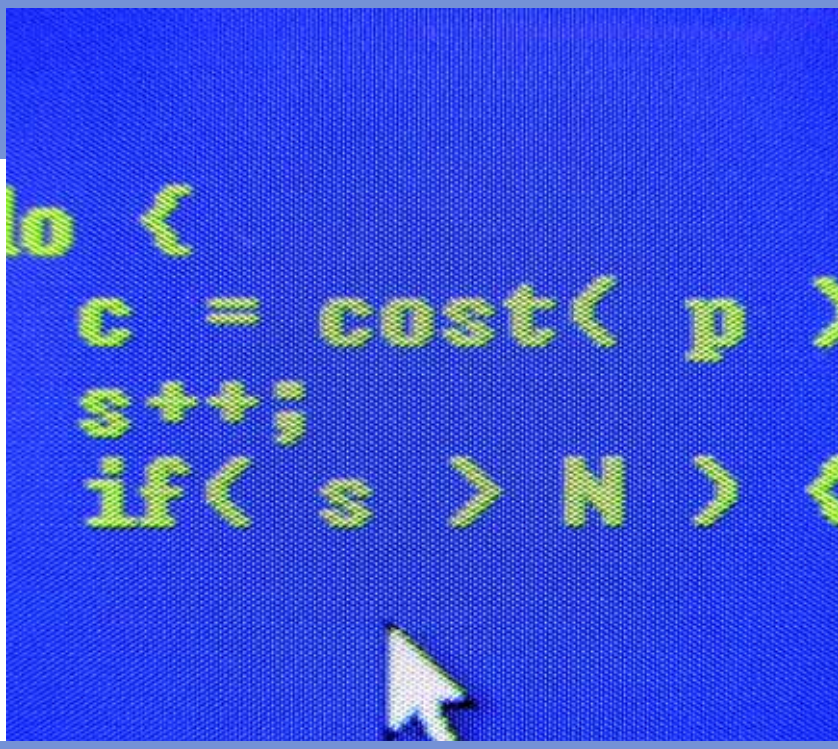
Programari lliure

Josep Anton Pérez López
Lluís Ribas i Xirgo

XP06/M2010/01166



Introducció al desenvolupament de programari

UOC
www.uoc.edu

David Megías Jiménez

Coordinador

Enginyer en Informàtica per la UAB. Màster en Tècniques Avançades d'Automatització de Processos per la UAB.

Doctor en Informàtica per la UAB. Professor dels Estudis d'Informàtica i Multimèdia de la UOC.

Jordi Mas

Coordinador

Enginyer de programari en l'empresa de codi obert Ximian, on treballa en la implementació del projecte lliure Mono. Com a voluntari, col·labora en el desenvolupament del processador de textos Abiword i en l'enginyeria de les versions en català del projecte Mozilla i Gnome. És també coordinador general de Softcatalà. Com a consultor ha treballat per a empreses com Menta, Telépolis, Vodafone, Lotus, eresMas, Amena i Terra España.

Josep Anton Pérez López

Autor

Llicenciat en Informàtica per la Universitat Autònoma de Barcelona. Màster en el programa Gráficos, Tractament d'Imatges i d'Intel·ligència Artificial, per la UAB. Actualment treballa com a professor en un centre d'educació secundària.

Lluís Ribas i Xirgo

Autor

Llicenciat en Ciències-Informàtica i doctorat en Informàtica per la Universitat Autònoma de Barcelona (UAB).

Professor titular del Departament d'Informàtica de la UAB. Consultor dels Estudis d'Informàtica i Multimèdia en la Universitat Oberta de Catalunya (UOC).

Primera edició: Febrer 2006

Material realitzat per Eureca Media, SL

Dipòsit legal: B-15.568-2005

Índex

Agraïments

Els autors agraeixen a la Fundació per a la Universitat Oberta de Catalunya (http://www.uoc.edu) el finançament de la primera edició d'aquesta obra, emmarcada en el màster internacional de Programari Lliure ofert per aquesta institució.

ANOTACIONS

1. Introducció a la programació

1.1. Introducció

En aquest capítol es revisen els fonaments de la programació i els conceptes bàsics del llenguatge C. Se suposa que el lector ja té coneixements previs de programació, sia en el llenguatge C o en algun altre llenguatge.

Per aquest motiu, s'incideix especialment en la metodologia de la programació i en els aspectes crítics del llenguatge C en lloc de posar l'èmfasi en els elements més bàsics de tots dos aspectes.

El coneixement profund d'un llenguatge de programació parteix no solament de la comprensió del seu lèxic, de la seva sintaxi i de la seva semàntica, sinó que a més requereix la comprensió dels objectius que n'han motivat el desenvolupament. Així doncs, en aquesta unitat es repassa la història del llenguatge de programació C des del prisma de la programació dels computadors.

Els programes descrits en un llenguatge de programació com C no els pot executar directament cap màquina. Per tant, és necessari tenir eines (és a dir, programes) que permetin obtenir altres programes que estiguin descrits com una seqüència d'ordres que sí pugui executar directament algun ordinador.

En aquest sentit, es descriurà un entorn de desenvolupament de programari de lliure accés disponible tant en plataformes Microsoft com GNU/Linux. Atès que les primeres requereixen un sistema operatiu que no es basa en el programari lliure, l'explicació se centrarà en les segones.

Nota

Una plataforma és, en aquest context, el conjunt format per un tipus d'ordinador i un sistema operatiu.

La resta de la unitat s'ocuparà del llenguatge de programació C que afecta el paradigma de la programació imperativa i el seu model d'execució. El model d'execució tracta de la manera com es duen a

ANOTACIONS

terme les instruccions indicades en un programa. En el paradigma de la programació imperativa, les instruccions són ordres que es duen a terme de manera immediata per a aconseguir un canvi en l'estat del processador i, en particular, per a l'emmagatzematge dels resultats dels càlculs fets en l'execució de les instruccions. Per aquest motiu, en els últims apartats s'incideix en tot allò que afecta l'avaluació d'expressions (és a dir, el càlcul dels resultats de fórmules), la selecció de la instrucció que cal dur a terme i l'obtenció de dades o la producció de resultats.

En aquest capítol, doncs, es pretén que el lector assoleixi els objectius següents:

1. Repassar els conceptes bàsics de la programació i el model d'execució dels programes.
2. Entendre el paradigma fonamental de la programació imperativa.
3. Adquirir les nocions de C necessàries per al seguiment del curs.
4. Saber utilitzar un entorn de desenvolupament de programari lliure per a la programació en C (GNU/Linux i eines GNU/C).

1.2. Una mica d'història de C

El llenguatge de programació C el va dissenyar Dennis Ritchie als laboratoris Bell per desenvolupar noves versions del sistema operatiu Unix, cap a l'any 1972. D'aquí ve la forta relació entre el C i la màquina.

Nota

Com a curiositat es pot dir que n'hi va haver prou amb unes tretze mil línies de codi C (i un miler escàs en assemblador) per a programar el sistema operatiu Unix de l'ordinador PDP-11.

El llenguatge assemblador és el que té una correspondència directa amb el llenguatge màquina que entén el processador. En altres pa-

ANOTACIONS

raules, cada instrucció del llenguatge màquina es correspon amb una instrucció del llenguatge assemblador.

Al contrari, les instruccions del llenguatge C poden equivaler a petits programes en el llenguatge màquina, però d'ús freqüent en els algoritmes que es programen als ordinadors. És a dir, es tracta d'un llenguatge en què s'empren instruccions que només pot processar una màquina abstracta, que no existeix en la realitat (el processador real només entén el llenguatge màquina). És per això que es parla del C com a llenguatge d'alt nivell d'abstracció i de l'assemblador com a llenguatge de baix nivell.

Aquesta màquina abstracta es construeix parcialment mitjançant un conjunt de programes que s'ocupen de gestionar la màquina real: el ***sistema operatiu***. L'altra part es construeix emprant un programa de traducció del llenguatge d'alt nivell a llenguatge màquina. Aquests programes es diuen ***compiladors*** si generen un codi que pot executar directament l'ordinador, o ***intèrprets*** si es necessita executar-los per a dur a terme el programa descrit en un llenguatge d'alt nivell.

Per al cas que ens ocupa, és molt interessant que el codi dels programes que constitueixen el sistema operatiu sigui el més independent de la màquina possible. Únicament d'aquesta manera serà viable adaptar un sistema operatiu a qualsevol ordinador de manera ràpida i fiable.

D'altra banda, cal que el compilador del llenguatge d'alt nivell sigui extremadament eficient. Per això, i atès els escassos recursos computacionals (fonamentalment, capacitat de memòria i velocitat d'execució) dels ordinadors d'aquells temps, es requereix que el llenguatge sigui simple i permeti una traducció molt ajustada a les característiques dels processadors.

Aquest va ser el motiu que en l'origen del C hi hagués el llenguatge anomenat B, desenvolupat per Ken Thompson per a programar Unix per al PDP-7 el 1970. Evidentment, aquesta versió del sistema operatiu també va incloure una part programada en assemblador, ja que hi havia operacions que només es podien fer amb la màquina real.

ANOTACIONS

És clar que es pot veure el llenguatge C com una versió posterior (i millorada amb inclusió de tipus de dades) del B. Més encara, el llenguatge B estava basat en el BCPL. Aquest llenguatge el va desenvolupar Martín Richards el 1967, amb la paraula *memòria* com a tipus de dades bàsic; és a dir, la unitat d'informació en què es divideix la memòria dels ordinadors. De fet, era un llenguatge assemblador millorat que demanava un compilador molt simple. En canvi, el programador tenia més control de la màquina real.

Malgrat el seu naixement com a llenguatge de programació de sistemes operatius i, per tant, amb la capacitat d'expressar operacions de baix nivell, C és un llenguatge de programació de **propòsit general**. És a dir, es poden programar algoritmes d'aplicacions (conjunts de programes) de característiques molt diferents com, per exemple, programari de comptabilitat d'empreses, maneig de bases de dades de reserves d'avions, gestió de flotes de transport de mercaderies, càlculs científics, etc.

Bibliografia

La definició de les regles sintàctiques i semàntiques del C apareix en l'obra següent:

B.W. Kernighan; D.M. Ritchie (1978). *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall. Concretament en l'apèndix "C Reference Manual".

La relativa simplicitat del llenguatge C, per l'escàs nombre d'instruccions que té, permet que els seus compiladors generin un codi en llenguatge màquina molt eficient i, a més, el converteixen en un llenguatge fàcilment transportable d'una màquina a una altra.

D'altra banda, el repertori d'instruccions de C possibilita fer una programació estructurada d'alt nivell d'abstracció, cosa que redunda en la programació sistemàtica, llegible i de manteniment fàcil.

No obstant això, aquesta simplicitat ha comportat la necessitat de disposar per a C d'un conjunt de funcions molt completes que li do-

ANOTACIONS

nen una gran potència per a desenvolupar aplicacions de tota mena. Moltes d'aquestes funcions són estàndard i estan disponibles a tots els compiladors de C.

Nota

Una funció és una seqüència d'instruccions que s'executa de manera unitària per a dur a terme alguna tasca concreta. Per tant, com més gran sigui el nombre de funcions ja programades, menys codi caldrà programar.

Les funcions estàndard de C es recullen en una biblioteca: la biblioteca estàndard de funcions. Així doncs, qualsevol programa pot emprar totes les funcions que requereixi de la biblioteca, ja que tots els compiladors la tenen.

Finalment, atesa la seva dependència de les funcions estàndard, C promou una programació modular en què els mateixos programadors també poden preparar funcions específiques en els seus programes.

Totes aquestes característiques van permetre una gran difusió de C que va normalitzar l'ANSI (American National Standards Association) el 1989 a partir dels treballs d'un comitè creat el 1983 per a "proporcionar una definició no ambigua i independent de màquina del llenguatge". La segona edició de Kernighan i Ritchie, que es va publicar el 1988, reflecteix aquesta versió que es coneix com a ANSI-C.

Nota

La versió original de C es va conèixer, a partir de llavors, com a K&R C, és a dir, com el *C de Kernighan i Ritchie*. D'aquesta manera, es distingeixen la versió original i l'estandarditzada per l'ANSI.

La resta de la unitat es dedicarà a explicar un entorn de desenvolupament de programes en C i a repassar la sintaxi i la semàntica d'aquest llenguatge.

ANOTACIONS

1.3. Entorn de programació en GNU/C

En el desenvolupament de programari de lliure accés és important emprar eines (programes) que també ho siguin, ja que un dels principis dels programes de lliure accés és que el seu codi pugui ser modificat pels usuaris.

L'ús de codi que pugui dependre de programari privat implica que aquesta possibilitat no existeixi i, per tant, que no se'l pugui considerar lliure. Per a evitar aquest problema, és necessari programar mitjançant programari de lliure accés.

GNU (www.gnu.org) és un projecte de desenvolupament de programari lliure iniciat per Richard Stallman l'any 1984, que té el el suport de la Fundació per al Programari Lliure (FSF).

