom,
            &(Element.Ex1), &(Element.Ex2));
   ponderat = Pes1 * Element.Ex1 + Pes2 * Element.Ex2;
   aux = (ponderat - 3) / 2;
   if (aux < 0) aux = 0;
   strcpy(Element.Qualificacio, NomsQualif[aux]);
   return Element;
}
```

Es declara localment a la funció un element de la llista, de tipus qualif, i aquest es retorna *com el valor de la funció* entranotes(). Ara, la part de lectura de les dades a la funció main queda simplificada així:

```
   qualif llista[NALUM];
   printf("\tNOTES ALUMNES\n");
   for(int i = 0; i < NALUM; i++) {
      llista[i] = entranotes();
   }
```

L'inconvenient d'aquesta funció és que necessita crear una estructura local qualif Element només per a després copiar-la a la llista. Podem fer que la funció entranotes escrigui directament al lloc desitjat de la llista, passant com argument *l'apuntador* a la variable on hem d'escriure:

```
void entranotes(qualif *Element)
{
   double ponderat;
   int aux;

   printf("\tEntra nom, cognom i dues notes: ");
   scanf("%s %s %lf %lf", (*Element).Alumne.Nom, (*Element).Alumne.Cognom
   ,
            &((*Element).Ex1), &((*Element).Ex2));
   ponderat = Pes1 * (*Element).Ex1 + Pes2 * (*Element).Ex2;
   aux = (ponderat - 3) / 2;
   if (aux < 0) aux = 0;
   strcpy((*Element).Qualificacio, NomsQualif[aux]);
}
```

Llavors s'ha de substituir la línia corresponent per:

```
   entranotes(&llista[i]);
```

O, equivalentment:

```
   entranotes(llista+i);
```

En general, passar el paràmetre amb l'apuntador té l'avantatge que no cal *copiar* una quantitat gran de memòria. Aquest avantatge serà més important com més memòria ocupi l'estructura en consideració.

Per accedir al contingut d'una estructura apuntada (com `*Element` en l'exemple anterior) hi ha dues sintaxis equivalents: la que hem fet servir en l'exemple és `(*Element).Ex1`, però també podem utilitzar `->`, escrivint `Element->Ex1` (que és més breu, i potser més clar). Així, la instrucció `strcpy` de la funció anterior esdevindria:

```
    strcpy(Element->Qualificacio, NomsQualif[aux]);
```

Exercici 8.4

Copia el programa `NotesAlum3.c` amb el nom `NotesAlum4.c` i modifica'l perquè s'utilitzi la funció `entranotes` que té com a argument un apuntador a una estructura `notes`. Modifica la funció `entranotes` perquè usi la sintaxi `Element->` en lloc de `(*Element).` Afegeix una funció amb prototipus

```
void imprimeixnotes(qualif *Element)
```

que imprimeixi els resultats d'un alumne, i substitueix les línies corresponents de la funció `main` per una crida a aquesta funció.

Alguns detalls avançats

Per avaluar polinomis esparsos també es pot aplicar la regla de Horner, però cal tenir present que entre dos monomis consecutius s'haurà de multiplicar pel valor de x un nombre de vegades equivalent a la diferència dels graus. Així per exemple tenim que

$$2.14x^{45} - x^{99} + 100.52x^{131} + 7x^{97139} = x^{45}(2.14 + x^{54}(-1 + x^{32}(100.52 + x^{97008} \cdot 7))).$$

Exercici opcional 8.5

Modifica el programa de l'exercici opcional 4.6 per tal que treballi amb polinomis esparsos codificats amb l'estructura `monomi` d'aquesta pràctica. Utilitza el programa resultant per trobar les dues arrels properes a $x = 1$ (una lleugerament més gran i una de més petita) del polinomi

$$-9.803\,x^3 + 12.23\,x^6 - 3.500\,x^{10} + 4.301\,x^{13} + 1.700\,x^{18} - 2.800\,x^{23}.$$

A més d'apuntadors a estructures, també podem tenir apuntadors com a components de les estructures, i reservar memòria per a elles amb `malloc`.

Exercici opcional 8.6

Als programes `NotesAlum*.c` es reserva memòria per a 20 caràcters al nom i 20 al cognom. Això fa que si els noms o cognoms són curts, no s'utilitzi una part de la memòria, i que no es puguin entrar noms i cognoms llargs.
Fes una còpia del programa `NotesAlum3.c`, anomenada `NotesAlum4.c`, i modifica-la de manera que la definició de les estructures sigui:

```
struct nomalum {    // Estructura definida sense typedef
   char *Nom, *Cognom;
};

typedef struct {    // Estructura definida amb typedef
   struct nomalum Alumne;
   double Ex1;
   double Ex2;
```

```
    char Qualificacio[12];
} qualif;
```

Fes els canvis necessaris perquè, un cop sabem la longitud del nom (respectivament cognom) es faci la reserva de memòria per a aquell espai i es guardi. El programa ha de funcionar correctament per a noms (i cognoms) de fins a 30 caràcters.

union

Considerem una estructura com la següent:

```
typedef struct {     // Estructura definida amb typedef
    char nom[20];
    char cognom[20];
    char nacio[20];
    char NIF[10];
    char NIE[10];
} dades;
```

En la pràctica, els camps `NIE` i `NIF` no s'usaran mai els dos alhora, perquè una persona que té `NIF` no té `NIE` i recíprocament. Aleshores, aquesta estructura està malgastant memòria. El **C** té una altra manera de combinar camps en una sola estructura, guardant-los en *la mateixa* posició de memòria, anomenada `union` i que té una sintaxi anàloga a les `struct`s. Usant unions, la declaració anterior es podria reformular com:

```
typedef union{
    char NIF[10];
    char NIE[10];
} codi;

typedef struct {     // Estructura definida amb typedef
    char nom[20];
    char cognom[20];
    char nacio[20];
    codi ID;
} dades;
```

Llavors si `Alumne` és una variable de tipus `dades`, podem accedir a `Alumne.ID.NIF` o `Alumne.ID.NIE` (i ocuparan les mateixes posicions de memòria.)

9 Estructures enllaçades

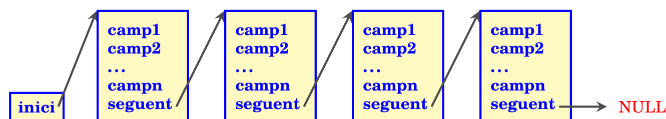
Un dels possibles usos de les `struct`s és per a definir estructures de dades *enllaçades*, que són col·leccions de dades més flexibles que els vectors (principalment a l'hora d'afegir o treure elements).

9.1 Llistes enllaçades

Una llista enllaçada és una col·lecció d'elements (*nodes*) disposats seqüencialment que permet la inserció i eliminació d'elements en qualsevol lloc de la seqüència (a diferència dels vectors) sense haver de conèixer la mida de la llista i de les dades.

Dues eines permeten el funcionament d'aquest mecanisme:

- Cada node conté, com un dels elements de l'estructura, un apuntador al node següent.



- Reservem i alliberem els espais de memòria dels nodes individualment, mitjançant l'assignació dinàmica.

  En l'ús de les llistes dinàmiques cal sempre tenir present fer la reserva de memòria com a pas previ a la creació d'un node i d'alliberar-la en el moment en què un node desapareix.

Exemple 9.1

En l'exercici 8.3 ens vam trobar amb la dificultat de recórrer simultàniament dos vectors (els monomis de dos polinomis P i Q) per tal de sumar-los i disposar els monomis de la suma per ordre de graus. Si tenim la possibilitat d'inserir monomis, com ens ho permeten les llistes enllaçades, l'algoritme de la suma resultarà conceptualment més simple.

Cal una **nova estructura** per als monomis, similar a la de l'exercici 8.3 però incloent un apuntador al monomi següent:

```
typedef struct monom{     // struct monom és "sobrenom" de monomi
   unsigned int k;
   double ak;
   struct monom *seguent;
} monomi;
```

- Com que el nom del tipus (`monomi`) no està definit fins al final del `typedef`, per tenir un apuntador al mateix tipus de node dins de l'estructura hi hem de fer referència com a `struct monom`.

En comptes de començar amb dues llistes de monomis per als dos sumands, per després crear-ne una tercera amb la suma, el que farem és crear una sola llista, inicialment buida (corresponent al polinomi zero), a la qual sumarem els monomis d'un en un, primer els d'un polinomi i després els de l'altre. L'efecte final és de sumar el segon polinomi al primer.

```
int main()
{
   monomi *inici = NULL, *actual = NULL;
   FILE *fin, *fout;
   unsigned int k;
   double ak;
```

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

    while( fscanf(fin, "%u %lf\n", &k, &ak) != EOF)
        suma_monomi(&inici, k, ak);

    fclose(fin);

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

    while( fscanf(fin, "%u %lf\n", &k, &ak) != EOF)
        suma_monomi(&inici, k, ak);

    fclose(fin);

    if((fout = fopen("PQespars.dat", "w")) == NULL)  {
        fprintf(stderr,
                "\n ERROR: El fitxer 'PQespars.dat' no es pot obrir...\n\n"
    );
        return 1;
    }

    for(actual = inici; actual!=NULL; actual = actual->seguent)
        fprintf(fout, "%u %lf\n", actual->k, actual->ak);
    fclose(fout);

    return 0;
}
```

- Observem en les últimes línies com fer un bucle que recorre una llista enllaçada: no cal saber la longitud de la llista, sinó que es passa de cada node al següent, fins arribar a l'últim (que es reconeix perquè el seu següent és NULL).

- Els bucles que llegeixen cada fitxer són enganyosament simples, ja que només criden la funció `suma_monomi` que s'encarrega d'afegir el monomi $a_k x^k$ al polinomi suma. Ara veurem com programar aquesta funció.

El que ha de fer la funció `suma_monomi` depèn de si a la llista de monomis que formen el polinomi ja calculat hi ha o no un monomi del mateix grau k que el que s'ha llegit. Si no hi ha cap monomi de grau k, caldrà afegir un node nou en el lloc oportú de la llista. En canvi, si n'hi ha un, la feina a fer és sumar els coeficients i, en el cas que es cancel·lin, treure el monomi de la llista.

