Gestió de dades estàtiques compostes

3

Isidre Guixà i Miranda
IES SEP Milà i Fontanals, d'Igualada

Programació estructurada i modular

Setembre del 2007
© Isidre Guixà i Miranda
IES SEP Milà i Fontanals
C/. Emili Vallès, 4
08700 - Igualada

En cas de suggeriment i/o detecció d'error podeu posar-vos en contacte via el correu electrònic iguixa@xtec.cat

Cap part d'aquesta publicació, incloent-hi el disseny general i de la coberta, no pot ser copiada, reproduïda, emmagatzemada o tramesa de cap manera ni per cap mitjà, tant si és elèctric, com químic, mecànic, òptic, d'enregistrament, de fotocòpia, o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Índex 3
Introducció 5
Objectius 7
1. Taules 8
  1.1. Estructures de dades. Classificació. 8
  1.2. Què són les taules? 11
  1.3. Taules en pseudocodi i en llenguatge C 12
  1.4. Algorismes bàsics en taules unidimensionals 16
    1.4.1. Recorregut total 17
    1.4.2. Afegir pel final 19
    1.4.3. Recerca d'un valor 20
    1.4.4. Exemples introductoris a la manipulació de taules 21
    1.4.5. Recerca d'un valor amb tècnica de sentinella 30
    1.4.6. Exemples d'aprofundiment en la manipulació de taules 34
  1.5. Gestió de taules multidimensionals 42
    1.5.1. Recorregut total 43
    1.5.2. Recerca d'un valor 44
    1.5.3. Exemples de manipulació de taules multidimensionals 46
2. Cadenes 54
  2.1. Cadenes en pseudocodi 58
  2.2. Ordre lexicogràfic 60
  2.3. Cadenes en llenguatge C 61
    2.3.1. Declaració de variables 61
    2.3.2. Escriptura de cadenes 62
    2.3.3. Lectura de cadenes 62
    2.3.4. Funcions de tractament de cadenes 66
  2.4. Exemples de manipulació de cadenes 68
3. Tuples i tipus de dades 70
  3.1. Tuples i tipus de dades en pseudocodi 70
  3.2. Tuples i tipus de dades en llenguatge C 74
  3.3. *Unions* en llenguatge C 76

Introducció

En les unitats didàctiques precedents hem treballat amb variables de tipus diferents que ens permetien emmagatzemar un valor del tipus corresponent. D'altra banda, ja hem comentat més d'una vegada que la informàtica pretén gestionar grans volums d'informació. Ràpidament apareix la pregunta: de quina manera podem tractar grans volums d'informació amb variables que només permeten emmagatzemar una dada?

Per poder donar resposta a aquesta qüestió apareixen uns nous tipus de dades que, a diferència dels tipus de dades vistos fins ara, ens han de permetre emmagatzemar paquets d'informació.

En la unitat didàctica "Introducció a la programació" hem iniciat l'aprenentatge de les dades estàtiques simples. Són estàtiques perquè es defineixen en el moment d'escriure el codi font del programa i són simples perquè ens permeten emmagatzemar-hi un únic valor (caràcter, natural, enter, real o lògic).

En aquesta unitat didàctica ampliarem la visió que tenim de les dades estàtiques amb el coneixement de les dades estàtiques compostes, que ens permetran l'emmagatzematge d'un conjunt de valors.

En el nucli d'activitat "Taules" presentem el tipus de dada taula ideat per a emmagatzemar conjunts de valors d'un mateix tipus, com: un conjunt de caràcters, un conjunt d'edats, un conjunt d'alçades,... Se'n mostra la seva definició aplicable a la majoria de llenguatges de programació estructurats i, en especial, en el llenguatge C, així com els algorismes més usuals de tractament.

En el nucli d'activitat "Cadenes" presentem un tipus especial de taula destinat a gestionar informació alfanumèrica (paraules, frases,...). La informació alfanumèrica es pot gestionar amb la utilització de taules de caràcters, però això és una bestiesa! Us imagineu que per a buscar una paraula en una frase haguéssim d'anar mirant caràcter a caràcter i no poguéssim tenir una percepció global de la paraula? A l'igual que en el primer nucli, se'n mostra la definició genèrica i la implementació en llenguatge C.

Per últim, en el nucli d'activitat "Tuples i tipus de dades" presentem el tipus de dada tupla ideat per a emmagatzemar conjunts de valors de diferents tipus, com per exemple, les dades d'una persona, consistents

en nom i cognoms que són dades alfanumèriques, pes i alçada que són dades numèriques, sexe i estat civil que poden ser dades caràcter,... Així mateix, en aquest nucli, mostrem les possibilitats que aporten els llenguatges estructurats per a que el programador defineixi tipus de dades. I tot això ho apliquem en el llenguatge C.

Per aconseguir els nostres objectius, heu de reproduir en el vostre ordinador i, a ser possible, en diverses plataformes, tots els exemples incorporats en el text, per a la qual cosa, en la secció "Recursos de contingut", trobareu tots els arxius necessaris, a més de les activitats i els exercicis d'autoavaluació.

Objectius

A l'acabament d'aquesta unitat didàctica, l'estudiant ha de ser capaç de:

1. Dissenyar algorismes en pseudocodi per a resoldre problemes amb la utilització de tipus de dades estàtiques compostes.

2. Identificar les situacions en què cal fer servir les dades compostes per a resoldre els problemes.

3. Combinar els diferents tipus de dades compostes en el disseny dels algorismes.

4. Fer servir de manera adequada el tipus de dada cadena.

5. Codificar en llenguatge C programes que facin servir taules, cadenes i estructures.

6. Fer servir de manera adequada el depurador del llenguatge C per al seguiment de l'execució dels programes que gestionen dades compostes.

1. Taules

Fins el moment present, dissenyem programes que defineixen variables que permeten guardar una dada en un moment donat. En la pràctica això no és suficient, doncs les aplicacions informàtiques gestionen grans volums d'informació. Ens proposem, doncs, introduir el concepte de taula per iniciar la gestió, en els programes que dissenyem, de conjunts d'informació d'un mateix tipus.

Les taules s'emmarquen dins els tipus de dades compostes ja que permeten gestionar conjunts d'informació. No són, però, els únics tipus de dades compostes existents i, per a situar-nos en el tema, abans de treballar les taules, introduirem el concepte d'estructures de dades i en farem alguna classificació.

1.1. Estructures de dades. Classificació.

Imaginem-nos que volem fer un estudi estadístic sobre les qualificacions que obtenen els alumnes d'una classe. És clar que necessitarem, en primer lloc, emmagatzemar les notes per, posteriorment, fer els càlculs que pertoquin (mitjana, desviació, etc.). On les emmagatzemem? Suposant que la classe té, a tot estirar, vint-i-cinc alumnes, caldria tenir vint-i-cinc variables de tipus adequat (real o natural, segons que les qualificacions tinguin decimals o no en tinguin). A més, com ho farà el programa per efectuar les lectures? Hi haurà d'haver vint-i-cinc instruccions llegir (ja que el nom de la variable canviarà per cada valor) i per tant no es pot utilitzar una instrucció repetitiva. I si el programa està preparat per a vint-i-cinc alumnes, però una classe en concret només en té vint?

Oi que estaria bé disposar d'una espècie de contenidor que ens permetés guardar fins a vint-i-cinc notes i que ens dotés d'eines per a resoldre les situacions abans esmentades? Es tractaria d'un contenidor per a emmagatzemar un conjunt de dades d'un mateix tipus (diguem-ne contenidor de tipus A).

Imaginem-nos una altra situació. Hem de gestionar les dades personals d'una persona (el nom, el DNI, la data de naixement, el sexe, l'adreça, etc.). Com ho fem? És evident que per a poder gestionar aquestes dades necessitarem, primer de tot, poder-les emmagatzemar per, posteriorment, efectuar les gestions pertinents (com, per exemple, imprimir la fitxa corresponent). On ho emmagatzemem? Au, som-hi! Ja

comencem a pensar en un cert nombre de variables de diferents tipus, en aquest cas, amb capacitat per a guardar les dades corresponents.
Oi que també estaria bé disposar d'una espècie de contenidor que ens permetés tenir agrupades les diferents dades corresponents a una persona, amb eines per a una gestió eficaç? Es tractaria d'un contenidor per a emmagatzemar un conjunt de dades relacionades entre si (diguem-ne contenidor de tipus B).

I encara més; si volem gestionar les dades personals dels alumnes d'una escola, oi que ens interessaria poder tenir un contenidor de tipus A amb capacitat per a un determinat nombre (la màxima quantitat d'alumnes) de contenidors de tipus B (cada un dels quals contingués les dades d'un alumne)?

Doncs bé, els llenguatges de programació acostumen a aportar les eines necessàries per a solucionar les situacions anteriors. Són les anomenades estructures de dades.

> Una estructura de dades és una representació d'un conjunt homogeni d'informació que permet un tractament global (tractant el conjunt) i unitari (tractant cada element del conjunt).

Les estructures de dades corresponen a representacions de la realitat i les podem classificar en internes i externes en funció del lloc en què han de ser representades.

> Estructures de dades internes són les que es representen en la memòria interna de l'ordinador.
>
> Estructures de dades externes són les que es representen en la memòria externa de l'ordinador, és a dir, en els suports d'emmagatzematge externs de què disposa l'ordinador.

Tots els algorismes i programes que hem desenvolupat fins ara gestionen dades, introduïdes per l'usuari o no, en la memòria de l'ordinador. Acabem de presentar les estructures de dades internes, les quals ens permetran donar coherència a conjunts homogenis de dades ubicats, també, en la memòria de l'ordinador. Però l'accés a les dades de la memòria desapareix quan s'acaba l'execució del programa i les dades desapareixen definitivament quan es talla el subministrament elèctric. És clar que necessitem mecanismes per a enregistrar les dades de manera que les puguem gestionar quan ens convingui,

augmentant-les, disminuint-les o modificant-les. En aquest punt entren en joc les estructures de dades externes.

La caducitat de la informació

De manera semblant, l'ésser humà reté en la seva memòria un gran volum d'informació, que desapareix amb la mort d'aquell, mentre que la informació enregistrada en els suports diferents utilitzats per l'ésser humà han esdevingut perennes al pas del temps.

Els grans volums d'informació que s'han de gestionar amb tractament informàtic han propiciat un gran desenvolupament de les estructures de dades externes. Aquestes van començar la seva evolució amb els arxius, dels quals cal distingir, per ordre d'aparició i de menor a major nombre de prestacions els seqüencials, els relatius i els indexats. Dels arxius indexats, màxima sofisticació en arxius, les estructures de dades externes van evolucionar amb l'aparició de les bases de dades, de les quals cal distingir, també per ordre d'aparició, les bases de dades en xarxa, les jeràrquiques i les relacionals.

Pel que fa a les dades, les estructures de dades internes se subdivideixen en dos grans apartats: estàtiques i dinàmiques.

> Les estructures de dades estàtiques existeixen durant tota l'execució i tenen una capacitat fixa, mentre que les dinàmiques es poden crear i destruir en temps d'execució, segons evolucioni el programa, i poden acomodar la seva capacitat a les necessitats reals de l'execució.

Les estructures de dades estàtiques han estat, tradicionalment, taules i tuples; les dinàmiques han estat llistes, piles, cues i arbres.

Molt més recent és el concepte d'objecte, que correspon al fet d'ajuntar, en un sol ens, les estructures de dades i els algorismes necessaris per a manipular-les. El concepte d'objecte provoca l'aparició d'un nou estil de programació, molt popular en l'actualitat, anomenat programació orientada a objectes que no és objecte d'estudi en la programació estructurada i modular.

La taula 1 esquematitza els diferents tipus de dades presentats.

Taula 1. Esquema sobre els tipus de dades presentats.

Tipus de dades			
Simples	Internes	• Natural • Enter • Real	Estàtiques
		• Caràcter	

Tipus de dades			
		• Lògic	
Compostes	Internes	• Taules • Tuples	
		• Llistes • Piles • Cues • Arbres	Dinàmiques
	Externes	Arxius	
		Bases de dades	

1.2. Què són les taules?

La primera estructura de dades que estudiarem és la taula.

> Una taula és un conjunt finit de dades del mateix tipus i de la mateixa grandària que estan col·locades consecutivament en la memòria de l'ordinador, són reconegudes per un nom comú –que és el nom de la taula– i a les quals s'accedeix per la posició que ocupen dins el conjunt.

Pensem en l'espai necessari per a emmagatzemar les notes de fins a vint-i-cinc alumnes; desitgem disposar d'un recipient on puguem encabir les vint-i-cinc notes (reals o naturals, segons que calgui tractament de decimals o no). El recipient que pertoca utilitzar és una taula de reals o naturals (totes les dades han de ser del mateix tipus) amb una capacitat màxima per a vint-i-cinc elements.

El fet que a cada element d'una taula s'accedeixi per la posició que ocupa dins el conjunt de dades fa que hi hagi un índex numèric que permeti l'accés corresponent. Els llenguatges utilitzen diferents possibilitats d'índex. Així, per a una taula de 100 posicions, podem tenir índexs dels tipus següents:

- Des d'1 fins a 100 (la majoria de llenguatges).
- Des de 0 fins a 99 (llenguatge C).
- Índex definible pel programador: "-50,...,0,...,49", "-100,...,-1".

Una taula no deixa de ser una variable i, per tant, cal tenir-la declarada a la zona de declaracions de variables. Per a poder efectuar aquesta declaració es necessita saber la dimensió (quantitat màxima de dades que pot emmagatzemar) i la naturalesa dels components.

Els elements que constitueixen una taula poden ser de qualsevol tipus: numèric (naturals, enters, reals), caràcter, lògic i altres estructures compostes (taules, tuples). En el cas de taules de taules (o sigui taules en què els components també són taules) es pot parlar directament de taules multidimensionals; la majoria de llenguatges disposen de la possibilitat de declarar-les directament.

Les taules, en general, no disposen d'operacions específiques (d'assignació, relacionals, aritmètiques, d'entrada/sortida, etc.) com la resta de tipus de dades vistos fins al moment. Les taules permeten, pel que fa als elements que contenen, totes les operacions permeses en els corresponents tipus de dades als quals pertanyen.

Operacions amb taules

En algun llenguatge ens podem trobar accions o funcions predefinides que ens permetin fer certs tipus d'operacions amb les taules.

Exemples d'operacions permeses en taules

És a dir, si tenim tres taules A, B i C d'enters, el llenguatge no ens permet fer coses semblants a A = B + C, operació que tothom entén com que la taula A recull el resultat de sumar, un a un, els elements de les taules B i C.

I doncs, no podrem fer un càlcul semblant a l'anterior? La resposta ha de ser afirmativa, però serà tasca del programador efectuar els càlculs necessaris. Els llenguatges no acostumen a oferir-nos aquests tipus de funcionalitats.

1.3. Taules en pseudocodi i en llenguatge C

La declaració d'una variable taula en pseudocodi segueix la sintaxi següent:

Array

El terme anglès que se sol fer servir per a referir-se a les taules és array. Cal anar amb compte amb les traduccions llatinoamericanes dels textos anglesos, ja que solen traduir array per arranjament, quan en realitat hauria de ser taula, vector o matriu.

```
var  <nom>: taula [<dim>] de <tipus>;
fivar
```

en la qual:

- <nom> és el nom de la variable,
- <dim> és un valor constant natural positiu no-zero, que correspon a la capacitat de la taula,
- <tipus> és el tipus de dada que pot emmagatzemar la taula a les diferents posicions.

Observem-ne alguns exemples:

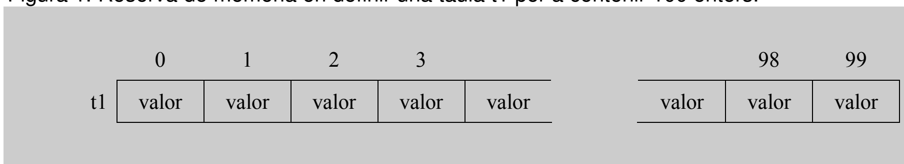
```
t1: taula [100] de enter;    /* taula de 100 enters */
t2: taula [100] de taula [50] de caràcter;
/* taula de 100 taules de 50 caràcters */
t3: taula [50] de caràcter;  /* taula de 50 caràcters */
```

El primer i cas i el tercer fan referència a una taula unidimensional. S'utilitza molt el terme vector per a fer referència a una taula unidimensional.

En el segon cas hi ha una taula bidimensional. En aquest cas i en els casos multidimensionals se sol fer servir el terme matriu.
En pseudocodi prenem la decisió de numerar les diferents posicions a partir de zero. Aquesta decisió està motivada per la similitud amb el llenguatge C (en què haureu de posar en pràctica els vostres coneixements).

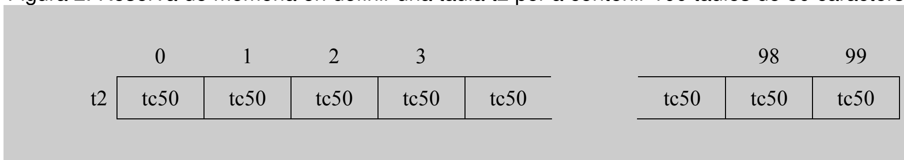
Gràficament, les anteriors declaracions produeixen, en la memòria de l'ordinador quan s'inicia l'execució del programa, les reserves d'espai que s'observen a les figures 1, 2 i 3

Figura 1. Reserva de memòria en definir una taula t1 per a contenir 100 enters.



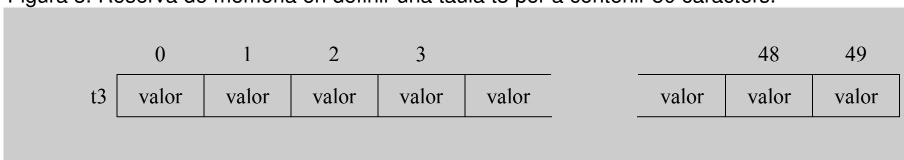
A la figura 1 hi observem cent cel·les cadascuna de les quals està numerada amb un número natural entre 0 i 99. Cada cel·la conté un <valor> de tipus enter (ja que la taula t1 és taula d'enters). No podem assegurar quin valor hi ha inicialment. És responsabilitat del programador inicialitzar les cel·les que convingui amb els valors que correspongui.

Figura 2. Reserva de memòria en definir una taula t2 per a contenir 100 taules de 50 caràcters.



A la figura 2 també hi observem cent cel·les numerades a partir de zero. Cada cel·la conté un <tc50>, que no és més que una taula de cinquanta caràcters.

Figura 3. Reserva de memòria en definir una taula t3 per a contenir 50 caràcters.



A la figura 3 hi observem cinquanta cel·les destinades a contenir, cadascuna, un valor de tipus caràcter.

