Departament d'Enginyeria
[DΣIM]
Informàtica i Matemàtiques
Universitat Rovira i Virgili

INTRODUCCIÓ AL LLENGUATGE C

Edició 2000

Documentació del seminari

Santiago Romaní, Pere Millán

sromani@etse.urv.es, pmillan@etse.urv.es

Pròleg d'edicions anteriors

Aquest document conté els apunts utilitzats durant el Seminari d'Introducció al Llenguatge C, dirigit als alumnes de segon curs de les Enginyeries Tècniques en Informàtica. És un seminari de caràcter introductori, per tal de proporcionar una base als alumnes, que hauran d'utilitzar aquest llenguatge per realitzar moltes de les pràctiques dels ensenyaments d'informàtica.

Encara que els apunts són un complement de les classes impartides al seminari, aquests estan realitzats de forma que es puguin anar seguint de forma autodidacta.

És necessari tenir coneixements previs de programació estructurada i llenguatge màquina doncs el seminari no pretén ensenyar a programar, sinó donar coneixements elementals de les característiques i peculiaritats del llenguatge C.

Els apunts s'estructuren en quatre capítols, donant un especial èmfasi al tema de punters, un dels punts més problemàtics i menys entesos a l'hora de programar en C.

Es disposa també d'una versió HTML d'aquests apunts, accessible per Internet a l'adreça `http://www.etse.urv.es/EngInf/assig/ec/SeminariC/`.

Pròleg de l'edició 2000

L'any 1998 es va encetar la reforma dels plans d'estudi de les Enginyeries Tècniques en Informàtica a la URV. A conseqüència d'aquests canvis les assignatures a les quals anava dirigit aquest seminari (les de la branca d'Arquitectura de Computadors, començant per Estructura de Computadors, impartida el primer quadrimestre del segon curs) han variat la seva ubicació al nou pla d'estudis 1998.

Actualment aquest document va dirigit als alumnes que estudien Estructura de Computadors I, el segon quadrimestre del primer curs. Els coneixements previs a que es fa referència s'imparteixen a les assignatures Computadors i Programació I.

La localització dels apunts en versió electrònica també ha variat, trobant-se ara a l'adreça `http://www.etse.urv.es/~pmillan/SeminariC/` *(de totes formes, es mantenen enllaços a aquesta nova ubicació des de les planes web de les assignatures de pla antic i pla nou: EC i EC I).*

El llenguatge C ha anat evolucionant des de la seva invenció, desenvolupant-se el C++ (bàsicament programació orientada a objectes en C) i el tant de moda llenguatge Java (basat en C++ i lligat a Internet). Aquesta documentació pot servir com a base per conèixer la sintaxi i funcionament bàsics del llenguatge i posteriorment passar a l'estudi de les seves evolucions: C++ i Java.

Pere Millán, Tarragona, abril de 2000.

Seminari d'introducció al llenguatge C.

Per SANTIAGO ROMANÍ i PERE MILLÁN.

Presentació del seminari:

Aquest seminari està destinat als alumnes de 2n curs d'Enginyeria Tècnica Informàtica (Sistemes i Gestió), i requereix coneixements previs en les següents matèries:

- **programació estructurada** (Introducció a la Programació),
- **llenguatge màquina** (Computadors II).

Té un caràcter introductori (no pretén ser un curs avançat), però ha de servir per proporcionar una bona base als alumnes que hauran de fer moltes de les pràctiques de 2n i 3r curs amb aquest llenguatge.

Per impartir el seminari es dediquen 12 hores, distribuïdes en sessions de 2 hores cadascuna. L'assignació d'hores a cadascuna de les 4 seccions que composen el seminari és de 3, 3, 4 i 2 hores respectivament.

Aquest seminari està basat principalment en el llibre "*El lenguaje de programación C*", de Brian W. Kernighan i Dennis M. Ritchie, Ed. Prentice Hall (2 ª edició traduïda al castellà), signatura biblioteca ETSE: `1203.23 C KER`. A més, aquests apunts inclouen les transparències utilitzades durant el seminari, amb alguns comentaris sobre cada exemple.

Els programes necessaris per aquest seminari (continguts al fitxer `PROGS_C.EXE`, fitxer autocomprimit que caldrà descomprimir executant-lo amb l'opció `-d` per crear els directoris corresponents) es poden aconseguir de dues formes: la primera accedint per Internet a la plana *World Wide Web* de l'assignatura d'Estructura de Computadors (`http://www.etse.urv.es/EngInf/assig/ec/`) i a partir d'aquesta plana anar a l'apartat corresponent al Seminari de C; ~~per utilitzar la segona cal tenir un compte d'usuari a la màquina `TROLL`, i accedir al directori `/disc1/assig/ec/SeminariC` mitjançant un programa de transferència de fitxers amb protocol FTP (en mode binari). Si es vol una còpia de les eines (editor, compilador, enllaçador - *linker* -, depurador, etc.) per desenvolupar programes en C, caldrà obtenir el fitxer `EINES_C.EXE` del directori del `TROLL` mencionat abans. Aquest directori també conté els fitxers `INSTRUCCIONS` i `INDEX`, fitxers de text on es troben les instruccions per descomprimir i una guia dels fitxers que es subministren~~.

La millor forma d'aprendre és anar posant en pràctica tot el que es planteja a la teoria. Així doncs, agafeu aquests apunts, un ordinador amb el compilador de C i els codis dels programes i proveu tot el que s'explica i el que es vulgui experimentar per acabar d'entendre-ho millor.

Contingut: (12 hores)

Secció 1. Programació bàsica. (3 hores)

1.1. Tipus de dades bàsics.
1.2. Operadors.
1.3. Control de flux.
1.4. Esquema bàsic d'un programa
1.5. Exercicis.

Secció 2. Programació estructurada. (3 hores)

2.1. Taules (*array*s) i *string*s.
2.2. Dades estructurades.
2.3. Funcions.
2.4. Àmbit de les variables.
2.5. Exercicis.

Secció 3. Punters. (4 hores)

3.1. Concepte i ús dels punters.
3.2. Punters a taules.
3.3. Punters a *string*s.
3.4. Punters a tuples.
3.5. Punters a paràmetres.
3.6. Dades dinàmiques.
3.7. Exercicis.

Secció 4. Accés a fitxers i Programació modular. (2 hores)

4.1. Entrada i sortida (fitxers).
4.2. Compilació separada.
4.3. Llibreries estàndard.

ÍNDEX:

1. Programació bàsica.

1.1. Tipus de dades elementals.

TIPUS	BYTES	RANG
char (caràcter)	1	-128 a 127
int (enter)	2	-32768 a 32767
long (enter llarg)	4	-2147483648 a 2147483647
short (enter curt)	2	-32768 a 32767
float (real)	4	3.4 exp -38 a 3.4 exp 38
double (real doble precisió)	8	1.7 exp -308 a 1.7 exp 308
void (no res)	-	-

- **`void`** és el tipus de dades buit, i es fa servir per expressar que una funció no retorna res, o no rep cap paràmetre, etc.

- Possibilitat de definir **`signed`** o **`unsigned`**. **`unsigned`** especifica que només s'admetran valors positius i, per tant, permet treballar amb el rang ampliat (per exemple, **`unsigned char`** tindrà un rang de 0 a 255).

- Definició de variables: primer es posa el tipus i després el nom de les variables d'aquell tipus (**`float`** és el tipus i **`real`** és el nom de la variable).

```
int           i,lon,numero;
char          c;
unsigned char c2,kar;
float         real;
double        real_gran;
```

1.2. Operadors.

Aritmètics

- Suma: `+`
- Resta: `-`
- Multiplicació: `*`
- Divisió: `/`
- Mòdul: `%`
- Autoincrement: `++` /* augmenta una variable en 1 */
- Autodecrement: `--` /* disminueix una variable en 1 */

Exemples:

```
int a,b,c;                      /* declaració de tres variables enteres */

a=2;
b=a*6;
c=(b+a)/(b-a);                  /* divisió entera */
c++;                            /* c = c + 1 */
```

Comparació

- Igual: `==` /* diferent del signe igual d'assignació */
- Diferent: `!=`
- Menor o igual: `<=`
- Major o igual: `>=`
- Major: `>`
- Menor: `<`

Lògics

- AND: `&&` /* AND sobre condicions lògiques */
- OR: `||`
- NOT: `!`

Exemples:

si **a** és igual a **b** i (no és cert que **b** és major que **c** o és cert que **a** és diferent de 6);

```
if ((a==b) && (!(b>c) || (a!=6)))
```

Assignació

- Assignació: `=`
- Multiplicar i assignar: `*=`
- Dividir i assignar: `/=`
- Mòdul i assignar: `%=`
- Restar i assignar: `-=`
- Sumar i assignar: `+=`
- etc. /* qualsevol operació en què una variable */ /* apareix com a font i com a destí */

Exemples:

```
x /= 2;          equival a     x = x/2;
x *= y+1;        equival a     x = x*(y+1);
```

Tractament de bits

- Operació AND: `&` /* AND bit a bit de 2 valors */
- Operació OR: `|`
- Operació XOR: `^`
- Operació NOT: `~` /* NOT bit a bit */
- Desplaçament a la dreta: `>>` /* desplaçar un cert número de bits */
- Desplaçament a l'esquerra: `<<`

Exemples:

```
int x,y;

x = ~((y & 0x001f) << 3);   /* x = y AND 001f hexa, desplaçat 3 bits
                               a l'esquerra, i tots els bits negats */
y >>= 1;                    /* desplaçar y un bit a la dreta (dividir per 2) */
```

Operador condicional

- Operació Si <condició> Llavors Retorna <A> Sinó Retorna <B>:

<condició> ? <A> : <B>

Exemples:

```
x = (a>b) ? a : 3;          /* si (a>b) llavors x=a ; sinó x=3 ; */
```

Resultat d'una operació

- Qualsevol expressió d'assignació entre parèntesi es converteix a la seva vegada en un operand per a una segona expressió, el valor del qual és el resultat de la primera expressió.

Exemples:

```
a=(b+=5);                 →   b=b+5; a=b;

a=(c++)*2;                →   a=c*2; c=c+1;
a=(++c)*2;                →   c=c+1; a=c*2;

a=3*(b=5+c*2)-1           →   b=5+c*2; a=3*b-1;

a=(b=c);                  →   b=c; a=b;
a=(b==c);                 →   si (b==c) llavors a=1;sinó a=0;
                              /* (b==c) retorna un booleà (FALS=0)
                                 (b=c) retorna el valor de b */

a=(b=(c=0)); → a=b=c=0; →   c=0; b=0; a=0;
                              /* sense parèntesi s'avalua de dreta a esquerra */
```

Conversió automàtica de tipus

- En expressions on estiguin involucrades variables i constants de diferents tipus, el C converteix automàticament totes les variables de l'expressió al tipus més gran, i després opera. El resultat es converteix posteriorment al tipus de la variable resultat.

Exemples:

```
char c,c2;
int i;
float r,r2;

c = 'A'             /* 'A' és una constant de tipus char; c pren per valor el
                       número ASCII (8 bits) de la lletra "A" */
```

```
c2= c+12;          /* c es converteix a enter, perquè 12 és un enter, se sumen
                   enters, i el resultat es converteix a caràcter (16 bits → 8 bits) */
i = c2;            /* el valor numèric 8 bits de c2 es passa a 16 bits cap a i */
r = i*c;           /* c es converteix a enter i el resultat de la multiplicació
                      es converteix a real */
c2= r/12.0;        /* es divideixen dos reals i el resultat es converteix en un
                      caràcter: enter de 8 bits (part entera del real, mòdul 256) */
r = 5/9;           /* es fa la divisió entera (5 i 9 són enters), i el resultat es
                      converteix en real (r = 0.0) */
r2= 5.0/9.0;       /* es fa la divisió real (5.0 i 9.0 són reals), i el resultat es
                      deixa com a real (r2  = 0.5555555555556) */
```

Exercici:

- Per practicar amb els operadors, proveu d'acabar el següent programa en C (escriure una única expressió per cada línia de comentari que comenci per '?'; per convertir la pregunta en un comentari, substituir l'interrogant per '/'). El procés consisteix en obrir un editor (NE.COM, per exemple), modificar el fitxer de text OPERADOR.C, compilar, linkar, i executar des de l'intèrpret de comandes del DOS (si no hi han errors de compilació!).

```
#include <stdio.h>
void main(void)
{
    char           c;         /*definició de variables */
    unsigned char  c2;
    int            i,j;
    float          real;
    double         real_gran;

?* assignar el caràcter 'Z' sobre la variable i */
?* assignar i*2 (desplaçament) a c i c2 simultàniament */
?* assignar CERT a j si i és més gran o igual que 90,
    altrament assignar FALS */
?* autoincrementar j, i assignar 100 dividit pel nou valor
    de j +4 a real (divisió entera) */
?* multiplicar real per 2, i assignar el seu nou valor
    dividit per 5 a real_gran (divisió real) */

    /* comandes d'impressió */
    printf("i: %d\n",i);
    printf("c: %c (%d)\n",c,c);
    printf("c2: %c (%d)\n",c2,c2);
    printf("j: %d\n",j);
    printf("real: %.2f\n",real);
    printf("real_gran: %.2f\n",real_gran);
}
```

- A continuació s'explica com treballar amb els programes del seminari. Es suposarà que els fonts de C es troben al directori "**`C:\SEMINARI`**", i separats per subdirectoris corresponents a les 4 seccions del seminari ("**`\SEC1`**", "**`\SEC2`**", ...). Es poden editar els fitxers directament sobre el directori on es troben; després, la compilació i el linkatge es farà de la següent forma (suposant que els programes implicats són accessibles amb el `PATH` que es tingui definit:

```
C:                          ;canviar directori de C: per defecte
C:\>CD SEMINARI\SEC1        ;al directori on es troba OPERADOR.C

C:\SEMINARI\SEC1>NE OPERADOR.C   ;editar...

C:\SEMINARI\SEC1>TCC OPERADOR.C  ;compilar i linkar...

C:\SEMINARI\SEC1>OPERADOR        ;provar programa
```

- El resultat de l'execució ha de ser el següent llistat per pantalla:

```
i: 90               ; valor enter de la lletra 'Z' (segons taula ASCII)
c: • (-76)          ; caràcter ASCII 180 <- i * 2 (rep. 8 bits Ca2 : -76)
c2: • (180)         ; caràcter ASCII 180 <- i * 2 (rep. 8 bits natural : 180)
j: 2                ; 1 <- CERT quan (i >= 90), autoincrementat
real: 32.00         ; 100 / (2+4) divisió entera, multiplicat per 2
real_gran: 6.40     ; 32 / 5 divisió real
```

1.3. Control de flux.

Estructura `if`

```
if (expressió booleana)
 {
      ( cos del LLAVORS )
 }
else                        /* optatiu */
 {
      ( cos del SINÓ )
 }
```

Si només hi ha una expressió, no són necessàries les claus. L'expressió booleana sempre ha d'anar entre parèntesi.

```
if (expressió booleana)
      (expressió del LLAVORS);
else
      (expressió del SINÓ);
```

Exemples:

ex 1.1: Només hi ha una expressió pel Llavors i una pel Sinó → no es posen claus:

```
if (a>=b) c='X';
 else     c='Y';    /* és equivalent a */    c=((a>=b)?'X':'Y');
```

ex 1.2: Una expressió booleana complexa, més vàries expressions dins la condició Llavors. No hi ha condició Sinó:

```
if ((a==b) && (!(b>c) || (a!=6)))
 {
    x*=a+b;
    y=x/(a-b) % 4;
 }
```

ex 1.3: Una expressió d'assignació dins d'una de comparació; s'executa l'expressió i es compara amb el resultat obtingut → tot amb una sola línia! (cal anar amb compte, però, amb aquestes "virgueries" del C):

```
if (a==(b=c))        /* és equivalent a */      b=c;
     a>>=2;                                     if (a==b) a>>=2;
```

ex 1.4: tres variants on la col·locació de les claus canvia la situació de l'expressió **b++**:

sempre	**si (a!=b) i (a!=c)**	**si (a!=b)**

`if (a==b) a>>=1;`
`else`
`  if (a==c) a>>=2;`
`  else`
`  a>>=3;`
`b++;`
	`if (a==b) a>>=1;`
	`else`
	`  if (a==c) a>>=2;`
	`  else`
	`  {`
	`  a>>=3;`
	`  b++;`
	`  }`
		`if (a==b) a>>=1;`
		`else`
		`  {`
		`  if (a==c) a>>=2;`
		`  else`
		`  a>>=3;`
		`  b++;`
		`  }`

ContraExemples:

cex 1.1: **`(a=b)`** no és una expressió d'avaluació, sinó d'assignació. Per tant, no retorna un booleà sinó el valor que pren la variable **`a`**. Si aquest valor és 0, la sentència `if` ho interpretarà com a FALS. Si el valor és diferent de 0, ho interpretarà com a CERT. En tots dos casos es copiarà el valor de **`b`** sobre **`a`**:

```
if (a=b) a*=2;
```

cex 1.2: Aquí el problema està en el punt i coma de darrera de l'expressió d'avaluació. Com que no hi han claus, s'interpreta que l'acció a realitzar en cas que es compleixi la condició és l'acció buida. Per tant, tant si la condició és certa com si no ho és la sentència `if` no fa res, i l'expressió **`a*=2`** s'executa sempre, ja que està fora del condicional:

```
if (a==b);
    a*=2;
```

Estructura `switch` (CASE)

```
switch (expressió)      /* una expressió qualsevol (en particular, una variable) */
 {
     case valor1 :      ...          /* cas (expressió==valor1) */
                        break;
     case valor2 :      ...
                        break;       /* break salta al final de l'estructura switch */
          ...

     case valorn :      ...
                        break;
     default    :                 /* Altrament de tots els altres casos (opcional) */
                        ...
 }
```

- Si no hi ha **`break`** al final de les sentències de cada cas, s'executaran les sentències del cas següent, ja que el codi generat per a tots els casos està consecutiu en memòria, i les paraules reservades **`case`** fan de simples etiquetes a on es salta (quan s'executa el `switch`) si es compleix la igualtat de l'expressió amb el valor de cada cas.