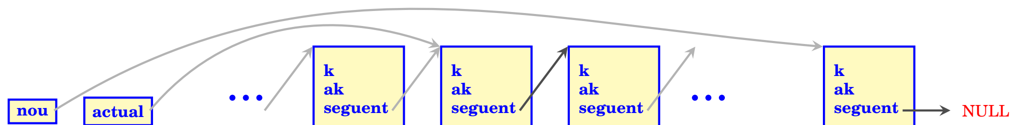
La simplicitat conceptual de l'algoritme té la contrapartida en la complexitat afegida de programar la inserció i esborrat d'elements en una llista enllaçada, que quedarà inclosa en aquesta funció. Vegem-ho per parts.

1. Per a cada monomi cal fer la **reserva de memòria** individualment:

```
        if((nou = (monomi *) malloc(sizeof(monomi))) == NULL) {
            fprintf(stderr,
              "\nERROR: No es possible assignar la memòria necessària
...\n\n");
            exit (1);
        }
```

En contrapartida, no ens cal saber el nombre de monomis per fer la reserva.

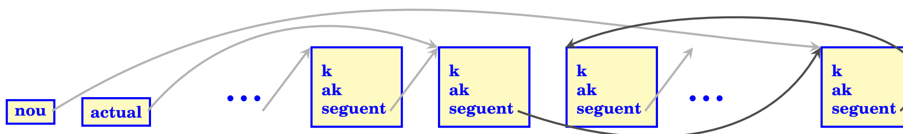
2. **Inserir un monomi nou** a la llista és una operació senzilla. Suposem que tenim un apuntador, `monomi *actual;` apuntant a l'element de la llista *anterior* a la posició on hem d'afegir el monomi, i el nou monomi en una adreça apuntada per `monomi *nou;` .



Llavors farem:

```
        nou->seguent = actual->seguent;
        actual->seguent = nou;
```

El resultat és la llista amb l'element inserit:



En quin lloc de la memòria es trobi físicament cada node és irrellevant, ja que en la llista, l'ordre queda determinat pels apuntadors que lliguen cadascun amb el següent.

Hi ha un cas especial, quan calgui inserir un node a l'inici de la llista (això passarà si la llista és buida o si volem sumar un monomi de grau més petit que els ja existents). En aquest cas caldria fer

```
      nou->seguent = inici;
      inici = nou;
```

Fixem-nos que aquesta inserció obliga a què la funció `suma_monomi` modifiqui el valor de l'apuntador a l'inici de la llista. Per aquest motiu, se li ha de passar *l'adreça* `&inici`, que és de tipus `monomi **` i per tant el seu prototipus ha de ser

```
void suma_monomi(monomi ** inici, unsigned int k, double ak);
```

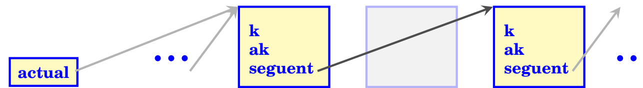
Igualment ens veiem portats a modificar lleugerament el codi que hem posat en vermell, que en realitat serà:

```
      nou->seguent = *inici;
      *inici = nou;
```

Fixa't que el cas particular d'inserir un element al final de la llista no requereix un tractament particular, ja que l'assignació `nou->seguent = actual->seguent;` en aquest cas li assigna el valor NULL que ja és correcte per a l'últim element.

3. En el cas que un coeficient es cancel·li i calgui **eliminar un node de la llista**, el procés és similar. Suposem que l'element a eliminar és el següent a l'apuntat per `monomi *actual;` Farem:

```
            aux = actual->seguent;
            actual->seguent = actual->seguent->seguent;
            free(aux);
```



També per eliminar nodes el cas particular d'eliminar el primer tindrà un codi lleugerament diferent:

```
         aux = *inici;
         *inici = (*inici)->seguent;
         free(aux);
```

4. Com hem vist, a la funció `suma_monomi` li passem el grau i el coeficient del nou monomi. Per **determinar en quin lloc de la llista** cal inserir-lo, resseguirem tots els termes ja presents amb un bucle, començant per l'inici i fins que en trobem un de grau més gran o trobem el final de la llista:

```
      actual = *inici;                                    // Trobar posició a
    la llista
      while ((actual->seguent !=NULL) && (actual->seguent->k < k))
         actual = actual->seguent;
```

Quan se surt del bucle `while`, poden donar-se dos casos: si el grau del monomi `actual->seguent` és estrictament més gran que `k` o no hi ha monomi següent, caldrà afegir un monomi. Alternativament, si el grau del monomi següent és igual a `k`, caldrà sumar els coeficients.

Reunint totes aquestes observacions, la funció `suma_monomi` quedarà així:

```
void suma_monomi(monomi ** inici, unsigned int k, double ak)
{
   monomi *actual = NULL, *aux = NULL, *nou = NULL;

   if ((*inici == NULL) || ((*inici)->k > k)){    // Cal inserir a l'inici
      if((nou = (monomi *) malloc(sizeof(monomi))) == NULL) {
      fprintf(stderr,
        "\nERROR: No es possible assignar la memòria necessària...\n\n");
      exit(1);
      }
      nou->k = k;
      nou->ak = ak;
      nou->seguent = *inici;
      *inici = nou;
   }
   else if ((*inici)->k == k){                    // Cal sumar a l'inici
      (*inici)->ak += ak;
      if (fabs((*inici)->ak) < tol){              // Si cancel·lació,
   eliminar
```

```
            aux = *inici;
            *inici = (*inici)->seguent;
            free(aux);
        }
    }
    else{
        actual = *inici;                                    // Trobar posició a la
    llista
        while ((actual->seguent !=NULL) && (actual->seguent->k < k))
            actual = actual->seguent;
        if ((actual->seguent == NULL) || (actual->seguent->k > k)){ // Cal
    inserir
            if((nou = (monomi *) malloc(sizeof(monomi))) == NULL) {
                fprintf(stderr,
                  "\nERROR: No es possible assignar la memòria necessària...\
    n\n");
                exit (1);
            }
            nou->k = k;
            nou->ak = ak;
            nou->seguent = actual->seguent;
            actual->seguent = nou;
        }
        else{                                               // Cal sumar
            actual->seguent->ak += ak;
            if (fabs(actual->seguent->ak) < tol){    // Si cancel·lació,
    eliminar
                aux = actual->seguent;
                actual->seguent = actual->seguent->seguent;
                free(aux);
            }
        }
    }
}
```

Escriu les funcions `main` i `suma_monomi` en un fitxer anomenat `sumaespars_enllacat.c`. Comprova el seu funcionament sumant els polinomis dels fitxers `Pespars.dat` i `Qespars.dat` del Campus Virtual.

Exercici 9.1

Un cop s'acaba l'execució del programa, caldria alliberar la memòria ocupada pels nodes existents a la llista. Crea una funció amb el següent prototipus que faci aquesta feina, i fes que es cridi aquesta funció després d'escriure el resultat al fitxer.

```
void buida_llista(monomi ** inici);
```

Exercici 9.2

Un cop tenim programada la suma de monomis en un ordre arbitrari, és fàcil afegir altres funcionalitats al programa. En aquest exercici es tractarà de programar el producte de polinomis. Copia el programa anterior en un nou fitxer `producteespars.c` i fes els següents canvis.

- Necessitaràs tres llistes diferents per a P, Q, i PQ. Per tant hauràs de declarar tres apuntadors `inici` a la funció `main`, com per exemple

```
monomi *iniciP, *iniciQ, *iniciPQ;
```

- Pots fer servir la funció `suma_monomi` per llegir cadascun dels polinomis P, Q dins la seva llista.

- El producte de dos polinomis

$$P(x) = \sum_{i=1}^{m} a_{k_i} x^{k_i}, \quad Q(x) = \sum_{j=1}^{n} b_{\ell_j} x^{\ell_j},$$

ve donat per

$$PQ(x) = \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{k_i} b_{\ell_j}) x^{\ell_j + k_i}.$$

Un cop tens les dues llistes corresponents a P i Q, pots fer un dos bucles (un dins l'altre) per recórrer-les, i per a cada parella de monomis $(a_{k_i} x^{k_i}, b_{\ell_j} x^{\ell_j})$, cridar la funció

```
suma_monomi (&iniciPQ, actualP->k + actualQ->k, actualP->ak * actualQ
   ->ak);
```

Comprova el teu programa multiplicant els polinomis dels fitxers `Pespars.dat` i `Qespars.dat` del Campus Virtual.

9.2 Variants

Tot i que un dels punt forts de les llistes enllaçades és poder inserir i eliminar nodes en qualsevol posició, de vegades es fan servir en situacions en què les insercions/eliminacions es produeixen sempre en un ordre determinat. Això permet simplificar el codi fins a cert punt.

Pila (stack)

S'anomena *pila* (stack en anglès) una estructura de dades lineal amb aquestes restriccions d'accés:

- Només es pot afegir elements al "cim" de la pila.

- Només es pot treure elements del "cim" de la pila.

Aquestes restriccions es poden resumir amb l'acrònim anglès `LIFO` (`L`ast `I`n, `F`irst `O`ut), és a dir, el primer element que es treu és l'últim que s'ha afegit. Per analogia amb objectes quotidians, una operació **empilar** (push) equivaldria a posar un plat sobre una pila de plats, i una operació **desempilar** (pop) a retirar-lo.

Una possible manera de programar una pila és mitjançant una llista enllaçada, en la qual cada node que s'insereix es posa a l'inici, i sempre que s'elimina un node es fa a l'inici.