Una taula no és més que una forma de mantenir agrupades un conjunt de variables del mateix tipus a les quals s'accedeix per la posició que ocupen dins el conjunt. La sintaxi que s'ha d'utilitzar per a accedir a la variable que es troba a la posició <pos> d'una taula <nom> és:

```
<nom>[<pos>]
```

És responsabilitat del programador controlar que no es demani una posició inexistent. La immensa majoria de llenguatges avorten la seva execució si el programa intenta accedir a una posició de taula inexistent.

La sintaxi per a declarar taules en llenguatge C és molt semblant a la que hem comentat per al pseudocodi:

```
...
<tipus> <nom>[<dim>];
...
```

Així, la codificació en llenguatge C dels exemples vistos més amunt seria:

```
int t1[100];
char t2[100][50];
char t3[50];
```

Una taula de C pot inicialitzar-se en el moment de la seva declaració, de la manera següent:

```
int t[5] = { 2, 4, 6, 8, 10};
```

o també:

```
int t[] = { 2, 4, 6, 8, 10};
```

La diferència entre les declaracions anteriors està en el fet que en la segona no s'ha indicat explícitament el nombre d'elements. En aquest cas, el programa compilador ho calcula pel nombre de valors especificats.

La sintaxi en llenguatge C per a accedir a una posició de la taula és idèntica a la del pseudocodi.

El llenguatge C suposa que l'usuari del llenguatge, és a dir, el programador, ja és prou intel·ligent per saber el que vol i, per tant, el llenguatge l'obeeix a cegues. Així, si un programa en C té una instrucció que obliga a accedir a una posició inexistent d'una taula, el llenguatge C no avorta l'execució del programa, a diferència de la immensa majoria de llenguatges. Què fa, doncs?

En general, una taula ocupa unes posicions consecutives de memòria. Suposem la declaració `int t[100]`. Aquesta declaració provoca que el programa en execució reservi una zona de 400 bytes (100 posicions per 4 bytes, que és la grandària de cada `int` en una plataforma de 32 bits). Quan el programa troba un accés del tipus `t[50]`, sap a quin byte ha

d'anar, ja que a partir del començament de la taula compta 50*4 bytes cap endavant i ja es troba situat en els dos bytes corresponents a l'enter `t[50]`. Què passa, doncs, si el programa troba un accés del tipus `t[150]`? Doncs que el programa actua exactament igual, i a partir de l'inici de la taula compta cap endavant 150*4 bytes. És clar que s'està accedint a una posició de memòria que no pertany a la taula: és problema del programador! Si el programa accedia a aquesta posició per a veure el valor `int` que hi havia enregistrat i fer-hi qualsevol tipus de tractament, ho continuarà fent, prenent els dos bytes que es troben en aquella posició de memòria i interpretant-los com un `int`. Si el programa accedia a la posició per a gravar-hi un valor `int`, l'hi gravarà, i pot succeir qualsevol barbaritat en el funcionament del programa i de l'ordinador, ja que es pot estar gravant en posicions de memòria corresponents a altres variables del programa i, fins i tot, en posicions de memòria corresponents a la gestió del sistema operatiu.

A l'apartat "Tipus de dades simples en el llenguatge C" de la unitat didàctica "Introducció a la programació" s'introdueixen els operadors ++ i -- del llenguatge C.

Sabem de l'existència dels operadors `++` i `--` per a incrementar i decrementar valors numèrics i que la seva utilització a la dreta o a l'esquerra de la variable que es vol incrementar o decrementar és, indiferent a efectes del valor final de la variable afectada. És a dir, en la següent situació, veiem que el valor final de la variable n s'incrementa en una unitat tant si l'operador ++ s'aplica a la seva esquerra com si s'aplica a la seva dreta.

```
unsigned int n;
...
n = 5; n++;
printf ("n = %u",n); /* Visualitza n = 6 */
n = 5; ++n;
printf ("n = %u",n); /* Visualitza n = 6 */
```

Doncs bé, no sempre és equivalent la utilització d'aquests operadors.

Aquests operadors es poden utilitzar sobre la variable que permet l'accés a les caselles d'una taula, de manera que podem trobar-nos amb situacions similars a la següent:

```
unsigned int n;
int t[20];
...
n = 5;
t[n++] = 10; /* Posa 10 a la casella t[5] i augmenta n en una unitat (6) */
n = 5;
t[++n] = 10; /* Augmenta n en una unitat (6) i posa 10 a la casella t[6] */
```

1.4. Algorismes bàsics en taules unidimensionals

En primer lloc, cal tenir present que una taula té un nom i una capacitat màxima, que és el número que s'ha utilitzat en el moment de la declaració. Però també hi ha una altra dada que s'ha de tenir present, i no és altra que la quantitat de dades emmagatzemades en un determinat moment dins la taula.

És a dir, davant una declaració del tipus:

```
t: taula[MAX] de enter;
```

ja sabem que `t` és una taula per a emmagatzemar enters, de capacitat `MAX`. Ara bé, quants enters té emmagatzemats en un determinat moment? Aquesta informació, bàsica per a qualsevol procés de programació, no forma part de la pròpia taula. Per tant, caldrà declarar una dada suplementària de tipus natural per a enregistrar-hi la quantitat d'elements guardats dins la taula:

```
qt: natural;
```

És responsabilitat del programador mantenir la coherència entre les dades emmagatzemades a `t` i el valor enregistrat a `qt`. Observeu que el nom `qt` no s'ha escollit de manera aleatòria. És aconsellable anomenar les variables que tenen alguna cosa a veure amb noms que ajudin a identificar-ne el significat.

D'altra banda, de poc ens servirà saber que la taula t té una capacitat `MAX` i que en un determinat moment està guardant `qt` valors si aquests no són consecutius. És a dir, la gràcia de disposar de `qt` està en el fet que aquests `qt` valors són a les primeres `qt` posicions de la taula, ja que llavors, indirectament, també sabem que la primera posició lliure és a la posició `qt`.

Observem, amb l'ajuda de la figura 4, el que estem comentant.

Figura 4. Enumeració de les posicions d'una taula de MAX posicions.

	0	1	2	3			MAX-2	MAX-1
t	valor	valor	valor	valor	valor	valor	valor	valor

Si la taula té `MAX` caselles, aquestes estan numerades des de `0` fins a `MAX-1`, ja que hem decidit que sempre consideraríem el començament de la numeració per zero. D'igual manera, si la variable `qt` conté la quantitat de posicions plenes i aquestes estan concentrades a partir de l'inici de la taula, vol dir que les posicions plenes van des de la posició `0`

fins a `qt-1` i, per tant, la primera casella disponible és la que es troba a la posició `qt`, sempre que `qt` sigui menor que `MAX`.

En els apartats següents veurem alguns algorismes bàsics de tractament de taules unidimensionals.

En el llenguatge C no hi ha cap tipus d'operadors sobre taules. És a dir, no es pot copiar (assignar) una taula en una altra, no es pot omplir una taula amb una sola assignació (excepte quan s'omple en la inicialització), no es pot operar aritmèticament entre taules, etc. En pseudocodi considerarem la mateixa situació.

1.4.1. Recorregut total

Suposem que tenim les declaracions següents:

```
t: taula[MAX] de <T>;
   /* suposem que <T> és un tipus existent */
qt: natural;
   /* variable que conté el nombre de dades guardades a t */
```

i que volem fer un recorregut per tota la taula per tal de fer un tractament amb cadascuna de les dades que conté en un determinat moment.

Per a poder fer això necessitem una variable entera que ens permeti anar senyalant les diferents caselles de la taula per tal de fer el tractament que calgui, des de la casella de la posició `0` fins a la casella de la posició `qt-1`. I com que es tracta d'anar passant de casella en casella fent el mateix en totes, estem davant un procés iteratiu, per la qual cosa hem d'utilitzar alguna instrucció repetitiva.

En aquest cas, la millor elecció és la instrucció `per`, ja que es coneix el nombre de vegades que cal repetir el procés. Així doncs:

```
var i: natural; fivar
...
per i = 0 fins qt-1 fer
   tractament de la casella t[i];
fiper
...
```

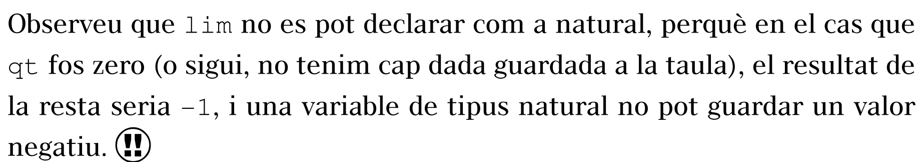
La instrucció repetitiva `per` inicialitza la variable `i` amb el valor zero (primera posició de la taula) i comprova si la i és menor o igual a `qt-1`. Per a fer això, l'ordinador avalua l'expressió `qt-1`, és a dir, n'efectua la resta. En el cas que es compleixi que `i` sigui menor al resultat de la resta es passa a executar el contingut del bucle, que en aquest cas consisteix en el tractament de la casella `t[i]`. Posteriorment s'incrementa la `i` i es

©Isidre Guixà i Miranda

torna a comprovar si la `i` és menor o igual a `qt-1` i, és clar, es torna a executar la resta. Conclusió: la resta s'està repetint a cada iteració, la qual cosa és millorable si la calculem prèviament i n'enregistrem el resultat en una variable. O sigui:

```
var   i: natural;
      lim: enter;
fivar
...
lim = qt - 1;
per i = 0 fins lim fer
   tractament de la casella t[i];
fiper
...
```

Observeu que `lim` no es pot declarar com a natural, perquè en el cas que `qt` fos zero (o sigui, no tenim cap dada guardada a la taula), el resultat de la resta seria `-1`, i una variable de tipus natural no pot guardar un valor negatiu.



Cal vigilar l'execució de càlculs repetitius en les condicions lògiques de les instruccions repetitives. A vegades els càlculs s'han de repetir a cada iteració perquè el contingut de les dades que cal operar ha canviat. Però quan els càlculs siguin constants, cal efectuar-los abans d'iniciar la instrucció repetitiva emmagatzemant-ne el valor en una variable que serà utilitzada a l'avaluació de la condició lògica a cada iteració.

La traducció corresponent del codi anterior en llenguatge C seria la següent:

```
unsigned int i;
...
for (i = 0; i < qt; i++)
   {  tractament de la casella t[i];  }
...
```

En el cas del llenguatge C no cal fer servir cap variable `lim`, perquè en lloc de posar:

```
i <= qt - 1
```

podem escriure l'expressió equivalent:

```
i < qt
```

Podem efectuar també el recorregut amb la instrucció iterativa `mentre`:

```
var i: natural; fivar
...
```

```
i = 0;
mentre i < qt fer
   tractament de la casella t[i];
   i++;
fimentre
...
```

De manera anàloga al darrer comentari que hem fet per a la codificació en C, no es necessita la variable entera `lim`.

La traducció corresponent en llenguatge C seria la següent:

```
unsigned int i;
...
i = 0;
while (i < qt)
{
   tractament de la casella t[i];
   i++;
}
...
```

En aquest cas no és utilitzable la instrucció iterativa `fer...mentre` o `repetir...fins`, atès que no hi ha cap seguretat respecte al fet que sempre s'hagi d'executar una vegada, com a mínim, el bucle. Penseu en el cas que `qt` sigui zero per tractar-se d'una taula buida.

1.4.2. Afegir pel final

Suposem que tenim les declaracions següents:

```
t: taula[MAX] de <T>;
   /* suposem que <T> és un tipus existent */
qt: natural;
   /* variable que conté el nombre de dades guardades a t */
```

i que volem afegir una dada <v> de tipus <T> al final de la taula.

Quan es demana afegir pel final es vol dir que cal afegir després del darrer valor que hi ha en aquest moment en la taula. En primer lloc caldrà comprovar que la taula disposa de lloc per a un element nou (`qt < MAX`) i en cas afirmatiu, assignarem el valor nou a la posició `qt` i incrementarem posteriorment el valor de `qt` en una unitat. És a dir:

```
si qt < MAX
   llavors t[qt] = <v>; /* utilitzant l'assignació en el tipus <T> */
           qt = qt + 1;
   sinó escriure ("No hi ha espai disponible");
fisi
```

1.4.3. Recerca d'un valor

Una altra situació típica que es presenta en la gestió de taules és la d'haver de cercar un valor en una taula. És obvi que ens podem trobar amb les tres situacions següents:

1) La dada cercada no és a la taula.
2) La dada cercada és una vegada a la taula.
3) La dada cercada és més d'una vegada a la taula.

La recerca consisteix a fer un recorregut per la taula a partir de l'inici, però no sempre s'ha d'arribar al final. Cal anar passant de posició en posició fins a trobar per primer cop la dada o fins a arribar al final de la taula sense haver-la trobat (aneu amb compte, ja que pot ser a l'última posició de la taula). En cas d'haver-la trobat, cal tenir clar com s'ha d'actuar davant la possibilitat que hi hagi valors repetits dins la taula. Si sabem que la taula no pot tenir valors repetits, no cal continuar la recerca. Però si en pot tenir, cal saber el que hem de fer: ens interessa trobar-los tots o havent-ne trobat un ja en tenim prou? En cas d'haver-los de trobar tots segur que el recorregut és total.

Així, suposem que tenim les declaracions següents

```
t: taula[MAX] de <T>;
   /* suposem que <T> és un tipus existent */
qt: natural;
   /* variable que conté el nombre de dades guardades a t */
```

i que volem efectuar la recerca d'un valor <v> de tipus <T>.

L'esquema seria:

```
var i : natural; /* Variable per moure'ns per la taula */
fivar
...
i = 0;
mentre i < qt i t[i] != <v> fer i++; fimentre
si i==qt (***)
   llavors /* Tractament conforme NO s'ha trobat el valor cercat */
   sinó /* Tractament conforme s'ha trobat el valor cercat */
fisi
```

La condició marcada amb (***) és fonamental. No és correcte canviar-la per `t[i] == <v>` doncs si no s'ha trobat, la variable `i` arriba a valer `qt` i no podem intentar accedir a una cel·la de la posició `qt` doncs aquesta variable ens indica que les posicions vàlides van des de la posició `0` fins la posició `qt-1`. (!!)

En llenguatge C seria:

```
int i; /* Variable per moure'ns per la taula */
...
i = 0;
while (i < qt && t[i] != <v>) i++;
if (i==qt)
   /* Tractament conforme NO s'ha trobat el valor cercat */
else
   /* Tractament conforme s'ha trobat el valor cercat */
fisi
```

1.4.4. Exemples introductoris a la manipulació de taules

Vegem alguns exemples introductoris al tractament de taules que ens serviran per a practicar l'accés adequat a les diferents posicions de les taules i l'aplicació adequada dels algorismes bàsics sobre taules: recorregut, recerca i afegir pel final.

Exemple 1 d'utilització de taula per a emmagatzemar comptadors

Volem efectuar un programa que demani a l'usuari un text que acabi en $ i que compti quantes vegades hi apareix cada vocal.

Anàlisi funcional

Tothom coneix el significat del concepte vocal. Ara bé, cal tenir present que informàticament parlant són diferents les majúscules de les minúscules i amb signes d'accentuació o sense. Considerarem les possibilitats d'accentuació de vocals en llengua catalana.

Anàlisi orgànica

De manera similar a quan volem comptar quants caràcters `A` hi ha en un text acabat en $, ens cal fer un recorregut per tot el text fins a arribar a la marca de fi, de manera que per cada caràcter cal esbrinar si es tracta d'una vocal i, en cas afirmatiu, comptar-la adequadament.

Per comptar quantes vegades apareix cada vocal necessitarem cinc variables de tipus acumulador. És factible tenir cinc variables diferents (per exemple, `qA`, `qE`, `qI`, `qO`, `qU`), però també és factible treballar amb una taula de cinc caselles on cada casella faci el paper d'acumulador d'una de les vocals.

Algorisme en pseudocodi:

```
programa comptar_vocals és
   var  q: taula[5] de natural;
```

```
        i: natural; c: caràcter;
   fivar
   netejar_pantalla;
   per i=0 fins 4 fer q[i] = 0; fiper /*inicialitzar la taula*/
   /* Suposarem que:
      Casella 0 : Vocals 'A', 'a', 'À', 'à', ...
      Casella 1: Vocals 'E', 'e', 'È', 'è', ...
   i així successivament */
   escriure ("Introdueixi text acabat en $');
   llegir (c);
   mentre c != '$' fer
      encasde c
         ser 'A', 'a', 'À', 'à' : q[0] = q[0]+1;
         ser 'E', 'e', 'È', 'è', 'É', 'e' : q[1] = q[1]+1;
         ser 'I', 'i', 'Í', 'í', 'Ï', 'ï' : q[2] = q[2]+1;
         ser 'O', 'o', 'Ò', 'ò', 'Ó', 'ó' : q[3] = q[3]+1;
         ser 'U', 'u', 'Ú', 'ú', 'Ü', 'ü' : q[4] = q[4]+1;
      fiencasde
      llegir (c);
   fimentre
   /* Ja hem recorregut tot el text i hem comptat les vocals */
   /* Farem un recorregut total per la taula per tal de
      visualitzar els resultats. No ho podem fer amb una
      estructura repetitiva perquè per cada casella haurem
      d'informar de la corresponent vocal comptada */
   escriure ("Vocals trobades en el text:"); saltar_línia;
   escriure ("Vocals A: ", q[0]); saltar_línia;
   escriure ("Vocals E: ", q[1]); saltar_línia;
   escriure ("Vocals I: ", q[2]); saltar_línia;
   escriure ("Vocals O: ", q[3]); saltar_línia;
   escriure ("Vocals U: ", q[4]); saltar_línia;
fiprograma
```

Codificació en llenguatge C:

Trobareu l'arxiu `u3n1p01.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u3n1p01.c */

#include <stdio.h>
#include <conio.h>

void main(void)
{
   unsigned int i;
   char c;
   unsigned int q[5];
   clrscr();
   printf ("Introdueixi un text.\n");
   printf ("L'aparició del símbol $ indicarà la fi.\n");
   for (i=0; i<5; i++) q[i] = 0;
   scanf("%c",&c);
   while (c != '$')
   {  switch (c)
      {  case 'A': case 'a': case 'À': case 'à':   q[0]++; break;
         case 'E': case 'e': case 'É': case 'é': case 'È': case 'è': q[1]++; break;
         case 'I': case 'i': case 'Í': case 'í': case 'Ï': case 'ï': q[2]++; break;
         case 'O': case 'o': case 'Ó': case 'ó': case 'Ò': case 'ò': q[3]++; break;
         case 'U': case 'u': case 'Ú': case 'ú': case 'Ü': case 'ü': q[4]++; break;
      }
```

