

Elements de Sistemes Operatius

Capítol 4

En el principi fou el C

J. Fornés M. Farreras Y. Becerra

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Grau en Enginyeria Informàtica — Sistemes Operatius

Versió de prova per a lectors · 23 de setembre de 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

Aquesta obra està subjecta a la llicència de Creative Commons Reconeixement-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.ca>

Aquest text s'ha escrit amb l'ajuda de Claude, el model de llenguatge d'Anthropic. Tots els errors que quedin són dels autors.

4 En el principi fou el C

C is not a big language, and it is not well served by a big book.

— Brian Kernighan i Dennis Ritchie,
The C Programming Language [1]

Aquest capítol no us ensenyarà a programar. Això ja ho sabeu fer: veniu de primer, on heu escrit uns quants milers de línies de Java i on us han explicat què és una variable, una condicional i un bucle. El que us cal ara és una altra cosa, més petita i més ràpida: passar aquell coneixement a un llenguatge nou.

I us caldrà unes quantes vegades més al llarg de la carrera. Parem-nos, doncs, un moment a pensar com es fa això, en comptes de fer-ho a batzegades, buscant a internet cada línia que no surt.

4.1 D'on surt el C

El 1969, quan en Ken Thompson va escriure la primera versió de Unix en llenguatge ensamblador del PDP-7 — la història és al capítol 1 —, tots els sistemes operatius del món estaven escrits en ensamblador. No era una decisió: era l'única manera de fer-ho. Un sistema operatiu toca registres, interrupcions i adreces físiques, i cap llenguatge d'alt nivell de l'època no us hi deixava arribar.

En Thompson va escriure un llenguatge, el B, per no haver de tornar a començar de zero a cada màquina. Era una simplificació del BCPL de Cambridge i tenia un problema seriós: no distingia tipus de dades. Tot era una paraula de màquina. Sobre un PDP-11, que sabia adreçar bytes, allò no s'aguantava.

Entre el 1971 i el 1973 en Dennis Ritchie va anar afegint tipus al B fins que allò va deixar de ser el B i va passar a dir-se C [2]. I el 1973

va passar el que ningú no havia fet abans: en Thompson i en Ritchie van reescriure el kernel de Unix en C.

El C no és un llenguatge que després es fes servir per escriure sistemes operatius. És un llenguatge que es va inventar *per* escriure un sistema operatiu, i totes les seves decisions — les bones i les dolentes — s’entenen des d’aquí.

D’aquella reescriptura en surt la portabilitat, que és el que va fer que Unix sortís dels Bell Labs i acabés arreu. I en surten també les rareses del llenguatge: que us deixi sumar 1 a una adreça, que no us comprovi si us en aneu d’una taula, que no tingui cadenes de caràcters. Un llenguatge que ha de saber escriure un byte concret en una posició concreta de memòria no us pot protegir gaire de vosaltres mateixos.

El llibre de referència segueix sent el de Kernighan i Ritchie [1]: unes dues-centes vuitanta pàgines que inclouen el manual del llenguatge sencer. Hi ha llibres de C de mil pàgines; desconfieu-ne.

4.2 Sis preguntes

Quan us trobeu davant d’un llenguatge imperatiu nou — i el C, el Java, el Python, el Go i el Rust ho són tots — hi ha sis preguntes que, contestades, us deixen escriure-hi qualsevol programa que ja sabíeu escriure:

1. Quins tipus de dades primitives suporta?
2. Quins tipus de dades estructurades hi puc definir?
3. Com hi faig instruccions seqüencials?
4. Com hi faig una condicional?
5. Com hi faig un bucle?
6. Com el compilo i en creo un executable?

Les cinc primeres són el llenguatge. La sisena no ho és — és l'entorn —, però sense ella no arribareu a executar res, i és on es perd més temps el primer dia. La resta d'aquest capítol són sis seccions, una per pregunta.

Que la recepta funciona es pot comprovar: l'annex B contesta aquestes mateixes sis preguntes per al Rust, que és el llenguatge que Linux ha començat a acceptar per als seus controladors. No cal per a aquesta assignatura; hi és per si algun dia us toca.