[2]Font: wikipedia, by User:Boivie, Domini públic `https://commons.wikimedia.org/w/index.php?curid=1439935`

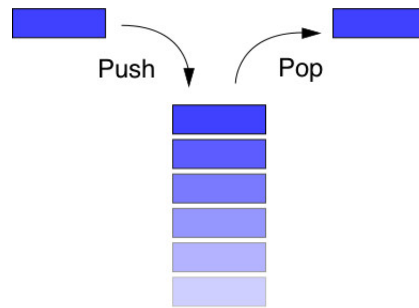


Figura 1: Representació d'una pila. [2]

Exercici 9.3

Ara farem servir una pila per llegir un fitxer amb dues columnes d'enters i tornar-lo a desar en un segon fitxer, amb l'ordre de les files invertit. La pila s'ha de programar a partir de l'estructura enllaçada següent:

```
typedef struct node{     // struct node és "sobrenom" de pila
   int x;
   int y;
   struct node *seguent;
} pila;
```

Hauràs de programar dues funcions, `empila` i `desempila`, amb els prototipus següents:

```
void empila(pila ** inici, int x, int y);
void desempila(pila ** inici, int *x, int *y);
```

Tal com el seu nom indica, `empila` ha d'inserir un node `nou` a l'inici de la llista apuntada pel paràmetre `**inici`, assignant els valors `x,y` a les components `nou->x` i `nou->y`.

Similarment, `desempila` ha de copiar, a les variables indicades, les components `->x` i `->y` del primer node de la llista apuntada pel paràmetre `**inici`, i eliminar aquest node de la llista.

Si has escrit adequadament les dues funcions, la funció `main` següent farà que s'imprimeixin al fitxer `sedad.dat` les files del fitxer `dades.dat` en l'ordre invertit:

```
int main()
{
   pila *inici = NULL;
   FILE *fin, *fout;
   int x, y;

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

   while( fscanf(fin, "%d %d\n", &x, &y) != EOF)
      empila(&inici, x, y);

   fclose(fin);

   if((fout = fopen("sedad.dat", "w")) == NULL)  {
      fprintf(stderr,
              "\n ERROR: El fitxer 'sedad.dat' no es pot obrir...\n\n");
      return 1;
   }
```

```
    while(inici != NULL){
        desempila(&inici, &x, &y);
        fprintf(fout, "%d %d\n", x, y);
    }
    fclose(fout);

    return 0;
}
```

Cua (queue)

S'anomena *cua* (queue en anglès) una estructura de dades lineal amb aquestes restriccions d'accés:

- Només es pot afegir elements al final de la cua.
- Només es pot treure elements de l'inici de la cua.

Aquestes restriccions es poden resumir amb l'acrònim anglès `FIFO` (`F`irst `I`n, `F`irst `O`ut), és a dir, el primer element que es treu és el primer que s'ha afegit.

Si volem programar una cua mitjançant una llista enllaçada, necessitarem, a més de l'apuntador a l'inici que sempre requereixen les llistes, un apuntador a l'últim element introduït, ja que allí és on inserirem els elements nous.

Exercici 9.4

Ara farem servir una cua per llegir des de `stdin` un fitxer amb dues columnes d'enters del qual no sabem el nombre de files. La idea és fer el mateix que en el programa `lectura_dinamica` de l'exemple 7.3, però amb la restricció que no es pot fer `rewind` en `stdin`. Per això el que farem és posar totes les files llegides en una cua alhora que les anem comptant, i després copiar-les en la matriu reservada amb `malloc` tot buidant la cua.

Pots usar la mateixa estructura `pila` de l'exercici anterior, però en comptes de la funció `empila` necessitaràs una nova funció

```
void encua(pila **inici, pila **ultim, int x, int y);
```

L'element nou que crea aquesta funció s'ha d'inserir després de l'apuntat per `ultim`, i modificar l'apuntador a `ultim`. L'`inici` només serà modificat en el cas que la llista sigui buida en el moment de cridar la funció, ja que llavors caldrà posar l'inici igual a l'últim.

Com que en la cua, els nodes es treuran en l'ordre en què s'han introduït, és a dir per l'inici, pots aprofitar la funció `desempila` de l'exemple anterior. Llavors la funció `main` que tens a continuació fa el mateix que la de l'exercici 7.4 però aplicat a `stdin`.

```
int main()
{
    int nlin = 0;
    int x,y;
    pila *inici = NULL, *ultim = NULL;
    int (*dades)[2]; // Apuntador a vectors de dues components: dades[i]
    apunta a dades[i][0]

    while( fscanf(stdin, "%d %d", &x, &y) != EOF){
        encua(&inici, &ultim, x, y);
        nlin++;
```

```
    }

    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(int i = 0; i < nlin; i++) desempila(&inici, &dades[i][0], &dades[i
][1]);

    // Ara la cua ja és buida i les dades són a la matriu:
    for(int i = 0; i < nlin ; i++) printf("%d %d\n", dades[i][0], dades[i
][1]);

    return 0;
}
```

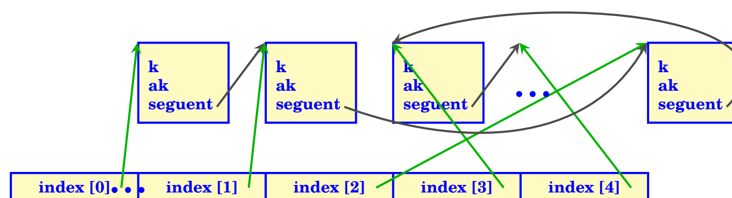
Prova el teu programa, primer redirigint un fitxer:

```
cat dades.dat | ./lectura_dinamica_cua
```

i després escrivint les dades directament a la consola. Per introduir el final del fitxer (`EOF`) hauràs de prémer `Ctrl-D` (en Linux o Mac) o `Ctrl-Z` (en Windows).

Indexació

L'accés seqüencial de les llistes enllaçades pot ser lent quan es vol accedir tant sols a una part o bé si es vol fer en un altre ordre. Una possible solució és utilitzar una variable que faci d'índex: un vector d'apuntadors a cada una de les entrades de la llista enllaçada.



Exercici 9.5

La llista calculada en el programa `sumaespars_enllacat2.c` està formada per nodes que són estructures de tipus `monomi` i estan ordenades de menor a major grau. Modificarem el programa per tal que, a més de la sortida en fitxer en el format de dues columnes $k a_k$, s'escrigui el polinomi en pantalla, amb els monomis en l'ordre oposat: de major a menor grau.

Copia el programa en un fitxer nou anomenat `sumaespars_enllacat3.c`. Començarem fent els canvis necessaris per crear un vector índex que estarà en el mateix ordre.

1. Declarem la variable:

```
    monomi ** index;
    unsigned int longitud, i;
```

2. Aprofitarem el bucle que desa el resultat al fitxer per comptar la longitud de la llista:

```
longitud = 0;
for(actual = inici; actual!=NULL; actual = actual->seguent){
   fprintf(fout, "%u %lf\n", actual->k, actual->ak);
   longitud++;
}
```

3. Llavors reservem la memòria per l'índex i hi guardem les dades:

```
actual = inici;
for(i = 0; i < longitud; i++){
   index[i] = actual;
   actual = actual -> seguent;
}
```

4. Ara els coeficients i exponents són accessibles via `index[i]->ak` i `index[i]->k` respectivament. Afegeix un bucle que recorri el vector `index` en ordre decreixent per tal d'imprimir per pantalla el polinomi. La sortida ha de tenir un format similar al següent:

```
-1.00x^100001  +3.30x^97139  -4.10x^5232 +3012.10x^5131  +1.00x^99
   +21.00x^5
```

L'ordre que té l'índex d'una llista enllaçada es pot utilitzar per a fer cerques de forma més eficient. En el nostre exemple, els monomis estan ordenats per grau. Si volem determinar el coeficient a_k d'un grau concret k (o determinar si apareix en el polinomi) a partir de la llista enllaçada, hem de començar per l'inici i anar mirant el següent fins que trobem el grau=k. Una manera de fer aquesta cerca més efectiva és utilitzar l'índex que hem creat, que sabem que està ordenat pel grau, i fer una cerca binària, que imita el mètode de la bisecció per a trobar zeros de funcions:

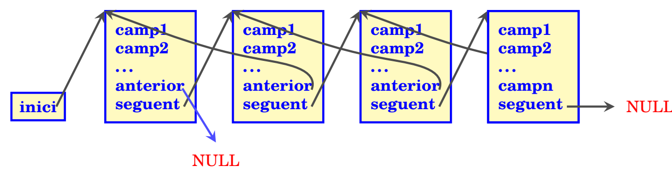
- Incialitzem dos enters `inf=0` i `sup=longitud-1`.
- Mirem el grau de `index[inf]`. Si és k, ja hem acabat. Si és major que k, també hem acabat, treient la conclusió que el grau k no apareix.
- Mirem el grau de `index[sup]`. Si és k, ja hem acabat. Si és menor que k, també hem acabat, treient la conclusió que el grau k no apareix.
- Iterem el procediment següent:
  - Calculem la part entera de (`sup`+`inf`)/2 i la guardem a una variable `pos`.
  - Si el grau de `index[pos]` és k, ja hem acabat.
  - Si el grau de `index[pos]` és menor que k, assignem `inf = pos`.
  - Altrament, el grau de `index[pos]` és més gran que k i assignem `sup = pos`.
- El procés acaba o bé trobant un índex `pos` tal que el grau satisfà `index[pos]->k == k` o bé amb `sup == inf + 1`, cosa que voldria dir que el polinomi no conté terme de grau k.

Exercici 9.6

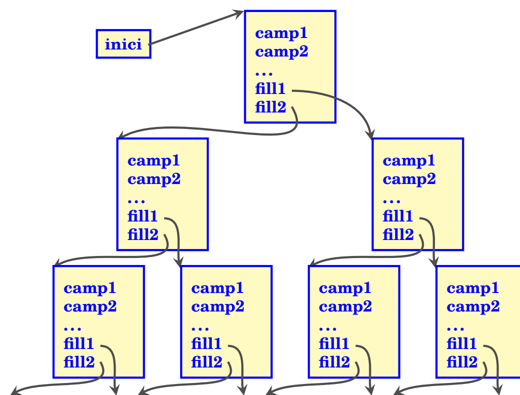
Afegiu a l'exemple que estem treballant (`sumaespars_enllacat.c`) que demani per pantalla un grau i, fent una cerca binària, mostri el coeficient de x^k en el polinomi (o zero si no apareix).