```
      scanf ("%c",&c);
   }
   printf ("El text conté les vocals següents:\n\n");
   printf ("Vocals A : %u\n",q[0]);
   printf ("Vocals E : %u\n",q[1]);
   printf ("Vocals I : %u\n",q[2]);
   printf ("Vocals O : %u\n",q[3]);
   printf ("Vocals U : %u\n",q[4]);
}
```

Exemple 2 d'utilització de taula per a emmagatzemar comptadors

Volem efectuar un programa que demani a l'usuari un text acabat en $ i que compti quantes vegades hi apareix cada caràcter.

Anàlisi funcional

Cal tenir present que l'usuari pot introduir fins a 256 caràcters diferents.

Anàlisi orgànica

En aquest cas ens veiem obligats a treballar amb una taula de 256 posicions. Seria impensable treballar amb 256 variables de tipus acumulador. D'altra banda, recordeu que la immensa majoria de llenguatges disposen de funcions predefinides. En concret us convé conèixer la possibilitat de, donat un caràcter, saber el lloc que ocupa dins la taula de codi ASCII, i viceversa, és a dir, donada una posició de la taula, saber quin és el caràcter que l'ocupa. Com que la majoria de llenguatges disposen d'aquesta opció, suposarem que també existeixen en pseudocodi. Així, suposarem que en pseudocodi disposem de les dues funcions següents:

```
funció car_asc (car: caràcter) retorna natural;
/* donat un caràcter <car> ens dóna el seu codi ASCII */

funció asc_car (núm: natural) retorna caràcter;
/* donat un codi ASCII <num> ens dóna el seu caràcter */
```

Fem servir funcions quan encara no les hem treballat. Bé, de moment l'única cosa que heu de saber és que quan tingueu un caràcter `X` i vulgueu saber quin n'és el codi ASCII, haureu d'escriure `car_asc(X)` i on hi hagi escrita aquesta expressió cal pensar que l'ordinador us ho substituirà pel valor que retorna, és a dir, pel codi ASCII. Passa el mateix si teniu un valor natural (suposadament, entre 0 i 255) i voleu conèixer el caràcter corresponent. Què passa, en el darrer cas, si l'usuari introdueix un valor <num> que no és dins els límits permesos? En pseudocodi prendrem aquesta incidència com un error de programació que provocaria l'avortament de l'execució del programa.

L'algorisme que s'ha de dissenyar consisteix a utilitzar com a acumulador de cada caràcter la posició de la taula que es correspon amb el codi ASCII del caràcter. És a dir, els caràcters `A` els comptarem en la casella numerada amb `65`, ja que aquest és el codi ASCII del caràcter `A`.

Algorisme en pseudocodi:

```
programa comptar_caràcters és
   var  q: taula[256] de natural;
        c: caràcter; i: natural;
   fivar
   netejar_pantalla;
   per i=0 fins 255 fer q[i] = 0; fiper /* netejar la taula */
   escriure ("Introdueixi text acabat en $');
   llegir (c);
   mentre c != '$' fer
      i = car_asc(c); /* i conté el codi ASCII del caràcter c */
      q[i] = q[i] + 1;
      llegir (c);
   fimentre
   /* Ja hem recorregut tot el text i hem comptat els caràcters*/
   /* Farem un recorregut total per la taula per tal de visualitzar els resultats. Només
      mostrarem els valors per a aquells caràcters que han aparegut alguna vegada dins el
      text */
   escriure ("Caràcters trobats en el text:"); saltar_línia;
   per i=0 fins 255 fer
      si q[i] > 0 llavors
         escriure ("Caràcter ", asc_car(i), " : ", q[i], " vegades.");
         saltar_línia;
      fisi
   fiper
fiprograma
```

Hem fet servir, dins el `mentre`, les instruccions següents:

```
i = car_asc(c);
q[i] = q[i] + 1;
```

S'hauria pogut estalviar una instrucció i posar, directament, el següent:

```
q[car_asc(c)] = q[car_asc(c)] + 1;
```

L'observació ens serveix per a veure que el resultat d'una funció, si és un valor natural, es pot utilitzar per a accedir a la posició d'una taula (sempre que el valor correspongui a una de les posicions que hi ha a la taula). Ara bé, també hem d'observar que és més eficient el codi amb dues instruccions que amb una de sola, ja que quan fem servir una única instrucció, s'executa dues vegades la crida a la funció `car_asc`, és a dir, repetim dues vegades els mateixos càlculs.

> Ja s'ha dit abans, però cal remarcar-ho: eviteu de repetir innecessàriament les crides a una funció, ja que el grau de càlcul d'aquesta pot ser important.

Codificació en llenguatge C:

Per tal d'efectuar la codificació en C ens és necessari conèixer quines funcions corresponen a `asc_car` i `car_asc`. Doncs bé, el llenguatge C no disposa d'aquestes funcions perquè no les necessita. Observeu el fragment següent de codi en C:

```
char c = 'A';
...
printf ("Caràcter c com a char és %c\n",c);
printf ("Caràcter c com a int és %d\n",c);
printf ("Caràcter c com a unsigned int és %u\n",c);
...
```

Per pantalla obtindríem:

```
Caràcter c com a char és A
Caràcter c com a int és 65
Caràcter c com a unsigned int és 65
```

És a dir, de manera automàtica, el llenguatge C sap interpretar un `char` com un `int` o `unsigned int`. En general, el llenguatge C interpreta una dada de qualsevol tipus com una d'un altre tipus de manera automàtica. S'ha de tenir en compte, però, els perills que això suposa. Recordeu el problema en el llenguatge C referent al comportament cíclic dins el rang de valors possibles en un determinat tipus de dada quan el valor que s'ha d'emmagatzemar és fora dels límits del rang. El mateix passa a continuació:

```
short n = 33449;
char c = 33449;
...
printf ("Enter 33449 com a short és %d\n",n);
printf ("Enter 33449 com a char és %c\n",c);
...
```

Per pantalla (en plataforma de 32 bits) obtindríem el següent:

```
Enter 33449 com a short és –32087
Enter 33449 com a char és ©
```

Això és possible perquè:

- el tipus `short` té el rang de valors entre –32.768 i 32.767, i com que 33.449 no està dins l'interval, es passa per la banda dreta i els valors tenen un comportament cíclic, li correspon el valor –32.087, ja que 33.449 passa 682 unitats de 32.767, que es compten per la banda esquerra a partir de –32.768 i apareix el valor –32.087;

- el tipus `char` té el rang de valors (com a número) entre –128 i 127, i com que 33.449 no està dins l'interval, es passa per la banda dreta i els

valors tenen un comportament cíclic, li correspon el caràcter © que està en el lloc 169, ja que 33.449 correspon a 130 vegades el valor 256 (grandària de l'interval) i encara manquen 169 unitats (com a número seria –87, ja que 169 passa 42 unitats de 127, que es converteixen en –87 si comencem a comptar a partir de –128).

En els exemples anteriors hem observat la interpretació diferent que el llenguatge C pot fer d'una dada en el moment de presentar-la en pantalla. Però el llenguatge C fa servir el mateix raonament per a assignar una dada d'un cert tipus a una altra dada d'un cert tipus. Així, si tenim una dada `char` i en volem saber el codi ASCII, només cal interpretar-la com a valor numèric. Cal tenir present, però, que el tipus char té valors numèrics entre -128 i 127. Si ens interessa que els valors siguin entre 0 i 255 caldrà utilitzar el tipus `unsigned char`.

Com que nosaltres desitgem incrementar el valor d'un comptador que es troba en una posició (de 0 a 255) d'una taula, la qual correspon al codi ASCII d'un caràcter determinat, només cal que tinguem present que hem de tractar el caràcter com a `unsigned char` i, així, directament, sense necessitat de tenir les funcions `asc_car` i `car_asc`, podem tenir la posició a partir del caràcter i el caràcter a partir de la posició.

Així doncs:

```
/* u3n1p02.c */

#include <stdio.h>
#include <conio.h>

void main(void)
{
   unsigned int i;
   unsigned char c;
   unsigned int q[256];
   clrscr();
   printf ("Introdueixi un text.\n");
   printf ("L'aparició del símbol $ indicarà la fi.\n");
   for (i=0; i<256; i++) q[i] = 0;
   scanf("%c",&c);
   while (c != '$')
   {
      q[c]++;
      scanf ("%c",&c);
   }
   printf ("El text conté els caràcters següents:\n\n");
   for (i=0; i<256; i++)
   {  if (q[i]) printf ("Caràcter %c : %u\n",i,q[i]); }
}
```

Trobareu l'arxiu `u3n1p02.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Exemple d'afegir pel final en una taula si no hi ha la informació

Volem fer un programa que permeti a l'usuari anar introduint, per teclat, valors enters fins que es cansi. Quan acabi, el programa ha d'informar per pantalla sobre el nombre de vegades que s'ha introduït cada número.

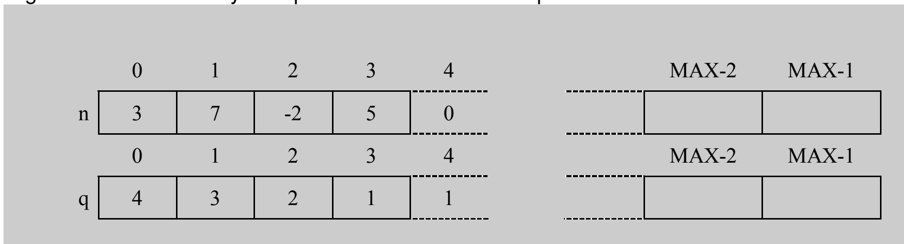
Anàlisi funcional

No cal cap coneixement especial per a resoldre aquest algorisme. Hem de tenir en compte que pot donar-se el cas que l'usuari intenti introduir una quantitat de nombres diferents que superi la capacitat prevista pel programa. En tal cas optem per avisar l'usuari sobre aquest fet en el moment en què es produeixi, i continuarem la comptabilització, si l'usuari ho desitja.

Anàlisi orgànica

Simplement s'ha de tenir en compte que necessitem una taula per a enregistrar els valors introduïts per l'usuari i una altra taula per a introduir-hi les vegades que ha sortit cada valor. És a dir, si l'usuari introdueix els valors 3, 7, –2, 3, –2, 5, 0, 7, 3, 7, 3 hem de tenir una situació en dues taules com mostra la figura 5.

Figura 5. Taules dissenyades per la resolució de l'exemple 3



La taula `n` emmagatzema els números que l'usuari hi ha introduït i la taula `q` s'utilitza com a taula d'acumuladors per als corresponents números.

Necessitarem també una variable per a saber fins a quina posició de les taules hi ha valors introduïts. Aquesta variable ens servirà també per a saber si hi ha espai disponible per a emmagatzemar un enter introduït per l'usuari que encara no hagi sortit mai.

Algorisme en pseudocodi:

```
programa comptar_enters és
   const MAX = 1000;
   /* Farem servir aquesta constant per a definir la capacitat de les taules que
   emmagatzemaran els valors i els acumuladors corresponents; en cas que es necessiti
   modificar el programa per donar més o menys capacitat a les taules, només caldrà
   modificar aquesta constant */
```

```
   ficonst
   var  n: taula [MAX] de enter;     /* diferents valors introduïts per l'usuari */
        q: taula [MAX] de natural;   /* acumuladors de cada valor */
        qn: natural;  /* variable per a saber quants valors hi ha */
        k: natural;  /* variable auxiliar per a moure'ns per la taula */
        c: caràcter;  /* variable auxiliar per a llegir caràcter de teclat */
        e: enter;  /* variable auxiliar per a llegir enter de teclat */
   fivar
   netejar_pantalla;
   qn = 0;  /* inicialització de la quantitat de valors entrats a la taula */
   /* no cal inicialitzar les taules; inicialitzarem una posició quan calgui */
   fer
      escriure ("Introdueixi número:"); llegir (e);
      /* cal cercar si el valor e ja és a la taula; en cal un recorregut parcial */
      k = 0;
      mentre k < qn i n[k] != e fer k = k+1; fimentre
      /* MOLT IMPORTANT el control sobre el valor de la k de manera que no surti de les
         posicions que s'han de cercar dins la taula (posicions 0..qn-1 ja que a qn hi
         tenim la quantitat de valors enters introduïts). Fixeu-vos que ha estat
         inicialitzat amb el valor 0 per a situar-se a l'inici de la taula i que la
         condició lògica correspon a la pregunta "estem en una posició de la taula i el
         valor que hi ha emmagatzemat és diferent al valor que estem cercant?". En cas
         afirmatiu passem a la posició següent de la taula (incrementant k en una unitat).
         És clar que en algun moment se surt del mentre. Tenim dues possibilitats: quan
         hem recorregut tota la taula sense haver trobat el valor e o quan hem trobat el
         valor e. I cal distingir els dos casos perquè l'actuació del programa ha de ser
         diferent. Si no s'ha trobat, cal intentar afegir-lo a la taula (si hi ha
         espai...). Si s'ha trobat, només cal incrementar el corresponent acumulador. */
      si k < qn            /* per esbrinar el motiu de sortida del mentre */ [***]
         llavors              /* l'element e s'ha trobat a la posició k */
            q[k] = q[k] + 1;  /* només cal augmentar l'acumulador */
         sinó                 /* l'element e no s'ha trobat a la taula */
            si qn == MAX    /* per esbrinar si hi ha espai */
               llavors         /* la taula ja és plena */
                  escriure ("No hi ha espai per a aquest valor.");
                  saltar_línia;
               sinó
                  /* la taula té espai; posem el valor a la posició qn, que és la primera
                     posició lliure */
                  n[qn] = e ; q[qn] = 1; qn = qn+1;
                  /* iniciem el corresponent acumulador amb 1 i incrementem qn */
            fisi
      fisi
      escriure ("Vol continuar (S/N)?");
      fer llegir (c);
      mentre c != 'S' i c != 's' i c != 'N' i c != 'n'
   mentre c == 'S' o c == 's'
   si qn == 0
      llavors
         escriure ("No s'ha introduït cap valor.");
         saltar_línia;
      sinó
         netejar_pantalla;
         per k = 0  fins qn-1 fer
            escriure ("Valor ", n[k], " : ", q[k]);
            saltar_línia;
         fiper
   fisi
fiprograma
```

Observeu la marca [***] dins l'algorisme. És molt important entendre com està formulada la condició per a esbrinar el motiu pel qual s'ha sortit del `mentre` anterior. Ens interessa controlar si l'element s'ha trobat o no. És molt corrent entre els estudiants de programació formular la condició corresponent de la manera següent:

```
n[k] == e
```

és a dir, quan se surt del `mentre`, com que `k` és la variable utilitzada per recórrer la taula, ens preguntem si a la darrera posició apuntada per `k` hi ha l'element cercat.

Aquest és un error de programació típic. És evident que si l'element `e` s'ha trobat, aquesta condició serà avaluada com a certa. Però, i si l'element `e` no és a la taula? En tal cas k arribarà a valer qn i aquí ens trobem amb l'error, ja que estem preguntant si a la posició `qn` hi ha l'element `e` i recordem que les posicions on hem de mirar són les `qn` primeres, és a dir, des de la `0` fins a la `qn-1`. Tingueu present el que pot succeir davant aquest error de programació:

– Si `qn < MAX`, és a dir, si la taula encara no està totalment plena, preguntem per una posició de la taula que no correspon a les posicions gestionades fins al moment. Què hi ha en aquella posició? Informàticament parlant, porqueria! Vés a saber què hi deu haver! Depèn de la seqüència de bits que hi hagi. Si esteu pensant que això passa perquè no hem inicialitzat totes les posicions de la taula **n** amb algun valor, esteu equivocats, ja que, amb quin valor enter inicialitzaríeu (és una taula d'enters) que no pugui entrar l'usuari com a valor enter que cal tractar? Per tant, podria ser que la condició fos avaluada com a certa o com a falsa, depenent de la casuística del moment.

– Si `qn == MAX`, és a dir, si la taula ja està totalment plena, preguntem per una posició que no és a la taula (les posicions de la taula són de `0` a `qn-1`), amb la qual cosa estem provocant un greu error en temps d'execució. En la immensa majoria de llenguatges, aquest error provoca l'avortament de l'execució del programa, amb un error que ens indica que hem sortit fora dels límits de la taula. El llenguatge C, però, no es queixa gens. Ja hem comentat moltes vegades que "Mister C" té per norma obeir a cegues el programador. Què fa en aquest cas? Considera la teòrica posició següent a la taula i compara el valor que hi hagi (sense cap sentit per a nosaltres, evidentment) amb el valor `e`.

Fixeu-vos en els tipus de tractament utilitzats en aquest programa:

• Recerca d'un valor en una taula (hi pot ser o no).

- Afegir un valor pel final.
- Recórrer tota la taula (en el moment de donar resultats).

Codificació en llenguatge C:

La codificació en C consisteix en la simple traducció del pseudocodi, tenint en compte que hem de controlar la introducció de les dades per teclat amb la funció `neteja_stdin()` que tenim implementada de forma adequada a la plataforma.

Trobareu l'arxiu `u3n1p03.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u3n1p03.c */

#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* per tenir compatibilitat entre plataformes */

#define MAX  1000

void main(void)
{
   int n[MAX];
   unsigned int q[MAX];
   unsigned int qn;
   unsigned int k;
   int e;
   char c;
   clrscr();
   qn = 0;
   do
   {  printf ("Introdueixi un enter.\n");
      do { k = scanf ("%d",&e); neteja_stdin(); }
      while (k == 0);
      for (k=0; k<qn && n[k] != e; k++);
      if (k < qn) q[k]++;
      else
         if (qn == MAX)  printf ("No hi ha espai per a emmagatzemar el valor\n");
         else {  n[qn] = e; q[qn] = 1; qn++; }
      printf ("Vol continuar (S/N)?");
      do { k = scanf ("%c",&c); neteja_stdin(); }
      while (k==0||(c!='S' && c!='s' && c!='N' && c!='n'));
   } while (c=='S' || c=='s');
   if (qn == 0) printf ("No s'ha introduït cap valor\n");
   else {  clrscr();
           for (k = 0; k < qn ; k++)
              printf ("Valor %d: %u\n",n[k],q[k]);
        }
}
```

1.4.5. Recerca d'un valor amb tècnica de sentinella

Considerem l'algorisme per a afegir un element al final d'una taula en cas que l'element no sigui a la taula, consistent en efectuar en primer

lloc la corresponent recerca i, afegir-lo a la taula en cas de no trobar-lo. El procés iteratiu on es controla la recerca és:

```
n-> taula on cal cercar l'element
qn -> quantitat de posicions on s'ha d'efectuar la recerca
e-> element que es vol cercar
k-> variable índex que ens permet recórrer la taula
...
k = 0;
mentre k < qn i n[k] != e fer k = k+1; fimentre
si k < qn
   llavors <tractament per haver-lo trobat a la posició k>
   sinó <tractament per no haver-lo trobat>
fisi
```

A l'exemple `u3n1p03` s'implementa l'algorisme per afegir un element al final d'una taula quan l'element no s'hi troba.

La condició del `mentre` ha de controlar a cada iteració si encara ens trobem dins la taula. Quan s'acaba el procés repetitiu trobem una altra condició per esbrinar quin ha estat el motiu de sortida i ja hem comentat en el seu moment la gran importància que té formular aquesta condició en termes de si `k < qn` en lloc de `q[k] == e`.