GNU és un acrònim recursiu (significa *GNU no és Unix*) per a indicar que es tractava del programari d'accés lliure desenvolupat sobre aquest sistema operatiu, però que no consistia en el sistema operatiu. Encara que inicialment el programari desenvolupat utilitzés una plataforma Unix, que és un sistema operatiu de propietat, no es va trigar a incorporar el nucli Linux com a base d'un sistema operatiu independent i complet: el GNU/Linux.

Nota

El nucli (*kernel*) d'un sistema operatiu és el programari que constitueix el seu nucli fonamental i d'aquí ve la seva denominació (*kernel* significa, entre altres coses, 'la part essencial'). Aquest nucli s'ocupa, bàsicament, de gestionar els recursos de la màquina per als programes que s'hi executen.

El nucli Linux, compatible amb el de Unix, va ser desenvolupat per Linus Torvalds el 1991 i va ser incorporat com a nucli del sistema operatiu de GNU un any més tard.

En tot cas, es pot fer notar que totes les anomenades *distribucions de Linux* són, de fet, versions del sistema operatiu GNU/Linux que, per tant, tenen programari de GNU (editor de textos Emacs, compilador

ANOTACIONS

de C, etc.) i també un altre programari lliure com pot ser el processador de textos TeX.

Per al desenvolupament de programes lliures s'emprarà, doncs, un ordinador amb el sistema operatiu Linux i el compilador de C de GNU (gcc). Encara que es pot emprar qualsevol editor de textos ASCII, sembla lògic emprar Emacs. Un editor de textos ASCII és el que només empra els caràcters definits en la taula ASCII per a emmagatzemar els textos. (En l'Emacs, la representació dels textos es pot ajustar per a distingir fàcilment els comentaris del codi del programa en C, per exemple.)

Encara que les explicacions sobre l'entorn de desenvolupament de programari que es faran al llarg d'aquesta unitat i les següents fan referència al programari de GNU, és convenient remarcar que també es poden emprar eines de programari lliure per a Windows, per exemple.

En els apartats següents es veurà com emprar el compilador `gcc` i es revisarà molt succintament l'organització del codi dels programes en C.

1.3.1. Un primer programa en C

Un programa és un text escrit en un llenguatge simple que permet expressar una sèrie d'accions sobre objectes (instruccions) de manera no ambigua.

Abans d'elaborar un programa, com en tot text escrit, haurem de conèixer les regles del llenguatge perquè allò que s'expressi sigui correcte tant lèxicament com sintàcticament.

Les normes del llenguatge de programació C es veuran progressivament al llarg de les unitats corresponents (les tres primeres).

A més a més, haurem de procurar que "això" tingui sentit i expressi exactament allò que es vol que faci el programa. Per si no fos prou,

ANOTACIONS

haurem de cuidar l'aspecte del text de manera que sigui possible captar el seu significat de manera ràpida i clara. Encara que algunes vegades s'indicarà alguna norma d'estil, normalment es descriurà implícitament en els exemples.

Nota

El símbol del dòlar ($) s'empra per a indicar que l'intèrpret d'ordres del sistema operatiu de l'ordinador en pot acceptar una de nova.

Per a escriure un programa en C, només cal executar l'`emacs`:

```
$ emacs hola.c &
```

Nota

El nom del fitxer que conté el programa en C té extensió ".c" perquè sigui fàcil identificar-lo com a tal.

Ara ja podem escriure el programa en C següent:

```
#include <stdio.h>
main( )
{
  printf( "Hola a tots! \n" );
} /* main */
```

Exemple

No és el mateix `printf` que `PrintF`.

És molt important tenir present que, en C (i també en C++ i en Java), **es distingeixen les majúscules de les minúscules**. Per tant, el text del programa ha de ser exactament com s'ha mostrat a excepció del text entre cometes i el que hi ha entre els símbols `/*` i `*/`.

L'editor de textos `emacs` disposa de menús desplegables en la part superior per a la majoria d'operacions i, d'altra banda, accepta ordres introduïdes mitjançant el teclat. Per a això, cal teclejar també la tecla de control ("CTRL" o "C-") o la de caràcter alternatiu juntament amb la del caràcter que indica l'ordre.

A tall de breu resum, es descriuen alguns de les ordres més emprades en la taula següent:

Taula 1.

Ordre	Seqüència	Explicació
Files → Find file	C-x, C-f	Obre un fitxer. El fitxer es copia en una memòria itnermèdia (o *buffer*) o àrea temporal per a la seva edició.
Files → Save buffer	C-x, C-S	Desa el contingut de la memòria intermèdia en el fitxer associat.
Files → Save buffer as	C-x, C-w	Escriu el contingut de la memòria intermèdia en el fitxer que s'hi indiqui.

ANOTACIONS

Ordre	Seqüència	Explicació
Files → Insert file	C-x, C-i	Insereix el contingut del fitxer indicat en la posició del text en el qual es trobi el cursor.
Files → Exit	C-x, C-c	Acaba l'execució d'`emacs`.
(cursor movement)	C-a	Situa el cursor al principi de la línia.
(cursor movement)	C-y	Situa el cursor al final de la línia.
(line killing)	C-k	Elimina la línia (primer el contingut i després el salt de línia).
Edit → Paste	C-y	Enganxa l'últim text eliminat o copiat.
Edit → Copy	C-y, ..., C-y	Per a copiar un tros de text, es pot eliminar primer i recuperar-lo en la mateixa posició i, finalment, en la posició de destinació.
Edit → Cut	C-w	Elimina el text des de l'última marca fins al cursor.
Edit → Undo	C-o	Desfà l'última ordre.

Nota

Per a una millor comprensió de les ordres es recomana llegir el manual d'emacs o, en tot cas, de l'editor que s'hagi triat per a escriure els programes.

Una vegada editat i desat el programa en C, cal compilar-lo per a obtenir un fitxer binari (amb zeros i uns) que contingui una versió del programa traduït a llenguatge màquina. Per a això, cal utilitzar el compilador `gcc`:

```
$ gcc -c hola.c
```

Nota

En un temps `gcc` va significar compilador de C de GNU, però atès que el compilador entén també altres llenguatges, ha passat a ser la col·lecció de compiladors de GNU. Per aquest motiu, és necessari indicar en quin llenguatge s'han escrit els programes mitjançant l'extensió del nom dels fitxers corresponents. En aquest cas, amb `hola.c`, emprarà el compilador de C.

Amb això, s'obtindrà un fitxer (`hola.o`), anomenat *fitxer objecte*. Aquest arxiu conté ja el programa en llenguatge màquina derivat del programa amb el codi C, anomenat també *codi font*. Però per desgràcia encara no és possible executar aquest programa, ja que requereix una funció (`printf`) que es troba en la biblioteca de funcions estàndard de C.

ANOTACIONS

Per a obtenir el codi executable del programa, s'haurà d'enllaçar (en anglès: *link*):

```
$ gcc hola.o -o hola
```

Com que la biblioteca de funcions estàndard està sempre en un lloc conegut pel compilador, no cal indicar-ho en la línia d'ordres. Això sí, s'haurà d'indicar en quin fitxer es vol el resultat (el fitxer executable) amb l'opció `-o` seguida del nom desitjat. En cas contrari, el resultat s'obté en un fitxer anomenat "`a.out`".

Habitualment, el procés de compilació i enllaç es fan directament mitjançant:

```
$ gcc hola.c -o hola
```

Si el fitxer font conté errors sintàctics, el compilador mostrarà els missatges d'error corresponents i els haurem de corregir abans de repetir el procediment de compilació.

En cas que tot hagi anat bé, tindrem un programa executable en un fitxer anomenat `hola` que ens saludarà vehementment cada vegada que l'executem:

```
$ ./hola
Hola a tots!
$
```

Nota

S'ha d'indicar el camí d'accés del fitxer executable perquè l'intèrpret d'ordres el pugui localitzar. Per motius de seguretat, el directori de treball no s'inclou per omissió en el conjunt de camins de recerca d'executables de l'intèrpret d'ordres.

Aquest procediment es repetirà per a cada programa que realitzem en C en un entorn GNU.

ANOTACIONS

1.3.2. Estructura d'un programa simple

En general, un programa en C s'hauria d'organitzar com segueix:

```
/* Fitxer: nom.c                                          */
/* Contingut: exemple d'estructura d'un programa en C     */
/* Autor: nom de l'autor                                  */
/* Revisió: preliminar                                    */

/* ORDRES DEL PREPROCESSADOR                              */
/* -inclusió de fitxers de capçaleres                     */
#include <stdio.h>
/* -definició de constants simbòliques                    */
#define FALS 0

/* FUNCIONS DEL PROGRAMADOR                               */

main( ) /* Funció principal:                              */
{       /* El programa es comença a executar aquí         */
    ... /* Cos de la funció principal                     */
} /* main */
```

En aquesta organització, les primeres línies del fitxer són comentaris que n'identifiquen el contingut, l'autor i la versió. Això és important, ja que cal tenir present que el codi font que creem ha de ser fàcilment utilitzable i modificable per altres persones... I per nosaltres mateixos!

Nota

Els comentaris són textos lliures que s'escriuen entre els dígrafs `/*` i `*/`.

Atesa la simplicitat del C, moltes de les operacions que es realitzen en un programa són, en realitat, crides a funcions estàndard. Així doncs, perquè el compilador conegui quins paràmetres tenen i quins valors tornen, és necessari incloure en el codi del nostre programa les declaracions de les funcions que s'empraran. Per a això, s'utilitza l'ordre `#include` de l'anomenat *preprocessador de C*, que s'ocupa de muntar un fitxer únic d'entrada per al compilador de C.

Els fitxers que contenen declaracions de funcions externes a un determinat arxiu font són anomenats *fitxers de capçaleres* (*header*

ANOTACIONS

files). I per això s'utilitza l'extensió ".h" per a indicar-ne el contingut.

Les capçaleres, en C, són la part en què es declara el nom d'una funció, els paràmetres que rep i el tipus de dada que torna.

Tant aquests fitxers com el del codi font d'un programa poden contenir definicions de constants simbòliques. A continuació presentem una mostra d'aquestes definicions:

```
#define BUIT           '\0'     /* El caracter ASCII NUL    */
#define NOMBRE_OCTAL   0173     /* Un valor octal           */
#define MAX_USUARIS    20
#define CODI_HEXA      0xf39b   /* Un valor hexadecimal     */
#define PI             3.1416   /* Un nombre real           */
#define PRECISIO       1e-10    /* Un altre nombre real     */
#define CADENA         "cadena de caràcters"
```

Aquestes constants simbòliques es reemplacen amb el seu valor al fitxer final que se subministra al compilador de C. És important recalcar que el seu ús ha de redundar en una llegibilitat més gran del codi i, d'altra banda, en una facilitat de canvi del programa, quan sigui necessari.

Cal tenir present que les constants numèriques enteres descrites en base 8, o octal, s'han de prefixar amb un 0 i les expressades en base 16, o hexadecimal, amb "0x".

Exemple

020 no és igual a 20, ja que aquest últim coincideix amb el valor vint en decimal i el primer és un nombre expressat en base 8, la representació binària de la qual és 010000, que equival a 16, en base 10.

Finalment, s'inclourà el programa en el cos de la funció principal. Aquesta funció ha de ser present en tot programa, ja que la primera

ANOTACIONS

instrucció que conté és la que es pren com a punt inicial del programa i, per tant, serà la primera a ser executada.

1.4. La programació imperativa

La programació consisteix en la traducció d'algoritmes a versions en llenguatges de programació que puguin ser executats directament o indirectament per un ordinador.

La majoria d'algoritmes consisteixen en una seqüència de passos que indiquen el que cal fer. Aquestes instruccions solen ser de caràcter imperatiu, és a dir, indiquen el que cal fer de manera incondicional.

La programació dels algoritmes expressats en aquests termes es denomina *programació imperativa*. Així doncs, en aquesta mena de programes, cada instrucció implica fer una acció determinada sobre el seu entorn, en aquest cas, en l'ordinador en què s'executa.

Per a entendre com s'executa una instrucció, és necessari veure com és l'entorn en què es duu a terme. La majoria dels processadors s'organitzen de manera que les dades i les instruccions es troben en la memòria principal i la unitat central de processament (CPU, de la sigla en anglès) és la que realitza l'algoritme següent per poder executar el programa en memòria:

1. Llegir de la memòria la instrucció que cal executar.
2. Llegir de la memòria les dades necessàries per a la seva execució.
3. Fer el càlcul o operació indicada en la instrucció i, segons l'operació que es faci, gravar el resultat en la memòria.
4. Determinar quina és la instrucció següent que cal executar.
5. Tornar al primer pas.

La CPU fa referència a les instruccions i a les dades que demana a la memòria o als resultats que hi vol escriure mitjançant el nombre

ANOTACIONS

de posició que hi ocupen. Aquesta posició que ocupen les dades i les instruccions es coneix com a *adreça de memòria*.