Estructures múltiplement enllaçades

Naturalment, en una estructura hi pot haver més d'un camp de tipus apuntador. Tot i que en aquestes pràctiques no ho farem servir, és convenient conèixer els usos més habituals, que són les llistes doblement enllaçades i els arbres.



En una llista doblement enllaçada, cada node conté un apuntador al node següent i també un apuntador a l'anterior. Això és útil si la feina a realitzar requereix poder recórrer la llista tant endavant com enrere.



En un arbre binari, cada node conté dos apuntadors als seus dos "nodes fills". Allà on aquest arbre s'acaba, els apuntadors valdran NULL. Aquesta estructura admet moltes variants, ja que les diferents "branques" de l'arbre poden tenir la mateixa longitud (es diu que l'arbre és equilibrat) o no, depenent de com s'insereixin o s'esborrin els nodes. Si l'estructura d'arbre reflecteix alguna propietat d'ordre de les dades que s'hi desen, pot ser adequada per a facilitar les cerques binàries, per exemple. Altres variacions es poden obtenir definint arbres on cada node té tres o més nodes fills.

Optimització del compilador

Els compiladors moderns disposen d'algoritmes que permeten analitzar el codi del programa i modificar-lo per obtenir-ne un d'equivalent (és a dir, que faci exactament el mateix) però que s'executa més ràpid. En alguns casos, el compilador pot ser capaç fins i tot de detectar que una funció recursiva és equivalent a un bucle, i crear un executable que evita la recursivitat. El compilador `gcc` disposa de 5 "nivells d'optimització" que se li poden fixar amb una opció en la consola (de `-O0`, que vol dir cap optimització, a `-O4`, que vol dir optimització màxima) o en un menú si fas servir un IDE com Code::Blocks.

Exercici 9.7

Tria algun exercici del curs en què hagis observat que el programa necessita un temps apreciable per fer la feina, i recompila'l amb l'opció `-O3`. Jutja l'efectivitat de l'optimització realitzada.

Quan el compilador optimitza un programa, el codi màquina resultant ja no té per què correspondre's línia a línia amb el que hem escrit. Per això, un programa compilat amb

optimització no es pot *depurar* amb `gdb` ni altres programes similars.
D'altra banda, si establim un nivell alt d'optimització, la informació que el compilador obté de l'anàlisi del nostre codi de vegades es reflecteix en avisos (*warnings*) addicionals relatius a possibles errors o omissions en el programa.

Alguns detalls avançats

Ús d'una pila per evitar la recursivitat

Molt sovint, un algoritme recursiu es pot reescriure sense que la funció es cridi a si mateixa, usant una pila. La idea és que una funció que es crida a si mateixa està de fet iterant un procés, i que això es pot fer en un bucle de manera més eficient. Ara bé, per aconseguir el mateix efecte que la recursivitat, hem de desar l'*estat del procés* d'alguna manera, i la pila farà aquesta funció.

Exercici opcional 9.8

El problema de les torres de Hanoi que vam resoldre de forma recursiva es pot abordar amb aquesta tècnica de forma molt natural. En aquest cas, l'estat que cal desar és la col·lecció de les tres piles de discs en els tres pals del problema. Podem usar l'estructura següent:

```
typedef struct disc
{
unsigned int mida;
struct disc *seguent;
} pila;
```

Caldrà declarar tres piles d'aquest tipus `pilaA`, `pilaB` i `pilaC`, i inicialitzar-les empilant els nombres n, $n-1$, $\ldots$, 1 a la primera pila. Llavors, cada pas del procés consisteix a desempilar un nombre d'una pila i empilar-lo en una altra.

Es pot demostrar que, si n és senar, aleshores en el pas k-èsim

- Si $k \equiv 1 \pmod 3$ juguen les piles A i C.
- Si $k \equiv 2 \pmod 3$ juguen les piles A i B.
- Si $k \equiv 0 \pmod 3$ juguen les piles B i C.

On, quan diem que juguen les piles X i Y pot ser que el pas sigui $X \to Y$ o $Y \to X$, però en cada cas només un dels dos moviments és possible. De fet, el disc que es mou és el menor dels que estan al capdamunt de les piles X i Y. D'altra banda, si n és parell, llavors la regla és:

- Si $k \equiv 1 \pmod 3$ juguen les piles A i B.
- Si $k \equiv 2 \pmod 3$ juguen les piles A i C.
- Si $k \equiv 0 \pmod 3$ juguen les piles B i C.

Aleshores, l'algoritme en el cas senar es pot expressar de la manera següent (on `cim(X)` és una funció que retorna el valor `mida` del disc del cim de la pila):

```
PER i DES DE 1 FINS 2^n-1 {      // 2^n-1 és el nombre total de passos
  SI i % 3 == 1 {
    SI cim(A) > cim(C) O ESBUIDA(A) {
      x = DESEMPILA (C)
      EMPILA (x, A)
    }
    SI NO {
      x = DESEMPILA (A)
      EMPILA (x, C)
    }
  }
  SI NO
  SI i % 3 == 2 {
    SI cim(A) > cim(B) O ESBUIDA(A) {
      x = DESEMPILA (B)
      EMPILA (x, A)
    }
    SI NO {
      x = DESEMPILA (A)
      EMPILA (x, B)
    }
  }
  SI NO{
    SI cim(C) > cim(B) O ESBUIDA(C) {
      x = DESEMPILA (B)
      EMPILA (x, C)
    }
    SI NO {
      x = DESEMPILA (C)
      EMPILA (x, B)
    }
  }
}
```

Programa aquest algoritme iteratiu (i l'anàleg per n parell) per al problema de les torres de Hanoi.

Qüestions d'eficiència

Piles i cues en arrays Tot i que no és difícil programar una pila (o una cua) com una forma de llista encadenada, tal com s'ha fet en aquesta pràctica, aquesta no és la única manera de fer-ho ni és sempre la més eficient. Una alternativa, quan el nombre màxim d'elements a la pila és conegut o no és molt gran, és declarar un `vector` de mida igual al màxim d'elements que es volen desar i una variable entera o apuntador `cim` per marcar el cim de la pila. Llavors, els elements `vector[0]`, ..., `vector[cim]` són els que es troben a la pila. Empilar consisteix en incrementar `cim` i posar un valor al nou `vector[cim]`, mentre que desempilar consisteix en agafar el valor de `vector[cim]` i decrementar `cim`. Això ocupa menys memòria i és més ràpid que la implementació com a llista enllaçada. És possible de fer perquè en aquest cas les insercions es donen només a un extrem de la llista; no és tan fàcil implementar de forma eficient la inserció de nodes en una posició intermèdia d'un vector.

Cerca seqüencial i cerca binària La cerca seqüencial és *terriblement* ineficient amb grans quantitats de dades ($\mathcal{O}(n)$). Les cerques binàries incrementen espectacularment

l'eficiència, ja que en cada pas es redueix a la meitat el rang de valors a comprovar, i per tant el nombre de comparacions a fer és $\mathcal{O}(\log_2(n))$.

No obstant, els compiladors i microprocessadors moderns tenen la capacitat d'optimitzar enormement l'accés seqüencial i els algoritmes senzills amb poques ramificacions. Això fa que en molts casos, amb les quantitats de dades amb què es treballa en la pràctica, una cerca seqüencial compilada amb optimització (opció `-O3` al `gcc`) sigui *més ràpida que la cerca binària*! Això no contradiu el fet anterior: per a qualsevol implementació de cerca binària i de cerca seqüencial, si el nombre de dades `n` és prou gran, la binària serà més eficient; només vol dir que *com de gran* ha de ser `n` perquè la cerca binària valgui la pena depèn de cada cas concret.

10 Biblioteques i fitxers de capçalera

La instrucció `#include <nomfitxer.h>` del *preprocessador* serveix per incorporar al nostre programa un *fitxer de capçalera*. Un fitxer de capçalera és un fitxer font, típicament amb l'extensió `.h`, que conté declaracions i definicions de macros que poden ser compartits per diversos fitxers font. Fins ara, hem fet servir els fitxers de capçalera `stdio.h` (per tenir `printf`, `scanf` i les instruccions de manipulació de fitxers), `math.h` (per tenir les funcions matemàtiques bàsiques) i `stdlib.h` (per a l'assignació dinàmica de memòria).

El *codi* de cadascuna de les funcions que utilitzem (i que no hàgim definit nosaltres) es troba, ja compilat, en algun fitxer de *biblioteca* del sistema. Per exemple, les funcions `printf`, `scanf`, `malloc` (i moltes més) es troben a la *biblioteca estàndard* del C, que s'encarrega de la comunicació amb el sistema operatiu (escriure i llegir de fitxers, reservar memòria, etc, són funcions que controla el sistema operatiu). Quan compilem un programa, automàticament s'enllaça qualsevol funció de les mencionades a la biblioteca estàndard del nostre sistema, que és on hi ha el codi màquina corresponent.

En Linux, moltes de les funcions matemàtiques (per exemple `sqrt`) no es troben a la biblioteca estàndard, sinó en una altra biblioteca, anomenada `libm`. Aquestes funcions tenen el prototipus al fitxer de capçalera `math.h`. Si el nostre programa fa servir `sqrt`, haurem de posar `#include< math.h>` a la capçalera perquè el compilador pugui crear el codi objecte adequat. Però per obtenir un executable, en Linux caldrà també enllaçar el codi que es troba a la biblioteca `libm`, que a diferència de la biblioteca estàndard no s'enllaça per defecte. Per això a la instrucció de compilació cal incloure-hi `-lm`.

En general (en qualsevol sistema operatiu) sempre que hàgim d'usar funcions d'una biblioteca que no sigui la standard, haurem de fer dues coses:

- `#include<nomfitxer.h>` en el fitxer `.c` perquè el compilador tingui accés als prototipus de les funcions.
- `-lnombiblioteca` en la instrucció de compilació (sempre al final) perquè l'enllaçador inclogui el codi compilat de les funcions.