Introduirem ara la tècnica de sentinella per a buscar un valor, la qual ens permetrà condicions més senzilles. Ara bé, aquesta tècnica té un preu: cal destinar una posició de la taula a la gestió d'aquesta tècnica; per tant, si a la taula on hem de fer la recerca necessitem `MAX` posicions per a emmagatzemar valors, ens caldrà declarar la taula de `MAX+1` posicions, garantint que sempre hi haurà una darrera posició lliure per a la tècnica de sentinella.

La tècnica consisteix en el següent: situem una còpia del valor que es vol cercar en la posició posterior a la darrera casella ocupada. Pel que hem dit en el paràgraf anterior, aquesta posició sempre està disponible. Posteriorment, comencem la recerca pel principi de la taula. No ens hem de preocupar, a cada iteració, respecte a si estem o no dins la taula. Només ens hem de preocupar de trobar l'element, que segur que trobarem perquè l'hem posat a dins. El procés iteratiu s'acaba quan trobem l'element. En tal cas, si la posició on s'ha trobat és la posició on l'havíem col·locat, indica que l'element cercat no era a la taula. En cas contrari indica que sí que hi era i, a més, tenim la posició on s'havia trobat.

Per què s'anomena "tècnica del sentinella". És obvi, no? L'element que s'ha de cercar, quan el situem a la posició posterior als valors existents, es converteix en un sentinella de la recerca, de manera que aquest mateix atura la recerca si no hi ha cap element que l'aturi abans.

Aquest mètode és útil per a recerques i, a més, és especialment útil quan l'element que cal cercar s'ha d'afegir a la taula en el cas que no hi sigui,

ja que mitja feina ja està feta (està situat a la posició on li pertocarà estar).

Exemple d'utilització de la tècnica de sentinella

Considerem el programa per controlar l'entrada d'un conjunt de números enters per part de l'usuari comptant quins números introdueix i la quantitat de cadascun.

Ja sabem dissenyar-ne una solució utilitzant dues taules: una per enregistrar els valors enters introduïts i una altra per a comptadors dels corresponents valors enters. És clar, que podem dissenyar-ne una versió aplicant la tècnica de sentinella al fet de cercar i afegir pel final a la primera de les taules, si no hi és, el valor enter introduït. Passem directament a l'algorisme en pseudocodi.

```
programa comptar_enters és
   const MAX = 1000;
   ficonst
   var  n: taula [MAX+1] de enter;
        /* Observem que declarem la taula amb una posició més per a la gestió de la
           tècnica de sentinella */
        q: taula [MAX] de natural;
        /* En aquest cas no destinem una posició a la gestió de la tècnica de sentinella
           perquè sobre aquesta taula no cal fer cap recerca */
        qn, k: natural;
        c: caràcter;
        e: enter;
   fivar
   netejar_pantalla;
   qn = 0;
   fer
      escriure ("Introdueixi número:"); llegir (e);
      k = 0; n[qn] = e;
      /* Com que la taula té les primeres qn posicions plenes (0..qn-1), situem l'element
         que es vol cercar com a sentinella a la posició qn, que segur que existeix */
      mentre n[k] != e fer k = k+1; fimentre
      /* Observeu que no es controla la posició de la taula */
      si k < qn                   /* no s'ha arribat a la posició del sentinella? */
         llavors                  /* l'element e existeix a la posició k */
            q[k] = q[k] + 1       /* només cal augmentar l'acumulador */
         sinó                     /* l'element e no s'ha trobat a la taula */
            si qn == MAX          /* per esbrinar si hi ha espai */
            /* comparem amb MAX i no amb MAX+1 perquè hem de garantir que sempre hi hagi
               la posició disponible per al sentinella */
               llavors            /* la taula ja és plena */
                  escriure ("No hi ha espai per a aquest valor.");
                  saltar_línia;
               sinó               /* hi ha espai; ja el tenim posat a la posició qn */
                  q[qn] = 1; qn = qn+1;   /* iniciem l'acumulador amb 1 i incrementem qn */
            fisi
      fisi
      escriure ("Vol continuar (S/N)?");
      fer llegir (c);
```

```
      mentre c!='S' i c!='s' i c!='N' i c!='n'
   mentre c == 'S' o c == 's'
   si qn == 0
      llavors
         escriure ("No s'ha introduït cap valor.");
         saltar_línia;
      sinó
         netejar_pantalla;
         per k = 0  fins qn-1 fer
            escriure ("Valor ", n[k], " : ", q[k]);
            saltar_línia;
         fiper
   fisi
fiprograma
```

Codificació en llenguatge C:

Trobareu l'arxiu `u3n1p04.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u3n1p04.c */

#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* per compatibilitat entre plataformes */

#define MAX  1000

void main(void)
{
   int n[MAX+1];
   unsigned int q[MAX];
   unsigned int qn;
   unsigned int k;
   int e;
   char c;
   clrscr();
   qn = 0;
   do
   {  printf ("Introdueixi un enter.\n");
      do { k = scanf ("%d",&e); neteja_stdin(); }
      while (k == 0);
      for (k=0, n[qn]=e; n[k] != e; k++);
      if (k < qn) q[k]++;
      else
         if (qn == MAX)  printf ("No hi ha espai per a emmagatzemar el valor\n");
         else {  q[qn] = 1; qn++; }
      printf ("Vol continuar (S/N)?");
      do { k = scanf ("%c",&c); neteja_stdin(); }
      while (k==0||(c!='S' && c!='s' && c!='N' && c!='n'));
   } while (c=='S' || c=='s');
   if (qn == 0) printf ("No s'ha introduït cap valor\n");
   else {  clrscr();
           for (k = 0; k < qn ; k++)
              printf ("Valor %d: %u\n",n[k],q[k]);
        }
}
```

1.4.6. Exemples d'aprofundiment en la manipulació de taules

Vegem alguns exemples per aprofundir en el tractament de taules.

Exemple 1 d'aprofundiment en el tractament de taules

Volem dissenyar un programa que permeti comptar quantes vegades apareix una tros de text (n caràcters) en un text acabat en $.

Plantegem-nos, primer, un programa adient per quan la partícula té 2 caràcters, per exemple, la partícula `LA`. La resolució d'aquesta situació particular és molt simple i ni tan sols necessitem de cap taula per a resoldre-ho. Vegem-ho.

Anàlisi funcional

Només volem fer notar que es tracta de cercar la partícula `LA`, que pot estar com a article o formant part d'una altra paraula.

Anàlisi orgànica

Resoldrem el problema analitzant els caràcters que teclegi l'usuari, de manera que cada vegada que detectem un caràcter `L` seguit d'un caràcter `A` haurà aparegut la partícula `LA` i la comptabilitzarem.

Per tant, contínuament compararem caràcters arribats per teclat amb els caràcters `L` i `A`. El problema es pot resoldre fàcilment mantenint en memòria el darrer caràcter llegit (per tant, utilitzant una única variable). Us proposem que resolgueu el problema guardant únicament el darrer caràcter llegit.

Ens interessa, però, abordar l'exemple per una altra banda. Farem servir dues variables de tipus caràcter on puguem tenir guardats els dos darrers caràcters llegits. En tal cas, saber si ha aparegut la partícula `LA` correspondrà a comparar el penúltim caràcter llegit amb el caràcter `L` i el darrer caràcter llegit amb el caràcter `A`. D'acord?

Algorisme en pseudocodi:

```
programa comptar_LA és
   var  n: natural;
        cp, cu: caràcter; /* caràcters penúltim i últim */
   fivar
   n=0;
   netejar_pantalla;
   escriure ("Introdueixi un text."); saltar_línia;
   escriure ("L'aparició del símbol $ indicarà la fi.");
   saltar_línia;
```

```
   cp = ' '; /* caràcter en blanc */
   llegir (cu);
   mentre cu != '$' fer
      si cp == 'L' i cu == 'A' llavors n++; fisi
      cp = cu; llegir (cu);
   fimentre
   escriure ("Hi ha ", n, " partícules LA abans de $.");
fiprograma
```

És necessari, en llegir el primer caràcter, haver inicialitzat la variable `cp` amb un espai en blanc: penseu què passaria si no l'haguéssim inicialitzat i per casualitat la memòria tingués un caràcter `L` en la posició ocupada per la variable i el primer caràcter introduït per l'usuari fos una `A`. El programa comptaria una aparició de la partícula quan en realitat no hauria aparegut.

Per altra banda, si penseu que aquest problema es pot solucionar fent inicialment dues lectures de caràcter, esteu equivocats. Qui ens assegura que l'usuari introduirà com a mínim dos caràcters? Podria ser que l'usuari introduís directament el $. En tal cas, el programa es quedaria "penjat" ja que esperaria un segon caràcter.

El programa s'ha fet amb dues variables de tipus caràcter (`cp` i `cu`), però es podria haver fet amb una taula de dos caràcters.

Codificació en llenguatge C:

```
/* u3n1p05.c */

#include <stdio.h>
#include <conio.h>

void main(void)
{
   unsigned int n=0;
   char cp,cu; /* caràcter penúltim i caràcter últim */
   clrscr();
   printf ("Introdueixi un text.\n");
   printf ("L'aparició del símbol $ indicarà la fi.\n");
   cp = ' ';
   scanf("%c",&cu);
   while (cu != '$')
   {  if (cp == 'L' && cu == 'A') n++;
      cp = cu; scanf ("%c",&cu);
   }
   printf ("Hi ha %u partícules LA abans de $.\n",n);
}
```

Trobareu l'arxiu `u3n1p05.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Anem ara a atacar la situació enunciada a l'inici de l'exemple. Volem generalitzar la situació presentada de manera que sigui fàcil canviar la partícula que busquem i que la llargada pugui ser superior a 2 caràcters.

Anàlisi orgànica

Si s'ha de poder canviar la partícula que busquem no té sentit mantenir comparacions concretes, enmig de l'algorisme, referents als caràcters que formen la partícula. És a dir, les comparacions amb la `L` i la `A` han de desaparèixer per comparacions amb variables que continguin els caràcters de la partícula.

I ja que es pretén generalitzar, potser podríem pensar que la partícula podria tenir un nombre variable de caràcters, oi?

Farem servir, per a aconseguir-ho, dues taules de caràcters de llargada igual a la de la partícula que s'ha de cercar. Una de les taules contindrà els caràcters d'aquesta partícula i l'altra servirà per a anar guardant els darrers caràcters llegits, de manera similar a la solució anterior.

Algorisme en pseudocodi:

```
programa comptar_part és
   const LLARGADA = 2;
   /* Aquesta constant ha de contenir el nombre de caràcters de
      la partícula que es vol cercar */
   ficonst
   var  n: natural;          /* acumulador d'aparicions */
        k: natural;          /* variable auxiliar */
        part: taula [LLARGADA] de caràcter;
        /* taula per a contenir la partícula que es vol cercar */
        c: taula [LLARGADA] de caràcter;
        /* taula per a contenir els darrers caràcters llegits */
   fivar
   n = 0;
   part[0] = 'L'; part[1] = 'A';
   /* En cas de voler cercar una altra partícula, només cal canviar el valor de la
      constant LLARGADA i inicialitzar les caselles de la taula part amb els caràcters
      de la partícula */
   netejar_pantalla;
   escriure ("Introdueixi un text."); saltar_línia;
   escriure ("L'aparició del símbol $ indicarà la fi."); saltar_línia;
   per k = 0 fins LLARGADA-2 fer c[k] = ' '; fiper
   llegir (c[LLARGADA-1]);
   /* Llegim el primer caràcter i el situem a la darrera posició de la taula.
      La resta de posicions tenen espais en blanc */
   mentre c[LLARGADA-1] != '$'
      k = 0;
      mentre k<LLARGADA i c[k] == part[k] fer k = k+1; fimentre
      /* Comprovem si tots els caràcters de la partícula són iguals als darrers caràcters
         llegits per teclat */
      /* En sortir del mentre cal esbrinar si el motiu de sortida ha estat que tots eren
         iguals (k haurà arribat a valer LLARGADA) o si, al contrari, no són iguals */
      si k == LLARGADA llavors n = n+1; fisi
      per k = 0 fins LLARGADA-2 fer c[k] = c[k+1]; fiper
      /* Movem una posició els darrers caràcters llegits per llegir-ne un de nou */
      llegir (c[LLARGADA-1]);
   fimentre
```

```
   /* Anem repetint el procés mentre no s'introdueixi el $ */
   escriure ("Hi ha ", n, " partícules ");
   per k = 0 fins LLARGADA-1 fer escriure (part[k]); fiper
   escriure (" abans de $."); saltar_línia;
fiprograma
```

Codificació en llenguatge C:

```
/* u3n1p06.c */

#include <stdio.h>
#include <conio.h>

#define LLARGADA 3   /* Doncs cerquem la particula PER */

void main(void)
{
   unsigned int n=0,k;
   char part[LLARGADA],c[LLARGADA];
   part[0]='P'; part[1]='E'; part[2]='R';
   clrscr();
   printf ("Introdueixi un text.\n");
   printf ("L'aparició del símbol $ indicarà la fi.\n");
   for (k=0; k<LLARGADA-1; k++) c[k]=' ';
   scanf("%c",&c[LLARGADA-1]);
   while (c[LLARGADA-1] != '$')
   {  for (k=0; k<LLARGADA && c[k]==part[k]; k++);
      if (k==LLARGADA) n++;
      for (k=0; k<LLARGADA-1; k++) c[k]=c[k+1];
      scanf ("%c",&c[LLARGADA-1]);
   }
   printf ("Hi ha %u partícules ",n);
   for (k=0; k<LLARGADA; k++) printf ("%c",part[k]);
   printf (" abans de $.\n");
}
```

Trobareu l'arxiu `u3n1p06.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Exemple 2 d'aprofundiment en el tractament de taules

Volem fer un programa que compti quantes vegades apareix la paraula PILOTA en un text acabat en $.

Anàlisi funcional

En aquest cas es tracta de cercar una paraula i no pas una partícula. Per tant, si l'usuari introdueix PILOTAR, no s'ha de comptabilitzar. Quan considerarem que tenim una paraula? Una paraula és una seqüència ininterrompuda de lletres. Des del punt de vista informàtic, les lletres estan agrupades de la 'A' a la 'Z' i de la 'a' a la 'z', però a part tenim les lletres accentuades i els caràcters específics de cada llengua ('ç', 'ñ', etc.). A més, hi pot haver paraules que continguin dígits? Possiblement sí. Fer un tractament real del problema pot ser fatigós i no és l'objectiu d'aquest exemple.

Considerarem paraula tota seqüència ininterrompuda de caràcters acabada per un espai en blanc o per un punt o per una coma. Evidentment podríem considerar molts més casos. Suposem únicament aquests tres.

Anàlisi orgànica

Pensem la resolució d'aquest problema de manera que sigui fàcil canviar la paraula que s'ha de cercar dins el text, tal com hem fet en el darrer exemple.

Algorisme en pseudocodi:

Ja heu comprovat que els programes es comencen a complicar. Des del principi d'aquest crèdit hem defensat la claredat i llegibilitat dels algorismes com un dels objectius primordials de la programació. Per tant, quan un algorisme es comença a complicar no ens ha de preocupar, al contrari, hem de trossejar-lo en diferents apartats, de manera que sigui més senzill fer-ne el seguiment. Aprofitarem aquest exemple per a iniciar-nos en aquest tipus de disseny.

```
programa comptar_paraula és
   const LLARGADA = ???     /* llargada de la paraula que es vol comptar */
   ficonst
   var   n: natural;        /* variable per a comptar les aparicions */
         k: natural;        /* variable auxiliar per guardar la longitud de la paraula
                               en curs */
         paraula: taula [LLARGADA] de caràcter;
                            /* taula per a contenir la paraula que es vol comptar */
         curs: taula [LLARGADA] de caràcter;
                            /* taula per a guardar la paraula en curs */
         c: caràcter;       /* variable per a llegir un caràcter del teclat */
   fivar
   inicialitzar_variables;
   llegir_paraula_de_teclat;          /* guardarem la longitud a la variable k */
   mentre k > 0 fer
      /* El procés acabarà quan tinguem 0 com a longitud de la paraula en curs */
      tractar_paraula_guardada;
      llegir_paraula_de_teclat;       /* guardarem la longitud a la variable k */
   fimentre
   donar_resultat;
fiprograma
```

En l'algorisme intentem deixar ben clara la seqüència de passos que cal seguir. No els hem definit encara, però els hem introduït, donant-los un nom entenedor. Evidentment, tal com es veu en l'algorisme, el que cal fer primer de tot és inicialitzar les variables amb els valors adequats i agafar la primera paraula que ens arribi per teclat. Aneu amb compte, perquè l'usuari pot introduir directament el $ sense cap paraula prèvia. En qualsevol cas, quan estiguem segurs que s'ha guardat una paraula

(longitud de la paraula superior a zero), passem a fer-ne el tractament per, posteriorment, repetir el procés amb la paraula següent.

En l'algorisme anterior trobem les sentències següents:

```
   inicialitzar_variables
   llegir_paraula_de_teclat
   tractar_paraula_guardada
   donar_resultat
```

Les quatre han estat declarades per nosaltres. Cadascuna defineix un conjunt d'instruccions que per si soles tenen significat. Són accions definides. (!!)

Acció i funció

Segur que alguns de vosaltres coneixeu la possibilitat de definir accions i funcions en els llenguatges de programació estructurada i modular. En aquest punt del crèdit està justificat començar a treballar amb accions, tot i que no han estat degudament introduïdes, si amb això aconseguim claredat en els algorismes.

Podeu pensar que una **acció** és un conjunt d'instruccions que globalment formen una entitat amb significat propi.

(!!) A la unitat didàctica "Organitzem programes i dades" s'estudien a fons els conceptes d'acció i funció.

Molt bé. Tenim unes accions declarades. Com en definim el contingut? De moment només ho farem en pseudocodi:

```
acció inicialitzar_variables és
   paraula[0] = '?'; paraula[1] = '?', paraula[2] = '?'
   ...
   paraula[LLARGADA-2] = '?'; paraula[LLARGADA-1] = '?';
   /* Omplim paraula amb els caràcters de la paraula que es vol comptar */
   n = 0;     /* Inicialitzem el comptador */
   netejar_pantalla;
   escriure ("Introdueixi un text."); saltar_línia;
   escriure ("L'aparició del símbol $ indicarà la fi."); saltar_línia;
   llegir (c);     /* Llegir el primer caràcter introduït per l'usuari */
fiacció

acció llegir_paraula_de_teclat és
   mentre c == ' ' o c == ',' o c == '.' fer llegir (c); fimentre
   /* Saltem tots els caràcters que no formen part d'una paraula i que ens podem trobar
      abans de l'inici de la paraula; no en fem res. Però per poder iniciar aquest procés
      hem de tenir un primer caràcter a la variable c. Per això hem acabat l'acció
      anterior amb la lectura de la variable c. Per què no es pot fer com a primera
      instrucció d'aquesta acció en lloc de la darrera instrucció de l'acció anterior? */
   k = 0;
   mentre  k < LLARGADA i c != '$' i c != ' ' i c != ',' i c != '.' fer
      curs[k] = c; k = k+1; llegir (c);
   fimentre
   /* Si el caràcter que tenim a c no és un dels caràcters que indiquen final de paraula,
      vol dir que és un caràcter de la paraula i, per tant, l'hem de guardar dins la taula
      curs. Atenció! Hem de fer això sempre que hi hagi espai a la taula ja que l'hem
      declarada de grandària igual a la de la paraula que estem cercant i és molt possible
      que l'usuari introdueixi paraules més llargues. Fixem-nos que, en el cas que la
      paraula introduïda per l'usuari sigui més llarga que la grandària de curs, en
```