Compte

Aquestes sis preguntes són el mínim per *escriure*, no per *escriure bé*. D'un llenguatge també n'heu d'acabar aprenent com s'hi organitza el codi en mòduls, què hi és idiomàtic i què no, i com s'hi tracten els errors. Però això s'aprèn llegint codi dels altres, i per llegir-ne primer l'heu de saber desxifrar.

4.3 Quins tipus primitius?

Pocs, i tots numèrics.

Hi ha els enters — `char`, `short`, `int`, `long` i `long long` —, que podeu qualificar de `signed` o `unsigned`, i els reals — `float`, `double` i `long double`. I s'ha acabat.

Tres absències que us sorprendran, venint del Java:

- **No hi ha tipus booleà.** A les condicions, zero vol dir fals i qualsevol altra cosa vol dir cert. Des del C99 hi ha `bool` si incloeu `stdbool.h`, però per sota continua sent un enter, i el codi que llegireu no el fa servir.
- **No hi ha cadenes de caràcters.** El tipus `string` no existeix. Una cadena és una taula de `char` acabada en el byte zero, i tot el que hi vulgueu fer ho fareu amb funcions de la llibreria (`strlen`, `strcpy`).
- **El `char` és un enter.** Un byte, concretament. Podeu sumar-hi, restar-hi i comparar-lo, i és exactament el que farem al `c-counta` d'aquest capítol per anar comptant salts de línia.

L'absència que us sorprendrà més, però, no és de tipus sinó de mides: **l'estàndard no diu quants bytes ocupa un int**. Només diu quins valors ha de poder representar cadascun i que cap no pot ser més petit que l'anterior. La mida real la tria el compilador de la màquina on compileu, i per això un programa que doni per fet que un `int` ocupa quatre bytes es trencarà el dia que canvieu de màquina.

L'operador `sizeof` us dona la mida en bytes de qualsevol tipus o variable, i és la manera correcta de fer aquesta pregunta:

Llistat 4.1. Les mides, preguntades a qui les sap

```

1 static void mida(const char *nom, size_t bytes)
2 {
3     char buf[64];
4
5     sprintf(buf, "%-12s %2zu bytes\n", nom, bytes);
6     write(1, buf, strlen(buf));
7 }
8
9 int main(void)
10 {
11     mida("char", sizeof(char));           ❶
12     mida("short", sizeof(short));
13     mida("int", sizeof(int));
14     mida("long", sizeof(long));
15     mida("long long", sizeof(long long));
16     mida("float", sizeof(float));
17     mida("double", sizeof(double));
18     mida("void *", sizeof(void *));       ❷
19     mida("size_t", sizeof(size_t));
20     return 0;
21 }
```

- ❶ Aquesta línia sempre escriu 1: l'estàndard *sí* que fixa que `sizeof(char)` val 1. El byte és la unitat de mesura, i la resta de tipus es mesuren en chars.
- ❷ Un punter també és un tipus, i també té mida. En una màquina de 64 bits n'ocupa 8, i això és, literalment, el que vol dir «de 64 bits».

```

alumne@labS0:~/S2$ ./c-tipus
char          1 bytes
short         2 bytes
int           4 bytes
long          8 bytes
long long     8 bytes
float         4 bytes
double        8 bytes
void *        8 bytes
size_t        8 bytes

```

L'últim de la llista, `size_t`, no és un tipus primitiu: és el nom que la llibreria dóna al tipus enter sense signe prou gran per mesurar qualsevol objecte de la màquina. El trobareu a mitja capçalera del sistema, i tornarà a sortir a la pregunta següent.

4.4 Quins tipus estructurats?

Quatre constructors, que es poden combinar sense límit.

4.4.1 La taula

```

int  v[10];           /* deu enters, de v[0] a v[9] */
char nom[32];        /* trenta-dos caràcters */

```

La mida ha de ser coneguda en compilar*, i els índexs van de 0 a $n - 1$, com al Java.

