Organitzem programes i dades.

4

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

Índex

Introducció

La programació estructurada i modular utilitza quatre recursos bàsics:

- el disseny modular,
- l'organització de les dades mitjançant estructures de dades apropiades,
- l'ús de les estructures seqüencial, condicional i repetitiva,
- l'aproximació a la codificació mitjançant pseudocodi.

En les unitats didàctiques anteriors s'han treballat a fons els dos darrers recursos i deu n'hi do, també, de la feina feta respecte el segon recurs. En tots els casos s'han plantejat exemples senzills, l'algorisme de resolució dels quals es feia amb poques operacions.

En el nucli d'activitat "Programació modular" presentem els mecanismes que proporcionen els llenguatges estructurats per a efectuar un disseny modular de les aplicacions. Així doncs, coneixerem els conceptes de mòdul i de subprograma aplicables a la majoria de llenguatges i els durem a la pràctica en el llenguatge C.

En referència al concepte de mòdul podem introduir que les grans aplicacions es torcegen en mòduls cadascun dels quals té una funcionalitat determinada. Així, en una aplicació per a una gestió integral d'un centre docent podríem tenir el mòdul que gestiona el personal que treballa en el centre, el mòdul que gestiona l'alumnat, el mòdul que gestiona els horaris, i la llista podria anar-se ampliant. És clar que els mòduls tenen una autonomia però que necessiten mantenir relacions amb la resta de mòduls que formen l'aplicació.

En referència al concepte de subprograma estudiarem amb deteniment els conceptes d'acció i funció als que s'ha fet referència en alguns punts de les unitats didàctiques anteriors. En tots dos casos hem trepitjat el món de la programació modular sense tenir tots els coneixements. No ho hem fet malament però tampoc no s'ha dit tot el que pertocava. Ara ens disposem a veure-ho amb deteniment.

En el nucli d'activitat "Tipus abstractes de dades" presentem el mètode ideal a utilitzar en la programació estructurada i modular per a la definició dels tipus de dades que havíem iniciat en les anteriors unitats didàctiques. La programació estructurada i modular amb la utilització de tipus abstractes de dades és la precursora de la programació orientada a objectes. Un tipus abstracte de dades és una definició de tipus de dada

acompanyat del conjunt d'operacions definit per a gestionar les seves dades.

Per últim, en el nucli d'activitat "Gestió de dades ordenades en memòria interna" volem mostrar les possibilitats que hi ha per mantenir la informació ordenada en la memòria i les avantatges d'accés que això proporciona així com el cost que té.

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 fitxers 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. Fer servir les tècniques de la programació modular per a desenvolupar programes informàtics.

2. Analitzar la conveniència de dissenyar accions i funcions per a resoldre problemes.

3. Fer servir de manera adequada les variables locals i globals.

4. Aplicar correctament el pas de paràmetres per valor i per referència.

5. Declarar i definir funcions en llenguatge C.

6. Conèixer i aplicar les funcions d'utilització més freqüent que aporta el llenguatge C.

7. Adquirir els hàbits per a una organització modular eficient en la codificació de programes en llenguatge C.

8. Identificar els diferents algorismes de recerca i d'ordenació en memòria interna.

9. Aplicar, de manera eficient, els algorismes de recerca en taules ordenades.

10. Analitzar l'eficiència dels algorismes dissenyats.

11. Identificar la conveniència d'utilitzar algorismes d'ordenació estables.

1. Programació modular. Implementació en C.

Al començament de la dècada dels setanta es comença a parlar de programació modular com a metodologia de disseny i programació. Aquesta suposa la divisió del programa en parts menors (mòduls i/o subprogrames) que es tracten i es munten (enllacen) de manera independent. La programació modular és més un mètode de disseny que un mètode de programació.

Hi ha, però, dues maneres de dissenyar els diferents mòduls que formaran l'aplicació: la primera forma és el resultat d'una anàlisi descendent i la segona forma és el resultat d'una anàlisi ascendent.

Començarem, doncs, pels conceptes d'anàlisi descendent i ascendent, per passar posteriorment al disseny dels mòduls i/o subprogrames que apareixen en la fase de disseny i finalitzar amb la corresponent implementació en el llenguatge C.

1.1. Disseny ascendent-descendent

Primer de tot, cal aclarir els conceptes de disseny ascendent i disseny descendent.

El disseny descendent és la tècnica de programació modular més utilitzada i consisteix en la descomposició del problema general en problemes més simples, denominats subproblemes, els més complexos dels quals es tornen a descompondre en altres subproblemes. S'anomena descendent perquè partint del problema gran passa a problemes més petits als quals donarà solució.

Disseny descendent

Aquesta metodologia de treball és anomenada, en anglès, disseny top-down.

El disseny ascendent és una altra manera de resoldre el problema. En aquest cas es comença plantejant problemes bàsics per als quals se cerca la corresponent solució i, a continuació, se construeix el problema en la seva totalitat.

Disseny ascendent

Aquesta metodologia de treball es coneix també com a disseny bottom-up.

Nosaltres fem servir l'anàlisi descendent per a resoldre problemes complexos. D'aquesta manera aconseguim unitats de tractament elementals que podrem programar mitjançant les tècniques de la programació estructurada.

Cada una de les unitats ha de realitzar una tasca específica, i la unió o l'enllaç de totes les unitats permetrà d'obtenir l'aplicació desitjada.

Cal distingir dos tipus d'unitats: els mòduls i els subprogrames.

1.2. Subprogrames i mòduls

El concepte de subprograma s'utilitza normalment per a identificar un conjunt d'instruccions que conjuntament desenvolupen una tasca concreta i que pot ser (o no), necessària en diferents llocs del programa.

A cada lloc on sigui necessari la utilització d'un subprograma dissenyat, se n'efectuarà una crida.

Com a exemples de subprogrames podem considerar:

- El conjunt d'instruccions corresponent a controlar la lectura d'un caràcter introduït per l'usuari corresponent a la resposta `'S'` o `'N'` per aquelles preguntes en que l'usuari s'ha de veure obligat a respondre SÍ o NO.
- El conjunt d'instruccions per demanar a l'usuari la forma de pagament associada a una transacció comercial, situació que pot aparèixer moltes vegades en una aplicació comercial.
- El conjunt d'instruccions per a visualitzar per pantalla la fitxa d'un client, que també pot aparèixer en diferents llocs en una aplicació comercial.

El concepte de mòdul es destina normalment a identificar grans apartats d'una aplicació.

Així, en una aplicació comercial podem distingir, com a mínim, els següents mòduls:

- Mòdul de compres (amb gestió de comandes a proveïdors, recepció de mercaderies i comprovació de les factures de proveïdors)
- Mòdul de vendes (amb gestió d'ofertes a clients, comandes de clients, lliurament de mercaderies i facturació)
- Mòdul de comissions (amb gestió de les comissions que els pertoca als comercials en funció de les vendes)

- Mòdul d'estadístiques (amb tot tipus d'estadístiques de compra i de venda)
- Mòdul de magatzem (amb tota la gestió d'estucatge dels articles)

S'aprecia que cadascun d'aquests mòduls pot estar compost, a la seva vegada, d'altres mòduls; tot depèn de la magnitud que l'analista els hi vulgui donar.

La implementació dels mòduls i dels subprogrames també acostuma a anar per camins diferents. No hi ha una única forma d'actuació, però s'acostuma a seguir els següents criteris:

- Un mòdul pot estar implementat en 1 o varis fitxers font.
- Un fitxer font pot contenir 1 o varis subprogrames. En cas de tenir varis subprogrames acostumen a ser d'una temàtica comuna.
- No és una bona pràctica incloure en un fitxer font subprogrames de diferents mòduls.

Així, per exemple, pensant en una aplicació comercial desenvolupada en C, podríem tenir:

- Mòdul de compres, format per diferents fitxers font:
  - Fitxer `datpro.c`, amb tots els subprogrames per a la gestió de proveïdors.
  - Fitxer `compro.c`, amb tots els subprogrames per a la gestió de les comandes a proveïdors.
  - Fitxer `albpro.c`, amb tots els subprogrames per a la gestió dels albarans de proveïdors.
  - Fitxer `facpro.c`, amb tots els subprogrames per a la gestió de les factures de proveïdors.

- Mòdul de vendes, format per diferents fitxers font:
  - Fitxer `datcli.c`, amb tots els subprogrames per a la gestió de clients.
  - Fitxer `ofecli.c`, amb tots els subprogrames per a la gestió de les ofertes a clients.
  - Fitxer `comcli.c`, amb tots els subprogrames per a la gestió de les comandes de clients.
  - Fitxer `albcli.c`, amb tots els subprogrames per a la gestió dels albarans a clients.
  - Fitxer `faccli.c`, amb tots els subprogrames per a la gestió de les factures de clients.

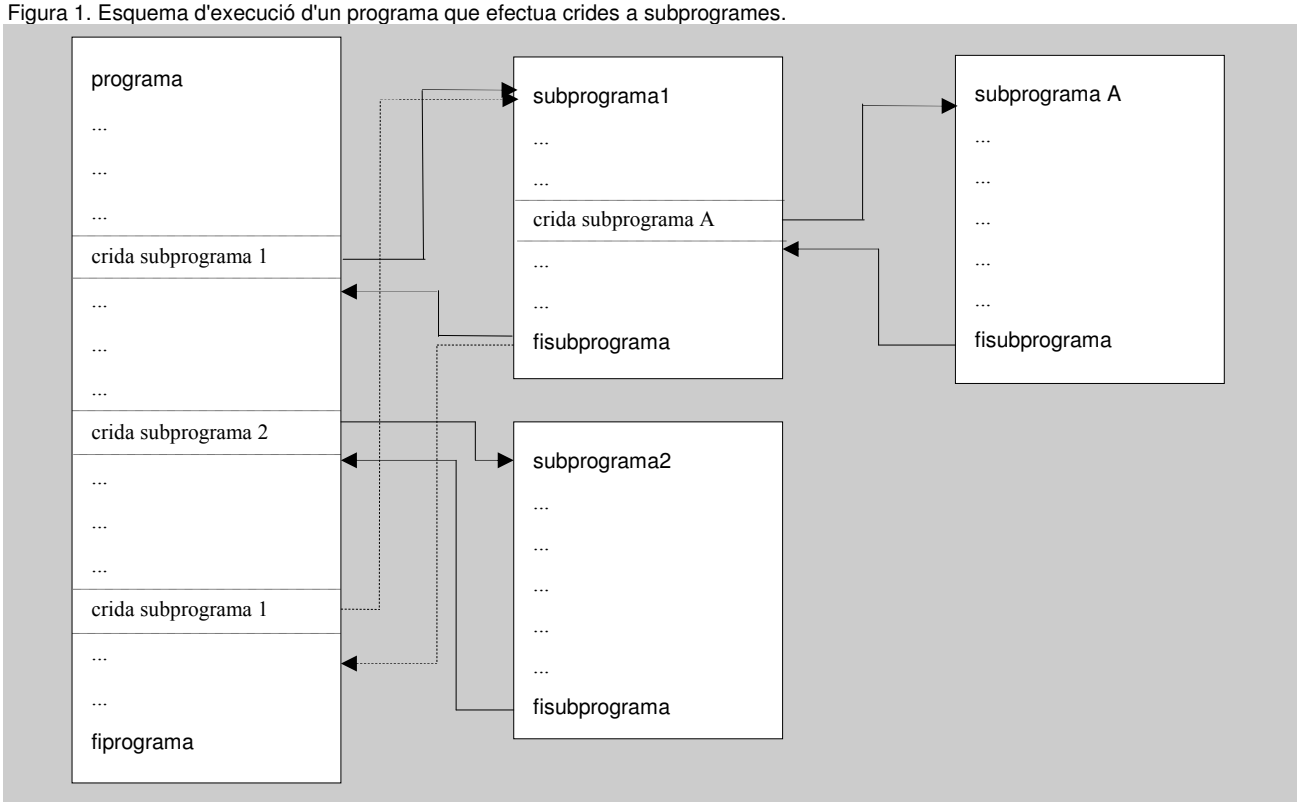
- I de forma similar amb els mòduls de comissions, estadístiques, magatzem,...

- Mòdul de funcionalitats genèriques, format per diferents fitxers font que contenen, entre d'altres:
  - Subprogrames per a la gestió de les formes de pagament
  - Subprogrames per a la gestió de les divises amb els seus canvis
  - Subprogrames per a la gestió dels països
  - Subprogrames per a la gestió dels departaments dels diferents països (províncies en el cas d'Espanya)

És clar que els subprogrames de qualsevol mòdul poden necessitar, en algun moment, de subprogrames d'altres mòduls. Així, en molts llocs es pot necessitar accedir a les divises, als països, als departaments dels països, a les formes de pagament, etcètera.

La figura 1 ens mostra l'esquema d'execució d'un programa que efectua crides a subprogrames, els quals, a la vegada, poden efectuar crides a altres subprogrames. No hem inclòs, a la figura, que els diversos programes poden estar agrupats en diferents fitxers.

Figura 1. Esquema d'execució d'un programa que efectua crides a subprogrames.



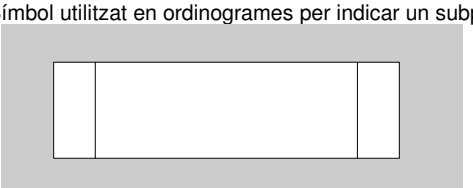
Un mateix subprograma es pot activar des de diferents punts del programa principal, i fins i tot des d'un altre subprograma. En tots dos casos, però, quan s'acaba l'execució del subprograma, el flux del programa torna a la línia següent, a aquella des d'on s'ha efectuat la crida.

El subprograma s'activa normalment mitjançant el seu propi nom. Opcionalment, al mateix temps que l'activem hi podem fer la transferència de dades que ha d'utilitzar i que han estat definides i generades en una altra part del programa.

Els subprogrames es classifiquen en dos grans grups generals: les funcions i les accions (també anomenades, aquestes darreres, procediments o subrutines). ‼

Abans de passar a l'estudi detallat de les accions i de les funcions, cal que conegueu el símbol utilitzat en els ordinogrames per a fer referència a un subprograma. La figura 2 ens el mostra.

Figura 2. Símbol utilitzat en ordinogrames per indicar un subprograma.



1.2.1. Accions

Una acció és un conjunt d'instruccions amb un objectiu comú, que pot necessitar o no dades externes per a la seva execució.

La sintaxi per definir una acció en pseudocodi és la següent:

```
acció <nom> [(<llista d'arguments>)] és
   [var ... fivar]
   ...
   /* conjunt d'instruccions */
   ...
fiacció
```

Observeu que:

- Una acció pot incorporar (és optatiu) una llista d'arguments que es fa servir per a transferir, des del programa que activa l'acció, les dades que necessitarà l'acció, si és el cas, per a la correcta execució.

- Una acció pot incorporar (és optatiu) un apartat de declaració de variables, les quals es consideren locals a la funció.

‼ A l'apartat "Variables globals i locals" veurem com es declaren les variables de les accions.

Per a efectuar una crida d'una acció des del programa principal o des d'una altra acció o funció només cal escriure:

```
<nom_de_l'acció> [(<llista de paràmetres>)];
```

La llista de paràmetres és obligatòria en el cas que l'acció s'hagi declarat amb aquesta llista. En aquesta situació s'exigeix que coincideixin el nombre, ordre i tipus dels paràmetres de la crida amb el nombre, ordre i tipus dels arguments de la declaració de l'acció. (!!)

En pseudocodi no ens importarà el lloc on estigui declarada l'acció, però els llenguatges són més rígids i cal seguir les instruccions al respecte. En alguns és obligatori després del programa principal. En altres és obligatori abans de cap aparició d'una corresponent crida d'activació. En definitiva, cal seguir la normativa de cada llenguatge.

1.2.2. Funcions

El concepte de funció és l'altre concepte clau en l'estudi de subprogrames.

> Una funció és un conjunt d'instruccions amb un objectiu comú, que pot necessitar o no dades externes per a la seva execució i que proporciona un resultat a qui l'ha activat.

La sintaxi per a definir una funció en pseudocodi és la següent:

```
funció <nom> [(<llista d'arguments>)] retorna <tipus> és
   [var ... fivar]
   ...
   /* conjunt d'instruccions */
   ...
   retorna <resultat>;
fifunció
```

Observeu que:

- Igual que les accions, pot tenir una llista d'arguments i una llista de variables locals.

(!!) A l'apartat "Variables globals i locals" veurem com es declaren les variables de les funcions.

- A diferència de les accions, les funcions sempre proporcionen un resultat a qui les ha activat i, per tant, en la seva declaració hi ha de constar el `<tipus>` del resultat; així mateix, el seu codi ha de contenir com a mínim una instrucció similar a:

```
retorna <resultat>;
```

que s'utilitza per retornar, a qui l'ha activat, el resultat de la funció. Hi pot haver diverses instruccions `retorna`, però no és aconsellable; el més estructurat és tenir una única instrucció `retorna` a l'acabament del codi de la funció.

Matemàticament, una funció és una operació en què, a partir d'unes dades anomenades arguments, es produeix un valor anomenat resultat. El tipus de dada obtinguda depèn de les operacions i és problema del llenguatge de programació que determinats tipus de dades siguin vàlids o no.

Com que les funcions retornen un resultat, acostumen a actuar com a operands d'expressions que fan servir aquell resultat. Així, la seva activació segueix la sintaxi similar a la següent:

```
<nom_de_funció> [(<llista de paràmetres>)]
```

i aquest codi s'emmarca dins una expressió on el tipus del resultat retornat per la funció hi tingui sentit.

De manera anàloga a la crida de les accions, la llista de paràmetres és obligatòria en el cas que la funció s'hagi declarat amb tal llista. En una situació com aquesta s'exigeix que coincideixin el nombre, ordre i tipus dels paràmetres de la crida amb el nombre, ordre i tipus dels arguments de la declaració de la funció. ‼

1.3. Accessibilitat de variables

Fins aquest moment s'ha vist que les dades que es fan servir als subprogrames (accions i funcions) són bàsicament les variables i els paràmetres. A continuació s'especifica amb més detall la manera d'utilitzar-les.

1.3.1. Variables globals i locals

Abans d'estudiar la manera com s'han de fer servir les variables i els paràmetres, cal distingir les variables globals de les variables locals.

Les variables globals són les que han estat definides al començament del programa principal i el seu àmbit de validesa i disponibilitat és total, és a dir, les podem utilitzar, actualitzant-les o no, el propi programa principal i els subprogrames cridats, tant des del programa principal com des d'altres subprogrames cridats també directament o indirectament des del programa principal.

A l'interior d'un subprograma es poden produir modificacions, imprevistes i no volgudes, pel fet d'usar variables globals. Aquestes modificacions reben el nom d'efectes laterals. (!!)

Accés des dels subprogrames

No s'ha d'accedir a les variables globals des dels subprogrames, perquè això pot causar errors en el programa principal o en la resta de subprogrames.

Quan es dissenya programes cal fugir de l'accés voluntari a les variables globals des dels subprogrames, ja que així s'evitaran errors difícils de detectar. D'aquesta manera, el subprograma no provocarà efectes laterals al programa principal ni, de retruc, a la resta de subprogrames.

Com es pot fugir de l'accés voluntari a les variable globals des dels subprogrames? No tenim cap altre remei que fer servir la transferència de dades entre el subprograma que efectua la crida de l'acció o la funció i la pròpia acció o la funció cridada. Aquesta transferència es defineix amb la llista d'arguments en la declaració de l'acció o la funció i amb la llista de paràmetres en el moment de la crida. Si no tinguéssim aquesta forma de comunicació (o una de similar) entre programa que efectua la crida i subprograma cridat, no hi hauria cap altre remei que accedir a les variables globals de manera directa des de dins els subprogrames.

(!!)
Vegeu, a l'exemple `u3n1p07` de l'apartat "Exemples d'aprofundiment en la manipulació de taules" de la unitat didàctica "Gestió de dades estàtiques compostes", la manipulació de variables declarades en el programa principal, dins diverses accions.

En no fer servir les variables globals, l'estructura del subprograma es converteix en una caixa negra per a la resta de subprogrames i el programa principal. D'aquesta manera aconseguim independència total de l'acció o la funció respecte del nom de les variables que contenen les dades que s'ha de gestionar en el programa que efectua la crida. Això últim és molt important, ja que d'aquesta manera podem utilitzar l'acció o la funció en un altre programa. (!!)

Fixeu-vos bé que estem aconseguint la reutilització de codi, un dels objectius primordials de la programació estructurada i modular i, en darrer terme, de la programació orientada a objectes.

Una variable local és aquella que està declarada dins d'un subprograma. L'àmbit d'ús de les variables locals és el subprograma en què s'han definit i tots els subprogrames interiors

d'aquest (és a dir, actua com a variable global per als subprogrames interiors).

Recordeu que en la sintaxi de la declaració d'accions i funcions en pseudocodi hi havia un apartat de declaració de variables abans de començar el codi de l'acció o la funció? Les variables declarades en aquesta zona són les variables locals. (!!)

1.3.2. Àmbit d'existència

Un concepte que cal no oblidar quan parlem de variables és el seu àmbit d'existència.

L'àmbit d'existència d'una variable és la part del programa en què la variable pot ser referenciada o s'hi pot accedir pel seu nom.

Què succeeix quan el nom d'una variable local coincideix amb el nom d'una variable global? Es produeix una col·lisió. En aquest cas, la variable global queda amagada per la local fins que se surt de l'àmbit d'existència de la local. Quan es torna a l'àmbit en què existia es recupera el tipus i el valor que tenia quan va quedar amagada. És a dir, es comporta com si la seva identitat hagués estat usurpada per l'altra dada, de manera que quan l'altra desapareix aquesta recupera la seva identitat. (!!)

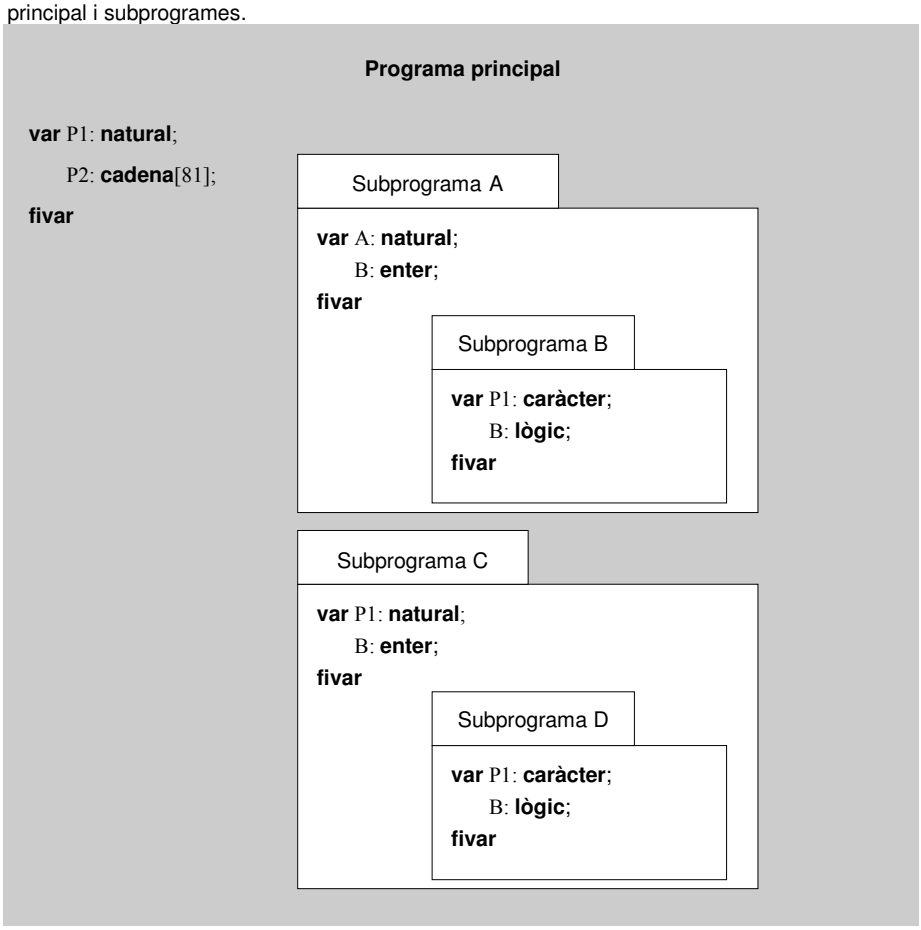
La figura 3 mostra un programa que conté, anomenem-lo programa principal, que conté crides a dos subprogrames A i C, els quals a la seva vegada efectuen crides a subprogrames B i D respectivament. La taula 1 ens mostra quines són les variables a les que es té accés des de dins de cada subprograma.