En el nivell més baix, cada adreça diferent de memòria és un únic byte i les dades i les instruccions s'identifiquen per l'adreça del primer dels seus bytes. En aquest nivell, la CPU coincideix amb la CPU física que té l'ordinador.

En el nivell de la màquina abstracta que executa C, es manté el fet que les referències a les dades i a les instruccions sigui l'adreça de la memòria física de l'ordinador, però les instruccions que pot executar la seva CPU d'alt nivell són més potents que les que pot dur a terme la real.

Independentment del nivell d'abstracció en què es treballi, la memòria és, de fet, l'entorn de la CPU. Cada instrucció fa, en aquest model d'execució, un canvi en l'entorn: pot modificar alguna dada en memòria i sempre implica determinar quina és l'adreça de la següent instrucció que s'ha d'executar.

Dit d'una altra manera: l'execució d'una instrucció representa un canvi en l'estat del programa, que es compon de l'adreça de la instrucció que s'està executant i del valor de les dades en memòria. Així doncs, dur a terme una instrucció implica canviar d'estat el programa.

En els pròxims apartats es descriuran els tipus de dades bàsiques que pot emprar un programa en C i les instruccions fonamentals per canviar el seu estat: l'assignació i la selecció condicional de la instrucció següent. Finalment, es veuran les funcions estàndard per a prendre dades de l'exterior (del teclat) i per a mostrar-les a l'exterior (per mitjà de la pantalla).

1.4.1. Tipus de dades bàsiques

Els tipus de dades primitius d'un llenguatge són aquells el tractament de les dades dels quals es pot fer amb les instruccions del mateix llenguatge; és a dir, que estan suportats pel llenguatge de programació corresponent.

ANOTACIONS

Compatibles amb enters

En C, els tipus de dades primitives més comunes són les compatibles amb enters. La representació binària d'aquests no és codificada, sinó que es correspon amb el valor numèric representat en base 2. Per tant, es pot calcular el seu valor numèric en base 10 sumant els productes dels valors intrínsecs (0 o 1) dels seus dígits (bits) pels seus valors posicionals ($2^{posició}$) corresponents.

Es tracten com a nombres naturals, o bé com a representacions d'enters en base 2, si poden ser negatius. En aquest últim cas, el bit més significatiu (el de més a l'esquerra) és sempre un 1 i el valor absolut s'obté restant el nombre natural representat del valor màxim representable amb el mateix nombre de bits més 1.

En tot cas, és important tenir present que el rang de valors d'aquestes dades depèn del nombre de bits que s'empri per a la seva representació. Així doncs, en la taula següent es mostren els diferents tipus de dades compatibles amb enters en un computador de 32 bits amb un sistema GNU.

Taula 2.

Especificació	Nombre de bits	Rang de valors
`(signed) char`	8 (1 byte)	de –128 a +127
`unsigned char`	8 (1 byte)	de 0 a 255
`(signed) short (int)`	16 (2 bytes)	de –32.768 a +32.767
`unsigned short (int)`	16 (2 bytes)	de 0 a 65.535
`(signed) int`	32 (4 bytes)	de –2.147.483.648 a +2.147.483.647
`unsigned (int)`	32 (4 bytes)	de 0 a 4.294.967.295
`(signed) long (int)`	32 (4 bytes)	de –2.147.483.648 a +2.147.483.647
`unsigned long (int)`	32 (4 bytes)	de 0 a 4.294.967.295
`(signed) long long (int)`	64 (8 bytes)	de -2^{63} a $+(2^{63}-1) \approx \pm 9{,}2 \times 10^{18}$
`unsigned long long (int)`	64 (8 bytes)	de 0 a $2^{64}-1 \approx 1{,}8 \times 10^{19}$

Nota

Les paraules de l'especificació entre parèntesis són opcionals en les declaracions de les variables corresponents. D'altra banda, cal tenir present que les especificacions poden variar lleument amb altres compiladors.

ANOTACIÓ

Cal recordar especialment els diferents rangs de valors que poden prendre les variables de cada tipus per al seu ús correcte en els programes. D'aquesta manera, és possible ajustar-ne la mida perquè realment sigui útil.

Exemple

En aquest estàndard, la lletra *A* majúscula es troba en la posició número 65.

El tipus caràcter (`char`) és, de fet, un enter que identifica una posició de la taula de caràcters ASCII. Per a evitar haver de traduir els caràcters a nombres, es poden introduir entre cometes simples (per exemple: `'A'`). També es poden representar codis no visibles com el salt de línia (`'\n'`) o la tabulació (`'\t'`).

Nombres reals

Aquest tipus de dades és més complex que l'anterior, ja que la seva representació binària es troba codificada en diferents camps. Així doncs, no es correspon amb el valor del nombre que es podria extreure dels bits que els formen.

Els nombres reals es representen mitjançant signe, mantissa i exponent. La mantissa expressa la part fraccionària del nombre i l'exponent és el nombre al qual s'eleva la base corresponent:

$$[+/-] \text{ mantissa x base}^{\text{exponent}}$$

En funció del nombre de bits que s'utilitzin per a representar-los, els valors de la mantissa i de l'exponent seran més grans o més petits. Els diferents tipus de reals i els seus rangs aproximats es mostren en la taula següent (vàlida en sistemes GNU de 32 bits):

Taula 3.

Especificació	Nombre de bits	Rang de valors
`float`	32 (4 bytes)	$\pm 3{,}4 \times 10^{\pm 38}$
`double`	64 (8 bytes)	$\pm 1{,}7 \times 10^{\pm 308}$
`long double`	96 (12 bytes)	$\pm 1{,}1 \times 10^{\pm 4.932}$

ANOTACIONS

Com es pot deduir de la taula anterior, és important ajustar el tipus de dades real al rang de valors que podrà adquirir una variable determinada per no ocupar memòria innecessàriament. També s'ha de preveure el contrari: l'ús d'un tipus de dades que no pugui assolir la representació dels valors extrems del rang emprat provocarà que no es representin adequadament i, com a conseqüència, que el programa corresponent es pugui comportar de manera erràtica.

Declaracions

La declaració d'una variable implica manifestar la seva existència davant del compilador i que aquest planifiqui una reserva d'espai en la memòria per a contenir les seves dades. La declaració es fa anteposant al nom de la variable l'especificació del seu tipus (`char`, `int`, `float`, `double`), que pot anar, al seu torn, precedida d'un o diversos modificadors (`signed`, `unsigned`, `short`, `long`). L'ús d'algun modificador fa innecessària l'especificació `int` excepte per a `long`. Per exemple:

```
unsigned short natural;  /* La variable 'natural' es          */
                         /* declara com un                    */
                         /* enter positiu.                    */
int          i, j, k;    /* Variables enteres amb signe.      */
char         opcio;      /* Variable de tipus caràcter.       */
flota        percentil;  /* Variable de tipus real.           */
```

Per comoditat, es pot dotar una variable d'un contingut inicial determinat. Per a això, n'hi ha prou d'afegir a la declaració un signe igual seguit del valor que tindrà en iniciar-se l'execució del programa. Per exemple:

```
int          import = 0;
char         opcio = 'N';
flota        angle = 0.0;
unsigned     comptador = MAXIM;
```

El nom de les variables pot contenir qualsevol combinació de caràcters alfabètics (és a dir, els de l'alfabet anglès, sense 'ñ', ni 'ç', ni cap tipus de caràcter accentuat), numèrics i també el símbol del subratllat (_); però no pot començar per cap dígit.

ANOTACIONS

És convenient que els noms amb què "batejem" les variables identifiquin el seu contingut o la seva utilització en el programa.

1.4.2. L'assignació i l'avaluació d'expressions

Tal com s'ha comentat, en la programació imperativa, l'execució de les instruccions implica canviar l'estat de l'entorn del programa o, el que és el mateix, canviar la referència de la instrucció que s'ha d'executar i, possiblement, el contingut d'alguna variable. Això últim ocorre quan la instrucció que s'executa és la d'assignació:

variable = expressió en termes de variables i valors constants;

La potència (i la possible dificultat de lectura dels programes) de C es troba precisament en les expressions.

Nota

Una expressió és qualsevol combinació sintàcticament vàlida d'operadors i operands que poden ser variables o constants.

De fet, qualsevol expressió es converteix en una instrucció si es posa un punt i coma al final: totes les instruccions de C acaben en punt i coma.

Evidentment, avaluar una expressió no té sentit si no s'assigna el resultat de la seva avaluació a alguna variable que el pugui emmagatzemar per a operacions posteriors. Així doncs, el primer operador que s'ha d'aprendre és el d'assignació:

```
enter = 23;
destinacio = origen;
```

És important no confondre l'operador d'assignació (=) amb el de comparació d'igualtat (==), ja que en C tots dos són operadors que es poden fer servir entre dades del mateix tipus.

ANOTACIONS

Operadors

A part de l'assignació, els operadors més habituals de C i que apareixen en altres llenguatges de programació es mostren en la taula següent:

Taula 4.

Classe	Operadors	Significat
Aritmètico	+ -	suma i resta
	* /	producte i divisió
	%	mòdul o resta de divisió entera (només per a enters)
Relacional	> >=	"major que" i "major i igual que"
	< <=	"menor que" i "menor i igual que"
	== !=	"igual a" i "diferent de"
Lògic	!	*NO* (proposició lògica)
	&& ||	*I* (s'han de complir totes les parts) i *O* lògiques

Els operadors aritmètics serveixen tant per als nombres reals com per als enters. Per aquest motiu es fan implícitament totes les operacions amb el tipus de dades de rang més gran.

A aquest comportament implícit dels operadors se l'anomena *promoció de tipus de dades* i es fa cada vegada que s'opera amb dades de tipus diferents.

Per exemple, el resultat d'una operació amb un enter i un real (les constants reals han de tenir un punt decimal o bé contenir la lletra "e" que separa mantissa d'exponent) serà sempre un nombre real. En canvi, si l'operació s'efectua només amb enters, el resultat serà sempre el del tipus de dades enter de rang més gran. D'aquesta manera:

```
real = 3 / 2 ;
```

té com a resultat assignar a la variable `real` el valor `1.0`, que és el resultat de la divisió entera entre 3 i 2 transformat en un real quan

ANOTACIONS

es fa l'operació d'assignació. Per això se n'escriu 1.0 (amb el punt decimal) en lloc d'1.

Fins i tot amb això, l'operador d'assignació sempre converteix el resultat de l'expressió font al tipus de dades de la variable receptora. Per exemple, l'assignació següent:

```
enter = 3.0 / 2 + 0.5;
```

assigna el valor 2 a `enter`. És a dir, es calcula la divisió real (el nombre 3.0 és real, ja que porta un punt decimal) i se suma 0.5 al resultat. Aquesta suma actua com a factor d'arrodoniment. El resultat és de tipus real i serà truncat (la seva part decimal serà eliminada) en ser assignat a la variable `enter`.

Coerció de tipus

Per a augmentar la llegibilitat del codi, evitar interpretacions equivocades i impedir l'ús erroni de la promoció automàtica de tipus, resulta convenient indicar de manera explícita que es fa un canvi de tipus de dades. Per a això, es pot recórrer a la coerció de tipus; és a dir, posar entre parèntesis el tipus de dades al qual es vol convertir un valor determinat (estigui en una variable o sigui un valor constant):

```
( especificacio_de_tipus ) operand
```

Així doncs, seguint l'exemple anterior, es pot convertir un nombre real a l'enter més pròxim mitjançant un arrodoniment de la manera següent:

```
enter = (int) (real + 0.5);
```

En aquest cas, s'indica que la suma es fa entre dos nombres reals i el resultat, que serà de tipus real, es converteix explícitament a enter mitjançant la coerció de tipus.

Operadors relacionals

Cal tenir present que en C no hi ha cap tipus de dades lògic que es correspongui amb 'fals' i 'cert'. Així doncs, qualsevol dada de tipus

ANOTACIONS

compatible amb enter serà indicació de 'fals' si és 0, i 'cert' si és diferent de 0.

Nota

Això pot no succeir amb les dades de tipus real, ja que cal tenir present que fins i tot els nombres infinitessimals a prop del 0 seran presos com a indicació de resultat lògic 'cert'.

Com a conseqüència d'això, els operadors relacionals tornen 0 per indicar que la relació no es compleix i diferent de 0 en cas afirmatiu. Els operadors && i || només avaluen les expressions necessàries per a determinar, respectivament, si es compleixen tots els casos o només alguns. Així doncs, && implica l'avaluació de l'expressió que en constitueix el segon argument només en cas que el primer hagi resultat positiu. D'una manera semblant, || només executarà l'avaluació del seu segon argument si el primer ha resultat 'fals'.

Així doncs:

```
(20 > 10) || ( 10.0 / 0.0 < 1.0 )
```

donarà com a resultat 'cert' malgrat que el segon argument no es pot avaluar (és una divisió per zero!).