Compte

El que **no** és com al Java: ningú no comprova que l'índex hi càpiga. Si escriviu a `v[10]`, el C us deixa: escriureu sobre els bytes que hi hagi després de la taula, siguin de qui siguin. El programa no

*Des del C99 no sempre: hi ha taules de mida variable. No les farem servir.

petarà en aquell moment; petarà més tard i en un altre lloc. La meitat de les hores que passareu depurant en aquesta assignatura sortiran d'aquí.

4.4.2 El punter

Una variable que conté una adreça de memòria. Es declara amb un asterisc i té dos operadors:

```
int a = 4;
int *p;           /* p és un punter a enter */

p = &a;           /* & : l'adreça de a */
*p = 7;           /* * : el que hi ha a l'adreça p */
                  /* ara a val 7 */
```

El punter és la peça que fa possible tota la resta. Un programa en C no passa mai una estructura gran a una funció: hi passa l'adreça. La crida al sistema `read()` no us torna els bytes llegits: li doneu l'adreça d'on els ha de deixar. Entretingueu-vos-hi ara, perquè a partir del capítol 5 ja no en sortirem.

Taula i punter estan estretament emparentats: el nom d'una taula, fet servir dins d'una expressió, és l'adreça del seu primer element. Per això una funció que rep una cadena la rep com a `char *` i no com a `char []`, i per això, dins d'aquella funció, ja no hi ha manera de saber quants elements tenia.

4.4.3 L'estructura

Un grapat de variables amb un nom col·lectiu, com els camps d'una classe sense mètodes:

```
struct punt {
    int x;
```

```
int y;
};

struct punt p;
p.x = 3;          /* amb un punter seria q->x = 3 */
```

N'hi ha una variant, la **union**, on tots els camps comparteixen la mateixa memòria i només un és vàlid a la vegada. La veureu a les estructures del kernel, on l'estalvi de memòria compta.

4.4.4 El nom nou

typedef no crea cap tipus: bateja un que ja existeix.

```
typedef unsigned long size_t;
typedef struct punt Punt;
```

Serveix per no haver d'escriure **struct punt** cinquanta vegades i, més important, per donar noms que expliquin què és una cosa en comptes de com està feta. Quan al manual llegiu que **read()** torna un **ssize_t**, el que us diu és «un enter prou gran per comptar bytes, i amb signe perquè el **-1** vol dir error»; quants bytes ocupi és cosa de la màquina.

Falta l'**enum**, que és una llista de constants enteres amb nom. El C no us impedirà assignar a una variable **enum** un valor que no sigui de la llista: recordeu d'on venim.

4.5 Com faig instruccions seqüencials?

Una instrucció és una expressió seguida d'un punt i coma. Un grup d'instruccions és un bloc, entre claus. Fins aquí, com al Java.

La sorpresa és que en C *tot* és una expressió, i per tant tot té valor. L'assignació el té:

```
b = (a = 3) + 1;    /* a val 3, b val 4 */
n = m = 0;         /* és el mateix que m = 0; n = m; */
```

Sembla una curiositat i no ho és: d'aquí surt l'idioma més freqüent del C de sistemes, que veureu a la secció dels bucles.

4.5.1 Les funcions

Una funció es declara amb el tipus que torna, el nom i els paràmetres:

Funció `error_i_exit`

```
void error_i_exit(const char *msg, int exit_status);
```

Això és un *prototip*: diu com es crida la funció, no què fa. Ha de ser visible *abans* de la primera crida, i per això els prototips viuen als fitxers `.h` i el codi als `.c`. Si no ho feu, el compilador no us podrà comprovar la crida i us ho dirà amb un avís d'*implicit declaration*.

Hi ha una funció amb nom fixat, `main`, que és per on comença tot:

Funció `main`

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

El *shell* del capítol 3 és qui li omple aquests dos paràmetres: `argc` és quantes paraules tenia la línia de comandes i `argv` és la taula amb aquelles paraules, comptant-hi el nom del programa a `argv[0]`. L'enter que torna el `main` és l'estat de sortida del programa, que la crida al sistema `wait()` farà arribar al pare (capítol 5); per conveni, 0 vol dir «tot bé».