Taula 1. Accessibilitat a les variables del programa i subprogrames de la figura 3 des de diferents punts d'execució

Punt d'execució	Variables accessibles	Procedència
Programa principal	P1 (natural) i P2 (cadena)	
Subprograma A	P1 (natural) i P2 (cadena)	Programa principal
	A (natural) i B (enter)	Subprograma A
Subprograma B	P2 (cadena)	Programa principal
	A (natural)	Subprograma A
	P1 (caràcter) i B (lògic)	Subprograma B
Subprograma C	P2 (cadena)	Programa principal
	P1 (natural) i B (enter)	Subprograma C

Punt d'execució	Variables accessibles	Procedència
Subprograma D	P2 (cadena)	Programa principal
	P1 (caràcter) i B (lògic)	Subprograma D

Figura 3. Exemple de definició de variables en diferents parts d'una aplicació: programa principal i subprogrames.



Cal tenir present que les variables passen a ocupar part de la memòria de l'ordinador en el moment en què l'execució del programa passa pel lloc on comencen a ser accessibles i desapareixen de la memòria en el moment en què l'ordinador finalitza l'execució del subprograma en el qual havien estat definides.

1.3.3. Transferència de dades: per valor i per referència

Els arguments que es fan servir en la declaració dels subprogrames s'encarreguen de fer la transferència de dades amb el subprograma que efectua la crida.

Amb la utilització dels arguments s'eviten els efectes laterals. Recordeu que emprem el terme arguments en la declaració del subprograma, mentre que en la crida és el terme paràmetres l'emprat per a referir-nos a les dades transferides. Hi ha dos termes més, diferents, però equivalents als

anteriors: paràmetres formals, equivalents a arguments, i paràmetres actuals, equivalents a paràmetres. (!!)

La transferència de les dades es pot fer bàsicament de dues maneres: per valor i per referència (també anomenat per variable).
El tipus de transferència es decideix per cada argument del subprograma de manera independent. És a dir, podem tenir arguments amb transferència per valor i arguments amb transferència per referència.

> En la transferència per valor, el valor del paràmetre actual es copia en una altra posició de memòria, a la qual es pot accedir pel nom del paràmetre formal.

Res del que fem en el subprograma no afectarà el paràmetre transferit (per valor) del programa principal (paràmetre actual) i, quan s'acabi el subprograma, s'alliberarà la memòria utilitzada pel paràmetre formal.

> En la transferència per referència, el subprograma rep la direcció de memòria en què es troba el paràmetre actual, a la qual es pot accedir llavors pel nom del paràmetre formal.

Qualsevol acció efectuada per mitjà del nom del paràmetre formal (transferit per referència) afecta la posició de memòria i, per tant, és equivalent a treballar amb el paràmetre actual. Quan s'abandona el subprograma, els canvis efectuats en el paràmetre formal queden reflectits en el paràmetre actual.

Observeu el perquè dels noms per valor i per variable o referència. En el primer es traspassa el valor del subprograma que efectua la crida al subprograma cridat, el qual rep el valor en una nova variable que està ocupant memòria. En el segon, el subprograma que efectua la crida "cedeix" la variable al subprograma cridat, el qual la fa servir amb un altre nom, però està treballant amb la mateixa variable.
No és convenient utilitzar la transferència per variable en funcions, ja que la declaració d'una funció porta implícita l'obtenció d'un resultat. Qualsevol altre retorn mitjançant un argument podria crear confusió. En aquests casos és preferible definir una acció i fer servir tots els arguments per referència necessaris.

La sintaxi utilitzada en pseudocodi per a declarar els arguments per valor i per referència és molt senzilla.

- Per referència: es precedeix l'argument amb la paraula reservada var.
- Per valor: l'argument no es precedeix amb cap paraula reservada.

Qualsevol paràmetre formal d'una acció o d'una funció es comporta com una variable local dins l'acció o la funció, de manera que és global per qualsevol subprograma activat des de dins de l'acció o la funció.

Tot paràmetre actual utilitzat en un argument per referència ha de ser, obligatòriament, una variable, mentre que en un argument per valor pot ser una variable o una constant. El motiu és clar: quan la transferència és per referència, s'accedeix a la variable de qui fa la crida, la qual pot ser modificada; per força ha de ser una variable i no pot ser una constant. ‼

1.3.4. Exemple de transferència per valor i per referència

En molts programes, el programador es troba amb la necessitat de rebre una resposta per teclat i controlar si el valor introduït és una 'S' o una 'N'.

Valors resposta

Els valors "S" o "N" són la resposta normalment utilitzada a preguntes com "Vol continuar?", "Hi està d'acord?", etc.

Volem generalitzar la programació d'aquesta feina que apareix en moltes ocasions. Per a aconseguir-ho cal dissenyar un subprograma que cridarem en les ocasions en què ens interessi. Aquestes són algunes de les preguntes que ens hem de formular:

- Dissenyem una acció o una funció?
- Fem servir arguments o no? En cas afirmatiu, quins i com és la seva transferència?

Les respostes les trobem en l'anàlisi del que pretenem. En efecte, ens interessa que el subprograma efectuï la lectura per teclat i que controli el valor introduït de manera que si no és correcte obligui a tornar-lo a introduir. És clar que quan acabem, el valor llegit s'haurà de comunicar a qui hagi efectuat la crida al subprograma, ja que està a l'espera de prendre decisions en funció del caràcter llegit. Conclusió: el subprograma que volem dissenyar ha de comunicar una dada a qui activa la crida. Tenim dues possibilitats:

1) Una funció que retorni el caràcter llegit i comprovat.
2) Una acció que utilitzi un argument per referència per a comunicar el caràcter llegit i comprovat.

Fixeu-vos que ni tan sols hem considerat la possibilitat de fer servir una acció que, sense argument per referència, treballi directament amb una

variable global, ja que d'aquesta manera estaríem obligats que tots els programes que volguessin activar la nostra acció tinguessin definida una variable amb el mateix nom que la que emprem en el cos de l'acció.
A més, com que es fa una lectura per teclat és molt probable que ens interessi poder situar el cursor en una posició determinada, de manera que si s'hagués de repetir la lectura perquè el valor introduït no fos correcte, es pugui tornar a efectuar la lectura amb el cursor situat en la posició inicial. Això implica que el subprograma ha de tenir coneixement de la posició en què ha d'efectuar la lectura i, per tant, caldrà dos arguments per a comunicar la fila i la columna on s'ha de fer la lectura. Aquests dos arguments poden, perfectament, ser passats per valor al subprograma, ja que no n'han de modificar el valor.

Un consell: definiu noms que tinguin a veure amb el que ha de fer el subprograma.

Dissenyarem l'algorisme per al cas de la funció i de l'acció. Abans, però, cal deixar constància que les accions i funcions s'han de batejar, és a dir, han de tenir nom, el qual no pot ser cap paraula reservada del llenguatge i no pot tenir espais en blanc ni alguns caràcters no gaire corrents.

```
acció llegir_SN_1 ( fil, col: natural, var c: caràcter) és
   fer
      cursor (col, fil);
      llegir (c);
   mentre c != 'S' i c != 's' i c != 'N' i c != 'n';
   si c == 's'
      llavors c = 'S';
      sinó si c == 'n' llavors c='N'; fisi
   fisi
   /* observeu que encara que l'usuari ho hagi entrat en
   minúscula, el caràcter c passa a ser majúscula */
fiacció
```

L'acció `llegir_SN_1` treballa amb els paràmetres formals `fil`, `col` (per valor) i `c` (per variable), els quals donen independència total a qui activi l'acció. El subprograma que efectuï la crida pot passar als arguments `fil` i `col` valors constants o variables amb qualsevol nom. Simplement hi ha d'haver coincidència de tipus. En canvi, a l'argument `c` li ha de passar obligatòriament una variable caràcter (de qualsevol nom), la qual quedarà emplenada amb el valor llegit per l'acció.

```
funció llegir_SN_2 ( fil, col: natural) retorna caràcter és
   var  c: caràcter;    fivar
   fer
      cursor (col, fil); llegir (c);
   mentre c != 'S' i c != 's' i c != 'N' i c != 'n';
   si c == 's' llavors c = 'S';
      sinó si c == 'n' llavors c='N'; fisi
   fisi
   retorna c;
fifunció
```

El cos de la funció és el mateix que el de l'acció, però en la funció no hi ha el paràmetre formal per referència `c` i per aquest motiu s'ha hagut de declarar com a variable local. Per comunicar, a qui efectua la crida, el valor `c` llegit per la funció, es retorna `c`. (!!)

Vegem un exemple de programa que efectua la crida de l'acció i de la funció en punts diferents.

```
programa exemple és
   var  opc: caràcter;    fivar
   netejar_pantalla;
   cursor (10,5); escriure ("Es considera una persona amb sort (S/N)?");
   llegir_SN_1 ( 5, 56, opc);
   cursor (10, 7); escriure ("Ha contestat: ", opc);
   cursor (10, 9); escriure ("Està content d'aquesta resposta (S/N)?");
   opc = llegir_SN_2 ( 9, 54);
   cursor (10, 11); escriure ("Ha contestat: ", opc);
fiprograma
```

En aquest cas, els arguments amb transferència per valor s'omplen, en la crida, amb valors constants. En canvi, l'argument amb transferència per referència s'omple, en la crida, amb la variable `opc`. El nom d'aquesta és totalment independent del paràmetre formal `c` de l'acció. Poden coincidir en nom o no; és indiferent.

1.4. Funcions en C

El programa principal de C és una funció.

Un programa en C està format per una o més funcions. Tot programa en C ha de contenir una funció anomenada `main`. És a dir, el concepte `funció` és fonamental en C, de tal manera que el mateix programa principal és una funció.

Mentre que molts llenguatges distingeixen entre acció (no retorna res) i funció (retorna una dada), el llenguatge C utilitza l'únic terme `funció`, que inclou els dos conceptes. El llenguatge té en compte, però, la distinció entre acció i funció.

> El concepte funció corresponent a subprograma que retorna un resultat és, en C, equivalent a una funció que retorna una dada. El concepte acció corresponent a subprograma que no retorna cap resultat és, en C, equivalent a una funció que retorna `void`.

Hem fet servir, en tot el que portem de crèdit, la paraula `void` davant de la paraula `main` i també en l'interior dels parèntesi que l'acompanyen Potser ara ja comenceu a trobar-hi explicació. La paraula `main` és el nom

de la funció corresponent al programa principal (pel qual s'inicia l'execució), i està definida, en tots els programes que hem fet, com a funció que no retorna res, o dit d'una altra manera, com a funció amb resultat de tipus `void`. Això no té per què ser així. Podríem fer que el programa transferís un resultat a qui l'activés (bé des del comandament de sistema operatiu, bé des d'un altre programa).

El tipus `void` especifica un conjunt buit de valors. En realitat `void` no és un tipus, encara que per manera de fer-lo servir, si el comparem amb la manera d'utilitzar els altres tipus fonamentals, es considera com a tal.

Atenció! No es pot declarar una variable de tipus `void`.

El tipus `void` es fa servir, entre altres situacions, per a:

- Indicar que una funció no accepta arguments:

```
int funció (void)
```

  Per això hem sempre hem posat `void` dins els parèntesi que acompanyen la declaració del programa principal `main`, tot i que no és obligatori.

- Declarar funcions que no retornen cap valor:

```
void funció (int, char)
```

1.4.1. Declaració de funcions

Un cop hem estudiat el concepte de funció, ens cal ara definir la declaració d'una funció en llenguatge C.

> La declaració d'una funció, també anomenada prototipus de la funció, indica quants arguments té la funció i de quin tipus són, i el tipus del valor retornat.

La seva sintaxi és la següent:

```
<tipus_retornat> <nom_funció> ( [<llista_arguments>] );
```

El prototipus d'una funció és una plantilla que utilitza el compilador per a comprovar que qualsevol crida a la funció és correcta, és a dir, que els paràmetres actuals són adequats als arguments i el valor retornat es tracta correctament.

Com a exemple, observem:

```
void llegir_SN_1 (unsigned, unsigned, char *);
char llegir_SN_2 ( unsigned, unsigned);
```

que són els prototipus, en llenguatge C, de les dues versions típiques del subprograma per efectuar una lectura de la típica resposta S/N a una pregunta:

- la primera versió corresponent a la traducció d'una acció en pseudocodi, on el valor llegit es retorna via argument per referència;
- la segona versió corresponent a la traducció d'una funció en pseudocodi, on el valor llegit es retorna via instrucció `retorna`.

Ens manca, però, saber com es defineix la transferència per valor i per referència.

Cal tenir clara la diferència que hi ha entre declaració i definició de la funció. La declaració d'una funció (prototipus) dóna característiques de la funció, però no defineix el procés que fa. La definició de la funció conté la declaració de les variables locals i el codi corresponent. (!!)

Una funció pot ser declarada implícitament o explícitament. Aquest és un concepte de l'àmbit de la compilació. El programa compilador, en el procés de traducció d'un programa font, comença per l'inici del codi i evoluciona seqüencialment fins a la fi. En aquest procés es troba elements com el programa principal, declaracions de variables globals, declaracions i definicions de funcions, crides a funcions, etc., els quals no té per què trobar-se en cap ordre concret. El programa compilador pren la decisió de declarar cada funció de manera implícita o explícita.

(!!) A l'apartat "Definició de funcions" veurem com s'indica el codi que defineix el procés que executa la funció.

La declaració és explícita quan el compilador detecta el prototipus de la funció o la seva definició abans de detectar-ne cap crida. La declaració és implícita quan es detecta alguna crida abans de trobar-se la declaració o la definició. En tal cas, el compilador de C construeix un prototipus per defecte, el qual consisteix en una funció que sempre retorna un resultat int i que té per llista d'arguments la construïda a partir dels tipus dels paràmetres actuals especificats en la crida a la funció. Això obliga que el tipus del resultat en la definició de la funció sigui int.

Diferències entre C i C++

La declaració implícita d'una funció no es preveu en el llenguatge C++.

Treballar amb declaracions implícites provoca problemes i per aquest motiu es recomana fer sempre la declaració explícita. Per fer la declaració explícita només cal declarar el prototipus abans de cap crida

Un consell: acostumeu-vos a treballar amb prototipus.

i, per això, els prototipus s'acostumen a situar a l'inici de qualsevol programa font. També es pot anar amb compte i situar la definició de les funcions abans de cap crida.

Fixem-nos que en la declaració de la funció apareix una llista opcional d'arguments. És una llista dels identificadors amb els seus tipus, separats per comes. Quan es tracta d'un prototipus es poden ometre els identificadors dels arguments, cosa que s'ha fet en els exemples anteriors. Podríem, però, haver escrit els prototipus com:

```
void llegir_SN_1 (unsigned fil, unsigned col, char *c);
char llegir_SN_2 (unsigned fil, unsigned col);
```

Els noms dels identificadors dels arguments que es fan servir en un prototipus no tenen per què ser els mateixos que els que s'utilitzen en la definició de la funció.

Oi que en tots els programes que hem fet en llenguatge C on efectuàvem crides a les funcions `scanf`, `printf`, `strcmp`, `strcat`, etc. ha estat necessari especificar a l'inici del programa l'`include` del fitxer de capçalera on consten les característiques de les funcions que s'han d'utilitzar? Les característiques a les quals ens referim no són altra cosa que els prototipus de les funcions corresponents, per facilitar al compilador la verificació sintàctica de les crides a les funcions de C que detecta en el programa font. (!!)

El tipus del resultat especifica quin tipus de dades retorna la funció. Pot ser qualsevol tipus fonamental del llenguatge o tipus definit per l'usuari però no pot ser una taula ni una funció. Si no s'especifica es suposa que és un `int`.

1.4.2. Definició de funcions

La definició d'una funció consta de la capçalera de funció, similar a un prototipus però amb l'obligatorietat d'incloure els identificadors de la llista d'arguments, i el cos de la funció. (!!)

```
<tipus_retornat> <nom_funció> ([<llista_arguments>])
{
   declaracions_de_variables_locals;
   sentències;
   [return [()]expressió[)]];
}
```

Les variables declarades en el cos de la funció són locals i, per definició, només s'hi pot accedir dins la funció. És a dir, a diferència del que succeeix en altres llenguatges i del que hem considerat en pseudocodi, les variables

declarades en una funció només són conegudes dins la funció i no són globals dins les funcions que puguin ser cridades des de la funció en què són globals. (!!)

El resultat d'una funció és retornat a qui l'ha cridat amb la sentència següent:

```
return [()]expressió[)];
```

Aquesta sentència pot ser la darrera o no i pot aparèixer més d'una vegada en el cos de la funció. En el cas que la funció no retorni un valor, es pot suprimir o especificar simplement la paraula `return`. Això últim tindria sentit si es volgués abandonar la funció abans d'arribar a la fi del cos d'instruccions.

Els paràmetres formals o arguments consisteixen en una llista d'identificadors amb els seus tipus, separats amb comes.

Així, podem considerar la codificació en C de la funció llegir_SN_2 i tindríem:

```
#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* Per compatibilitzar diferents plataformes */

char llegir_SN_2 (unsigned int fil, unsigned int col)
{
   char c;
   int n;
   do
   {  gotoxy (col, fil);
      n = scanf ("%c", &c); neteja_stdin();
   } while (n==0 || ( c != 'S' && c != 's' && c!= 'n' && c!= 'N'));
   if (c == 's') c='S';
   else if (c == 'n') c='N';
   return c;
}
```

Hi ha, però, una altra sintaxi per a la definició de les funcions, més antiga i que ja està quasi en desús. Consisteix a efectuar la declaració dels arguments després de la capçalera de la funció i abans de la clau d'obertura del cos de la funció amb la mateixa sintaxi que la declaració de variables. Així, tindríem:

```
#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* Per compatibilitzar diferents plataformes */

char llegir_SN_2 (fil, col)
unsigned int fil, col;
```

```
{
   char c;
   int n;
   do
   {  gotoxy (col, fil);
      n = scanf ("%c", &c); fflush (stdin);
   } while (!n || ( c != 'S' && c != 's' && c!= 'n' && c!= 'N'));
   if (c == 's') c='S';
   else if (c == 'n') c='N';
   return c;
}
```

1.4.3. Crida de funcions

Per a executar una funció cal cridar-la. La crida a una funció consta del nom d'aquesta i de la llista de paràmetres actuals, tancats entre parèntesis i separats per comes.
Vegem la crida en un programa de la funció `llegir_SN_2`:

```
/* u4n1p01.c */

#include <stdio.h>
#include <conio.h>
#include <compat.h>  /* Per compatibilitzar les plataformes */

char llegir_SN_2 (unsigned int fil, unsigned int col)
{...}   /* Aquí hi va el codi vist uns paràgrafs amunt */

void main(void)
{
   char opc;
   clrscr();
   gotoxy (10,5); printf ("Es considera una persona amb sort (S/N)?");
   opc = llegir_SN_2 (5, 56);
   gotoxy (10,7); printf ("Ha contestat: %c", opc);
   gotoxy (10,9); printf ("Està content d'aquesta resposta (S/N)?");
   opc = llegir_SN_2 (9, 54);
   gotoxy (10,11); printf ("Ha contestat: %c", opc);
}
```

‼ Trobareu el fitxer `u4n1p01.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

En aquest exemple podem observar la utilització d'una funció que retorna un valor `char` que s'emmagatzema a la variable `opc`. Tota funció, retorni o no un valor, pot cridar-se sense formar part de cap expressió. En cas que retorni un valor i la crida no formi part d'una expressió, el resultat de la funció es perd.

1.4.4. Accessibilitat de variables en C

Vegem com és l'accés a les variables en funció dels diversos paràmetres que hi estan relacionats.

Variables globals i locals

Coneixem, en el llenguatge C, que les variables declarades en el cos d'una funció són locals a la funció i no són globals a cap de les funcions cridades des de la funció en què estan declarades. Per aquest motiu, si una funció G cridada des d'una funció F ha de tenir accés a variables de la funció F, caldrà efectuar una transferència de dades mitjançant arguments.

Tingueu present que el programa `main` no deixa de ser una funció. Per tant, totes les variables declarades dins el seu cos són locals. Això està en total contradicció amb les regles del joc establertes en el pseudocodi, les quals són vàlides per a molts llenguatges de programació.

En C, l'àmbit d'accessibilitat d'una variable pot estar limitat a un bloc, a un fitxer, a una funció o a una declaració d'una funció. (!!)

> Una variable és declarada en àmbit `extern` quan es declara fora de qualsevol bloc en un programa; en principi s'hi pot accedir des del seu punt de definició o declaració fins al final del fitxer font on ha estat declarada.

En llenguatge C entenem per bloc qualsevol conjunt de codi limitat per les claus.

> Una variable és declarada en àmbit `intern` quan la declaració s'efectua dins d'un bloc; en principi s'hi accedeix dins el bloc i dins els blocs continguts dins d'aquest per sota del punt de declaració.

Les variables definides en un bloc han d'estar situades obligatòriament a l'inici del bloc. En C++ poden, però, definir-se en qualsevol lloc del bloc. En tal cas es pot accedir a la variable a partir del punt on s'ha declarat i fins a la fi del bloc.

Igual que en pseudocodi, la declaració d'una variable en un bloc amaga l'accessibilitat de qualsevol variable externa al bloc amb el mateix nom.
Els arguments declarats en la llista d'arguments de la declaració d'una funció o un prototipus existeixen únicament en la pròpia declaració de la funció. Aquest fet motiva que els identificadors dels arguments en un prototipus es puguin obviar.

Els arguments declarats en la llista d'arguments de la definició d'una funció són interns a la funció, amb la mateixa accessibilitat que les variables declarades a l'inici del cos de la funció.

Emmagatzematge `auto-register-static-extern`

Per defecte, totes les variables porten associat un tipus d'emmagatzematge que en determina l'accessibilitat i existència. Els conceptes d'accessibilitat i d'existència, tant per a les variables com per a les funcions, es poden alterar amb uns determinats qualificadors que es fan servir davant la definició de la variable o la funció.

Aquests qualificadors són `auto` i `register` (utilitzables únicament amb variables locals), `extern` (utilitzable amb variables globals i amb funcions) i `static` (utilitzable amb variables locals, globals i amb funcions).

1) Variables declarades com a `auto`

Aquest és el qualificador que assigna per defecte el compilador quan troba variables locals sense qualificar.

> Una variable declarada com a `auto` només és visible dins el bloc on està definida.

Les variables `auto` no són inicialitzades automàticament, és a dir, poden contenir "porqueria". Per tant, és necessari inicialitzar-les explícitament.

