

TEMA 1.

CONCEPTES FONAMENTALS.

(PART 1)

TEORIA.

FONAMENTS DE PROGRAMACIÓ II

CURS: 2024-25

GRAUS: GEI, GEI-Biotec

UNIVERSITAT ROVIRA I VIRGILI

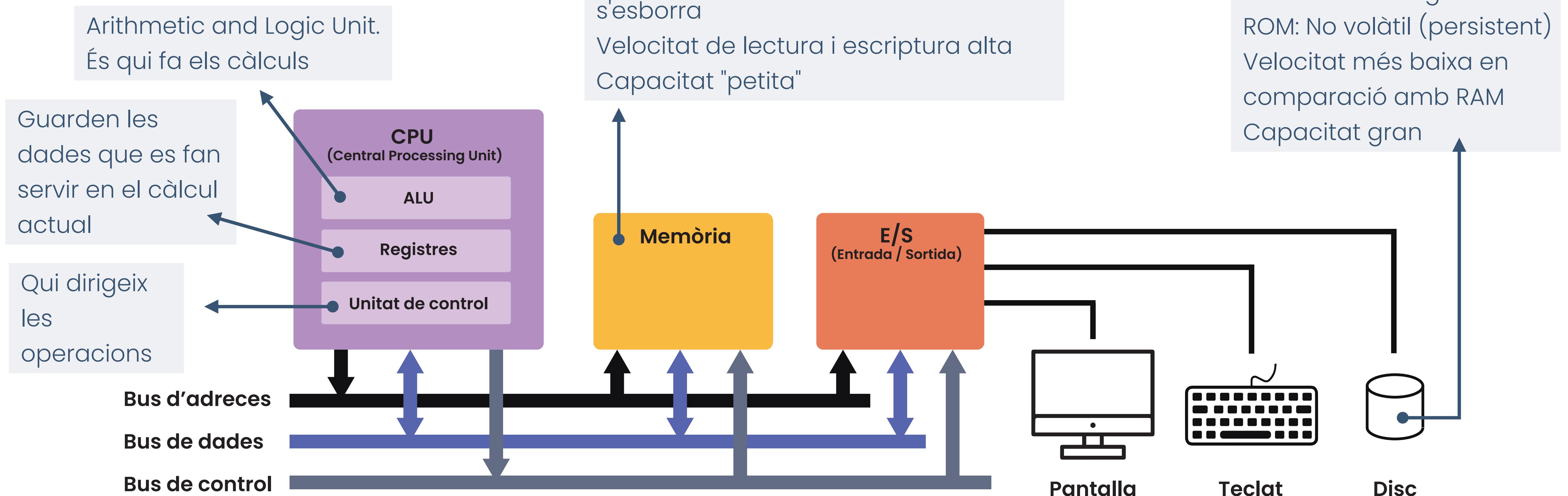
COMENCEM PEL COMENÇAMENT

Estructura d'un computador



John Von Neumann

Arquitectura de Von Neumann



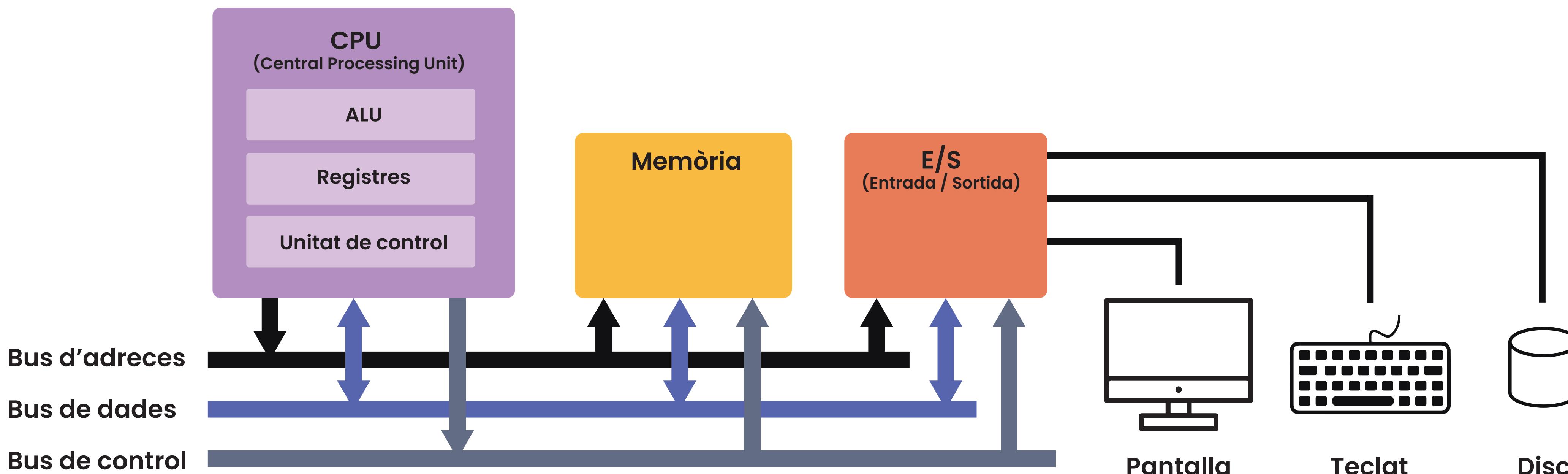
Estructura d'un computador



John Von Neumann

Arquitectura de Von Neumann

- Per executar un programa (que està a disc) l'hem de carregar a memòria RAM.
- Per tant, totes les instruccions i dades que necessitem pel programa estan a memòria RAM.
- Per tant, necessitem entendre bé com funciona la memòria.
- A *Fonaments de Computadors* veureu el cicle *Fetch & Execute* amb detall.



Estructura d'un computador

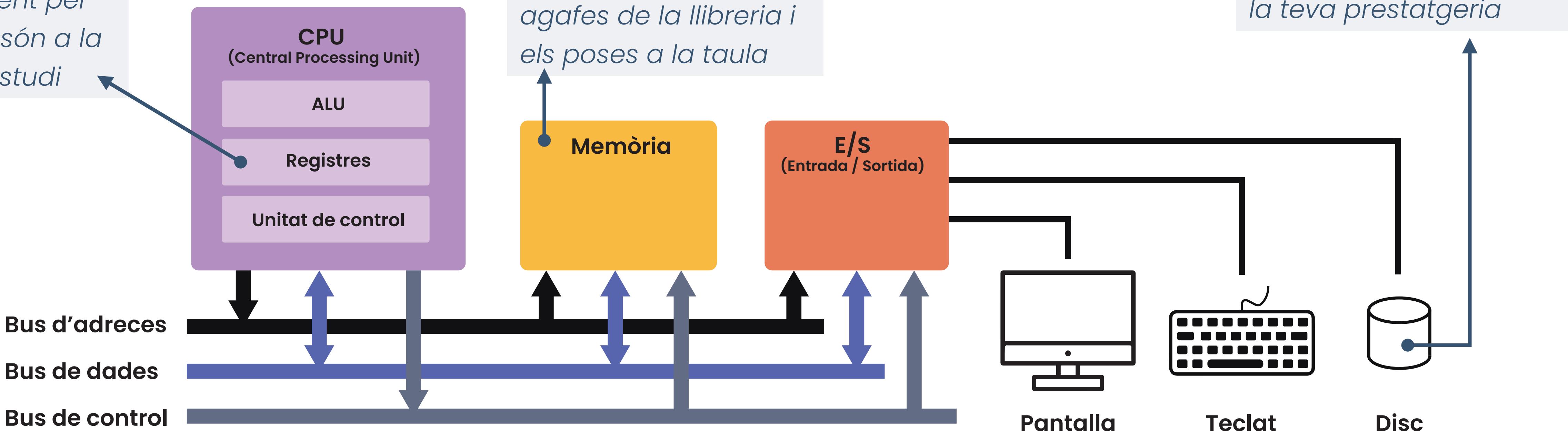


John Von Neumann

Per recordar-ho millor: Analogia dades-llibres

- Si considerem que les dades que necessita un programa són llibres...

Els llibres que estàs fent servir actualment per estudiar són a la taula d'estudi

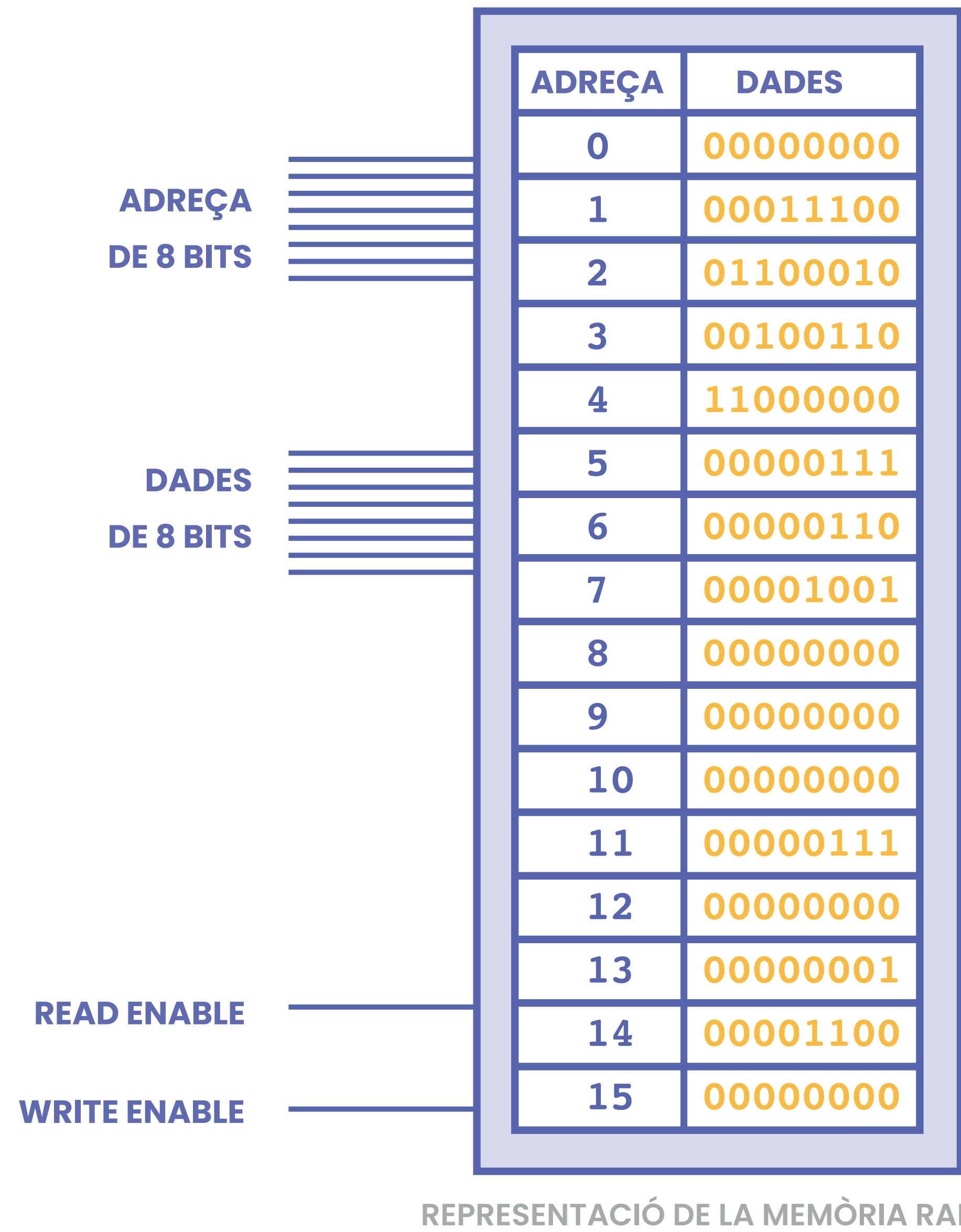


La memòria RAM

Estructura (conceptual)