4.5.2 Escriure per pantalla

En aquesta assignatura escriurem sempre amb la crida al sistema `write()`, mai amb `printf` ni amb `cout`.

No és per fer-vos la vida difícil. El `printf` és una funció de la llibreria de C que, quan li sembla, acabarà cridant `write()`; pel mig hi ha un buffer que decideix pel seu compte quan escriu de debò. Aquest buffer és precisament el que us amagarà l'ordre real de les coses el dia que tingueu dos processos escrivint alhora, que serà aviat. Amb `write()` veieu el sistema, que és de què va el llibre.

La feina es reparteix en dos passos. El `sprintf` formata el text i el deixa en una taula de `char`; el `write()` envia aquella taula a un canal:

```
char buf[128];

sprintf(buf, "L'exemple número %d\n", 1);
write(1, buf, strlen(buf));
```

Fixeu-vos en el color: el `write()` surt en vermell i el `sprintf` no. És el criteri de tot el llibre — en vermell les crides al sistema, en negre les funcions de llibreria —, i aquí es veu bé la diferència que hi ha entre les dues coses. El `sprintf` no surt del vostre procés. El `write()` sí: demana al kernel que escrigui.

Aquest parell sortirà a cada exemple del llibre, i hi sortirà així, sencer, com al laboratori: el `write()` és justament el que us interessa veure.

El que sí que tenim empaquetat és l'altra cosa que es repeteix a tot arreu. Una crida al sistema pot fallar, i quan falla torna `-1` i deixa el motiu a l'`errno`; el `perror` el tradueix a un missatge. Als exemples, quan una crida falla no hi ha res més a fer que dir-ho i plegar, i per això tots fan servir la mateixa funció del laboratori, que és a `codi/eines.h`:

Llistat 4.2. L'única eina de la casa

```

1 void error_i_exit(const char *msg, int exit_status)
2 {
3     perror(msg);
4     exit(exit_status);
5 }

```

El segon paràmetre és l'estat de sortida que rebrà el pare; als exemples sempre és 1, que vol dir «alguna cosa ha anat malament».

I aquí surt el tercer color, que és l'últim: l'`error_i_exit` i el `perror` van en **blau** perquè no són ni una cosa ni l'altra. L'`error_i_exit` és *nostra* —la trobareu a `codi/eines.h` i enlloc més—, el `perror` és de la llibreria, i totes dues fan el mateix ofici: dir què ha fallat. Que no siguin vermelles vol dir el que sembla que vol dir: qui travessa la frontera és el `write()` que el `perror` fa per sota, no elles.

Amb això ja podem escriure el primer programa complet. És l'esquelet de tots els que fareu al laboratori: comprovar els arguments, fer la feina i tornar un estat.

Llistat 4.3. L'esquelet de qualsevol programa

```

1 static void Usage(const char *jo)                                ❶
2 {
3     char buf[128];
4
5     sprintf(buf, "Usage: %s nom [nom...]\n"
6               "Saluda cadascun dels noms que rep.\n", jo);
7     write(2, buf, strlen(buf));                                  ❷
8     exit(1);
9 }
10
11 int main(int argc, char *argv[])                                ❸
12 {
13     char buf[128];
14     int i;
15
16     if (argc < 2)                                                ❹
17         Usage(argv[0]);

```

```
18
19     for (i = 1; i < argc; i++) {
20         sprintf(buf, "Hola, %s!\n", argv[i]);
21         write(1, buf, strlen(buf));
22     }
23     return 0;
24 }
```