2) Variables declarades com a `static`

Una variable `static` és inicialitzada una única vegada quan comença l'execució del programa. No és reinicialitzada cada vegada que s'executa el bloc que la conté –en cas de ser interna–. Si la variable no és inicialitzada explícitament, C la inicialitza automàticament a zero (les numèriques a zero i les caràcter amb '\0').

> Una variable declarada com a `static` només és visible dins el bloc -si es tracta de variable interna- o dins el fitxer font –si es tracta de variable externa– en què estigui definida.

El valor d'una variable `static` –en el cas de ser interna– és permanent en lloc d'aparèixer i desaparèixer en l'inici i final de l'execució del bloc que la conté.

En el cas de variable externa, el qualificador `static` permet tenir declarades altres variables `static` amb el mateix nom en altres fitxers font corresponents al mateix programa.

3) Variables declarades com a `extern`

> Una variable declarada com a `extern` fa referència a una variable definida amb el mateix nom en l'àmbit extern en qualsevol lloc del programa –del mateix o diferent codi font–.

Una variable `extern` no inicialitzada explícitament, és inicialitzada a zero pel compilador.

El següent codi ens mostra el funcionament de les declaracions `extern`:

```
/* u4n1p02.c  */

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

void f1 (void)
{
   extern int v;   /* No és variable local a f1 */
   /* Es suposa que quan s'executi f1, ja existirà v */
   printf ("Valor de v dins f1 en entrar: %d\n",v);
   v = 10;
   printf ("Valor de v dins f1 en sortir: %d\n",v);
}

void f2 (void)
{
   extern int v;   /* No és variable local a f2 */
   /* Es suposa que quan s'executi f2, ja existirà v */
   printf ("Valor de v dins f2 en entrar: %d\n",v);
   v = 20;
   printf ("Valor de v dins f2 en sortir: %d\n",v);
}

void main(void)
{
   int v;
   clrscr();
   v=30;
   printf("Valor de v dins main: %d\n",v);
   f1();
   printf("Valor de v dins main: %d\n",v);
```

Trobareu el fitxer `u4n1p02.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
   f2();
   printf("Valor de v dins main: %d\n",v);
}

int v=15;
```

L'execució d'aquest programa dona:

```
Valor de v dins main: 30
Valor de v dins f1 en entrar: 15
Valor de v dins f1 en sortir: 10
Valor de v dins main: 30
Valor de v dins f2 en entrar: 10
Valor de v dins f2 en sortir: 20
Valor de v dins main: 30
```

Observem la lògica del procés:

- El programa principal `main` té declarada una variable local `v` que pren el valor 30 i que, durant tota l'execució continua valent 30.

- Les funcions `f1` i `f2` declaren una variable `extern v`. Això vol dir que és necessari que hi hagi una declaració externa de la variable `v`, ja sigui dins el mateix font (com és el cas, en la darrera línia del fitxer) o en un altre programa font (que l'enllaçador s'encarregaria d'enllaçar per obtenir el codi executable). Els missatges que s'imprimeixen des de l'interior de les funcions `f1` i `f2` demostren que la variable v accedida és la declarada a l'exterior de les funciona (darrera línia de codi).

4) Variables declarades com a `register`

L'última forma d'emmagatzematge de les variables que veurem és la declarada com a `register`.

> Una variable d'àmbit intern qualificada com a `register` indica al compilador que la variable sigui emmagatzemada, si és possible, en un registre de la màquina.

El nombre de registres utilitzables per aquest tipus de variable depèn de la màquina. Si no és possible emmagatzemar-la en un registre, rep el tractament d'`automàtica`.

Registres interns

A l'interior del processador de les màquines hi ha unes zones reservades per a l'emmagatzematge de petites quantitats d'informació. Són els registres interns. El

processador utilitza aquests registres per a l'emmagatzematge temporal de dades o d'adreces de memòria; és a dir, la posició de memòria on s'emmagatzemen les dades.

Amb la finalitat d'augmentar la velocitat de procés, quan una dada és sol·licitada amb freqüència, es pot deixar en un registre interior del processador.

Aquest tipus de declaració és vàlid per a variables de tipus `enter` (`char`, `unsigned char`, `int`, etc.) i de tipus `punter`, el qual encara no hem presentat. Una variable declarada com a `register` només és visible dins el bloc on està definida. Aquestes variables no són inicialitzades automàticament, per la qual cosa se les ha d'inicialitzar explícitament; en cas contrari, contenen valor "porqueria".

1.4.5. Declaració de funcions en àmbit `extern`

Una funció també pot ser qualificada com a `static` o `extern`.

> Una funció declarada com a `static` és accessible únicament dins el fitxer font en què està definida.

La declaració com a `static` de funcions possibilita tenir diferents funcions amb el mateix nom en diferents fitxers font que formen part del mateix programa.

> Una funció declarada com a `extern` és accessible des de tots els fitxers font que formen un programa.

La declaració d'una funció com a `static` o `extern` es pot fer en el prototipus o en la definició. Per defecte, una funció sense qualificar és declarada `extern`.

1.4.6. Pas per valor i per referència

Quan es crida una funció, el valor del primer paràmetre actual és passat al primer paràmetre formal, el valor del segon paràmetre actual és passat al segon paràmetre formal, i així successivament. Per defecte, tots els arguments, excepte les taules, són transferits per valor.

Per això no hem tingut cap problema a codificar la funció `llegir_SN_2`, perquè tots els paràmetres eren per valor. La codificació s'ha convertit, en aquest cas, en una simple traducció.

Com es codifica en C la transferència de dades per referència?

Ja sabeu que en aquest tipus de transferència la funció ha de rebre la direcció de memòria en què es troba el paràmetre actual, la qual passa a ser accessible pel nom del paràmetre formal.

Mentre molts llenguatges aporten funcionalitats similars a la utilitzada en pseudocodi (una partícula especial, com `var`, que indica que el pas de paràmetre és per referència), el llenguatge C obliga a treballar directament amb les direccions de memòria, per la qual cosa és imprescindible la utilització de punters.

1.5. Introducció al tipus punter en C

En aquest apartat estudiarem un tipus de variable molt important en la programació modular en llenguatge C: la variable `punter`.

> Un `punter` és una variable destinada a contenir una direcció de memòria.

És a dir, un punter que està emmagatzemant un valor conté una direcció de memòria a la qual es pot accedir per a llegir-ne o modificar-ne el contingut.

Els punters s'utilitzen per a apuntar a conjunts de dades i manipular-los, per a apuntar a blocs de memòria assignats dinàmicament i per a permetre el pas d'arguments per referència en les crides a funcions.

Quan es treballa amb punters és freqüent trobar errors motivats per la creació de punters que apunten a alguna part de memòria inesperada, i es produeixen violacions de la memòria quan s'intenta modificar els continguts apuntats. Per tant, cal posar la màxima atenció quan es treballa amb punters, ja que el llenguatge C suposa que el programador sap el que fa, com ho fa i per què ho fa. Aquests errors poden provocar destrosses importants i, fins i tot, l'aturada del sistema operatiu.

L'estudi dels punters és la part de més difícil assimilació en l'aprenentatge del llenguatge C, però no es pot obviar. Fins aquest moment hem anat passant sense que ens haguessin presentat aquest subjecte, però hem arribat al límit. És imprescindible fer servir els punters per a permetre el pas d'arguments per referència.

La utilització dels punters no es limita, però, al pas d'arguments per referència, sinó que és l'eina que facilita el llenguatge C per a la gestió de dades dinàmiques (llistes, piles, cues, arbres,...).

Introduïm, ara, les característiques imprescindibles dels punters per a permetre el pas d'arguments per referència.

1.5.1. Declaració de punters

La sintaxi per a declarar una variable de tipus punter és la següent:

```
<tipus_de_dada> *<nom_variable>;
```

En principi, un punter està pensat per a apuntar a dades d'un tipus determinat. L'excepció és quan es defineix un punter que apunta al tipus `void`. Deixem de banda ara, però, aquest cas.

Exemples de declaració de punters:

```
int *a;      /* a és una variable punter destinada a apuntar a una dada int */
char *s;     /* s és una variable punter destinada a apuntar a una dada char */
int *t[100]; /* t és una taula de 100 punters per apuntar cadascun a una dada int */
```

Fixeu-vos bé que tots aquests punters estan destinats a apuntar una zona de memòria però encara no n'apunten cap.

Quant ocupa en memòria un punter? L'espai requerit ha de ser el nombre de bytes necessaris per a especificar una direcció màquina. Valors típics són 2 o 4 bytes. És a dir, l'espai que ocupa un punter no té a veure amb el tipus de dada destinat a apuntar, sinó amb la manera com està direccionada la memòria a la màquina en què s'executa el programa.

1.5.2. Assignació de valor a punter

No perdeu de vista que un punter és una variable i, per tant, se li pot aplicar tot el que s'ha comentat sobre la qualificació de variables i els valors amb què, en certs casos, el compilador les inicialitza de manera automàtica.

El llenguatge C aporta la constant `NULL` (definida a varis fitxers de capçalera i que correspon al valor zero) que es pot assignar a un punter i indica que el punter no està apuntant a cap zona de memòria.

El llenguatge C garanteix que un punter que apunti a un objecte vàlid mai tindrà el valor `NULL`. Aquest valor s'utilitza per a indicar que hi ha hagut un error o que una determinada operació que ha de retornar un punter no ha acabat amb èxit.

En general no té sentit assignar enters a punters perquè és el sistema operatiu qui gestiona la memòria i per tant és aquest, i només aquest, qui coneix en tot moment quines direccions de memòria són lliures i quines són ocupades.

Per tant, com s'assigna l'adreça en què es troba una dada a una variable punter? Necessitem l'operador direcció de &. (!!)

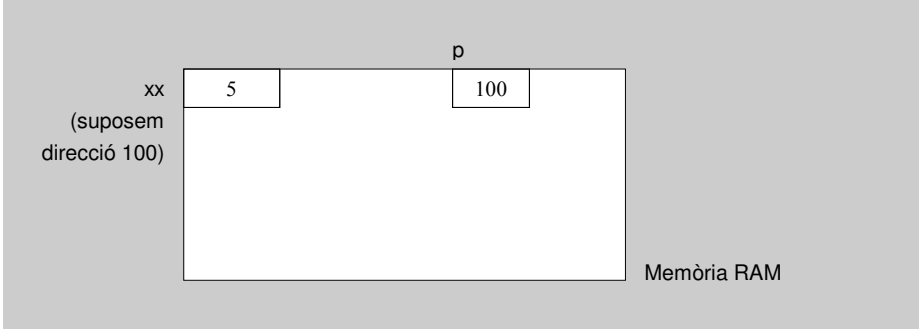
> L'operador unitari `&` retorna com a resultat la direcció del seu operand.

Per exemplificar la utilització de l'operador `&`, observis el següent codi que provoca la situació representada gràficament a la figura 4.

```
int xx = 5;  /* xx és una variable int que emmagatzema el valor 5 */
...
int *p; /* p és una variable punter a int que no apunta enlloc intencionadament */
/* si es vol assegurar que p no apunti enlloc caldria assignar-li el valor NULL */
...
p = &xx; /* p apunta a la direcció de memòria on es troba la variable xx */
```

La figura 4 mostra gràficament la situació

Figura 4. Exemple de variable punter `p` que conté la direcció 100 de la variable apuntada `xx`.



1.5.3. Accés a una dada mitjançant un punter

Coneixem com aconseguir que un punter apunti a una zona de memòria ocupada per una variable. Podem accedir al valor emmagatzemat en la posició de memòria apuntada pel punter? Evidentment! Necessitem l'operador d'indirecció *. (!!)

L'operador unitari `*` pren el seu operand com una direcció i dóna el seu contingut com a resultat.

Així, donada la situació:

```
int xx = 5;  /* xx és una variable int que emmagatzema el valor 5 */
int *p = &xx;/* p és una variable punter que apunta on es troba la variabl xx */
```

si volem conèixer el valor que hi ha a la posició de memòria apuntada per `p` (valor 5 a la posició 100 de la memòria) hem d'escriure `*p`.

És a dir, hem de deduir que `*p` és equivalent, a tots els efectes, a `xx` (variable apuntada per `p`). En efecte, les parelles següents d'instruccions són equivalents:

```
printf ("El valor de la variable xx és %d\n", xx);
printf ("El valor de la variable xx és %d\n", *p);

xx = 25;
*p = 25;

printf ("Introdueixi nou valor per xx: "); scanf ("%d", &xx);
printf ("Introdueixi nou valor per xx: "); scanf ("%d", p);
```

No veieu res d'estrany a les darreres dues línies? És clar que seguint la sintaxi que hem utilitzat fins ara per a la funció `scanf`, que consisteix a anteposar l'operador `&` davant el nom de la variable que s'ha d'emplenar, tal com es veu en `&xx`, caldria posar `&*p`.

Fixeu-vos que els operadors direcció de `&` i d'indirecció `*` són antagònics, de manera que l'un contraresta l'altre. Així, si `p` és el punter, resulta que `*p` és la dada on apunta `p` i `&*p` és la direcció de la dada a la qual apunta `p`, és a dir, el mateix `p`. (!!)

1.5.4. Importància del tipus d'objecte que es vol apuntar

Com coneix C el nombre de bytes que ha d'assignar a una variable des d'una determinada direcció? És a dir, quan escrivim:

```
*p = 25;
```

com sap el llenguatge si ha de considerar 25 com un enter de 4 (`int`) o 8 (`long`) bytes per tal d'assignar-lo a la posició a la qual apunta `p`? La resposta a aquest interrogant està en el fet que C pren com a referència

el tipus d'objecte apuntat en la seva declaració i assigna el nombre de bytes corresponent a aquell tipus.

L'exemple següent servirà per aclarir-nos la importància del tipus d'objecte apuntat en la declaració d'un punter.

```
/* u4n1p03.c */

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

void main(void)
{
   int i = 5, *pi = &i;
   float f = 10.3, *pf = &f;
   clrscr();
   printf ("El valor de i és %d\n", *pi);
   printf ("El valor de f és %g\n", *pf);
   pf = &i;
   pi = &f;
   printf ("El valor de i és %d\n", *pf); [1]
   printf ("El valor de f és %g\n", *pi); [2]
}
```

Trobareu el fitxer `u4n1p03.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

L'execució d'aquest programa escriu per pantalla quatre línies. Les dues primeres amb els resultats esperats:

```
El valor de i és 5
El valor de f és 10.3
```

però les dues últimes amb valors de previsió impossible. Per quin motiu?

El punter `pi` està destinat a apuntar a dades `int` i, en canvi, la instrucció `pi = &f` està agafant l'adreça de memòria on es troba el `float f` i l'assigna a `pi`. Per tant, `pi` està emmagatzemant la direcció de memòria en què s'inicia l'emmagatzematge de `f`. A la instrucció **`[2]`**, `*pi` subministra a la funció `printf` els bytes emmagatzemats a partir de l'inici del lloc en què es troba `f`, però interpretats com a `int` i fent servir només els bytes que utilitza el tipus `int` de la màquina. Evidentment, els bytes que representen el valor `10.3` de f com a `float`, interpretats com a `int` donen un altre valor, que és el que intentarà escriure la funció `printf` a la instrucció **`[2]`**. Un comportament semblant passa amb `i`, `pf` i la instrucció **`[1]`**.

Tots aquests problemes es deriven del fet que fem servir de manera incorrecta els punters `pi` i `pf`. Ara bé, el compilador ens ha avisat. En temps de compilació ha donat dos errors de tipus warning fixats precisament en les instruccions:

```
pi = &f;
pf = &i;
```

i amb text explicatiu, segons el compilador.

Així, els compiladors de *Borland C* acostumen a donar un error similar a:

```
Suspicious pointer conversion in function main
```

mentre que els compiladors *gcc* faciliten un missatge similar a:

```
Assignment from incompatible pointer type
```

En qualsevol cas el compilador ens avisa d'una conversió de tipus entre punters que no veu gaire clara.

Ja coneixeu la filosofia del llenguatge C: el programador ja sap el que es fa.

1.5.5. Punters en el pas d'arguments per referència

En el llenguatge C, per a passar arguments per referència o per variable cal fer servir els punters. Això està motivat perquè en el pas per referència o per variable, el paràmetre formal ha de rebre la direcció de memòria del paràmetre actual i, si de direccions es tracta, els punters apareixen pel mig.

Com que el paràmetre formal ha de rebre una direcció, haurà de ser obligatòriament un punter al tipus de dada a què correspon el paràmetre actual. Així, la codificació d'una acció o funció amb prototipus en pseudocodi:

```
funció/acció FA ( p1: enter, p2: caràcter, var p3: enter) ...
```

seria en llenguatge C:

```
[<tipus>/void] FA (int p1, char p2, int *p3);
```

Acabeu de veure que cal tenir en compte el pas per referència en el moment de declarar la funció, però també cal tenir-lo present en la crida de la funció i en la seva definició.

En efecte, si la crida en pseudocodi és semblant a:

```
var  c: caràcter; t: enter;  fivar
...
FA ( 10, c, t);
/* Els dos primers paràmetres poden ser constants o variables,
però el tercer ha de ser obligatòriament una variable ja que es
passa per referència */
```

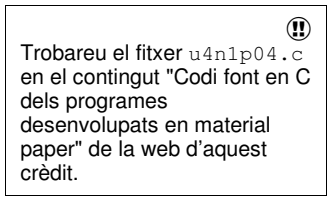
en el llenguatge C seria:

```
char c; int t;
...
FA (10, c, &t);
```

ja que cal passar al paràmetre formal `p3` la direcció en què es troba el paràmetre actual `t`.

A més de tenir en compte el pas per referència en la declaració (variables punter) i en la crida (operador d'indirecció &) de la funció, cal tenir-lo present en la definició de la funció. En efecte, en qualsevol accés que s'hagi de fer a la dada que correspon al paràmetre actual mitjançant el paràmetre formal (que és un punter) caldrà fer servir l'operador direcció de *.

Ara ja tenim els elements necessaris per a efectuar la codificació en llenguatge C de les dues versions típiques del subprograma per efectuar una lectura de la típica resposta S/N a una pregunta (versió com a funció que retorna el valor llegit i versió com a acció que facilita el valor llegit via paràmetre formal per referència):



Trobareu el fitxer `u4n1p04.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u4n1p04.c */

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

void llegir_SN_1 (unsigned int fil, unsigned int col, char *pc)
{
   int n;
   do
   {  gotoxy (col, fil);
      n = scanf ("%c", pc); neteja_stdin ();
   } while (!n || ( *pc != 'S' && *pc != 's' && *pc!= 'n' && *pc!= 'N'));
   if (*pc == 's') *pc='S';
   else if (*pc == 'n') *pc='N';
}

char llegir_SN_2 (unsigned int fil, unsigned int col)
{
   char c;
   int n;
   do
   {  gotoxy (col, fil);
      n = scanf ("%c", &c); fflush (stdin);
   } while (!n || ( c != 'S' && c != 's' && c!= 'n' && c!= 'N'));
   if (c == 's') c='S';
   else if (c == 'n') c='N';
   return c;
}

void main(void)
{
   char opc;
```

```
   clrscr();
   gotoxy (10,5); printf ("Es considera una persona amb sort (S/N)?");
   llegir_SN_1 (5, 56, &opc);
   gotoxy (10,7); printf ("Ha contestat: %c", opc);
   gotoxy (10,9); printf ("Està content d'aquesta resposta (S/N)?");
   opc = llegir_SN_2 (9, 54);
   gotoxy (10,11); printf ("Ha contestat: %c", opc);
}
```

1.5.6. Error d'accés a punter *NULL*

És molt freqüent trobar programes mal codificats en llenguatge C que acaben la seva execució amb missatges similars a:

```
Null pointer assignment
```

o

```
Violació de segment
```

L'execució del programa a vegades finalitza correctament i a vegades finalitza abans d'hora, amb missatges similars als anteriors. Per quin motiu? El programa següent és un exemple d'aquest problema.

```
/* u4n1p05.c */

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

void main(void)
{
   int *p=NULL;
   printf ("Entri un valor enter:"); scanf("%d",p);
   printf ("El valor introduït és %d\n",*p);
}
```


Trobareu el fitxer `u4n1p05.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

Observeu el punter `p` destinat a apuntar un `int`; en cap moment s'ha produït l'assignació corresponent. En la instrucció `scanf` es fa servir la direcció de memòria teòricament apuntada per `p` per a emmagatzemar el valor introduït per l'usuari. Però aquesta posició de memòria no existeix! El programa pot funcionar o no, vés a saber, i si acaba l'execució dóna l'avís que s'ha produït una assignació per mitjà de punter nul (punter que no apunta enlloc).

A més, en l'exemple `u4n1p05` hem tingut la precaució d'inicialitzar el punter `p` amb el valor `NULL`. Recordeu que les variables locals no s'inicialitzen implícitament i que, per tant, poden tenir valor "porqueria". Si executeu el programa sense la inicialització a `NULL`, pot ser que el missatge de tipus *NULL pointer assignment* aparegui o no

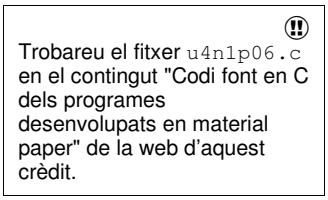
aparegui. El que sí podem assegurar és que el programa és incorrecte, ja que `p` pot estar apuntant a una direcció qualsevol de memòria, el valor de la qual serà modificat per l'execució de la instrucció `scanf`. I si la posició de memòria correspon a altres variables del programa? (En aquest cas no és possible ja que no hi ha cap altre variable). I si correspon a àrees de memòria utilitzades pel sistema operatiu? Evidentment, no es pot assegurar res sobre el funcionament d'un programa com aquest.

Mentre no sapigueu fer assignació dinàmica de memòria, absteniu-vos d'actuar com en l'exemple presentat. La manera correcta de treballar és declarar un enter (no el punter) i treballar amb la posició de memòria en què el programa ha col·locat l'enter, com es veu a la versió següent.

```
/* u4n1p06.c */

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

void main(void)
{
   int j;
   printf ("Entri un valor enter:"); scanf("%d",&j);
   printf ("El valor introduït és %d\n",j);
}
```

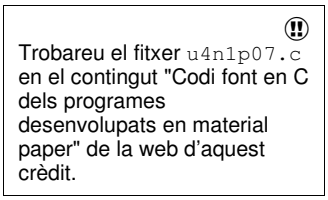

Trobareu el fitxer `u4n1p06.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

De tota manera, si es vol treballar amb un punter, cal declarar igualment l'enter `j` i el punter `p` i fer que el punter `p` apunti a l'enter `j` tal com veiem tot seguit:

```
/* u4n1p07.c */

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

void main(void)
{
   int j, *p;
   p = &j;
   printf ("Entri un valor enter:"); scanf("%d",p);
   printf ("El valor introduït és %d\n",*p);
}
```


Trobareu el fitxer `u4n1p07.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

1.5.7. Punters i taules

Per acabar aquesta introducció sobre els punters, heu de saber que el nom d'una taula en el llenguatge C és un punter constant al tipus de dada dels elements que emmagatzema la taula i que apunta a la posició de memòria en què es troba la primera casella de la taula.

És a dir, tenir:

```
int t[100];
```

implica que `t` és un punter a l'inici d'una zona de memòria que ocupa l'espai de 100 elements de tipus `int`.

També heu de conèixer l'existència del que s'anomena aritmètica de punters, que permet efectuar certes operacions aritmètiques amb aquests. ‼

En particular, si `p` és un punter a un tipus `T` que ocupa `n` bytes i `p` apunta a una variable `v` del tipus `T`, resulta que `p+1` apunta a una suposada variable de tipus `T`, que estaria situada en memòria a continuació de `v`; `p+2` apunta a una suposada variable de tipus `T` que estaria situada en memòria a continuació de l'anterior, i així successivament.