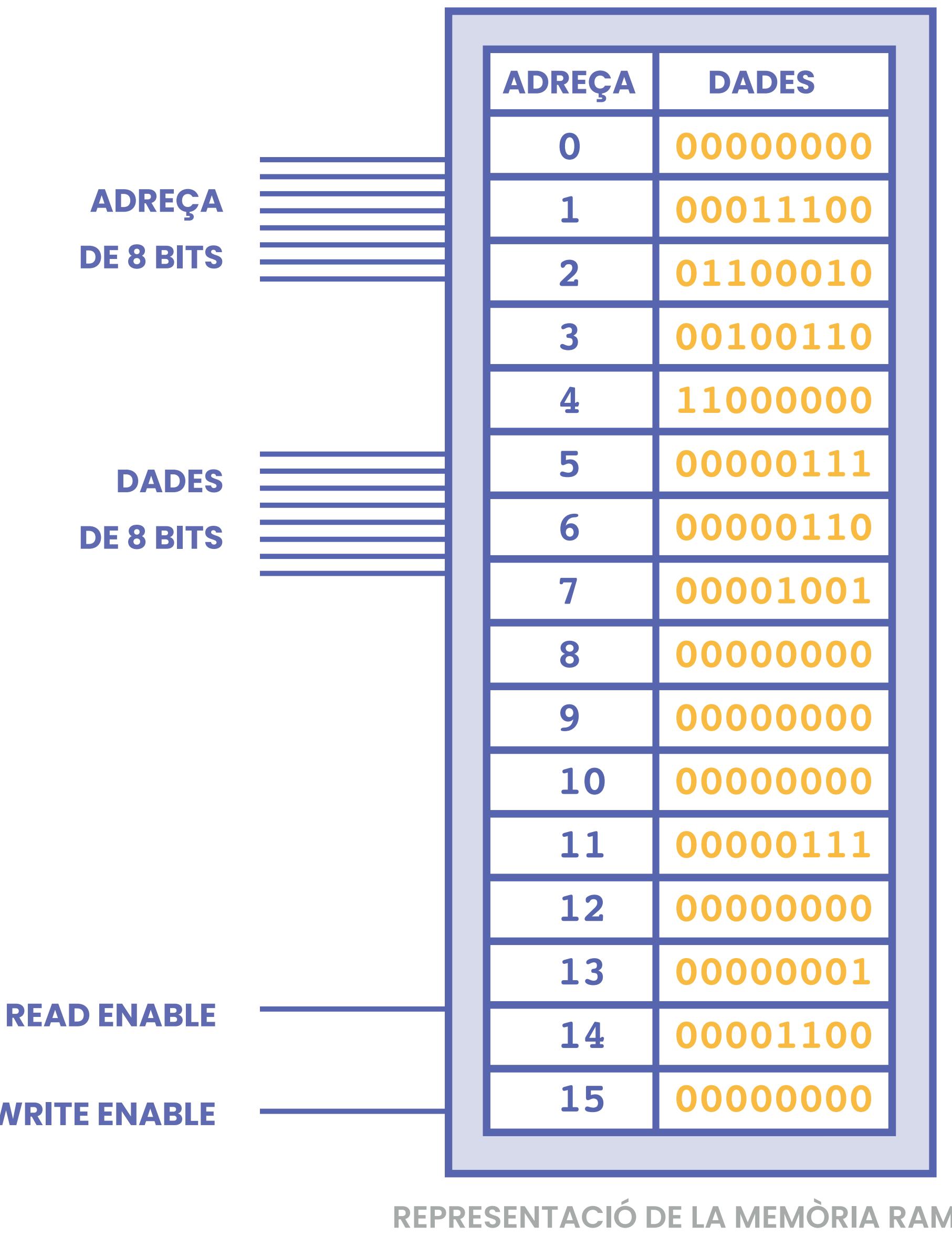
- Internament, la memòria és un conjunt de circuits capaç d'emmagatzemar dades
- Conceptualment, nosaltres ho representarem com una taula on, per cada adreça, hi podem guardar dades



0	0
1	28
2	98
3	38
4	192
5	7
6	6
7	9
8	0
9	0
10	0
11	7
12	0
13	1
14	12
15	0

REPRESENTACIÓ SIMPLIFICADA
DE LA MEMÒRIA RAM

La memòria RAM



0	0
1	28
2	98
3	38
4	192
5	7
6	6
7	9
8	0
9	0
10	0
11	7
12	0
13	1
14	12
15	0

REPRESENTACIÓ SIMPLIFICADA
DE LA MEMÒRIA RAM



ANALOGIA CONCEPTUAL

La memòria RAM

Per què "RAM"?

- Es diu RAM (Random Access Memory) perquè hi puc accedir de manera **arbitrària** en cost constant
 - És a dir, trigo el mateix en accedir a la posició 1 que a la posició 5000
 - En contrast amb les memòries d'accés seqüencial, on per accedir a la posició 5000, primer necessito passar per totes les prèvies

Exemples d'accés seqüencial (no random)



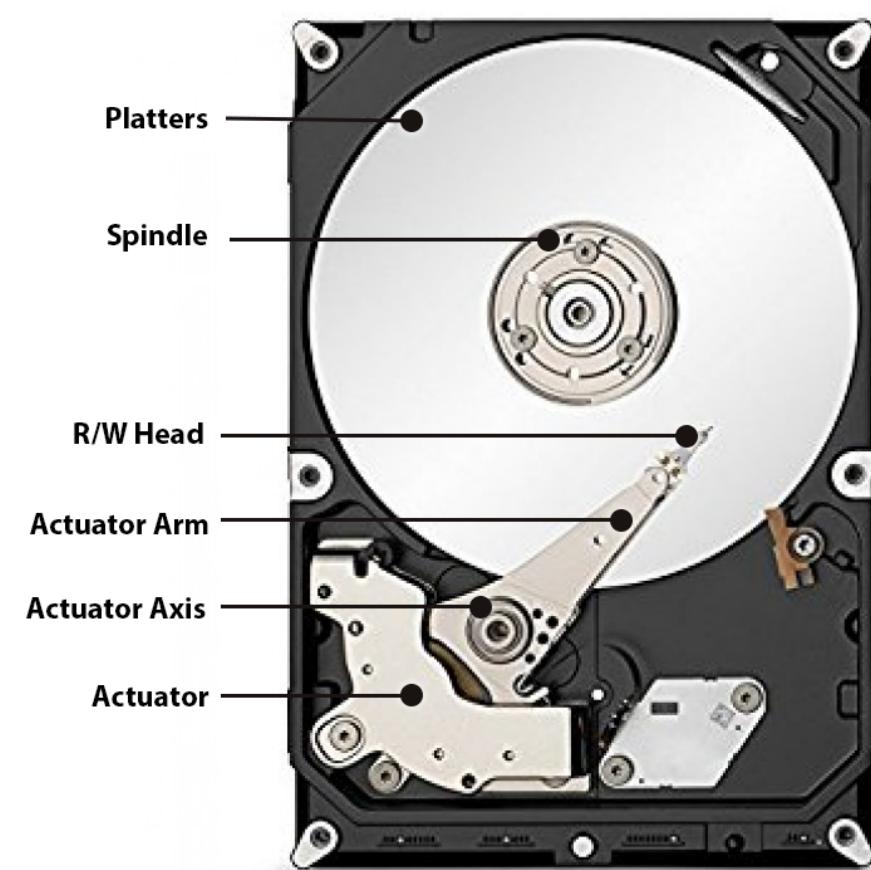
UN REPRODUCTOR DE LP



UNA CINTA VHS



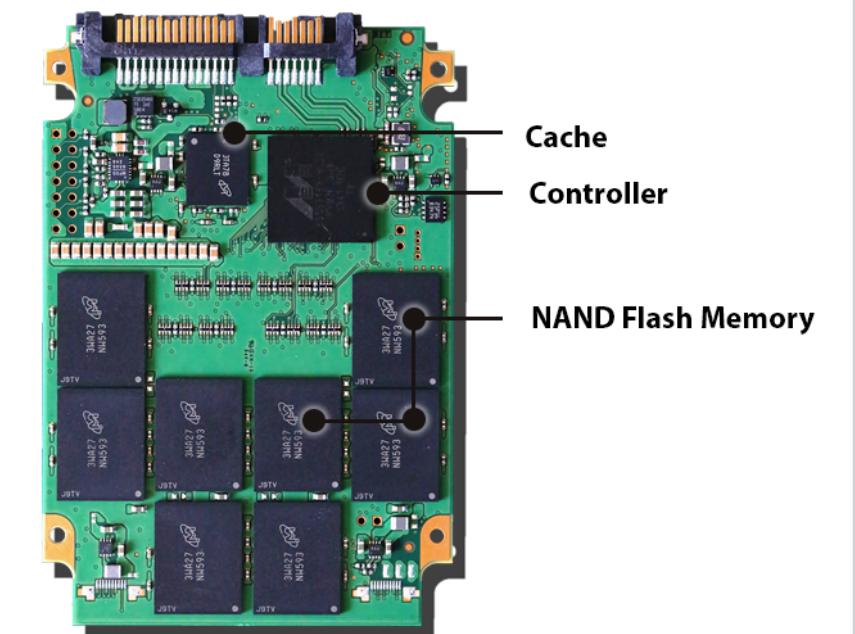
UNA CINTA DE CASSETTE



UN DISC DUR (HDD)
(SENSE LA CARCASSA)



OJO CUIDAO!



UN SOLID STATE DISK (SSD)
(SENSE LA CARCASSA)

Tot i que els actuals SSDs tenen accés "Random", no és correcte referir-s'hi com a memòria RAM

VARIABLES | CONSTANTS

Variables

Declaració de variables

- Una variable emmagatzema a la memòria un valor que es pot consultar i modificar dins un programa
- Com declarem una variable?
 - Assignar l'**identificador** que farem servir quan vulguem treballar amb la variable i indicar-ne el **tipus**
 - Com a resultat, es reserva un espai de memòria de la mida del tipus.

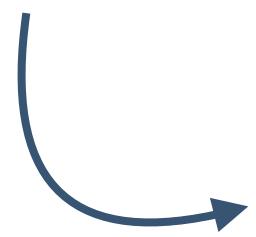


- **Què passa a memòria quan fem `int a;`?**
- **Quin valor tindrà la variable a?**
- **On es guarda la variable a?**

```
#include <stdio.h>

int main() {
    int a;
    return 0;
}
```

Espai de memòria
reservat per a la
nostra variable "a"



valor ???

Memòria de 32 bits

Variables

Declaració de variables

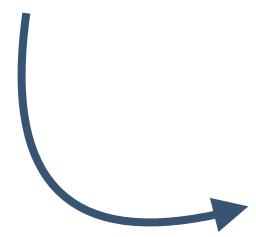
- Una variable emmagatzema a la memòria un valor que es pot consultar i modificar dins un programa
- Com declarem una variable?
 - Assignar l'**identificador** que farem servir quan vulguem treballar amb la variable i indicar-ne el **tipus**
 - Com a resultat, es reserva un espai de memòria de la mida del tipus.

Recordeu

- En pseudocodi, definim totes les variables dins una zona delimitada per les paraules reservades **var** i **fvar**

```
var  
    identificador: tipus;  
    identificador2: tipus;  
fvar
```

Espai de memòria
reservat per a la
nostra variable "a"



valor ???

Memòria de 32 bits

Variables

Inicialització de variables (vs. assignació)

- Quan **declarem** una variable simplement n'especifiquem el nom i el tipus i es reserva la memòria. Però el seu valor és desconegut
- Per donar-li valors tenim dues alternatives:

**Inicialitzar-la
(durant la declaració)**

```
● ● ●  
int main( ){  
    int a = 100;  
    return 0;  
}
```

**Assignar-li un valor
(després de la declaració)**

```
● ● ●  
int main( ){  
    int a;  
    a = 100;  
    return 0;  
}
```

No confongueu la terminologia entre **declarar**, **definir** (ho veurem més endavant), **inicialitzar**, i **assignar**.

Àmbit de les variables

- Això ja ho vau veure a *Disseny Descendent*. Anem a repassar-ho.

- **Variables Locals:**

- Són les variables que es defineixen dins d'un bloc o funció
- El seu àmbit es limita al bloc on estan definits, és a dir, fora d'aquest àmbit no es poden fer servir (*no es veuen*) (*veurem per què més endavant*).

- **Variables Globals:**

- Es defineixen fora de totes les funcions (inclòs el programa principal).
- Es poden consultar i modificar (*es veuen*) des de qualsevol punt del programa.

```
#include <stdio.h>

int num = 100; → Var. GLOBAL

int suma(int a, int b){
    int resultat;
    resultat = a + b;
    return (resultat); → Var. LOCAL a la funció suma
}

int main(){
    int a = 10; → Var. LOCAL a main
    return 0;
}
```



En general, no es recomana l'ús de variables globals, i en aquesta assignatura, està **prohibit** fer-les servir.