```
     aquesta queda introduït el tros que hi cap i dins la variable c hi haurà el caràcter
     següent, que no serà cap dels caràcters de finalització de paraula. */
fiacció

acció tractar_paraula_guardada és
   /* Quan s'entra en aquesta acció, a curs hi ha una paraula, o tros de paraula, i k
     conté la seva llargada */
   si k == LLARGADA i (c == '$' o c == ' ' o c == '.' o c == ',')
     llavors
       /* En aquest cas, la taula curs està plena amb un tros de llargada igual a la de
         la paraula que estem comptant i no s'ha quedat cap caràcter pendent d'espai a
         curs, ja que c conté el caràcter següent al tros guardat i acabem de comprovar
         que és un dels caràcters que indica fi de paraula. */
       k = 0;     /* Comparem el contingut de les dues taules, caràcter a caràcter */
       mentre k < LLARGADA i paraula[k] == curs[k] fer   k = k + 1; fimentre
       /* En sortir del mentre cal esbrinar si el motiu de sortida ha estat que tots
         eren iguals (k haurà arribat a valer LLARGADA) o si, al contrari, no són
         iguals */
       si k == LLARGADA llavors n++; fisi
     sinó
       /* En aquest cas, la taula curs o conté un tros de llargada inferior a la de la
         taula que estem comptant o conté un tros de llargada igual però s'han quedat
         caràcters de la paraula en curs sense poder entrar a la taula curs; és a dir,
         en qualsevol cas, vol dir que la paraula en curs no és igual a la paraula que
         estem comptabilitzant. Per tant, en aquest cas no hem d'incrementar n. Però el
         que sí s'ha de fer és acabar de saltar tots aquells caràcters que puguin
         quedar de la paraula en curs que no hagin cabut dins curs. El primer, en
         aquest cas, ja el tenim a la variable c. Aquesta és la causa del mentre
         següent.*/
       mentre c != '$' i c != ' '  i c != ',' i c != '.' fer llegir (c); fimentre
   fisi
fiacció
```

Quan se surt de l'acció anterior, la variable `c` sempre conté el caràcter següent a la darrera paraula tractada. Per tant, quan en el bucle principal del programa es passa a executar l'acció `llegir_paraula_teclat`, la variable `c` ja conté un caràcter. Per això no es pot tenir com a primera instrucció d'aquesta acció la lectura de la variable `c`, ja que estaríem perdent el contingut que hi havia. Però la primera vegada que s'executa l'acció `llegir_paraula_teclat`, la variable `c` no estaria plena. Per a garantir que en qualsevol execució d'aquesta acció la variable `c` contingui ja un caràcter hem afegit com a darrera instrucció de l'acció `inicialitzar_variables` la lectura del primer caràcter.

```
acció donar_resultat és
   escriure ("La paraula ");
   per k = 0 fins LLARGADA - 1 fer escriure (paraula[k]); fiper
   escriure (" ha sortit ", n, " vegades."); saltar_línia;
fiacció
```

Codificació en llenguatge C:

En l'algorisme en pseudocodi hem començat a fer servir accions. El llenguatge C també les inclou, però la seva definició no és tan simple com en el pseudocodi. Per tant, de moment, la codificació en C no preveurà la utilització d'accions definides.

Com es codifica, doncs? Molt fàcil. Només cal substituir la declaració de l'acció pel codi corresponent. Evidentment, si en l'algorisme una mateixa acció es crida des de dos llocs diferents, en codificar-lo en C caldrà substituir totes les crides pel corresponent codi. És clar que hi ha repetició de codi, cosa no aconsellable. Això desapareixerà quan es treballi amb accions en C.

Trobareu l'arxiu `u3n1p07.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u3n1p07.c */

#include <stdio.h>
#include <conio.h>

#define LLARGADA 6   /* Doncs cerquem la particula PER */

void main(void)
{
   unsigned int n,k;
   char paraula[LLARGADA],curs[LLARGADA],c;

   /* Substitució de l'acció inicialitzar_variables */
   paraula[0]='P'; paraula[1]='I'; paraula[2]='L';
   paraula[3]='O'; paraula[4]='T'; paraula[5]='A';
   n = 0;
   clrscr();
   printf ("Introdueixi un text.\n");
   printf ("L'aparició del símbol $ indicarà la fi.\n");
   scanf("%c",&c);

   /* Substitució de l'acció llegir_paraula_teclat */
   while (c==' ' || c==',' || c=='.') scanf("%c",&c);
   k = 0;
   while ( k < LLARGADA && c!='$' && c!=' ' && c!=',' && c!='.')
   {  curs[k] = c; k++; scanf("%c",&c); }

   /* Bucle del programa */
   while (k > 0)
   {
      /* Substitució de l'acció tractar_paraula_guardada */
      if ( k==LLARGADA && ( c=='$' || c==' ' || c=='.' || c==','))
      {
         for (k=0; k<LLARGADA && paraula[k]==curs[k]; k++);
         if (k==LLARGADA) n++;
      }
      else
         while (c!='$' && c!=' ' && c!=',' && c!='.') scanf("%c",&c);

      /* Substitució de l'acció llegir_paraula_teclat */
      while (c==' ' || c==',' || c=='.') scanf("%c",&c);
      k = 0;
      while ( k < LLARGADA && c!='$' && c!=' ' && c!=',' && c!='.')
      {  curs[k] = c; k++; scanf("%c",&c); }
```

```
    }

    /* Substitució de l'acció donar_resultat */
    printf ("La paraula");
    for (k=0; k<LLARGADA; k++) printf ("%c",paraula[k]);
    printf ("ha sortit %u vegades.\n",n);
}
```

1.5. Gestió de taules multidimensionals

Els llenguatges de programació acostumen a facilitar la possibilitat de definir i gestionar taules multidimensionals.

En general ens podem trobar amb diferents maneres de definir taules multidimensionals:

- Com una taula tal que cada cel·la és a la seva vegada una taula i així successivament. La sintaxis d'una definició d'aquest estil seria:

```
var nom: taula [dim1] de taula [dim2] de ... de <T>;
```

  En la definició precedent la partícula <T> es refereix a un tipus de dada conegut. Així, per exemple, si necessitem definir un taula de dues dimensions per a simular uns mots encreuats de 10 files i 12 columnes, definiríem:

```
var encreuats: taula [10] de taula [12] de caràcter;
```

- Com una taula acompanyada directament pels valors de les diferents dimensions tal i com es veu a sintaxis:

```
var nom: taula [dim1, dim2, ...] de <T>;
```

  Emprant aquesta sintaxis, la taula de dues dimensions per a simular uns mots encreuats de 10 files i 12 columnes, es definiria com:

```
var encreuats: taula [10,12] de caràcter;
```

  Aquesta segona possibilitat de sintaxis és la que utilitzarem, de moment, per implementar les taules multidimensionals en el llenguatge C. (!!)

No importa quina sintaxis utilitzar en pseudocodi donat que ambdues són prou clares.

L'accés a les diferents caselles d'una taula multidimensional s'ha d'efectuar, de forma similar a les taules unidimensionals, indicant les

coordenades de la cel·la a accedir relatives a les diferents dimensions. Així, per enregistrar els valors `A`, `B` i `C` a la taula de la figura 6, cal indicar:

```
t[1,2] = 'A'; t[2,8] = 'B'; t[3,5] = 'C';
```

o també:

```
t[1][2] = 'A'; t[2][8] = 'B'; t[3][5] = 'C';
```

En pseudocodi no importa quina sintaxis utilitzar donat que ambdues són prou clares. El llenguatge C ens obliga a utilitzar la segona opció, indicant cada coordenada entre claudàtors. (!!)

Figura 6 . Taula bidimensional de nom `t` amb dimensions 5 x 10

t	0	1	2	3	4	5	6	7	8	9
0										
1			A							
2									B	
3						C				
4										

Fixem-nos que en les taules multidimensionals també considerarem, com en les unidimensionals, que les coordenades de les cel·les relatives a les diferents dimensions s'enumeren a partir de zero. Això pot ser diferent segons els llenguatges, però com que el llenguatge C ens obliga, aplicarem el mateix criteri en el pseudocodi (i així no us marejarem...)

1.5.1. Recorregut total

La instrucció repetitiva més útil per a efectuar un recorregut per una taula de `n` dimensions és, de forma similar als recorreguts en les taules unidimensionals, la instrucció `per`, utilitzant-ne `n` imbricades una dins l'altra, de la forma següent:

```
var i1, i2,...,in : enter; fivar
/* Es suposa que DIM1, DIM2,..., DIMn són les dimensions de la
   taula i poden ser constants o variables amb el valor adequat
*/
...
per i1=0 fins DIM1-1 fer
   per i2=0 fins DIM2-1 fer
      ...
      per in=0 fins DIMn-1 fer
         tractar la cel·la t[i1,i2,...,in]
      fiper
```

```
      ...
   fiper
fiper
```

En llenguatge C seria:

```
int i1, i2,...,in;
...
for (i1=0; i1<DIM1; i1++)
   for (i2=0; i2<DIM2; i2++)
      ...
      for (in=0; in<DIMn; in++)
         tractar la cel·la t[i1][i2]...[in];
```

Fixem-nos que no ens cal utilitzar les claus {} per a cada instrucció donat que dins de cada `for` només hi ha un altre `for`. L'excepció pot ser en el punt més intern, en cas que el tractament de la cel·la estigui constituït per més d'una instrucció. En tal cas, el conjunt d'instruccions per al tractament de la cel·la haurien d'anar entre claus {}.

1.5.2. Recerca d'un valor

Per a efectuar la recerca d'un valor dins una taula de `n` dimensions utilitzaríem, de forma similar a les recerques en les taules unidimensionals, la instrucció `mentre`, utilitzant-ne `n` imbricades una dins l'altra. Vegem-ho en el cas `n=3` cercant un valor `<v>` en una taula `t`:

```
var i1, i2, i3 : enter; /* Variables per recórrer les cel·les */
    trobat : booleà;    /* Variable semàfor inicialitzada amb fals i que prendrà el
                           valor cert en el moment de trobar-se el valor cercat */
fivar
/* Es suposa que DIM1, DIM2, DIM3 són les dimensions de la
   taula i poden ser constants o variables amb el valor adequat
*/
...
trobat = fals;
i1 = 0;
mentre no trobat i i1 < DIM1 fer
   i2 = 0;
   mentre no trobat i i2 < DIM2 fer
      i3 = 0;
      mentre no trobat i i3 < DIM3 i t[i1,i2,i3] != <v> fer
         i3++;
      fimentre
      si i3 < DIM3 (***)
         llavors trobat = cert; /* El valor es troba a les coordinades (i1,i2,...,in)
         sinó i2++;
      fisi
   fimentre
   si no trobat
      llavors i1++;
   fisi
fimentre
```

Fixem-nos en la importància de la condició (***) per a detectar si hem trobat o no el valor cercat. Recordem que s'ha de preguntar per la posició que té la variable `i3` en sortir del mentre i no pas pel valor que hi ha a la cel·la `t[i1,i2,i3]` doncs en cas de no haver trobat el valor, `qt` valdria `DIM3` i la cel·la `t[i1,i2,DIM3]` possiblement no existeixi. ‼
Fixem-nos també, que quan sortim d'un mentre no extern, només incrementem el valor de la corresponent variable i en cas de no haver trobat encara el valor. D'aquesta manera assegurem que en sortir de tots els mentre, les variables `i1`, `i2` i `i3` contenen les coordenades de la cel·la on resideix el valor cercat.

I per últim, fixem-nos que la variable trobat ens serveix per aturar els recorreguts pels diferents `mentre` en el moment en que s'hagi trobat l'element cercat.

En llenguatge C seria:

```
int i1, i2, i3; /* Variables per recórrer les cel·les */
int trobat;     /* Variable semàfor inicialitzada amb fals i que prendrà el
                   valor cert en el moment de trobar-se el valor cercat */
...
trobat = 0;
i1 = 0;
while (!trobat && i1 < DIM1)
{  i2 = 0;
   while (!trobat && i2 < DIM2)
   {  i3 = 0;
      while (!trobat && i3 < DIM3 && t[i1][i2][i3] != <v>) i3++;
      if (i3 < DIM3)
         trobat = 1; /* El valor es troba a les coordinades (i1,i2,...,in)
      else i2++;
   }
   if (!trobat) i1++;
}
```

En aquest algorisme, la utilització de la instrucció `for` del llenguatge C és un pel més complicada per aconseguir que les variables i1, i2 i i3 no modifiquin el seu valor en cas d'haver localitzat el valor cercat. Seria:

```
int i1, i2, i3; /* Variables per recórrer les cel·les */
int trobat;     /* Variable semàfor inicialitzada amb fals i que prendrà el
                   valor cert en el moment de trobar-se el valor cercat */
...
for (trobat = 0, i1 = 0; i1 < DIM1; i1++)
{  for (i2 = 0; i2 < DIM2; i2++)
   {  for (i3 = 0; i3 < DIM3 && t[i1][i2][i3] != <v>; i3++);
      if (i3 < DIM3)
      {  trobat = 1; /* El valor es troba a les coordinades (i1,i2,...,in)
         break; (*)
      }
   }
   if (trobat) break; (**)
}
```

La instrucció (*) provoca que s'efectuï la sortida del `for (i2=...` sense executar-se `i2++` i així no perdem el valor emmagatzemat dins i2 en cas d'haver-se trobat el valor cercat. La instrucció (**) provoca que s'efectuï la sortida del `for (i1=...` sense executar-se `i1++` i així tampoc perdem el valor emmagatzemat dins i1.

1.5.3. Exemples de manipulació de taules multidimensionals

Vegem alguns exemples que ens permetran practicar la manipulació de taules multidimensionals.

Exemple 1 d'utilització de taules multidimensionals.

Fer un programa que defineixi una taula bidimensional de 5 files i 5 columnes de números enters i ompli totes les cel·les amb el producte de les coordenades que corresponen a la cel·la. Posteriorment visualitzar el contingut de la taula.

Algorisme en pseudocodi:

```
programa omplir_taula és
   const FIL = 5;     /* Nombre de files */
         COL = 5;     /* Nombre de columnes */
   ficonst
   var
      t: taula [FIL][COL] de enter;
      f,c: natural;      /* Per recórrer files i columnes */
   fivar
   per f=0 fins FIL-1 fer
      per c=0 fins COL-1 fer
         t[f][c]=f*c;
      fiper
   fiper
   netejar_pantalla;
   escriure ("Contingut de la taula"); saltar_línia;
   escriure ("*********************"); saltar_línia;
   per f=0 fins FIL-1 fer
      per c=0 fins COL-1 fer
         escriure (t[f][c]);
      fiper
      saltar_línia;
   fiper
fiprograma
```

Codificació en llenguatge C:

```
#include <stdio.h>
#include <conio.h>
#define FIL 5   /* Nombre de files */
#define COL 5   /* Nombre de columnes */

void main(void)
{
   int t[FIL][COL];
   int f,c;   /* Per recórres les files i les columnes */

   for (f=0; f<FIL; f++)
      for (c=0; c<COL; c++)
         t[f][c]=f*c;

   clrscr();
   printf ("Contingut de la taula:\n");
   printf ("**********************\n");
   for (f=0; f<FIL; f++)
   {
      for (c=0; c<COL; c++) printf("%4d ",t[f][c]);
      printf ("\n");
   }
}
```

Trobareu l'arxiu `u3n1p08.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Exemple 2 d'utilització de taules multidimensionals.

Volem fer un programa que executi el *Joc de la Vida*.

Anàlisi funcional

Al número d'octubre de *Scientific American* de 1970, a la secció *Mathematical games*, Martin Gardner hi exposava, per primera vegada, una descripció del joc *Life* (el *Joc de la Vida*), que havia estat creat el mateix any pel matemàtic britànic John Horton Conway (Liverpool, 1937).

El *Joc de la Vida* és un joc per a zero jugadors, cosa que vol dir que l'evolució està completament determinada per l'estat inicial i les regles del joc, sense que calgui cap intervenció d'un jugador humà, una vegada s'ha iniciat el procés.

El *Joc de la Vida* es juga en una quadrícula (el món), en la qual, a cadascuna de les seves cel·les, hi ha un ésser que pot ser viu o mort. L'ésser de cada cel·la té vuit éssers veïns que són els que ocupen les cel·les adjacents a la cel·la considerada i el món evoluciona en estats consecutius (passos) de manera que l'estat (viu o mort) de tots els éssers en un cert moment determinen l'estat que tindran en l'estat següent, després d'un *pas*, a partir d'aquestes dues regles:

- Un ésser mort que tingui, exactament, tres veïns vius, passa a ser un ésser viu.
- Un ésser viu amb 2 o 3 veins vius roman viu; en cas contrari (cap o un veí viu o 4 o més veïns vius) mor per aïllament o per superpoblació.

Amb aquestes regles del joc i a partir d'un estat inicial, el joc consisteix en veure l'evolució del món. Moltes configuracions inicials d'éssers no sobreviuen més de tres o quatre passos, en canvi, d'altres, porten a evolucions molt vistoses i, encara, n'hi ha d'altres que són estables o bé cícliques. Les més interessants han estat batejades. La figura 7 ens en mostra algunes.

Figura 7. Estats inicials interessants del *Joc de la Vida.*

Quan tingueu fet el programa podreu comprovar, respecte la figura 7, que:

- el *bloc* i la *barca* corresponen a vides estàtiques (no canvien mai)
- el *semàfor* i el *gripau* són oscil·ladors (conjunt repetitiu de vides estàtiques)
- el *planejador* i l'*astronau* són naus espacials que recórren el tauler al llarg del temps

Si us pica la curiositat, comentar-vos que hi ha un munt de pàgines a Internet amb informació referent al *Joc de la Vida*.

Anàlisi orgànica

Sembla que la millor manera de plantejar el joc és definir una taula bidimensional per representar el món i definir-hi un estat inicial. De fet, com que la figura 7 ens mostra sis estats inicials interessants, podríem definir-los tots ells en un menú inicial de manera que l'usuari pogués escollir l'estat inicial que cregués oportú.