- Només es pot estalviar el `break` al final de les sentències de l'últim cas (sigui un `default` o no).

Exemples:

ex 1.5:

```
 switch (a)
  {
     case 0:   b++;                  /* cas de que a==0, incrementar b */
               c='A';                /* i assignar 'A' a la variable c */
               break;                /* sortir de l'estructura switch */
     case 1:   c='B';
               break;                /* cas a==1 → una sola sentència */
     case 3:   if(b<10)              /* cas a==3 → sentències complexes */
                {
                    if(c!='Z') c='X';
                    b-=2;
                }
               else
                    c='Y';
               b+=5;
               break;
     default:  c='D';                /* cas de que a no sigui 0 ni 1 ni 3 */
  }
```

ex 1.6: Aprofitant aquest funcionament tan particular del `switch` de C (en combinació amb el `break`), es pot implementar condicions "semicomplexes", és a dir, que no s'executi un conjunt de sentències només quan es produeix una igualtat exacta:

```
switch (c)
{
    case 'A':
    case 'B': a++;          /* a++ si c=='A' o c=='B' */
    case 'C': b++;          /* b++ si c=='A' o c=='B' o c=='C' */
              break;        /* evita executar --b */
    case 'D': --b;          /* no cal el break perquè és l'últim cas */
}
```

Estructura `for`

```
for (inicialització; condició de mentre; increment)
 {
     ...
 }
```

- L'estructura **`for`**, en el llenguatge C, és equivalent a l'estructura **`while`**: consta d'unes inicialitzacions, una condició de fi, i unes sentències d'increment del comptador del bucle que s'executaran després d'executar el cos del bucle un pas. La condició de fi fa referència a una expressió booleana que determina la continuïtat del bucle mentre l'expressió sigui CERT. Per tant, és idèntica a la condició de fi d'un bucle **`while`**. Llavors, per què no es fa servir una sola estructura? La diferència és conceptual: si el bucle té un número d'iteracions conegut quan s'escriu el programa (recórrer una taula de 30 elements, per exemple), llavors es posarà un **`for`**; si el número d'iteracions variarà en funció del context quan s'executi el programa (llegir caràcters d'un fitxer fins que es trobi un Return), llavors es posarà un **`while`**.

Exemples:

ex 1.7: Bucle que va de 1 a **`max`** (encara que `max` pugui ser variable, conceptualment el número d'iteracions està ben delimitat). Cal fixar-se en la condició de mentre del bucle: `i<=max` (**no** és pas `i==max`):

```
s=0;
for(i=1; i<=max; i++)
{
    s += i;
}
```

ex 1.8: El mateix algorisme que abans, però aquesta vegada s'ha introduït la inicialització de la variable **`s`** dins del lloc corresponent a les inicialitzacions del `for` (separats per comes). A més, com que només hi ha una sentència dins del cos del bucle, es poden ometre les claus:

```
for (s=0, i=1; i<=max; i++)
     s += i;
```

ex 1.9: Una vegada més, es pot incloure sentències dins les clàusules del bucle. En aquest exemple s'ha introduït l'única sentència del bucle dins del lloc d'increment (també separat per comes). Aquesta, però, no és una pràctica recomanable. També cal observar que ara el cos del bucle està buit i, per tant, cal posar el punt i coma darrera de la sentència (equival a obrir i tancar claus `{}`):

```
for (s=0, i=1; i<=max; s += i, i++);
```

ContraExemples:

cex 1.3: Si per error es posa un punt i coma darrera de la sentència `for`, el resultat és que el compilador suposa que no hi ha cos del bucle (no dóna cap missatge d'avís!). En l'exemple, la sentència **`s += i`** s'executaria un sol cop, després d'haver realitzat totes les iteracions del bucle (la sentència següent al bucle):

```
for (s=0, i=1; i<=max; i++);
    s += i;
```

cex 1.4: Si per error **no** es posa un punt i coma darrera de la sentència `for`, el resultat és que el compilador suposa que el cos de bucle és la sentència següent al `for`:

```
for (s=0, i=1; i<=max; i++, s += i)
```

cex 1.5: Si la condició de mentre només inclou l'últim cas, el bucle no s'executarà cap vegada, ja que inicialment la condició és falsa:

```
for (s=0, i=1; i==max; i++, s += i);
```

Estructura `while`

```
                                /* inicialitzacions */
while (condició de mentre)
{
    ...                         /* cos del bucle */
                                /* increments */
}
```

- En aquesta estructura, les inicialitzacions i els increments (actualitzacions de les variables que determinen la condició de mentre) no tenen un lloc preassignat, com en cas del `for`. La condició de mentre és l'expressió booleana tal que, si és CERT, s'executa el cos del bucle un altre cop.

Exemples:

ex 1.10: Exemple d'estructura `while` equivalent als exemples de l'estructura `for`:

```
s=0;
I=1;                    /* inicialitzacions */
while (i<=m)
{
    s += I;             /* cos del bucle */
    I++;                /* increment */
}
```

ex 1.11: Exemple simbòlic (funcions `llegir` i `escriure`) on la condició de mentre és complexa:

```
i=0;
c=llegir();
while((c!='F') && (i<1000))
 {
    escriure(c);
    i++;
    c=llegir();
 }
```

ex 1.12: El mateix algorisme que abans, però incloent-hi les accions d'actualització de les variables **`c`** i **`i`** dins de la condició de comparació. En el cas de **`c`**, primer s'executa l'acció **`llegir()`** (perquè està dins de parèntesi) i el resultat (el que s'ha llegit) és emmagatzemat en **`c`** i comparat amb `'F'`. En el cas d'**`i`**, primer es compara **`i`** amb 1000 i després s'incrementa (perquè l'operador d'autoincrement està darrera la variable). Com que ara només hi ha una sentència dins del cos del bucle, es poden ometre les claus. Cal repetir que aquesta tècnica d'incloure sentències dins de les expressions de comparació, si bé dóna codi font més curt (i no sempre un codi executable més curt), no és recomanable per a una programació clara, estructurada, i lliure d'efectes laterals (efectes imprevistos en l'execució):

```
i=0;
while ((c=llegir())!='F') && ((i++)<1000))
    escriure(c);
```

ex 1.13: Per veure l'equivalència entre **`while`** i **`for`**, el següent exemple mostra com es pot fer el mateix exemple de `while`, tot i que el més lògic és fer servir `while` ja que el número d'iteracions del bucle és indeterminat (depèn del moment en que es llegeixi un caràcter `'F'` o, en el seu defecte, quan s'hagin llegit 1000 caràcters):

```
c=llegir()
for(i=0; (c!='F') && (i<1000); i++)
 {
    escriure(c);
    c=llegir();
 }
```

Estructura `do - while`

```
                        /* inicialitzacions */
do {
     ...                /* cos del bucle */
                        /* increments */
 } while (condició de repetir);
```

- En aquesta estructura, el bucle es continuarà executant mentre la condició booleana de repetir sigui CERT. És equivalent al `while`, però amb la diferència que l'avaluació de la condició es realitza després de l'execució del bucle (sempre executarà el bucle almenys una vegada). Observar que sempre hi ha un punt i coma després de `while` (per distingir el **`do-while`** de **`while`**).

Exemples:

ex 1.14: El mateix exemple que el de l'estructura `while`, però aquesta vegada sempre s'escriurà un caràcter com a mínim, és a dir, la `F` última:

```
i=0;
do {
    c=llegir();
    escriure(c);
    i++;
 } while ((c!='F') && (i<1000));
```

ex 1.15: L'equivalent de l'exemple anterior amb l'estructura algorísmica **Repetir-Fins** (la condició de fi és la contraria a la condició de repetir o condició de mentre):

```
i:=0;
repeat
    c:=llegir();
    escriure(c);
    i:=i+1;
until ((c='F') OR (i>=1000));
```

Exercici

- Per practicar les sentències de control de flux, completeu el següent programa `COMPTAMM.C`, que ha de comptar el número de lletres minúscules i el número de lletres majúscules introduïdes per teclat. Per llegir les lletres es disposa de la funció **`getchar()`**, que retorna el codi ASCII del caràcter llegit pel dispositiu estàndard d'entrada (en aquest cas, el teclat). Ara bé, cal prémer el `<Return>` per a que una línia de caràcters comenci a ser introduïda pel dispositiu d'entrada. És a dir, es pot escriure tota una línia i, quan s'hagi premut el `<Return>`, els caràcters de la línia seran llegits un a un cada vegada que s'executi una crida a la funció **`getchar()`**:

```
#include <stdio.h>
void main(void) {
    char c;                 /* declaració de variables */
    short puls,maju,minu;

    /* inicialitzacions */
    /* inici bucle */

    c=getchar();            /* capturar lletra */

         /* cos del bucle */

         /* increments */
    /* fi bucle */

/* imprimir resultats */
    printf("nº minúscules = %d\n",minu);
    printf("nº majúscules = %d\n",maju);
    printf("nº pulsacions = %d\n",puls);
}
```

- Es pot fer servir qualsevol l'estructura repetitiva que es cregui convenient.

1.4. Esquema bàsic d'un programa.

Funció `main`

- Funció principal del programa. És la funció que comença l'execució del programa. Dins d'ella es defineixen el cos principal del programa, les variables locals associades al cos principal, i les crides a les funcions més importants del programa. El nom **`main`** és fix, al contrari de la resta de funcions.

Exemples:

ex 1.16:

```
void main(void) {              /* inici programa */
    char c;                    /* declaració de variables locals principals */
    int i;

    i=0;                       /* instruccions del cos principal */
    do { c=llegir();
         escriure(c);
         i++;
    } while ((c!='F') && (i<1000));
}                              /* fi del programa */
```

Comentaris

- Els comentaris es posen entre uns delimitadors especials de principi i fi de comentari: `/*` i `*/` respectivament. Aquests comentaris poden ser de vàries línies, però cal anar amb compte de tancar-los, perquè sinó poden englobar tot el text que ve al darrera, és a dir, les sentències de C que apareixen abans del següent fi de comentari.

Exemples:

ex 1.17: comentari llarg de varies línies:

```
/* Programa de pràctica de control de flux: s'han de comptar
les lletres */
```

cex 1.6: comentari inacabat → la sentència **`i++`** no es compilarà!:

```
if (c!='F')   /* si no s'ha arribat al final
    i++;      /* incrementa el comptador */
```

Defines

- Els **`define`** són identificadors simbòlics que es substitueixen pel seu valor "textual" (no tenen tipus, només es corresponen a tires de text) en temps de compilació. Conceptualment són *macros*. És usual definir-los al principi del programa. La definició comença per **`#define`**, seguida del nom simbòlic (usualment es posen en majúscules per distingir-los de les variables, que es posen en minúscula), i del valor textual a substituir (el caràcter '#' és l'indicador de directiva de compilació, és a dir, `define` **no** és una sentència de C sinó una directiva per al compilador). Es fa servir per definir valors constants que es faran servir al llarg del programa;

Exemples:

ex 1.18: Es defineixen les constants `FI` i `MAX`, que seran substituïdes en la condició de mentre com si s'hagués escrit `'F'` i `1000` (el compilador ho escriu per nosaltres en tots els llocs que troba els noms simbòlics — després d'haver-los definit — i després ho compila):

```
#define FI     'F'        /* la cadena "FI" és equivalent a la cadena "'F'" */
#define MAX     1000      /* la cadena "MAX" és equivalent a la cadena "1000" */

void main(void)
{
    char c;
    int i;

    i=0;
    while(((c=llegir())!=FI) && ((i++)<=MAX))   /* substitucions... */
         escriure(c);
}
```

Includes

- La directiva **`include`** serveix per "incrustar" fitxers de text en el punt del programa on es troba l'*include*. Això es fa servir per importar fitxers de definicions anomenats *Headers*. Aquest fitxers contenen informació necessària per al compilador a l'hora de referenciar variables, tipus, constants, i funcions externes que proporcionen les llibreries i mòduls C suplementaris al mòdul (conjunt de rutines C) que s'estan implementant (que serà el mòdul principal si inclou la funció `main`). També es fan servir els *Headers* per importar `defines`. Per conveni, els noms dels fitxers *Header* tenen l'extensió "`.H`";

- Un exemple concret és el que s'ha anat fent servir fins ara, que és l'`include` del fitxer "`STDIO.H`". Aquest *Header* aporta definicions necessàries per a l'ús de certes rutines i `defines`, com ara **`printf()`**, **`getchar()`** o **`EOF`**. Dins del directori del Turbo C es troba un subdirectori anomenat `INCLUDE`, que conté l'`STDIO.H` i més fitxers de tipus *Header*, que fan referència al sistema de funcions estàndard de C, que es subministra amb les llibreries estàndard. Aquestes llibreries es troben al directori `LIB`, i el linkador les enllaça automàticament al programa en C quan s'executa la comanda `TCC`. Es pot editar el fitxer `STDIO.H` o qualsevol altre *Header*, on es trobaran les definicions de funcions i `defines` estàndard.

- Les funcions estàndard estan disponibles en tots els entorns on es pugui programar en C. Per tant, es poden transportar els programes que únicament fan servir aquestes llibreries entre diferents plataformes (es transporten els fonts, és a dir, els fitxers `.C`, i es recompilen).