10.1 Biblioteques definides per l'estàndard del C

En C hi ha disponibles molts més fitxers de capçalera i biblioteques, i se'n poden desenvolupar de nous si és necessari. En aquesta pràctica ens familiaritzarem amb alguns dels que tenim disponibles i poden ser útils quan fem matemàtiques.

complex.h

Aquest fitxer de capçalera, igual com `stdio.h` i `stdlib.h`, forma part del C estàndard, tot i que només des del C99. Per aquest motiu pot ser que amb alguns compiladors calgui usar l'opció `-std =C99` o `-std=C11`.

Introdueix tipus de dades anomenats `float complex`, `double complex` i `long double complex` apropiats per tractar amb nombres complexos. Les dues components, real i imaginària, del nombre complex, es desen com a `float`/`double`/`long double` en posicions consecutives de memòria, com si haguéssim declarat

```
typedef struct
{
  double partreal;
  double partimag;
} complex;
```

No obstant, ens podem oblidar d'aquestes interioritats, ja que la sintaxi de `complex.h` ens permet operar directament (sumar, restar, multiplicar i dividir) amb els nombres complexos. La unitat imaginària és una constant anomenada `I` (majúscula).

Exemple 10.1

Deseu el programa següent com `complex.c`, compileu-lo i executeu-lo.

```
// Programa complex.c
// Autor: Professors Dpt Matemàtiques
// Data: Abril 2020
// Exemple bàsic d'ús de complex.h

#include <stdio.h>
#include <complex.h>

int main()
{
   double complex z = 1.0 + 3.0 * I;
   double complex w = 1.0 + -3.0 * I;

   printf("El nombre complex z val %lf%+lfi.\n", creal(z), cimag(z));
   printf("El nombre complex w val %lf%+lfi.\n", creal(w), cimag(w));
   printf("La seva suma val %lf%+lfi.\n", creal(w + z), cimag(w + z));
   printf("El seu producte val %lf%+lfi.\n", creal(w * z), cimag(w * z));

   return 0;
}
```

- Observem que es pot assignar un valor complex directament a una variable complexa. També es poden fer assignacions com `w = z - 1`.
- S'accedeix a les parts real i imaginària d'un nombre complex amb les funcions `creal` i `cimag`.
- En canvi no hi ha un format de `printf` ni `scanf` dedicat als complexos; cal usar un codi de format per a la part real i la part imaginària per separat. El modificador `+` fa que s'imprimeixi el nombre sempre amb el signe que tingui, per això s'ha posat `%+lf` per a la part imaginària.
- La funcionalitat bàsica dels nombres complexos no requereix enllaçar cap biblioteca concreta. per tant la instrucció de compilació habitual (o fer clic al botó corresponent de Code::Blocks) serveix per compilar el programa.

Exercici 10.1

Modifiqueu el programa anterior perquè demani a l'usuari (amb `scanf`) un nombre complex, i en calculi el quadrat, l'invers, el conjugat, el mòdul i l'argument (el mòdul i l'argument es calculen amb les funcions `cabs` i `carg`). Si ho compileu amb Linux necessitareu l'opció `-lm` ja que aquestes funcions es troben a la biblioteca matemàtica.

Exemple 10.2

El programa següent ens servirà per representar l'anomenat *conjunt de Mandelbrot*, un dels conjunts fractals més cèlebres, que ara descriurem. Donat un nombre complex w fixat, considerem la successió de nombres complexos

$$z_0 = 0,$$
$$z_n = z_{n-1}^2 + w, n \geq 1.$$

Per alguns nombres w aquesta successió està acotada (hi ha un valor $R \in \mathbb{R}$ tal que $|z_n| < R$ per a tot n); per exemple, si $w = 0$, és la successió constant igual a zero. Per altres nombres, el mòdul de z_n tendeix a infinit; per exemple, si $w = 1$ llavors z_n és una successió creixent d'enters. El *conjunt de Mandelbrot* és el conjunt dels w per als quals la successió és acotada.

Per als nombres w que *no* són del conjunt de Mandelbrot, això es pot determinar en un nombre finit de passos, ja que és conegut que $\lim |z_n| = \infty$ si i només si per algun n el terme z_n té mòdul més gran que 2. Aleshores, si calculant "molts" termes z_n es mantenen tots de mòdul ≤ 2, podem concloure amb "gran" probabilitat que w pertany al conjunt de Mandelbrot. En particular, el conjunt de Mandelbrot està inclòs dins del "cercle" format pels nombres de mòdul ≤ 2, i el que farem és representar-lo gràficament (de fet, una aproximació) dins el "quadrat" format pels $w = x + yi$ amb $(x, y) \in [-2, 2] \times [-2, 2]$. Més precisament,

1. Dividim aquest quadrat en 1024×1024 quadradets (*píxels*).
2. Per a cada píxel calculem els primers 200 termes de la successió z_n amb w el nombre complex a la cantonada superior esquerra del píxel.
3. Pintem el píxel de blanc si algun $|z_n| > 2$, i de negre en cas contrari.

La part pintada de negre serà una aproximació del conjunt de Mandelbrot.

```
// Programa mandelbrot.c
// Autor: Professors Dpt Matemàtiques
// Data: Abril 2020
// Representa el conjunt de Mandelbrot

#include <stdio.h>
#include <complex.h>

#define xmin -2.
#define xmax 2.
#define ymin -2.
#define ymax 2.

#define pixels 1024
#define maxiter 200

int main()
{
   double increment = (xmax - xmin) / pixels; // Mida del pixel
   complex double w, z;
   double x, y; // part real / imaginària de w
   int k;

   char* nomfitxer = "mandelbrot.ppm";
   FILE * fitxer = fopen(nomfitxer, "wb");
   unsigned char negre[] = {0, 0, 0};
   unsigned char blanc[] = {255, 255, 255};
```

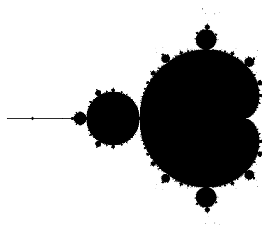
```
    // Els fitxers ppm tenen una capçalera amb metadades
    // P6 identifica el tipus de fitxer
    // Una línia que compença per # és un comentari
    // Tres nombres determinen l'amplada en pixels, l'alçada en pixels
    // i el valor màxim d'intensitat en vermell, verd i blau
    fprintf(fitxer, "P6\n# Conjunt de Mandelbrot\n%d\n%d\n%d\n",
            pixels, pixels, 255);

    for(y = ymax; y > ymin; y = y - increment) {
        for(x = xmin; x < xmax; x = x + increment) {
            w = x + y * I;
            z = w;
            for(k = 1; k < maxiter && (cabs(z) < 2.0); k++)
                z = z * z + w;
            if(k >= maxiter)  // Dins del conjunt de Mandelbrot
                fwrite(negre, 3, 1, fitxer);
            else // Fora del conjunt de Mandelbrot
                fwrite(blanc, 3, 1, fitxer);
        }
    }

    fclose(fitxer);
    return 0;
}
```

- El resultat d'aquest programa es desa en un fitxer gràfic, `mandelbrot.ppm`, que pots obrir amb un programa de manipulació de gràfics i tornar-lo a desar en format `png`, per exemple. Com que el format del fitxer és binari, l'obrim en mode `"wb"`, i hi escrivim amb la instrucció `fwrite` els 3 bytes `{0,0,0}` o `{255,255,255}` segons si volem pintar el píxel de negre o de blanc.

- Els píxels estan ordenats per files (com una matriu) començant per dalt a l'esquerra. Per això hi ha els dos bucles que modifiquen les variables `y` (part imaginària de w, baixant de 2 a -2) i `x` (part real de w, d'esquerra a dreta de -2 a 2).

- El tercer bucle, indexat amb `k`, calcula els primers 200 termes (com a màxim) de la successió z_n. Si en algun moment el mòdul (`cabs(z)`) de z_n és superior a 2, el bucle s'atura i `k` no arriba a valdre 200; aquest és el criteri de l'`if` que decideix de quin color pintar el píxel.

Un cop compilat i executat el programa, heu d'obtenir un fitxer `mandelbrot.ppm`, que obert amb qualsevol programa gràfic té aquest aspecte:



Exercici 10.2

Algunes parts de la frontera del conjunt de Mandelbrot resulten molt interessants i atractives. Modifica el programa anterior per fer que l'usuari pugui escollir el rectangle representat, i també la quantitat de píxels en què es divideix. Prova'l per veure la part del conjunt de Mandelbrot dins del rectangle dels $w = x + yi$ on $(x, y) \in [0.27085, 0.27100] \times [0.004640, 0.004810]$.

Algunes representacions del conjunt de Mandelbrot aconsegueixen un aspecte més vistós representant de colors diferents els valors w de fora del conjunt segons el nombre d'iteracions k que calen per obtenir un mòdul més gran que 2. Modifica el programa perquè en comptes del blanc (255,255,255) s'escrigui al fitxer un color que depengui de `k`. (El més senzill és escriure el valor (k, k, k) però això sempre donarà matisos de gris. Si ho vols acolorit, la intensitat de vermell, verd i blau no poden ser iguals).

Quan estiguis satisfet amb l'aspecte visual, desa el nou programa amb el nom `mandelbrot2.c`.

Exercici 10.3

Si incloem el fitxer de capçalera `tgmath.h` (i l'opció `-lm` en compilar amb Linux) podem usar les funcions matemàtiques (`exp`, `sin`, `cos`, `sqrt`, etc.) en el domini complex també. Podem així representar altres conjunts fractals dins $\mathbb{C}$, com conjunts de Julia associats a funcions qualssevol.

Si $f : \mathbb{C} \to \mathbb{C}$ és una funció complexa, podem definir per a cada nombre complex w una successió fent:

$$z_0 = w$$
$$z_n = f(z_{n-1}), n \geq 1.$$

De forma similar al conjunt de Mandelbrot, el conjunt de Julia de f està format per aquells w tals que la successió és fitada.