En el cas anterior, els parèntesis eren innecessaris, ja que els operadors relacionals tenen més prioritat que els lògics. Tot i així, és convenient emprar parèntesis en les expressions per a dotar-les de més claredat i aclarir qualsevol dubte sobre l'ordre d'avaluació dels diferents operadors que puguin contenir.

Altres operadors

Com s'ha comentat, C va ser un llenguatge inicialment concebut per a la programació de sistemes operatius i, com a conseqüència, amb un alt grau de relació amb la màquina, que es manifesta en l'existència d'un conjunt d'operadors orientats a facilitar una traducció molt eficient a instruccions del llenguatge màquina.

ANOTACIONS

En particular, disposa dels operadors d'autoincrement (++) i autodecrement (--), que s'apliquen directament sobre variables el contingut de les quals sigui compatible amb enters. Per exemple:

```
comptador++; /* Equivalent a: comptador = comptador + 1; */

--descompte; /* Equivalent a: descompte = descompte - 1; */
```

La diferència entre la forma prefixa (és a dir, que precedeix la variable) i la forma postfixa dels operadors està en el moment en què es fa l'increment o decrement: en forma prefixa, es fa abans d'emprar el contingut de la variable.

Exemple

Vegeu com es modifiquen els continguts de les variables en l'exemple següent:

```
a = 5;   /* ( a == 5 )                */
b = ++a; /* ( a == 6 ) && ( b == 6 ) */
c = b--; /* ( c == 6 ) && ( b == 5 ) */
```

D'altra banda, també es poden fer operacions entre bits. Aquestes operacions es fan entre cada un dels bits que formen part d'una dada compatible amb enter i un altre. Així doncs, és possible fer una I, una O, una O-EX (només un dels dos pot ser cert) i una negació lògica bit a bit entre els que formen part d'una dada i els d'una altra. (Un bit a zero representa 'fals' i un a un, 'cert'.)

Els símbols emprats per a aquests operadors en el nivell de bit són:

- Per a la *I* lògica: `&` (*ampersand*)
- Per a la O lògica: `|` (barra vertical)
- Per a la O exclusiva lògica: `^` (accent circumflex)
- Per a la negació o complement lògic: ~ (titlla)

Malgrat que són operacions vàlides entre dades compatibles amb enters, igual que els operadors lògics, és molt important tenir present que no donen el mateix resultat. Per exemple: `( 1 && 2 )` és cert,

ANOTACIONS

però `( 1 & 2 )` és fals, ja que dóna 0. Per a comprovar-ho, n'hi ha prou de veure el que passa a escala de bit:

```
    1 == 0000 0000 0000 0001
    2 == 0000 0000 0000 0010
1 & 2 == 0000 0000 0000 0000
```

La llista d'operadors en C no acaba aquí. N'hi ha d'altres que veurem més tard i alguns que no es tractaran.

1.4.3. Instruccions de selecció

En el model d'execució dels programes, les instruccions s'executen en seqüència, una després de l'altra, en el mateix ordre en què apareixen en el codi. Certament, aquesta execució purament seqüencial no permetria fer programes molt complexos, ja que sempre farien les mateixes operacions. És per això que s'han de tenir instruccions que permetin controlar el flux d'execució del programa. En altres paraules, disposar d'instruccions que permetin alterar l'ordre seqüencial de la seva execució.

En aquest apartat es comenten les instruccions de C que permeten seleccionar entre diferents seqüències d'instruccions. De manera breu, es resumeixen en la taula següent:

Taula 5.

Instrucció	Significat

`if( condició )`
`  instrucció_si ;`
`else`
`  instrucció_no ;`
	La `condició` ha de ser una expressió l'avaluació de la qual doni com a resultat una dada de tipus compatible amb enter. Si el resultat és diferent de zero, es considera que la condició es compleix i s'executa `instrucció_si`. En cas contrari, s'executa `instrucció_no`. L'**`else`** és opcional.

`switch( expressió ) {`
`  case valor_1 :`
`  instruccions`
`  case valor_2 :`
`  instruccions`
`  default :`
`  instruccions`
`} /* switch */`
	L'avaluació de l'`expressió` ha de resultar en una dada compatible amb enter. Aquest resultat es compara amb els valors indicats en cada `case` i, si és igual a algun d'ells, s'executen totes les instruccions a partir de la primera indicada en el cas i corresponent fins al final del bloc del **`switch`**. Es pot "trencar" aquesta seqüència introduint una instrucció **`break;`** que acaba l'execució de la seqüència d'instruccions. Opcionalment, es pot indicar un cas per omissió (**`default`**) que permet especificar quines instruccions s'executaran si el resultat de l'expressió no produeix cap dada coincident amb els casos previstos.

En el cas de l'`if` és possible executar més d'una instrucció, tant si la condició es compleix com si no, agrupant les instruccions en un

bloc. Els blocs d'instruccions són instruccions agrupades entre claus:

```
{ instruccio_1; instruccio_2; ... instruccio_N; }
```

En aquest sentit, és recomanable que totes les instruccions condicionals agrupin les instruccions que s'han d'executar en cada cas:

```
if( condicio )
{ instruccions }
else
{ instruccions }
/* if */
```

D'aquesta manera s'eviten casos confusos com el següent:

```
if( a > b )
  major = a ;
  menor = b ;
diferencia = major - menor;
```

En aquest cas s'assigna `b` a `menor`, independentment de la condició, ja que l'única instrucció de l'`if` és la d'assignació a `major`.

Com es pot apreciar, les instruccions que pertanyen a un mateix bloc comencen sempre en la mateixa columna. Per a facilitar la identificació dels blocs, han de presentar una sagnia a la dreta respecte de la columna inicial de la instrucció que els governa (en aquest cas: `if`, `switch` i `case`).

Per conveni, cada bloc d'instruccions ha de presentar una sagnia a la dreta respecte de la instrucció que en determina l'execució.

Atès que resulta freqüent haver d'assignar un valor o un altre a una variable en funció d'una condició, és possible, per a aquests casos,

ANOTACIONS

fer servir un operador d'assignació condicional en lloc d'una instrucció `if`:

```
condicio ? expressio_si_cert : expressio_si_fals
```

Així doncs, en lloc del següent:

```
if( condicio )  var = expressio_si_cert;
else            var = expressio_si fals;
```

Es pot escriure:

```
var = condicio ? expressio_si_cert : expressio_si_fals;
```

Més encara, es pot utilitzar aquest operador dins de qualsevol expressió. Per exemple:

```
cost = ( km>km_contracte? km-km_contracte : 0 ) * COST_KM;
```

És preferible limitar l'ús de l'operador condicional als casos en què es faciliti la lectura.

1.4.4. Funcions estàndard d'entrada i de sortida

El llenguatge C només disposa d'operadors i instruccions de control de flux. Qualsevol altra operació que es vulgui fer s'ha de programar o bé s'han d'utilitzar les funcions que hi hagi en la nostra biblioteca de programes.

Nota

Ja hem comentat que una funció no és més que una sèrie d'instruccions que s'executa com una unitat per a dur a terme una tasca concreta. Com a idea, pot agafar la de les funcions matemàtiques, que fan alguna operació amb els arguments donats i tornen el resultat calculat.

ANOTACIONS

El llenguatge C té un ampli conjunt de funcions estàndard, entre les quals hi ha les d'entrada i de sortida de dades, que veurem en aquesta secció.

El compilador de C ha de saber (nom i tipus de dades dels arguments i del valor tornat) quines funcions utilitzarà el nostre programa per a poder generar el codi executable de manera correcta. Per tant, és necessari incloure els fitxers de capçalera que en continguin les declaracions en el codi del nostre programa. En aquest cas:

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

Funcions de sortida estàndard

La sortida estàndard és el lloc on es mostren les dades produïdes (missatges, resultats, etc.) pel programa que està en execució. Normalment, aquesta sortida és la pantalla de l'ordinador o una finestra dins de la pantalla. En aquest últim cas, es tracta de la finestra associada al programa.

```
printf( "format" [, llista_de_camps ] )
```

Aquesta funció imprimeix a la pantalla (la sortida estàndard) el text contingut en "format". En aquest text se substitueixen els caràcters especials, que han d'anar precedits per la barra invertida (\), pel seu significat en ASCII. A més a més, se substitueixen els especificadors de camp, que van precedits per un %, pel valor resultant de l'expressió (normalment, el contingut d'una variable) corresponent indicada en la llista de camps. Aquest valor s'imprimeix segons el format indicat en el mateix especificador.

La taula següent mostra la correspondència entre els símbols de la cadena del format d'impressió i els caràcters ASCII. *n* s'utilitza per a indicar un dígit d'un nombre:

Taula 6.

Indicació de caràcter	Caràcter ASCII
\n	*new line* (salt de línia)
\f	*form feed* (salt de pàgina)

ANOTACIONS

Indicació de caràcter	Caràcter ASCII
\b	*backspace* (retrocés)
\t	*tabulator* (tabulador)
\nnn	ASCII nombre *nnn*
\0nnn	ASCII nombre *nnn* (en octal)
\0Xnn	ASCII nombre *nn* (en hexadecimal)
\\	*backslash* (barra invertida)

Els especificadors de camp tenen el format següent:

```
%[-][+][amplada[.precisio]]tipus_de_dada
```

Els claudàtors indiquen que l'element és opcional. El signe *menys* s'empra per a indicar alineació dreta, cosa habitual en la impressió de nombres. Per a aquests, a més, si s'especifica el signe *més* s'aconseguirà que es mostrin precedits pel seu signe, tant si és positiu com si és negatiu. L'amplada s'emprarà per a indicar el nombre mínim de caràcters que s'utilitzaran per a mostrar el camp corresponent i, en el cas particular dels nombres reals, es pot especificar el nombre de dígits que es vol mostrar en la part decimal mitjançant la precisió. El tipus de dada que es vol mostrar s'ha d'incloure obligatòriament i pot ser una de les següents:

Taula 7.

Enters		Reals		D'altres	
%d	En decimal	%f	En punt flotant	%c	Caràcter
%i	En decimal	%e	Amb format exponencial: [+/-]0.000e[+/-]000 amb e minúscula o majúscula (%E)	%s	Cadena de caràcters
%u	En decimal sense signe			%%	El signe de %
%o	En octal sense signe			(llistat no complet)	
%x	En hexadecimal	%g	En format e, f, o d.		

Exemple

```
printf( "l'import de la factura núm.: %5d", num_fact );
printf( "de Sr./-a. %s puja a %.2f€\n", client, import );
```

Per als tipus numèrics és possible prefixar l'indicador de tipus amb una *ela* a la manera que es fa en la declaració de tipus amb `long`. En aquest cas, el tipus `double` s'ha de tractar com un "`long float`" i, per tant, com a "`%lf`".

ANOTACIONS

Exemple

```
putchar( '\n' );
```

Exemple

```
puts( "Hola a tots!\n" );
```

putchar(caracter **)**

Mostra el caràcter indicat per la sortida estàndard.

puts("cadena de caracters" **)**

Mostra una cadena de caràcters per la sortida estàndard.

Funcions d'entrada estàndard

S'ocupen d'obtenir dades de l'entrada estàndard que, habitualment, es tracta del teclat. Tornen algun valor addicional que informa del resultat del procés de lectura. El valor tornat no té per què emprar-se si no es necessita, ja que moltes vegades se sap que l'entrada de dades es pot efectuar sense gaires problemes.

scanf("format" [, llista_de_&variables] **)**

Nota

La memòria intermèdia del teclat és la memòria en què s'emmagatzema el que s'hi escriu.

Llegeix la memòria intermèdia del teclat per traslladar-ne el contingut a les variables que té com a arguments. La lectura s'efectua segons el format indicat, de manera similar a l'especificació de camps emprada per a `printf`.

Per a poder dipositar les dades llegides en les variables indicades, aquesta funció requereix que els arguments de la llista de variables siguin les adreces de memòria en què es troben. Per aquest motiu, és necessari fer servir l'operador "adreça de" (`&`). D'aquesta manera, `scanf` deixa directament a les zones de memòria corresponent la informació que hagi llegit i, naturalment, la variable afectada veurà modificat el seu contingut amb la nova dada.

És important tenir present que si s'especifiquen menys arguments que especificadors de camp en el format, els resultats poden ser imprevisibles, ja que la funció canviarà el contingut d'alguna zona de memòria totalment aleatòria.