Això permet, entre altres coses, que qualsevol operació que es pugui dur a terme mitjançant la indexació de la taula es pugui també fer mitjançant punters. O sigui, fer:

```
int i;
...
for (i=0; i<100; i++) printf ("Posició %d: %d\n", i, t[i]);
```

és equivalent a:

```
int i;
...
for (i=0;  i<100; i++) printf ("Posició %d: %d\n", i, *(t+i));
```

Com és possible això? Quan el llenguatge avalua l'expressió `t[i]` sap que a partir de la direcció d'inici de la taula (és a dir, a partir de la direcció emmagatzemada a `t`) ha d'avançar `i` elements per a accedir al contingut de l'element especificat per l'índex. I com avalua el llenguatge l'expressió `*(t+i)`? Molt fàcil. Ja hem dit que `t` té la direcció d'inici de la taula. Pel que hem dit més amunt, `*t` és equivalent a la primera casella de la taula (posició zero), `*(t+1)` és equivalent a la segona casella de la taula (posició 1), i així successivament.

Per acabar, observeu que si tenim `int *p`, són equivalents `p = t` i `p = &t[0]`, ja que `t` té la direcció en què comença la taula, o sigui, la direcció en què es troba la primera casella, i l'expressió `&t[0]` ens proporciona precisament aquesta direcció. ‼

1.5.8. Punters a estructures

Suposem la situació:

```
struct T
{
   int c1;
   char c2;
   char c3[25];
} t;
struct T *p;
...
p = &t;
```

és a dir, tenim la variable `t` de tipus `T` a la qual apunta el punter `p`.

Sabem omplir els camps de la variable `t` de manera directa:

```
t.c1 = 25;
t.c2 = 'S';
strcpy (t.c3, "Supercalifragilístic");
```

però, i per emplenar-los amb la utilització del punter `p`? El punter `p` és un punter a la variable estructura. Per tant, `*p` és equivalent a la variable apuntada `t` i, per tant, les instruccions anteriors quedarien com a:

```
(*p).c1 = 25;
(*p).c2 = 'S';
strcpy ((*p).c3, "Supercalifragilístic");
```

El llenguatge C, per tal d'abreujar la notació, incorpora l'operador `->`, que es pot fer servir en lloc de la notació `(* ).`. És a dir, les instruccions anteriors quedarien com a: ‼

```
p->c1 = 25;
p->c2 = 'S';
strcpy (p->c3, "Supercalifragilístic");
```

1.6. Prototipus de funcions en C

Des de l'inici d'aquest crèdit hem fet servir funcions que aporta el llenguatge C, de les quals hem ofert una pseudodeclaració. En efecte, recordeu quan definíem les funcions que aporta el llenguatge C per al tractament de cadenes en què, per exemple, presentàvem la funció per copiar una cadena en una altra de la manera següent:

```
strcpy ( cadena destí, cadena origen);
```

Aquesta declaració és del tot incorrecta, tot i que ens va servir per a entendre la funcionalitat que proporciona. La sintaxi correcta, que trobareu en qualsevol llibre sobre el llenguatge C i en les ajudes dels mateixos entorns de desenvolupament, és:

```
char *strcpy ( char *destí, const char *origen);
```

sintaxi que en aquest moment ja podem utilitzar.

No ens oblidem d'un comentari explicatiu sobre la funció `scanf`. Enteneu ara el motiu pel qual les variables que s'han d'emplenar van precedides pel símbol `&`? Són variables que hem de passar a la funció `scanf` perquè aquesta les empleni; per tant, després d'executar la funció n'han d'haver variat el contingut i això implica que la transferència de dades hagi de ser per referència. Per què quan es llegeix una cadena no es posa el símbol `&`? El nom d'una cadena és un punter (una cadena és una taula de caràcters) i, per tant, ja és la direcció en què cal emplenar les variables amb la lectura efectuada per la funció `scanf`.

En els subapartats següents veurem les principals funcions que aporta el llenguatge C amb la seva sintaxi correcta.

1.6.1. Funcions d'entrada/sortida

Les funcions d'entrada/sortida són les següents:

Escriptura amb format per la sortida estàndard

```
#include <stdio.h>
int printf (const char *format [, argument]...);
```

El qualificador const al davant d'una variable per referència impossibilita que la funció modifiqui el contingut de la variable.

Aquesta funció escriu, utilitzant el `<format>` especificat, una sèrie de caràcters a la sortida estàndard. Retorna un valor enter igual al nombre de caràcters escrits.

Lectura amb format per l'entrada estàndard

```
#include <stdio.h>
int scanf (const char *format [, argument]...);
```

Aquesta funció llegeix dades de l'entrada estàndard, les interpreta d'acord amb el `<format>` indicat i les emmagatzema en els arguments especificats que han de ser passats per referència. Retorna un enter corresponent al nombre de dades llegides i assignades.

Netejar la memòria intermèdia (buffer) de l'entrada estàndard

```
#include <stdio.h>
int fflush (FILE *stream);
```

Neteja el buffer associat amb l'stream, que serà stdin si volem netejar el buffer associat al teclat.

A la unitat didàctica "Tractament de la informació a memòria externa. Gestió d'fitxers seqüencials" veureu la gestió d'fitxers i, en especial, el tipus `FILE` del llenguatge C.

El tipus `FILE`

De moment encara no coneixem el tipus `FILE`, que es fa servir per a treballar amb fitxers en disc. En el llenguatge C totes les entrades i sortides estan associades a tipus `FILE`. Així, l'entrada estàndard –`stdin`– i la sortida estàndard –`stdout`– són punters a `FILE` ja declarats en el llenguatge.

Llegir un caràcter per l'entrada estàndard

```
#include <stdio.h>
int getchar (void);
```

Llegeix un caràcter de l'entrada estàndard. Retorna el caràcter llegit o el caràcter EOF (*end of file*) si s'ha detectat el final del fitxer o s'ha produït un error. És equivalent a la funció `scanf` quan s'utilitza per a llegir un caràcter.

El caràcter EOF

Les funcions que estem presentant es fan servir en entrada i sortida estàndards, les quals acostumen a ser el teclat i la pantalla. Sabeu, però, que aquestes entrades i sortides es poden redireccionar de manera que s'efectuïn sobre un fitxer en disc, per exemple. En tal cas té molt de sentit poder arribar al final, cosa que es detecta amb el caràcter `EOF` després d'haver intentat llegir el caràcter que ja no existeix perquè s'ha arribat al final. Això és important remarcar-ho, ja que altres llenguatges obliguen a esbrinar si hi ha encara un caràcter per llegir abans de llegir-lo. El llenguatge C, en aquest aspecte, actua amb fets consumats: cal llegir i després cal comprovar si s'ha llegit alguna cosa.

Té sentit la possibilitat del caràcter `EOF` quan es treballa amb el teclat? La resposta és afirmativa. Aquest caràcter correspon a la pulsació de les tecles `<Ctrl+D>` en sistemes operatius Linux o `<Ctrl+Z>` en sistemes operatius Windows.

Escriure un caràcter per la sortida estàndard

```
#include <stdio.h>
int putchar (int c);
```

Escriu un caràcter (l'argument) a la sortida estàndard. Retorna el mateix caràcter escrit o un `EOF` si succeeix algun tipus d'error. És equivalent a la funció `printf` quan s'utilitza per a escriure un caràcter.

Lectura de pulsació de tecla

En els compiladors de la familia *Borland*, i no pas en els compiladors *gcc*, existeixen les següents funcions:

```
#include <conio.h>
int getch (void);
int getche (void);
```

La funció `getch` llegeix un caràcter del teclat sense visualitzar-lo en la pantalla (sense eco), mentre que la funció `getche` llegeix un caràcter de teclat visualitzant-lo en el monitor (amb eco). Retornen el caràcter llegit. Aquestes funcions efectuen la lectura en el moment que detecten la pulsació de tecla, és a dir, no esperen la pulsació de la tecla `<intro>` per a validar l'entrada de l'usuari.

Pulsació de tecles especials

El resultat és un byte quan la tecla premuda es correspon amb un dels caràcters de la taula `ASCII`. El resultat són dos bytes quan la tecla o combinació de tecles premudes es correspon amb alguna de les especificades en la taula dels codis estesos; per exemple, F1 produeix dos bytes, el primer és zero i el segon és el que identifica la tecla. Per això, en aquests casos cal cridar la funció `getch` o `getche` dues vegades, ja que la segona proporciona el codi que identifica la tecla.

Lectura d'una cadena de caràcters per l'entrada estàndard

```
#include <stdio.h>
char *gets(char *var);
```

La funció `gets` llegeix una línia de l'entrada estàndard i l'emmagatzema a la cadena de caràcters especificada per `var`. Aquesta variable representa la cadena de caràcters que contindrà tots els caràcters teclejats, excepte el caràcter final de línia `'\n'` que és automàticament canviat pel caràcter `'\0'`.

És responsabilitat del programador haver destinat espai suficient a la variable `var`. El llenguatge C omple a partir de l'inici de la cadena fins que troba el final de línia a l'entrada, amb la qual cosa, si `var` no és prou gran, pot enregistrar dades en espai corresponent a altres variables.

Aquesta funció retorna un punter a la cadena de caràcters llegida, és a dir, retorna la cadena de caràcters llegida, cosa interessant perquè permet utilitzar aquesta funció com a argument d'una altra funció. Així mateix, un valor `NULL` per a aquest punter indica error o final d'entrada (`EOF`).

A diferència de la funció `scanf`, permet l'entrada d'una cadena de caràcters formada per diverses paraules separades per espais en blanc.

Escriure una cadena de caràcters per la sortida estàndard

```
#include <stdio.h>
int puts (const char *var);
```

Aquesta funció escriu la cadena de caràcters apuntada per `var` a la sortida estàndard i canvia el valor nul `'\0'` pel caràcter de nova línia `'\n'`. Retorna un valor positiu (corresponent al codi `ASCII` del darrer caràcter escrit) si s'executa satisfactòriament; en cas contrari, retorna el valor `EOF`.

1.6.2. Funcions de tractament de cadenes

Repassem els prototipus (amb sintaxis correcta) de les instruccions de tractament de cadenes que ja coneixem i presentem noves funcions.

Comparació de cadenes

```
#include <string.h>
size_t strlen (char *cadena);
int strcmp (const char *cadena1, const char *cadena2);
int strncmp (const char *cadena1, const char *cadena2, size_t llarg);
```

Recordem la seva funcionalitat, que consisteix a retornar un valor numèric enter que té diversos significats. El tipus `size_t` acostuma a ser sinònim d'`unsigned int`. Cal, però, fer servir `size_t`, ja que ens assegura compatibilitat en altres plataformes en què aquest tipus pugui correspondre a un tipus fonamental del llenguatge diferent a `unsigned int`.

Assignació de cadenes

```
#include <string.h>
char *strcpy (char *destí, const char *origen);
char *strncpy (char *destí, const char *origen, size_t llarg);
```

Totes retornen un punter a la cadena `destí`, de manera semblant a la funció `gets`.

Concatenació de cadenes

```
#include <string.h>
char *strcat (char *destí, const char *origen);
char *strncat (char *destí, const char *origen, size_t llarg);
```

Totes retornen un punter a la cadena `destí`, de manera semblant a la funció `gets`.

Recerca dins cadenes

```
#include <string.h>
char *strchr (const char *cadena, int c);
```

Retorna un punter a la primera aparició de `c` dins la cadena o un valor `NULL` si el caràcter no hi és. El caràcter `c` pot ser el caràcter nul `'\0'`.

```
#include <string.h>
char *strrchr (const char *cadena, int c);
```

Retorna un punter a la darrera aparició de `c` dins la cadena o un valor `NULL` si el caràcter no hi és. El caràcter `c` pot ser el caràcter nul `'\0'`.

```
#include <string.h>
size_t strcspn (const char *cad1, const char *cad2);
```

Retorna la posició del primer caràcter de `cad1` que pertany al conjunt de caràcters continguts en `cad2`. Aquest valor és la longitud de la subcadena de `cad1` formada pels primers caràcters no inclosos a `cad2`. Si no hi ha cap caràcter de `cad1` que pertanyi a `cad2`, el resultat és la posició del caràcter `'\0'` de `cad1`, o sigui, la longitud de `cad1`.

```
#include <string.h>
size_t strspn (const char *cad1, const char *cad2);
```

Retorna la posició del primer caràcter de `cad1` que no pertany al conjunt de caràcters continguts en `cad2`. Aquest valor és la longitud de la subcadena de `cad1` formada pels primers caràcters inclosos a `cad2`.

```
#include <string.h>
char *strstr (const char *cad1, const char *cad2);
```

Retorna un punter a la primera aparició de `cad2` en `cad1` o un valor `NULL` si `cad2` no es troba dins de `cad1`.

Conversió de majúscules i minúscules per a un caràcter

```
#include <ctype.h>
int tolower (int c);
int toupper (int c);
```

Tot i no ser funcions de tractament de cadenes, incloem en aquest apartat les funcions tolower i toupper que permeten obtenir la traducció

a minúscula i a majúscula de qualsevol caràcter. La funció `tolower` retorna el caràcter `c` en minúscula i la funció `toupper` en majúscula, si la conversió és procedent. Cal tenir present que només converteixen els caràcters inclosos en els intervals `A..Z` i `a..z`.

Conversió majúscules-minúscules per a cadenes

En els compiladors de la familia *Borland*, i no pas en els compiladors *gcc*, existeixen les següents funcions:

```
#include <string.h>
char *strlwr (char *cadena);
char *strupr (char *cadena);
```

La funció `strlwr` converteix les lletres majúscules en minúscules i la funció `strupr` converteix les lletres minúscules en majúscules. Retornen el punter a la pròpia cadena, ja transformada.

Molta atenció! Les lletres que sap gestionar són les que hi ha en els intervals `A...Z` i `a...z` de la taula ASCII. Per tant, no ens convertirà altres caràcters que per a nosaltres són lletres, com és el cas de les lletres accentuades, la ç, la ñ, etc.

Podria ser interessant tenir, programades per nosaltres, unes funcions que transformessin majúscules en minúscules i viceversa i que incloguessin totes les lletres de la nostra llengua.

Els compiladors *gcc* no faciliten les funcions `strlwr` i `strupr` però sí les funcions `tolower` i `toupper`. Per a mantenir, en els exemples que anem desenvolupant en aquest material, la compatibilitat entre plataformes *Borland* i *gcc*, sembla lògic implementar les funcions `strlwr` i `strupr` per a la plataforma *gcc* incloent-les a el fitxer `compat.h` que hem anat construint.

La declaració i definició de les funcions a incloure dins `compat.h` són:

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

char * strlwr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)tolower(s[i]);
   return s;
}

char *strupr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)toupper(s[i]);
   return s;
}
```

El fitxer `compat.h` el varem iniciar a l'apartat "Exemples d'utilització de l'estructura `repetir...fins`" de la unitat didàctica "Estructures de control".

El següent exemple mostra la utilització de les funcions `strlwr` i `strupr`.

```
/* u4n1p08.c */

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

void main(void)
{
   char nom[]="Pepito Grillo";

   clrscr();
   printf ("El nom en el seu estat inicial: %s\n",nom);
   printf ("El nom convertit a majúscules : %s\n",strupr(nom));
   printf ("El nom convertit a minúscules : %s\n",strlwr(nom));
   printf ("El nom en el seu estat final  : %s\n",nom);
}
```

Trobareu el fitxer `u4n1p08.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

El resultat que s'obté en executar el programa és:

```
El nom en el seu estat inicial: Pepito Grillo
El nom convertit a majúscules : PEPITO GRILLO
El nom convertit a minúscules : pepito grillo
El nom en el seu estat final  : pepito grillo
```

Escriptura amb format cap una cadena

```
#include <stdio.h>
int sprintf (char *cadena, const char *format [, argument]...);
```

El qualificador const al davant d'una variable per referència impossibilita que la funció modifiqui el contingut de la variable.

Aquesta funció escriu, de manera similar a com ho fa la funció `printf`, utilitzant el `<format>` especificat, una sèrie de caràcters a la cadena que s'indica en el primer argument. Retorna un valor enter igual al nombre de caràcters escrits.

Cal que `cadena` disposi de l'espai igual o superior al necessari. Del contrari es podrà destruir variables contigües en l'espai de memòria.

Lectura amb format des d'una cadena

```
#include <stdio.h>
int sscanf (char *cadena, const char *format [, argument]...);
```

Aquesta funció llegeix, de manera similar a com ho fa la funció `scanf`, dades de la cadena indicada en el primer argument, les interpreta d'acord amb el `<format>` indicat i les emmagatzema en els arguments especificats que han de ser passats per referència. Retorna un enter corresponent al nombre de dades llegides i assignades.

1.6.3. Funcions per a conversió de dades

En ocasions, podem trobar-nos amb la necessitat d'efectuar una conversió de cadenes a nombres i viceversa.

```
#include <stdlib.h>
double atof (const char *cadena);
int atoi (const char *cadena);
long atol (const char *cadena);
```

Aquestes funcions construeixen valors numèrics (`double`, `int`, `long`) a partir de les cadenes passades per argument. La construcció es fa des de l'inici de la cadena i fins que troben un caràcter no reconegut com a part del nombre que s'ha de construir.

És molt important, en la utilització d'aquestes funcions, incorporar els corresponents `include`. En cas contrari, en temps d'execució, el comportament de les funcions pot ser que no sigui el que s'espera.

```
#include <stdlib.h>
char *fcvt (double num, int decs, int *pdec, int *signe);
```

La funció `fcvt` construeix una cadena de caràcters a partir d'un nombre real `num` i retorna un punter al seu inici. Inclou tants dígits després del punt decimal com indiqui `decs`, arrodonint si és menester. No inclou, a la cadena, el punt decimal, però retorna la posició on pertocaria (a partir de zero) dins `pdec` i a `signe`, hi retorna zero si el nombre era positiu o un valor diferent de zero si era negatiu.

El següent exemple mostra la utilització d'aquestes funcions de conversió:

```
/* u4n1p09.c */

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

void main(void)
{
   char cadena_entera[]="12345";
   char cadena_real[]="23.467";
   double v1=243.42734, v2=-2437.21732;
   char cadena[255];
   int posicio_punt_decimal;
   int signe;

   int valor_enter = atoi(cadena_entera);
   double valor_real = atof (cadena_real);
```

Trobareu el fitxer `u4n1p09.c` en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
   printf ("La cadena %s com a valor enter: %d\n",cadena_entera,valor_enter);
   printf ("La cadena %s com a valor real: %lf\n",cadena_real,valor_real);

   printf ("\nFuncionament de la funció fcvt:\n");

   strcpy(cadena,fcvt(v1,1,&posicio_punt_decimal,&signe));
   printf ("\nCadena a partir de %lf indicant 1 decimal: %s\n",v1,cadena);
   printf ("Informa que el punt decimal pertoca a la posició %d\n",posicio_punt_decimal);
   printf ("i el signe és %d (0->positiu, !=0->negatiu)\n",signe);

   strcpy(cadena,fcvt(v2,2,&posicio_punt_decimal,&signe));
   printf ("\nCadena a partir de %lf indicant 2 decimals: %s\n",v2,cadena);
   printf ("Informa que el punt decimal pertoca a la posició %d\n",posicio_punt_decimal);
   printf ("i el signe és %d (0->positiu, !=0->negatiu)\n",signe);
}
```

El resultat que s'obté en executar el programa és:

```
La cadena 12345 com a valor enter: 12345
La cadena 23.467 com a valor real: 23.467000

Funcionament de la funció fcvt:

Cadena a partir de 243.427340 indicant 1 decimal: 2434
Informa que el punt decimal pertoca a la posició 3
i el signe és 0 (0->positiu, !=0->negatiu)

Cadena a partir de -2437.217320 indicant 2 decimals: 243722
Informa que el punt decimal pertoca a la posició 4
i el signe és 1 (0->positiu, !=0->negatiu)
```

1.6.4. Funcions matemàtiques

Les declaracions per les funcions matemàtiques es troben, majoritàriament, en el fitxer de capçalera `math.h`. És molt important incloure aquest fitxer en el principi del programa font, ja que si no, en temps d'execució, ens podem trobar amb situacions estranyes i no desitjades.

Algunes de les funcions matemàtiques (trigonomètriques, logarítmiques, etc.) acostumen a utilitzar-se en programes que han d'efectuar càlculs matemàtics per a àrees de la ciència o l'enginyeria. No se solen fer servir mai en programes anomenats de gestió o d'empresa. Per a entendre els càlculs que desenvolupen aquestes funcions, cal tenir els coneixements corresponents de trigonometria, funcions exponencial i logarítmica i altres coneixements matemàtics. Suposem que els teniu o, si més no, que teniu les capacitats necessàries per a aconseguir-los a partir de llibres de matemàtiques.

No us preocupeu, però, si no els coneixeu en aquest moment. En aquest apartat es fa un breu apunt de les funcions matemàtiques que aporta el llenguatge C. No les acostumarem a utilitzar totes en els programes desenvolupats en aquest material i tampoc no les necessitareu si, en un futur, us dediqueu a la programació d'informàtica de gestió. Però si us dediqueu a desenvolupar aplicacions en àmbits científics o numèrics, es convertiran en la vostra mà dreta.

Funcions trigonomètriques

```
#include <math.h>
double acos (double x);
```

Proporciona l'arccosinus de `x` entre 0 i π radians. El valor `x` ha d'estar entre 1 i –1; en cas contrari es produeix un error de temps d'execució.

```
#include <math.h>
double asin (double x);
```

Proporciona l'arcsinus de `x` entre –π/2 i π/2 radians. El valor `x` ha d'estar entre 1 i –1; en cas contrari es produeix un error de temps d'execució.

```
#include <math.h>
double atan (double x);
```

Proporciona l'arctangent de `x` entre –π/2 i π/2 radians.

```
#include <math.h>
double atan2 (double y, double x);
```

Proporciona l'arctangent de `y/x` entre –π i π radians.

```
#include <math.h>
double cos (double x);
double sin (double x);
double tan (double x);
double cosh (double x);
double sinh (double x);
double tanh (double x);
```

Proporcionen, respectivament, cosinus, sinus, tangent, cosinus hiperbòlic, sinus hiperbòlic i tangent hiperbòlica de `x`, el valor dels quals es considera en radians.

Funcions potència - exponencial - logarítmica

```
#include <math.h>
double exp (double x);
double pow (double x, double y);
```

Proporcionen, respectivament, el resultat de la potència e^x (on `e` = = `2.718282...`) i x^y (on `x` i `y` han de tenir valors vàlids per l'operació).

```
#include <math.h>
double sqrt (double x);
```

Proporciona l'arrel quadrada de x.

```
#include <math.h>
double log (double x);
double log10 (double x);
```

Proporcionen, respectivament, el logaritme natural i el logaritme decimal de `x`.

Funcions d'arrodoniment

```
#include <math.h>
double ceil (double x);
```

Proporciona el valor `double` corresponent al menor valor enter major o igual a `x`.

```
#include <math.h>
double floor (double x);
```

Proporciona el valor `double` corresponent al major valor enter menor o igual a `x`.

```
#include <math.h>
double fabs (double x);
long labs (long x);
int abs (int x);
```

Proporcionen el valor absolut corresponent a `x`.

1.6.5. Funcions diverses

En veurem algunes tot seguit.

Posicionament del cursor en pantalla

Les següents funcions `gotoxy`, `wherex` i `wherey` només existeixen en els compiladors *Borland*.

```
#include <conio.h>
void gotoxy (int col, int fil);
```

A l'apartat "Instruccions d'utilització freqüent" de la unitat didàctica "Introducció a la programació", es va presentar una versió d'fitxer `conio.h` per a compiladors *gcc* que hem anat utilitzant al llarg del material i que incorpora una versió de la funció `gotoxy` per a compiladors *gcc*.