Constants

Constants de veritat o constants de mentida?

- Una constant, en principi, és una variable a la qual no podem canviar-li el valor un cop ja l'hem especificat.
- El valor que prenen s'anomena **literal**.

Constants "de veritat"

- Amb el modificador **const**, protegim els continguts de la variable perquè no es pugui modificar
- És necessari inicialitzar-la!



```
● ● ●  
int main(){  
    const int a = 100;  
    return 0;  
}
```



```
● ● ●  
int main(){  
    const int a;  
    return 0;  
}
```

- Si no la inicialitzem, valdrà 0 per defecte i no la podrem canviar (la constant més inútil de la història?).
Alguns compiladors no donen error.



```
● ● ●  
int main(){  
    const int a = 100;  
    a = a + 1;  
    return 0;  
}
```

- Si intentem modificar una constant, no ens ho permet.

error: assignment of read-only variable 'a'

Constants

Constants de veritat o constants de mentida?

- Una constant, en principi, és una variable a la qual no podem canviar-li el valor un cop ja l'hem especificat.
- El valor que prenen s'anomena **literal**.

Constants "de veritat"

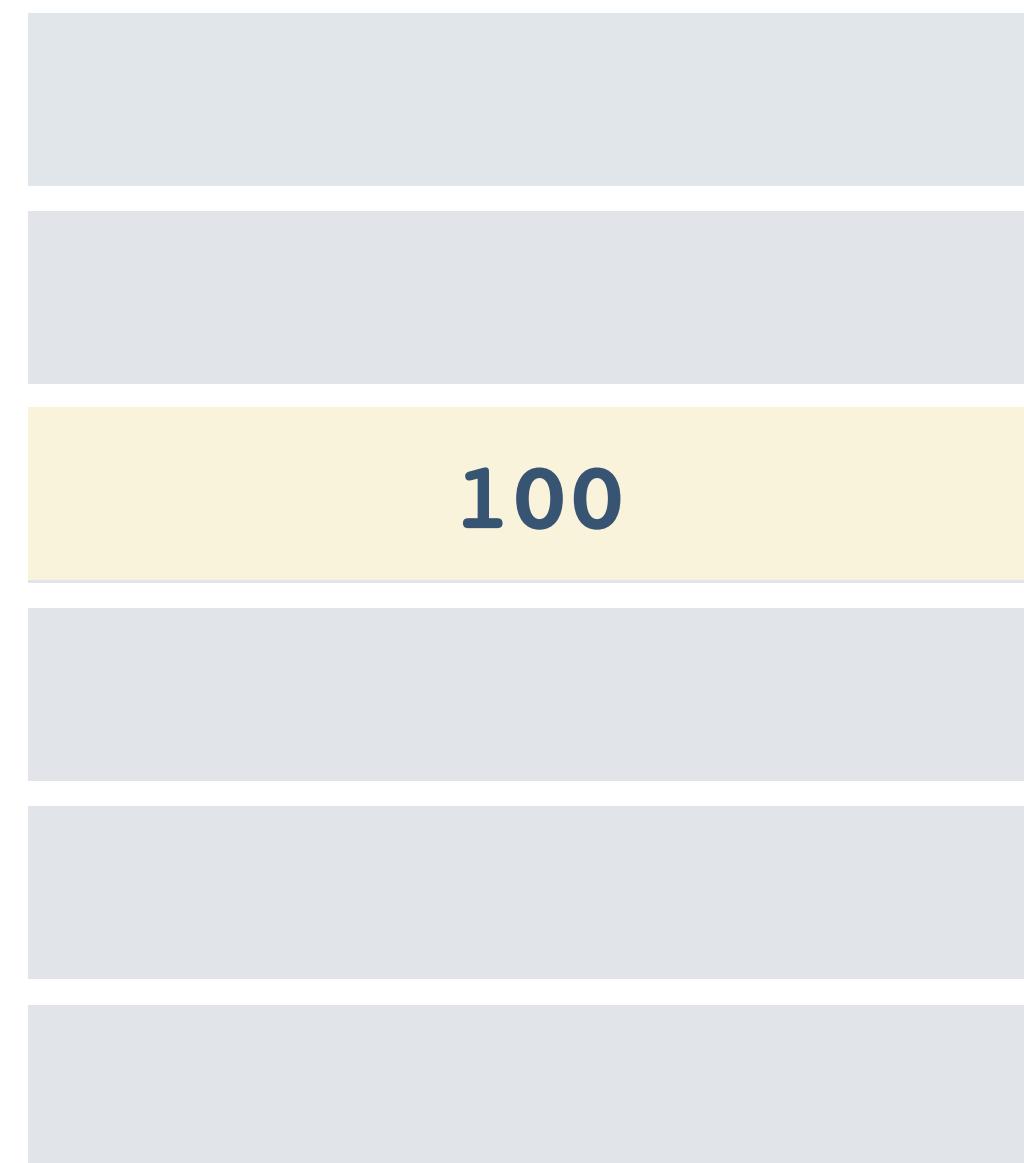
- Amb el modificador **const**, protegim els continguts de la variable perquè no es pugui modificar
- És necessari inicialitzar-la!



```
int main(){
    const int a = 100;
    return 0;
}
```

Per què són "de veritat"?

- Perquè són realment variables, i tenen un tipus.
- És a dir, ocupen un espai a memòria.
- Simplement són de només lectura.



Memòria de 32 bits

Constants

Constants de veritat o constants de mentida?

- Una constant, en principi, és una variable a la qual no podem canviar-li el valor un cop ja l'hem especificat.
- El valor que prenen s'anomena **literal**.

Constants "de mentida"

- Als laboratoris us haureu acostumat a definir constants amb la primitiva **#define**



```
● ● ●
#include <stdio.h>
#define A 100

int main(){
    printf("A = %d\n", A);
    return 0;
}
```

A dark-themed terminal window with three colored dots (red, yellow, green) in the top-left corner. It contains a C program with a '#define' directive. A large green circle with a white checkmark is overlaid on the right side of the window.

Per què són "de mentida"?

- No són variables, és a dir, no ocupen espai a memòria.
- No tenen tipus.
- El preprocessador fa un "search and replace" de totes les instàncies de "A" i ho substitueix per "100"

Constants

Constants de veritat o constants de mentida?

- Una constant, en principi, és una variable a la qual no podem canviar-li el valor un cop ja l'hem especificat.
- El valor que prenen s'anomena **literal**.

Recordeu

- En pseudocodi, definim totes les constants dins una zona delimitada per les paraules reservades **const** i **fconst**
- Els identificadors els escrivim sempre en majúscules, per distingir-ho de les variables. (No feu servir majúscules per les variables, ni minúscules per les constants!)

Números màgics hard-coded NO

```
const MAX := 100;  
fconst  
...  
a = b + 100;  
c = 100 + a;  
...  
X
```

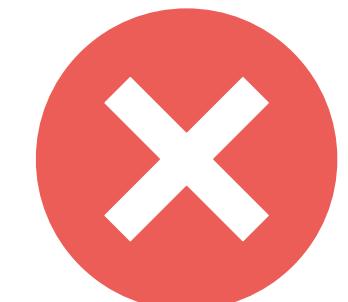
```
...  
a = b + MAX;  
c = MAX + a;  
...  
✓
```

```
const PI := 3.1416;  
MAX_CLIENTS := 100;  
fconst
```

Us ho poso més gran perquè no pogueu dir que no ho heu vist

Números màgics hard-coded NO

```
...
a = b + 100;
c = 100 + a;
...
```



```
const
    MAX := 100;
```

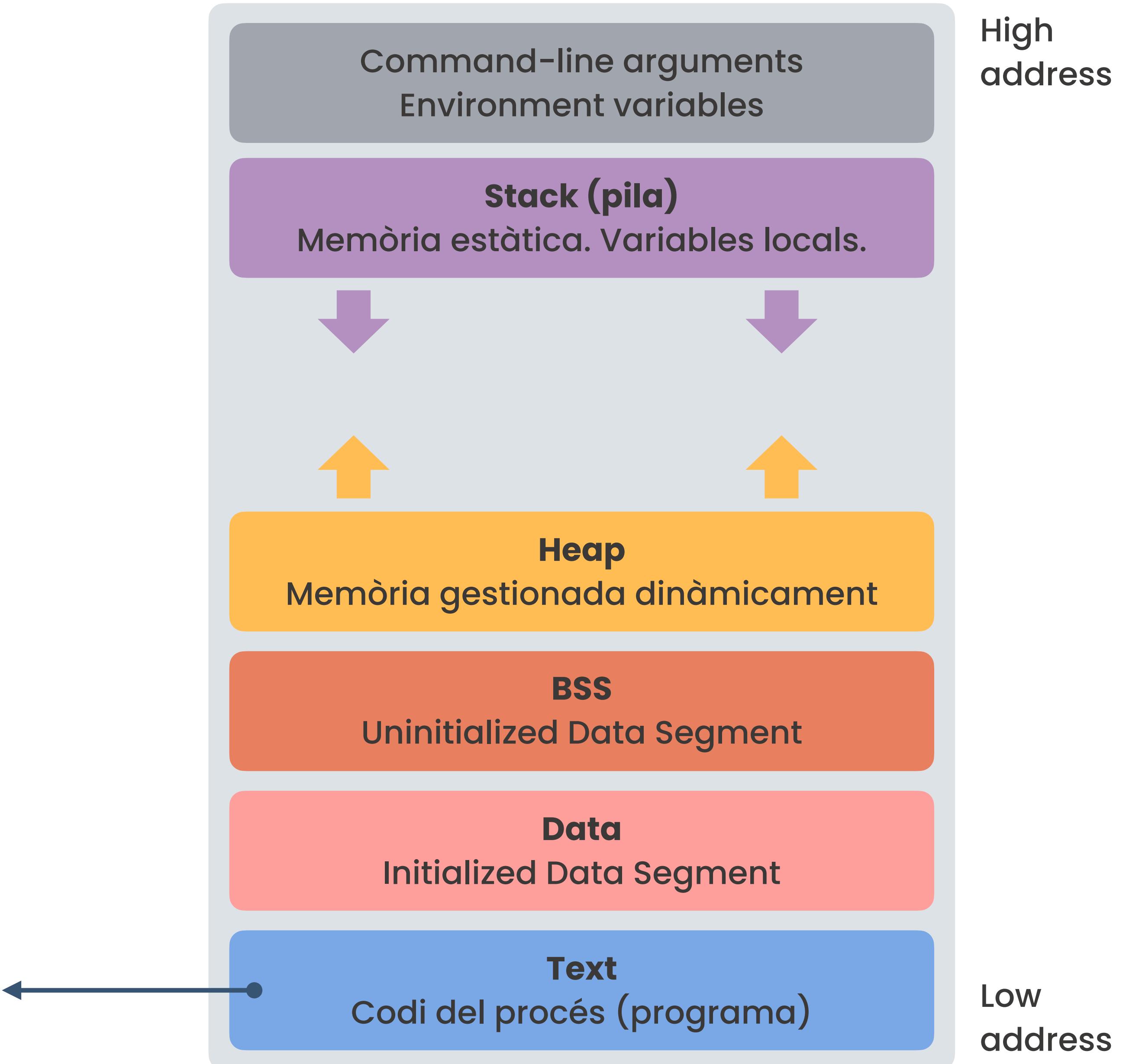
```
fconst
    ...
a = b + MAX;
c = MAX + a;
...
```



On es guarden, les variables?