Escriu un programa per representar els conjunt de Julia de la funció $f(z) = z^2 - 0.618$. Després modifica'l per representar els de les funcions $f(z) = \sqrt{z^3 - 0.187}$ i $f(z) = \sqrt{z^5 - 0.861}$.

time.h

Hi ha molts programes de càlculs intensius en què el temps d'execució és un factor limitant crític. A l'hore d'escriure un programa és important tenir en compte el temps d'execució, quan escollim les instruccions precises que executaran l'algoritme desitjat. Quan es fan milions de càlculs per assolir un objectiu, escollir la millor opció pot provocar que la diferència en el temps d'execució entre dos programes diferents sigui de minuts, hores, dies, setmanes, mesos, anys o segles. La capçalera `time.h` inclou diferents funcions que en particular ens permeten saber amb precisió el temps emprat en executar un codi concret. D'aquesta manera podem fer provatures per decidir si certs canvis en el codi seran útils o no per fer-lo més ràpid.

Exemple 10.3

Ho il·lustrem amb el següent exemple, on es compara el temps emprat en inicialitzar una matriu i presentar-la per pantalla, segons si aquesta matriu es tracta com un `array` 1 o 2-dimensional o si se li assigna memòria dinàmicament de diferents maneres. La tasca feta per les matrius `M1`, `M2`, `M3` i `M4` és la mateixa, però potser una manera de fer-ho és més simple, i una altra més ràpida?

```
// Programa time_matrius.c
// Autor: Professors Dpt Matemàtiques
// Data: Abril 2020
```

```
// Comparació de temps d'execució segons mètode per tractar amb matrius.

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

const unsigned long int dim = 100;

int main()
{
   int i, j;
   clock_t t0, t1, t2, t3, t4;
   t0 = clock();

   double M1[dim * dim];
   for(i = 0; i < (dim * dim); i++) {
      M1[i] = rand() / 1000.;
      printf("M1[%d]=%4.2lf\n", i, M1[i]);
   }
   t1 = clock();
   printf("temps execució matriu M1:  %lf segons\n\n\n",
          (double)(t1 - t0) / CLOCKS_PER_SEC);


   double M2[dim][dim];
   for(i = 0; i < dim; i++) {
      for(j = 0; j < dim; j++) {
         M2[i][j] = rand() / 1000.;
         printf("M2[%d][%d]=%4.2lf\n", i, j, M2[i][j]);
      }
   }
   t2 = clock();
   printf("temps execució matriu M2:  %lf segons\n\n\n",
          (double)(t2 - t1) / CLOCKS_PER_SEC);

   double (*M3) [dim];
   M3 = (double (*)[dim]) malloc(dim * dim * sizeof(double));
   if(M3 == NULL) {
      printf("No hi ha prou memòria per a M3\n");
      return 1;
   }
   for(i = 0; i < dim; i++) {
      for(j = 0; j < dim; j++) {
         M3[i][j] = rand() / 1000.;
         printf("M3[%d][%d]=%4.2lf\n", i, j, M3[i][j]);
      }
   }
   t3 = clock();
   printf("temps execució matriu M3:  %lf segons\n\n\n",
          (double)(t3 - t2) / CLOCKS_PER_SEC);

   double **M4;
   M4 = (double **)malloc(dim * sizeof(double *));
   if(M4 == NULL) {
      printf("No hi ha prou memòria per a M4\n");
      return 1;
   }
   for(i = 0; i < dim; i++) {
      M4[i] = (double *)malloc(dim * sizeof(double));
      if(M4[i] == NULL) {
         printf("No hi ha prou memòria per a M4\n");
```

```
            return 1;
        }
    }
    for(i = 0; i < dim; i++) {
        for(j = 0; j < dim; j++) {
            M4[i][j] = rand() / 1000.;
            printf("M4[%d][%d]=%4.2lf\n", i, j, M4[i][j]);
        }
    }
    t4 = clock();

    printf("temps execució matriu M1:  %lf segons\n",
           (double)(t1 - t0) / CLOCKS_PER_SEC);
    printf("temps execució matriu M2:  %lf segons\n",
           (double)(t2 - t1) / CLOCKS_PER_SEC);
    printf("temps execució matriu M3:  %lf segons\n",
           (double)(t3 - t2) / CLOCKS_PER_SEC);
    printf("temps execució matriu M4:  %lf segons\n",
           (double)(t4 - t3) / CLOCKS_PER_SEC);

    return 0;
}
```

- `time.h` introdueix el tipus de variable `clock_t` i la funció `clock` que retorna el nombre de *ticks* de rellotge transcorreguts; els ticks són una durada constant (però que pot dependre del sistema) i que es poden convertir a segons dividint per la constant `CLOCKS_PER_SEC`.

- Si compiles i executes el programa repetidament, observaràs petits canvis en els temps mesurats. Per obtenir una estimació correcta dels temps val la pena fer la mesura diverses vegades i calcular una mitjana. Fet això, podem decidir si la diferència de temps és un factor decisiu a l'hora d'escollir un dels quatre mètodes. La conclusió només s'aplica a aquest programa concret; si es fan altres càlculs amb la matriu la resposta pot ser una altra i caldria refer l'experiment. També pot ser interessant modificar els valors de la variable dim i observar com canvien els temps d'execució. Recorda de la pràctica 6 que els mètodes amb assignació dinàmica de `M3` i `M4` poden funcionar per mides més grans, i aquest pot ser un criteri més rellevant que el temps d'execució!

- Si executes el programa des de la consola (en Linux o Mac) pots mesurar el temps *total* d'execució amb la instrucció `time`. En aquest cas, podries fer `time ./time_matrius`. En Windows, obrint el programa des de la consola `PowerShell` pots fer `Measure-Command { time_matrius .exe }` Si l'executes des d'un IDE com Code::Blocks, pot ser que per defecte ja se't presenti un temps total d'execució.

  Observa que hi ha una petita diferència entre aquest temps que et proposa la consola i el temps total que es dedueix de la sortida del programa, degut a què la pròpia consola necessita un temps per obrir i tancar el programa, accedir a la pantalla, etc.

10.2 Biblioteques matemàtiques de codi obert

A més de les biblioteques que formen part del C estàndard, al llarg dels anys la comunitat de programadors ha posat a la disposició de tots una sèrie de biblioteques en codi obert que poden ser d'utilitat. N'hi ha una llista disponible a `https://en.cppreference.com/w/c/links/libs`. A diferència de les biblioteques que hem fet servir fins ara, no hi ha cap garantia que aquestes

biblioteques estiguin instal·lades en el vostre sistema. Si no ho estan, utilitzar-les pot ser més o menys senzill segons el cas.

En aquesta secció mencionem dos exemples que provenen del projecte GNU. Si treballes amb GNU/Linux, probablement tens instal·lades aquestes biblioteques o estan disponible en el paquet adequat (en Ubuntu, els paquets s'anomenen `libgsl-dev` i `libgmp3-dev`). Si uses Windows la presència d'aquestes biblioteques dependrà del sistema d'instal·lació que hagis fet servir. El programa `MSYS2` conté, a més d'una versió actualitzada del compilador `gcc`, un sistema de gestió de paquets anomenat `pacman` que permet instal·lar de forma senzilla biblioteques addicionals. En aquesta última part de la pràctica suposarem que tens instal·lades les biblioteques `gmp` i `gsl`.

gmp.h

Segurament, en fer programes com el càlcul del factorial, t'ha resultat frustrant no poder calcular el factorial de nombres més grans que 25; o potser en algun moment t'has preguntat si no seria possible guardar nombres reals amb una precisió més gran que un `long double`. Després de tot, SageMath i Python bé que ho permeten. La biblioteca `gmp` permet treballar amb enters arbitràriament grans, i amb nombres en punt flotant amb precisió prefixada. A diferència de `complex.h`, però, els tipus de nombres definits en `gmp.h` en realitat estan construïts com vectors reservats en memòria dinàmica, i no permeten assignacions directes ni l'ús dels operadors `+, -, *, /`. En comptes d'aquests, cal usar *funcions* per a *operar*, i és necessari també *inicialitzar* els nombres (per reservar la memòria necessària).

Exemple 10.4

El programa següent és una modificació del `factorial.c` de l'exercici 5.16, que fa el càlcul en una variable de tipus `mpz_t` ("multiprecisionℤ_type"). Per compilar-lo des de línia de comandes haureu d'escriure

```
gcc -o factorial_gmp factorial_gmp.c -lgmp
```

on la última opció `-lgmp` és necessària per enllaçar el codi de la biblioteca `gmp`.

```c
// Factorial_gmp.c
// Autor: Professors Dpt Matemàtiques
// Data: Abril 2020
// Càlcul de factorials grans

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

int main(int argc, char *argv[])
{
   unsigned int n;

   mpz_t factorial;
   mpz_init(factorial);

   if(argc == 1) {
      printf("Ús: ./factorial_gmp n retorna el factorial del nombre n\n")
   ;
      return 1;
   }
   n = atoi(argv[1]);

   printf("El factorial de %u és ", n);
```

```
    for(mpz_set_ui(factorial, 1); n > 1; n--)
        mpz_mul_ui(factorial, factorial, n);

    gmp_printf("%Zd\n",factorial);

    mpz_clear(factorial);
    return 0;
}
```

- Observem que immediatament després de declarar la variable `factorial` de tipus `mpz_t` és necessari inicialitzar-la amb `mpz_init`.

  Per donar-li el valor inicial 1 que després anirem multiplicant per tots els enters menors o iguals a n cal usar una altra funció, en aquest cas `mpz_set_ui`. És possible combinar aquesta instrucció amb la inicialització, fent `mpz_init_set_ui(factorial,1);`

  Quan hem acabat d'usar la variable, fem `mpz_clear(factorial)`; com que de fet la variable usa assignació dinàmica de memòria, cal alliberar aquesta memòria.