- ❶ La funció `Usage` és un conveni del laboratori: quan els arguments no són els que toquen, diu com s'ha de cridar el programa, explica què fa i plega.
- ❷ Els missatges d'error van al canal 2, no a l'1. Així, si algú redirigeix la sortida a un fitxer — capítol 3 —, l'error se segueix veient a la pantalla.
- ❸ La signatura de sempre. Els noms `argc` i `argv` són lliures; els tipus, no.
- ❹ `argc` val 1 quan no heu escrit cap argument, perquè el nom del programa hi compta.
- ❺ Estat 0: tot bé. La `Usage`, en canvi, acaba amb estat 1.

```
alumne@labS0:~/S2$ ./c-hola Jordi món
Hola, Jordi!
Hola, món!
alumne@labS0:~/S2$ ./c-hola
Usage: ./c-hola nom [nom...]
Saluda cadascun dels noms que rep.
alumne@labS0:~/S2$ echo $?
1
```

4.6 Com faig una condicional?

```
if (a > b)
    gran = a;
else
    gran = b;
```

Sense sorpreses, tret d'una: la condició no és un booleà, és un enter, i val cert si no és zero. Per tant això altre és legal, compila i no fa el que sembla:

```
if (a = b)      /* assigna b a a, i mira si el resultat és zero */
```

i Compte

El clàssic = on volíeu == és, de llarg, l'error que més temps us farà perdre. Compileu sempre amb **-Wall**: el gcc us avisa d'aquest cas concret.

Hi ha també el **switch**, per triar entre valors enters constants, amb dos paranyes: els casos **cauen** d'un a l'altre si no hi poseu un **break**, i el **default** és opcional.

I hi ha l'operador ternari, que és una condicional que té valor i que per tant es pot posar dins d'una expressió:

```
gran = (a > b) ? a : b;
```

4.7 Com faig un bucle?

Els tres de sempre:

```
while (condició) { ... }      /* comprova, i potser no entra */
do { ... } while (condició);  /* entra, i després comprova */
for (i = 0; i < n; i++) { ... } /* inicialitza; comprova; avança */
```

El **for** del C no és el del Java modern: no recorre colleccions, no hi ha **for each**. Les tres parts són expressions qualssevol i qualsevol de les tres pot ser buida; **for (; ;)** és un bucle infinit. Dins de tots tres, **break** en surt i **continue** salta a la iteració següent.

I ara sí, l'idioma que us anunciava a la secció de les instruccions:

```
while ((llegits = read(0, buf, MIDA)) > 0) { ... }
```

Es llegeix de dins cap enfora: crida el `read()`, en desa el resultat a `llegits` i, com que l'assignació és una expressió amb valor, compara *aquell mateix* valor amb zero. En una línia hi ha la lectura, el guardat i la condició de sortida. Trobareu aquesta forma a tot el codi de sistemes que llegiu, i val més que la reconegueu ara que no pas que us la trobeu pel camí.

Aquest és el programa sencer. Compta línies i bytes de l'entrada estàndard, i no obre cap fitxer: llegeix del canal 0, sigui el que sigui.

Llistat 4.4. Un `wc -l` de butxaca

```

1  #define MIDA 512
2
3  int main(void)
4  {
5      char buf[MIDA], sortida[64];
6      ssize_t llegits;
7      long bytes = 0, linies = 0;
8      int i;
9
10     while ((llegits = read(0, buf, MIDA)) > 0) {
11         bytes += llegits;
12         for (i = 0; i < llegits; i++)
13             if (buf[i] == '\n')
14                 linies++;
15     }
16     if (llegits < 0)
17         error_i_exit("read", 1);
18
19     sprintf(sortida, "%ld línies, %ld bytes\n", linies, bytes);
20     write(1, sortida, strlen(sortida));
21     return 0;
22 }
```

- ❶ `ssize_t` i no `int`: el `read()` torna quants bytes ha llegit, i `-1` si hi ha hagut un error. Necessita signe.

- ② L'idioma. El bucle s'acaba quan el `read()` torna 0, que vol dir que no hi ha més dades.
- ③ Aquí es veu que el `char` és un enter: el comparem amb la constant de caràcter `'\n'`, que val 10.
- ④ Sortir del bucle no vol dir que tot hagi anat bé. Si el `read()` ha tornat `-1`, ho hem de dir.