Estructura de la memòria d'un programa en C

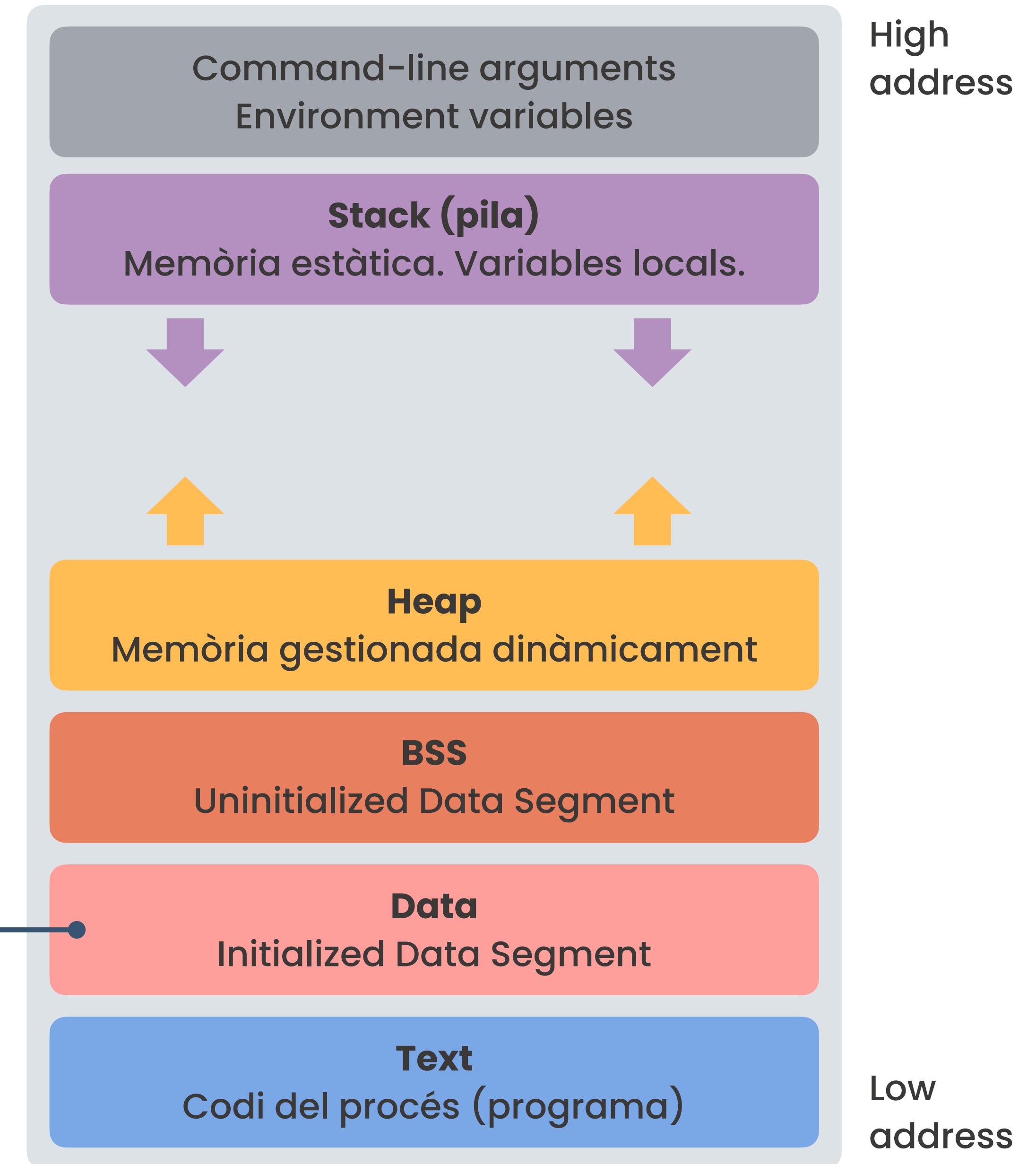
Codi màquina del programa que estem executant.
Acostuma a ser de només lectura.



On es guarden, les variables?

Estructura de la memòria d'un programa en C

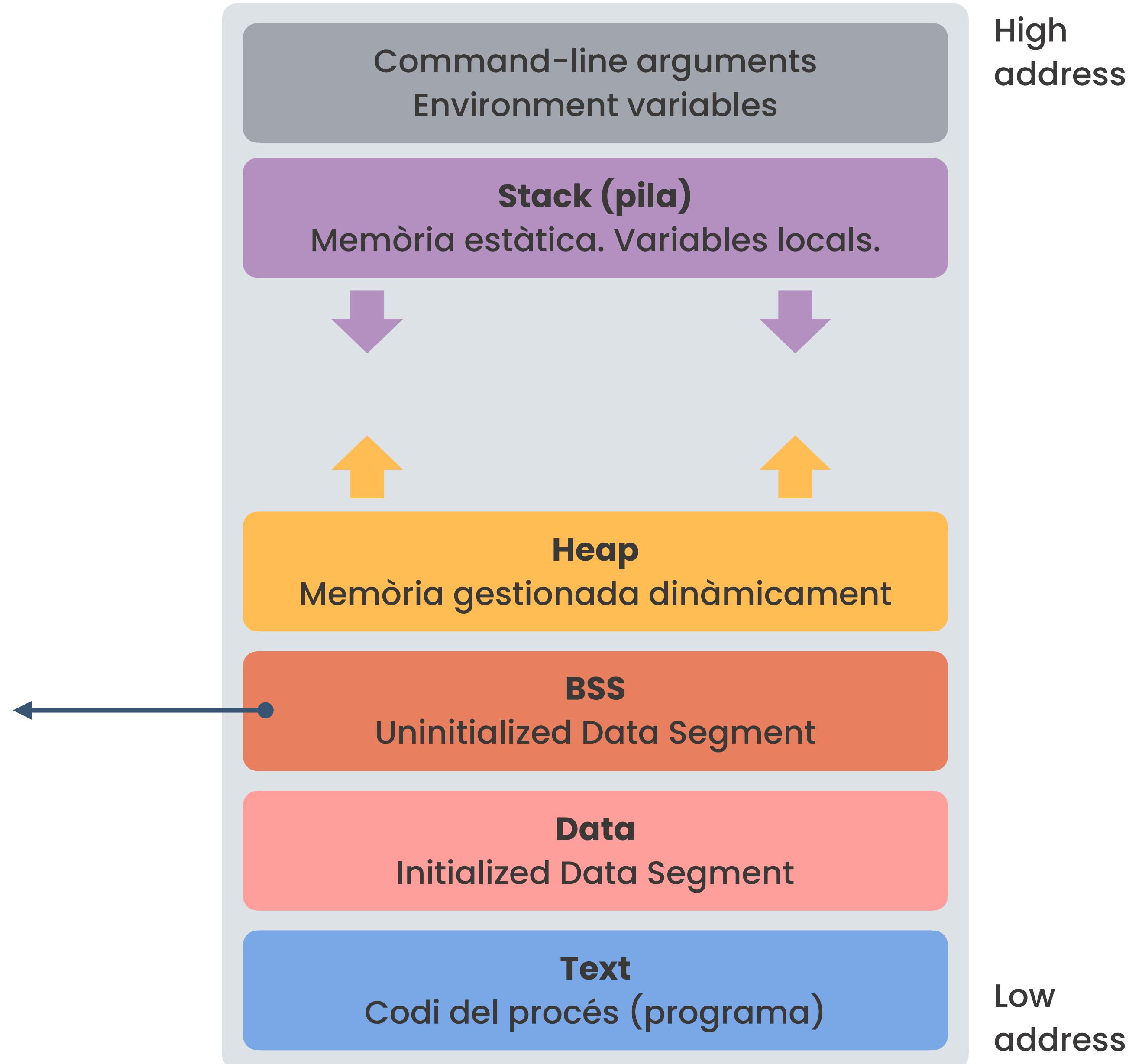
Conté les variables globals que hem inicialitzat explícitament al programa.
e.g. int maxcount = 99; fora de qualsevol funció



On es guarden, les variables?

Estructura de la memòria d'un programa en C

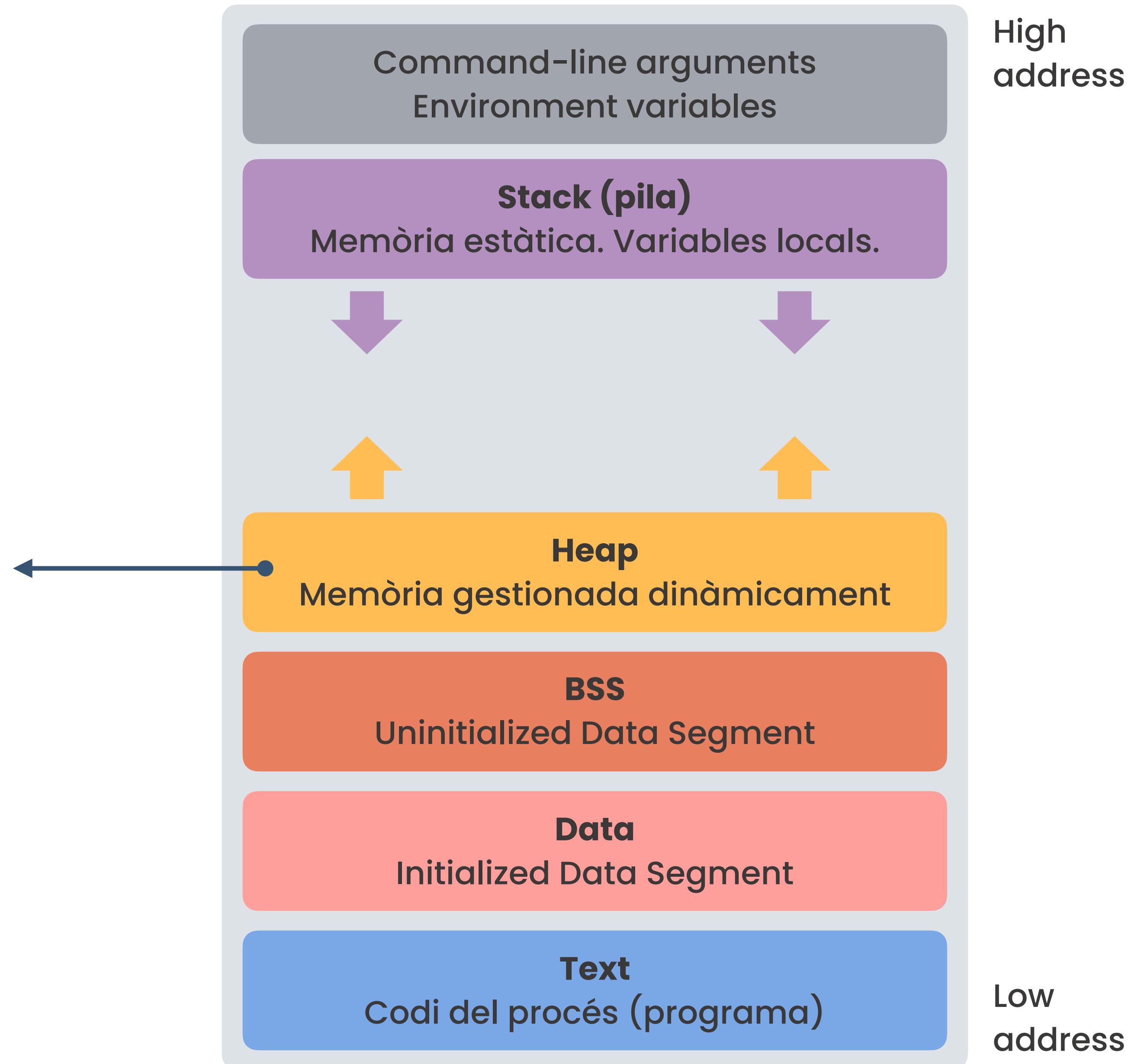
Conté les variables globals no inicialitzades.
e.g. `int maxcount;` fora de qualsevol funció
Les dades d'aquest segment s'inicialitzen a 0 (o a
NULL pointers) abans d'executar el programa.



On es guarden, les variables?

Estructura de la memòria d'un programa en C

Aquesta memòria l'hem de gestionar nosaltres.
Hem de demanar espai, comprovar si ens l'ha
pogut assignar, fer-lo servir, i després alliberar-lo.
Ho veurem més endavant.