Exemples:

ex 1.19: Importar les definicions pel *define* **EOF** i la funció **printf()**, del *Header* STDIO.H. Si el nom del fitxer *Header* està entre els signes '**<**' i '**>**', el compilador buscarà el fitxer al directori INCLUDE:

```
#include <stdio.h>

void main(void)
{
    int i;

    i=EOF;
    printf("Hello, world number %d!\n",i);
}
```

ex 1.20: Importar un fitxer *Header* que està en algun altre lloc del disc. Cal posar el nom i el camí del fitxer entre cometes:

```
#include "C:\TC\EXEMPLES\MYINC.H"
```

Printf() i Scanf()

- La funció estàndard **printf** serveix per imprimir dades de qualsevol tipus elemental (caràcters, enters, reals, *string*s, ...) en varis formats (decimal, octal, hexadecimal, punt fix, punt flotant, ...). El mètode consisteix en passar com a paràmetres una cadena de format més les variables involucrades. Aquesta cadena de format conté caràcters i codis especials per indicar la posició i format de les variables que s'imprimiran. Per exemple, l'expressió:

```
printf("el caràcter %c apareix %d vegades en el text\n",c,nc);
```

indica que per pantalla ha de sortir la frase esmentada, però substituint (en el lloc on apareixen) els codis %c i %d pel valor de les dues variables **c** i **nc** (les variables s'assignen als codis per ordre d'aparició). En aquest cas, %c indica que c s'ha de visualitzar com a caràcter, i %d indica que nc ho ha de fer com a decimal (una variable o expressió d'un tipus es pot visualitzar en un format d'un altre tipus, per exemple, un enter es pot visualitzar com a caràcter, ...). Alguns dels codis disponibles són:

%c → caràcter	**%d** → decimal	**%f** → real punt fix
%s → *string*	**%o** → octal	**%e** → real punt flotant
	%x → hexadecimal	

- Dins de la cadena de format, com de qualsevol *string* que s'utilitzi en l'entorn del C, apareixen uns codis de caràcters especials, que són necessaris per especificar caràcters que no es poden escriure tal qual perquè interfereixen amb l'edició del programa. Per exemple el <Return> (si es vol introduir un codi <Return> dins d'una cadena, no es pot fer prement la tecla <Return>, ja que passaria a l'edició de la línia següent). També quan hi ha una ambigüitat amb la pròpia definició de la cadena: per exemple, el caràcter cometes (si es vol introduir unes cometes en

la cadena, s'ha de distingir-les de les cometes per tancar la cadena). Aquests codis estan formats per un caràcter '\' seguit d'un identificador del codi a introduir. Alguns d'aquests codis són:

\n	→ nova línia (Return)	****	→ barra invertida
\t	→ tabulador	**\"**	→ cometes
\b	→ retrocés	**\'**	→ apòstrof
\r	→ principi línia	**\%**	→ tant per cent
\0	→ codi nul (ASCII 0)	**\x**hh	→ caràcter ASCII número hh (hexa)

- En el cas de la funció **`scanf()`**, la cadena de format només pot contenir codis de format (%d, %s, ...), i les variables han d'anar precedides del signe '**&**', per passar-les per referència (això s'explicarà més endavant).

Exemples:

ex 1.21: En aquest exemple la variable de tipus caràcter **`lletra`** és preguntada amb **`scanf`** i impresa amb **`printf`** dues vegades: una en forma de caràcter, amb %c i l'altra en forma d'enter, amb %d (codi ASCII del caràcter):

```
#include <stdio.h>

void main(void)
{
    char lletra;

    printf("\nintrodueix una lletra :");
    scanf("%c",&lletra);

    printf("el codi \\ASCII\\ de la lletra \"%c\"\tés %d\n",
              lletra,lletra);
}
```

1.5. Exercicis.

Exercici 1.1: Fer un programa que imprimeixi la taula dels codis ASCII del 32 al 96, mostrant codi ASCII i caràcter, i tabulat en 4 columnes.

Exercici 1.2: Fer un programa que compti les lletres, les paraules i les línies d'una entrada de text, fins que trobi el caràcter EOF.

Exercici 1.3: Fer un programa que pregunti un número i imprimeixi el seu factorial.

2. Programació estructurada.

2.1.- Taules (*array*s) i *string*s.

- Per definir un *array* d'una o vàries dimensions, s'ha d'escriure el tipus i el nom de la taula (com en les variables simples), però després del nom s'han d'escriure uns claudàtors amb un número d'elements per a cada dimensió. Per exemple, es defineix **`taula`** (1D, 9 enters), **`a`** (2D, 3x80 enters), **`m`** (2D, 3x4 reals) i **`m3`** (3D, 6x6x6 reals), més algunes altres variables simples:

```
int      i, j, taula[9], s, a[3][80];
float    m[3][4], m3[6][6][6];
```

- El número d'elements de cada dimensió defineix el rang d'índexs, que sempre és des de 0 fins a n-1, per a cada dimensió. Per exemple, la variable `taula` té 9 elements, que són:

```
taula[0], taula[1], ..., taula[8]
```

- Els *array*s de dues dimensions s'emmagatzemen per files, és a dir, el primer índex fa referència a la fila i el segon a la columna. En el cas general (`n` dimensions), el primer índex és el més significatiu i l'últim índex és el menys significatiu. És a dir, els elements es van emmagatzemant per variació dels índexs de més a la dreta cap als índexs de més a l'esquerra:

```
m[0][0], m[0][1], m[0][2], m[0][3]
m[1][0], m[1][1], m[1][2], m[1][3]
m[2][0], m[2][1], m[2][2], m[2][3]
```

- L'accés als *array*s es fa sobre elements individuals. Cada element de la taula es comporta com una variable del tipus que s'ha definit l'*array*. Per identificar un element, cal posar el nom de l'*array* seguit d'una expressió entre claudàtors (una per cada dimensió) que calcularà l'índex concret. Aquesta expressió pot ser una constant, una variable o un càlcul. Alguns exemples d'accés a **`taula`** i a **`m`**:

```
for(i=0;i<9;i++)  taula[i]=0;
taula[4]++;
taula[i]+= taula[i+1]*16;
taula[i*3+2]= taula[(j++)*2+1];
for(i=0;i<3;i++)
 {
    s=0;
    for(j=0;j<3;j++) s+= m[i][j];
    m[i][3]=s;
 }
```

- Cal anar amb compte quan es posa un càlcul, ja que es podria sortir del rang i accedir a alguna posició de memòria que no correspon a l'*array*. El compilador no pot donar cap error, ja que és un error d'execució. Sí que es pot, però, generar codi de comprovació per a que cada vegada que es faci un accés es verifiqui un possible mal funcionament del programa (opció de compilació que es pot activar en la fase de proves/depuració);

- El què sí que pot ser molt problemàtic és la modificació d'alguna variable dins l'expressió d'índex (efectes laterals). No és recomanable l'ús d'expressions com "`taula[j++]=j;`" si no se sap exactament com s'avaluen (la variable s'incrementa abans o després de carregar el seu valor sobre la taula? sobre quin element es carrega realment?).

*String*s

- Els *string*s són vectors de caràcters. L'única diferència és que, per conveni, un *string* sempre s'acaba amb '`\0`' (ASCII 0). D'aquesta forma es poden definir cadenes de caràcters de longitud variable. L'inconvenient de les cadenes de longitud variable és que sovint desaprofiten alguns bytes de l'espai total reservat per a l'*array*. Per tant, a l'hora de definir una cadena de caràcters, cal tenir en compte la longitud màxima que es preveu per la cadena, més un element addicional per guardar un 0 de fi de *string* (cal que existeixi sempre).

Exemples:

ex 2.1: Al subdirectori `SEMINARI\SEC2` es troba el fitxer `CADENA.C`, que correspon al següent codi. Aquest programa defineix un vector de caràcters i l'omple, un per un, amb la cadena `"hola"`, més el corresponent fi d'string `'\0'`:

```
#include <stdio.h>
void main(void) {
    char sal[5];              /* vector de 5 caràcters */

    sal[0]='h';               /* el primer element del vector és sal[0] */
    sal[1]='o';
    sal[2]='l';
    sal[3]='a';
    sal[4]='\0';              /* també es pot posar sal[4]=0; */
    printf("%s\n",sal);       /* %s indica que s'ha d'imprimir un string */
}
```

cex 2.1: Proveu de compilar i executar el programa `CADENA`. Després proveu d'eliminar la línia que introdueix el caràcter 0 i torneu a compilar i executar. El resultat és imprevisible, ja que la funció **`printf`** rep el punter on comença el vector en memòria, i el va accedint i imprimint de forma seqüencial fins que troba un caràcter 0. Si no s'ha definit, s'imprimiran els bytes que hi hagin emmagatzemats a continuació del vector, fins que algun d'ells sigui 0:

```
#include <stdio.h>
void main(void)
{
     char sal[5];

    sal[0]='h';
    sal[1]='o';
    sal[2]='l';
    sal[3]='a';
    printf("%s\n",sal);
}
```

ex 2.2: Per copiar un *string* a sobre d'un altre, no es pot fer servir l'operador d'assignació '=', ja que el C no pot copiar vectors d'aquesta forma (s'han de copiar element a element). Per treballar amb *string*s, però, es disposa d'unes rutines de llibreria que estan especificades en el *Header* `STRING.H`. Així, es pot fer servir la funció **`strcpy()`**, que copia el segon *string* sobre el primer (suposa que el primer sigui prou gran):

```
#include <stdio.h>
#include <string.h>                  /* defineix la funció strcpy() */

void main(void)
{
    int i;
    char sa[20];                      /* string de 20 caràcters, sentinella inclòs
*/

    strcpy(sa,"hola\tadiós\n");  /* el 2on paràmetre és un string constant */
                        /* els símbols "\t" i "\n" són un únic caràcter */
    for (i=0; i<12; i++)
         printf("\'%c\'\t%d\n",sa[i],sa[i]);
}
```

cex 2.2: Contraexemple de com no es pot assignar un *string*:

```
void main(void)
{
    char sa[20];

    sa="hola\tadiós\n";          /* no es copien els caràcters sobre sa */
    :
    :
```

ex 2.3: Si no es vol (o no es pot) fer servir les rutines estàndard de tractament de *string*s, sempre es pot copiar-los caràcter a caràcter. Per inicialitzar un *string* amb una cadena constant, en canvi, es pot fer servir la possibilitat del C d'inicialitzar una variable quan es defineix el seu tipus. En el cas dels *string*s, fins i tot es pot deixar que el compilador calculi la mida de la cadena, si no es posa el rang:

```
#include <stdio.h>
void main(void)
{
    int I=0;                              /* defineix i inicialitza l'enter i */
    char s1[20];                          /* defineix i dimensiona la cadena s1 */
    char s2[]="hola\tadiós\n";            /* defineix, inicialitza i dimensiona la
                                             cadena s2 (12 caràcters en total) */
    do
    {
         s1[i]=s2[i];          /* copia la cadena s1 sobre s2, caràcter a caràcter */
         i++;
    }while (s1[i-1]!='\0');               /* fins que s'ha copiat el sentinella */

    for (i=0; i<20; I++)                  /* imprimeix els 20 caràcters de s2 */
         printf("\'%c\'\t%d\n",s1[i],s1[i]);
}
```

Exercici

- Per practicar amb els *array*s, es proposa un exercici sobre *string*s, que consisteix en llegir del teclat una cadena de com a màxim 40 caràcters, i imprimir per pantalla línies que comencin per la primera lletra, i a cada línia afegir una lletra més del *string*, fins imprimir tot el *string*. Per llegir la cadena feu servir la funció **`gets()`** (*Header* `STDIO.H`). El final de cadena és '`\0`' (ASCII 0).

Algorisme:

```
importacions (includes)
Principal
    definicions de variables (vector de 40 caràcters, etc.)

    inicialitzar
    gets(cadena);       /* suposar que s'escriuen menys de 40 caràcters */

    per i=1 fins final(cadena[i]) fer
         guardar caràcter i
         escriure un fi d'string sobre el caràcter i
         imprimir l'string
         recuperar caràcter i
    fiper
Fi
```

2.2. Dades estructurades.

- Els tipus estructurats (tuples) en C es defineixen amb la paraula **`struct`**, seguida del nom del tipus i, entre claus, tots els camps de l'estructura, especificant primer el tipus i després el nom del camp (els tipus poden ser estructurats, és a dir, una estructura pot contenir camps que siguin a la seva vegada altres estructures o vectors, strings, matrius, ...). Per exemple:

```
struct llibre
    {
         char nom[50];
          char autor[30];
         int editorial;
         int tipus;
         int pagines;
    };
```

- Per definir variables estructurades dels tipus prèviament definits (usualment es defineixen abans del **`main`**), només s'ha d'indicar **`struct`** `nom_tipus` com a identificador de tipus per a aquella variable (o *array*). Per exemple:

```
void main(void)
{
    int i,j;                    /* tipus elemental */
    struct llibre a;            /* variable estructurada de tipus llibre */
    struct llibre biblio[500];  /* vector de 500 elements de tipus llibre */
```

- Per accedir a un camp d'una estructura, s'indica el nom de la variable seguida d'un punt, més el nom del camp. Cada camp es comporta igual que una variable del tipus del camp (l'estructura és un conjunt de subvariables). Per exemple:

```
a.pagines = 176;
strcpy(a.autor,"Miguel Delibes");
biblio[i].tipus = 1;
biblio[i].nom[j] = ' ';
```

Typedef

- Una funcionalitat del C que resulta útil quan es defineixen tipus estructurats és la possibilitat d'afegir tipus definits pel programador (estructurats o no) als tipus elementals ja existents (`char`, `int`, ...). La sentència **`typedef`** permet fer-ho, indicant primer la composició del nou tipus i després el nom del nou tipus. Aquestes definicions es fan al principi del fitxer, i a la definició d'un tipus es poden referenciar altres tipus que s'hagin creat anteriorment. Per exemple:

```
typedef int enter; /* defineix el nou tipus enter a partir del tipus bàsic int */

typedef struct      /* defineix la següent estructura com el nou tipus llibre */
    {
         char nom[50];
         char autor[30];
         enter editorial;     /* es fa servir el tipus predefinit enter */
         enter pagines;
    }llibre;
```

- Una vegada s'han definit els nous tipus, es poden declarar variables d'aquests tipus tal com es fa amb els tipus bàsic. És a dir, nom del tipus seguit del nom de la variable. En el cas de les estructures, s'estalvia la duplicació de la paraula `struct` cada vegada que es vol referir a una determinada estructura. Per exemple:

```
void main(void)
{
    enter i,j;
    llibre a;
    llibre biblio[500];
```

2.3. Funcions.

- En C, totes les funcions són globals. És a dir, es defineixen totes a fora del `main`, i cap funció pot estar inclosa dins d'una altra funció.

- Per definir una funció, primer cal escriure el tipus del valor que retorna aquella funció, el nom de la funció, i la llista dels paràmetres entre parèntesi, especificant el tipus i el nom de cada paràmetre. Si la funció no retorna res (procediment), llavors cal indicar que retorna un **`void`**. Si no té paràmetres, cal posar **`void`** dins del parèntesi de paràmetres.