Posiciona el cursor a les coordenades de pantalla corresponents a la columna `col` (a partir de columna 1) i a la fila `fil` (a partir de fila 1).

```
#include <conio.h>
int wherex (void);
int wherey (void);
```

Retornen respectivament la columna i la fila on es troba situat el cursor.

Llibreria `ncurses` per a *gcc*

Els compiladors *gcc* faciliten la llibreria `ncurses` per a una completa gestió de pantalla. Fins i tot hi ha paquets de programari lliure que permeten, a partir de codi generat per a compiladors *Borland* on s'usen les funcions de `conio.h`, generar aplicacions compilades i enllaçades amb *gcc*.

Crida al sistema

```
#include <stdlib.h>
int system (const char *<cadena>);
```

Permet enviar qualsevol ordre al sistema operatiu, la qual es transfereix a través de l'argument. Retorna zero si tot és correcte i –1 si hi ha hagut error.

Nombres aleatoris

En ocasions pot ser necessari utilitzar nombres aleatoris per a fer simulacions, càlculs de probabilitats, etc. Els llenguatges acostumen a aportar utilitats per a generar nombres aleatoris. En realitat no són aleatoris del tot, ja que el llenguatge fa servir una rutina de generació i, per tant, segueix sempre la mateixa mecànica. Diguem, doncs, que es tracta de nombres pseudoaleatoris.

```
#include <stdlib.h>
int rand (void);
```

Retorna un nombre pseudoaleatori enter, entre 0 i el valor màxim per a un `int`.

```
#include <stdlib.h>
void srand (unsigned int arg);
```

L'hora del processador és un exemple d'argument que varia a cada execució.

Fixa el punt d'inici (anomenat llavor), per a generar nombres pseudoaleatoris, és a dir, inicialitza el generador rand de nombres pseudoaleatoris segons el valor de l'argument. Quan aquesta funció no s'utilitza, el valor del primer nombre pseudoaleatori generat és sempre el mateix per a cada execució. Si fem servir aquesta funció amb un argument que variï a cada execució, ens estarem apropant a una veritable generació de nombres pseudoaleatoris.

Control temporal

```
#include <time.h>
clock_t clock (void);
```

Retorna el temps utilitzat pel processador en el procés en curs (si estem en un sistema multiprocés) mesurat en tics de rellotge. Per a tenir la mesura en segons cal dividir aquest resultat per la constant CLK_TCK definida a el fitxer de capçalera `time.h`. Dóna com a resultat –1 si no és possible retornar el temps utilitzat.

El tipus `clock_t` acostuma a ser sinònim de `long`. Cal, però, utilitzar `clock_t` perquè ens assegura compatibilitat en altres plataformes on aquest tipus pugui correspondre a un tipus fonamental del llenguatge diferent de `long`.

El sistema operatiu DOS no és multiprocés. En aquest sistema, aquesta funció retorna el temps transcorregut des que el processador s'ha posat en marxa.

```
#include <time.h>
time_t time(time_t *seg);
```

Retorna el nombre de segons transcorreguts des de les 0 hores del dia 1 de gener de 1970.

```
#include <time.h>
char *ctime(const time_t *seg);
```

Converteix el temps emmagatzemat com un valor `time_t` en una cadena de caràcters en la notació anglesa:

```
Mon Jun 28 9:30:00 1999\n\0
```

El tipus `time_t` acostuma a ser sinònim de `long`. Cal, però, utilitzar `time_t` perquè ens assegura compatibilitat en altres plataformes en què aquest tipus pugui correspondre a un tipus fonamental del llenguatge diferent de `long`.

```
#include <time.h>
struct tm *localtime(const time_t *seg);
```

Interpreta el nombre de segons de l'argument com la quantitat de segons transcorreguts des de les 0 hores del dia 1 de gener de 1970 i dóna com a resultat la data i l'hora que corresponen a la quantitat interpretada. El resultat és emmagatzemat en una estructura estàtica

de tipus `tm` que és enregistrada cada vegada que s'executa la funció. Aquesta estructura és definida com:

```
struct tm
{
   int tm_sec;     /* Segons entre 0 i 59 */
   int tm_min;     /* Minuts entre 0 i 59 */
   int tm_hour;    /* Hores entre 0 i 23 */
   int tm_mday;    /* Dia del mes entre 1 i 31 */
   int tm_mon;     /* Mes entre 0 (gener) i 11 (desembre) */
   int tm_year;    /* Any (actual menys 1900) */
   int tm_wday;    /* Dia de la setmana entre 0 (diumenge) i 6 (dissabte) */
   int tm_yday;    /* Dia de l'any entre 0 (1 de gener) i 365 */
   int tm_isdst;   /* Conté 1 si l'argument correspon a una data i 0 altrament */
}
```

1.7. Arguments en la línia d'ordres en programes en C

Sabem codificar programes en llenguatge C que estan formats (si hem seguit un disseny descendent) per moltes funcions, les quals sabem declarar i definir, amb els paràmetres necessaris.

Sabem també que la funció `main` és la funció per la qual s'inicia l'execució d'un programa. Què passa si ens interessa que en iniciar-se l'execució del programa la funció `main` rebi uns valors determinats de l'exterior (des de la línia d'ordres que ha posat en execució el programa)? Ens cal poder definir una llista d'arguments a la funció `main`.

La crida des de l'exterior és clara:

```
nom_programa [valor_1 [...]]
```

Com es correspon aquesta crida a la funció `main` amb la seva declaració d'arguments? El prototipus genèric de la funció `main` és:

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

L'argument `argc` és un enter que indica el nombre d'arguments passats per la línia d'ordres, incloent-hi el nom del programa. El tractament és automàtic. El processador s'encarrega de subministrar aquest valor a la funció `main`. Aquest valor serà almenys 1.
L'argument `argv` és una taula d'`argc` punters a cadenes de caràcters de manera que cada punter de la taula apunta a un argument, de la manera següent:

```
argv[0] apunta a una direcció que conté el nom del programa
argv[i] apunta a una direcció que conté l'argument i
```

La funció `main` retorna, per defecte, un `int`, a no ser que s'indica `void`, com en tots els exemples desenvolupats fins el moment.

1.8. Implementació en C d'aplicacions organitzades en mòduls.

Ja sabem que una gran aplicació acostuma a estar organitzada en diferents mòduls, cadascun dels quals pot estar implementat en un o varis fitxers font, en els quals, a la vegada, podem tenir un o varis subprogrames. Sabem també que el compilador és l'encarregat de traduir cadascun dels fitxers font per a obtenir el corresponent fitxer objecte i l'enllaçador és el responsable d'efectuar l'enllaç de tots els fitxers objecte juntament amb les llibreries que corresponguin per a obtenir el fitxer executable. Ràpidament intuïm algunes situacions problemàtiques:

- Les definicions dels tipus de dades que puguem necessitar en diferents fitxers, ha de repetir-se en cadascun d'ells? Pensem que el compilador necessita conèixer la definició dels tipus de dades per a poder compilar cada fitxer font.

- El codi dels subprogrames cridats en un fitxer, ha de residir forçosament a el fitxer? Repetim el codi de subprogrames en diferents fitxers quan els subprogrames es crida en diversos fitxers? Pensem que el compilador necessita conèixer la declaració (prototipus) de tots els subprogrames cridats en un fitxer font.

La resposta a les dues situacions és que no hem de repetir definicions de tipus de dades ni definicions de subprogrames, doncs la repetició ens portaria els problemes següents:

- Possibles errors produïts en la reescriptura del codi (encara que sigui copiant i enganxant).

- Obligatorietat de repetir qualsevol modificació que s'efectuï en una funció (per tal de millorar o arreglar el seu comportament) en tots els programes en què aquella funció estigui escrita

- El programa enllaçador, en la fase d'enllaç, trobaria codi repetit.

Tots els llenguatges de programació incorporen mecanismes per a donar solució a aquestes situacions. El llenguatge C aporta l'ús dels fitxers de capçalera i les directrius de precompilació.

1.8.1. Fitxers de capçalera. Necessitat.

Un fitxer de capçalera, normalment amb extensió `.h`, és un fitxer destinat a contenir les definicions dels tipus de dades i les declaracions (prototipus) de les funcions.

Un fitxer font (extensió `.c`) que utilitzi tipus de dades i efectuï crides a funcions, ha d'incorporar en el seu inici (capçalera) la declaració dels fitxers de capçalera que contenen les definicions dels tipus de dades utilitzats i les declaracions (prototipus) de les funcions cridades. La inclusió s'efectua amb la directriu de precompilació `#include`.

A l'apartat "Introducció al precompilador C" de la unitat didàctica "Introducció a la programació" hem introduit el concepte de directriu de precompilació.

Aquesta pràctica no ens ve de nou, doncs en tots els programes hem anat utilitzant la directriu de precompilació `#include`.

Com a exemple, recuperem el fitxer `compat.h` que hem anat construint, per a mantenir la compatibilitat entre plataformes *Borland* i *gcc*. Vegem-ne la versió actual per a *gcc*:

```
/* compat.h per a compiladors gcc */

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

void neteja_stdin(void)
{
   int ch;
   do
      ch=getchar();
   while (ch != EOF && ch != '\n');

   clearerr(stdin);
}

char *strlwr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)tolower(s[i]);
   return s;
}

char *strupr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)toupper(s[i]);
   return s;
}
```

Observem que aquest fitxer té extensió `.h` i recordem que l'hem utilitzat en tots els programes on hem necessitat la utilització de les funcions `neteja_stdin()`, `strlwr()` i `strupr()`. Recordem que la versió de `compat.h` per a *Borland* no incorpora les dues darreres funcions doncs ja les facilita *Borland*.

És un fitxer de capçalera? Bé, la seva utilització via `#include` ens ho fa pensar... Però no, no és un correcte fitxer de capçalera!

Recordem que un fitxer de capçalera ha de contenir definició de tipus de dades (si és necessari, cosa que no succeeix en el cas que ens ocupa) i declaració (prototipus) de funcions. El nostre fitxer `compat.h` inclou, errròniament, la definició de les funcions.

Canviem, a partir d'ara, el fitxer `compat.h`, per una versió correcta que anomenarem `verscomp.h`:

```
/* verscomp.h   per a compiladors gcc */

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

void neteja_stdin(void);
char *strlwr(char *s);
char *strupr(char *s);
```

Però, llavors, on resideix el codi de les funcions? Doncs aquest s'ha d'ubicar en un fitxer `.c`, que pot tenir qualsevol nom, però s'aconsella que tingui el mateix nom que el fitxer de capçalera. Ara però, degut a aquesta separació entre `.h` i `.c`, resulta que tots els programes desenvolupats fins el moment, que contenien un únic fitxer `.c`, el qual compilàvem i enllaçàvem, passen a tenir més d'un fitxer .c i això ens porta a conèixer com aconseguir, en C, un programa executable a partir de més d'un fitxer `.c`.

Sabem que el compilador, en compilar cada fitxer font, necessita tenir coneixement sobre els tipus de dades utilitzats en el fitxer font i sobre les funcions cridades en el fitxer font. Per aquest motiu, en el fitxer `.c` incloem la `include` del seu fitxer de capçalera. En el nostre cas obtenim:

Per a veure com aconseguir el codi executable a partir de varis fitxers font en *Anjuta* i *Turbo C++*, vegeu l'activitat "Obtenció de programes a partir de varis arxius font".

```
/* verscomp.c   per a compiladors gcc */

#include "verscomp.h"      /* "" enlloc de <> !!! */

void neteja_stdin(void)
{
   int ch;
```

```
   do
      ch=getchar();
   while (ch != EOF && ch != '\n');

   clearerr(stdin);
}

char *strlwr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)tolower(s[i]);
   return s;
}

char *strupr(char *s)
{
   int i, n=strlen(s);
   for (i=0; i<n; i++) s[i]=(char)toupper(s[i]);
   return s;
}
```

On s'ha d'ubicar els fitxers `.h` i `.c`?

El fitxer `compat.h` utilitzat fins el moment el teníem ubicat en el directori on l'entorn de programació té ubicats tots els fitxers de capçalera que incorpora. Això, però, no és obligatori.

Ubicació dels fitxers de capçalera en els IDE Anjuta i Turbo C++

L'IDE Anjuta cerca els fitxers declarats amb la directiva `#include` dins el directori `/usr/include` entre altres directoris.

L'IDE Turbo C++ cerca els fitxers declarats amb la directiva `#include` dins el directori `xxx\include` on `xxx` correspon al directori on s'ha instal·lat Turbo C++.

Un fitxer de capçalera es pot situar en qualsevol ubicació, però llavors cal informar correctament al font en la directriu `#include`:

- La utilització de la sintaxis `#include <nom_fitxer>` indica al compilador que ha de cercar el fitxer dins els directoris preestablerts en l'entorn de desenvolupament.

- La utilització de la sintaxis `#include "nom_fitxer"` indica al compilador que ha de cercar el fitxer en la ruta que incorpora el propi nom (en l'actual directori de treball si no hi ha ruta) i, en cas de no trobar-lo, en els directoris preestablerts en l'entorn de desenvolupament.

En tots els programes desenvolupats fins el moment havíem incorporat `#include <compat.h>` i teníem el fitxer `compat.h` en el directori adequat. A partir d'ara, donat que ens apareix el fitxer `verscomp.c` el qual ens interessa tenir-lo en el directori de treball per a que ens sigui

fàcil la seva localització en la fase de compilació i enllaç, situarem també el corresponent `verscomp.h` en el directori de treball i utilitzarem la sintaxis `#include "verscomp.h"`. Aquesta decisió per als fitxers `verscomp.h` i `verscomp.c` la farem extensiva a tots els fitxers font que desenvolupem a partir d'aquest moment.

Aquesta nova forma de treballar (separació en arxius `.h` i `.c`) provoca canvis en l'organització global dels fitxers necessaris per a obtenir el programa executable. Així, d'una situació en la que teníem:

- el fitxer `compat.h` en el directori `INCLUDE` de l'entorn de programació, amb les definicions dels tipus de dades i de les funcions (prototipus i codi),

- i el fitxer `programa.c` que contenia el `main()` i possiblement altres funcions i que incorporava un `#include <compat.h>`,

passem a una nova situació en la que tenim:

- el fitxer `verscomp.h` en el directori de treball, amb les definicions dels tipus de dades i les declaracions (prototipus) de les funcions,
- el fitxer `verscomp.c` en el directori de treball, amb la definició de les funcions declarades a `verscomp.h` i amb una `#include "verscomp.h"` per a que el compilador verifiqui la coherència entre ambdós fitxers,
- i el fitxer `programa.c` en el directori de treball, amb el `main()` i possiblement altres funcions, i amb una `#include "verscomp.h"` per a que el compilador pugui verificar la coherència de les crides de funcions de `verscomp.h` efectuades en `programa.c` i de la definició dels tipus de dades de `verscomp.h` per les que es declaren variables a `programa.c`.

Per aconseguir el fitxer executable, es segueix el procés:

1) Compilem els fitxers `verscomp.c` i `programa.c` obtenint els fitxers objecte
2) Enllacem els fitxers objecte per obtenir el fitxer executable.

La situació presentada amb dos fitxers es produeix, en la pràctica, en múltiples fitxers `.c` i `.h` corresponents a diversos mòduls. Es compilen tots els `.c` i finalment, en la fase d'enllaç, s'obté el fitxer executable.

Necessitat dels fitxers de capçalera

És necessari utilitzar els fitxers de capçalera? Per què ens hem de complicar la vida distingint els arxius `.h` i `.c`?

I tant que és necessari! Pensem en una aplicació dissenyada per mòduls, amb la següent estructura de fitxers:

- Fitxer `f.c` amb el corresponent `f.h`
- Fitxer `f1.c`, amb el corresponent `f1.h`, que conté crides a funcions de `f` i/o utilitza tipus de dades definits a `f`.
- Fitxer `f2.c`, amb el corresponent `f2.h`, que conté crides a funcions de `f` i/o utilitza tipus de dades definits a `f`.
- Fitxer `p.c` que conté el `main()` i que conté crides a funcions de `f1` i/o de `f2` i /o utilitza tipus de dades definits a `f1` i/o `f2`.

Suposem, per un moment, que no tinguéssim la distinció entre `f.c` i `f.h` i que el corresponent contingut residís en un únic fitxer `f.hc` (l'extensió d'un arxiu per a ser utilitzat en una `#include` pot ser qualsevol). En tal situació, per a que el compilador pogués compilar sense problemes `f1.c` i `f2.c`, aquesta fitxers font haurien d'incorporar la `#include "f.hc"`. Fins aquí cap problema! Però fixem-nos que els fitxers objecte `f1` i `f2` resultants de la compilació de `f1.c` i `f2.c` incorporen la definició de les funcions existents a `f.hc` i, per tant, en la fase d'enllaç ens apareixerà un greu problema per duplicitat de definicions de funcions.

> El programa enllaçador no pot trobar, en la fase d'enllaç, més d'una definició per a una mateixa funció.

En canvi, si mantenim la separació entre `f.c` i `f.h`, tindrem:

- Fitxer `f.c` amb `#include "f.h"`
- Fitxer `f1.c`, amb `#include "f1.h"` i `#include "f.h"`
- Fitxer `f2.c`, amb `#include "f2.h"` i `#include "f.h"`
- Fitxer `p.c`, amb `#include "f1.h"` i `#include "f2.h"`

Ara, el compilador també podrà compilar sense problemes `f.c`, `f1.c` i `f2.c`. Els fitxers objecte `f1` i `f2` resultants de la compilació de `f1.c` i `f2.c` incorporen la declaració (prototipus sense la definició) de les funcions existents a `f.h`. El codi de les funcions de `f.h` es troba dins el fitxer objecte resultat de la compilació de `f.c`. Per tant, en la fase d'enllaç el programa enllaçador, en enllaçar els fitxers objecte de `f.c`, `f1.c` i `f2.c` no es trobarà amb duplicitat de definicions per una mateixa funció.

Malgrat pugui semblar que la separació en .h i .c no sempre és necessària, és convenient utilitzar-la sempre, doncs evitarem efectes no desitjats el dia que vulguem reutilitzar els fitxers en altres aplicacions. ‼

1.8.2. Precompilació condicional. Necessitat.

Plantegem-nos una aplicació estructurada en els següents arxius:

- Fitxer `f.c` amb `#include "f.h"`
- Fitxer `f1.c`, amb `#include "f1.h"` i fitxer `f1.h` amb `#include "f.h"`
- Fitxer `f2.c`, amb `#include "f2.h"` i fitxer `f2.h` amb `#include "f.h"`
- Fitxer `p.c`, amb `#include "f1.h"` i `#include "f2.h"`

El compilador sap compilar, sense problemes, `f.c`, `f1.c` i `f2.c`. Però fixeu-vos bé què succeeix en compilar `p.c`. Aquest fitxer incorpora dues vegades el contingut del fitxer `f.h`: un com a conseqüència de la `#include "f1.h"` i un altre com a conseqüència de la `#include "f2.h"`. Per tant, el compilador no podrà traduir el fitxer `p.c` donat que es trobarà duplicitat de definicions de tipus de dades i de declaracions de funcions.

> El programa compilador no pot trobar, en la fase de compilació d'un fitxer, més d'una definició per un mateix tipus de dada i més d'una declaració (prototipus) per a una mateixa funció.

Com es pot solucionar aquest problema? Cal fer servir les directrius de compilació condicional `#if`, `#elif`, `#else` i `#endif`. ‼

La seva sintaxi és la següent:

```
#if <expressió_1>
   [grup_de_línies;]
[#elif <expressió_2>
   grup_de_línies;]
[#elif <expressió_3>
   grup_de_línies;]
...
[#elif <expressió_n>
   grup_de_línies;]
[#else
   grup_de_línies;]
#endif
```

És a dir, és una estructura condicional similar a l'`opció` del pseudocodi. El preprocessador selecciona un únic grup_de_línies per passar-lo al compilador. El grup_de_línies seleccionat és aquell que es correspongui amb el primer valor vertader de l'expressió que segueix a `#if` o `#elif`. Si totes les expressions són falses, s'executarà el grup_de_línies a continuació de `#else` si existeix aquesta opció. Cada directriu `#if` ha de tenir el seu corresponent `#endif`. Les expressions han de ser de tipus enter i poden incloure únicament constants enteres, constants d'un únic caràcter i l'operador `defined`. En la utilització de l'operador `defined` està la solució al problema de duplicació de definicions presentat anteriorment.

L'operador `defined` pot ser utilitzat en una expressió de constants enteres d'acord amb la sintaxi següent:

```
defined (<identificador>)
```

L'expressió constant a què dóna lloc aquest operador es considera certa (diferent de zero) si `identificador` està actualment definit, i es considera falsa en cas contrari.

Amb aquesta funcionalitat, la solució al nostre problema passa per acostumar-nos a afegir el codi de precompilació tal i com es veu a continuació.

- Fitxer `f.h`

```
#if !defined (_NOM_F_)
   #define _NOM_F_
   <contingut que correspongui - el que hi havia a f.h>;
#endif
```

- Fitxer `f1.h`

```
#if !defined (_NOM_F1_)
   #define _NOM_F1_
   #include "f.h"
   <definició que correspongui - el que hi havia a f1.h>;
#endif
```

- Fitxer `f2.h`

```
#if !defined (_NOM_F2_)
   #define _NOM_F2_
   #include "f.h"
   <definició que correspongui - el que hi havia a f2.h>;
#endif
```

És a dir, cada fitxer `*.h` defineix una constant simbòlica (que pot tenir qualsevol nom, però és aconsellable una notació semblant a la que fa servir el mateix llenguatge C en els seus fitxers `*.h`, i que consisteix a definir les constants simbòliques amb el mateix nom del fitxer, en majúscules i entre un o dos símbols `'_'`) quan s'inclou el codi del fitxer. Vegem què passa quan el precompilador es trobi en el fitxer `p.c` les `include`:

```
#include "f1.h"
#include "f2.h"
```

Quan s'intenta executar la primera `include`, el preprocessador comprova si està declarada la constant simbòlica `_NOM_F1_` i com que no ho està, passa al compilador el codi corresponent, és a dir, efectua la `include` del fitxer `f.h` i posteriorment les definicions de dades i declaracions de funcions que corresponen a `f1`. Però, quan s'efectua la `include` del fitxer `f.h`, es torna a repetir el mateix procés. El preprocessador comprova si està declarada la constant simbòlica `_NOM_F_` i com que no ho està, passa al compilador el codi que correspon a la definició de dades i declaracions de funcions que corresponen a `f`. Però fixeu-vos que han quedat definides les constants simbòliques `_NOM_F1_` i `_NOM_F_`.

Quan s'intenta executar la segona `include` en el fitxer `p.c`, el preprocessador comprova la no-existència de la constant simbòlica `_NOM_F2_` i passa al compilador el contingut de `f2.h` que incorpora, en primer lloc, la `include` del fitxer `f.h` i aquesta inclusió comprova també la no-existència de la constant simbòlica `_NOM_F_` la qual ara sí existeix i, per tant, no duplicarà el contingut del fitxer `f.h`. D'aquesta manera el fitxer `p.c` és compilable sense problemes de duplicats.

Per acabar, volem comentar l'existència, entre altres, de les directrius de compilació següents:

```
#ifdef <identificador>     /* equivalent a if defined (<identificador>) */
#ifndef <idenficador> /* equivalent a if !defined (<identificador>) */
```

que provenen de versions antigues de C i que es mantenen per compatibilitat.