I com que el programa no sap d'on li vénen les dades, el *shell* pot decidir-ho per ell, amb les redireccions del capítol 3 o amb una pipe de les del capítol 10:

```
alumne@labS0:~/S2$ ./c-conta < /etc/passwd
82 línies, 4970 bytes
alumne@labS0:~/S2$ ls /bin | ./c-conta
6808 línies, 77694 bytes
```

4.8 Com compilo i creo un executable?

Quan escriviu `gcc -o prog prog.c` sembla que passi una sola cosa. En passen quatre, i us convé saber quina us ha fallat:

1. El **preprocessador** treballa sobre el text. Copia els fitxers dels `#include` allà on són, substitueix els `#define` i decideix quins trossos de codi sobreviuen als `#ifdef`. No entén de C: mou text.
2. El **compilador** tradueix el resultat a ensamblador de la màquina. És qui es queixa dels errors de sintaxi i de tipus.
3. L'**ensamblador** en fa codi màquina: un fitxer objecte, `.o`, amb la llista dels noms que ofereix i dels que li falten.
4. L'**enllaçador** ajunta els fitxers objecte i les llibreries, resol els noms pendents i escriu l'executable. Si us diu *undefined reference*, us ha arribat fins aquí una funció que ningú no ha implementat.

Les cinc opcions del `gcc` que us demanaran al laboratori són `-c`, `-o`, `-I`, `-L` i `-l`:

```
# només fins al .o: preprocessa, compila i assembla
gcc -Wall -c prog.c

# enllaça els objectes i escriu l'executable «prog»
gcc -o prog prog.o eines.o

# -I: on ha de buscar els fitxers .h
gcc -Wall -c -I./include prog.c

# -L: on ha de buscar les llibreries. -l: quina llibreria
gcc -o prog prog.o -L/usr/local/lib -lm
```

La `-l` és la que sorprèn: `-lm` vol dir «la llibreria `m`», que al disc es diu `libm.so` o `libm.a`. El prefix `lib` i l'extensió els posa l'enllaçador.

4.8.1 Per què separar `.h` i `.c`

Perquè el compilador processa un fitxer cada cop i no en sap res dels altres. Del vostre mòdul, la resta del programa n'ha de conèixer els prototips — com es criden les funcions — però no el cos. Els prototips van al `.h`, que inclou tothom qui les faci servir; el cos va al `.c`, que es compila una sola vegada i produeix un `.o` que l'enllaçador enganxa a qui el necessiti.

Un `.h` sempre porta la protecció que heu vist a `codi/eines.h`, perquè incloure'l dues vegades no doni error:

```
#ifndef EINES_H
#define EINES_H
... prototips ...
#endif
```

Si acabeu tenint uns quants `.o` que van sempre junts, la comanda `ar` els empaqueta en una llibreria (`libeines.a`) que ja podreu passar al `gcc` amb `-L` i `-l`.

I abans de lliurar res, passeu-hi l'**indent**: sagna el fitxer sencer amb un criteri uniforme. Un codi mal sagnat no és un problema estètic; és un codi on la clau que falta no es veu.

4.9 El makefile

Ja teniu totes les peces, i ja veieu el problema: cada cop que toqueu una línia hauríeu de recordar quins .o s'han de refer i en quin ordre. Amb tres fitxers és avorrit; amb trenta és impossible.

La comanda **make** ho fa per vosaltres a partir d'un fitxer de text que es diu **makefile** i que conté **regles**. Una regla té tres parts:

```
objectiu: dependència1 dependència2 ... dependènciaN
[TAB] com generar l'objectiu a partir de les dependències
```

i Compte

Aquell [TAB] és un tabulador de debò, no vuit espais. És l'error més freqüent del primer makefile de cadascú, i el **make** el denuncia amb un missatge que no ajuda gens: *missing separator*.