- Després de l'especificació del nom i el tipus dels paràmetres i del resultat, s'especifica entre claus les variables locals i les instruccions associades a aquella funció. Les instruccions, a més de les sentències C, poden ser crides a altres funcions (no hi ha límit de número de crides imbricades) o fins i tot a ella mateixa (recursivitat). Al final de la funció cal posar la sentència **`return`** per indicar entre parèntesi el resultat que ha de retornar, excepte en el cas que no retorni res.

ex 2.4: 1ª versió del programa que calcula el Màxim Comú Divisor de dos números. La funció **`Mcd()`** rep 2 enters per paràmetre i retorna un altre enter amb el MCD (aplicant l'algorisme d'Euler):

```
#include <stdio.h>

int Mcd(int m, int n)          /* defineix nom i tipus de funció */
 {
    int x,y,r;                 /* variables locals a Mcd */

    x = (m>n ? m : n);         /* x  pren per valor el màxim de m i n */
    y = (m>n ? n : m);         /* y  pren per valor el mínim de m i n */
    do
     {
         r = x % y;
         x = y;
         y = r;                /* algorisme d'Euler */
     }
    while (y != 0);
    return(x);                 /* retorn de resultat */
 }


void main(void)                /* defineix nom i tipus de la funció principal */
{
    int x,y;                   /* variables locals a main */

    printf("introdueix 2 números:\n");
    scanf("%d",&x);
    scanf("%d",&y);
    printf("m.c.d. = %d\n",Mcd(x,y)); /* crida a la funció Mcd(), i
                                         després a la funció printf() */
}
```

- Alguns informàtics prefereixen escriure els programes començant per les funcions més importants i acabant per les més simples (mètode *Top-Down*). En C, però, abans de fer servir una funció s'ha de declarar (el mateix que amb els tipus, les estructures, les variables, etc.), amb la qual cosa no es pot escriure primer les funcions més importants perquè aquestes criden a les més simples. Per solventar això, es poden declarar només els prototipus de les funcions abans d'escriure el seu codi. Els prototipus són simplement el nom i els tipus dels paràmetres i el resultat, però seguits per un punt i coma.

ex 2.5: Segona versió del MCD. Aquesta vegada especificant el prototipus abans de **`main()`** (és molt important posar el punt i coma darrera els prototipus, però no quan es defineix el cos de la funció):

```
#include <stdio.h>

int Mcd(int m, int n);          /* prototipus de la funció Mcd */

void main(void)
{
    int x,y;

    printf("introdueix 2 números:\n");
    scanf("%d",&x);
    scanf("%d",&y);
    printf("m.c.d. = %d\n",Mcd(x,y));
}

int Mcd(int m, int n)           /* implementació de la funció Mcd */
 {
    int x,y,r;

    x = (m>n ? m : n);
    y = (m>n ? n : m);
    do
     {
         r = x % y;
         x = y;
         y = r;
     }while (y != 0);
    return(x);
 }
```

- A les funcions que només tinguin una línia de codi també s'han de posar les claus. Per exemple:

```
int maxim(int a, int b)
    { return (a>b ? a : b); }

int minim(int a, int b)
    { return (a>b ? b : a); }
```

A la funció `Mcd` es cridarien de la següent forma:

```
int Mcd(int m, int n)
 {
    int x,y,r;

    x = maxim(m,n);
    y = minim(m,n);
    :
    :
```

- En els casos en que el codi de la funció sigui molt petit, es pot recórrer a la definició de macros, a través de la directiva **`define`**. Cal tenir en compte, però, que només es tracta d'una substitució literal en temps de compilació. És a dir, no es realitza cap mena de crida, i el codi es replica cada cop que s'invoca a la macro. Per exemple, es pot definir els símbols **`max`** i **`min`** de la següent forma:

```
#define max(a,b) (((a)>(b)) ? (a):(b))
#define min(a,b) (((a)>(b)) ? (b):(a))
```

els paràmetres **`a`** i **`b`** es substituiran quan es trobi un invocació al `define`, pel símbol que hi hagi en el seu lloc. Això s'indica posant els "pseudoparàmetres" entre parèntesi quan s'escriu el text de substitució: `(a)` i `(b)`. Per invocar a la macro es fa com si fos una funció:

```
int Mcd(int m, int n)
 {
    int x,y,r;

    x = max(m,n);        /* es substitueix per   x = (m>n ? m:n); */
    y = min(m,n);        /* es substitueix per   y = (m>n ? n:m); */
    :
    :
```

- Dins dels *Headers* estàndard es troben els prototipus de les funcions de les llibreries estàndard, o bé un conjunt de macros predefinides. Per exemple, dins de `STDIO.H` es te el prototipus de la funció **`printf()`**, i dins de `STDLIB.H` hi ha les macros de **`max`** i **`min`** (en els llibres de C es troba la descripció de totes les funcions i macros estàndard disponibles):

```
#include <stdio.h>           /* defineix el prototipus printf() */
#include <stdlib.h>          /* defineix les macros max i min */

int Mcd(int m, int n)
 {
    int x,y;
    x = max(m,n);            /* fa servir les macros max i min */
    y = min(m,n);
}

void main(void)
{
    printf("%d\n",Mcd(15,230));  /* fa servir la funció printf() */
}
```

Paràmetres *array*

- Els vectors i les matrius, en C, sempre es passen com a paràmetres per referència (de fet, és recomanable que sigui així, per no copiar molts valors a la pila). Quan es defineix un paràmetre en una funció, només s'indica el tipus dels elements, el nom del paràmetre *array*, i dos claudàtors sense número d'elements, ja que aquest número serà el de l'*array* que es passi per paràmetre quan es crida la funció (només es posa el nom de l'*array* a referenciar).

ex 2.6: El programa `MAXCAR` calcula quin és el caràcter que apareix més vegades dins d'una cadena (un *array*). Cal observar com la funció `inicialitzar()` accedeix als components del vector que es passi com a paràmetre, perquè es passa per referència:

```
#include <stdio.h>

void inicialitzar(int t[])                  /* paràmetre array t */
 {
    int i;              /* se suposa que el vector té, com a mínim, 27 elements */

    for(i=0; i<=26; i++) t[i]=0;            /* accés per referència */
 }

char maxcar(char s[], int tc[])             /* paràmetres array s i tc */
 {
    char c;
    int i, max, imax;

    for(i=0; s[i]!=0; I++)                  /* recorre el string associat a s */
     {
         c = s[i];
         if ((c >= 'a') && (c <= 'z'))
              tc[c-'a']++;
     }                          /* apunta quantes vegades apareix cada lletra */
    max = 0;
    imax = -1;
    for(i=0; i<=26; i++)
         if (tc[i] > max)
          {
              max = tc[i];
              imax = i;
          }                 /* detecta el número màxim de lletres aparegudes */
    return(imax+'a');
 }

void main(void)
{
    int ticar[27];                      /* un número per cada lletra */
    char c, frase[80];                   /* string de caràcters a llegir del teclat
*/

    inicialitzar(ticar);                /* inicialitza el vector ticar a 0's */
    printf("introdueix una frase:\n");
    gets(frase);                        /* llegeix frase (referència) */
    c = maxcar(frase,ticar);            /* retorna lletra de moda */
    printf("lletra que apareix més vegades = %c\n",c);
}
```

Paràmetres estructurats

- Els paràmetres estructurats (tuples) es poden passar per valor, és a dir, es copiarà tot el contingut d'una variable estructurada sobre el paràmetre associat (incloent-hi tots els elements dels vectors que contingui). Això pot sobrecarregar l'ús de la pila. Per tant, si les estructures són massa complexes cal passar paràmetres per referència (s'explicarà a la secció 3).

ex 2.7: En el programa PUNTS2D es defineix un tipus nou que consta de 2 reals simples (aquest tipus es pot fer servir per definir paràmetres i resultats de funcions). Es tracta d'introduir 2 punts 2D i calcular el punt mig, també en 2D:

```
#include <stdio.h>

typedef struct
 {  float x;
    float y;} punt2D;                  /* definició de tipus */

punt2D preguntar_punt(void)            /* retorna una variable de tipus 2D */
 {
    punt2D p;

    printf("\tcoordenada x : ");
    scanf("%f",&p.x);
    printf("\tcoordenada y : ");
    scanf("%f",&p.y);
    return(p);                         /* retorna dos floats */
 }

punt2D punt_mig(punt2D p1, punt2D p2)
 {                                     /* rep 2 punts2D i retorna el punt mig */
    punt2D r;

    r.x = (p1.x + p2.x) / 2;
    r.y = (p1.y + p2.y) / 2;
    return(r);
 }

void main(void)
{
    punt2D a,b,c;

    printf("introdueix punt 1\n");
    a = preguntar_punt();              /* copia el resultat sobre a */
    printf("introdueix punt 2\n");
    b = preguntar_punt();              /* copia el resultat sobre b */

    c = punt_mig(a,b);   /* copia a i b sobre p1 i p2, i el resultat sobre c */

    printf("\npunt mig = (%.2f,%.2f)\n",c.x,c.y);
}
```

Exercici

Modificar el fitxer `PUNTS2D.C` per obtenir el fitxer `CERCLE2D.C` corresponent a un programa en C que, a partir de 2 punts 2D, imprimeixi per pantalla les coordenades del centre i el radi del cercle que té com a diàmetre la recta que uneix els dos punts. És a dir, que imprimeixi el punt mig i la distància d'aquest punt mig a qualsevol dels 2 punts d'entrada.

Per fer la funció **`distancia()`** es necessitarà la funció arrel quadrada, que està definida a `MATH.H`. La funció té el següent prototipus: `double sqrt(double x)`. la funció **`distancia()`** aplicarà el teorema de Pitàgores, segons el qual la distància entre 2 punts es pot calcular com a l'arrel quadrada de la suma de les diferències de coordenades al quadrat. Aquesta funció tindrà un prototipus com aquest:

```
float distancia(punt2D p1, punt2D p2);
```

Paràmetres de `main`

- La funció `main` també pot rebre paràmetres i retornar resultat. Això s'entén si pensem que el Sistema Operatiu pot reconèixer les paraules (opcions) que s'escriuen darrera de la comanda a executar, i passar aquesta llista de paraules a l'aplicació que invoca la comanda. Per exemple;

```
C:\>DIR TC /w
```

invoca la comanda `DIR` i li passa com a paràmetre les paraules "`DIR`", "`TC`" i "`/w`", per a que el `DIR` operi en conseqüència (el nom de la comanda o programa també el considera com un paràmetre). El retorn de resultat es refereix a un enter que accepta el S.O. com a codi d'error de l'aplicació (0 : execució correcta; no 0 : número d'error).

- Els dos tipus de paràmetre que rep el `main` són el número de paraules (enter) i una taula de *string*s. El què retorna és només enter. Els noms dels paràmetres poden ser qualsevol, encara que en molts llibres apareixen com **`argc`** i **`argv`**.

ex 2.8: A `PARAM.C`, el programa imprimeix el número i la llista de paràmetres. Per verificar l'efecte del pas de paràmetres cap a `main` compilar el `PARAM.C`, i executar-lo escrivint algunes paraules darrera el nom del programa. Per verificar l'efecte del retorn de resultat, es pot estudiar i executar el fitxer `NOPAR.BAT`, que crida a `PARAM` una vegada amb paràmetres i l'altra sense, i canvia l'execució en funció del codi d'error retornat.

```
#include <stdio.h>
int main(int n_param,char *t_param[])
{
    int i;

    printf("\nHas introduït %d paràmetres:\n\n",n_param);
    for(i=0; i<n_param; i++) printf("\t%s\n",t_param[i]);

    return(n_param==1);
}                          /* retorna CERT si només s'ha passat el nom */
```

2.4.- Àmbit de les variables.

- Totes les variables que es defineixen dins d'una funció (incloent-hi la funció `main`) són locals a aquella funció. Totes les variables que es defineixen fora de tota funció són globals a totes les funcions. És a dir, es poden accedir des de qualsevol lloc del programa. No cal dir, però, que les variables locals són preferibles a les globals (programació estructurada).

ex 2.9: El fitxer `ORDENA_L.C` conté un programa que inicialitza un vector amb valors aleatoris, i després l'ordena i l'imprimeix, tot amb variables locals:

```
#include <stdio.h>
#include <stdlib.h>           /* funcions randomize() i random() */

#define N_VECT 10                     /* número d'elements del vector */

void inirand_t(int n, int t[])     /* inicialitza el vector (per referència) */
 {  int i;

    for(i=0; i<n; i++) t[i] = random(80);
 }

void ordena_t(int n, int t[])      /* ordena el vector (per referència) */
 {                    /* n serveix per saber el número d'elements del vector (?) */
    int i,j,temp;

    for(i=0; i<(n-1); I++)         /* mètode de la bombolla */
         for(j=i+1; j<n; j++)
              if (t[j]<t[i]) {
                              temp = t[j];
                              t[j] = t[i];
                              t[j] = temp;
                             }
 }

void impri_t(int n, int t[])         /* imprimeix el vector */
 {  int i;

    for(i=0; i<n; i++) printf("%3d:\t%d\n",i,t[i]);
 }

void main(void)
{
    int vector[N_VECT];           /* vector LOCAL a main */

    randomize();
    inirand_t(N_VECT,vector);
    printf("Taula inicial :\n");
    impri_t(N_VECT,vector);
    ordena_t(N_VECT,vector);
    printf("\nTaula ordenada :\n");
    impri_t(N_VECT,vector);
}
```

ex 2.10: El fitxer `ORDENA_G.C` conté un programa que fa el mateix que `ORDENA_L.C`, però fent ús de les variables globals. En concret defineix el vector com a global, i totes les funcions hi accedeixen directament:

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

#define N_VECT 10

int vector[N_VECT];                  /* definició de variable GLOBAL */

void inirand_t(void)
 {
    int i;

    for(i=0; i<N_VECT; i++)
         vector[i] = random(80);     /* accés a variable global */
 }

void ordena_t(void)
 {
    int i,j,temp;

    for(i=0; i<(N_VECT-1); i++)
         for(j=i+1; j<N_VECT; j++)   /* accés a variable global */
              if (vector[j]<vector[i]) {
                                      temp = vector[j];
                                      vector[j] = vector[i];
                                      vector[i] = temp;
                                          }
 }

void impri_t(void)
 {
    int I;                            /* accés a variable global */

    for(i=0; i<N_VECT; i++) printf("%3d:\t%d\n",i,vector[i]);
 }

void main(void)
{
    randomize();
    inirand_t();                      /* no hi han paràmetres */
    printf("Taula inicial :\n");
    impri_t();
    ordena_t();
    printf("\nTaula ordenada :\n");
    impri_t();
}
```

- Tot i que sembla que aquesta versió és més senzilla (no hi han paràmetres), resulta ser molt més difícil de modificar. Per exemple, si algun dia es volen afegir més vectors que puguin ser operats per les funcions **`impri_t`**, **`ordena_t`**, etc.

2.5. Exercicis.

Exercici 2.1: Modificar `ORDENA_L.C` per a definir 2 vectors, un el doble de llarg que l'altre, i aplicar el següent algorisme:

```
omplir v i w amb números aleatoris
ordenar v i w
copiar un 100 sobre els elements de w que existeixin en v
ordenar w
imprimir w
```

Exercici 2.2: observar que és molt més costós modificar `ORDENA_G.C` per fer el mateix que abans.

Exercici 2.3: fer un programa per gestionar una agenda de telèfons (màxim 200 telèfons). Definir el tipus "telèfon" i estructurar el programa amb les funcions que es creguin convenients:

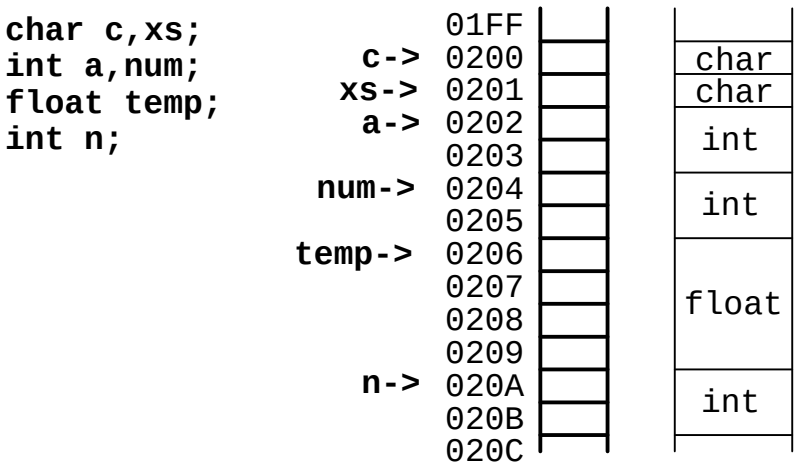
```
tipus telèfon
    nom,
    adreça,
    prefix,
    número,
    etc.

funcions principals
    inserir,
    esborrar,
    consultar,
    modificar,
    etc.
```

3. Punters.

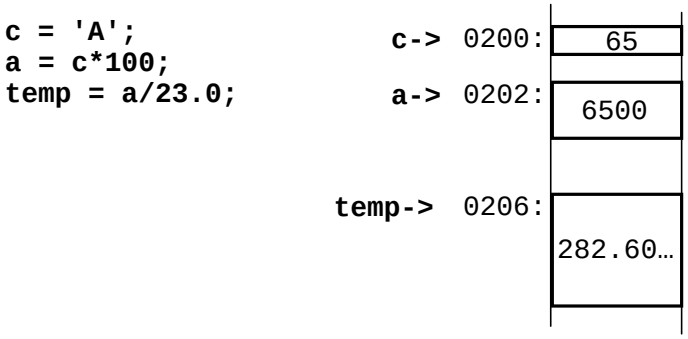
3.1. Concepte i ús dels punters.

Adreça de les variables

- Quan definim una variable d'un cert tipus, es reserva espai a memòria per contenir el valor de la variable, en funció del tipus. El nom de la variable queda associat a l'adreça de memòria on comença aquest espai reservat:



- Quan actualitzem una variable, canvia el valor de les posicions de memòria assignades a aquella variable. Si no inicialitzem una variable abans de llegir el seu contingut, el seu valor serà indeterminat.



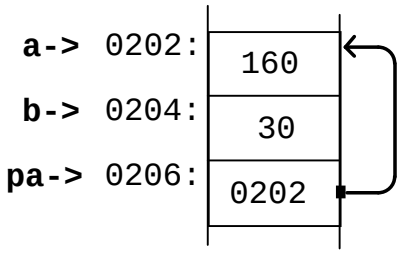
- Per fer referència al valor d'una variable escribim el seu nom. Per fer referència a l'adreça d'una variable disposem de l'operador d'adreça '**&**' col.locat davant del nom de la variable. Per exemple, si vull escriure l'adreça i el contingut de la variable **c**, ho puc fer amb **`printf()`** de la següent forma (el codi de `printf` per escriure punters és '**%p**'):

```
printf("Variable c\tadreça: %p\tvalor: %c\n", &c, c);
```

Variables de tipus punter

- Les variables de tipus punter són aquelles que el seu valor és l'adreça d'una altra variable. Per tant, una variable de tipus punter té una adreça i unes posicions de memòria assignades, però el seu valor "apunta" a un altre lloc de la memòria.