ANOTACIONS

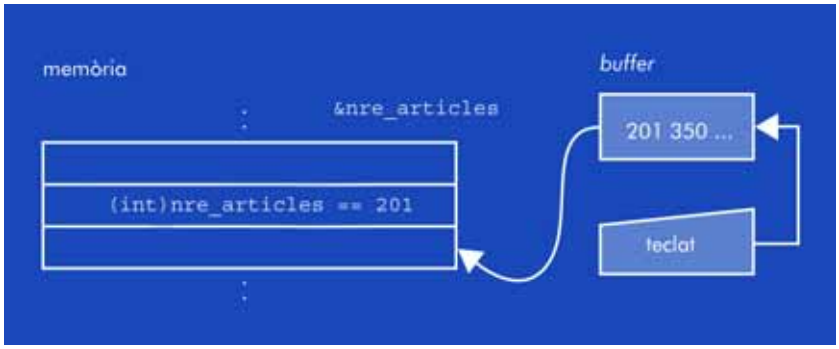
Exemple

```
scanf( "%d", &nre_articles );
```

En l'exemple anterior, `scanf` llegeix els caràcters que s'escriguin per convertir-los (presumiblement) a un enter. El valor que s'obtingui es col·locarà en l'adreça indicada en el seu argument, és a dir, en el de la variable `nre_articles`. Aquesta lectura de caràcters de la memòria intermèdia del teclat es deté quan el caràcter llegit no es correspon amb un possible caràcter del format especificat. Aquest caràcter es torna a la memòria intermèdia per a una possible lectura posterior.

Per a l'entrada que es mostra en la figura següent, la funció deté la lectura després de l'espai en blanc que separa els nombres escrits. Aquest i la resta de caràcters es queden a la memòria intermèdia per a lectures posteriors.

Figura 1.



La funció `scanf` torna el nombre de dades correctament llegides. És a dir, tots aquells per als quals s'ha trobat algun text compatible amb una representació del seu tipus.

`getchar( )`

Torna un caràcter llegit per l'entrada estàndard (habitualment, la memòria intermèdia del teclat).

Exemple

```
opcio = getchar();
```

En cas que no pugui llegir cap caràcter, torna el caràcter `EOF`. Aquesta constant està definida a `stdio.h` i, per tant, es pot emprar per a determinar si la lectura ha tingut èxit o, al contrari, s'ha produït algun error o s'ha arribat al final de les dades d'entrada.

ANOTACIONS

```
gets( cadena_caràcters )
```

Llegeix de l'entrada estàndard tota una sèrie de caràcters fins a trobar un final de línia (caràcter `'\n'`). Aquest últim caràcter és llegit però no s'emmagatzema en la cadena de caràcters que té per argument.

Exemple

```
gets( nom_usuari );
```

Si no llegeix res torna `NULL`, que és una constant definida a `stdio.h` i el valor de la qual és 0.

1.5. Resum

En aquest capítol s'ha vist el procediment d'execució dels programes en un computador. La unitat que s'ocupa de processar la informació (unitat central de processament o CPU) llegeix una instrucció de la memòria i procedeix a la seva execució. Aquesta operació implicarà un canvi de l'estat de l'entorn del programa, és a dir, del contingut d'alguna de les seves variables i de l'adreça de la instrucció següent.

Aquest model d'execució de les instruccions, que segueixen la majoria de processadors reals, es repeteix en el paradigma de la programació imperativa per als llenguatges d'alt nivell. En particular, s'ha vist com això és cert en el llenguatge de programació C.

Per aquest motiu, s'han repassat les instruccions d'aquest llenguatge que permeten modificar l'entorn mitjançant canvis en les dades i canvis en l'ordre seqüencial en què es troben les instruccions en la memòria.

Els canvis que afecten les dades dels programes són, de fet, assignacions a les variables que les contenen. S'ha vist que les variables són espais en la memòria de l'ordinador als quals podem fer referència mitjançant el nom amb què es declaren i als quals, internament, es fa referència mitjançant l'adreça de la primera paraula de la memòria que ocupen.

S'ha recalcat la manera d'avaluar les expressions tenint en compte la prioritat entre els operadors i com es pot organitzar millor mitjan-

ANOTACIONS

çant l'ús de parèntesis. En aquest sentit, s'ha comentat la conveniència d'emprar coercions de tipus explícites per a evitar usos equívocs de la promoció automàtica dels tipus de dades. Aquesta promoció s'ha de fer perquè els operadors puguin treballar sempre amb operands d'un mateix tipus, que sempre coincideix amb el de més rang de representació.

Quant al control del flux d'execució que, normalment, segueix l'ordre en què es troben les instruccions en memòria, s'han esmentat les instruccions bàsiques de selecció de seqüències d'instruccions: la instrucció `if` i la instrucció `switch`.

Aprofitant aquestes explicacions, s'ha introduït la manera especial de considerar les dades lògiques ('fals' i 'cert') en C. Així doncs, qualsevol dada pot ser, en un moment concret, emprat com a valor lògic. En relació amb aquest tema, s'ha comentat la manera especial d'avaluar dels operadors *I* i *O* lògics, que no avaluen les expressions de la dreta en cas que puguin determinar el valor lògic resultant amb el primer argument (el que és donat per l'expressió que els precedeix).

En l'últim apartat, s'han repassat les funcions estàndard bàsiques d'entrada i de sortida de dades, de manera que es puguin construir programes per a provar no solament tots els conceptes i elements de C explicats, sinó també l'entorn de desenvolupament de GNU/C.

Així doncs, es pot comprovar a la pràctica en què consisteix la compilació i l'enllaç dels programes. La tasca del compilador de C és la de traduir el programa en C a un programa en llenguatge màquina. L'enllaçador s'ocupa d'afegir al codi d'aquesta versió el codi de les funcions de la biblioteca que s'utilitzin en el programa. Amb aquest procés, s'obté un codi executable.

1.6. Exercicis d'autoavaluació

1) Editeu, compileu (i enllaceu), executeu i comproveu el funcionament del següent programa:

```
#include <stdio.h>
main()
```

ANOTACIONS

```
{
   int a, b, suma;
   printf( "Escriu un nombre enter: " );
   scanf( "%d", &a );
   printf( "Escriu un altre nombre enter: " );
   scanf( "%d", &b );
   suma = a + b;
   printf( "%d + %d = %d\n", a, b, suma );
} /* main */
```

2) Feu un programa que, amb un import en euros i un percentatge d'IVA determinat, calculi el total.

3) Feu un programa que calculi quant costaria un quilo o un litre d'un producte sabent el preu d'un envàs i la quantitat de producte que conté.

4) Modifiqueu el programa anterior perquè calculi el preu del producte per a la quantitat que es vol, que també s'haurà de donar com a entrada.

5) Feu un programa que calculi el canvi que cal tornar coneixent l'import total que s'ha de cobrar i la quantitat rebuda com a pagament. El programa ha d'advertir si l'import pagat és insuficient.

6) Feu un programa que, donat el nombre de litres aproximat que hi ha al dipòsit d'un cotxe, el seu consum mitjà cada cent quilòmetres i una distància en quilòmetres, indiqui si és pot recórrer. En cas negatiu, ha d'indicar quants litres caldria afegir al dipòsit.

1.6.1. Solucionari

1) N'hi ha prou de seguir els passos indicats. Com a exemple, si el fitxer es digués "suma.c", això és el que s'hauria de fer després d'haver-lo creat:

```
$ gcc suma.c -o suma
$ suma
Escriu un nombre enter: 154
Escriu un altre nombre enter: 703
154 + 703 = 857
$
```

ANOTACIONS

2)

```
#include <stdio.h>
main()
{
   float import, IVA, total;

   printf( "Import = " );
   scanf( "%f", &import );
   printf( "%% IVA = " );
   scanf( "%f", &IVA );
   total = import * ( 1.0 + IVA / 100.0 );
   printf( "Total = %.2f\n", total );
} /* main */
```

3)

```
#include <stdio.h>
main()
{
   float preu, preu_unit;
   int quantitat;

   printf( "Preu = " );
   scanf( "%f", &preu );
   printf( "Quantitat (grams o millilitres) = " );
   scanf( "%d", &quantitat );
   preu_unit = preu * 1000.0 / (float) quantitat;
   printf( "Preu per kg/l = %.2f\n", preu_unit );
} /* main */
```

4)

```
#include <stdio.h>
main()
{
   float preu, preu_unit, preu_compra;
   int quantitat, quanti_compra;

   printf( "Preu = " );
   scanf( "%f", &preu );
   printf( "Quantitat (grams o millilitres) = " );
   scanf( "%d", &quantitat );
   printf( "Quantitat que s'ha d'adquirir = " );
   scanf( "%d", &quanti_compra );
   preu_unit = preu / (float) quantitat;
```

ANOTACIONS

```
   preu_compra = preu_unit * quanti_compra;
   printf( "Preu de compra = %.2f\n", preu_compra );
} /* main */
```

5)

```
#include <stdio.h>
main()
{
   float import, pagament;

   printf( "Import = " );
   scanf( "%f", &import );
   printf( "Pagament = " );
   scanf( "%f", &pagament );
   if( pagament < import ) {
     printf( "Import de pagament insuficient.\n" );
   } else {
     printf("Canvi=%.2feuros.\n",pagament-import );
  } /* if */
} /* main */
```

6)

```
#include <stdio.h>
#define RESERVA 10
main()
{
 int litres, distancia, consum;
 float consum_mitja;
 printf( "Litres al dipòsit = " );
 scanf( "%d", &litres );
 printf( "Consum mitjà cada 100 km = " );
 scanf( "%f", &consum_mitja );
 printf( "Distància que s'ha de recórrer = " );
 scanf( "%d", &distancia );
 consum = consum_mitja * (float) distancia / 100.0;
 if( litres < consum ) {
 printf( "Falten %d l\n", consum-litres+RESERVA );
 } else {
   printf( "Es pot fer el recorregut.\n" );
 } /* if */
} /* main */
```

ANOTACIONS

2. La programació estructurada

2.1. Introducció

La programació eficaç és la que obté un codi llegible i fàcilment actualitzable en un temps de desenvolupament raonable i l'execució de la qual es fa de manera eficient.

Afortunadament, els compiladors i intèrprets de codi de programes escrits en llenguatges d'alt nivell fan certes optimitzacions per reduir el cost de la seva execució. Per exemple, molts compiladors tenen la capacitat d'utilitzar el mateix espai de memòria per a diverses variables, sempre que no s'utilitzin simultàniament, és clar. A més d'aquesta i altres optimitzacions en l'ús de la memòria, també poden incloure millores en el codi executable tendents a disminuir el temps final d'execució. Poden aprofitar, per exemple, els factors comuns de les expressions per a evitar la repetició de càlculs.

Amb tot això, els programadors poden centrar la seva tasca en la preparació de programes llegibles i de manteniment fàcil. Per exemple, no hi ha cap problema a donar noms significatius a les variables, ja que la longitud dels noms no té conseqüències en un codi compilat. En aquest mateix sentit, no resulta lògic emprar trucs de programació orientats a obtenir un codi executable més eficient si això representa disminuir la llegibilitat del codi font. Generalment, els trucs de programació no comporten una millora significativa del cost d'execució i, en canvi, impliquen dificultat de manteniment del programa i dependència d'un entorn de desenvolupament i d'una màquina determinats.

Per aquest motiu, en aquest capítol s'explica com s'organitza el codi font d'un programa. L'organització correcta dels programes implica un increment notable de la seva llegibilitat i, com a conseqüència, una disminució dels errors de programació i facilitat de manteniment i actualització posterior. Més encara, resultarà més fàcil aprofitar parts dels seus codis en altres programes.

ANOTACIONS

Nota

Un salt és un canvi en l'ordre d'execució de les instruccions pel qual la instrucció següent no és la que troba a continuació de la que s'està executant.

En els primers apartats es tracta de la programació estructurada. Aquest paradigma de la programació està basat en la programació imperativa, a la qual imposa restriccions respecte dels salts que es poden efectuar durant l'execució d'un programa.

Amb aquestes restriccions s'aconsegueix augmentar la llegibilitat del codi font, cosa que permet que els lectors determinin amb exactitud el flux d'execució de les instruccions.

És freqüent, en programació, trobar blocs d'instruccions que s'han d'executar repetitivament. Per tant, és necessari veure com es disposa el codi corresponent de manera estructurada. En general, aquests casos deriven de la necessitat de processar una sèrie de dades. Així doncs, en el primer apartat, no solament veurem la programació estructurada, sinó també els esquemes algorítmics per a fer programes que tractin sèries de dades i les corresponents instruccions en C.

Per molt que la programació estructurada estigui encaminada a reduir errors en la progr50amació, no es poden eliminar. Així doncs, es dedica un apartat a la depuració d'errors de programació i a les eines que ens poden ajudar a fer això: els depuradors.

La segona part es dedica a l'organització lògica de les dades dels programes. Cal tenir present que la informació que processen els computadors i el seu resultat està constituït habitualment per una col·lecció variada de dades. Aquestes dades, al seu torn, poden estar constituïdes per d'altres de més simples.

En els llenguatges de programació es dóna suport a uns tipus bàsics de dades, és a dir, s'inclouen mecanismes (declaracions, operacions i instruccions) per a emprar-les en els codis font que s'escriuen amb ells.

Per tant, la representació de la informació que s'utilitza en un programa s'ha de fer en termes de variables que continguin dades dels tipus fonamentals al qual el llenguatge de programació corresponent doni suport. No obstant això, resulta convenient agrupar conjunts de dades que estiguin molt relacionades les unes amb les altres en la informació que representen. Per exemple, manejar com una única en-

ANOTACIONS

titat el nombre de dies de cada un dels mesos de l'any; el dia, el mes i l'any d'una data; o la llista dels dies festius de l'any.