Sigui quin sigui l'estat inicial escollit, el programa hauria d'iniciar un procés repetitiu fins que l'usuari premés alguna tecla per provocar l'aturada del procés i pogués decidir d'iniciar un altre joc o finalitzar el programa. Com que encara no tenim mecanismes per aconseguir aquest efecte, deixarem que el programa evolucioni infinitament fins que l'usuari avorti l'execució del programa.

Per a controlar el procés i poder aplicar les regles del joc, sembla lògic disposar d'una taula auxiliar per anotar-hi quin ha de ser l'estat de cada cel·la en el següent pas a partir de l'estat actual. Per aconseguir-ho, per a cada cel·la avaluarem el nombre de cel·les vives que l'envolten i llavors podrem prendre decisió respecte si ha de continuar viva o, en cas d'estar morta, si ha de tornar a néixer en el següent pas.
També cal tenir en compte que el *Joc de la Vida* es defineix en un tauler infinit. Per simular-ho en la nostra taula, considerarem que les vores dreta i esquerra estan unides (com si estesin cosides) a l'igual que les vores superior i inferior. I, per tant, haurem d'anar en compte quan la cel·la tractada estigui en una de les quatre vores del tauler, doncs:

- si es troba a la vora esquerra, cal tenir en compte que les cel·les de la seva esquerra són les cel·les que es troben a la vora dreta del tauler
- si es troba a la vora dreta, cal tenir en compte que les cel·les de la seva dreta són les cel·les que es troben a la vora esquerra del tauler
- si es troba a la vora superior, cal tenir en compte que les cel·les de la fila per damunt d'ella són les cel·les que es troben a la vora inferior del tauler
- si es troba a la vora inferior, cal tenir en compte que les cel·les de la fila per sota d'ella són les cel·les que es troben a la vora superior del tauler

Per últim, per afavorir que l'usuari pugui apreciar perfectament l'evolució del sistema, sembla lògic alentir el procés de visualització, fet que es pot assolir fàcilment amb les funcions que acostumen a incorporar tots els llenguatges i, per tant, suposarem que també existeixen en pseudocodi. Així, suposarem que en pseudocodi disposem del procediment següent:

Procediment

Un procediment és, de forma similar a una funció, un conjunt agrupat d'instruccions, però, a diferència de les funcions, no retorna cap resultat. Hi ha llenguatges, com C, que agrupen els dos conceptes sota l'únic nom de funció

```
procediment pausa (segons: natural);
/* atura l'execució del programa el nombre de segons indicat */
```

El llenguatge C incorpora la funció `sleep (unsigned seconds)` per aconseguir l'aturada.

Algorisme en pseudocodi:

```
programa joc_de_la_vida és
   const X = 80;   /* Número de columnes */
         Y = 25;   /* Número de files */
   ficonst
   var
      mon, esttat: taula [X][Y] de caràcter;
      ix,iy: enter;   /* Per moure'ns pel món */
      ixd: enter; /* Per calcular, donada una coordenada X, la coordenada de la dreta */
      ixe: enter; /* Per calcular, donada una coordenada X, la coordenada de l'esquerra */
      iys: enter; /* Per calcular, donada una coordenada Y, la coordenada superior */
      iyi: enter; /* Per calcular, donada una coordenada Y, la coordenada inferior */
```

```
      vius: enter; /* Per comptar quants éssers vius hi ha al voltant d'una cel·la */
      aux: enter; opc: caràcter;
   fivar

   netejar_món;    /* No hi ha cap ésser viu */
   mostrar_menú_i_escollir_estat_inicial;
   situar_estat_inicial;
   procés;   /* Posa en marxa el Joc de la Vida amb els valors indicats */
fiprograma
```

```
acció netejar_món és
   per ix = 0 fins X-1 fer
      per iy = 0 fins Y-1 fer
         mon[ix][iy]=' ';
      fiper
   fiper
fiacció
```

```
acció mostrar_menú_i_escollir_estat_inicial és
   escriure ("Estats inicials a escollir:");
   escriure ("**************************");
   escriure ("1. Bloc\n");
   escriure ("2. Barca\n");
   escriure ("3. Semàfor\n");
   escriure ("4. Gripau\n");
   escriure ("5. Planejador\n");
   escriure ("6. Astronau\n\n");
   escriure ("0. Sortida\n");
   escriure ("Premi opció:");
   repetir
      llegir(opc);
   fins (opc>='0' i opc <='6');
fiacció
```

```
acció situar_estat_inicial és
   encasde opc
      ser '1': mon[10][10]='X'; mon[10][11]='X'; mon[11][10]='X'; mon[11][11]='X';
      ser '2': mon[10][10]='X'; mon[10][11]='X'; mon[11][10]='X'; mon[11][12]='X';
               mon[12][11]='X';
      ser '3': mon[10][10]='X'; mon[10][11]='X'; mon[10][12]='X';
      ser '4': mon[10][11]='X'; mon[10][12]='X'; mon[10][13]='X'; mon[11][10]='X';
               mon[11][11]='X'; mon[10][12]='X';
      ser '5': mon[10][10]='X'; mon[10][11]='X'; mon[10][12]='X'; mon[11][10]='X';
               mon[12][11]='X';
      ser '6': mon[10][11]='X'; mon[10][14]='X'; mon[11][10]='X'; mon[12][10]='X';
               mon[12][14]='X'; mon[13][10]='X'; mon[13][11]='X'; mon[13][12]='X';
               mon[13][13]='X';
   fiencasde
fiacció
```

```
acció procés és
   si opc!='0'
      llavors
         escriure ("Per finalitzar l'execució premi CTRL+C");
         escriure ("Premi qualsevol tecla per iniciar l'execució...");
         esperar_tecla; netejar_pantalla;
         mentre 1==1 fer   /* Bucle infinit */
```

```
        per ix=0 fins X-1 fer
           per iy=0 fins Y-1 fer
              vius=0;
              si ix<X-1 llavors ixd=ix+1; sinó ixd=0; fisi
              si ix>0 llavors ixe=ix-1; sinó ixe=X-1; fisi
              si iy<Y-1 llavors iyi=iy+1; sinó iyi=0; fisi
              si iy>0 llavors iys=iy-1; sinó iys=Y-1; fisi
              /* Les 4 instruccions anteriors calculen, donada la cel·la (ix,iy)
                 quina és la coordenada de la dreta i de l'esquerra, controlant
                 el cas en que s'estigui en un extrem del tauler */
              si mon[ixd][iys]=='X' llavors vius++; fisi
              si mon[ixd][iy]=='X' llavors vius++; fisi
              si mon[ixd][iyi]=='X' llavors vius++; fisi
              si mon[ixe][iys]=='X' llavors vius++; fisi
              si mon[ixe][iy]=='X' llavors vius++; fisi
              si mon[ixe][iyi]=='X' llavors vius++; fisi
              si mon[ix][iys]=='X' llavors vius++; fisi
              si mon[ix][iyi]=='X' llavors vius++; fisi
              /* Les 8 instruccions anteriors calculen, donada la cel·la (ix,iy)
                 el nombre d'éssers vius que hi ha en les 8 cel·les del seu voltant */

              /* Ara que ja tenim el nombre d'éssers vius que hi ha en les 8 cel·les
                 adjacents, podem deduir l'estat de la cel·la en el següent pas */
              estat[ix][iy]=mon[ix][iy];
              si mon[ix][iy]==' '
                 llavors                /* la cel·la (ix,iy) ara està morta */
                    si vius==3
                       llavors estat[ix][iy]='X';   /* passarà a estar viva */
                    fisi
                 sinó                   /* la cel·la (ix,iy) ara està viva */
                    si vius<2 || vius>3
                       llavors estat[ix][iy]=' ';   /* passarà a estar morta */
                    fisi
              fisi
           fiper
        fiper
        /* Els dos "per" anteriors han calculat quin ha de ser l'estat següent
           de totes les cel·les. Ara ja podem passar aquest estat següent de la
           taula "estat" a la taula "mon" i visualitzar el nou estat de totes
           les cel·les */
        per ix = 0 fins X-1 fer
           per iy = 0 fins Y-1 fer
              mon[ix][iy]=estat[ix][iy];
              cursor(ix,iy);escriure(mon[ix][iy]);
           fiper
        fiper
        pausa (1);   /* Aturada d'1 segon */
     fimentre
  fisi
fiacció
```

Codificació en llenguatge C:

Trobareu l'arxiu `u3n1p09.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u3n1p09.c */

#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* Per compatibilitat entre plataformes */

#define X 80 /* Número de columnes */
#define Y 25 /* Número de files */

void main(void)
{
   char mon[X][Y];
   char estat[X][Y];
   int ix,iy; /* Per moure'ns pel món */
   int ixd;   /* Per calcular, donada una coordenada X, la coordenada de la dreta */
   int ixe;   /* Per calcular, donada una coordenada X, la coordenada de l'esquerra */
   int iys;   /* Per calcular, donada una coordenada Y, la coordenada superior */
   int iyi;   /* Per calcular, donada una coordenada Y, la coordenada inferior */
   int vius;  /* Per comptar quants éssers vius hi ha al voltant d'una cel·la */

   int aux;
   char opc;

   clrscr();
   for (ix=0; ix<X; ix++)
      for (iy=0; iy<Y; iy++)
         mon[ix][iy]=' ';

   printf ("Estats inicials a escollir:\n");
   printf ("**************************\n\n");
   printf ("1. Bloc\n");
   printf ("2. Barca\n");
   printf ("3. Semàfor\n");
   printf ("4. Gripau\n");
   printf ("5. Planejador\n");
   printf ("6. Astronau\n\n");
   printf ("0. Sortida\n");
   do
   {  printf("Premi opció:"); aux=scanf("%c",&opc); neteja_stdin(); }
   while (aux==0 || opc<'0' || opc>'6');

   switch(opc)
   {
      case '1': mon[10][10]='X'; mon[10][11]='X';
                mon[11][10]='X'; mon[11][11]='X';
                break;
      case '2': mon[10][10]='X'; mon[10][11]='X';
                mon[11][10]='X'; mon[11][12]='X';
                mon[12][11]='X';
                break;
      case '3': mon[10][10]='X'; mon[10][11]='X'; mon[10][12]='X';
                break;
      case '4': mon[10][11]='X'; mon[10][12]='X'; mon[10][13]='X';
                mon[11][10]='X'; mon[11][11]='X'; mon[10][12]='X';
                break;
      case '5': mon[10][10]='X'; mon[10][11]='X'; mon[10][12]='X';
                mon[11][10]='X';
                mon[12][11]='X';
                break;
```

```
      case '6': mon[10][11]='X'; mon[10][14]='X';
                mon[11][10]='X';
                mon[12][10]='X'; mon[12][14]='X';
                mon[13][10]='X'; mon[13][11]='X'; mon[13][12]='X'; mon[13][13]='X';
                break;
   }

   if (opc!='0')
   {
      printf ("\n\nPer finalitzar l'execució premi CTRL+C");
      printf ("\nPremi qualsevol tecla per iniciar l'execució...");
      getch();
      clrscr();
      while (1)
      {  for (ix=0; ix<X; ix++)
            for (iy=0; iy<Y; iy++)
            {  vius=0;
               if (ix<X-1) ixd=ix+1; else ixd=0;
               if (ix>0) ixe=ix-1; else ixe=X-1;
               if (iy<Y-1) iyi=iy+1; else iyi=0;
               if (iy>0) iys=iy-1; else iys=Y-1;

               if (mon[ixd][iys]=='X') vius++;
               if (mon[ixd][iy]=='X') vius++;
               if (mon[ixd][iyi]=='X') vius++;
               if (mon[ixe][iys]=='X') vius++;
               if (mon[ixe][iy]=='X') vius++;
               if (mon[ixe][iyi]=='X') vius++;
               if (mon[ix][iys]=='X') vius++;
               if (mon[ix][iyi]=='X') vius++;

               estat[ix][iy]=mon[ix][iy];
               if (mon[ix][iy]==' ')
               {  if (vius==3) estat[ix][iy]='X'; }
               else
               {  if (vius<2 || vius>3) estat[ix][iy]=' '; }
            }

         for (ix=0; ix<X; ix++)
            for (iy=0; iy<Y; iy++)
            {
               mon[ix][iy]=estat[ix][iy];
               gotoxy(ix,iy);putchar(mon[ix][iy]);
            }
         sleep(1);
      }
   }
}
```

2. Cadenes

Un dels objectius primordials de la informàtica és gestionar de manera àgil i eficaç grans volums d'informació, la qual acostuma a ser numèrica o textual. Com actuem en les situacions en les quals hem de gestionar tires de caràcters? De fet, l'únic tipus de dada, conegut fins el moment, que podem fer servir per a tractar una tira de caràcters és la taula de caràcters.

Quan hem introduït el concepte de taula ja hem fet notar que els llenguatges no ens acostumen a facilitar operacions en l'àmbit de la taula. És a dir, donades tres taules A, B i C d'enters (les suposem de la mateixa grandària), el llenguatge no ens permet fer A = B + C, en què és clar que tothom entén aquesta expressió de la manera següent: les caselles de A prenen per valor els resultats de les sumes de les corresponents caselles de B i C. Si necessitem l'operació anterior, caldrà programar-la amb un procés iteratiu que faci un recorregut per totes les caselles.

En les taules de caràcters es donen situacions semblants. Així, en ocasions tenim dues taules de caràcters d'igual llargada que hem de comparar per a decidir si contenen o no la mateixa tira de caràcters i ho hem de programar, comparant posició a posició, ja que el llenguatge no ens permet fer una comparació del tipus `taula1 == taula2`.

Les consideracions anteriors ens porten a una paradoxa. D'una banda la informàtica pretén fer més àgil als usuaris el tractament de grans volums d'informació entre els quals hi ha molta dada alfanumèrica, i, de l'altra, el programador ha de tractar caràcter a caràcter per a gestionar les dades alfanumèriques.

> Una dada alfanumèrica és la formada per caràcters de qualsevol tipus (lletres, símbols, dígits, etc.).

Anem a comprovar el cost que té el tractament de les dades alfanumèriques gestionant-les via taules de caràcters, que és l'únic mecanisme de què disposem actualment.

Volem un programa que demani a l'usuari el seu nom i que posteriorment tregui per pantalla un missatge personalitzat de salutació.

Anàlisi funcional

No necessitem cap coneixement especial per a poder dissenyar el corresponent algorisme. Bé, suposo que tots teniu clar què vol dir un missatge personalitzat: si un usuari té per nom PEP, un missatge personalitzat seria, per exemple, BON DIA, PEP!

Anàlisi orgànica

Si el programa ha de demanar el nom a l'usuari, ens caldrà espai de memòria per a poder-lo emmagatzemar i posteriorment utilitzar, per a donar el missatge personalitzat. No decidirem pas un munt de variables de tipus caràcter (c1, c2, c3, ...) perquè ens portaria molts problemes. El lògic és treballar amb una taula de caràcters.

Ara bé, quina longitud donem a la taula? Aquesta decisió no la podem prendre en temps d'execució sinó que l'hem de prendre en temps de disseny i codificació. Bé, no tenim cap més remei que pensar en una grandària màxima que lògicament no sigui sobrepassada pel nom que introdueixi l'usuari. Quina pot ser la llargada màxima d'un nom? En tenim prou amb 80 caràcters? Sembla que sí, oi? Tot i que sembli prou espai, el programa ha de saber com actuar davant una entrada superior a la capacitat decidida. Prenem la decisió que en el cas d'entrada superior a 80 caràcters, la resta de caràcters seran negligits.

Algorisme en pseudocodi:

Hem decidit una taula de 80 caràcters (deixem aquest valor com a constant i així serà més fàcil de canviar) per a guardar en memòria el nom introduït per l'usuari. Com farem en pseudocodi l'entrada per teclat? No tenim, de moment, cap altre remei que fer-ho caràcter a caràcter. Com podem esbrinar que l'usuari ha acabat d'introduir el seu nom?

L'usuari acaba la introducció del seu nom mitjançant la tecla <return>, la qual provoca l'enviament al processador del caràcter <Line Feed> (<Final de línia>), el qual correspon al caràcter de codi 10 a la taula ASCII. Per tant, serà necessari controlar l'aparició d'aquest caràcter per saber quan s'acaba l'entrada del nom i assegurar-nos que no sobrepassa la grandària màxima permesa.

Els caràcters LF i CR

Ja sabeu que els 32 primers caràcters de la taula ASCII corresponen a caràcters especials i caràcters de control. Entre aquests es troben el <Line Feed> (LF) que té el codi 10 i el <Carriage Return> (CR) que té el codi 13.

És important saber, en l'entorn en què es treballa, l'actuació de l'ordinador davant la pulsació de la tecla <return>. En pseudocodi prenem el conveni anterior per similitud al funcionament del llenguatge C.