```
int a,b;
int *pa;

a = 160;
b = 30;
pa = &a;
```



- Per definir una variable punter cal especificar el tipus de les variables on apuntarà, més el nom de la variable punter precedit per un asterisc '*****'. Per exemple, es pot definir un punter a variables de tipus enter de la següent forma: "**`int *pa;`**". Un punter a **int** no pot apuntar a un **char**, per exemple. És a dir, el compilador fa verificació de tipus sobre els apuntadors (cosa que no passa amb el llenguatge assemblador).

ex 3.1: `PUNTER.C` és un programa que mostra com declarar una variable punter, com assignar-li una adreça d'una altra variable, i com escriure les adreces i el contingut de variables "normals" i de variables punter (el valor d'una variable punter és una adreça):

```
#include <stdio.h>

void main(void)
{
 int a, b;
 int *pa;

 a = 160;
 b = 30;
 pa = &a;            /* pa pren per valor l'adreça d' a */

 printf("Variable a\tadreça: %p\tvalor: %d\n",&a,a);
 printf("Variable b\tadreça: %ptvalor: %d\n",&b,b);
 printf("Variable pa\tadreça: %ptvalor: %p\n",&pa,pa);
}
```

Accés indirecte

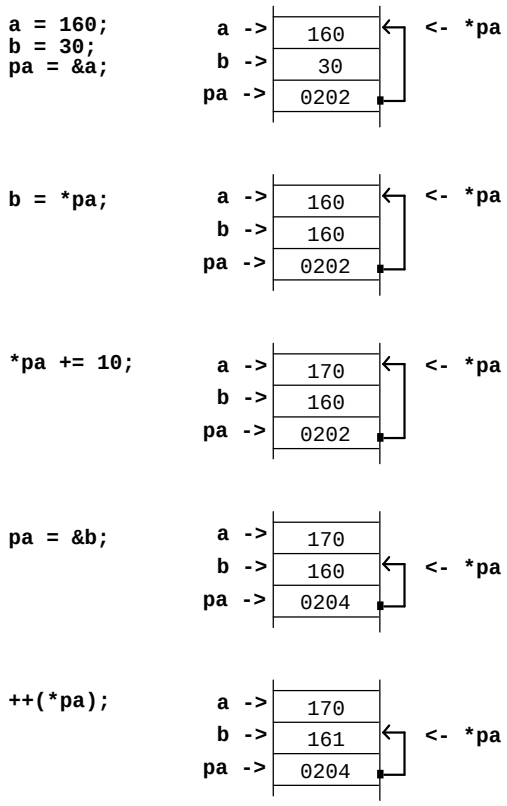
- A partir del moment en què una variable punter està apuntant a algun lloc concret de la memòria, es pot accedir a aquella posició a través del punter com si accedíssim a una variable (del tipus que s'apunta). Es tracta de l'accés indirecte. S'hi accedeix escrivint el nom de la variable punter precedit de l'asterisc. Conceptualment, si definim la variable `int *pa;` tindrem la següent informació:

`&pa` : adreça variable **`pa`**
`pa` : contingut variable **`pa`** = punter a variable `int`
`*pa` : contingut de la variable `int` apuntada per **`pa`** = valor enter

D'alguna forma, si he definit "`int *pa;` " podem considerar que "`*pa` " és una variable de tipus **`int`** i, per tant, podem operar de la mateixa manera (llegir el seu valor, actualitzar-la, incrementar-la, etc.). Cal tenir present, però, que la variable `pa` sempre ha d'estar apuntant a alguna variable entera, i que modificant `*pa` estarem modificant el contingut de la variable entera referenciada.

ex 3.2: `PUNTER2.C` mostra diverses operacions fent servir l'accés a través d'un punter (accés indirecte). Fixar-s'hi que el punter pot referènciar a qualsevol variable (prèvia inicialització):

```
#include <stdio.h>

void main(void)
{
 int a,b;
 int *pa;

 a  =    160;            /* inicialitzem a */
 b  =    30;             /* inicialitzem b */
 pa =    &a;             /* pa apuntarà a a */
 b  =    *pa;            /* b pren per valor el contingut d' a, a través de pa */
 *pa +=  10;             /* sumo 10 a a, a través de pa */
 pa =    &b;             /* pa apuntarà a b */
 ++(*pa);                /* autoincremento b, a través de pa */

 printf("Valor   a: %d\n",a);
 printf("Valor   b: %d\n",b);
 printf("Valor *pa: %d\n",*pa);   /* imprimirà el valor de b!!! */
}
```



Inicialització de punters

- Si una variable no s'inicialitza, el seu valor és indeterminat. Si un punter no s'inicialitza, apunta a un lloc indeterminat de memòria. Cal inicialitzar **sempre** els punters abans de fer-los servir. Per treballar amb punters buits (que temporalment no apuntin enlloc), es poden inicialitzar a un valor especial anomenat `NULL` (punter nul). `NULL` és una constant simbòlica que està definida a la majoria de *Headers* estàndard (`STDLIB.H`, `STDIO.H`, `STRING.H`, etc.)

cex 3.1: Com a contraexemple provar de posar la inicialització de `pa` sobre `a` entre comentaris. L'accés on apunta una variable punter no inicialitzada és totalment imprevissible (es pot "penjar" l'ordinador):

```
#include <stdio.h>

void main(void)
{
 int a,b;
 int *pa;

 a  = 160;
 b  = 30;
                          /* pa = &a; */
 b  = *pa;
 *pa += 10;
 pa = &b;
 ++(*pa);

 printf("Valor   a: %d\n",a);
 printf("Valor   b: %d\n",b);
 printf("Valor *pa: %d\n",*pa);
}
```

- Si un punter ha d'estar assignat sempre a una única variable, el podem inicialitzar quan el declarem, tot i que sempre el podrem canviar posteriorment:

```
void main(void)
{
 int a = 160, b = 30;
 int *pa = &a;

 b    =  *pa;
 *pa +=  10;
 pa   =  &b;
 ++(*pa);

 etc.
```

Exercici

- Editeu el fitxer `EXERCI_1.C` i substituïu els interrogants per les instruccions que implementin el que es demana en els comentaris adjunts. El resultat ha de ser:

```
x = 16   y = 4   z = 7
```

3.2. Punters a taules.

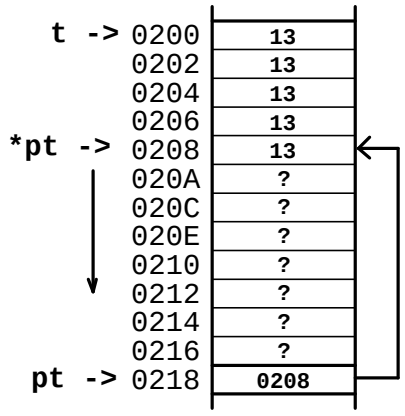
- Per obtenir l'adreça d'un element d'una taula, simplement s'escriu l'operador adreça '&' davant de l'element de la taula referenciat. Per exemple:

```
int t[10];
int *pt;

pt = &t[4];          /* pt conté l'adreça de l'element t[4] */
*pt = 13;            /* posem un 13 dins de t[4] , a través de pt */
```

- Podem fer servir els punters per recòrrer tota una taula d'elements, posicionant el punter al principi de la taula (sobre **`t[0]`**), i incrementant el contingut de la variable punter per "saltar" al següent element de la taula (els elements d'una taula s'emmagatzemen en posicions consecutives de memòria). No s'ha de tenir en compte quantes posicions de memòria ocupa cada element, ja que l'increment d'un punter s'indica amb número d'elements, i el compilador multiplica aquest número per les posicions de memòria que ocupen els elements, en funció del tipus (en el PC, un punter a **`int`** es multiplica per 2, un punter a **`float`** per 4 i un punter a **`char`** per 1).

ex 3.3: Inicialitzar tots els elements d'un vector de 10 elements a través d'un punter (del mateix tipus que els elements de la taula):

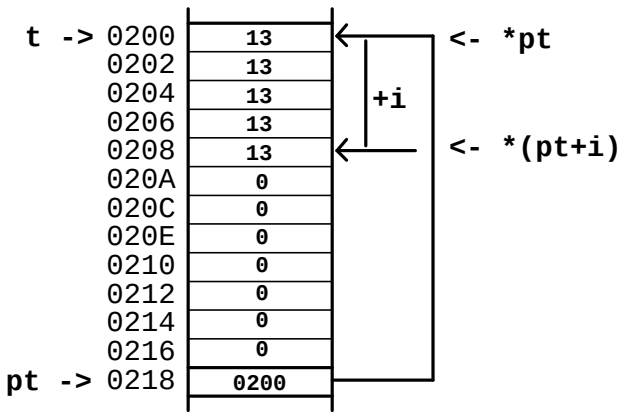
```
#include <stdio.h>

void main(void)
{
 int t[10];
 int *pt, i;

 pt = &t[0];                 /* posicionar pt sobre el primer element del vector */

 for (i=0; i<10; I++)        /* repetir 10 vegades ... */
  {
    *pt = 13;                /* col.loca un 13 sobre l'element que està apuntant */
    pt++;                    /* pt apuntarà al següent enter (sumar 2 a pt) */
  }
 for(i=0; i<10; i++) printf("%d\n",t[i]);   /* escriu el vector */
}
```



- Una altra forma d'accedir a una taula a través de punters és fer servir el punter com a base i indicar un desplaçament fins a l'element que es vol referenciar. Una altra vegada el desplaçament s'expressarà en número d'elements, i el compilador s'encarrega de convertir-lo a desplaçament en posicions de memòria (en funció del tipus dels elements i del *Hardware* que es fa servir). Per expressar l'adreça base d'una taula podem fer-ho donant l'adreça del seu primer element o, simplement, amb el nom de la taula sense cap índex.

ex 3.4: Aquest és el mateix algorisme que l'exemple anterior, però fent servir el mode d'accés base més desplaçament (mode d'adreçament):

```
#include <stdio.h>

void main(void)
{
 int t[10];
 int *pt, i;

 pt = t;                    /* pt pren per valor l'adreça base del vector */

 for(i=0; i<10; i++)
                            /* accés punter base més desplaçament */
    *(pt + i) = 13;         /* (cal posar el punter i el desplaçament entre parèntesi,
                               abans de posar l'asterisc) */

 for(i=0; i<10; i++) printf("%d\n",t[i]);
}
```



- Un aspecte curiós del llenguatge C és que permet intercanviar els mètodes de referència d'elements dels punters i de les taules. És a dir, podem posar el desplaçament respecte a un punter entre claudàtors, i podem posar l'índex d'un vector sumat a l'adreça del vector i tot junt precedit d' un asterisc. Cal tenir present, però, que el punter és variable (`pt++;` està permés) i el nom de la taula no (`t++` **no** està permés). Exemples:

```
pt[i] = 13;                 <=>     *(pt + i) = 13

printf("%d\n", *(t + I));   <=>     printf("%d\n", t[i]);
```

Exercici

- Editeu el fitxer `EXERCI_2.C` i substituïu els interrogants per les instruccions que implementin el que es demana en els comentaris adjunts. El resultat ha de ser:

```
ts = "Es kui no m'e$coltes mai!"
```

Punters a matrius

- La utilització de punters com a base de matrius segueix el mateix principi que sobre vectors, però cal tenir en compte que el desplaçament de l'índex de files ha d'anar multiplicat pel número d'elements que hi ha a cada columna (no obstant això, ara tampoc cal multiplicar el desplaçament total per la mida dels elements).

- Implementarem, per exemple, l'algorisme que posa a zero tots els elements d'una matriu de 9 x 6 enters llargs (4 bytes per element):

```
#define N_Fil      9
#define N_Col      6

long ml[N_Fil][N_Col];
int i,j;

for(i=0; i<N_Fil; i++)
    for(j=0; j<N_Col; j++)
              ml[i][j] = 0;
```

- Podem substituir l'accés als elements `ml[i][j]` mitjançant l'ús del punter `pl`, i el càlcul corresponent al desplaçament de l'element en funció dels índexos:

ex 3.4: Accés a matrius mitjançant punters i desplaçaments:

```
#define N_Fil      9
#define N_Col      6

void main(void)
{
    long ml[N_Fil][N_Col];
    int i,j;
    long *pl = ml;

    for(i=0; i<N_Fil; i++)
         for(j=0; j<N_Col; j++)
             *(pl+ i*N_Col+ j) = 0;
}
```

També es pot escollir una representació de tipus matriu per als punters, és a dir:

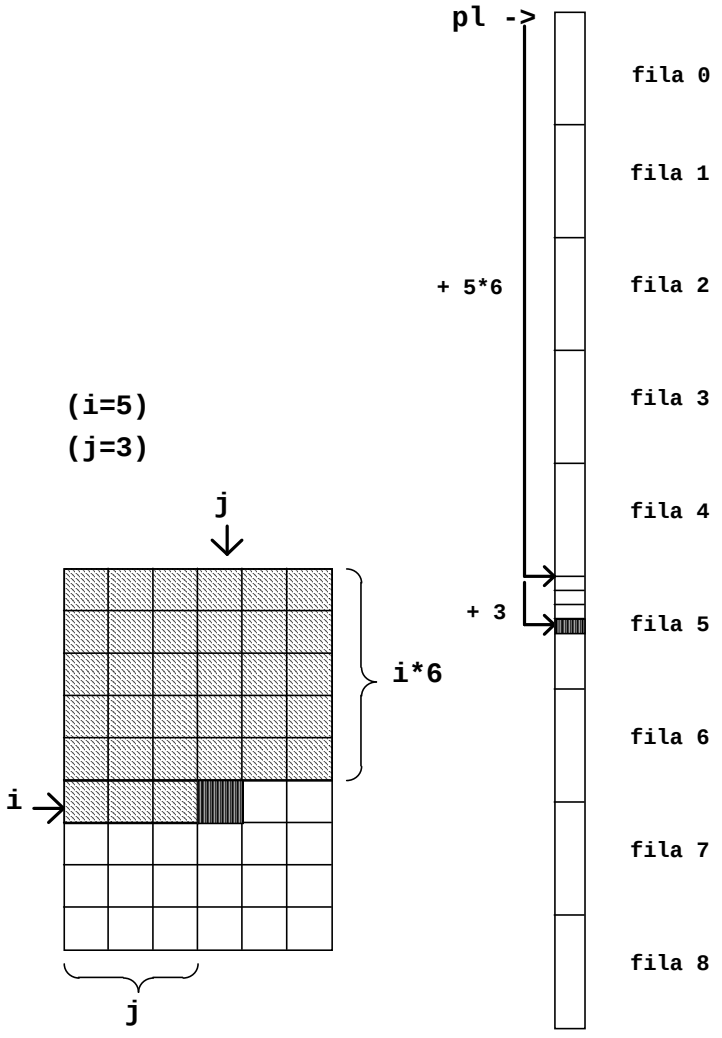
```
*(pl+ i*N_Col+ j) = 0;    <=>    pl[i][j] = 0;
```

- La pregunta que es planteja és per què fer servir punters si obtenim el mateix resultat que amb les taules. La resposta és que es permet un mecanisme molt lligat a les possibilitats dels Llenguatges Màquina. És a dir, els accesos amb punters corresponen als modes d'adreçament, i per tant, es pot fer una programació més òptima si es pensa en aquests termes. Per exemple, per inicialitzar una matriu tota a zeros, els elements de la qual sabem que estan emmagatzemats consecutivament, podriem implementar la inicialització com un sol bucle, de la següent forma:

```
for(i=0; i < N_Fil*N_Col; i++) *(pl + i) = 0;
```

Anant una mica més enllà, podem implementar l'accés amb mode d'adreçament registre indirecte autoincrementat, en comptes de registre base més desplaçament:

```
for(i=0; i < N_Fil*N_Col; i++, pl++) *pl = 0;
```



- Pel cas de matrius de 3 o més dimensions es pot aplicar el mateix principi, tenint en compte que els índexos es multiplicaran pel rang de l'índex immediatament inferior. Exemple:

```
*( p3D + ((i * N_J) + j) * N_K + k)
```

3.3. Punters a *strings*.

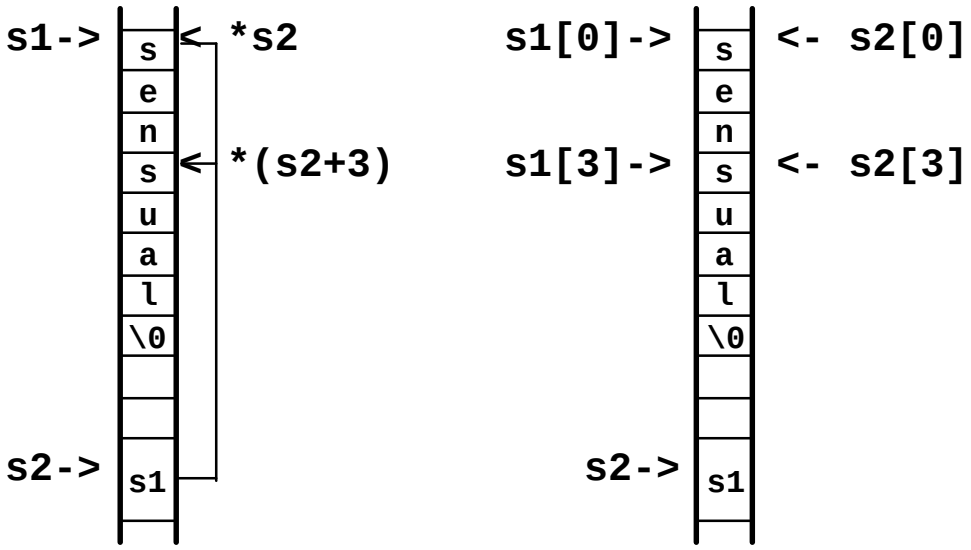
- Tot el que s'ha dit sobre els punters a taules és igualment aplicable als *strings*. Tot i això caldrà fer algunes puntualitzacions referents a la definició i còpia de *strings*. En primer lloc, la definició d'un punter a `char` **no** és un *string*. Com a molt és un punter a un vector de caràcters. És a dir, només per escriure: "`char *s2;`" **no** n'hi ha prou per definir un *string*, ja que així només definim l'espai de memòria per a un punter a caràcter.

- En segon lloc, no podem copiar *strings* només amb el signe '=' perquè són vectors; sí que es permet copiar l'adreça d'un vector de caràcters sobre una variable de tipus punter a `char`, però **només es copia l'adreça!**.

- En tercer i últim lloc, es pot accedir a un caràcter d'un *string* a través d'un punter a `char` seguit de l'índex del caràcter entre claudàtors, ja que correspon al mode d'adreçament punter base més desplaçament.

ex 3.5: `PUTSTRI1.C` mostra com es defineix un *string* de 10 caràcters, més un punter a `char` que apuntarà a aquest *string* (vector):

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

void main(void)
{
    char s1[10];                    /* vector de 10 caràcters */
    char *s2;                       /* punter a string */

    strcpy(s1,"sensual");           /* inicialitzo vector */
    s2 = s1;                        /* inicialitzo punter */
    s2[3] = 'x';                    /* accés al vector a través del punter */
    printf("%s\t%s\n",s1,s2);       /* imprimeix strings (vector i punter) */
}
```



cex 3.2: El següent contraexemple mostra que no es pot accedir a un punter a `char` com si fos una cadena de caràcters, ja que no té espai reservat per als caràcters de la cadena (l'execució d'aquest programa pot ser catastròfica perquè accedeix a memòria a través d'un punter no inicialitzat!):

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

void main(void)
{
    char *s1;

    strcpy(s1,"sensual");
    printf("string 1: %s\n",s1);
}
```

- De vegades es pot veure en alguns programes de C com es defineixen frases constants que s'imprimiran en algun moment, i que són assignades a punters a caràcter mitjançant l'autoinicialització. Aquestes cadenes es guarden en algun lloc de memòria cada cop que es carrega el programa, i el punter queda inicialitzat al principi de la cadena. No obstant això, el punter es pot canviar per a que apunti a alguna altra cadena.

ex 3.6: `PUTSTRI2.C` representa com assignar una cadena constant a un punter a `char`:

```
#include <stdio.h>

void main(void)
{
    char s1[10] = "sensual";
    char *s2 = "senxual";

    printf("string 2: %s\n",s2);
    s2 = s1;
    printf("string 2: %s\n",s2);
}
```

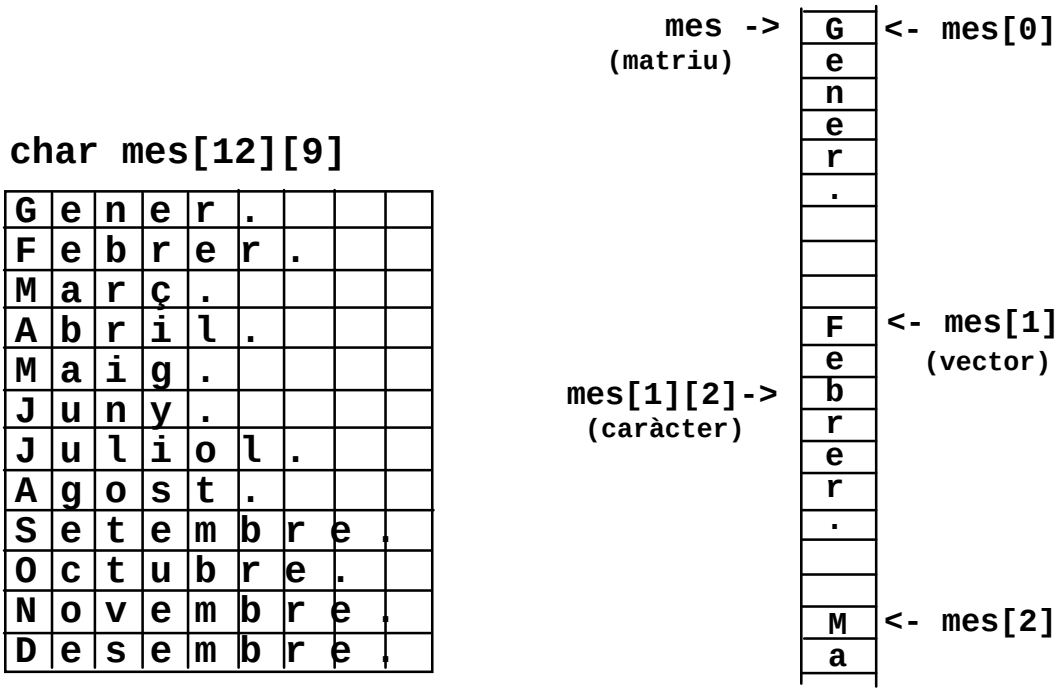
- Moltes vegades necessitem llistes de missatges. Una forma d'emmagatzemar aquestes llistes és fer un vector de vectors de caràcter, és a dir, una matriu de caràcters. Per exemple, la llista dels noms dels mesos es podria introduir com un vector de 12 vectors de 9 `char`s cadascun (la longitud màxima que té "desembre", per exemple, i el caràcter fi de *string*):

```
char mes[12][9];

void inicialitzar_taules_mes(void)
{
    strcpy(mes[0],"Gener");
    strcpy(mes[1],"Febrer");
    strcpy(mes[2],"Març");
    strcpy(mes[3],"Abril");
:
:
}
```

ex 3.7: `STRIMES1.C` mostra com gràcies a l'autoinicialització en declarar les variables, podem introduir els noms dels mesos d'una forma molt més simple:

```
#include <stdio.h>

char mesos[12][9] = {"Gener","Febrer","Març","Abril","Maig",
                     "Juny","Juliol","Agost","Setembre",
                     "Octubre","Novembre","Desembre" };

int diesmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31}

void main(void)
{
    for(i=0; i<12; i++)
         printf("%s\t%d\n",mes[i],diesmes[i]);
}
```

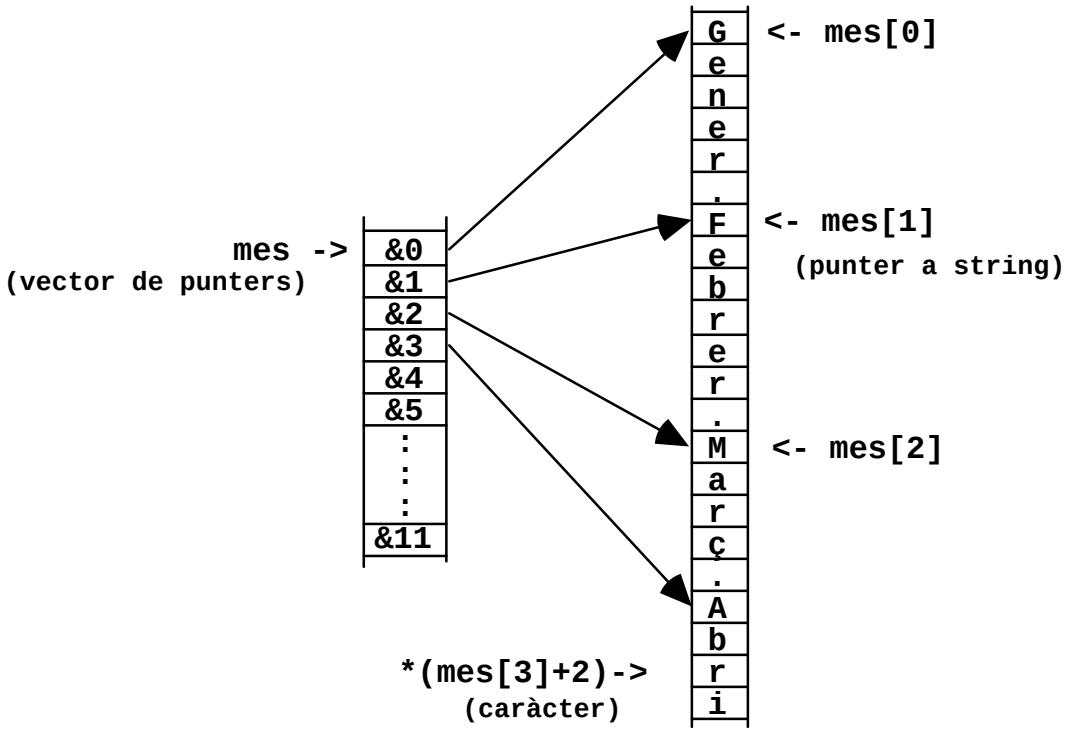
- El problema que sorgeix amb aquest mètode és que els *strings* són variables, mentre que el vector de vectors definirà una longitud fixa i màxima per a tots els *strings*. Per tant, si els *strings* són més curts estarem perdent bytes de memòria, mentre que si són més llargs no els podrem emmagatzemar:



- La solució passa per definir la llista de *strings* com un vector de punters a `char`, en el supòsit que tinguéssim totes les cadenes emmagatzemades a memòria de forma consecutiva (sense perdre cap byte), i els punters assignats al principi de cada cadena (la cadena s'acaba amb el caràcter sentinella).

ex 3.8: `STRIMES2.C` insereix a memòria les cadenes corresponents als noms dels mesos i inicialitza els punters del vector `mesos` quan el declara:

```
#include <stdio.h>

char *mesos[12] = {"Gener","Febrer","Març","Abril","Maig",
                   "Juny","Juliol","Agost","Setembre",
                   "Octubre","Novembre","Desembre" };

int diesmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31}

void main(void)
{
    for(i=0; i<12; i++)
         printf("%s\t%d\n",mes[i],diesmes[i]);
}
```



Exercici

- Editeu el fitxer `EXERCI_3.C` i substituïu els interrogants per les instruccions que implementin el que es demana en els comentaris adjunts. El resultat ha de ser:

```
a = 4.80
tf = ( 0.0, 0.0, 9.6, 0.0, 6.1, 0.0)
m :
    0.0   0.0   0.0   0.0
    0.0   0.0   12.6  0.0
    0.0   0.0   0.0   2.9
```

3.4. Punters a tuples.

- Quan definim punters a variables que són tuples, l'accés als camps de la tupla a través de l'apuntador es fa amb el nom de la variable apuntador seguit de '`->`' més el nom del camp. Això permet al C distingir entre l'accés a un camp d'una variable estructura (per la suma del desplaçament del camp a l'adreça de la variable) i l'accés a un camp d'una estructura referenciada per una variable apuntador (sumant el desplaçament del camp a l'adreça que conté la variable).

ex 3.9: Exemple on es veu l'accés als camps d'estructures referenciades per un apuntador del tipus corresponent:

```
typedef struct
 {
    char nom[50];
    char autor[30];
    int tipus;
    int pagines;
 }llibre;                              /*definició de l'estructura  llibre */

void main(void)
{
 int i,j;
 llibre a,*pa, *pb, biblio[500];       /* declació de variables, punters */
                                       /* i vectors a estructures llibre */
 pa = &a;
 pa->pagines = 176;
 strcpy(pa->nom,"El Perfum");          /* accés als camps d' a  */
 pa->autor[0] = 'P';                   /* a través del punter pa */

 pb = biblio;                  /* pb = adreça vector biblio */
 pb[1]->tipus = 1;             /* pb[1] és punter a l'element 1 de biblio */
 (pb+2)->pagines = 120;        /* (pb+2) és punter a l'element 2 de biblio */
 for(i=0; i<500; i++, pb++)/* recòrrer el vector biblio amb el vector pb */
  {
    pb->tipus = 1;
    if (pb->pagines == 0)
     {
         for(j=0; j<49; j++)
             pb->nom[j] = ' ';
         pb->nom[49] = '\0';
     }
 }
}
```

Exercici

- Editeu el fitxer `EXERCI_4.C` i substituïu els interrogants per les instruccions que implementin el que es demana en els comentaris adjunts. El programa no dóna cap resultat, però cal entendre molt bé com funciona el tema dels punters per saber-lo fer.

3.5. Punters a paràmetres.

- En aquest apartat es tracta el concepte de pas de paràmetres per referència, doncs es tracta de rebre l'adreça d'una variable i, per tant, un paràmetre per referència és llavors un punter. Es tracta de definir els paràmetres d'una funció com a punters a variables del tipus associat al paràmetre (tipus més asterísc, seguit del nom del paràmetre). Quan es cridi a la funció, però, caldrà anar amb compte de passar l'adreça de les variables i no el valor (cal aplicar l'operador '&' davant el nom de les variables que es passen com a paràmetre).

ex 3.10: Aquest exemple mostra com definir el pas de variables de tipus enter per referència. El programa principal crida a una funció que intercanvia el valor de dos variables enteres si la primera variable és menor que la segona. Com que passem les adreces de les variables, els canvis des de la funció a través dels punters afecten al contingut d'aquestes variables:

```
#include <stdio.h>

void maxmin(int *x, int *y)  /* dos paràmetres per referència */
{
    int temp;

    if(*x < *y)
     {
         temp = *x;          /* intercanviar valors variables externes */
         *x = *y;            /* associades als paràmetres (punters) */
         *y = temp;
     }
}

void main(void)
{
    int a,b;

    printf("introdueix 2 números:\n");
    scanf("%d",&a);          /* scanf() també necessita l'adreça de les */
    scanf("%d",&b);          /* variables perquè les ha de modificar */
    maxmin(&a,&b);           /* passem l'adreça de les variables amb '&' */
    printf("màxim : %d\n",a);
    printf("mínim : %d\n",b);
}
```

- Quan es tracta de passar un vector com a paràmetre és molt aconsellable fer-ho per referència. En el cas dels paràmetres només cal escriure uns claudàtors sense cap número dins. Per exemple:

```
void codifica(char s[])
{
    int i=0;

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

- En el cas dels *strings*, però, s'acostuma a definir els paràmetres com a punters a `char`, però és equivalent al mètode anterior;

ex 3.11: Definició de paràmetre *string* com a punter. El programa demana una frase i crida a una funció per a que la codifiqui i retorni la seva longitud:

```
#include <stdio.h>

int codifica(char *s) {
 int n=0;

 while(s[i++] != '\0') {
    s[i]++;              /* accés a un punter a char com si fos un vector */
    n++;
  }
 return(n);
}

void main(void) {
 int a;
 char frase[80];

 printf("introdueix una frase:\n");
 gets(frase);
 a = codifica(frase);         /* el nom del vector sense claudàtors és l'adreça
                                 base del vector */
 printf("frase codificada\n");
 printf("%s\n",frase);
 printf("%d car. codificats\n",a);
}
```

- En el cas dels tuples per paràmetre cal anar amb compte, ja que el C permet passar-les per valor (fa una còpia del contingut a la pila). Si les dades estructurades ocupen molt d'espai serà interessant passar-les per referència. Quan es passen per referència l'accés als camps del paràmetre és farà amb l'operador '`->`', ja que es tracta d'un punter a una estructura;

ex 3.12: El següent exemple mostra el pas de paràmetres de tuples tant per valor (només lectura) com per referència (lectura/escriptura):

```
#include <stdio.h>

typedef struct
 {  float x;
    float y;} punt2D;

void preguntar_punt(punt2D *p)      /* pas per referència */
 {
    printf("\tcoordenada x : ");
    scanf("%f",&(p->x));            /* passar l'adreça dels camps de */
    printf("\tcoordenada y : ");   /* l'estructura apuntada per p */
    scanf("%f",&(p->y));
 }
```

```
void puntmig(punt2D *p1, punt2D p2)
 {
    p1->x = (p1->x + p2.x) / 2;      /* per referència:  p1->x */
    p1->y = (p1->y + p2.y) / 2;      /* per valor:       p2.x */
 }


void main(void)
{
    punt2D a,b;

    preguntar_punt(&a);
    preguntar_punt(&b);
    punt_mig(&a,b);                  /* si és per referència, operador '&' */
    printf("\n(%.2f,%.2f)\n",a.x,a.y);
}
```

3.6. Dades dinàmiques.

- La qüestió de les dades dinàmiques fa referència a demanar memòria lliure al S.O. en temps d'execució. Per fer això disposem de la crida estàndard **`malloc()`** (definida a `ALLOC.H`), que se li passa la mida de l'espai de memòria a reservar i retorna un punter al principi del bloc assignat (si no hi ha prou memòria disponible retornarà un punter a `NULL`). Per tant, caldrà treballar amb punters.

- Per calcular l'espai que necessitem amb precisió disposem de l'operador de C **`sizeof()`**, que retorna la mida en bytes d'un tipus o d'una estructura (depenent del *hardware* i del compilador). La funció **`free()`** serveix per alliberar la memòria assignada (cal no descuidar-se d'alliberar tota la memòria que s'ha reservat).

ex 3.13: Aquest exemple demana memòria dinàmica al sistema per a un enter, una dada de tipus llibre, i un vector de 500 elements de tipus llibre. L'ús del `sizeof` permet independitzar el programa de la mida de les dades:

```
#include <alloc.h>              /* defineix malloc() i free() */

typedef struct
 {
    char nom[50];
    char autor[30];
    int tipus;
    int pagines;
 }llibre;          /* en un PC, 84 posicions de memòria per dada tipus llibre */


void main(void)
{
 int *pint;
 llibre *pa, *biblio;   /* no distingim entre un punter a una dada o un punter a
```

```
                                        un vector de dades */

 pint = (int *) malloc(sizeof(int)); /* reservar espai per un enter */
 *pint = 10;                         /* gravem un 10 a la posició de
                                        memòria que s'ha assignat */
  /* reservem espai per una dada tipus llibre */
 pa = (llibre *) malloc(sizeof(llibre));
 pa->pagines = 176;                  /* accedim a un camp de la dada */

  /* reservem espai per 500 dades tipus llibre */
 biblio = (llibre *) malloc(sizeof(llibre)*500);

 if(biblio!=NULL)                    /* si ens ha donat tota la memòria que voliem */
    biblio[2]->tipus = 1;            /* accedim a un camp del segon element */
:
:
 free(pint);
 free(pa);
 free(biblio);                       /* allibera tota la memòria assignada */
}
```

3.7. Exercicis.

Exercici 3.1: Com a exercici per practicar amb taules de punters a *string*s us proposem que feu un programa que declari un vector de 1000 `char`s, que permeti a l'usuari introduir paraules o frases que s'aniran emmagatzemant de forma consecutiva dins d'aquest vector (separades pel caràcter `'\0'`); que un vector de punters a `char` (ÍNDEX) vagi emmagatzemant les adreces de les paraules dins del vector de caràcters (màxim 50 paraules); que acabi la introducció de les paraules amb la paraula clau `"FI"` (no s'inclourà en la llista); i que, finalment, imprimeixi per pantalla la llista de paraules o frases per ordre alfabètic.Com a pista per a resoldre el problema podeu treballar sobre el fitxer `PCHR_ORD.C`, que mostra un esquema bàsic de funcions més comentaris per a resoldre el problema.

```
#include <stdio.h>
#include <string.h>
/* feu servir la funció strcmp(s1,s2); per comparar strings:
    aquesta funció retorna un enter que val 0 si s1=s2,
    <0 si s1<s2, >0 si s1>s2 */

char text[1000];   /* vector on es guardaran les frases
                      separades per '\0' */
char *frase[50];   /* vector on es guardaran els punters
                      al text */

void main(void)
{
 /* declaració variables locals */

 /* inicialitzacions */
```

```
 /* bucle llegir frase */
    /* llegir frase amb gets(&text[i]); on i és l'índex del
       caràcter on es col.locarà la següent frase */
/* repetir fins que la frase sigui "FI"*/

/* ordenar les frases fent servir, per exemple, l'algorisme
   de la bombolla */
/* NO INTERCANVIAR ELS CARACTERS, NOMES ELS PUNTERS! */
/* per i=0 fins i=nºfrases-2 fer */
    /* per j=i+1 fins nºfrases-1 fer */
    /* si (frase[j]>frase[i]) llavors
           frase[i]<=>frase[j] (intercanviar punters) */
   /* fiper j */
/* fiper i */

/* bucle imprimir frases */
}
```

Exercici 3.2: Per rematar l'exercici anterior, amplieu els rangs a 40.000 caràcters i 500 frases, estructureu el programa en funcions d'introduir, ordenar, i imprimir frases, i guardeu els caràcters sobre memòria dinàmica.

4. Accés a fitxers i Programació modular.

4.1. Entrada i sortida (fitxers).

Entrada i Sortida estàndard

- Quan llegim caràcters amb **`getchar()`** o escrivim informació de text amb **`printf()`**, se suposa que ho hem de fer a través del teclat i la pantalla. De vegades, però, la informació a introduir o visualitzar és massa extensa com per teclejar-la "a mà" o veure-la en pantalla. En aquests casos és interessant poder "xuclar" o gravar la informació usant fitxers, però sense canviar el programa. Això és possible gràcies al concepte de canals d'informació. El programa llegeix o escriu informació als canals, i el SO s'encarrega de "transportar" aquesta informació als dispositius adients.

- Els canals d'entrada i sortida estàndard estan "connectats", per defecte, al teclat i a la pantalla. Si quan instanciem un programa des de l'intèrpret de comandes especifiquem que els canals d'entrada o de sortida són fitxers, el SO s'encarregarà d'obrir els fitxers i transferir la informació entre aquests i el programa. En altres paraules, el programa no sap si llegeix de teclat o de fitxer, només sap que llegeix del canal estàndard d'entrada.

ex 4.1: El programa calcula el número de línies, paraules i caràcters que s'introdueixen per l'entrada estàndard i imprimeix els resultats per la sortida estàndard. Per detectar línies busca el caràcter fi de línia '`\n`'. Per detectar paraules es defineix un booleà (`int estat`) que recorda si estava llegint una paraula quan troba un espai en blanc, un fi de línia o un tabulador ('`\t`'):

```
#include <stdio.h>

#define DINTRE  1   /* dintre d'una paraula */
#define FORA    0   /* fora d'una paraula */

/* compta línies, paraules i caràcters de l'entrada */

void main(void)
{
 int c, nl, np, nc, estat;

 estat = FORA;
 nl = np = nc = 0;
 while ((c = getchar()) != EOF) {
    ++nc;
    if (c == '\n') ++nl;
    if (c == ' ' || c == '\n' || c == '\t') estat = FORA;
    else if (estat == FORA) { estat = DINTRE;
                              ++np;
                            }
   }
  printf("línies:%d\nparaules:%d\ncaràcters:%d\n",nl,np,nc);
}
```

- Per practicar amb aquests conceptes disposem del fitxer `COMPTAPR.C` al directori `SEC4`. Si el compileu i l'executeu, haureu d'introduir un cert número de línies de text seguides d'un caràcter '**^Z**' (tecla `Ctrl` + `Z`), més un `<Return>`, per simular un fi de fitxer (**EOF**):

```
C:\...\> COMPTAPR
la mar estaba salada
salada estaba la mar
mar serena...
^Z
línies : 3
paraules : 12
caràcters : 52
```

Per redireccionar el canal d'entrada a un fitxer, des de l'intèrpret de comandes escriviu el nom del programa més un nom de fitxer precedit pel caràcter '**<**'. Per exemple, comptem les línies, paraules i caràcters del fitxer `COMPTAPR.C`:

```
C:\...\> COMPTAPR <COMPTAPR.C
línies : 35
paraules : 148
caràcters : 1173
```

Per redireccionar el canal de sortida a un fitxer, des de l'interpret de comandes escriviu el nom del programa més un nom de fitxer precedit pel caràcter '**>**'. Per exemple, comptem les línies, paraules i caràcters del teclat i escriviu els resultats sobre el fitxer `RESULT.TXT` (el crearà en aquell moment). Després visualitzarem el fitxer de resultats amb `TYPE`:

```
C:\...\> COMPTAPR >RESULT.TXT
la mar estaba salada
salada estaba la mar
mar serena...
^Z

C:\...\> TYPE RESULT.TXT
línies : 3
paraules : 12
caràcters : 52
```

També podem combinar els dos redireccionaments:

```
C:\...\> COMPTAPR <COMPTAPR.C >RESULT2.TXT
```

- Una de les possibilitats més interessants del redireccionament de canals és la de transferir informació entre els perifèrics de l'ordinador i el programa (sense canviar el codi del programa). Podem, per exemple, llegir el text del port sèrie i escriure el resultat en la impressora:

```
C:\...\> COMPTAPR <COM1: >PRN:
```

Accés a fitxers

- Per accedir als fitxers directament des de programa disposem d'una sèrie de funcions estàndard, les definicions de les quals es troben al *Header* `STDIO.H`. Algunes de les més importants són:

tipus:
`FILE`

defines:
`EOF`

funcions d'operació de fitxers :
`fopen();`
`fclose();`
`fread();`
`fwrite();`
`fseek();`
`feof();`
`ferror();`

funcions d'entrada/sortida sobre fitxers:
`fprintf();`
`fscanf();`
`fgetc();`
`fputc();`
`fgets();`
`fputs();`

ex 4.2: El programa `DEFINES.C` implementa un algorisme que imprimeix les línies d'un fitxer que comencin per la paraula clau "`#define`":

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

void main(void)
{
 char lin[80];
 FILE *fit;

 printf("Nom d'un fitxer : "); scanf("%s",lin);

 fit = fopen(lin,"r");
 if(fit==NULL)  printf("Error obrint fitxer\n");
 else
 {
    while (!feof(fit));
     {
         fgets(lin,80,fit);
         if(strncmp(lin,"#define",7)==0) printf("%s",lin);
     }
    fclose(fit);
 }
}
```

- El procediment normal per operar amb fitxers és cridar a la funció **`fopen()`**, que rep com a paràmetres el nom del fitxer a disc que es vol obrir (nom físic) i el mode d'accés al fitxer (`"r"` llegir, `"w"` escriure, etc.), i retorna un apuntador a una estructura `FILE`, que serà emmagatzemada sobre una variable de programa, i que serà aquesta variable la que es farà servir per identificar el fitxer en totes les altres funcions (nom lògic). Si no ha pogut obrir el fitxer retorna un punter `NULL`.

- La resta de funcions d'operació permeten llegir *buffer*s (blocs) de caràcters, escriure *buffer*s, posicionament dins del fitxer, detecció de fi de fitxer, detecció d'errors i tancament de fitxer. És important no oblidar-nos de tancar tots els fitxers abans de sortir del programa, per no perdre informació.

- Les funcions d'entrada/sortida sobre fitxers són les mateixes que les d'entrada i sortida típiques (**`printf()`**, **`gets()`**, etc.) però aquesta vegada la informació es transfereix al fitxer indicat (s'ha de passar per paràmetre el punter a `FILE` corresponent). De fet es pot considerar a les típiques com un cas particular de les de fitxers, ja que les primeres accedeixen als punters a "fitxers" d'entrada i sortida estàndard.

Exercici

- A partir del fitxer `DEFCON2.C`, implementeu un programa que imprimeixi la llista de línies d'un fitxer d'entrada que comencin per una paraula clau que introduirà l'usuari. A més, l'usuari podrà introduir els noms dels fitxers d'entrada i de sortida per la línia de comandes del S.O., és a dir, pas de paràmetres a la funció `main()`. Si entra un paràmetre, es tracta del nom del fitxer d'entrada. Si hi han dos paràmetres, són el fitxer d'entrada i el de sortida:

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

int main(int nc, char *targ[])
{
 char lin[80];           /* per llegir linies de texte */
 FILE *fit_e, *fit_s;

 if(nc==1)               /* no s'han introduit paràmetres */
 {
    printf("introdueix almenys 1 paràmetre\n");
    return(1);
  }
 else
 {
    fit_e = fopen(targ[1],"r");
    if(fit_e==NULL)
     {
         printf("Error obrint fitxer\n");
         return(2);
     }
    else
     {
       if(nc>2)        /* hi ha més d'un paràmetre */
         {
              fit_s = fopen(targ[2],"w");

    /* etc...*/

     }
 }
}
```

4.2. Compilació separada.

- Si un programa és molt llarg, és convenient "partir-lo" en troços modulars. És a dir, agrupar en fitxers separats les funcions que estan lògicament relacionades, per exemple, les funcions d'inicialització, càlculs, representació gràfica, funcions principals, etc. Després es pot compilar cada mòdul per separat, testejar de forma independent, reaprofitar funcions per altres programes, etc.