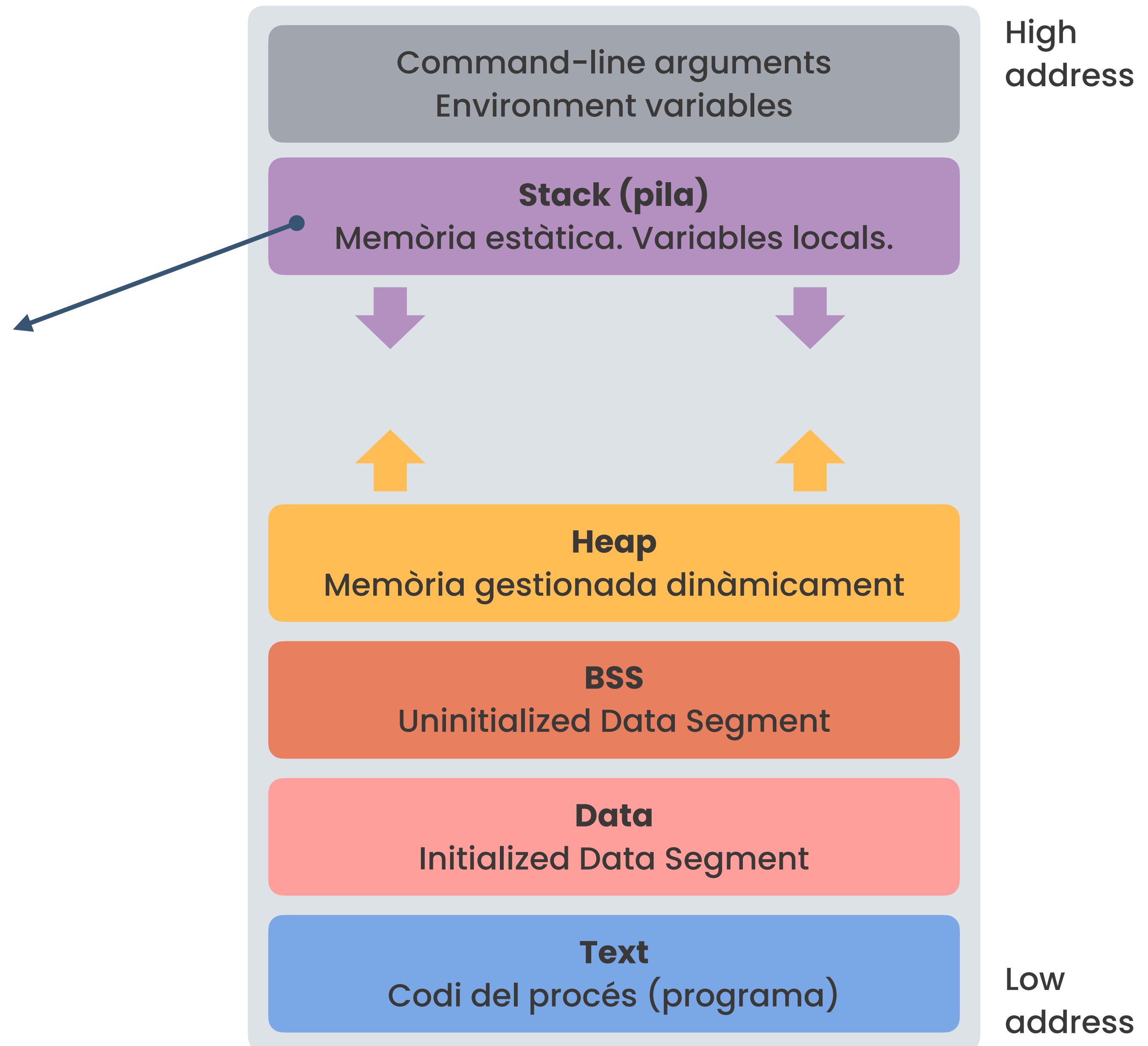
On es guarden, les variables?

Estructura de la memòria d'un programa en C

On es guarden les variables automàtiques (locals).
e.g. si dins del main declarem
`int taula[100];` aquesta taula es guarda a la pila

També s'hi guarda la informació de les crides de funcions (l'adreça de retorn, l'environment de la funció que crida...). [Ho veurem més endavant]

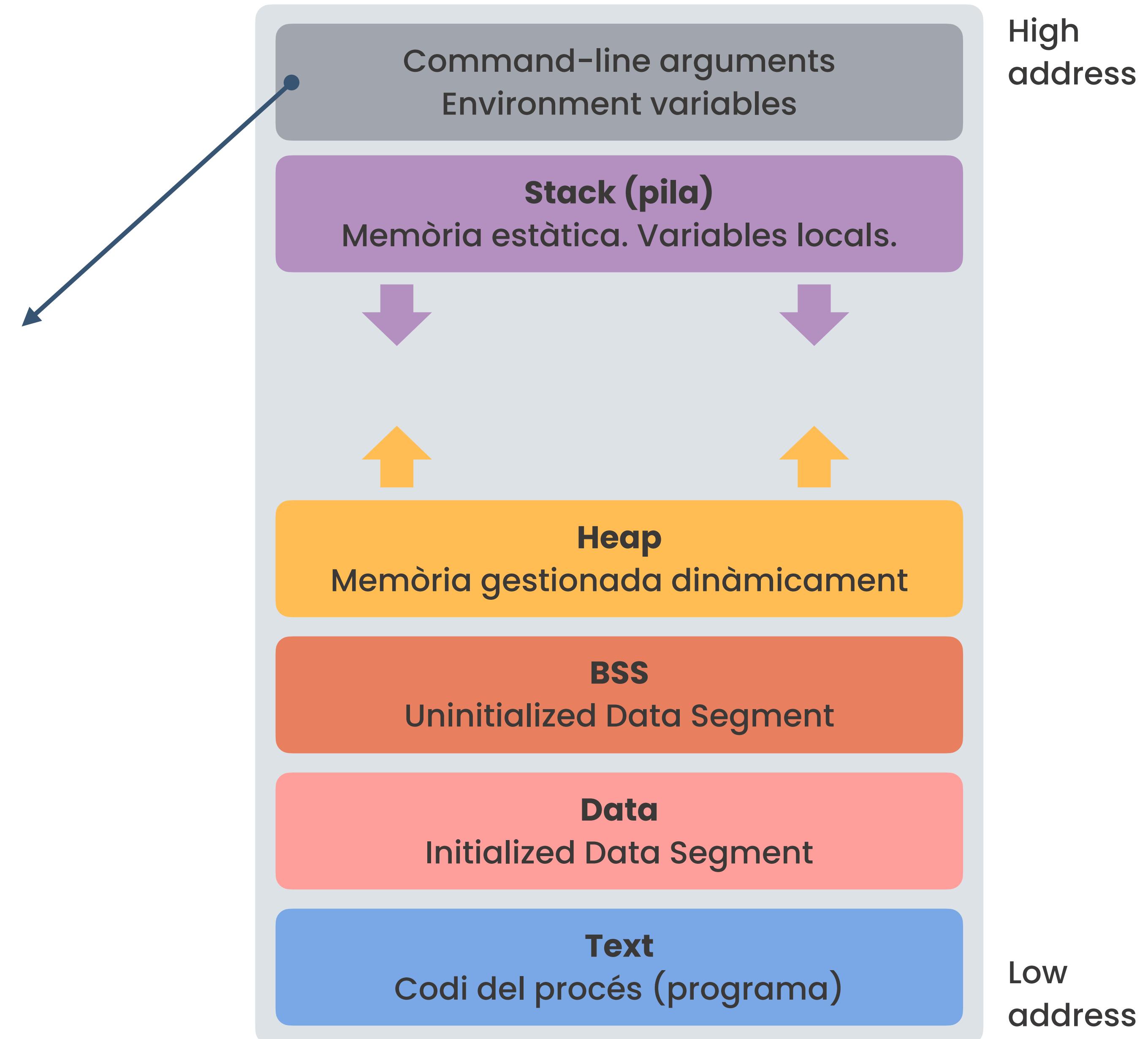
Es pot expandir però té un límit petit.



On es guarden, les variables?

Estructura de la memòria d'un programa en C

Quan executem un programa, des del sistema operatiu podem passar-li arguments. Aquests es guarden aquí.
Ho aprendrem a fer més endavant.



On es guarden, les variables?

Estructura de la memòria d'un programa en C

On es guarden a, b i c?

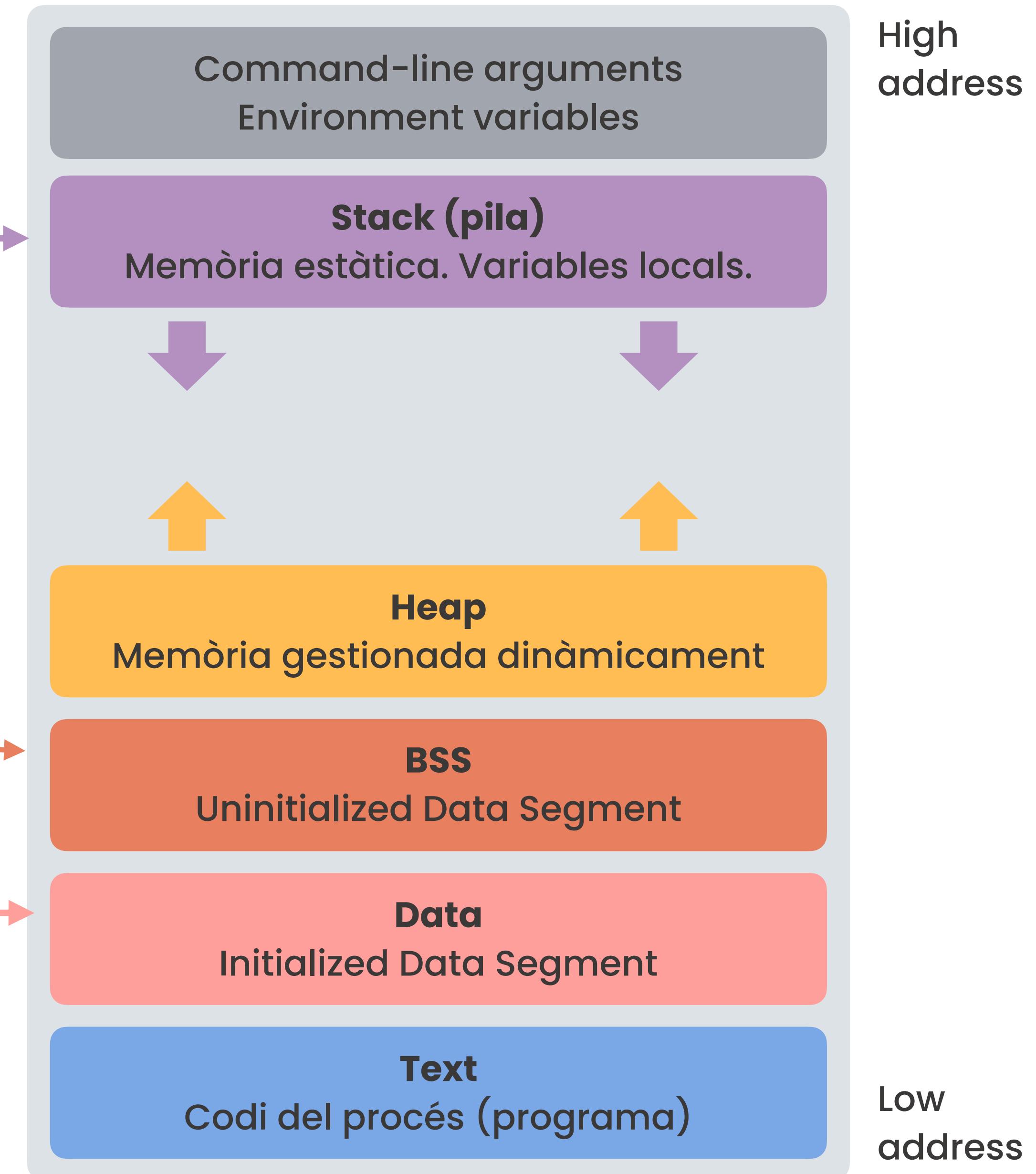
```
#include <stdio.h>

// Initialized global variable
int a = 5;

// Uninitialized global variable
int b;

int main() {
    // Local variables
    int c = 10;
    int taula[100];

    return 0;
}
```



Resum de pseudocodi: estructura, variables i constants

```
algorisme nom_programa és
var
    $ Aquí variables
fvar

const
    $ Aquí constants
fconst

inici
    $ Aquí comença l'execució
    ...
falgorisme
```

Comentaris

\$ Això és un comentari

Sentències

- Indentació dins de blocs
- Acabades en un punt-i-coma ;

Bloc de variables (si n'hi ha)

- Aquí no s'inicialitza el valor de la variable, només es declara

nom_de_variable : tipus;

- Tipus de variables simples:
 - enter (3), real (3.0) , booleà (cert/fals) , caràcter ('A')

Bloc de constants (si n'hi ha)

- Les constants van en majúscules
- No tenen tipus

NOM_CONSTANT := valor;

Instrucció d'assignació

:=

Resum de pseudocodi: estructura, variables i constants

```
algorisme nom_programa és
    var
        $ Aquí variables
    fvar

    const
        $ Aquí constants
    fconst

    inici
        $ Aquí comença l'execució
        ...
    falgorisme
```

Comentaris

\$ Això és un comentari

Sentències

- Indentació
- Acausalitat

WARNING

Bloc d'instruccions

A FP1 heu fet servir indistintament l'operador d'assignació esquerra `<-` i l'operador `:=`

nom_de_variable

- Tipus
- estàndards

En aquesta assignatura farem servir l'operador `:=`

Si voleu fer servir l'operador `<-`, podeu fer-ho, però heu de ser consistents a tot el codi.