```
programa benvinguda és
   const   MAX = 80;
           LF = 10;
   ficonst
   var   nom: taula [MAX] de caràcter;
         c: caràcter;    /* Variable per llegir del teclat */
         i: natural;     /* Per controlar la llargada del nom */
         k: natural;     /* Variable auxiliar */
   fivar
   netejar_pantalla;
   escriure ("Entri el seu nom, si li plau: ");
   i = 0;
   llegir (c);
   mentre i < MAX i c != LF fer
      nom[i] = c; llegir (c); i = i + 1;
   fimentre
   saltar_línia; saltar_línia;
   escriure ("Benvingut ");
   per k = 0 fins i - 1 fer escriure ( nom[k] ); fiper
   saltar_línia;
fiprograma
```

Codificació en llenguatge C:

```
/* u3n2p01.c */

#include <stdio.h>
#include <conio.h>

#define MAX 80
#define LF 10

void main(void)
{
   char nom[MAX],c;
   unsigned int i,k;
   clrscr(); printf ("Entri el seu nom, si li plau: ");
   scanf ("%c",&c); i=0;
   while (i<MAX && c!=LF)
   {  nom[i] = c; scanf ("%c",&c); i++; }
   printf ("\n\nBenvingut ");
   for (k=0; k<i; k++) printf ("%c",nom[k]);
   printf ("\n");
}
```

Trobareu l'arxiu `u3n2p01.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Com podeu observar, és un veritable enrenou fer tractament de tires de caràcters utilitzant les taules de caràcters. Per això, i atès que en els programes informàtics el tractament de tires de caràcters és continu, els programes incorporen un tipus de dada especial per a aquest tractament.

> El tipus de dada cadena és equivalent a una taula de caràcters amb tractament especial.

En què consisteix el tractament especial? Bàsicament, en un parell de fets:

1) Incorporen, en la dada mateixa, el coneixement sobre la longitud real de la tira de caràcters.

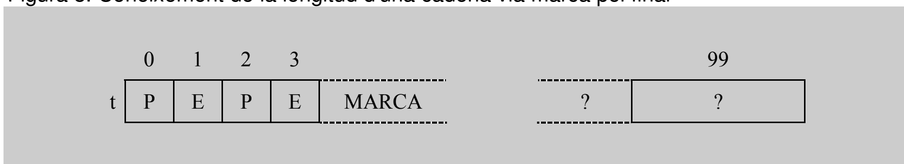
2) El llenguatge aporta operacions d'assignació, relacionals, d'entrada/sortida i d'altres que permeten que el programador els pugui fer servir de manera molt més simplificada que amb el tractament de taules.

En efecte, alguns llenguatges gestionen el tipus cadena com una taula de caràcters que també emmagatzema la longitud real de la tira de caràcters. Aquest emmagatzematge es pot fer, bàsicament, de dues maneres:

1) Hi ha llenguatges que ocupen la posició següent al darrer caràcter emmagatzemat amb un caràcter especial que fa la funció de sentinella. És el cas del llenguatge C i el que suposarem en pseudocodi.

La figura 8 ens exemplifica aquest cas per una cadena amb capacitat per a 100 caràcters on hem introduït PEPE.

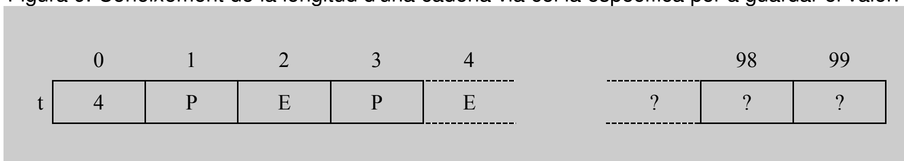
Figura 8. Coneixement de la longitud d'una cadena via marca pel final



2) Altres llenguatges fan servir una posició especial de la taula per a guardar la llargada de la tira de caràcters emmagatzemada. És el cas del llenguatge Pascal, que utilitza la posició zero per guardar-hi la longitud dels caràcters emmagatzemats a partir de la posició 1.

La figura 9 ens exemplifica aquest cas per una cadena amb capacitat per a 100 caràcters on hem introduït PEPE.

Figura 9. Coneixement de la longitud d'una cadena via cel·la específica per a guardar el valor.



Evidentment, en aquest darrer cas, la posició que emmagatzema la longitud és una posició especial, ja que conté un valor numèric, la qual cosa pot entendre's com una contradicció amb el fet que totes les posicions d'una taula són del mateix tipus. Aquesta contradicció no és

tal, ja que el tipus cadena és un tipus específic, que podem entendre com una taula, tot i que en realitat té la seva pròpia gestió.

Ja hem comentat que el llenguatge C fa servir un caràcter especial de sentinella i que en pseudocodi considerarem el mateix sistema. Aquest caràcter és el '\x0' (o simplement '\0'), és a dir, el caràcter que ocupa el lloc zero en la taula ASCII.

2.1. Cadenes en pseudocodi

> Una cadena en pseudocodi és una taula de caràcters on hem de garantir l'existència d'una marca de final que acordem que sigui '\0'.

De les cadenes en pseudocodi n'hem de conèixer: com declarar-les, de quins operados disposem i quines instruccions facilita el pseudocodi per a la seva gestió:

Declaració de variables de tipus cadena

```
var nom: cadena[MAX];
```

La constant `MAX` defineix la llargada de la cadena comptant l'espai destinat a la sentinella. Per tant, si volem garantir un espai real de 10 caràcters haurem de declarar `cadena[11]`.

Una variable de tipus cadena admet el tractament com a taula de caràcters. En aquest cas haurem d'anar amb compte amb la gestió de la sentinella.

Una constant de tipus cadena consisteix en una tira de caràcters limitada amb les dobles cometes com, per exemple, `"Hola"`. Cal tenir present que és totalment diferent `'H'` que `"H"`. En el primer cas tenim una constant `caràcter`, mentre que en el segon tenim una constant `cadena` (que en realitat ocupa dos caràcters: la lletra `'H'` i la sentinella). (!!)

Donada una variable declarada com a taula de caràcters, permetrem la utilització d'operacions sobre cadenes sempre que hi haguem posat la sentinella. (!!)

La cadena buida és aquella cadena que no té cap caràcter, és a dir, la que a la posició zero de la taula, hi té la sentinella. (!!)

Operadors sobre cadenes

– Assignació: =

– Relacionals: <, ≤, >, ≥, ==, !=

Què s'entén per comparació de cadenes? Com es comparen? Aquí intervé el concepte d'ordre lexicogràfic.

– Concatenació: +

Ho definim de la manera següent:

> S'entén per concatenació de cadenes el fet d'aconseguir, a partir de dues cadenes, una cadena nova construïda amb tots els caràcters de la primera (en el mateix ordre i sense la marca '\0') seguits de tots els caràcters de la segona (en el mateix ordre i amb la marca '\0').

És a dir, donades les declaracions:

```
part1, part2: cadena [41];
resultat: cadena[81];
```

i les assignacions següents:

```
part1 = "Plou i fa sol,";
part2 = " les bruixes es pentinen.";
resultat = part1 + part2;
```

la variable `resultat` conté la cadena `"Plou i fa sol, les bruixes es pentinen"`.

Instruccions sobre cadenes

– De lectura: **`llegir`**`(x);`
/* `x` és una variable de tipus `cadena` si es vol llegir una cadena */

– D'escriptura: **`escriure`**`(x);`
/* `x` és variable o constant `cadena` si es vol escriure una cadena */

– De càlcul de la seva longitud: **`loncad`**`(x);`
/* `x` és variable o constant `cadena` */

Es tracta d'una funció que, donada una cadena **x**, en retorna la longitud.

> Cal entendre longitud d'una cadena com el nombre de caràcters que la formen des de la posició zero fins al caràcter anterior a la sentinella. No s'ha de confondre la longitud de la cadena amb la capacitat de la cadena.

Què succeeix quan una cadena s'intenta omplir amb més caràcters que la capacitat que té? En pseudocodi considerarem que la part que no hi cap es perd. Ara bé, el funcionament del tipus cadena pot diferir en els diferents llenguatges. Sempre que s'ha de codificar en un determinat llenguatge cal conèixer-ne el funcionament referent a les estructures de control i als tipus de dades permesos.

2.2. Ordre lexicogràfic

Sembla que tots tenim clar quan dues cadenes són iguals i quan no ho són. Així, `"HOLA"` i `"CASA"` són diferents tot i que tinguin la mateixa longitud. Però, què ens dirà l'ordinador quan es troba comparacions com les següents?:

```
"HOLA" > "Pep"
"Joanna" <= "ANNA"
"CASA" <= "CASAL"
```

L'ordinador aplica l'ordre lexicogràfic per a saber quin és l'ordre relatiu entre dues cadenes i poder, així, respondre a les comparacions entre aquestes.

> L'ordre lexicogràfic entre cadenes consisteix a comparar els caràcters de les dues cadenes, posició per posició, i des de l'inici de les cadenes fins que una de les dues s'acabi o els caràcters siguin diferents o totes dues s'acabin.
>
> En el cas que el procés s'acabi a causa de la fi d'una cadena, aquesta és menor que l'altra.
>
> En el cas que s'acabi a causa de dos caràcters diferents, la cadena que conté el caràcter de posició més baixa a la taula ASCII és la cadena menor.
> Si totes dues arriben a la fi, vol dir que les cadenes són iguals.

És a dir, podríem dissenyar l'algorisme que ens ha de servir per a saber quina de les dues cadenes és la menor o si són iguals. Així, si `cad1` i `cad2` són cadenes tindríem:

```
var i: natural; fivar
...
i = 0;
mentre cad1[i] == cad2[i] i cad2[i] != '\0' fer i = i + 1;
fimentre
/* anem comparant mentre els caràcters siguin iguals i no
   s'arribi a la fi de les dues cadenes; fixeu-vos que la fi la
   controlem amb una de les dues cadenes, però no cal amb les
   dues, ja que si una arribés i l'altra no, ja donaria falsa
   la primera comparació relativa a si els dos caràcters són
   iguals */
si cad1[i] == cad2[i]
   llavors /* segur que tots dos han arribat a la sentinella */
      SÓN IGUALS
   sinó /* cal esbrinar quina és menor */
      opció
         cas cad1[i] =='\0': cad1 ÉS MENOR que cad2; [1]
         cas cad2[i] =='\0': cad2 ÉS MENOR que cad1; [2]
         cas cad1[i] < cad2[i]: cad1 ÉS MENOR que cad2; [3]
         cas cad1[i] > cad2[i]: cad2 ÉS MENOR que cad1; [4]
      fiopció
fisi
```

Observeu que [1] i [2] són innecessaris perquè queden coberts per [3] i [4], ja que el caràcter '\0' (marca de final) és menor que qualsevol caràcter de la taula ASCII (ja que ocupa el lloc 0). (!!)

2.3. Cadenes en llenguatge C

En aquest apartat estudiarem la declaració de variables i l'escriptura i lectura de cadenes en llenguatge C.

2.3.1. Declaració de variables

És idèntica a la declaració d'una taula de caràcters. Cal fer la reserva de l'espai per la sentinella, que s'acostuma a anomenar caràcter nul.

D'igual manera que amb les taules genèriques, es pot inicialitzar en el moment de fer la seva declaració, de la manera següent:

```
char cad[5] = { 'A', 'B', 'C', 'D', '\0'};
```

o també:

```
char cad[5] = "ABCD";
```

o també:

```
char cad[] = "ABCD";
```

o també:

```
char cad[5] = { 65, 66, 67, 68, 0 };
```

Una constant cadena en C és, igual que en pseudocodi, una tira de caràcters limitada entre cometes dobles.

2.3.2. Escriptura de cadenes

Fins ara ens hem vist obligats a escriure una cadena posició a posició, amb la funció `printf` utilitzant l'especificació de format `%c` per a cada caràcter.

La funció `printf` disposa d'una altra especificació de format per a les cadenes: `%s`, que provoca l'escriptura de la cadena des de la posició zero fins a la posició que conté la marca `'\0'`, la qual no és escrita.

Així, per a escriure la cadena `cad` anterior mitjançant la funció `printf`:

```
printf ("%s\n", cad);
```

La utilització de l'especificació de format permet definir l'amplada de la sortida i la justificació esquerra o dreta. Així,

```
printf ("%-10s", cad);
```

escriurà la cadena `"ABCD"`, amb una amplada de 10 caràcters i justificant per l'esquerra. Si no es posa el símbol `'-'` o es posa `'+'`, es justifica per la dreta.

> A l'apartat "Instruccions de sortida en llenguatge C" de la unitat didàctica "Introducció a la programació", s'introdueixen les especificacions de format possibles per a la instrucció `printf`.

Hi ha, però, altres maneres d'escriure cadenes en la sortida estàndard: la funció `puts` que no necessita cap especificació de format. Tindríem:

```
puts (cad);
```

2.3.3. Lectura de cadenes

Fins ara, per a llegir una cadena, no hem tingut cap altra solució que actuar caràcter a caràcter. La funció `scanf` també ens permet llegir

cadenes amb la corresponent especificació de format `%s`. Però, atenció, el funcionament canvia. (!!)

Suposeu que teniu una variable `c` de tipus caràcter i voleu efectuar la lectura per teclat per tal d'omplir `c`. Oi que faríeu el següent?:

```
scanf ("%c", &c);
```

Doncs ara suposeu que teniu una variable `s` de tipus cadena (grandària suficient) i voleu llegir per teclat el valor per a omplir `s`. No heu de posar `&s` sinó directament `s`. No és ara el moment d'explicar per què no es posa el símbol `&` quan es tracta d'una cadena i en canvi sí que es posa quan es tracta de qualsevol altre tipus de dada. Creieu-vos-ho. O sigui:

```
scanf ("%s", s);
```

Seguint amb el mateix exemple, suposeu que l'usuari escriu per teclat el text:

```
Això és C
```

Què haurà llegit el programa a la variable `s`? Fem que l'escrigui amb el corresponent `printf`:

```
printf ("%s", s);
```

En pantalla ens apareix:

```
Això
```

Què ha passat amb la resta del text? Què està passant? Recordeu que la funció `scanf` llegeix dades delimitades per espais en blanc. Per tant, llegeix una paraula i la resta queda en la memòria intermèdia (buffer) del teclat a l'espera de la lectura següent.

Espais en blanc eren: ' ' (espai en blanc), '\t' (conjunt de símbols que C interpreta com un tabulador), '\n' (conjunt de símbols que C interpreta com un salt de línia, un <return> o un <intro>.

Per solucionar aquest problema, `scanf` admet una especificació de format personalitzada que té la sintaxi següent:

```
%[conjunt_caràcters]
```

que provoca la lectura de caràcters fins que n'arribi un que no estigui dins el conjunt de caràcters, cosa que fa que s'acabi la lectura, i també:

```
%[^conjunt_caràcters]
```

que és l'invers i provoca la lectura de caràcters fins que apareix un dels caràcters del conjunt.

Així, per al cas anterior, si es desitja que la lectura s'efectuï fins que l'usuari premi <return>, caldrà escriure:

```
scanf ("%[^\n]", s);
```

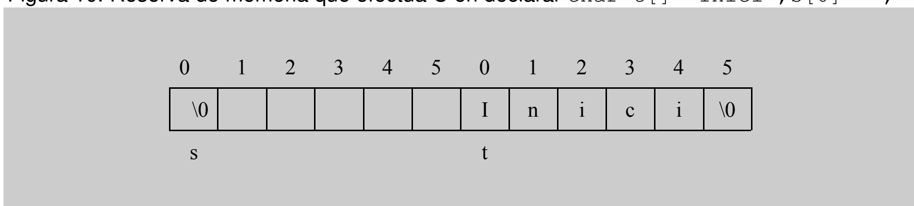
Què passa si l'espai assignat a la variable és insuficient per a encabir tot el que arriba per l'entrada estàndard? Sí, és el que us imagineu. El llenguatge C fa el mateix tractament que en les taules quan ens sortim de l'espai reservat: ocupa les posicions de memòria següents, la qual cosa quasi segur que provocarà errors, els quals es poden traduir en l'avortament de l'execució del programa o en l'obtenció de resultats no esperats. Així, observeu l'exemple següent:

```
/* u3n2p02.c */

#include <stdio.h>
#include <conio.h>

void main(void)
{
   char t[]="Inici",s[6]="";
   clrscr();
   printf ("Valor inicial de s: %s\n",s);
   printf ("Valor inicial de t: %s\n\n",t);
   printf ("Entreu un text d'entre 6 i 11 caràcters:");
   scanf ("%[^\n]",s);
   printf ("\n\nValor final de s: %s\n",s);
   printf ("Valor final de t: %s\n\n",t);
}
```

Trobareu l'arxiu `u3n2p02.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

La figura 10 mostra la reserva de memòria que efectua el programa obtingut amb *Turbo C++*. Sí, no hi ha cap error... La reserva de memòria s'efectua en ordre invers a la declaració de les variables. Així, la declaració char t[]="Inici",s[6]=""; provoca la reserva d'espai per a la variable s seguida de la reserva d'espai per a la variable t. En altres plataformes pot no ser exactament igual que com aquí es presenta, però la problemàtica que segueix es pot produir de forma molt similar.

Figura 10. Reserva de memòria que efectua C en declarar `char t[]="Inici",s[6]="";`

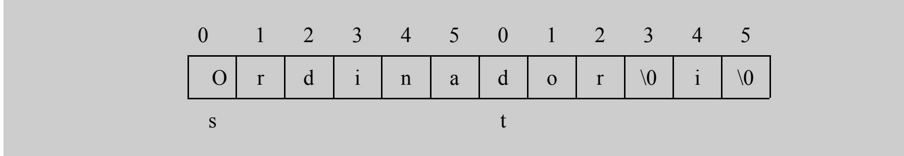


Suposem que l'usuari fa cas del que el programa demana (un text entre sis i onze caràcters) i introdueix:

```
Ordinador
```

Aquesta paraula queda emmagatzemada a partir de la primera posició de la cadena `s`. Com que té més caràcters que espais reservats a la cadena `s`, s'emmagatzema també a les posicions contigües que corresponen a la cadena `t`, i s'obté la situació que mostra la figura 11.

Figura 11. El text emmagatzemat a la variable s matxaca el text emmagatzemat a la variable t.



0	1	2	3	4	5	0	1	2	3	4	5
O	r	d	i	n	a	d	o	r	\0	i	\0
s						t					

Fixeu-vos a la figura 11 que el text emmagatzemat està acompanyat de la sentinella corresponent, que queda a la posició 3 de la taula `t`.

L'execució del programa, què treu per pantalla?

```
Valor inicial de s:
Valor inicial de t: Inici

Entreu un text d'entre 6 i 11 caràcters: Ordinador

Valor final de s: Ordinador
Valor final de t: dor
```

Veieu les causes d'aquest comportament, oi? En primer lloc, en pintar el valor final de la cadena `s`, el programa escriu tot el text que troba a partir de la primera posició de la cadena i fins que troba un caràcter `'\0'`. Per això surt tota la paraula "Ordinador". En segon lloc, en pintar el valor final de la cadena `t`, el programa escriu tot el text que troba a partir de la primera posició de la cadena que, en aquest cas, correspon a "dor", ja que després de la lletra `'r'` troba el caràcter `'\0'`. En un tros de la cadena `t` encara hi ha caràcters de la inicialització, però no es veuen perquè estan després d'un caràcter `'\0'`. A tots els efectes, la variable `t` està emmagatzemant, com a cadena, el text `"dor"`, mentre que com a taula de caràcters està emmagatzemant els caràcters `'d'`, `'o'`, `'r'`, `'\0'`, `'i'`, `'\0'`.

Fixeu-vos també que en aquest programa s'ha reservat dotze bytes, per la qual cosa s'ha demanat a l'usuari que introdueixi a tot estirar onze caràcters (cal pensar en el `\0` afegit). Penseu que si l'usuari introdueix més d'onze caràcters, el programa intenta actuar de la mateixa manera i, per tant, pot estar enregistrant caràcters en posicions de memòria destinades al propi funcionament del programa. Tant pot funcionar com pot quedar tot "penjat". Dependrà d'on vagin a parar els caràcters llegits sense espai reservat.

Potser us pregunteu si hi ha alguna manera de limitar la quantitat de caràcters llegits per la funció `scanf`. En efecte, recordeu com és una especificació de format?

```
%[*][<ample>][h|l]<tipus>
```

Per tant, en aquest cas podríem aconseguir que la lectura de la cadena `s` quedés limitada a cinc caràcters posant:

```
scanf ("%5s", s);
```

Recordeu que en cas d'actuar d'aquesta manera poden quedar caràcters pendents de tractament en la memòria intermèdia (buffer) del teclat, els quals seran tractats a la propera lectura. En cas de voler-lo eliminar ja coneixem la funció `neteja_stdin()` que hem implementat per a diferents plataformes.

A la solució anterior, però, si l'usuari vol introduir com a text les cinc vocals separades amb un espai en blanc (és a dir, text de llargada entre 6 i 11 caràcters que inclou espais en blanc), només en llegirà la primera, ja que l'espai en blanc fa que s'acabi l'entrada de dades. Podem barrejar la solució referent a la màxima longitud d'entrada amb la lectura fins que aparegui un determinat caràcter. Així ens quedaria:

```
scanf ("%5[^\n]", s);
```

Però la funció `scanf` no és l'única manera d'efectuar una lectura de cadena de caràcters per a l'entrada estàndard. Disposem de la funció `gets`, similar a la funció `puts`.