- El mateix efecte d'assignar valor a una variable `mpz_t` es pot aconseguir a partir d'una cadena, fent `mpz_set_str(factorial,"1",10);` Com que la cadena pot ser arbitràriament llarga, es poden assignar valors que no cabrien en variables ordinàries de C, com podria ser, per exemple, `mpz_set_str (pi, "31415926535897932384626433832795028 84", 10);` El paràmetre `10` indica la base en què s'ha escrit el nombre.

- La instrucció `mpz_mul_ui` fa una multiplicació d'un nombre de mida arbitrària amb un `unsigned int`. El prototipus d'aquesta funció és

```
void mpz_mul_ui (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
```

  i el que fa és posar en `ROP` el resultat de multiplicar `OP1` per `OP2`. Com podeu suposar, també hi ha funcions per a sumar, restar, i fer la divisió entera d'enters de tipus `mpz_t` i entre aquests i enters ordinaris de C. Podeu trobar tota la informació a `https://gmplib.org`.

- Aquests enters no es poden presentar amb `printf`, per això hi ha una instrucció pròpia `gmp_printf`. Seria equivalent la instrucció `mpz_out_str(stdout, 10, factorial);` que permet escriure en un fitxer (canviant `stdout` per la nansa corresponent) i escriure en una base arbitrària (canviant `10` per la base a utilitzar).

Exercici 10.4

En l'exemple anterior, el valor n del qual volem el factorial es desa en una variable `unsigned int`. Això imposa un límit al nombre més gran del qual el programa `factorial_gmp` pot calcular el factorial. Canvia la declaració de n per `mpz_t n;` i fes tots els canvis necessaris perquè segueixi calculant el factorial. La funció de multiplicar que necessitaràs és `mpz_mul`. Desa aquest programa amb el nom `factorial_gmp2`.

Una reflexió: Quin és el nombre més gran del qual es pot calcular el factorial amb el programa `factorial_gmp` original? Mesura amb `clock` el temps que necessita cadascun dels dos programes per calcular uns quants factorials. Decideix si té cap interès, en la pràctica, canviar el tipus de n a `unsigned long long int` o a `mpz_t`.

gsl

La biblioteca gsl (GNU Scientific Library) proporciona una col·lecció de tipus de variables i funcions matemàtiques associades, que van des de generadors de nombres aleatoris o funcions per a l'Àlgebra Lineal fins a rutines per a l'estadística com l'estimació de paràmetres pel mètode dels mínims quadrats. Són més de mil funcions que podeu trobar descrites a la web https://www.gnu.org/software/gsl, i que es poden classificar temàticament en els següents àmbits:

Nombres complexos	Arrels de polinomis	Vectors i matrius
Funcions especials	Permutacions	Ordenació
Àlgebra Lineal	Transformada ràpida de Fourier	Quadratura
Nombres aleatoris	Distribucions aleatòries	Estadística
Optimització	Equacions diferencials	Derivació numèrica
Interpolació	Aproximació de Chebyshev	Acceleració de sèries
Transformades discretes de Hankel	Cerca d'arrels	Punt flotant IEEE
Constants físiques	Transformades discretes d'ondeta	B-splines

No treballarem amb aquestes funcions en el curs, i en la majoria de sistemes la llibreria no hi està instal·lada per defecte. Perquè us feu una idea seu funcionament, vegem un exemple de com es poden manipular permutacions en gsl. El tipus bàsic de variable associat s'anomena gsl_permutation, i és de fet un struct:

```
typedef struct
{
long int size;
long int * data;
} gsl_permutation;
```

La component size ha de contenir la mida de la permutació, és a dir que si volem representar $\sigma \in S_n$, el valor de size serà n. La component data és un apuntador a l'adreça on hi haurà la permutació en si, representada com un vector format pels valors $\{0, 1, \ldots, n-1\}$ en l'ordre que correspon a la permutació.

Exemple 10.5

El programa següent descompon una permutació en cicles (per exemple, la permutació $(2,4,3,0,1)$ de S_5 es descompon com a producte dels dos cicles $(0\ 2\ 3)(1\ 4)$). Per a fer-ho, utilitza la funció predefinida gsl_permutation_linear_to_canonical. Per representar els cicles fa servir una variable de tipus gsl_permutation en la que

1. Els elements fixos es representen tots explícitament com cicles d'ordre 1 (per exemple: (0)).
2. En cada cicle, s'escriu com a primer element el més petit.
3. Els cicles s'ordenen en ordre decreixent del primer element.

Així, quan trobem en la successió de nombres un valor inferior a l'anterior, sabrem que comença un cicle. la identitat de S_5 es representaria per la successió $[4,3,2,1,0]$. La permutació $(2,4,3,0,1)$ es representaria com $[1,4,0,2,3]$. Cal compilar el programa amb la instrucció

```
gcc -o permutacions permutacions.c -lgsl -lgslcblas
```

```
#include<stdio.h>
#include<gsl/gsl_permutation.h>

int main()
```

```
{
   long unsigned int n;
   gsl_permutation * sigma;
   gsl_permutation * cicles;

   printf("De quina mida és la teva permutació? ");
   scanf("%lu", &n);

   sigma = gsl_permutation_alloc(n);
   cicles = gsl_permutation_alloc(n);
   if(sigma == NULL || cicles == NULL) {
      printf("No s'ha pogut crear la permutació.\n");
      return 1;
   }

   printf("S'ha creat correctament una permutació d'ordre %lu. Introdueix
    les seves components: ",
          sigma->size);
   for(long unsigned int i = 0; i < sigma->size; i++)
      scanf("%lu", sigma->data + i);

   gsl_permutation_valid(
      sigma); // Comprova que els nombres introduïts defineixen una
   permutació.

   gsl_permutation_linear_to_canonical(cicles, sigma);

   printf("La descomposició en cicles de la permutació és:\n(");
   for(long unsigned int i = 0; i < cicles->size; i++) {
      printf("%lu", cicles->data[i]);
      if(i < cicles->size - 1 && cicles->data[i] > cicles->data[i + 1])
         printf(")(");
   }
   puts(")");
   return 0;
}
```

Alguns detalls avançats

Creació d'una biblioteca

No és estrany que, en escriure un programa, t'adonis que algunes de les funcions que escrius podrien ser útils de forma general en altres programes que facis en el futur. Per exemple, potser has escrit algun codi per treballar amb matrius. És molt probable que en algun moment et torni a fer falta. En aquest cas, és bona idea posar totes aquestes funcions en un fitxer a part, deslligat del fitxer on hi ha la funció main.

Exercici opcional 10.5

Copia les funcions `matriuxvector` i `imprimeixmatriu` del programa `matriuxvector` que vas fer en l'exercici 3.6 en un nou fitxer anomenat `func_matrius.c`. Copia també en aquest arxiu les funcions `llegeixvector` i `llegeixmatriu` fets en l'exercici 3.5. Com que aquestes funcions utilitzen `printf` i `scanf`, posa a l'inici la línia `#include<stdio.h>`.
Copia *només els prototipus* de les mateixes funcions en un nou fitxer *de capçalera* (header file), anomenat `func_matrius.h`.
Ara fes un nou programa, `iteramatriu.c`, en el qual podràs fer servir les funcions de

`func_matrius` si inclous la línia

```
#include "func_matrius.h"
```

El programa li ha de preguntar una matriu A i un vector v a l'usuari, i després un enter k, i ha de calcular el producte $A^k v$ pel procediment de trobar $v' = Av$, $v'' = Av'$, etcètera. No és necessari guardar en memòria tots els iterats, només imprimir v i $A^k v$.
Ves amb compte; probablement la funció `matriuxvector` no funcionarà bé si li fas posar el resultat del càlcul al mateix vector que estàs multiplicant.
Compila el programa amb la instrucció

```
gcc -Wall -o iteramatriu iteramatriu.c func_matrius.c
```

Observació: Tot i que en aquest cas no és estrictament necessari, seria recomanable incloure al fitxer `func_matrius.c` també l'`#include "func_matrius.h"`, així les funcions d'aquest fitxer es poden cridar mútuament sense importar l'ordre en què estan definides. No obstant, si diversos fitxers `.c` carreguen el mateix fitxer de capçalera, podem acabar tenint definicions duplicades i errors. Per evitar-ho, incloem al fitxer de de capçalera una definició que té per única finalitat detectar si ja ha estat carregat. Posa les següents dues línies a dalt de tot del fitxer `func_matrius.h`:

```
#ifndef FUNC_MATRIUS_H     // El que ve a sota només val si NO s'ha
   fet abans
#define FUNC_MATRIUS_H
```

i com a última línia ,

```
#endif
```

Aplicació: Tenim un model simplificat per predir el nombre d'alumnes en cada curs d'un cert grau al llarg dels anys, de la manera següent. Anomenem

$$\begin{aligned}
x_0 &= \text{el nombre d'estudiants que han entrat via PAU } = 100\\
x_1 &= \text{el nombre d'estudiants que estan fent primer}\\
x_2 &= \text{el nombre d'estudiants que estan fent segon}\\
x_3 &= \text{el nombre d'estudiants que estan fent tercer}\\
x_4 &= \text{el nombre d'estudiants que estan fent quart}
\end{aligned}$$

Aleshores, els nombres $(x'_0, x'_1, x'_2, x'_3, x'_4)$ que descriuen la situació esperada el proper any es calculen com

$$\begin{aligned}
x'_0 &= x_0 && \text{(l'entrada d'estudiants és constant)}\\
x'_1 &= x_0 + 0.2x_1 && \text{(un 20\% d'estudiants repeteix primer)}\\
x'_2 &= 0.6x_1 + 0.15x_2 && \text{(un 60\% de primer passen a segon, un 15\% repeteix a segon)}\\
x'_3 &= 0.8x_2 + 0.08x_3 && \text{(un 80\% de segon passen a tercer, etc.)}\\
x'_4 &= 0.9x_3 + 0.05x_4
\end{aligned}$$

(Els nombres són totalment ficticis, però es podrien modificar perquè el model s'ajusti a la realitat d'un grau concret). Tradueix (a mà) aquestes equacions en una igualtat matricial

$$\begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = A \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

i utilitza el teu programa per explorar si el nombre d'estudiants a cada curs s'estabilitza amb el pas dels anys (pots començar amb $x_1 = \cdots = x_4 = 0$ o inventar-te una situació inicial).