Bloc d'instruccions

- Les d'FP1
- No tenen tipus

`NOM_CONSTANT := valor;`

Instrucció d'assignació

`:=`

TIPUS I OPERADORS

Tipus de variables

- Enters (3)
- Reals (3.0)
- Booleà (cert/fals)
- Caràcter ('A')

Variables simples:

```
var  
pes: enter;  
index_MC: real;  
c1, c2: caràcter;  
b: booleà;  
fvar
```

	En C			
	En pseudocodi	Tipus	Mida	Domini o Rang
Numèriques	enter	short ¹	(2 bytes) 16 bits	-32768 ... 32767
		int ¹	(4 bytes) 32 bits	-2147483648...2147483647
		long ¹	(8 bytes) 64 bits	-9223372036854775808... 9223372036854775807
Booleanes	real	float	(4 bytes) 32 bits	-3.4E+38 ... 3.4E+38
		double	(8 bytes) 64 bits	-1.7E-308 ... 1.7E+308
		long double	(16 bytes) 128 bits	-3.4E-4092 ... 3.4E+4092
Caràcter	booleà	bool	(1 byte) 8 bits	true ($\neq 0$), false (0)
	caràcter	char ¹	(1 byte) 8 bits	-127 ... 128

¹La paraula **unsigned** es pot afegir al tipus.

Compte! Les mides que ocupa cada tipus depenen de cada màquina!

Variables numèriques

Precaucions



Què passa si assigno un real a un enter? O un enter a un real? Puc convertir tipus? Quan ho he de fer explícitament?

Com puc saber quin rang de nombres puc representar amb un tipus determinat?

➔ Ho repassarem a la classe de laboratori

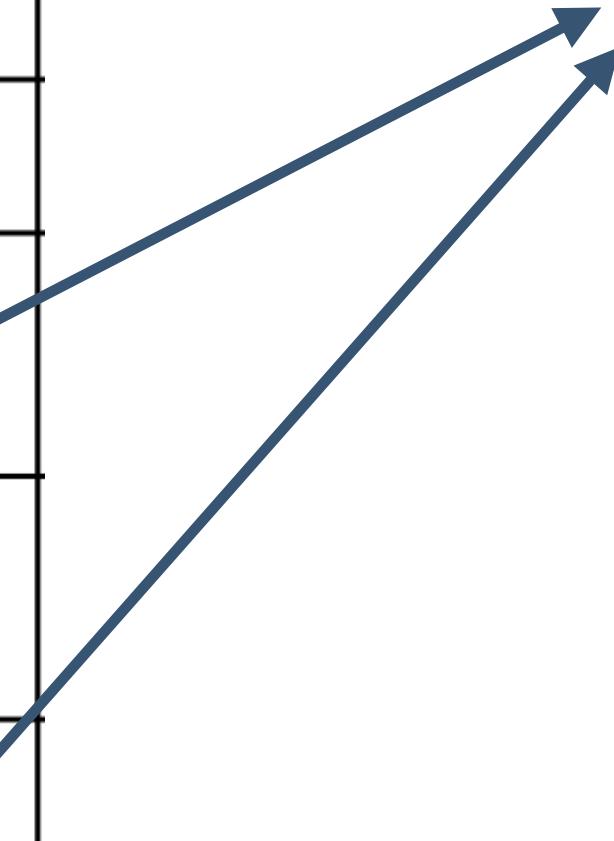
Variables numèriques

Operacions

- Quines operacions podem fer entre variables numèriques?

Operacions numèriques

Op.	Significat
+	La suma
-	Resta, o canvi de signe
*	Multiplicació
div	Divisió entera, de dos enters
mod	Residu, de dividir dos enters
/	Divisió amb decimals. Cal desar resultat en un real, sinó perdem els decimals.



Divisió entera o decimal?

Cal comprendre com funciona la divisió perquè, si no es fa bé, pot portar a què els programes no funcionin adequadament (i no ens n'adonem fins que sigui massa tard!)

El divisor no pot ser zero. És una situació que sovint caldrà comprovar quan fem una divisió

Variables numèriques

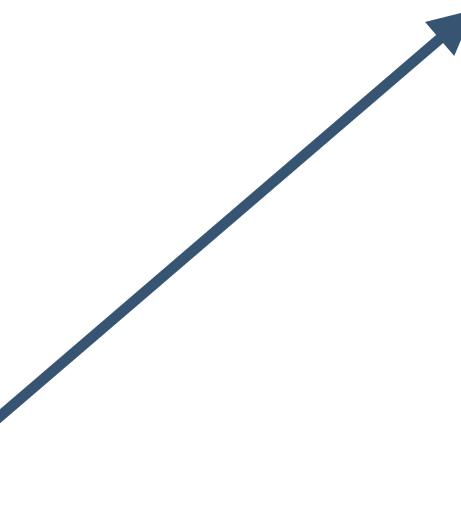
Operacions

- Quines operacions podem fer entre variables numèriques?

Operacions numèriques

Op.	Significat
+	La suma
-	Resta, o canvi de signe
*	Multiplicació
div	Divisió entera, de dos enters
mod	Residu, de dividir dos enters
/	Divisió amb decimals. Cal desar resultat en un real, sinó perdem els decimals.

El **mòdul** només es pot fer entre nombres **enters**



Variables numèriques

Operacions

- Quines operacions podem fer entre variables numèriques?

Operacions numèriques

Op.	Significat
+	La suma
-	Resta, o canvi de signe
*	Multiplicació
div	Divisió entera, de dos enters
mod	Residu, de dividir dos enters
/	Divisió amb decimals. Cal desar resultat en un real, sinó perdem els decimals.

Operacions relacionals

- Els operadors relacionals permeten realitzar comparacions:
 $=, \neq, <, >, \leq, \geq$
- El resultat d'una operació serà cert o fals en funció de si el resultat de la comparació es compleix o no.

Variables booleanes

El tipus booleà permet definir variables que desaran valors **cert** o **fals**.

Operacions lògiques

- Operador lòtic **i (and)**: És cert quan **tots** els membres de l'expressió són certs.
- Operador lòtic **o (or)**: És cert quan **algun** dels membres de l'expressió és cert.
- Operador lòtic **no (not)**: Operador de **negació**. És cert quan l'expressió que modifica és fals, i fals quan l'expressió és certa.

Recordeu les taules de la veritat

NOT



INPUT

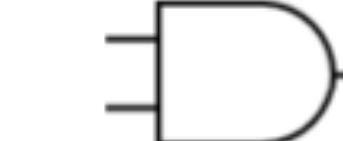
A

OUTPUT

0

1

AND



INPUT

A

OUTPUT

0

1

0

0

0

1

OR



INPUT

A

OUTPUT

0

1

0

1

0

1

1

1

Variables caràcter

Permet definir variables per emmagatzemar símbols que utilitzarem per representar informació textual

- Aquests símbols formen part d'un codi de caràcters (ASCII)
- Cada símbol té un número d'ordre dins el codi de caràcters
- Per definir un caràcter literal, el posem entre cometes simples: '
• E.g. 'a', 'A', '1', '?' ...

Regular ASCII Chart (character codes 0 - 127)									
000	(nul)	016 ► (dle)	032 sp	048 0	064 0	080 P	096 `	112 p	
001 ☺ (soh)	017 ◀ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q		
002 ☻ (stx)	018 ↵ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r		
003 ♥ (etx)	019 !! (dc3)	035 #	051 3	067 C	083 S	099 c	115 s		
004 ♦ (eot)	020 ¶ (dc4)	036 \$	052 4	068 D	084 T	100 d	116 t		
005 ☢ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u		
006 ☣ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v		
007 ☤ (bel)	023 ↴ (etb)	039 '	055 7	071 G	087 W	103 g	119 w		
008 ☨ (bs)	024 ↵ (can)	040 (056 8	072 H	088 X	104 h	120 x		
009 (tab)	025 ↶ (em)	041)	057 9	073 I	089 Y	105 i	121 y		
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z		
011 ☠ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [107 k	123 {		
012 ☧ (np)	028 L (fs)	044 ,	060 <	076 L	092 \	108 l	124		
013 (cr)	029 ↲ (gs)	045 –	061 =	077 M	093]	109 m	125 }		
014 ☢ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~		
015 ☣ (si)	031 ↳ (us)	047 /	063 ?	079 O	095 _	111 o	127 ¸		

Recordeu

La doble personalitat dels caràcters

En C, un caràcter és realment un enter, i per tant, el podem tractar com a enter.

Quin resultat genera això?

```
char car;  
car = 'A';  
car++;  
printf("car = %c", car);
```

Sortida:

```
car = B
```

Variables caràcter

Permet definir variables per emmagatzemar símbols que utilitzarem per representar informació textual

- Aquests símbols formen part d'un codi de caràcters (ASCII)
- Cada símbol té un número d'ordre dins el codi de caràcters
- Per definir un caràcter literal, el posem entre cometes simples: '
• E.g. 'a', 'A', '1', '?' ...

Regular ASCII Chart (character codes 0 - 127)									
000	(nul)	016 ► (dle)	032 sp	048 0	064 0	080 P	096 `	112 p	
001 ☺ (soh)	017 ◀ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q		
002 ☻ (stx)	018 ↵ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r		
003 ♥ (etx)	019 !! (dc3)	035 #	051 3	067 C	083 S	099 c	115 s		
004 ♦ (eot)	020 ¶ (dc4)	036 \$	052 4	068 D	084 T	100 d	116 t		
005 ☢ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u		
006 ☣ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v		
007 ☤ (bel)	023 ↴ (etb)	039 '	055 7	071 G	087 W	103 g	119 w		
008 ☥ (bs)	024 ↵ (can)	040 (056 8	072 H	088 X	104 h	120 x		
009 (tab)	025 ↶ (em)	041)	057 9	073 I	089 Y	105 i	121 y		
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z		
011 ☧ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [107 k	123 {		
012 ☨ (np)	028 L (fs)	044 ,	060 <	076 L	092 \	108 l	124		
013 (cr)	029 ↳ (gs)	045 –	061 =	077 M	093]	109 m	125 }		
014 ☧ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~		
015 ☧ (si)	031 ↷ (us)	047 /	063 ?	079 O	095 _	111 o	127 ¸		

Recordeu

La doble personalitat dels caràcters

En C, un caràcter és realment un enter, i per tant, el podem tractar com a enter.

Quin resultat genera això?

```
char car;  
  
car = 'G';  
  
car = car + 'a' - 'A';  
  
printf("car = %c", car);
```

Sortida:

```
car = g
```

Variables caràcter

Permet definir variables per emmagatzemar símbols que utilitzarem per representar informació textual

- Aquests símbols formen part d'un codi de caràcters (ASCII)
- Cada símbol té un número d'ordre dins el codi de caràcters
- Per definir un caràcter literal, el posem entre cometes simples: '
• E.g. 'a', 'A', '1', '?' ...

Regular ASCII Chart (character codes 0 - 127)									
000	(nul)	016 ► (dle)	032 sp	048 0	064 0	080 P	096 `	112 p	
001 ☺ (soh)	017 ◀ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q		
002 ☻ (stx)	018 ↵ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r		
003 ♥ (etx)	019 !! (dc3)	035 #	051 3	067 C	083 S	099 c	115 s		
004 ♦ (eot)	020 ¶ (dc4)	036 \$	052 4	068 D	084 T	100 d	116 t		
005 ☢ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u		
006 ☣ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v		
007 ☤ (bel)	023 ↴ (etb)	039 '	055 7	071 G	087 W	103 g	119 w		
008 ☥ (bs)	024 ↵ (can)	040 (056 8	072 H	088 X	104 h	120 x		
009 (tab)	025 ↶ (em)	041)	057 9	073 I	089 Y	105 i	121 y		
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z		
011 ☧ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [107 k	123 {		
012 ☨ (np)	028 L (fs)	044 ,	060 <	076 L	092 \	108 l	124		
013 (cr)	029 ↲ (gs)	045 –	061 =	077 M	093]	109 m	125 }		
014 ☧ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~		
015 ☧ (si)	031 ↳ (us)	047 /	063 ?	079 O	095 _	111 o	127 ¸		

Recordeu

La doble personalitat dels caràcters

En C, un caràcter és realment un enter, i per tant, el podem tractar com a enter.