```
gets (cad);
```

La funcionalitat d'aquesta funció rau en la capacitat d'emmagatzemar a partir de la primera posició de la cadena `cad` els caràcters que arriben per l'entrada estàndard fins a l'arribada del caràcter `'\n'` (que es produeix amb <return>). Aquesta funció, a diferència de la funció `scanf`, permet l'entrada d'una cadena de caràcters formada per diverses paraules separades per espais en blanc, sense cap tipus de format.

2.3.4. Funcions de tractament de cadenes

El llenguatge C no disposa d'operadors sobre cadena.
És a dir, en C no podem fer servir l'operador d'assignació per a assignar valor a una cadena (excepte en la seva declaració), ni els operadors

relacionals (<, >, ==, !=, ≤, ≥) per comparar cadenes, ni cap operador de concatenació, com tenen molts llenguatges.

Com s'aconsegueix, doncs, l'assignació, la comparació i la concatenació? Amb unes funcions especials que aporta el llenguatge C. En realitat, el llenguatge proporciona un ampli nombre de funcions per tractament de cadenes. A continuació s'anomenen les necessàries per a poder tenir la mateixa funcionalitat que ens proporciona el pseudocodi. Tingueu present que encara no hem abordat el tema de les funcions i, per tant, en aquesta presentació farem servir una terminologia entenedora, amb conceptes explicats fins al moment, però que no es correspon amb la terminologia real, que és la que trobareu en qualsevol manual del llenguatge C.

Càlcul de longitud

```
strlen (cadena);
```

Retorna la longitud, en caràcters, de la cadena, sense incloure el caràcter nul.

Assignació

```
strcpy (cadena_destí, cadena_origen);
```

Copia `cadena_origen` en `cadena_destí` incloent-hi la sentinella.

```
strncpy (cadena_destí, cadena_origen, llarg);
```

Copia `llarg` (nombre natural) caràcters de la `cadena_origen` en `cadena_destí` (sobreescrivint els caràcters de `cadena_destí`). Si `llarg` és menor que la longitud de `cadena_origen`, no s'afegeix automàticament la sentinella a `cadena_destí`. Si `llarg` és major que la longitud de `cadena_origen`, llavors `cadena_destí` s'omple amb caràcters nuls (`'\0'`) fins a la longitud `llarg`.

Concatenació

```
strcat (cadena_destí, cadena_origen);
```

Afegeix `cadena_destí` en `cadena_origen`, i acaba la cadena resultant amb la sentinella.

```
strncat (cadena_destí, cadena_origen, llarg);
```

Afegeix els primers `llarg` caràcters de `cadena_origen` en `cadena_destí` i acaba la cadena resultant amb el caràcter nul. Si `llarg`

és major que la longitud de `cadena_origen`, s'utilitza com a valor de `llarg` la longitud de `cadena_origen`.

Comparació

```
strcmp (cadena1, cadena2);
```

Compara la `cadena1` amb la `cadena2` lexicogràficament i retorna un valor `int`:

< 0 si `cadena1` és menor que `cadena2`,
= 0 si `cadena1` i `cadena2` són iguals,
> 0 si `cadena1` és major que `cadena2`.

```
strncmp (cadena1, cadena2, llarg);
```

Compara lexicogràficament els primers `llarg` caràcters de `cadena1` i `cadena2` i retorna un valor enter de la mateixa manera que la funció `strcmp`. Si `llarg` és major que la longitud de `cadena1`, es pren com a valor `llarg` la longitud de `cadena1`.

2.4. Exemples de manipulació de cadenes

Fer un programa, fent servir el tractament de cadenes, que demani a l'usuari el seu nom i que posteriorment tregui per pantalla un missatge personalitzat de salutació.

Algorisme en pseudocodi:

```
programa benvinguda és
   const MAX = 81;
   ficonst
   var  nom: taula [MAX] de caràcter;
   fivar
   netejar_pantalla;
   escriure ("Entri el seu nom, si li plau: ");
   llegir (nom);
   netejar_buffer_teclat;
   saltar_línia;
   saltar_línia;
   escriure ("Benvingut ", nom);
fiprograma
```

Codificació en llenguatge C:

```
/* u3n2p03.c */

#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* Per compatibilitat entre plataformes */

#define MAX 81

void main(void)
{
   char nom[MAX];
   clrscr();
   printf ("Entri el seu nom, si li plau: ");
   scanf ("%80[^\n]",nom);
   neteja_stdin();
   printf ("\n\nBenvingut %s",nom);
}
```

Trobareu l’arxiu `u3n2p03.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d’aquest crèdit.

3. Tuples i tipus de dades

Sabem que el tipus de dada taula permet agrupar en una sola variable un conjunt de variables del mateix tipus i que el tipus de dada cadena no és altra cosa que un tipus especial de taula que té funcionalitats potents per a la gestió de tires alfanumèriques.

Ens manca, però, la manera de poder agrupar en una sola variable un conjunt de variables de tipus diferent o igual però que fan referència a un mateix concepte. Suposem que volem tractar diferents dades d'una persona com, per exemple: DNI, data de naixement, pes, alçada, nom i cognoms, adreça, sexe, etc. Potser penseu que algunes d'aquestes dades caldrà declarar-les com a cadena, d'altres com a caràcter, algunes com a real, etc. Però si totes fan referència a una mateixa persona, ens pot interessar poder-les tenir agrupades sota un mateix nom.

> Una tupla és un conjunt finit de dades d'igual o diferent tipus que estan col·locades consecutivament en la memòria de l'ordinador, són reconegudes per un nom en comú (tupla) i s'hi accedeix pel nom de les diferents dades que la formen.

Per tant, per a definir tupla caldrà definir tots els elements que la componen donant-ne per a cada un el nom i el tipus. Això fa que definir tupla sigui una feina tediosa i feixuga que, a més, potser cal repetir en diferents llocs del programa. Aquest fet no passa en les taules, ja que per a definir una taula d'una capacitat i un contingut determinats només necessitem una línia de codi.

Per aquest motiu, els llenguatges de programació acostumen a permetre la declaració de tipus nous de dades per part del programador, de manera que una vegada definits es poden declarar variables de tots els tipus coneguts pel programa (els que ja aporta el llenguatge més els definits pel programador). Aquesta tècnica permet, en el cas de tuples, definir un tipus de dada per cada tipus de tupla amb el qual s'ha de treballar per, posteriorment, declarar variables del tipus definit.

3.1. Tuples i tipus de dades en pseudocodi

La declaració d'una variable tupla en pseudocodi segueix la sintaxi següent:

```
var  <nom>:  tupla
                <camp_1>: <tipus_X>;
                <camp_2>: <tipus_X>;
                ...
                <camp_n>: <tipus_X>;
             fitupla
fivar
```

on:

- `<nom>` és el nom de la variable,
- `<camp_X>` és el nom de cada apartat que forma la tupla,
- `<tipus_X>` és el tipus de dada que pot emmagatzemar cada apartat de la tupla.

En molts llenguatges no s'utilitza el terme tupla sinó el de registre, motivat per la forta associació que hi ha entre el concepte de tupla (variable en memòria) i el concepte de registre en arxius (informació emmagatzemada en arxius). Entendreu aquesta associació quan estudieu els sistemes gestors d'arxius. (!!)

El terme anglès que s'acostuma a utilitzar per a referir-se a les tuples és record.

Pensem que hem de gestionar persones. Podríem efectuar la declaració:

```
p: tupla
      dni: cadena[10];
      nom, cognom1, cognom2: cadena[16];
      pes, altura: real;
      edat: natural;
      sexe: caràcter;
   fitupla
```

Utilitzem el mot "altura" per referir-nos a l'alçada d'una persona quan en realitat hauríem d'utilitzar "alçada", per no tenir variables amb la 'ç' formant part del nom, de la mateixa manera que és convenient no posar accents ni 'ñ'.

Gràficament, la declaració anterior, en iniciar-se l'execució del programa, produeix en la memòria de l'ordinador la reserva d'espai similar al que mostra la figura 12.

Figura 12. Reserva de memòria en declarar una tupla.

	DNI	nom	cognom1	cognom2	pes	altura.	edat	sexe
p	valor	valor	valor	valor	valor	valor	valor	valor

Cada apartat ocupa el que li correspon en funció del tipus de dada amb què ha estat definit. Així, `DNI` ocupa deu caràcters perquè és una `cadena[10]`, `nom`, `cognom1` i `cognom2` ocupen setze caràcters perquè són `cadena[16]`, `pes` i `altura` ocupen l'espai que correspon a un real, i així successivament.

En la declaració anterior s'ha definit una única variable de nom `p`. S'hauria pogut aprofitar per a definir diverses variables d'igual manera

que en els tipus de dades que aporta el llenguatge. És a dir, si s'hagués volgut definir tres variables, p1, p2 i p3, hauríem fet:

```
p1, p2, p3:  tupla
                ...
             fitupla
```

És a dir, podem definir una o diverses variables per a gestionar persones. Però és possible que necessitem en algun altre lloc del programa una o més noves variables de les mateixes característiques. Cal tornar a fer la declaració? Hem de repetir totes aquelles línies de codi? Això comença a ser feixuc. A més, quan una mateixa declaració s'ha de repetir a diversos llocs facilita l'aparició d'errors que serien fàcilment evitables si la declaració es fes una única vegada. Per aquesta raó és molt interessant la possibilitat de definir tipus nous de dades.

En pseudocodi es pot definir tipus de dades amb la sintaxi següent:

```
tipus   <nom_1> = <declaració_tipus>;
        <nom_2> = <declaració_tipus>;
        ...
        <nom_n> = <declaració_tipus>;
fitipus
```

on:

- <nom_X> és el nom del tipus nou de dada de la qual posteriorment es podran declarar variables;

- <declaració_tipus> és la declaració de cada tipus de la mateixa manera que es definirien les variables que estem tipificant.

Cal tenir present que la declaració de tipus no provoca reserva de memòria per cap variable. És únicament una definició que ens permetrà, quan ens interessi, declarar-ne variables. Es pot fer servir la definició de tipus de dades per a definicions de taules, tuples i, en general, qualsevol tipus de dada que hi hagi en el llenguatge.

La declaració de tipus cal situar-la abans de la declaració de les variables (ja que la declaració de variables es pot basar en els tipus definits pel programador) i després de la declaració de les constants (ja que la declaració de tipus es pot basar en les constants definides).
Així doncs, podríem efectuar les declaracions següents:

```
tipus persona = tupla
                   dni: cadena[10];
                   nom, cognom1, cognom2: cadena[16];
                   pes, altura: real;
```

```
                  edat: natural;
                  sexe: caràcter;
                fitupla
     tpersona = taula[MAX] de persona;
fitipus
var  p1, p2, p3: persona;
     tp1, tp2: tpersona;
fivar
```

En aquest exemple podeu veure com s'ha definit el tipus `persona` i, posteriorment, el tipus `tpersona` (taula de persones). És a dir, quan un tipus ja està definit es pot fer servir per a declarar variables i altres tipus de dades. Aquestes declaracions de tipus no han provocat cap reserva de memòria per a emmagatzemar dades.

En el mateix exemple hem definit cinc variables, tres de les quals s'han basat en el tipus `persona` i les altres dues, en el tipus `tpersona`. Per tant, en memòria tindrem espai reservat per a tres persones, al qual s'accedirà pels noms `p1`, `p2` i `p3`, i espai per a dues taules (`tp1` i `tp2`) de `MAX` persones cadascuna.

Bé, ja sabem declarar tuples i tipus. Ens manca saber com es pot accedir als diferents camps o apartats que componen una variable tupla. L'accés als apartats que formen una tupla és tan senzill com:

```
<nom_variable>.<nom_apartat>
```

És a dir, si volem gestionar les dades d'una persona com les de més amunt, podem fer actuacions semblants a:

```
var  p: persona;
fivar
...
escriure ("Introdueixi el dni: "); llegir (p.dni);
escriure ("Introdueixi el nom: "); llegir (p.nom);
...
p.edat = 20; p.pes = 68; p.altura = 1.80;
...
escriure ("La persona de dni ", p.dni, " que s'anomena ",\
          p.nom, " ", p.cognom1, " i ", p.cognom2,\
          " té ", p.edat, " anys.");
...
```

Quines operacions estan permeses pel que fa a tuples? Això depèn del llenguatge, però en pseudocodi considerarem que és possible assignar una tupla a una altra, sempre que siguin del mateix tipus. A part d'aquesta facilitat, la resta d'operacions caldrà fer-les apartat per apartat, seguint les regles del joc establertes per a cada tipus de dada.

3.2. Tuples i tipus de dades en llenguatge C

Tots els conceptes referents a tuples i tipus de dades introduïts en pseudocodi són vàlids en el llenguatge C, llevat que en C és obligatori haver definit el tipus de dada per a poder definir-ne variables.

El terme equivalent a `tupla` que utilitza el llenguatge C és el d'`estructura`. La sintaxi que es fa servir per a definir una estructura és la següent:

```
struct <nom_tipus_estructura>
{
   /* declaració dels apartats */
} [llista_variables];
```

on:

- `<nom_tipus_estructura>` és el nom del tipus de dada que s'ha definit i pel qual es podran declarar variables quan el programador ho cregui convenient;
- `/* declaració dels apartats */` correspon a les declaracions de tots els camps que formen la tupla, amb la mateixa sintaxi que es faria servir per a definir-los com a variables; en C s'acostumen a anomenar membres de l'estructura;
- `[llista_variables]` és optatiu i permet declarar variables del tipus estructura declarat en el mateix moment en què es declara l'estructura.

Per a declarar variables del tipus estructura no declarades en la definició de l'estructura, cal seguir la sintaxi següent:

```
struct <nom_tipus_estructura> [variable[, variable]...];
```

Si codifiquem l'exemple "persona" que havíem vist en pseudocodi, tindrem:

```
struct persona
{
   char dni[10], nom[16], cognom1[16], cognom2[16];
   float pes, alçada;
   unsigned int edat;
   char sexe;
} p1, p2, p3, tp1[MAX], tp2[MAX];
```

En la mateixa definició de l'estructura hem declarat cinc variables. Si en algun altre lloc necessitem declarar més variables, faríem:

```
struct persona p4, p5, tp3[MAX], tp4[MAX];
```

Coneixem de l'existència en el llenguatge C de la possibilitat de redefinir els tipus de dades amb la sentència `typedef`. També la podem fer servir per a redefinir estructures creades pel programador. Així, podríem tenir:

A l'apartat "Redefinició de tipus de dades en C" de la unitat didàctica "Introducció a la programació" s'introduia la possibilitat de redefinir tipus de dades en el llenguatge C.

```
typedef struct per
{
   char dni[10];
   char nom[16], cognom1[16], cognom2[16];
   float pes, altura;
   unsigned int edat;
   char sexe;
} persona;
```

Amb la definició anterior, `persona` és un tipus de dada redefinició de l'`struct per`. Amb aquesta definició podríem fer el següent:

```
persona p1, p2, p3, pt1[MAX], pt2[MAX];
```

També es pot aconseguir la mateixa funcionalitat amb dos passos:

```
struct per
{
   char dni[10];
   char nom[16], cognom1[16], cognom2[16];
   float pes, altura;
   unsigned int edat;
   char sexe;
};
typedef struct per persona;
```

Quan es disposa d'una estructura amb una redefinició, es pot declarar variables utilitzant l'estructura o la redefinició. Us aconsellem que actueu sempre de la mateixa manera per a evitar confusions.

També podem declarar un nou tipus basat en una taula de tipus ja existents. Així, podem declarar el tipus `tpersona` per referir-nos a una taula de persones, fent:

```
typedef struct persona tpersona[MAX];
```

i, posteriorment, ja podríem declarar variables com:

```
tpersona tp1, tp2;
```

on `tp1` i `tp2` serien taules de `MAX` elements de tipus `persona`.

Per a accedir a un membre d'una variable de tipus estructura se segueix la mateixa nomenclatura que en pseudocodi:

```
<nom_variable>.<nom_membre>
```

i el tractament que correspon donar a cada membre és el mateix que caldria donar si es tractés d'una variable, tenint en compte, doncs, el tipus de dada de cada membre. És a dir, els membres que siguin `float` cal tractar-los com a `float`, els membres que siguin `cadenes de C`, com a `cadenes de C` i així amb tots els tipus possibles de membres.

Quines són les operacions que es poden efectuar amb les variables de tipus estructura?

1) Inicialització en el moment de la seva declaració.

```
struct persona p={"12345678B", "Josep", "Feliu", "Bordiu", 75, 1.80, 28, 'H'};
```

Cal tenir present l'ordre en què han estat declarats els membres dins l'estructura (`DNI, nom, cognom1, cognom2, pes, altura, edat, sexe`).

2) Assignació d'una variable estructura a una altra del mateix tipus amb l'operador d'assignació '='.

```
struct persona p1, p2;
...
/* suposem p1 plena amb dades */
p2 = p1;
/* acabem d'omplir p2 amb una còpia de les dades de p1 */
```

3.3. *Unions* en llenguatge C

Una `union` és una variable semblant a una `struct` que pot contenir, en diferents instants de l'execució del programa, dades de tipus diferents. Això permet manipular tipus de dades diferents utilitzant una mateixa zona de memòria, la reservada per la variable en qüestió.

La declaració d'una `union` té la mateixa forma que la declaració d'una `struct`, llevat que en lloc de fer servir la paraula reservada `struct` s'utilitza la paraula reservada `union`. Per tant, tot el que s'ha exposat per a les `struct` és aplicable a les `union` amb l'excepció que dels membres especificats, en un moment d'execució, només un hi és present.

Exemple:

```
union dia
{
   unsigned int ndia;
   char sdia[15];
} d1, d2, d3;
```

En aquest exemple, `dia` és un tipus de dada que permet emmagatzemar una variable `unsigned int` (accedida pel nom `ndia`) i una variable taula de quinze caràcters (s'hi accedeix pel nom `sdia`), però únicament una de les dues variables, no les dues a la vegada.

Per tant, per a emmagatzemar els membres d'una `union` es necessita una zona de memòria igual a la que ocupa el membre més gran de la `union`. En l'exemple anterior l'espai reservat per cada una de les variables `d1`, `d2` i `d3` és de quinze bytes, que corresponen al membre `sdia` encara que en temps d'execució s'emmagatzemi en aquestes variables valors corresponents al membre `ndia` (2 o 4 bytes segons versió).

El valor emmagatzemat en una `union` és sobreescrit cada vegada que s'assigna un valor al mateix membre o a un membre diferent.