La regla mínima diu que **c-hola** depèn de **c-hola.c** i que es fa amb un **gcc**:

```
c-hola: c-hola.c
        gcc -o c-hola c-hola.c
```

El que fa el **make** amb això és una sola cosa, i convé dir-la bé: mira les dates de l'objectiu i de les dependències i **només executa la comanda si alguna dependència és més nova que l'objectiu**. Si no heu tocat res, us dirà que no hi ha res a fer. D'aquí surt tota

la resta: les dependències no són decoració, són el que li permet no recompilar el que ja està bé. I si les poseu malament el **make** no us avisarà: simplement us deixarà un executable vell amb el codi nou, i us hi passareu la tarda.

Les regles s'encadenen. Si un objectiu depèn d'un fitxer que és, al seu torn, objectiu d'una altra regla, el **make** hi baixa primer. Amb això, i amb dues variables per no repetir el nom del compilador i les opcions, ja teniu el makefile de la sessió de laboratori:

Llistat 4.5. Un makefile complet, i prou

```
# el mínim per construir el c-conta del capítol 3
CC      = gcc
CFLAGS  = -Wall -Wextra -g

c-conta: c-conta.o eines.o
        $(CC) -o c-conta c-conta.o eines.o

c-conta.o: c-conta.c eines.h
        $(CC) $(CFLAGS) -c c-conta.c

eines.o: eines.c eines.h
        $(CC) $(CFLAGS) -c eines.c

clean:
        rm -f c-conta *.o
```

Tres coses més i ho deixem:

- **El primer objectiu del fitxer és el que es fa per defecte.** **make** sense arguments fa el **c-conta**; **make eines.o** fa només això.
- **Una variable** es defineix amb **CC = gcc** i es fa servir amb **\$(CC)**. Serveix per canviar les opcions en un sol lloc.
- **Hi ha objectius que no són fitxers.** El **clean** no genera cap fitxer que es digui **clean**: és una comanda amb nom. Com que el fitxer no existeix mai, sempre es considera desfasat i sempre s'executa.

▲ Qüestió 1

Si escriviu `make` dues vegades seguides sense tocar cap fitxer, la segona no fa res. Feu ara `touch eines.h` i torneu-hi. Què recompila, i per què?

💡 Resposta

Refà `c-conta.o` i `eines.o`, perquè totes dues regles tenen `eines.h` com a dependència i ara és més nou que els `.o`. I un cop refets, els `.o` són més nous que `c-conta`, de manera que també torna a enllaçar. El `make` no ha mirat el contingut de cap fitxer: només dates.

4.10 Exercicis

1. Compileu i executeu el `c-tipus` del llistat 4.1 a la vostra màquina i al laboratori. Surt el mateix? Hi ha alguna línia que hagi de sortir igual a tot arreu?
2. Afegiu al `c-tipus` una línia per a `sizeof(struct punt)`, amb l'estructura de dos enters d'aquest capítol. Us surt la suma de les mides dels camps? Si no, busqueu què vol dir *alineació*.
3. Modifiqueu el `c-conta` perquè compti també paraules. Direm que una paraula s'acaba en un espai, un salt de línia o un tabulador. Compareu el resultat amb el del `wc`.
4. Traieu el `-Wall` del makefile i canvieu el `==` de la línia dels salts de línia per un `=`. Compila? Què compta ara el programa? Torneu a posar el `-Wall` i mireu què us diu el compilador.
5. Al makefile de l'exemple, esborreu `eines.h` de les dependències de `c-conta.o`. Afegiu ara un paràmetre a l'`error_i_exit`, tant al `.h` com al `.c`. Feu `make`. Què passa, i per què és pitjor que un error de compilació?

Bibliografia

- [1] Brian W. Kernighan i Dennis M. Ritchie. *The C Programming Language*. 2a ed. Englewood Cliffs, NJ: Prentice Hall, 1988. ISBN: 0-13-110362-8.
- [2] Dennis M. Ritchie. «The Development of the C Language». A: *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, 1993, pàg. 201 - 208. DOI: [10.1145/154766.155580](https://doi.org/10.1145/154766.155580).