Quin resultat genera això?

```
char car;  
car = '2';  
car = car - '0';  
printf("car = %d", car);
```

Sortida:

```
car = 2
```

(El número 2,
no el caràcter '2'!)

Resum de pseudocodi: operacions

Suma

```
a := a + 2;
```

Resta

```
a := a - 2;
```

Multiplicació

```
a := a * 2;
```

Divisió entera

```
a := a div 2;
```

Divisió amb decimals

```
a := a / 2;
```

Mòdul

```
a := a mod 2;
```

Operacions de comparació

- Igualtat
- Desigualtat
- Menor que
- Major que
- Menor o igual que
- Major o igual que

=
≠
<
>
≤
≥

Operacions lògiques

- And
- Or
- Not (negació)

i
o
no

```
(x < 2) i (no(trobat))
```

INTERACCIÓ AMB L'USUARI

(Llegir de teclat, escriure per pantalla)

Resum de pseudocodi: Interacció amb l'usuari

Escriure per la sortida estàndard (per pantalla)

- Mostra un text per pantalla, permetent concatenar diferents textos i variables.
- Quan acaba d'escriure fa un salt de línia

```
escriure("text", variable, "text", variable, ...);
```

Llegir per la entrada estàndard (de teclat)

- Llegeix un enter, real, o caràcter per teclat. Aquest valor s'emmagatzema a la variable que s'indica per paràmetre.
- Quan s'acaba d'entrar la dada, cal que l'usuari premi <Enter>

```
llegir(nom_variable);
```

WARNING

AVÍS: La funció de pseudocodi que fem servir per llegir de teclat patirà una lleugera modificació durant el curs.

ESTRUCTURES DE CONTROL

Estructures de control

- Les instruccions dels algorismes s'executen de forma **seqüencial** (i.e., una després de l'altra)
- La programació estructurada disposa de blocs que permeten **controlar el flux d'execució**. Són les estructures de control.

Tipus d'estructures de control

Estructures de selecció (Condicionals)

- Les instruccions de selecció trien quin bloc de codi executar en funció d'una condició donada.
- E.g.: `if` / `if-else` / `switch`

Estructures d'iteració (Bucle)

- Les instruccions iteratives executen repetidament un bloc de codi mentre una condició especificada sigui certa.
- E.g.: `while` / `do while` / `for`

Estructures de salt

- Les instruccions de salt transfereixen el control a una altra part del programa de manera incondicional.
- E.g.: `break` / `continue` / `goto` / `return`



BIBLIOTECA DE HOGWARTS

Estructures de selecció (Condicionals)

- Les instruccions de selecció trien quin bloc de codi executar en funció d'una condició donada.
- E.g.: `if` / `if-else` / `switch`

Estructures d'iteració (Bucles)

- Les instruccions iteratives executen repetidament un bloc de codi mentre una condició especificada sigui certa.
- E.g.: `while` / `do while` / `for`



SECCIÓ RESTRINGIDA DE LA BIBLIOTECA DE HOGWARTS

Estructures de salt

- Les instruccions de salt transfereixen el control a una altra part del programa de manera incondicional.
- E.g.: `break` / `continue` / `goto` / `return`

Estructures de selecció (condicionals)

- Condicional **simple**: **si**

```
si (condició) llavors
    instruccions;
fsi
```

- Condicional **doble**: **si-sinó**

```
si (condició) llavors
    instruccions1;
sino
    instruccions2;
fsi
```

- Condicional **múltiple**: **si-sinó si-sinó**

```
si (condició) llavors
    instruccions1;
sino si (condició2) llavors
    instruccions2;
sino
    instruccions_n;
fsi
```

- Condicional **múltiple** especial: **switch**

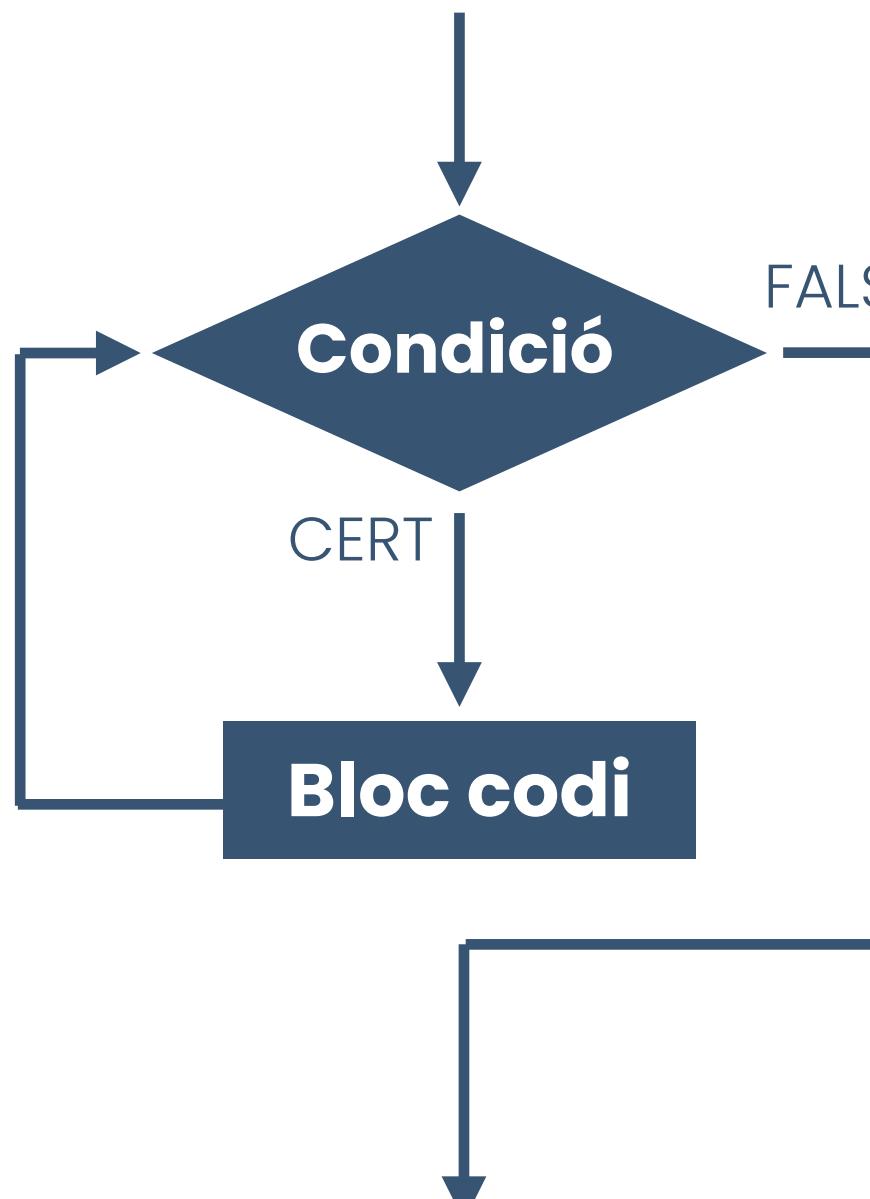
```
opcio (expressió)
    valor1: instruccions1; break;
    valor2: instruccions2; break;
    ...
    valorN: instruccionsn; break;
en altre cas:
    instruccions_per_defecte;
fopcio
```

Estructures iteratives (bucles)

Com sabem quan fer servir
mentre i quan **fer-mentre**?

Estructura **mentre**

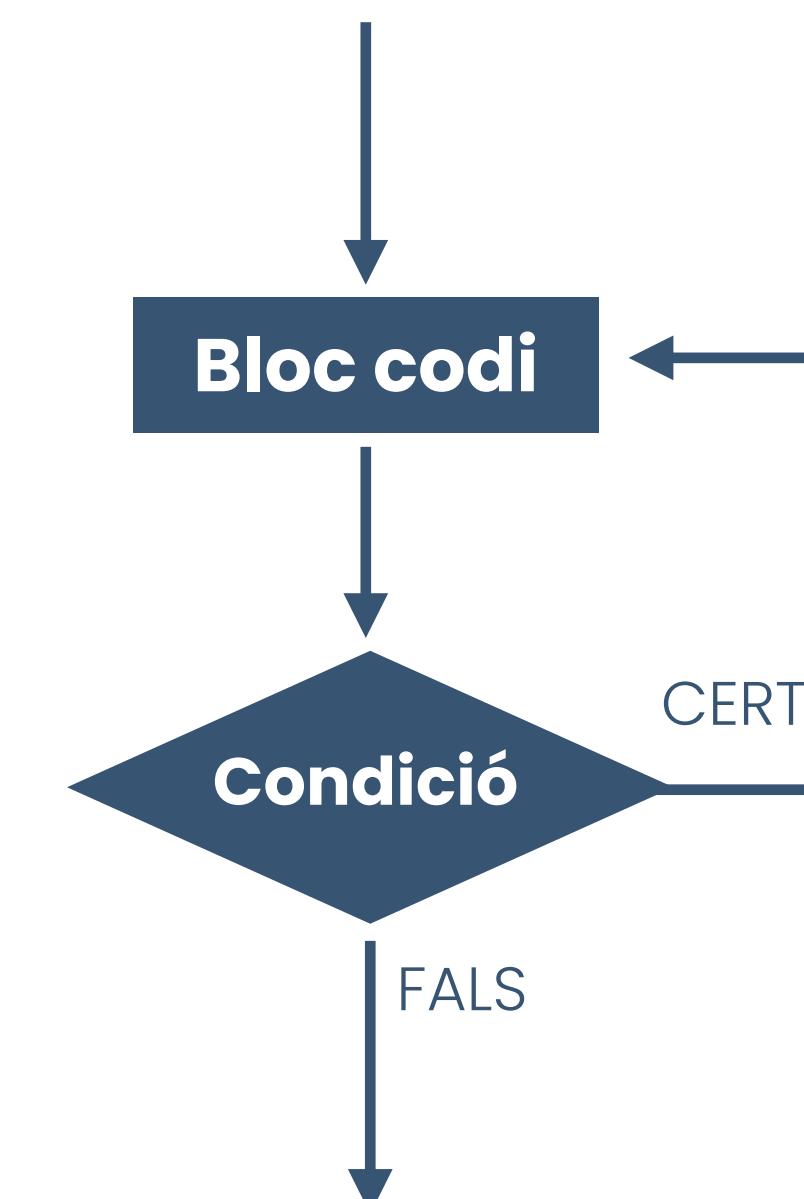
```
$ Inicialitzar variable control  
mentre (condició_NO_fi) fer  
    instruccions;  
    $ Actualitzar variable control  
fmentre
```



"mentre"

Estructura **fer-mentre**

```
$ Inicialitzar variable control  
fer  
    instruccions;  
    $ Actualitzar variable control  
mentre (condició_NO_fi);
```



"fer mentre"



L'estruutura **mentre** és un bucle on avaluem la condició **abans** d'executar el cos. En l'estruutura **fer-mentre** avaluem la condició **després** d'executar el cos.

Com sabem quan fer servir
mentre i quan **fer-mentre**?

Quan volem fer una acció repetidament, no sabem quantes vegades, però sabem que com a mínim volem fer-la una vegada, farem servir fer-mentre.

- Per **exemple**, problemes de tipus validació d'entrada d'usuari: demanem que l'usuari introduceixi alguna cosa i li seguim demanant mentre no sigui correcte.
 - Un exemple seria demanar a l'usuari que introduceixi un nombre positiu.

Opció 1 (mentre)

```
algorisme validacio_usuari és
var
    nombre: enter;
fvar
inici
    escriure("Introdueix un nombre positiu.");
    llegir(nombre);
    mentre (nombre <= 0) fer
        escriure("Introdueix un nombre positiu.");
        llegir(nombre);
fmentre
    escriure("El nombre és ", nombre);
falgorisme
```



Repetició de codi

Com sabem quan fer servir
mentre i quan **fer-mentre**?

Quan volem fer una acció repetidament, no sabem quantes vegades, però sabem que com a mínim volem fer-la una vegada, farem servir fer-mentre.