En la segona part, doncs, tractarem dels aspectes de la programació que permeten organitzar les dades en estructures més grans. En particular, veurem les classes d'estructures que hi ha i com s'utilitzen variables que continguin aquestes dades estructurades. També veurem com la definició de tipus de dades a partir d'altres tipus, estructurades o no, beneficia la programació. Com que aquests nous tipus no són reconeguts en el llenguatge de programació, s'anomenen *tipus de dades abstractes*.

Nota

Un tipus de dades abstracte és el que representa una informació no prevista en el llenguatge de programació emprat. Es pot donar el cas que hi hagi tipus de dades suportades en algun llenguatge de programació que, tanmateix, no ho estiguin en d'altres i calgui tractar-los com a abstractes.

L'última part introdueix els principis de la programació modular, que resulta fonamental per a entendre la programació en C. En aquest model de programació, el codi font es divideix en petits programes estructurats que s'ocupen de fer tasques tan específiques dins del programa global. De fet, amb això s'aconsegueix dividir el programa en subprogrames de lectura i comprensió més fàcil. Aquests subprogrames constitueixen els anomenats *mòduls*, i d'aquí es deriva el nom d'aquesta tècnica de programació.

En C tots els mòduls són funcions que solen fer accions molt concretes sobre unes quantes variables del programa. Més encara, cada funció se sol especialitzar en un tipus de dades concret.

Atesa la importància d'aquest tema, s'insistirà en els aspectes de la declaració, definició i ús de les funcions en C. En especial, tot el que es refereix al mecanisme que s'utilitza durant l'execució dels programes per a proporcionar i obtenir dades de les funcions que inclouen.

Finalment, tractarem les macros del preprocessador de C. Aquestes macros tenen una aparença similar a la de les crides de les funcions en C i poden portar a confusió en la interpretació del codi font del programa que les utilitzi.

L'objectiu principal d'aquesta unitat és que el lector aprengui a organitzar correctament el codi font d'un programa, ja que és un indicador fonamental de la qualitat de la programació. De manera més

ANOTACIONS

concreta, l'estudi d'aquest capítol pretén que s'assoleixin els objectius següents:

1. Conèixer en què consisteix la programació estructurada.
2. Saber aplicar correctament els esquemes algorítmics de tractament de seqüències de dades.
3. Identificar els sistemes que s'han d'emprar per a la depuració d'errors d'un programa.
4. Saber quines són les estructures de dades bàsiques i com s'han d'emprar.
5. Saber en què consisteix la programació modular.
6. Conèixer la mecànica de l'execució de les funcions en C.

2.2. Principis de la programació estructurada

Lectures complementàries

E.W. Dijkstra (1968). *The goto statement consireded harmful.*

E.W. Dijkstra (1970). *Notes on structured programming.*

La programació estructurada és una tècnica de programació que va resultar de l'anàlisi de les estructures de control de flux subjacents a tot programa de computador. El producte d'aquest estudi va revelar que és possible construir qualsevol estructura de control de flux mitjançant tres estructures bàsiques: la seqüencial, la condicional i la iterativa.

La programació estructurada consisteix en l'organització del codi de manera que el flux d'execució de les seves instruccions resulti evident als seus lectors.

Un teorema formulat l'any 1966 per Böhm i Jacopini diu que tot "programa propi" hauria de tenir un únic punt d'entrada i un únic punt de sortida, de manera que tota instrucció entre aquests dos punts és executable i no hi ha bucles infinits.

La conjunció d'aquestes propostes proporciona les bases per a la construcció de programes estructurats en què les estructures de con-

ANOTACIONS

trol de flux es poden realitzar mitjançant un conjunt d'instruccions molt reduït.

De fet, l'estructura seqüencial no necessita cap instrucció addicional, ja que els programes s'executen normalment duent a terme les instruccions en l'ordre en el qual apareixen en el codi font.

En el capítol anterior s'ha comentat la instrucció `if`, que permet una execució condicional de blocs d'instruccions. Cal tenir present que, perquè sigui un programa propi, hi ha d'haver la possibilitat que s'executin tots els blocs d'instruccions.

Les estructures de control de flux iteratives es comentaran en l'apartat següent. Val la pena indicar que, quant a la programació estructurada, només cal una única estructura de control de flux iterativa, a partir de la qual es poden construir totes les altres.

2.3. Instruccions iteratives

Les instruccions iteratives són instruccions de control de flux que permeten repetir l'execució d'un bloc d'instruccions. En la taula següent es mostren les que són presents en C:

Taula 8.

instrucció	Significat
`while( condició ) { instruccions } /* while */`	S'executen totes les `instruccions` en el bloc del bucle mentre l'expressió de la `condició` doni com a resultat una dada de tipus compatible amb enter diferent de zero; és a dir, mentre la condició es compleixi. Les `instruccions` poden no executar-se mai.
`do { instruccions } while ( condició );`	De manera similar al bucle `while`, s'executen totes les `instruccions` al bloc del bucle mentre l'expressió de la `condició` es compleixi. La diferència resideix en el fet que les `instruccions` s'executaran, almenys, una vegada. (La comprovació de la condició i, per tant, de la possible repetició de les instruccions, es fa al final del bloc.)
`for( inicialització ; condició ; continuació ) { instruccions } /* for */`	El comportament és semblant a un bucle `while`; és a dir, mentre es compleix la condició s'executen les instruccions del seu bloc. En aquest cas, tanmateix, és possible indicar quina instrucció o instruccions es volen executar de manera prèvia a l'inici del bucle (`inicialització`) i quina instrucció o instruccions s'han d'executar cada vegada que acaba l'execució de les instruccions (`continuació`).

Com es pot apreciar, tots els bucles es poden reduir a un bucle "mentre". Tot i així, hi ha casos en què resulta més lògic emprar alguna de les seves variacions.

Cal tenir present que l'estructura del flux de control d'un programa en un llenguatge d'alt nivell no reflecteix el que realment fa el processador (salts condicionals i incondicionals) en l'aspecte del control del flux de l'execució d'un programa. Tot i així, el llenguatge C té instruccions que ens apropen a la realitat de la màquina, com la de sortida forçada de bucle (`break;`) i la de la continuació forçada de bucle (`continue;`). A més a més, també té una instrucció de salt incondicional (`goto`) que no s'hauria d'emprar en cap programa d'alt nivell.

Normalment, la programació d'un bucle implica determinar quin és el bloc d'instruccions que cal repetir i, sobretot, sota quines condicions cal executar-lo. En aquest sentit, és molt important tenir present que la condició que governa un bucle és la que determina la validesa de la repetició i, especialment, el seu acabament quan no es compleix. Noteu que hi ha d'haver algun cas en què l'avaluació de l'expressió de la condició doni com a resultat un valor 'fals'. En cas contrari, el bucle es repetiria indefinidament (això és el que es diria un cas de "bucle infinit").

Havent determinat el bloc iteratiu i la condició que el governa, també es pot programar la possible preparació de l'entorn abans del bucle i les instruccions que siguin necessàries per a la seva conclusió: la seva inicialització i el seu acabament.

La instrucció iterativa s'ha d'escollir tenint en compte la condició que governa el bucle i de la seva possible inicialització.

En els casos en què sigui possible que no s'executin les instruccions del bucle, és convenient emprar `while`. Per exemple, per a calcular quants divisors té un nombre enter positiu concret:

```
/* ... */
/* Inicialització:  _____________________________*/
divisor = 1;  /* Candidat a divisor */
```

ANOTACIONS

```
ndiv = 0;      /* Nombre de divisors */
/* Bucle:  _____________________________________*/
while( divisor < nombre ) {
 if( nombre % divisor == 0 ) ndiv = ndiv + 1;
 divisor = divisor + 1;
} /* while */
/* Acabament: _____________________________*/
if( nombre > 0 ) ndiv = ndiv + 1;
/* ... */
```

De vegades, la condició que governa el bucle depèn d'alguna variable que es pot prendre com un comptador de repeticions; és a dir, el seu contingut reflecteix el nombre d'iteracions fet. En aquests casos, es pot considerar l'ús d'un bucle `for`. En concret, es podria interpretar com "iterar el següent conjunt d'instruccions per a tots els valors d'un comptador entre un inici i un final donats". En l'exemple següent, es mostra aquesta interpretació en codi C:

```
/* ... */
unsigned int comptador;
/* ... */
for( comptador = INICI ;
 comptador <= FINAL ;
 comptador = comptador + INCREMENT
) {
  instruccions
} /* for */
/* ... */
```

Malgrat que l'exemple anterior és molt comú, no cal que la variable que actuï de comptador s'hagi d'incrementar, ni que ho hagi de fer en un pas fix, ni que la condició només hagi de consistir a comprovar que hagi arribat al seu valor final i, ni molt menys, que sigui una variable addicional que no s'empri en les instruccions del cos que s'ha d'iterar. De fet, seria molt útil que fos una variable el contingut de la qual es modifiqués amb cada iteració i que, amb això, es pogués emprar com a comptador.

ANOTACIONS

És recomanable evitar l'ús de `for` per als casos en què no hi hagi comptadors. En lloc d'aquesta instrucció, és molt millor emprar `while`.

En alguns casos, gran part de la inicialització coincidiria amb el cos del bucle, o bé es fa necessari evidenciar que el bucle s'executarà almenys una vegada. Si aquesta circumstància es dóna, és convenient utilitzar una estructura `do...while`. Com a exemple, vegem el codi d'un programa que fa la suma de diferents imports fins que l'import llegit sigui zero:

```
/* ... */
float total, import;
/* ... */
total = 0.00;
printf( "SUMA" );
do {
   printf( " + " );
   scanf( "%f", &import );
   total = total + import;
} while( import != 0.00 );
printf( " = %.2f", total );
/* ... */
```

La constant numèrica real `0.00` s'utilitza per a indicar que només són significatius dos dígits fraccionaris, ja que, a tots els efectes, tant seria optar per escriure `0.0` o, fins i tot, `0` (en l'últim cas, el nombre enter es convertiria en un nombre real abans de fer l'assignació).

Sigui quin sigui el cas, la norma d'aplicació és mantenir sempre un codi intel·ligible. D'altra banda, l'elecció del tipus d'instrucció iterativa depèn del gust estètic del programador i de la seva experiència, sense més conseqüències en l'eficiència del programa quant al cost d'execució.

2.4. Processament de seqüències de dades

Molta de la informació que es tracta consta de seqüències de dades que es poden donar explícitament o implícitament.

ANOTACIONS

Exemple

En el primer cas, es tractaria del processament d'informació d'una sèrie de dades procedents del dispositiu d'entrada estàndard.

Un exemple del segon seria aquell en què s'han de processar una sèrie de valors que adquireix una mateixa variable.

En tots dos casos, el tractament de la seqüència es pot observar en el codi del programa, ja que s'ha de fer mitjançant alguna instrucció iterativa. Habitualment, aquest bucle es correspon amb un esquema algorítmic determinat. En els apartats següents veurem els dos esquemes fonamentals per al tractament de seqüències de dades.

2.4.1. Esquemes algorítmics: recorregut i recerca

Els esquemes algorítmics per al processament de seqüències de dades són uns patrons que es repeteixen sovint en molts algoritmes. Com a conseqüència, hi ha uns patrons equivalents en els llenguatges de programació com el C. En aquest apartat veurem els patrons bàsics de tractament de seqüències: el de recorregut i el de recerca.

Recorregut

Un recorregut d'una seqüència implica fer un tractament idèntic a tots els seus membres.

En altres paraules, implica tractar cada un dels elements de la seqüència, des del primer fins a l'últim.

Si el nombre d'elements de què constarà la seqüència es coneix *a priori* i la inicialització del bucle és molt simple, llavors pot ser convenient emprar un bucle `for`. En cas contrari, els bucles més adequats són

ANOTACIONS

el bucle `while` o el `do…while` si se sap que hi haurà, almenys, un element en la seqüència de dades.

L'esquema algorítmic del recorregut d'una seqüència seria, en la seva versió per a C, el que es presenta a continuació:

```
/* inicialització per al processament de la seqüència      */
/* (pot incloure el tractament del primer element          */
while(! /* final de seqüència */ ) {
   /* tractar l'element */
   /* avançar en seqüència */
} /* while */
/* acabament del processament de la seqüència              */
/* (pot incloure el tractament de l'últim element)         */
```

El patró anterior es podria fer amb alguna altra instrucció iterativa, si les circumstàncies ho aconsellessin.

Per a il·lustrar diversos exemples de recorregut, suposem que es vol obtenir la temperatura mitjana a la zona d'una estació meteorològica. Per a això, procedirem a fer un programa al qual se subministrin les temperatures registrades a intervals regulars pel termòmetre de l'estació i obtingui la mitjana dels valors introduïts.