Malgrat pugui semblar que la utilització de les directrius de compilació condicional en els `.h` no sempre és necessària, és convenient utilitzar-la sempre, doncs evitarem efectes no desitjats el dia que vulguem reutilitzar els fitxers en altres aplicacions. ‼

2. Tipus abstractes de dades.

L'aparició de la programació estructurada en la dècada dels 60 va anar acompanyada de l'aparició del concepte tipus de dada, que ja ens és conegut, definit com un conjunt de valors que serveix de domini per a certes operacions.

Tots els llenguatges estructurats aporten un conjunt de tipus de dades i alguns permeten la definició de tipus de dades per part de l'usuari a partir dels tipus de dades aportats pel llenguatge.

Recordem la sintaxis en pseudocodi per a la definició de nous tipus de dades:

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

Recordem, també, la sintaxis que facilita el llenguatge C per a la definició de nous tipus de dades:

```
typedef <nom_actual_tipus> <nom_nou_tipus>;
```

> Els tipus de dades definits pel programador possibiliten la definició de valors de dades més propers al problema que es pretén gestionar.

Així, si en un programa hem de gestionar els dies de la setmana, ens pot convenir definir un tipus de dada `dia` que permeti treballar amb valors com `dilluns`, `dimarts`, `dimecres`, `dijous`, `divendres`, `dissabte` i `diumenge`, enlloc d'haver de tractar els dies amb els tipus de dades facilitats pel llenguatge (per exemple, amb nombres naturals de l'1 al 7).

Un altre exemple podria ser la definició d'una tupla (`struct` en C) per a la gestió de persones (amb camps com nom, cognoms, data de naixement, pes, alçada,...). Si ens interessa, i el llenguatge ens ho permet, podem definir el tipus `persona` de manera que podrem declarar variables d'aquest tipus, sense haver de pensar en com està definida.

La possibilitat d'utilitzar tipus de dades és fantàstica, però es queda curta, doncs a l'hora de gestionar les variables dels corresponents tipus hem de baixar a definició del tipus de dada i, en realitat, gestionar les dades dels tipus definits pel llenguatge i utilitzats en la definició del tipus de dada.

Pensem, en el llenguatge C, en la següent definició del tipus `persona`:

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

A partir d'aquesta definició, en els programes podrem utilitzar-la per a la declaració de variables:

```
persona p1, p2;
persona t[MAX];
```

Però, com hem de fer per omplir amb dades les variables `p1` i `p2`? No hi ha més remei que conèixer la definició del tipus persona i accedir camp a camp, segons les funcionalitats que té el tipus de dada de cadascun dels camps. Així, tindrem:

```
strcat(p1.nom,"Pepe");  /* Per introduir el nom */
p1.pes=75.8;            /* Per introduir el pes */
p1.sexe='H';            /* Per introduir el sexe */
```

És a dir, hem de conèixer que `nom` és una cadena i, per tant, cal utilitzar les instruccions adequades per donar valor a una cadena, que `pes` és un real i, per tant, cal utilitzar la instrucció d'assignació per a nombres reals, i que `sexe` és un caràcter i, per tant, cal utilitzar la instrucció d'assignació per a caràcters. Què hem guanyat amb el tipus de dada que no tinguéssim sense ell? Suposo que teniu la resposta: el tipus de dada ens permet una definició propera al problema a resoldre (la gestió de persones).

Se us acut alguna solució per a que el programador que utilitzi el tipus de dada `persona` no tingui per què conèixer la seva estructura interna per a gestionar variables de tipus `persona`? La resposta és clara: dissenyem accions i funcions adequats a la gestió del tipus de dada per a ser utilitzades pel programador usuari del tipus de dades.

És a dir, si per una variable del tipus de dades `persona` el programador ha de conèixer, en un determinat moment, l'edat que emmagatzema, li aniria molt bé disposar d'una funció similar a

```
unsigned int getEdatPersona (persona p);
```

que podria utilitzar sobre la persona en qüestió (passada per paràmetre) per obtenir la seva edat. I si de modificar l'edat es tracta, també li aniria molt bé disposar d'una funció similar a

```
int setEdatPersona (persona *p, unsigned int novaEdat);
```

que podria utilitzar sobre la persona en qüestió (primer paràmetre per referència) per canviar-li l'edat (segon paràmetre per valor) i per la que a més obtindria un valor enter indicador del possible error en la manipulació de l'edat (per exemple, que s'hagués passat una edat negativa).

Fixem-nos que les funcions `getEdatPersona` i `setEdatPersona` permeten que el programador usuari del tipus `persona` no tingui per què saber com és la implementació interna del concepte `edat`. El programador implementador del tipus persona sí que ha de conèixer la implementació interna del tipus `persona` per a implementar les diferents funcions.

Arribem, doncs, a la conclusió de que convé complementar els tipus de dades amb el conjunt d'operacions per a gestionar-los. Això s'anomena desenvolupament amb tipus abstractes de dades.

> Un tipus abstracte de dades (anomenat TAD) és un model matemàtic format per una col·lecció d'operacions definides sobre un conjunt de dades.

Els tipus abstractes de dades són la lògica evolució dels tipus de dades, doncs contenen la definició que aporta el tipus de dada més el conjunt d'operacions que sobre ell es poden aplicar juntament amb les diverses propietats que determinen inequívocament el seu comportament.

2.1. Especificació versus implementació.

El concepte de tipus abstracte de dades va ser introduït per diversos científics (citem com a pioners a S.N. Zilles, J.V. Guttag i el grup ADJ,

format por J.A. Goguen, J.W. Thatcher, E.G. Wagner y J.B. Wright) a mitjans de la dècada dels 70. Tots aquests autors varen coincidir en la necessitat d'utilitzar una notació formal per descriure el comportament de les operacions, no només per impedir qualsevol interpretació ambigua, sinó per identificar clarament el model matemàtic definit pel TAD.

En realitat, el concepte de TAD ja existeix en els llenguatges de programació estructurats sota la forma dels tipus predefinits, que es poden considerar com tipus abstractes. Per exemple, considerem el tipus de dades `short` que proporciona el llenguatge C; la definició del TAD corresponent consisteix en determinar:

- Quins són els seus valors? Els valors enters positius i negatius compresos entre –32768 (-2^{15}) i 32767 ($2^{15}-1$).
- Quines són les seves operacions? La suma, la resta, el producte, la divisió entera i el residu.
- Quines són les propietats que verifiquen aquestes operacions? N'hi ha moltes; per exemple: a+b = b+a; a*0 = 0, etcètera.

Resumint, es pot definir un tipus abstracte de dades com un conjunt de valors sobre els que s'aplica un conjunt donat d'operacions que verifiquin unes determinades propietats.

Per què abstracte? Aquest és un punt clau! El qualificatiu "abstracte" no significa "surrealista", sinó que prové de "abstracció" i respon al fet de que els valors d'un tipus poden ser manipulats mitjançant les seves operacions si es coneixen les propietats que aquestes verifiquen, sense que sigui necessari cap coneixement posterior sobre el tipus; en concret, la seva implementació a la plataforma és absolutament irrellevant. En el cas dels `short` del llenguatge C, qualsevol programa escrit en aquest llenguatge pot efectuar l'operació `x+y` (sent `x` i `y` dues variables `short`) amb la certesa de que sempre calcularà la suma dels enters `x` i `y`, sigui quina sigui la plataforma on s'executi el programa.

En altres paraules, la manipulació de les variables d'un tipus només depèn del comportament descrit en la seva especificació (en anglès, *specification*) i és independent de la seva implementació (en anglès, *implementation*). ‼

L'especificació d'un TAD consisteix en establir les propietats que el defineixen.

Per a que sigui útil, una especificació ha de ser:

- Precisa, només ha de dir allò realment imprescindible..
- General, adaptable a diferents contextos.
- Llegible, que serveixi com a instrument de comunicació entre l'especificador i els usuaris del tipus, per una banda; i entre l'especificador i l'implementador, per altra.
- No ambigua, que eviti posteriors problemes d'interpretació.

L'especificació del tipus, que és única, defineix totalment el seu comportament a qualsevol usuari que el necessiti. Segons el seu grau de formalisme, pot ser més o menys fàcil d'escriure i de llegir, i més o menys propensa a ser ambigua o incompleta.

La implementació d'un TAD consisteix en determinar una representació pels valors del tipus i en codificar les seves operacions a partir d'aquesta representació, tot això utilitzant un llenguatge de programació convencional.

Per a que sigui útil, una implementació ha de ser:

- Estructurada, per facilitar-ne el seu desenvolupament
- Eficient, per a optimitzar l'us de recursos de la màquina
- Llegible, per a facilitar el seu manteniment

Fixem-nos que l'especificació és única mentre que d'implementacions n'hi pot haver de diferents. (!!)

Altres definicions vinculades a aquests conceptes són:

- Univers d'especificació, format per tota la informació de l'especificació del TAD.
- Univers d'implementació, format per tota la informació de la implementació del TAD.

No és objectiu d'aquest material el coneixement de la nomenclatura emprada per a la definició formal dels diferents universos.

El programador usuari de TAD's només ha de conèixer l'especificació. El programador implementador de TAD's ha de conèixer la implementació que ha posar en pràctica.

2.2. Implementació en llenguatge C

Per a dur a la pràctica la implementació dels TADs en llenguatge C, utilitzarem els arxius de capçalera. Així, si en un programa hem de gestionar diferents tipus de dades, com ara persones i dates, i arribem a la conclusió de que ens convé un TAD per a la gestió de dates i un TAD per a la gestió de persones, ens organitzarem de la següent forma:

- Per a cadascun dels TADs a definir, tindrem dos arxius:
  - Arxiu `nom_tad.h` amb la definició del tipus de dada corresponent al TAD i les declaracions de les funcions (prototipus) per a la seva manipulació, amb una breu explicació del funcionament de cada funció.

    Aquest arxiu inclourà les directives de precompilació condicionals per evitar efectes col·laterals de duplicitat de definicions.

  - Arxiu `nom_tad.c` amb la definició de les funcions (codi) definides en el corresponent arxiu `nom_tad.h`.

    Aquest arxiu ha d'incorporar la instrucció `#include "nom_tad.h"` per a que el compilador comprovi la coherència de les definicions d'ambdós arxius.

- Per a la gestió del programa tindrem tants arxius com mòduls s'hagi cregut convenient en la fase d'anàlisi.

  Els diferents mòduls incorporaran tantes instruccions `#include "nom_tad.h"` com sigui necessari en funció dels TADs que hagin de gestionar.

2.3. Programació tradicional versus programació amb TADs.

Un enfocament tradicional de la programació (equació proposada per Wirth) diria que:

Programa = Dades + Algorismes

En aquesta equació, l'apartat algorisme es pot subdividir en dos tipus: algorismes de gestió de dades i algorismes de control del programa. Per tant, l'equació esdevé:

Programa = Dades + Algorismes de Dades + Algorismes de Control

Fixem-nos que la part de l'equació formada per les dades més els algorismes de dades no són altra cosa que la implementació dels TADs. Així, l'equació tradicional de la programació es converteix, en un enfocament de desenvolupament amb tipus abstractes de dades com:

Programa = Implementació de TADs + Algorismes de Control

Nikaus Wirth

Nascut el 15 de febrer del 1934 a Winterthur (Suïssa) és un dels gran científics de la informàtica.

El seu reconeixement bé causat per què va ser el cap de disseny de llenguatges de programació com Euler, Algol W, Pascal, Modula, Modula-2 i Oberon i també va dedicar gran part del seu temps en l'equip de disseny i implementació d'alguns sistemes operatius.

Així mateix és autor de dos textos clàssics en la enginyeria del programari. D'una banda, l'article de desenvolupament d'un programa per successius refinaments (*Program development by stepwise refinement*). D'altra, el seu llibre *Algoritmos + Estructuras de datos = Programa*, que va rebre un gran reconeixement i que encara avui és útil en l'ensenyament de la programació.

En 1984 va rebre el premi Turing pel desenvolupament dels llenguatges de programació i es va jubilar en 1999.

2.4. Programació amb TADs versus programació orientada a objectes

La programació amb TAD's és la precursora de la programació orientada a objectes (POO) o, dit d'altra manera, la POO és una extensió natural de la utilització dels tipus abstractes de dades.

Si als conceptes associats amb els TADs se'ls agrega l'herència i el polimorfisme, el resultat és la POO.

Programació orientada a objectes

Aquest apartat només pretén donar unes idees sobre la POO a partir dels coneixements actuals sobre TADs.

L'herència és una facilitat que permet estendre un tipus de dada (anomenat base), afegint-li més camps, obtenint nous tipus de dades (anomenats derivats). L'efecte de l'herència es pot simular en la majoria dels llenguatges tradicionals, però el resultat és un programa menys elegant i de programació més dificultosa.

La reutilització de components, tan anomenada pels adeptes a la POO, és precisament la possibilitat de definir una funció sobre un tipus derivat a

partir de la funció sobre el tipus base afegint-li la funcionalitat que correspongui a l'extensió efectuada. També es pot simular amb llenguatges tradicionals, però tornem a topar amb una programació menys elegant i més dificultosa.

L'altre component important de la programació orientada a objectes és la utilització del polimorfisme, consistent en que un subprograma (acció o funció) definit sobre un tipus base pugui operar de formes diferents sobre les diferents extensions que puguem tenir del tipus base, sense que el programador hagi d'especificar exactament la versió de subprograma a executar.

3. Gestió de dades ordenades en memòria interna

Sabem com omplir una taula afegint pel final i com fer recorreguts per les taules per tal de gestionar totes les dades emmagatzemades o per a cercar un element en concret. Els algorismes corresponents, no tenen en compte, però, cap informació referent a l'ordre relatiu de les dades emmagatzemades.

La recerca d'informació és una operació molt freqüent en el tractament de taules i es pot millorar si disposem de la informació ordenada. Com a exemple, plantegeu-vos la recerca d'un número de telèfon en una guia telefònica. Com el cerqueu? Ho teniu clar, oi?

Per començar heu de conèixer el nom de l'abonat i la població a què pertany el telèfon que busqueu. Una vegada es disposa d'aquesta informació cal agafar la guia telefònica que conté la població. Si estem en una central telefònica que disposa de totes les guies telefòniques, és molt probable que estiguin classificades per ordre alfabètic de províncies o zones, de manera que la detecció de la guia que ens interessa és molt ràpida. Una vegada la guia és a les nostres mans, cerquem la població i la trobem de manera ràpida perquè les poblacions estan ordenades dins la guia. Amb la població detectada, cerquem l'abonat, cosa també ràpida perquè els abonats estan ordenats pels cognoms i el nom, de manera que de seguida trobem el número de l'abonat o descobrim que no figura a la guia.

Aquest exemple ens serveix per a comprovar que la recerca en informació ordenada és molt més ràpida que en informació no ordenada. Però també ens serveix per a adonar-nos que cal tenir molt clar quin és el criteri d'ordre establert en el conjunt d'informació per tal d'assolir la millor estratègia de recerca. Així, per exemple, el mètode anterior per a trobar el número d'un abonat és vàlid si treballem amb les guies telefòniques d'abonats per poblacions, però no ens serveix gens si treballem amb les guies grogues (guies telefòniques per activitats econòmiques) o amb les guies blaves (guies telefòniques per carrers).

Així doncs, ens convé saber gestionar dades ordenades i aquesta gestió cal tenir-la en compte en tres situacions diferents: en inserir informació en un conjunt de dades ordenades (inserció ordenada), en cercar informació en un conjunt de dades ordenades (recerca ordenada) i en ordenar la informació d'un conjunt de dades.

3.1. Inserció ordenada

El mètode de la inserció ordenada consisteix a aconseguir que una taula s'empleni de manera ordenada, és a dir, que en cap moment deixi d'estar ordenada.

És clar que la primera inserció que es produeixi a la taula correspondrà situar-la a la primera posició i la taula restarà ordenada perquè encara no hi ha cap element. A partir d'aquest moment, ja sempre cal esbrinar la posició en què cal situar un nou element. Hi ha, bàsicament, dues possibilitats:

1) Que pertoqui situar-lo a la posició posterior a la darrera ocupada.
2) Que pertoqui situar-lo en una posició ja ocupada, la qual cosa implicarà desplaçar tots els elements situats a partir d'aquella posició una casella més endavant (cap a la fi de la taula).

Aquestes possibilitats s'han de complementar, evidentment, amb les preguntes següents:

- Queda espai a la taula per a afegir-hi nous elements?
- Hi pot haver elements repetits segons el criteri d'ordenació? I segons altres criteris?
- En el cas que es permetin elements repetits, on se situen: abans o després dels que ja existien?

3.1.1. Algorisme en pseudocodi

Suposarem un tipus de dada `TD` que conté un apartat `tc` de tipus `TC`. El tipus `TC` té definit un `ordre` (això no passa en tots els tipus de dades). Vegeu l'algorisme per a inserir elements en una taula d'elements de tipus `TD` de manera que es mantingui ordenada per l'apartat `tc`.

```
funció inserció_ordenada (var t: taula[MAX] de TD, var nt: natural, e: TD) retorna enter
és
/*
   Arguments:
      t  : Taula ordenada ASC pel camp tc on s'ha d'inserir ordenadament un element.
           Capacitat MAX.
      e  : Element que es vol inserir
      nt : Quantitat real d'elements existents a la taula
   Retorna:
      0  : Si s'ha pogut realitzar la inserció. En tal cas, t i n hauran estat modificats
      1  : No s'ha pogut realitzar la inserció per manca d'espai
*/
```

```
   var  i: natural;
        j: enter;
   fivar
   si n == MAX llavors retorna 1; fisi
   i = 0;
   /* La taula està ordenada ascendentment. No es demana controlar que no hi pugui haver
      elements repetits respecte de l'apartat que manté l'ordre. Per tant, si a la taula
      ja hi ha un element que en l'apartat tc conté el mateix valor que conté l'element
      que hi volem afegir, no hi ha d'haver problema. Normalment l'element que s'insereix
      s'afegeix al final de tots els elements amb igual valor en el camp tc que l'element
      que es vol inserir. */
   mentre i < nt i t[i].tc <= e.tc fer i = i + 1; fimentre
   /* Quan surt del mentre, la variable i conté la posició en què ha de situar-se
      l'element e. Pot donar-se el cas que s'hagi de situar a la posició n, cosa que
      succeeix si el valor e.tc és superior al valor dels camps t[i].tc per totes les
      posicions plenes de la taula (des de 0 fins a nt-1).
      Sigui quin sigui el valor de i, cal moure els elements de les posicions i a nt-1
      una posició cap avall, la qual cosa s'ha de fer començant per l'element de la
      posició nt-1 i, seguint cap amunt, fins a l'element de la posició i */
   j = nt-1;
   mentre j >= i fer
      t[j+1] = t[j] ;
      /* ALERTA: L'assignació potser no pot fer-se amb el signe =. Depèn del tipus TD */
      j = j-1;
   fimentre
   /* En aquest moment ja s'ha deixat l'espai de la posició i lliure per posar-hi
      l'element e */
   t[i] = e;
   /* ALERTA: L'assignació potser no pot fer-se amb el signe =. Depèn del tipus TD */
   nt = nt+1;
   retorna 0;
fifunció
```

Observeu que heu de definir la variable `j` com a `entera` i no pas com a `natural`, ja que pren el valor `-1` quan la taula està buida (quan `nt = 0`). Si la definim com a `natural` provocarem un error. (!!)

3.1.2. Exemple en llenguatge C

Volem fer un programa que permeti omplir una taula de persones (segons la definició de tipus `persona` existent a l'arxiu `persona1.h`) de manera que es mantingui ordenada pel camp nif.

```
/* u4n3p01.c */

#include "persona1.h"
#include "conio.h"
#include "generic1.h"
#include <string.h>
#include <stdio.h>

#define MAXPER 20

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per);
```

(!!) Trobareu els fitxers `u4n3p01.c` i `persona1.h` i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
void main(void)
{
   struct t_persona p[MAXPER];  /* Taula de MAX persones */
   unsigned int np=0;         /* Número de persones introduïdes a p */
   struct t_persona e;        /* Element a afegir a la taula */
   char opc='S';
   unsigned int i;
   while (opc=='S')
   {
      inputPersona (&e);
      insercio_ordenada (p, &np, e);
      if (np<MAXPER) pregunta ("Vol afegir-ne més?", &opc, "SN");
      else opc='N';
   }
   clrscr();
   gotoxy (1,1);
   printf ("NIF       PRIMER COGNOM        SEGON COGNOM         NOM
NAIXEMENT");
   gotoxy (1,2);
   printf ("-------------------------------------------------------------------------------
-");
   for (i=0; i<np; i++) outputPersonaTab (i+3, p[i]);
   missatge_ae("Premi tecla per finalitzar.   ");
   clrscr();
}

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per)
{
   unsigned int i;
   int j;
   if (*np == MAXPER) return 1;
   for (i=0; i<*np && strcmp(tp[i].nif,per.nif)<=0; i++);
   for (j=(*np)-1; j>=(int)i; j--) tp[j+1]=tp[j];
   tp[i]=per;
   (*np)++;
   return 0;
}
```

Hem definit la variable j com a int (doncs pot arribar a ser negativa amb valor `-1` quan la taula és buida) i la variable i com a unsigned int. Què passa quan fem la comparació j >= i ? El llenguatge C fa una interpretació del valor emmagatzemat a la variable j (int) com a unsigned int, amb la qual cosa, quan el valor que conté la j és –1, passa a ser interpretat com a 2^{32}–1 i, evidentment, el considera major que el valor de i si aquest és inferior a 2^{32}–1. Com es pot solucionar això? Doncs obligant que la variable i que és unsigned int sigui interpretada com a int, cosa que no ens portarà problemes perquè els valors que emmagatzema la variable i són relativament petits. Aquesta obligació s'aconsegueix amb la notació cast per la conversió de tipus:

```
j >= (int)i
```

en què hem fet que la variable i sigui considerada int per a resoldre la comparació.
En la comparació dels camps nif s'ha hagut d'utilitzar la funció de cadenes strcmp perquè és l'única manera de què disposem per a comparar cadenes en llenguatge C.

3.2. Recerca seqüencial

Ja sabem efectuar la recerca seqüencial en taules per a les quals no coneixem cap tipus d'ordre. Ara, però, veurem que si hem de fer una recerca seqüencial en una taula ordenada i la recerca del valor s'efectua per l'apartat que manté ordenada la taula, podem aprofitar aquest fet per a desestimar l'existència del valor cercat en el moment en què la posició de la taula conté un valor superior (en cas d'ordenació ascendent) o inferior (en cas d'ordenació descendent) al valor cercat.

Davant d'un algorisme d'aquest estil també hem de saber com s'ha d'actuar si es dóna la possibilitat que hi hagi elements repetits. La resposta és senzilla. Si hem trobat un element amb el valor cercat, en cas d'haver-n'hi de repetits, estaran a continuació, ja que la taula està ordenada per l'apartat que conté el valor cercat.

3.2.1. Algorisme en pseudocodi

Suposarem un tipus de dada TD que conté un apartat tc de tipus TC. El tipus TC té definit un ordre (això no passa en tots els tipus de dades). Vegeu l'algorisme per a cercar element(s) amb valor v per l'apartat tc en una taula d'elements de tipus TD ordenada per l'apartat tc.