En l'exercici anterior, la instrucció de compilació fa que cada vegada que es compila el teu nou programa també es compilin les funcions `llegeixmatriu`, `escriumatriu` i `vectorxmatriu` del fitxer `func_matrius.c`. Si aquest fitxer contingués *moltes* funcions o fossin molt complexes, haver-les de compilar cada vegada podria ser una despesa supèrflua de temps i recursos de l'ordinador. Podria valdre la pena deixar aquestes funcions, ja compilades, en una biblioteca creada per tu, i enllaçar-la amb una opció `-lfunc_matrius` del linker. Ara aprendrem a fer això[a].

Exercici opcional 10.6

El fitxer `func_matrius.c` no conté una funció `main`, i per tant no es pot compilar per obtenir un programa executable. No obstant, les funcions que defineix es poden compilar per obtenir el codi objecte corresponent, i crear el fitxer de biblioteca, amb les instruccions següents. Noteu que els fitxers de biblioteca en Linux duen l'extensió `.so` (shared object), i en Windows l'extensió `.dll` (dynamic link library):

```
gcc -Wall -c -fpic func_matrius.c
gcc -shared -o libfunc_matrius.so func_matrius.o      # (en Linux o
   mac)
gcc -shared -o libfunc_matrius.dll func_matrius.o     # (en Windows)
```

La primera instrucció genera un fitxer objecte `func_matrius.o`, a partir del qual la segona crea el fitxer de biblioteca pròpiament dit. És important que el nom del fitxer de biblioteca comenci per `lib`, ja que l'enllaçador busca noms de biblioteca amb aquest inici (que s'omet en la instrucció del compilador). Aquest fitxer haurà d'estar disponible no només per compilar qualsevol programa que la faci servir, sinó també en el moment d'executar-se aquest programa, en un directori conegut pel sistema. Com es fa això pot ser diferent segons els sistemes operatius.

Windows En Windows, és suficient que el fitxer `.dll` estigui al mateix directori que el programa executable. Per tant, ara ja pots compilar el programa així:

```
# Des de la consola de Windows:
gcc -L . -Wall -o iteramatriu.exe iteramatriu.c -lfunc_matrius
```

i se t'executarà com de costum. La última opció, `-lfunc_matrius`, és la que fa enllaçar la biblioteca `libfunc_matrius.dll` (sense que calgui tornar-la a compilar); naturalment, en qualsevol cas el teu programa ha de seguir contenint la línia `#include "func_matrius.h"` i el fitxer de capçalera `func_matrius.h` ha de ser present al mateix directori.

Linux Tot i que hi pot haver maneres alternatives de fer-ho segons la distribució, la descripció que donarem és vàlida per a les distribucions de Linux més usuals.
És una bona idea col·leccionar totes les biblioteques que programis en un mateix directori; per als següents passos suposarem que el teu *home* és a `/home/nomusuari/` i que has creat un directori anomenat `/home/nomusuari/lib/` per posar-hi les teves biblioteques; substitueix aquests noms de camí, en les instruccions següents, pels que s'apliquin en el teu ordinador. Suposarem que ja has copiat el fitxer de biblioteca `libfunc_matrius.so` a `/home/nomusuari/lib/`

Opció 1 Si només tu faràs servir les biblioteques de /home/nomusuari/lib/, una forma fàcil de dir al sistema que ha de buscar les teves biblioteques en aquest directori és afegir una línia al fitxer .bashrc que es troba al teu home. Per exemple, executant aquesta instrucció des de la consola, només la primera vegada que poses una biblioteca al directori /home/nomusuari/lib/.

```
# Canviar el directori "/home/nomusuari/lib" pel que correspongui
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/nomusuari/lib/'
   >> ~/.bashrc
```

Opció 2 Si vols que les biblioteques de /home/nomusuari/lib/ estiguin disponibles per a tots els usuaris de l'ordinador, és només un xic més complicat però necessitaràs ser administrador (*sudoer*). La primera vegada que hagis posat una biblioteca en aquest directori, executa les instruccions següents a la consola:

```
# Canviar el directori "/home/nomusuari/lib" pel que correspongui
sudo echo "/home/nomusuari/lib" > /etc/ld.so.conf.d/mevesLibs.conf
sudo ldconfig
```

Si més endavant poses noves biblioteques al directori escollit, només necessitaràs tornar a executar sudo ldconfig.

A més, podries plantejar-te posar les biblioteques a /usr/lib/mevesLibs/ i no dins el teu home (modificant les línies anteriors perquè /etc/ld.so.conf.d/meveslibs.conf apunti a aquest directori) i també posar el fitxer de capçalera a /usr/include/mevesh/.

Un cop fets aquests canvis en el sistema, cada vegada que necessitis compilar un programa (com iteramatriu.c) que faci referència a les funcions de la biblioteca func_matrius ho podràs fer amb la instrucció

```
gcc -L/home/nomusuari/lib/ -Wall -o iteramatriu iteramatriu.c -
   lfunc_matrius
```

La última opció, -lfunc_matrius, és la que fa enllaçar la biblioteca libfunc_matrius.so (sense que calgui tornar-la a compilar); naturalment, en qualsevol cas el teu programa ha de seguir contenint la línia #include "func_matrius.h" i el fitxer de capçalera func_matrius.h ha de ser present al mateix directori.

Usar funcions de C des de SageMath

Com hem tingut ocasió de comprovar al llarg del curs, en el cas de necessitar l'ordinador per fer algun càlcul matemàtic, normalment és més senzill obrir SageMath i usar alguna de les instruccions ja existents que programar-ho en C. No obstant, us trobareu en ocasions que SageMath no disposa de la funció que necessitaríeu, i programar dins de SageMath (o Python) a vegades resulta en codi massa lent per a les nostres necessitats. En aquesta situació, pensarem en el C, però el que voldríem seria poder usar amb la facilitat de SageMath unes funcions prèviament compilades en C, i no haver de fer tota la feina en C. En aquesta secció veurem com usar una biblioteca .so o .dll com les creades abans des de dins de SageMath o Python.

Exemple 10.6

El primer que farem serà comprovar com es crida la funció `matriuxvector` de la biblioteca `func_matrius` des de SageMath. Òbviament això no té gaire interès en si mateix, ja que SageMath disposa de funcions pròpies molt més ben elaborades per operar amb matrius, però veurem quina és la sintaxi que ens cal.
Obriu una consola de SageMath (per exemple, un notebook de `jupyter`). Executeu les dues línies següents:

```
from ctypes import *
libmat = CDLL("libfunc_matrius.so")    # En Linux
libmat = CDLL("libfunc_matrius.dll")   # En Windows
```

La primera instrucció carrega el paquet `ctypes` que ens dona accés als tipus de variables del C, necessaris per poder enviar i rebre dades de les funcions de la nostra biblioteca. La segona, carrega la biblioteca pròpiament. Si ara escriviu `libmat.matriuxvector` hauríeu de rebre una resposta del tipus

```
<_FuncPtr object at 0x...>
```

cosa que indica que la funció `matriuxvector` de la nostra biblioteca ha estat carregada! Ara necessitem saber el prototipus d'aquesta funció, per poder-la cridar. Recordem que era:

```
void matriuxvector (int dim, double [][dim], double[], double[]);
```

Des de SageMath, un enter de C s'ha d'anomenar `c_int`, un double, `c_double`, etcètera. Farem servir la nostra funció per a fer el producte Mv on

$$M = \begin{pmatrix} 1.12 & 2.21 \\ 3.45 & 5.43 \end{pmatrix}, \quad v = \begin{pmatrix} 0.33 \\ 3.00 \end{pmatrix}$$

El primer que hem de fer és definir la matriu i el vector i convertir-los en el format de C. Executeu en la finestra de SageMath el codi següent:

```
M = matrix([[1.12,2.21],[3.45,5.43]])
v = vector([0.33, 3.00])
show(M)
show(v)

c_vec2 = c_double * int(2)
c_v = c_vec2 (v[0],v[1])

c_mat2x2 = c_double * int(4)
c_M = c_mat2x2(M[0,0],M[0,1],M[1,0],M[1,1])

c_w = c_vec2()
```

Un array de C de n components (en aquest cas $n = 2$) es crea en SageMath (o Python) fent el "producte" del tipus corresponent (en aquest cas `c_double`) per `int(n)`. Les línies anteriors creen el tipus `c_vec2` (vector de C de 2 components) i `c_mat2x2` (vector de C de 4 components) i la matriu `M` la convertim en un vector `c_M` de C de quatre components, el vector `v` en un vector `c_v` de C de dues components, i definim un vector `c_w` on posarem el resultat. Ara ja podem cridar la funció:

```
libmat.matriuxvector(c_int(2),c_M,c_v,c_w)
w=vector(c_w)
show(w)
```

La nostra funció ha posat el producte a `c_w`, i després l'hem tornat a convertir en un vector de SageMath. Si per altra banda fem el càlcul directament en SageMath escrivint `M*v`, veurem que els resultats coincideixen, confirmant que hem cridat la nostra funció amb èxit.

Exercici opcional 10.7

Crea una biblioteca `hofstadter_conway` que contingui la funció `hc` de l'exercici 3.4, obre SageMath i defineix dins SageMath una funció que calculi el terme n-èsim de la successió de Hofstadter-Conway cridant la funció de la teva biblioteca.

[a] Tècnicament caldria distingir entre *biblioteques estàtiques* i *biblioteques dinàmiques*, però la distinció i els avantatges de cada tipus queden fora dels nostres objectius. Crearem una biblioteca dinàmica per després poder-la enllaçar des de SageMath, per exemple (crear-ne una d'estàtica és més senzill i no us costarà aprendre a fer-ho si mai ho necessiteu).