- Quan en un mòdul es crida a funcions d'altres mòduls, cal definir els prototipus de les funcions "externes", per a que el compilador pugui fer comprovació de tipus quan compili el mòdul que crida (no té accés als altres mòduls). Per fer això el C proporciona l'operador **`extern`**, que indica que les funcions i variables que s'especifiquen darrera es troben en altres mòduls.

ex 4.3: Com a exemple pedagògic definirem el mòdul `LONGCAD.C` amb la funció `strlen()` que retorna la longitud d'una cadena de caràcters, i un mòdul `PRIMER.C` amb la funció principal `main()` que crida a la funció `strlen()` :

/* longcad.c */

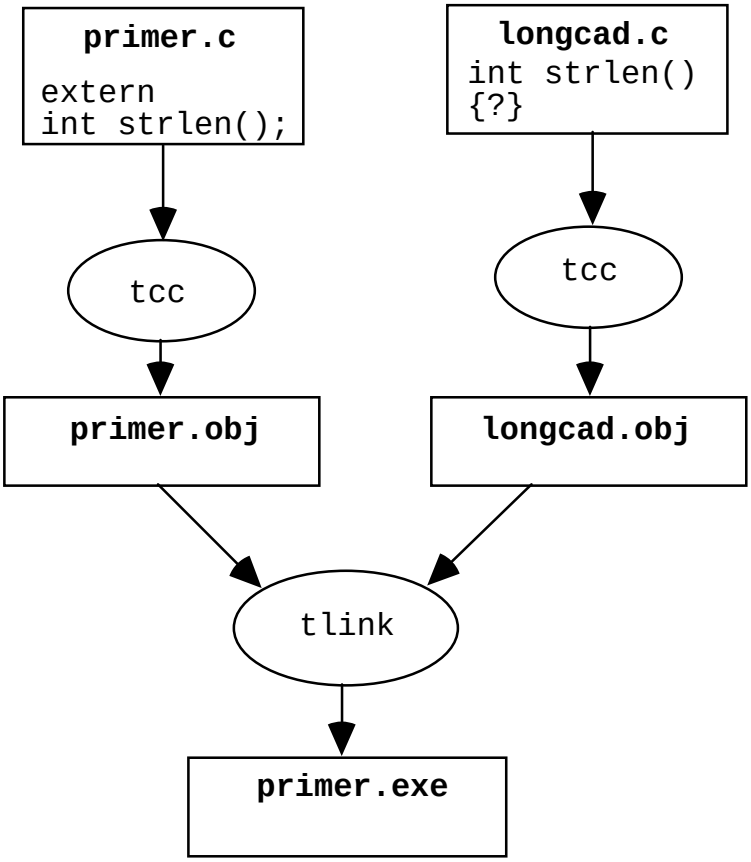
```
int strlen(char s[])
{
    int i;

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

/* primer.c */

```
#include <stdio.h>          /* prototipus extern de funcions printf, ... */

extern int strlen(char s[]);        /* prototipus de la funció externa */

void main(void)
{
    int l;
    char frase[80];

    printf("introdueix un frase :\n");

    gets(frase);            /* crides a funcions externes */
    l=strlen(frase);

    printf("la frase té %d caràcters\n",l);
}
```

Representació gràfica de la compilació separada:

```
C:\...\> tcc -c longcad.c
C:\...\> tcc -c primer.c
C:\...\> tlink primer.obj longcad.obj
C:\...\> primer
```



- Si volem "amagar" la implementació de les funcions d'un mòdul, i volem especificar els prototipus de les funcions públiques, la solució és declarar un fitxer de capçaleres ("**.H**"), que podrem incloure (**#include**) en els mòduls que vulguin fer servir aquestes funcions. Quan fem l'`include` dels nostres fitxers *Header*, hem d'escriure el *path* del fitxer entre cometes ("") i no entre signes de menor-major (<>), ja que els segons serveixen per buscar els fitxers en els directoris declarats al compilador com a directoris de fitxers de capçalera per defecte (`C:\TC\INCLUDE`). També podem declarar els nostres directoris com a directoris per defecte.

ex 4.4: El mateix algorisme que abans, però la declaració de prototipus es deixa en el fitxer `LONGCAD.H`, per a que tots els mòduls que vulguin cridar a la funció `strlen()` tinguin el prototipus ja declarat:

/* longcad.h */

```
extern int strlen(char s[]);
```

/* longcad.c */

```
int strlen(char s[])
{
    int i;

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

/* primer.c */

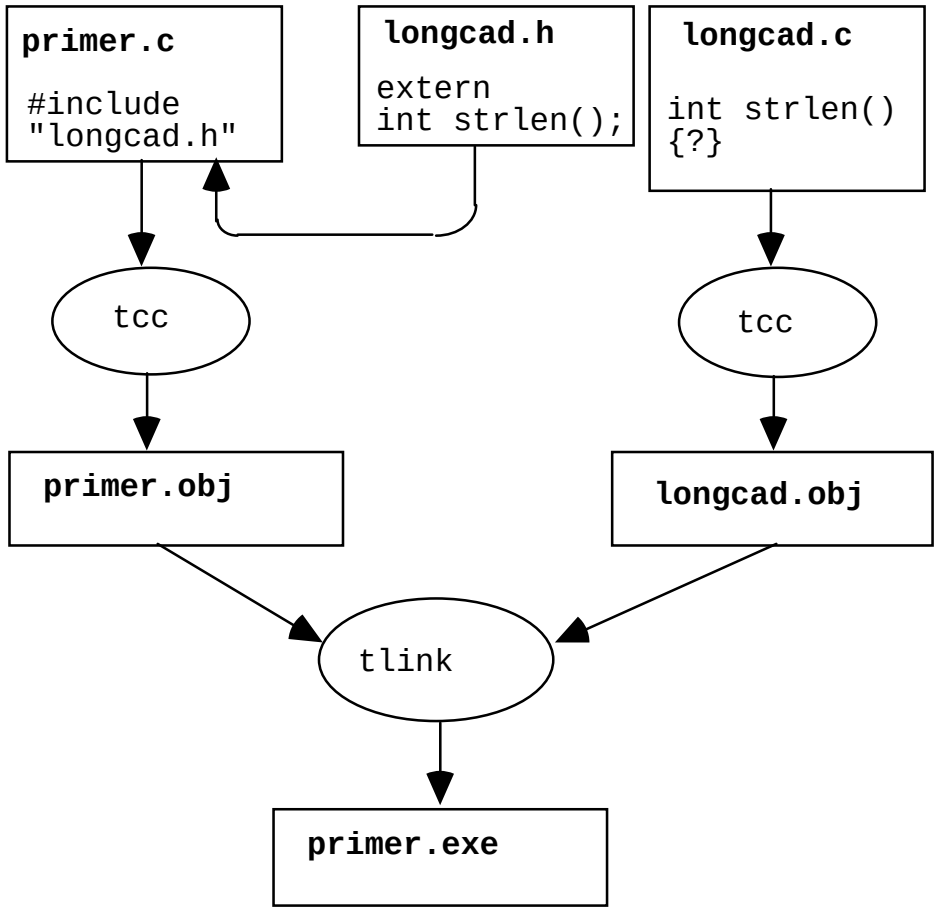
```
#include <stdio.h>
#include "longcad.h"      /* inclou declaració de prototipus */
                          /* fixar-s'hi amb les cometes → el fitxer LONGCAD.H
                             es troba al diretori actual */
void main(void)
{
    int l;
    char frase[80];

    printf("introdueix un frase :\n");
    gets(frase);
    l=strlen(frase);
    printf("la frase té %d caràcters\n",l);
}
```

- Cal comentar que de vegades es pot observar que es fan includes de fitxers `.C`. Aquesta "tècnica" **no és compilació separada**, ja que s'inclouen els moduls i es compilen tots junts, la qual cosa incrementa el temps de compilació així com la possibilitat de definir llibreries de funcions compartibles entre diferents programes.

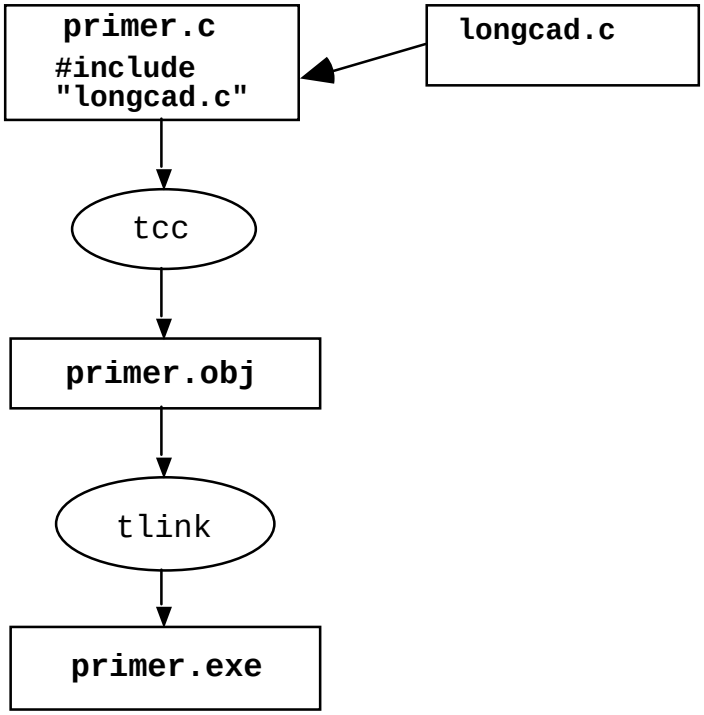
cex 4.1: En aquest contraexemple, el fitxer `LONGCAD.C` és incluit en el fitxer `PRIMER.C` i es compila tot alhora:

```
#include <stdio.h>
#include "longcad.c"

void main(void) {
    int l;
    char frase[80];

    printf("introdueix un frase :\n");
    gets(frase);
    l=strlen(frase);
    printf("la frase te %d caràcters\n" ,l);
}
```

Representació gràfica de la compilació separada, amb fitxers de prototipus :



Representació gràfica de la inclusió d'un fitxer `.C`:

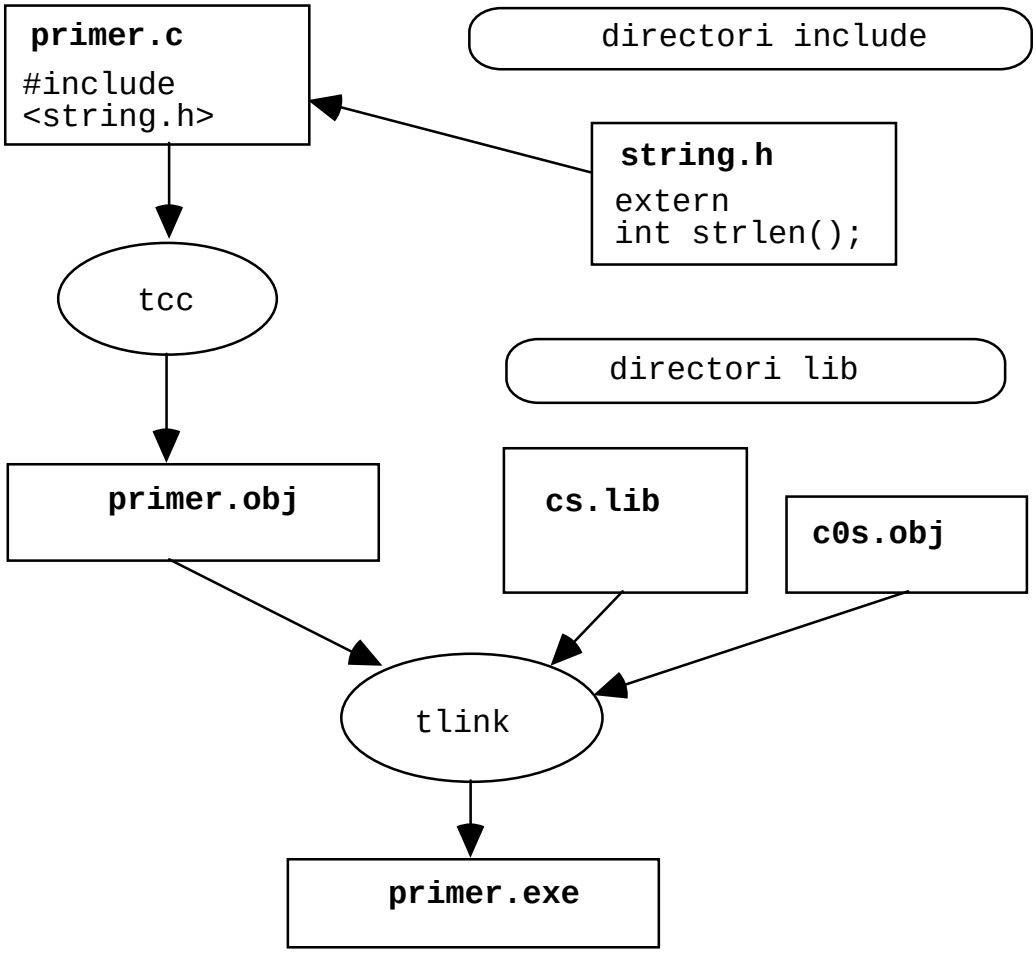


- Per últim, indicar que els mòduls `.OBJ` es poden agrupar en llibreries de funcions amb l'aplicació `LIB.EXE` (o `TLIB.EXE`). Un exemple són les llibreries estàndard, que podreu observar al directori `C:\TC\LIB` (estan repetides per a diferents models de compilació `CT.LIB`, `CS.LIB`, `CC.LIB`, `CM.LIB`, `CL.LIB` i `CH.LIB`, que corresponen als models *Tiny*, *Small*, *Compact*, *Medium*, *Large* i *Huge*). També és necessari el codi d'arranc d'aplicació (*Startup* → `C0T.OBJ`, `C0S.OBJ`, etc.), que el `TLINK` enllaçarà amb el nostre fitxer `.EXE` de forma automàtica, per a que la nostra aplicació pugui arrencar.

ex 4.5: Utilitzar la rutina estàndard `strlen()` per calcular la longitud d'una cadena:

/* primer.c */

```
#include <stdio.h>
#include <string.h>            /* prototipus funció strlen() estàndard */

void main(void)
{
    int l;
    char frase[80];

    printf("introdueix un frase :\n");
    gets(frase);
    l=strlen(frase);
    printf("la frase te %d caràcters\n",l);
}
```



4.3. Exercicis.

Exercici 4.1: Feu un programa per gestionar una agenda de telèfons (màxim 50.000 telèfons). Definir el típus "telèfon" i estructureu el programa en les funcions que cregueu convenients. El programa mantindrà les dades de l'agenda sobre fitxers i, quan carregui la informació a memòria, caldrà gestionar memòria dinàmica per manegar grans volums de dades (no cal que es carregui tota la informació de cop, podem paginar per primera lletra del primer cognom, per exemple). Si el programa és molt llarg, estructurar-lo en diferents mòduls (compilació separada).

```
tipus telèfon
     nom,
     adreça,
     prefix,
     número, ...

funcions principals
     inserir,
     esborrar,
     consultar,
     modificar, ...
```

Exercici 4.2: Fer un programa editor de text, tipus Norton Editor.

La versió `HTML` d'aquests apunts es va generar a partir d'un fitxer `RTF` amb la utilitat *shareware* **`rtftohtml`** versió 3.0.a1 de Chris Hector (chris@sunpack.com).

La darrera versió d'aquesta utilitat i altres informacions sobre la mateixa es poden trobar a `http://www.sunpack.com/RTF`.