```
funció cerca_ordenada (t: taula[MAX] de TD, nt: natural, v: tc) retorna enter és
/*
   Arguments:
      t  : Taula ordenada ASC pel camp tc on s'ha de cercar elements amb un valor v en
           l'apartat tc
      v  : Valor a cercar
      nt : Quantitat real d'elements existents a la taula
   Retorna:
      Pos: Primera posició de la taula en què es troba un element amb el valor cercat
      -1 : No s'ha trobat cap element amb el valor cercat
*/
   var  i: natural;
   fivar
   i = 0;
   mentre i < nt i t[i].tc < v fer i = i + 1; fimentre
    /* En sortir del mentre, podem trobar-nos en una de les tres situacions següents:
       - Hem sobrepassat el nombre d'elements reals que conté la taula, és a dir, i>=nt
         i, per tant, no hi ha cap element amb el valor cercat.
       - Estem dins el conjunt d'elements reals que conté la taula, però hem sobrepassat
```

```
        el valor cercat, és a dir, i<nt i t[i].tc>v  i, per  tant, no hi ha cap element
        amb el valor cercat
      - Estem dins el conjunt d'elements reals que conté la taula i ens trobem en el
        primer element que en l'apartat tc conté el valor cercat, és a dir, i<nt i
        t[i].tc==v; la recerca ha tingut èxit
   */
   si i < nt i t[i].tc == v llavors retorna i;
   sinó retorna -1;
   fisi
fifunció
```

3.2.2. Exemple en llenguatge C

Volem fer un programa que permeti cercar una persona donat un NIF en una taula de persones (segons la definició de tipus `persona` existent a l'arxiu `persona1.h`) ordenada ascendentment pel camp NIF.

```
/* u4n3p02.c
 */

#include "persona1.h"
#include "generic1.h"
#include "conio.h"
#include "verscomp.h"
#include <stdio.h>
#include <string.h>

#define MAXPER 20

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per);
int cerca_ordenada (struct t_persona *tp, unsigned int np, char *nif);

void main(void)
{
   struct t_persona p[MAXPER];  /* Taula de MAX persones */
   unsigned int np=0;         /* Número de persones introduïdes a p */
   struct t_persona e;          /* Element a afegir a la taula */
   char opc='S';
   int i;
   char nif[10];
   while (opc=='S')
   {
      inputPersona (&e);
      insercio_ordenada (p, &np, e);
      if (np<MAXPER) pregunta ("Vol afegir-ne més?", &opc, "SN");
      else opc='N';
   }
   pregunta ("Vol cercar alguna persona pel NIF?", &opc, "SN");
   while (opc=='S')
   {
      pantallaPersona();
      do
      { gotoxy(40,8); i = scanf ("%9s",nif); neteja_stdin(); }
      while (i == 0);
      strupr(nif);
```

Trobareu els fitxers `u4n3p02.c` i `persona1.h` i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
      if ((i=cerca_ordenada(p,np,nif)) >= 0)
      {
         do
         {
            outputPersonaFor(p[i]); i++;
            if (i==np)
            {
               missatge_ae ("Ja no hi ha més persones a la taula.   ");
               opc='N';
            }
            else
               pregunta ("Vol veure més persones amb el mateix NIF?", &opc, "SN");
         }  while (opc=='S' && strcmp(p[i].nif,nif)==0);
         if (opc=='S') missatge_ae ("No hi ha més persones amb el NIF sol.licitat.   ");
      }
      else missatge_ae ("No hi ha cap persona amb el NIF sol.licitat.   ");
      pregunta ("Vol cercar més persones pel NIF?", &opc, "SN");
   }
   clrscr();
}

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per)
{
   unsigned int i;
   int j;
   if (*np == MAXPER) return 1;
   for (i=0; i<*np && strcmp(tp[i].nif,per.nif)<=0; i++);
   for (j=(*np)-1; j>=(int)i; j--) tp[j+1]=tp[j];
   tp[i]=per;
   (*np)++;
   return 0;
}

int cerca_ordenada (struct t_persona *tp, unsigned int np, char *nif)
{
   unsigned int i;
   for (i=0; i<np && strcmp(tp[i].nif, nif) < 0; i++);
   if (i<np && strcmp(tp[i].nif, nif) == 0) return i;
   else return -1;
}
```

3.3. Recerca dicotòmica

Aquest és un dels millors algorismes de recerca aplicables en taules ordenades. Es basa en el fet que quan es cerca un valor en una taula ordenada, si considerem que és dividida en dos trossos, el valor cercat només es pot trobar en un dels dos trossos (excepte en el cas que n'hi hagi de repetits –els quals estan agrupats– i uns hagin quedat en un tros i els altres en l'altre tros). Si dels trossos coneixem els valors primer i darrer, podem decidir ràpidament si el valor cercat té possibilitats o no d'estar dins el tros en qüestió.

L'algorisme consisteix a dividir la taula per la meitat i quedar-nos amb el primer tros si el valor cercat està inclòs en l'interval de valors format pel primer i darrer elements del tros. Si no és així, es comprova el mateix fet en el segon tros, en el qual prosseguirem la recerca si l'inclou. En el cas contrari ja es pot decidir que el valor cercat no és a la taula.

Intuïu d'on prové el nom de recerca dicotòmica? D'anar dividint la taula en dos i decidir per quin dels dos trossos es continua.

3.3.1. Algorisme en pseudocodi

Suposarem un tipus de dada TD que conté un apartat tc de tipus TC. El tipus TC té definit un ordre (això no passa en tots els tipus de dades). Vegeu l'algorisme per a cercar element(s) amb valor v per l'apartat tc en una taula d'elements de tipus TD ordenada per l'apartat tc.

```
funció cerca_dicotòmica (t: taula[MAX]  de TD, nt: natural, v: tc) retorna enter és
/*
   Arguments:
      t  : Taula ordenada ASC pel camp tc on s'ha de cercar elements amb un valor v en
           l'apartat tc
      v  : Valor que es vol cercar
      nt : Quantitat real d'elements existents a la taula; posicions 0 a nt-1
   Retorna:
      Pos: Primera posició de la taula en què es troba un element amb el valor cercat
      -1 : No s'ha trobat cap element amb el valor cercat
*/
   var  inf: natural; /* Conté la posició menor del tros de taula que s'està tractant */
        sup: natural; /* Conté la posició major del tros de taula que s'està tractant */
        mig: natural; /* Conté la posició per on trenquem la taula en dos trossos */
        posible: lògic;  /* Per saber si hem de continuar -pot estar o no en algun tros- */
   fivar
   inf=0;
   sup=nt-1;
   possible = v ≥ t[inf].tc i v ≤ t[sup].tc;
   /* Comprovem, d'entrada, si el valor pot estar dins la taula */
   mentre possible i inf < sup fer
      mig = (inf + sup) div 2;
      si v ≤ t[pos].tc
      llavors
      /* Cas en què el valor pot estar en el primer tros. Continuem amb el primer tros */
         sup = mig;
      sinó si v ≥ t[mig+1].tc
           llavors
           /* Cas en què el valor pot estar en el segon tros. Seguim amb el segon tros */
              inf = mig+1;
           sinó
           /* El valor no pot estar a la taula perquè no pot estar en cap tros */
              possible = fals;
           fisi
      fisi
   fimentre
```

```
   /* Quan sortim del mentre hem de decidir el motiu pel qual s'ha sortit. Comproveu que
      en cas d'existir, se surt del mentre quan s'arriba a una taula d'un únic element i,
      per tant, per comprovar si s'ha trobat cal preguntar sobre el valor que conté tal
      element; pot donar-se el cas que s'arribi a una taula d'un sol element i aquest no
      contingui el valor cercat; per tant, és obligarori preguntar sobre el valor de la
      posició inf en el moment en què se surt del mentre */
   si t[inf].tc != v
      llavors
         /* No hi ha cap element amb el valor cercat */
         retorna -1;
      sinó
         /* La variable inf té la posició del primer element que conté el valor cercat */
         retorna inf;
   fisi
fifunció
```

Hi ha nombroses variants de l'algorisme de recerca dicotòmica. El que s'ha presentat aquí no és el més eficient, però sí que creiem que és el de més senzilla programació. La diferència d'eficiència respecte d'altres dissenys de recerca per dicotomia és mínima.

3.3.2. Exemple en llenguatge C

Volem fer un programa que permeti cercar una persona donat un NIF en una taula de persones (segons la definició de tipus `persona` existent a l'arxiu `persona1.h`) ordenada ascendentment pel camp NIF utilitzant la recerca dicotòmica.

```
/* u4n3p03.c */

#include "persona1.h"
#include "generic1.h"
#include "conio.h"
#include "verscomp.h"
#include <stdio.h>
#include <string.h>

#define MAXPER 20

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per);
int cerca_dicotomica (struct t_persona *tp, unsigned int np, char *nif);

void main(void)
{
   struct t_persona p[MAXPER];  /* Taula de MAX persones */
   unsigned int np=0;         /* Número de persones introduïdes a p */
   struct t_persona e;          /* Element a afegir a la taula */
   char opc='S';
   int i;
   char nif[10];
   while (opc=='S')
   {
      inputPersona (&e);
```


Trobareu els fitxers `u4n3p03.c` i `persona1.h` i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
      insercio_ordenada (p, &np, e);
      if (np<MAXPER) pregunta ("Vol afegir-ne més?", &opc, "SN");
      else opc='N';
   }
   pregunta ("Vol cercar alguna persona pel NIF?", &opc, "SN");
   while (opc=='S')
   {
      pantallaPersona();
      do
      { gotoxy(40,8); i = scanf ("%9s",nif); neteja_stdin(); }
      while (i == 0);
      strupr(nif);
      if ((i=cerca_dicotomica(p,np,nif)) >= 0)
      {
         do
         {
            outputPersonaFor(p[i]); i++;
            if (i==np)
            {
               missatge_ae ("Ja no hi ha més persones a la taula.   ");
               opc='N';
            }
            else
               pregunta ("Vol veure més persones amb el mateix NIF?", &opc, "SN");
         }  while (opc=='S' && strcmp(p[i].nif,nif)==0);
         if (opc=='S') missatge_ae ("No hi ha més persones amb el NIF sol.licitat.   ");
      }
      else missatge_ae ("No hi ha cap persona amb el NIF sol.licitat.   ");
      pregunta ("Vol cercar més persones pel NIF?", &opc, "SN");
   }
   clrscr();
}

int insercio_ordenada (struct t_persona *tp, unsigned int *np, struct t_persona per)
{
   unsigned int i;
   int j;
   if (*np == MAXPER) return 1;
   for (i=0; i<*np && strcmp(tp[i].nif,per.nif)<=0; i++);
   for (j=(*np)-1; j>=(int)i; j--) tp[j+1]=tp[j];
   tp[i]=per;
   (*np)++;
   return 0;
}

int cerca_dicotomica (struct t_persona *tp, unsigned int np, char *nif)
{
   unsigned int inf=0;
   unsigned int sup=np-1;
   unsigned int mig,possible;
   possible = strcmp (nif,tp[inf].nif) >= 0 && strcmp (nif,tp[sup].nif) <= 0 ;
   while (possible && inf < sup)
   {
      mig = (inf + sup) / 2;
      if ( strcmp (nif, tp[mig].nif) <= 0 ) sup = mig;
      else if (strcmp (nif, tp[mig+1].nif) >= 0 ) inf = mig + 1;
           else possible = 0;
   }
```

```
   if ( strcmp (nif, tp[inf].nif) == 0) return inf;
   else return -1;
}
```

3.4. Ordre d'un algorisme

Des de principi del crèdit hem perseguit l'objectiu de dissenyar algorismes eficients per a resoldre els problemes que se'ns plantegen. Hi ha un concepte que ens ajudarà a definir l'eficiència dels algorismes que han de gestionar grans volums de dades en funció del volum d'aquestes dades. Es tracta del concepte d'ordre de l'algorisme.

Per tal de tractar correctament aquest concepte cal tenir una base matemàtica que no podem suposar que tingueu. A més, aquest concepte fins i tot l'hauríem pogut obviar. No ens sembla, però, la solució correcta, ja que en molts llibres d'informàtica us trobareu amb frases del tipus: "és un algorisme lineal", "és un algorisme quadràtic", "és un algorisme d'ordre logarítmic", i de semblants. Intentarem, per tant, donar unes nocions intuïtives d'aquest concepte, de manera que el sapigueu interpretar i, fins i tot, aproximar.

> Si considerem un volum d'informació consistent en n elements, direm que un algorisme és d'ordre k si per gestionar tota la informació cal un conjunt de $An^k + Bn^{k-1} + ...+ Z$ operacions on A, B, ..., Z són constants reals amb A diferent de zero.

És a dir, suposem que per fer un tipus de gestió en una taula de n elements, sabem calcular que el nombre d'operacions necessàries (multiplicacions, sumes, restes, etc.) és $4n^3 - 2n^2 + 5$. En tal cas, diríem que l'algorisme dissenyat és d'ordre 3.

No us avanceu! Hem suposat que sabem calcular la fórmula que ens diu el nombre d'operacions (polinòmica en aquest cas, però que pot tenir parts exponencials, logarítmiques, etc.) No és l'objectiu entrar en l'obtenció d'aquesta fórmula. Per a això ja hi ha els matemàtics i els informàtics teòrics. Simplement cal entendre el concepte i saber "aproximar" la fórmula en certs casos. ‼

Trobarem algorismes de diferents ordres, que reben noms especials:

- Algorisme d'ordre 0 o ordre constant. Sigui quin sigui el valor de n el procés utilitza el mateix nombre d'operacions. N'hi ha molt pocs, d'algorismes constants.

- Algorisme d'ordre 1 o ordre lineal. El nombre d'operacions segueix una fórmula polinòmica de grau 1. S'acostuma a notar dient que és $\theta(n)$.

- Algorisme d'ordre 2 o ordre quadràtic. El nombre d'operacions segueix una fórmula polinòmica de grau 2. S'acostuma a notar dient que és $\theta(n^2)$.

- Algorisme d'ordre 3 o ordre cúbic. El nombre d'operacions segueix una fórmula polinòmica de grau 3. S'acostuma a notar dient que és $\theta(n^3)$.

- Algorisme d'ordre logarítmic. El nombre d'operacions segueix una fórmula del tipus A log n + B, on A i B són constants reals amb A diferent de zero. S'acostuma a notar dient que és $\theta(\log n)$.

Suposem que intuïu el concepte. Donada la fórmula que calcula el nombre d'operacions, l'ordre de l'algorisme és el terme predominant dins la fórmula. És a dir, si la fórmula és polinòmica, tots sabeu que el terme predominant és el de grau més alt. Ara bé, en la fórmula:

$3n^2 - 5\log n + n\log n - n + e^n$

quin és el terme predominant? Evidentment, e^n. Per tant diríem que l'algorisme és d'ordre exponencial i escriuríem d'ordre $\theta(e^n)$.

Per què ens quedem només amb el terme predominant? Els programes informàtics es dissenyen per gestionar grans volums d'informació. És clar que per a valors grans de n només ens interessa el terme predominant, ja que és el que ens dóna idea de la quantitat d'operacions necessàries. La taula 2 ens mostra els càlculs per alguns valors predominants segons el volum n d'informació.

Taula 2. Exemples de càlculs dels termes predominants segons el volum d'informació.

Termes predominants	n = 10	n = 100	n = 1000	n = 10000
n^3	10^3	10^6	10^9	10^{12}
$\log_{10} n$	1	2	3	4
e^n	22026.47	2.69E + 43	Immens No calculable	Immens No calculable

És important distingir el menor ordre entre un conjunt d'ordres possibles, de manera que si tenim algorismes diferents amb ordres diferents per a resoldre un mateix problema, ens interessarà poder tenir ordenats aquests algorismes de menor a major ordre, per tal d'escollir el de menor ordre, ja que serà el més eficient.

En aquest sentit cal tenir en compte que un algorisme logarítmic sempre

és més eficient que un algorisme polinòmic i aquest és més eficient que un algorisme exponencial. Això, combinat amb el coneixement que s'extreu de les expressions següents:

$$\theta(1) < \theta(n) < \theta(n^2) < \theta(n^3) < \theta(n^4) < \ldots$$
$$\theta(\log n) < \theta(\log^2 n) < \theta(\log^3 n) < \theta(\log^4 n) < \ldots$$
$$\theta(e^n) < \theta(e^{2n}) < \theta(e^{3n}) < \theta(e^{4n}) < \theta(e^{5n}) < \ldots$$

ens permet decidir quin algorisme és millor coneixent les fórmules corresponents.

I, com podem intuir les fórmules? L'experiència és fonamental. Vegeu, però, en què ens basem per assegurar que tots els algorismes de recerca i recorreguts de taules que coneixem, a excepció de la recerca dicotòmica, són lineals.

En tots els algorismes esmentats, si la taula té n elements per a recórrer, els algorismes que efectuàvem consistien a començar per la primera posició i anar passant posició a posició. En el cas de la recerca, podia ser que no s'arribés al final de la taula, però, en el pitjor dels casos, calia arribar al final. Per tant, passàvem per les n posicions i per cada posició fèiem un determinat nombre constant d'operacions. O sigui, tenim una fórmula del tipus An+B amb la A diferent de zero. Tots els algorismes anomenats són lineals.

Comentem el cas de la cerca dicotòmica, que és l'únic que ens queda pendent. Aquest serà l'únic exemple en què plantejarem un raonament matemàtic (molt light, però), el qual està a l'abast dels vostres coneixements.

Donada una taula amb n elements, fixeu-vos que, a tot estirar, anem partint la taula en dos trossos, per la meitat, fins a arribar a un tros d'un únic element, situació en la qual prenem la decisió final. Per tant:

- Pas 1. Trenquem la taula en dos trossos de n/2 elements cadascun.
- Pas 2. Trenquem el tros en dos trossos de n/4 elements cadascun.
- Pas 3. Trenquem el tros en dos trossos de n/8 elements cadascun.
- ...
- Pas X. Trenquem el tros en dos trossos de $n/2^x$ elements cadascun.

Suposem que aquest és el darrer pas. Per tant, ha de ser $n/2^x = 1$, perquè ja hem dit que en el darrer pas arribem a tenir un tros amb un únic element. Ens interessa saber quants passos hem hagut de fer, ja que això ens donarà el nombre d'operacions necessàries. Per tant hem d'aïllar x de l'expressió $n/2^x = 1$. Aquesta expressió és equivalent a n =

2^x, la qual, a la vegada, és equivalent a $x = \log_2 n$. Ja tenim la x. La cerca dicotòmica és d'ordre logarítmic.

3.5. Mètodes d'ordenació

És clar que l'ordre dels valors en una taula podria tenir influència sobre l'elecció d'un algorisme de tractament. Així, per exemple, la cerca per dicotomia només pot ser efectuada sobre una taula ordenada segons el criteri de recerca.

Freqüentment és necessària la classificació dels valors d'una taula segons l'ordre (creixent o decreixent) d'un dels apartats dels seus elements. Hi ha nombrosos algorismes per a efectuar aquest tipus d'operació, anomenada ordenació interna per oposició a l'ordenació externa que ordena els fitxers emmagatzemats en suports externs.

Cal conèixer, ni que sigui d'una manera lleugera, els diferents mètodes d'ordenació interna:

- ordenació per inserció,
- ordenació per selecció,
- ordenació per intercanvi,
- ordenació per comptabilització de freqüències,
- ordenació ràpida,

indicant-hi, sense entrar en detalls d'avaluació, l'**ordre** i l'**estabilitat**.

3.5.1. Estabilitat

L'estabilitat és un element clau en els algorismes d'ordenació.

> Un algorisme d'ordenació és estable quan es pot assegurar que en acabar el procés d'ordenació, els elements amb valors repetits pel criteri d'ordenació mantenen la posició relativa que tenien abans d'iniciar el procés.

Per exemple, suposem la taula t de caràcters de la figura 5. Observem que el caràcter A hi és dues vegades. Hem marcat amb l'ajut dels asteriscos quina és la posició relativa entre els dos. Un algorisme d'ordenació és estable si ens assegura que, després de la seva execució, la taula quedarà

com mostra la figura 6, on els dos caràcters A mantenen la posició relativa que tenien abans de començar el procés d'ordenació.

Figura 5 . Taula de caràcters que pretenem ordenar.

	0	1	2	3	4	5	6	7	8
t	Z	B	A*	D	A**	M	F	K	E

Figura 6. Taula de caràcters de la figura 5 una vegada ordenada amb un mètode estable.

	0	1	2	3	4	5	6	7	8
t	A*	A**	B	D	E	F	K	M	Z

Un algorisme es considera estable quan garanteix aquesta funcionalitat en qualsevol execució. Un algorisme no estable pot produir un resultat semblant en algunes execucions. Per tant, no hem de caure en l'error de comprovar l'estabilitat d'un algorisme basant-nos en una situació concreta. L'estabilitat es raona sobre el disseny de l'algorisme.

Us preguntareu quin és l'interès que té l'estabilitat. Moltes vegades pot interessar executar un procés d'ordenació estable per assegurar que els elements que tenen valors idèntics segons el criteri d'ordenació mantinguin l'ordre relatiu. Vegem-ne alguna situació concreta.

Suposem que es té una taula de socis d'una societat esportiva ordenada per la data de naixement dels seus socis. Suposem que l'emplenat de la taula es produeix per mitjà d'un procés d'inserció ordenada de manera que, quan es dóna d'alta com a soci una persona amb data de naixement coincident amb la d'algun soci existent, la inserció s'efectua immediatament després de tots els socis amb mateixa data de naixement (tal com hem aplicat en l'algorisme de la inserció ordenada quan hi havia valors repetits). Per tant, els socis estan ordenats per data de naixement i, en cas de repetició, per antiguitat com a soci.

Suposem que ara volem aplicar un procés d'ordenació segons el sexe. Evidentment n'hi haurà de repetits, si no és que la societat esportiva només tingui dos socis, un de cada sexe. Suposem que dins del grup corresponent a un mateix sexe es vol mantenir els socis ordenats per data de naixement. Com que partim d'un conjunt de socis ja ordenat per data de naixement, si l'algorisme d'ordenació és estable, podem assegurar que els socis del mateix sexe continuen ordenats per data de naixement i, encara més, els que tenien la mateixa data de naixement, continuen ordenats per antiguitat com a soci.

En la situació anterior partíem d'un conjunt ordenat per la seva pròpia gestió d'insercions. A vegades, però, partim d'un conjunt d'elements que no mantenen cap tipus d'ordre. Imaginem-nos un conjunt de persones que volem ordenar de la mateixa manera que apareixen en una guia telefònica: primer cognom – segon cognom – nom.

Per aconseguir això es pot aplicar un procés d'ordenació complex que tingui en compte els tres apartats de manera conjunta. Però a vegades això no és factible i s'actua com segueix. Primer s'ordena pel nom de les persones. Posteriorment, s'aplica un mètode d'ordenació estable sobre el segon cognom, el qual garanteix que tindrem les persones ordenades pel segon cognom i, en cas de coincidència, pel nom. Per acabar, s'aplica un altre cop el mètode d'ordenació estable sobre el primer cognom, de manera que s'obté el conjunt de persones ordenat pel primer cognom i, en cas de coincidència, pel segon cognom i, si també hi ha coincidència, pel nom.

Abans de presentar els algorismes més coneguts d'ordenació, definirem l'entorn per a tots ells: ordenarem ascendentment una taula `t` de `MAX` elements del tipus `TD` que conté un apartat `tc` de tipus `TC`, el qual té definit un ordre i que, per tant, podem fer-hi servir els operadors relacionals.

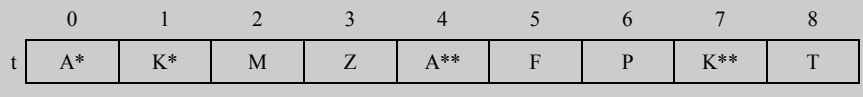
3.5.2. Ordenació per inserció

L'algorisme d'ordenació per inserció es basa en el mètode següent.

> Suposant que tenim el tros `t[0...i-1]` ordenat, inserirem el valor `t[i]` de manera ordenada i obtenim el tros `t[0...i]` ordenat.

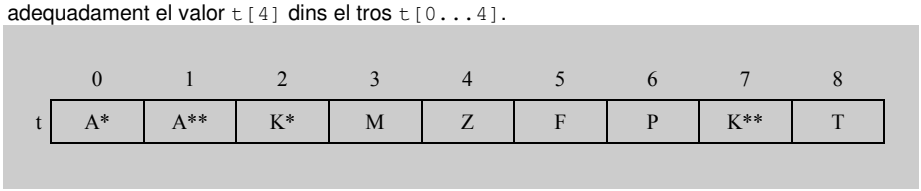
És a dir, donada la taula de la figura 7, en la qual podem observar que tenim el tros `t[0...3]` ordenat, es tracta d'inserir adequadament el valor `t[4]` en el tros `t[0...4]` de manera que aquest tros resti ordenat.

Figura 7. Procés d'ordenació per inserció. El tros `t[0...3]` està ordenat. Procedim a inserir-hi ordenadament el valor `t[4]` de manera que el tros `t[0...4]` resti ordenat.



A més interessa, si pot ser, que l'algorisme sigui estable. El resultat ha de ser el que mostra la figura 8.

Figura 8. Procés d'ordenació per inserció. Situació de la taula de la figura 7 una vegada inserit adequadament el valor `t[4]` dins el tros `t[0...4]`.



Aquest procés s'ha de repetir des d'un inici, en què cal partir d'un tros ja ordenat per anar obtenint, a cada repetició, un tros ordenat amb una casella més, fins a tenir tota la taula ordenada.

En general, quin és el major tros inicial pel qual es pot assegurar l'estat d'ordenat sense necessitat de fer cap comprovació? Evidentment la resposta és el tros format per la primera casella `t[0]`.

Per tant, l'algorisme és:

```
var  i, j, k:  natural;
     aux: TD;
fivar
per i = 1 fins MAX-1 fer
   j = 0;
   mentre j < i i t[j].tc ≤ t[i].tc fer j = j+1; fimentre (*)
   aux = t[i];
   k = i-1;
   mentre k ≥ j fer
      t[k+1] = t[k];
      k = k-1;
   fimentre
   t[j] = aux;
fiper
```

L'estructura iterativa `per` garanteix el procés de situar l'element de la casella `t[i]` dins el tros `t[0...i]` i això ho fa a partir de la casella de la posició `1` i fins a la darrera posició `MAX-1`.

Endinsant-nos en el procés que es fa dins el `per`, observeu que per saber on s'ha de situar l'element de la casella `t[i]`, s'inicia un procés de comparació (primer `mentre` dins el `per`) des de la posició `0` en direcció cap la posició `i-1`, el qual s'atura quan es troba la posició `j` en què pertoca inserir l'element `t[i]`. Però per posar-l'hi, cal fer espai. Això és el que es fa en el segon `mentre` dins el `per`.

L'algorisme és quadràtic, ja que fa un recorregut per tota la taula (o quasi tota, ja que deixa de tractar la primera posició), la qual cosa implica ordre lineal, però per cada posició que es tracta torna a fer un recorregut

seqüencial de bona part de la taula. És a dir, tenim un tractament seqüencial dins un altre tractament seqüencial. Per tant, tenim un algorisme quadràtic.

Aquest algorisme d'ordenació és estable. Aquest fet està garantit per la comparació ≤ a la línia (*). Si fos <, també ordenaria però el mètode no seria estable.

Observeu també que per poder moure elements de posició, ens cal una variable `aux` del tipus `TD` d'elements de la taula. Totes les comparacions es fan segons el criteri d'ordenació `tc` però l'intercanvi és d'un element `TD` sencer, no únicament del criteri `tc`.

3.5.3. Ordenació per selecció

L'algorisme d'ordenació per selecció es basa en el mètode següent:

> Suposant que tenim el tros `t[0...i-1]` ordenat i contenint tots els valors no superiors als del tros `t[i...MAX-1]`, cercarem el menor valor d'entre `t[i...MAX-1]` i l'intercanviarem amb el valor `t[i]`, i obtindrem el tros `t[0...i]` ordenat que contindrà tots els valors no superiors als del tros `t[i+1...MAX-1]`.

És a dir, donada la taula següent, en la qual podem observar que tenim el tros `t[0...3]` ordenat i que conté tots els valors no superiors als del tros `t[4...8]`, es tracta de seleccionar adequadament el menor valor d'entre `t[4...8]` per intercanviar-lo amb el valor `t[4]`.

Figura 9. Procés d'ordenació per selecció. El tros `t[0...3]` està ordenat amb tots els valors no superiors als del tros `t[4...8]`. Procedim a seleccionar el menor valor de t[4...8] per a intercanviar-lo amb el valor `t[4]` de manera que el tros `t[0...4]` resti ordenat amb tots els valors no superiors als del tros `t[5...8]`

	0	1	2	3	4	5	6	7	8
t	A	C	E	F	R*	R**	S	M	T

A més interessa, si pot ser, que l'algorisme sigui estable. El resultat ha de ser el que mostra la figura 10.

Figura 10. Procés d'ordenació per selecció. Situació de la taula de la figura 9 una vegada inserit adequadament el menor valor del tros `t[4...8]` a la posició t[4] via intercanvi.

	0	1	2	3	4	5	6	7	8
t	A	C	E	F	M	R**	S	R*	T

Aquest procés s'ha de repetir des d'un inici en què cal partir d'un tros ordenat amb tots els valors no superiors als del tros restant per anar obtenint, a cada repetició, un tros també ordenat amb una casella més i amb tots els valors no superiors als del tros restant, fins a tenir tota la taula ordenada.

En general, quin és el major tros inicial pel qual es pot assegurar aquest estat sense necessitat de fer cap comprovació? Cap. Hem de partir d'un tros buit.

Per tant, l'algorisme és el següent:

```
var  i, j, pos: natural;
     aux: TD;
fivar
per i = 0 fins MAX-2 fer
   pos = i;
   per j=i+1 fins MAX-1 fer
      si t[j].tc < t[pos].tc llavors pos = j; fisi
   fiper
   aux = t[pos];
   t[pos] = t[i];
   t[i] = aux;
fiper
```

L'estructura iterativa externa `per` garanteix el procés repetitiu iniciant la recerca del menor element de tota la taula per situar-lo a la posició `t[0]` fins a la recerca del menor element del tros `t[MAX-2...MAX-1]`.

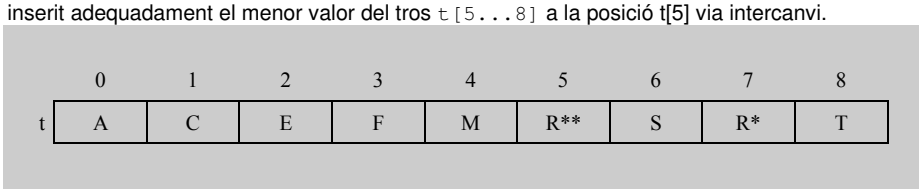
Endinsant-nos en el procés que es fa dins el `per` extern, observeu que per cercar el menor element del tros `t[i..MAX-1]` es fa un recorregut total guardant la posició `pos` on es troba el menor element per, quan s'acabi el recorregut, intercanviar-lo amb l'element de la posició `i`.

Observeu que l'algorisme és quadràtic, ja que fa un recorregut per tota la taula (o quasi tota, perquè deixa de tractar la darrera posició), la qual cosa implica ordre lineal, però per cada posició que s'ha de tractar torna a fer un recorregut seqüencial de bona part de la taula. És a dir, tenim un tractament seqüencial dins un altre tractament seqüencial. Per tant, tenim un algorisme quadràtic. Està comprovat, matemàticament, que és més ràpid que l'algorisme d'inserció, tot i que tots dos són quadràtics.

Aquest algorisme d'ordenació és no estable. Aquest fet el podeu comprovar en l'exemple utilitzat en la introducció de l'algorisme. Observeu que en trobar com a menor element del tros `t[4...8]` el caràcter `M` de la posició `7` i intercanviar-lo amb el caràcter `R` de la posició `4`, aquest passa a ocupar la posició `7`. En conseqüència, els caràcters `R*` i `R**` han intercanviat la seva posició relativa.

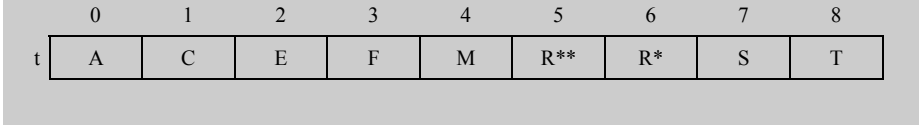
En la següent iteració, el menor element que s'agafarà serà el caràcter `R**` de la posició `5` que quedarà a la mateixa posició `5` i s'obté la taula de la figura 11.

Figura 11. Procés d'ordenació per selecció. Situació de la taula de la figura 10 una vegada inserit adequadament el menor valor del tros `t[5...8]` a la posició t[5] via intercanvi.



Si continuem el procés fins al final, obtindrem la taula de la figura 12, on s'observa que els elements de les posicions 5 i 6 no mantenen la posició relativa inicial. Si inicialment haguessin estat situats de manera diferent, podria donar-se el cas que en acabar mantinguessin la posició relativa. Però l'estabilitat d'un algorisme es basa a assegurar la posició relativa en qualsevol circumstància.

Figura 12. Procés d'ordenació per selecció. Situació de la taula de la figura 11 una vegada finalitzat el procés d'ordenació.



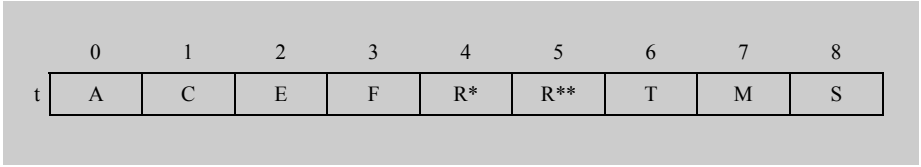
3.5.4. Ordenació per intercanvi

L'algorisme d'ordenació per intercanvi es basa en el mètode següent:

> Suposant que tenim el tros `t[0...i-1]` ordenat i amb cap valor superior als que hi ha al tros `t[i...MAX-1]`, pujarem (com una bombolla dins l'aigua) el menor valor d'entre `t[i...MAX-1]` a `t[i]`, examinant dos a dos els elements de `t[i..MAX-1]` a partir de `(MAX-1, MAX-2)` en direcció cap a `(i+1, i)` intercanviant si és necessari, obtenint el tros `t[0...i]` ordenat i amb cap valor superior als del tros `t[i+1...MAX-1]`.

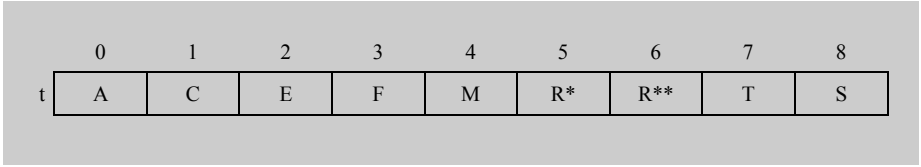
És a dir, donada la taula de la figura 13, en la qual podem observar que tenim el tros `t[0...3]` ordenat i amb cap valor superior als del tros `t[4...8]`, es tracta de passar el menor valor del tros `t[4...8]` cap a `t[4]`, mitjançant intercanvi, comparant els elements dos a dos des del final i fins a la posició `4`.

Figura 13. Procés d'ordenació per intercanvi. El tros `t[0...3]` està ordenat amb tots els valors no superiors als del tros `t[4...8]`. Procedim a pujar (via intercanvis dos a dos) el menor valor de t[4...8] fins a `t[4]` de manera que el tros `t[0...4]` resti ordenat amb tots els valors no superiors als del tros `t[5...8]`



A més interessa, si pot ser, que l'algorisme sigui estable. El resultat ha de ser el que mostra la figura 14.

Figura 14. Procés d'ordenació per intercanvi. Situació de la taula de la figura 13 una vegada pujat adequadament el menor valor del tros `t[4...8]` a la posició t[4] via intercanvi.



Aquest procés s'ha de repetir des d'un inici en què cal partir d'un tros ordenat amb cap valor superior als del tros restant per anar obtenint, a cada repetició, un tros també ordenat amb una casella més i amb cap valor superior als del tros restant, fins a tenir tota la taula ordenada.

En general, quin és el major tros inicial pel qual es pot assegurar aquest estat sense necessitat de fer cap comprovació? Cap. Hem de partir d'un tros buit.

El que s'acaba d'introduir és l'algorisme de la bombolla tradicional, el qual té una millora immediata i consisteix a controlar el fet que no hi hagi hagut cap intercanvi en el procés de pujar des de `(MAX-1,MAX-2)` fins a `(i+1,i)`. En tal cas, la taula ja està ordenada i no cal continuar el procés.

Per tant, l'algorisme és:

```
var  i,j:    natural;
     canvi:  lògic;
     aux:    TD;
fivar
i = 0;
canvi = cert;
mentre i < MAX - 1 i canvi fer
   j = MAX-1;
   canvi = fals;
   mentre j>i fer
(*)  si t[j].tc < t[j-1].tc llavors
        aux = t[j];
        t[j] = t[j-1];
        t[j-1] = aux;
        canvi = cert;
```

```
      fisi
      j = j-1;
   fimentre
   i = i+1;
fimentre
```

L'estructura iterativa externa `mentre` garanteix el procés repetitiu iniciat en portar el menor element de tota la taula a la posició `t[0]` fins a portar el menor element del tros `t[MAX-2...MAX-1]` a la posició `t[MAX-2]`.

Endinsant-nos en el procés que es fa dins el `mentre` extern, observeu que per portar el menor element del tros `t[i..MAX-1]` a la posició `i` es fa un recorregut total per aquest tros intercanviant els elements quan és necessari.

L'algorisme és quadràtic, ja que fa un recorregut per tota la taula (o quasi tota, perquè deixa de tractar la darrera posició), la qual cosa implica ordre lineal, però per cada posició que s'ha de tractar torna a fer un recorregut seqüencial de bona part de la taula. És a dir, tenim un tractament seqüencial dins un altre tractament seqüencial. Per tant, tenim un algorisme quadràtic. Pel que fa a rapidesa, és pitjor que els algorismes de selecció i inserció. El nom de Bubble sort (algorisme de la bombolla) amb què ha estat batejat en els països anglosaxons constitueix el seu èxit.

Aquest algorisme d'ordenació és estable. Aquest fet està garantit per la comparació < a la línia (*). Si fos ≤, també ordenaria però el mètode no seria estable.

3.5.5. Exemple en llenguatge C

Volem fer un programa que permeti ordenar una taula de persones (segons la definició de tipus persona existent a l'arxiu persona1.h) pel primer cognom – segon cognom – nom. Això implica tres processos d'ordenació on el primer pot no ser estable, però el segon i tercer sí que ho han de ser. Com a utilització dels tres algorismes d'ordenació coneguts, es demana aplicar l'algorisme de selecció (no estable) per a ordenar la taula per nom, i posteriorment aplicar els algorismes de la bombolla i d'inserció per a ordenar la taula pel segon i primer cognoms.

```
/* u4n3p04.c */

#include "persona1.h"
#include "generic1.h"
#include "conio.h"
#include "verscomp.h"
```


Trobareu els fitxers `u4n3p04.c` i `persona1.h` i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

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

#define MAXPER 20

void omplir_taula(struct t_persona *p, unsigned int *np)
{
   struct t_persona p0={"00000000A","Esteve","Andujar","Mercè",{1,1,1962},69,168};
   struct t_persona p1={"11111111B","Bañeres","Pujol","Eduard",{2,3,1964},70,174};
   struct t_persona p2={"22222222C","Roca","Andujar","Lluisa",{3,9,1966},75,193};
   struct t_persona p3={"33333333D","Bravo","Borbon","Núria",{4,8,1968},55,190};
   struct t_persona p4={"44444444E","Castañe","Andujar","Aranzazu",{5,6,1970},75,165};
   struct t_persona p5={"55555555F","Benet","Borbon","Meritxell",{6,5,1972},83,185};
   struct t_persona p6={"66666666G","Millera","Pujol","Ruben",{7,3,1974},99,175};
   struct t_persona p7={"77777777H","Folguera","Borbon","Mònica",{8,12,1978},77,170};
   struct t_persona p8={"88888888I","Hospital","Roca","Ramon",{9,11,1976},48,180};
   struct t_persona p9={"99999999J","Perez","Roca","Pau",{10,10,1980},80,190};
   p[0]=p0;
   p[1]=p1;
   p[2]=p2;
   p[3]=p3;
   p[4]=p4;
   p[5]=p5;
   p[6]=p6;
   p[7]=p7;
   p[8]=p8;
   p[9]=p9;
   *np=10;
}

void visualitzar_taula (struct t_persona *p, unsigned int np)
{
   unsigned int i;
   clrscr();
   gotoxy (1,1);
   printf ("NIF       PRIMER COGNOM        SEGON COGNOM         NOM
NAIXEMENT");
   gotoxy (1,2);
   printf ("-------------------------------------------------------------------------------
-");
   for (i=0; i<np; i++) outputPersonaTab (i+3, p[i]);
}

void ord_seleccio (struct t_persona *p, unsigned int np)
{
   int i, j, pos;
   struct t_persona aux;
   for (i=0; i<=np-2; i++)
   {
      pos = i;
      for (j=i+1; j<np; j++)  if (strcmp(p[j].nom,p[pos].nom)<0) pos=j;
      aux = p[pos];
      p[pos] = p[i];
      p[i] = aux;
   }
}
```

```
void ord_bombolla (struct t_persona *p, unsigned int np)
{
   int i, j, canvi;
   struct t_persona aux;
   canvi = 1;
   for (i=0; i<np-1 && canvi; i++)
   {
      j = np-1; canvi = 0;
      for (j=np-1; j>i; j--)
         if (strcmp(p[j].cognom2,p[j-1].cognom2)<0)
         {
            aux = p[j];
            p[j] = p[j-1];
            p[j-1] = aux;
            canvi = 1;
         }
   }
}

void ord_insercio (struct t_persona *p, unsigned int np)
{
   int i, j, k;
   struct t_persona aux;
   for (i=1; i<=np-1; i++)
   {
      for (j=0; j<i && strcmp(p[j].cognom1,p[i].cognom1)<=0; j++);
      aux = p[i];
      for (k=i-1; k>=j; k--) p[k+1] = p[k];
      p[j] = aux;
   }
}

void main(void)
{
   struct t_persona p[MAXPER];  /* Taula de MAX persones */
   unsigned int np=0;        /* Número de persones introduïdes a p */
   omplir_taula(p,&np);
   visualitzar_taula(p,np);
   missatge_ae ("Contingut de la taula sense ordenar.   ");
   ord_seleccio(p,np);
   visualitzar_taula(p,np);
   missatge_ae ("Contingut de la taula ordenada per NOM via SELECCIO.  ");
   ord_bombolla(p,np);
   visualitzar_taula(p,np);
   missatge_ae ("Contingut de la taula ordenada pel 2n COGNOM via BOMBOLLA.  ");
   ord_insercio(p,np);
   visualitzar_taula(p,np);
   missatge_ae ("Contingut de la taula ordenada per 1r. COGNOM via INSERCIÓ.  ");
}
```

3.5.6. Ordenació per comptabilització de freqüències

L'algorisme que segueix és un algorisme lineal i estable (per tant, molt bo), però no és utilitzable en qualsevol situació, al contrari, s'ha de complir unes condicions molt restrictives:

- El valor pel qual s'ordena ha de ser enter.
- Els valors continguts a la taula no defineixin un interval gaire nombrós, suposem des de `A` fins a `Z`.

L'ordenació no es fa sobre la mateixa taula sinó sobre una altra d'idèntica a l'original.

```
var   FREQ:taula [Z-A+1] de enter
      tord:   taula [MAX] de TD;
      i:      natural;
fivar
per i = 0 fins Z - A fer FREQ[ i ] = 0; fiper (1)
per i = 0 fins MAX - 1 fer FREQ[ t[ i ] - A ] = FREQ[ t[ i ] - A ] + 1; fiper (2)
per i = 1 fins Z - A fer FREQ[ i ] = FREQ[ i ] + FREQ[ i - 1 ]; fiper (3)
i = MAX-1;
mentre i ≥ 0 fer
   FREQ[ t [ i ] - A ] = FREQ[ t [ i ] - A ] - 1;
   tord [ FREQ[ t [ i ] - A ] ] = t[ i ];
fimentre (4)
```

Vegem la funcionalitat fent el seguiment d'un exemple. Suposem una taula d'edats d'alumnes de cicles formatius. Podem suposar que les edats són compreses entre 17 (A) i 96 (Z) anys, interval no gaire nombrós. Per tant, declarem FREQ de 80 posicions (Z-A+1).

Per a majors de 96 anys

Si teniu més de 96 anys no us enfadeu amb nosaltres. Estareu d'acord que no és gaire normal ser alumne d'un cicle formatiu. Us felicitem pel vostre coratge.

La figura 15 ens visualitza un seguiment de l'ordenació per comptabilització de freqüències sobre una taula d'edats.

Figura 15. Evolució del procés d'ordenació per comptabilització de freqüències.

	0	1	2	3	4	5	6	7	8	9
t	17*	18*	20*	17**	23	18**	20**	19*	19**	17***

Estat de FREQ després de (1):

	0	1	2	3	4	5	6	7	8	
FREQ	0	0	0	0	0	0	0	0	0	...

Estat de FREQ després de (2):

	0	1	2	3	4	5	6	7	8	
FREQ	3	2	2	2	0	0	1	0	0	...

Estat de FREQ després de (3):

	0	1	2	3	4	5	6	7	8	
FREQ	3	5	7	9	9	9	10	10	10	...

Taula ordenada després de (4):

	0	1	2	3	4	5	6	7	8	9
tord	17*	17**	17***	18*	18**	19*	19**	20*	20**	23

3.5.7. Ordenació ràpida

Quicksort

Hoare va idear el mètode quicksort, d'ordenació ràpida.

El mètode d'ordenació ràpida, anomenat quicksort (ideat per Hoare) és actualment el més eficient i ràpid dels mètodes d'ordenació interna. És també conegut com a mètode d'ordenació per partició. És d'ordre θ(n*log(n)).

La idea central de l'algorisme consisteix en el següent:

1) S'agafa un element X d'una posició qualsevol de la taula.

2) S'intenta col·locar X a la posició correcta de la taula, de tal manera que tots els elements que es trobin a la seva esquerra siguin menors o iguals a X i tots els elements que es trobin a la seva dreta siguin majors o iguals a X.

3) Es repeteixen els passos anteriors, però ara per als conjunts de dades que es troben a l'esquerra i a la dreta de la posició correcta de X dins la taula.

4) El procés acaba quan tots els elements es troben a la posició correcta dins la taula.

És possible que conegueu el concepte de recursivitat.

> Un algorisme és recursiu si el seu disseny es basa en l'aplicació del propi algorisme per a resoldre un problema més senzill, de manera que una vegada aquest hagi acabat provoca la fi del mateix algorisme.

Del que s'ha exposat es dedueix que l'algorisme d'ordenació ràpida es pot programar mitjançant recursivitat. Però això s'escapa de l'àmbit d'aquest materials.

Només heu de tenir en compte que la majoria de llenguatges aporten la rutina d'ordenació per quicksort. En tal cas no ens caldrà programar-la. La podrem utilitzar directament.

El llenguatge C no es queda enrere i també la incorpora. La seva utilització pressuposa, però, certs coneixements que en aquests moments encara no teniu.