Així doncs, el cos del bucle consisteix, simplement, a acumular la temperatura (tractar l'element) i a llegir una nova temperatura (avançar en la seqüència):

```
/* ... */
acumulat = acumulat + temperatura;
quantitat = quantitat + 1;
scanf( "%f", &temperatura );
/* ... */
```

En aquest bloc iteratiu es pot observar que `temperatura` ha de tenir un valor determinat abans de poder-se acumular en la variable `acumulat`, la qual, al seu torn, també ha d'estar inicialitzada. Similarment, `quantitat` s'haurà d'inicialitzar a zero.

ANOTACIONS

És per això que la fase d'inicialització i preparació de la seqüència ja està llesta:

```
/* ... */
unsigned int quantitat;
float acumulat;
float temperatura;
/* ... */
quantitat = 0;
acumulat = 0.00;
scanf( "%f", &temperatura );
/* bucle... */
```

Encara queda per resoldre el problema d'establir la condició d'acabament de la seqüència de dades. En aquest sentit, pot ser que la seqüència de dades tingui una marca de final de seqüència o que la seva longitud sigui coneguda.

En el primer dels casos, la marca de final ha de ser un element especial de la seqüència amb un valor diferent del que pugui tenir qualsevol altra dada. En aquest sentit, com que se sap que una temperatura no pot ser mai inferior a –273,16°C (i, molt menys, una temperatura ambiental), es pot emprar aquest valor com a marca de final. Per claredat, aquesta marca serà una constant definida en el preprocessador:

```
#define MIN_TEMP -273.16
```

Quan es trobi, no s'haurà de processar i, en canvi, sí que s'ha de fer l'acabament del recorregut, calculant la mitjana:

```
/* ... */
float mitjana;
/* ... final del bucle */
if( quantitat > 0 ) {
   mitjana = acumulat / (float) quantitat;
} else {
   mitjana = MIN_TEMP;
} /* if */
/* ... */
```

ANOTACIONS

En el càlcul de la mitjana, es comprova primer que hi hagi alguna dada significativa per a computar-se. En cas contrari, s'assigna a la mitjana la temperatura de marca de final. Tot i així, el codi del recorregut seria el que ensenyem a continuació:

```
/* ... */
quantitat = 0;
acumulat = 0.00;
scanf( "%f", &temperatura );
while(! ( temperatura == MIN_TEMP ) ) {
   acumulat = acumulat + temperatura;
   quantitat = quantitat + 1;
   scanf( "%f", &temperatura );
} /* while */
if( quantitat > 0 ) {
 mitjana = acumulat / (float) quantitat;
} else {
   mitjana = MIN_TEMP;
} /* if */
/* ... */
```

Si la marca de final de seqüència es proporcionés a part, la instrucció iterativa hauria de ser una `do...while`. En aquest cas, se suposarà que la seqüència d'entrada la formen elements amb dues dades: la temperatura i un valor sencer pres com a valor lògic que indica si és l'últim element:

```
/* ... */
quantitat = 0;
acumulat = 0.00;
do {
   scanf( "%f", &temperatura );
   acumulat = acumulat + temperatura;
   quantitat = quantitat + 1;
   scanf( "%u", &es_ultim );
} while(! es_ultim );
if( quantitat >0 ) {
   mitjana = acumulat / (float) quantitat;
} else {
   mitjana = MIN_TEMP;
} /* if */
/* ... */
```

ANOTACIONS

En cas que es conegués el nombre de temperatures (NTEMP) que s'han registrat, n'hi hauria prou d'emprar un bucle de tipus `for`:

```
/* ... */
acumulat = 0.00;
for( quant = 1; quant <= NTEMP; quant = quant + 1
) {
   scanf( "%f", &temperatura );
   acumulat = acumulat + temperatura;
} /* for */
mitjana = acumulat / (float) NTEMP;
/* ... */
```

Cerca

Les cerques consisteixen en recorreguts, majoritàriament parcials, de seqüències de dades d'entrada. Es recorren les dades d'una seqüència d'entrada fins a trobar la que satisfaci una condició determinada. Evidentment, si no es troba cap element que satisfaci la condició, es farà el recorregut complet de la seqüència.

En general, la cerca consisteix a recórrer una seqüència de dades d'entrada fins que es compleixi una determinada condició o s'acabin els elements de la seqüència. No és necessari que la condició afecti un únic element.

Seguint l'exemple anterior, és possible fer una recerca que aturi el recorregut quan la mitjana progressiva es mantingui en un marge de $\pm 1^{o}C$ respecte de la temperatura detectada durant més de deu registres.

L'esquema algorítmic és molt semblant al del recorregut, excepte pel fet que s'incorpora la condició de cerca i que, a la sortida del bucle, és necessari comprovar si la recerca s'ha resolt satisfactòriament o no:

```
/* inicialitzacio per al processament de la sequencia */
/* (pot incloure el tractament del primer element)    */
```

ANOTACIONS

```
trobat = FALS;
while(! /* final de seqüència */ && !trobat ) {
   /* tractar l'element */
   if( /* condició de trobat */ ) {
    trobat = CERT;
   } else {
    /* avançar en seqüència */
   } /* if */
} /* while */
/* acabament del processament de la seqüència          */
if( trobat ) {
   /* instruccions */
} else {
   /* instruccions */
} /* if */
```

En aquest esquema se suposa que s'han definit les constants FALS i CERT de la manera següent:

```
#define FALS 0
#define CERT 1
```

Si s'aplica el patró anterior a la recerca d'una mitjana progressiva estable, el codi font seria el següent:

```
/* ... */
quantitat = 0;
acumulat = 0.00;
scanf( "%f", &temperatura );
seguits = 0;
trobat = FALS;
while(! ( temperatura == MIN_TEMP ) && ! trobat ) {
   acumulat = acumulat + temperatura;
   quantitat = quantitat + 1;
   mitjana = acumulat / (float) quantitat;
   if( mitjana<□=temperatura+1.0 || temperatura-1.0<□=mitjana
{
      seguits = seguits + 1;
   } else {
```

ANOTACIONS

```
    seguits = 0;
   } /* if */
   if( seguits == 10 ) {
    trobat = CERT;
   } else {
    scanf( "%f", &temperatura );
   } /* if */
} /* while */
/* ... */
```

En els casos de cerca no sol ser convenient emprar `for`, ja que sol ser una instrucció iterativa que utilitza un comptador que pren una sèrie de valors des d'un d'inici fins a un final. És a dir, fa un recorregut per la seqüència implícita de tots els valors que pren la variable de càlcul.

2.4.2. Filtres i canonades

Els filtres són programes que generen una seqüència de dades a partir d'un recorregut d'una seqüència de dades d'entrada. Habitualment, la seqüència de dades de sortida conté les dades processades de la d'entrada.

S'hi aplica el nom de *filtres* perquè és molt comú que la seqüència de sortida sigui, simplement, una seqüència de dades com la d'entrada en què s'han suprimit alguns dels seus elements.

Un filtre seria, per exemple, un programa que tingués com a sortida les sumes parcials dels nombres d'entrada:

```
#include <stdio.h>
main()
{
   float suma, sumand;
   suma = 0.00;
   while( scanf( "%f", &sumand ) == 1 ) {
    suma = suma + sumand;
    printf( " %.2f " suma );
   } /* while */
} /* main */
```

ANOTACIONS

Un altre filtre, potser més útil, podria ser un programa que substitueixi els tabuladors pel nombre d'espais en blanc necessaris fins a la columna de tabulació següent:

```
#include <stdio.h>
#define TAB 8
main()
{
   char caracter;
   unsigned short posicio, tabulador;
   posicio = 0;
   caracter = getchar();
   while( caracter != EOF ) {
    switch( caracter ) {
     case '\t':/* avança fins a columna següent */
      for( tabulador = posicio;
         tabulador < TAB;
         tabulador = tabulador + 1 ) {
       putchar( ' ' );
      } /* for */
      posicio = 0;
      break;
     case '\n': /* nova línia implica columna 0 */
      putchar( caracter );
      posicio = 0;
      break;
     default:
      putchar( caracter );
      posicio = (posicio + 1) % TAB;
    } /* switch */
    caracter = getchar();
   } /* while */
} /* main */
```

Aquests petits programes poden resultar útils per si mateixos o bé combinats. Així doncs, la seqüència de dades de sortida d'un pot constituir la seqüència d'entrada d'un altre, i així es constitueix el que anomenem una ***canonada*** (*pipe*, en anglès): la idea visual és que per un extrem de la canonada s'introdueix un flux de dades i per l'altre s'obté un altre flux de dades ja processat. En el camí, la canonada pot incloure un o més filtres que retenen i/o transformen les dades.

ANOTACIONS

Exemple

Un filtre podria convertir una seqüència de dades d'entrada consistents en tres nombres (codi d'article, preu i quantitat) en una seqüència de dades de sortida de dos nombres (codi i import) i el següent podria consistir en un filtre de suma, per recollir l'import total.

Perquè això sigui possible, és necessari tenir l'ajuda del sistema operatiu. Així doncs, no cal que l'entrada de dades s'efectuï per mitjà del teclat ni tampoc que la sortida de dades sigui obligatòriament per la pantalla, com a dispositius d'entrada i sortida estàndard que són. En Linux (i també en altres SO) es pot redirigir l'entrada i la sortida estàndard de dades mitjançant les ordres de redirecció. D'aquesta manera, es pot aconseguir que l'entrada estàndard de dades sigui un fitxer determinat i que les dades de sortida s'emmagatzemin en un altre fitxer que s'empri com a sortida estàndard.

En l'exemple anterior, es pot suposar que hi ha un fitxer (`tiquet.dat`) amb les dades d'entrada i volem obtenir el total de la compra. Per a això, podem emprar un filtre per a calcular els imports parcials, la sortida dels quals serà l'entrada d'un altre que obtingui el total.

Per a aplicar el primer filtre, serà necessari que executem el programa corresponent (que anomenarem `calcula_imports`) redirigint l'entrada estàndard al fitxer `tiquet.dat`, i la sortida al fitxer `imports.dat`:

```
$ calcula_imports < tiquet.dat > imports.dat
```

Amb això, `imports.dat` recollirà la seqüència de parells de dades (codi d'article i import) que el programa hagi generat per la sortida estàndard. Les redireccions es determinen mitjançant els símbols "menor que" per a l'entrada estàndard i "major que" per a la sortida estàndard.

Si volem calcular els imports d'altres compres per després calcular la suma de totes, serà convenient afegir a `imports.dat` tots els imports parcials de tots els tiquets de compra. Això és possible mitjançant l'operador de redirecció de sortida doblat, el significat

ANOTACIONS

del qual podria ser "afegir al fitxer la sortida estàndard del programa":

```
$ calcula_imports < un_altre_tiquet.dat >> imports.dat
```

Quan hàgim recollit tots els imports parcials que vulguem sumar, podrem procedir a cridar el programa que calcula la suma:

```
$ suma <imports.dat
```

Si només ens interessa la suma d'un únic tiquet de compra, podem muntar una canonada en què la sortida del càlcul dels imports parcials sigui l'entrada de la suma:

```
$ calcula_imports <tiquet.dat | suma
```

Com es pot observar en l'ordre anterior, la canonada es munta amb l'operador de canonada representat pel símbol de la barra vertical. Les dades de la sortida estàndard de l'execució del que li precedeix les transmet com a entrada estàndard al programa que tingui a continuació.

2.5. Depuració de programes

La depuració de programes consisteix a eliminar els errors que continguin. Els errors es poden deure tant a la programació com a l'algoritme programat. Així doncs, la depuració d'un programa pot implicar un canvi en l'algoritme corresponent. Quan la causa de l'error és en l'algoritme o en la seva programació incorrecta es parla d'**error de lògica**. En canvi, si l'error té la seva raó en la violació de les normes del llenguatge de programació es parla d'un **error de sintaxi** (encara que alguns errors tinguin una naturalesa lèxica o semàntica).

Nota

Debug és el terme anglès per a referir-se a la depuració d'errors de programes de computador. El verb es

ANOTACIONS

pot traduir per 'eliminar bestioles' i té el seu origen en un informe de 1945 sobre una prova de l'ordinador Mark II feta a la Universitat de Harvard. En l'informe es va registrar que es va trobar una arna en un relé que provocava el seu mal funcionament. Per a provar que s'havia tret la bestiola (i resolt l'error), es va incloure l'arna en el mateix informe. Es va subjectar amb cinta adhesiva i es va afegir un peu que deia "primer cas d'una arna trobada". Va ser també la primera aparició del verb *debug* ('treure bestioles') que va prendre l'accepció actual.

Els errors de sintaxi són detectats pel compilador, ja que li impedeixen generar codi executable. Si el compilador pot generar codi malgrat la possible existència d'un error, el compilador sol emetre un avís.