- Per **exemple**, problemes de tipus validació d'entrada d'usuari: demanem que l'usuari introduceixi alguna cosa i li seguim demanant mentre no sigui correcte.
 - Un exemple seria demanar a l'usuari que introduceixi un nombre positiu.

Opció 2 (mentre)

```
algorisme validacio_usuari és
var
    nombre: enter;
fvar
inici
    $ Inicialitzo nombre a un valor que faci
        entrar al bucle
    nombre := -1;
mentre (nombre <= 0) fer
    escriure("Introduceix un nombre positiu.");
    llegir(nombre);
fmentre
    escriure("El nombre és ", nombre);
falgorisme
```



Solució "ortopèdica"

Com sabem quan fer servir
mentre i quan **fer-mentre**?

Quan volem fer una acció repetidament, no sabem quantes vegades, però sabem que com a mínim volem fer-la una vegada, farem servir fer-mentre.

- Per **exemple**, problemes de tipus validació d'entrada d'usuari: demanem que l'usuari introduceixi alguna cosa i li seguim demanant mentre no sigui correcte.
 - Un exemple seria demanar a l'usuari que introduceixi un nombre positiu.
- Aquest problema, es resol de manera més natural amb una estructura **fer-mentre**.

Opció 3 (fer-mentre)

```
algorisme validacio_usuari és
var
    nombre: enter;
fvar
inici
    fer
        escriure("Introdueix un nombre positiu.");
        llegir(nombre);
        mentre (nombre <= 0);
            escriure("El nombre és ", nombre);
    falgorisme
```



Estructures iteratives (bucles)

Estructura per

```
per (var:=valor_inicial; condició_no_fi; increment_var)
    instruccions;
fper
```

- Recordeu l'ordre d'execució d'un bucle "per":

1. S'assigna el valor inicial a la variable de control
2. Es comprova la condició. Si és certa, s'entra al loop.
3. S'executa el cos del "per"
4. S'executa l'increment de la variable de control
5. Tornem al pas (2)

Quin resultat genera això?



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

```
int main() {
    int i;
    for(i = 0; i < 5; i++) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Estructures iteratives (bucles)

Estructura per

```
per (var:=valor_inicial; condició_no_fi; increment_var)
    instruccions;
fper
```

- Recordeu l'ordre d'execució d'un bucle "per":

1. S'assigna el valor inicial a la variable de control
2. Es comprova la condició. Si és certa, s'entra al loop.
3. S'executa el cos del "per"
4. S'executa l'increment de la variable de control
5. Tornem al pas (2)



En pseudocodi no es pot:

- Declarar la variable a la capçalera del "per"
- Incrementar la variable amb operador "++" (no existeix en pseudocodi)

Estructures de selecció vs. iteració

Condicional vs. bucle

si (no mal_de_panxa) llavors



fsi

mentre (no mal_de_panxa) fer



fmentre

Quantes bosses de patates m'he menjat en cada cas?

Tindré mal de panxa al final?

Estructures de selecció vs. iteració

Mentre vs. fer mentre

mentre (no mal_de_panxa) fer



fmentre

fer



mentre (no mal_de_panxa);

I ara? Pot ser que m'hagi menjat alguna bossa de
Jumpers tot i tenir mal de panxa?

Secció restringida: estructures de salt

Instrucció break

- Serveix per a sortir d'un **for**, un **switch**, un **do-while** o un **while**.
- Després d'executar **break**, la propera instrucció a executar serà la següent després del bucle (no es faran més iteracions del bucle).
- Està permès (és necessari, de fet!) el seu ús dins l'estructura **switch**.



En aquest curs, està **prohibit** fer servir breaks per sortir de bucles

Quin resultat genera això?



```
#include <stdio.h>
#define N 5
int main() {
    int arr[N] = {1, 2, 3, 4, 5};
    int i;

    for(i = 0; i < N; i++) {
        if (arr[i] == 3) {
            break;
        }
        printf("%d ", arr[i]);
    }
    /* El break va a parar aquí */
    return 0;
}
```

Output: 1 2

Secció restringida: estructures de salt

Instrucció **continue**

- La instrucció **continue** transfereix l'execució directament a la següent iteració del bucle, saltant-se qualsevol codi que resti per executar dins del cos del bucle per a l'iteració actual.
- La diferència amb el **break** és que **continue** se salta la iteració actual (i seguim fent iteracions), mentre que **break** surt del bucle totalment (i ja no fem més iteracions).



En aquest curs, està **prohibit** fer servir **continue** en qualsevol cas

Quin resultat genera això?

```
#include <stdio.h>
#define N 5
int main() {
    int arr[N] = {1, 2, 3, 4, 5};
    int i;

    for(i = 0; i < N; i++) {
        if (arr[i] == 3) {
            continue;
        }
        printf("%d ", arr[i]);
    }
    /* El continue va a parar aquí*/
}
return 0;
```

Output: 1 2 4 5

Secció restringida: estructures de salt

Instrucció goto

- La instrucció **goto** està totalment desaconsellada en la programació estructurada. Possiblement només la fareu servir en assemblador.



Està **prohibit** fer servir **goto** en qualsevol circumstància

- Serveix per fer un salt incondicional a una altra part del programa que hem etiquetat prèviament.

Només a tall d'exemple "històric".



```
#include <stdio.h>
#define N 5

int main() {
    int arr[N] = {1, 2, 3, 4, 5};
    int i;

    for(i = 0; i < N; i++) {
        if (arr[i] == 3) {
            goto acabar;
        }
        printf("%d ", arr[i]);
    }

acabar:
    printf("\nSalto aquí. Feu el favor
de no fer servir goto.\n");
    return 0;
}
```

Alternatives a les instruccions de salt

Si no podem fer servir les instruccions de salt, com ho fem?

Alternativa a "break"

- Generalment es fa servir un break per sortir d'un bucle quan hi ha una condició addicional per sortir.
- Per evitar fer servir break per sortir, hem d'explicitar les dues condicions:



```
while (condició1) {  
    if (condició2) {  
        break;  
    }  
    ...  
}
```



```
while (condició1 && !condició2) {  
    ...  
}
```

Alternatives a les instruccions de salt

Si no podem fer servir les instruccions de salt, com ho fem?

Alternativa a "continue"

- Generalment s'utilitza quan hi ha una condició que fa que no volguem executar el codi que normalment executem en el bucle
- S'utilitza per saltar a la següent iteració d'un bucle, evitant la resta del codi.

```
● ● ●  
  
for (int i = 0; i < n; i++) {  
    if (condició){  
        // Em salto aquesta iteració  
        continue;  
    }  
    // Aquí codi  
}
```

- Per evitar fer servir continue, neguem la condició i posem el codi dins. Quan la condició no sigui certa, ja no executarem aquell codi, sense necessitat de continue.

```
● ● ●  
  
for (int i = 0; i < n; i++) {  
    if (!condició) {  
        // Codi  
    }  
}
```

Cerca vs. recorregut

Tractament de seqüències

- A l' hora de tractar seqüències, qualsevol tractament que haguem de fer es pot incloure en una d'aquestes dues categories:

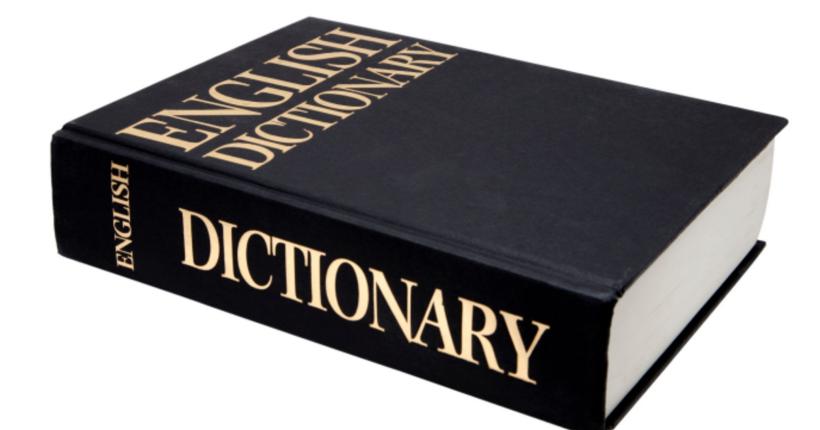
RECORREGUT

- En problemes de recorregut, necessitem visitar **tots** els elements de la seqüència, del primer a l'últim.
- Si deixem de visitar-ne algun o sortim del procés massa aviat, el resultat és incorrecte
- Sabem el nombre d'iteracions que farem
- Per exemple:
 - Calcular la mitjana d'una seqüència de nombres.
Necessitem visitar-los tots o la mitjana serà incorrecta.

$(2, 3, -5, 4, 8, 9, -1)$

CERCA

- En problemes de cerca, el procés **s'atura** quan es troba un element que compleix la condició
- Si no trobem aquest element (la condició no es compleix), acabarem recorrent tota la seqüència
- No sabem quantes iteracions farem en el procés
- Per exemple:
 - Trobar la definició de la paraula "casa" al diccionari. *Un cop l'hem trobat, no cal que seguim mirant les altres paraules.*



Donada una col·lecció de llibres, comptar
quants van ser publicats l'any 2000

RECORREGUT

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

CERCA

Comprovar si hi ha valors repetits en una seqüència de números

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Buscar al diccionari quantes paraules acaben amb el sufix "-isme".

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una seqüència de números

Trobar la temperatura mínima d'un vector de temperatures

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

Buscar al diccionari quantes paraules acaben amb el sufix "-isme".

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una seqüència de números

Saber si en un vector hi ha algun zero

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

Buscar al diccionari quantes paraules acaben amb el sufix "-isme".

Trobar la temperatura mínima d'un vector de temperatures

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una seqüència de números

Calcular la mitjana d'una seqüència de nombres introduïts per teclat

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

Buscar al diccionari quantes paraules acaben amb el sufix "-isme".

Trobar la temperatura mínima d'un vector de temperatures

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una seqüència de números

Saber si en un vector hi ha algun zero

Saber si dues seqüències de nombres
són iguals

RECORREGUT

Donada una col·lecció de llibres, comptar
quants van ser publicats l'any 2000

Buscar al diccionari quantes paraules
acaben amb el sufix "-isme".

Trobar la temperatura mínima d'un
vector de temperatures

Calcular la mitjana d'una seqüència de
nombres introduïts per teclat

CERCA

Saber si, al diccionari, hi ha més de 10
paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una
seqüència de números

Saber si en un vector hi ha algun zero

RECORREGUT

Donada una col·lecció de llibres, comptar quants van ser publicats l'any 2000

Buscar al diccionari quantes paraules acaben amb el sufix "-isme".

Trobar la temperatura mínima d'un vector de temperatures

Calcular la mitjana d'una seqüència de nombres introduïts per teclat

CERCA

Saber si, al diccionari, hi ha més de 10 paraules que comencin amb la lletra "Y"

Comprovar si hi ha valors repetits en una seqüència de números

Saber si en un vector hi ha algun zero

Saber si dues seqüències de nombres són iguals

per vs. mentre

AKA Recorregut vs. cerca

- Puc escriure el mateix bucle fent servir "per" i "mentre". Quin he de fer servir doncs?

TEAM "PER"

```
per (inicialització; condició; actualització) fer
    Cos del bucle
fper
```

TEAM "MENTRE"

```
inicialització
mentre (condició) fer
    Cos del bucle
    Actualització
fmentre
```

Els bucles **per** es fan servir quan tenim una **iteració definida**.

Vol dir que sabem el nombre d'iteracions que farem.

*E.g. Comptar quants nombres menors que 100 són múltiples
de 3.*

Els bucles **mentre** es fan servir quan tenim
una **iteració indefinida**.

No sabem quantes iteracions farem.

*E.g. esbrinar si en una paraula hi ha
alguna majúscula*

per vs. mentre

Què és això?



```
#include <stdio.h>
#include <stdbool.h>
#define N 10
#define NUM_SECRET 6

int main() {
    int i;
    bool trobat;
    for (i=0, trobat=false; i<N && !trobat; i++){
        if(i == NUM_SECRET){
            trobat=true;
        }
    }
    return 0;
}
```

Un "for" amb dues condicions!
Això és un "while" disfressat!



Si necessitem posar més d'una condició a un **for** és que estem davant d'un esquema de cerca i no de recorregut, i ens indica que hem de fer servir un **while**.