Per exemple, és possible que una expressió en un `if` contingui un operador d'assignació, però l'habitual és que es tracti d'una confusió entre operadors d'assignació i de comparació:

```
/* ... */
if( a = 5 ) b = 6;
/* ... */
```

Més encara, en el codi anterior es tracta d'un error de programació, ja que la instrucció sembla indicar que és possible que `b` pugui no ser 6. Si atenem la condició d'`if`, es tracta d'una assignació del valor 5 a la variable `a`, amb resultat igual al valor assignat. Així doncs, com que el valor 5 és diferent de zero, el resultat és sempre afirmatiu i, consegüentment, `b` sempre pren el valor 6.

Per aquest motiu, és molt recomanable que el compilador ens doni tots els avisos que pugui. Per a això, s'ha d'executar amb l'argument següent:

```
$ gcc -Wall -o programa programa.c
```

L'argument `-Wall` indica que es doni avís de la majoria de casos en els quals hi pugui haver algun error lògic. Malgrat que l'argument sembla indicar que se'ns avisarà sobre qualsevol situació, encara hi ha alguns casos sobre els quals no avisa.

ANOTACIONS

Els errors més difícils de detectar són els errors lògics que escapen fins i tot als avisos del compilador. Aquests errors són deguts a una programació indeguda de l'algoritme corresponent, o bé al fet que el mateix algoritme és incorrecte. En tot cas, després de la seva detecció cal procedir a la seva localització en el codi font.

Per a la localització dels errors serà necessari determinar en quin estat de l'entorn es produeixen; és a dir, sota quines condicions s'esdevenen. Per tant, cal esbrinar quins valors de les variables condueixen el flux d'execució del programa a la instrucció en què es manifesta l'error.

Malauradament, els errors se solen manifestar en un punt posterior al de l'estat en què realment es produeix l'error del comportament del programa. Així doncs, cal poder observar l'estat del programa en qualsevol moment per a seguir-ne l'evolució fins a la manifestació de l'error per tal de detectar l'error que el causa.

Per a augmentar l'***observabilitat*** d'un programa, és habitual introduir testimonis (també anomenats *espietes*) en el seu codi per tal que ens mostrin el contingut de determinades variables. De totes maneres, aquest procediment comporta la modificació del programa cada vegada que s'introdueixen testimonis nous, s'eliminen els que resulten innecessaris o es modifiquen.

D'altra banda, per a localitzar millor l'error, s'ha de poder tenir control sobre el flux d'execució. La ***controlabilitat*** implica la capacitat de modificar el contingut de les variables i de triar entre diferents fluxos d'execució. Per a aconseguir un grau de control determinat, és necessari introduir canvis significatius en el programa que s'examina.

En lloc de tot l'anterior, és millor emprar una eina que ens permeti observar i controlar l'execució dels programes per a la seva depuració. Aquestes eines són els anomenats *depuradors* (*debuggers*, en anglès).

Perquè un depurador pugui fer la seva feina, és necessari compilar els programes de manera que el codi resultant inclogui informació

ANOTACIONS

relativa al codi font. Així doncs, per a depurar un programa, l'haurem de compilar amb l'opció `-g`:

```
$ gcc -Wall -o programa -g programa.c
```

En GNU/C hi ha un depurador anomenat `gdb` que ens permet executar un programa, fer que es pari en condicions determinades, examinar l'estat del programa quan estigui aturat i, finalment, canviar-lo per poder experimentar possibles solucions.

El depurador s'invoca de la manera següent:

```
$ gdb programa
```

En la taula següent es mostren algunes de les ordres que li podem indicar en GDB:

Taula 8.

Ordre	Acció
`run`	Comença a executar el programa per la seva primera instrucció. El programa només s'aturarà en un punt de parada, quan el depurador rebi un avís de parada (és a dir, amb el tecleig de control i C simultàniament), o bé quan esperi alguna entrada de dades.
`break núm_línia`	Estableix un punt de parada abans de la primera instrucció que es troba en la línia indicada del codi font. Si s'omet el número de línia, llavors l'estableix en la primera instrucció de la línia actual; és a dir, en la qual s'ha aturat.
`clear núm_línia`	Elimina el punt de parada establert en la línia indicada o, si s'omet, en la línia actual.
`c`	Continua l'execució després d'una detenció.
`next`	Executa la instrucció següent i s'atura.
`print expressió`	Imprimeix el resultat d'avaluar l'expressió indicada. En particular, l'expressió pot ser una variable i el resultat de la seva avaluació, el seu contingut.
`help`	Mostra la llista de les ordres.
`quit`	Acaba l'execució de GDB.

Els punts de parada o *breakpoints* són marques en el codi executable que permeten al depurador conèixer si ha de parar l'execució del programa en curs o, al contrari, ha de continuar permetent la seva execució. Aquestes marques es poden fixar o eliminar amb el mateix depurador. Amb això, és possible executar porcions de codi de manera unitària.

En particular, pot ser convenient introduir un punt de parada en `main` abans de procedir a la seva execució, de manera que s'aturi

ANOTACIONS

en la primera instrucció i ens permeti fer-ne un millor seguiment. A aquest efecte, n'hi ha prou amb l'ordre `break main`, ja que és possible indicar noms de funcions com a punts de parada.

De totes maneres, és molt més pràctic emprar algun entorn gràfic en què es pugui veure el codi font mentre la sortida del programa s'executa. Per a això, es pot utilitzar, per exemple, el DDD (*data display debugger*) o l'XXGDB. Tots dos entorns empren el GDB com a depurador i, per tant, disposen de les mateixes opcions. No obstant això, el seu maneig és més fàcil perquè la majoria d'ordres estan a la vista i, en tot cas, als menús desplegables de què disposen.

2.6. Estructures de dades

Els tipus de dades bàsiques (compatibles amb enters i reals) es poden agrupar en estructures homogènies o heterogènies, de manera que es facilita (i aclareix) l'accés als seus components dins d'un programa.

Una estructura homogènia és aquella les dades de la qual són totes del mateix tipus i una d'heterogènia pot estar formada per dades de tipus diferent.

En els apartats següents es revisaran les principals estructures de dades en C, tot i que n'hi ha en tots els llenguatges de programació estructurada. Cada apartat s'organitza de manera que es vegi com es poden dur a terme les operacions següents sobre les variables:

– Declarar-les, perquè el compilador els reservi l'espai corresponent.

– Inicialitzar-les, de manera que el compilador els doni un contingut inicial (que pot canviar) en el programa executable resultant.

– Referenciar-les, de manera que es pugui accedir al seu contingut, tant per a modificar-lo com per a llegir-lo.

Com que és obligatori anteposar el tipus de la variable en la seva declaració, resulta convenient identificar els tipus de dades estructurades amb un nom de tipus. Aquests tipus de dades nous es coneixen com a *tipus de dades abstractes*. L'últim apartat hi estarà dedicat.

ANOTACIONS

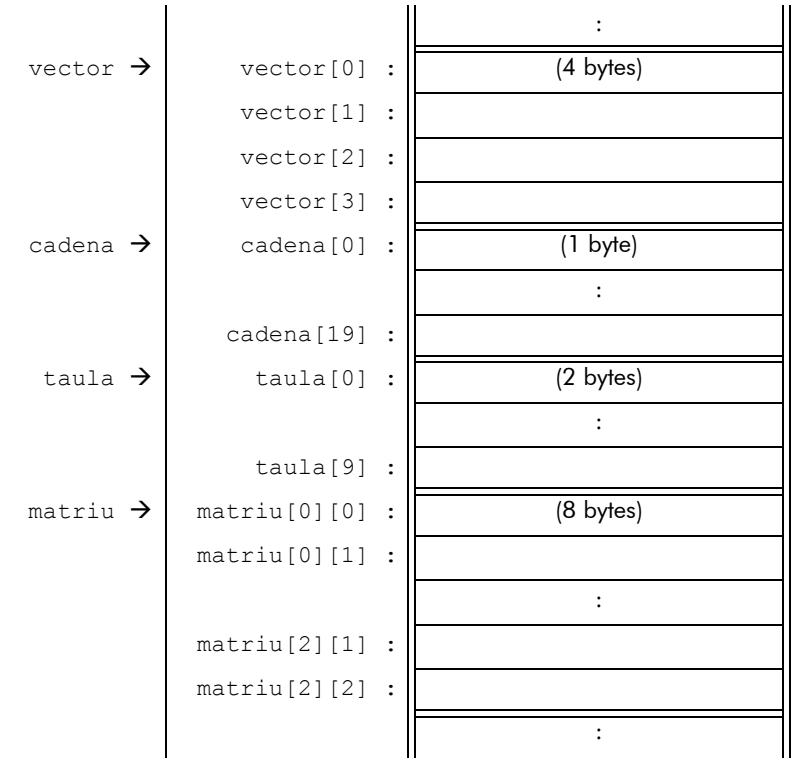
2.7. Matrius

Les matrius són estructures de dades homogènies de mida fixa. És a dir, es representa sempre una informació que utilitza un nombre determinat de dades. També es diuen **arranjaments** (una traducció bastant directa del terme *array* en anglès), **taules** (per a les d'una o dues dimensions) o **vectors** (si són unidimensionals). En el cas particular dels vectors de caràcters, reben el nom de **cadenes de caràcters** o *strings*, en anglès.

2.7.1. Declaració

A continuació us mostrem quatre declaracions de matrius diferents i un esquema de la seva distribució en la memòria del computador. El nombre de bytes de cada divisió depèn del tipus de dada de cada matriu corresponent. En l'esquema ja s'avança el nom de cada dada dins de la matriu, en la qual es distingeix el nom comú de la matriu i una identificació particular de la dada, que es correspon amb la posició de l'element. És molt important tenir present que les posicions sempre es numeren des de 0, en C.

Figura 2.



ANOTACIONS

A continuació, us mostrem les declaracions de les variables que conduirien fins a la distribució en memòria que s'ha vist:

```
int             vector[4];
char            cadena[20];
unsigned short  taula[10];
double          matriu[3][3];
```

La primera declaració prepara un vector de quatre enters amb signe; la segona, una cadena de vint caràcters; la tercera, una taula de deu enters positius, i l'última reserva espai per a una matriu de 3 × 3 nombres reals de doble precisió.

Nota

La matriu quadrada s'emmagatzema en la memòria per files; és a dir, primer apareix la primera fila, després la segona i així fins a l'última.

En cas que calgués declarar una matriu amb un nombre més gran de dimensions, n'hi hauria prou d'afegir la seva mida entre claudàtors en la posició que es vol entre el nom de l'estructura i el punt i coma final.

Com s'ha comentat, les cadenes de caràcters són, realment, matrius unidimensionals en C. És a dir, que tenen una longitud màxima fixada per l'espai reservat al vector que els correspon. Tot i així, les cadenes de caràcters poden ser de longitud variable i, per tant, s'utilitza un marcador de final. En aquest cas, es tracta del caràcter NUL del codi ASCII, el valor numèric del qual és 0. En l'exemple anterior, la cadena pot ser qualsevol text de fins a dinou caràcters, ja que és necessari preveure que l'últim caràcter és el de final de cadena (`'\0'`).

En tots els casos, especialment quan es tracta de variables que hagin de contenir valors constants, es pot donar un valor inicial a cada un dels elements que contenen.

Cal tenir present que les matrius es desen en memòria per files i que és possible no especificar la primera dimensió (la que apareix immediatament després del nom de la variable) d'una matriu. En aquest cas, prendrà les dimensions necessàries per contenir les dades pre-

ANOTACIONS

sents en la seva inicialització. La resta de dimensions han d'estar fixades de manera que cada element de la primera dimensió tingui una ocupació de memòria coneguda.

En els exemples següents podem observar diferents inicialitzacions per a les declaracions de les variables anteriors.

```
 int             vector[4] = { 0, 1, 2, 3 };
 char            cadena[20] = { 'H', ' o ' 'l', ' a '\0' };
unsigned short  taula[10] = { 98, 76, 54, 32, 1, };
double          matriu[3][3] =  {{0.0, 0.1, 0.2},
                                 {1.0, 1.1, 1.2},
                                 {2.0, 2.1, 2.2} };
```

En el cas de la cadena de caràcters, els elements en posicions posteriors a l'ocupada per `'\0'` no tindran cap valor inicial. És més, podran tenir qualsevol valor. Es tracta, doncs, d'una inicialització incompleta.

Per a facilitar la inicialització de les cadenes de caràcters també es pot fer de la manera següent:

```
char cadena[20] = "Hola";
```

Si, a més, la cadena no ha de canviar de valor, és possible aprofitar que no és necessari especificar la dimensió, si es pot calcular per mitjà de la inicialització que es fa de la variable corresponent:

```
char cadena[] = "Hola";
```

En el cas de la taula, es fa una inicialització completa en indicar, amb l'última coma, que tots els elements posteriors tindran el mateix valor que l'últim donat.

2.7.2. Referència

Per a fer referència, en alguna expressió, a un element d'una matriu, n'hi ha prou d'indicar el seu nom i la posició que hi ocupa:

$$\text{matriu}[i_0][i_1]...[i_n]$$

ANOTACIONS

on i_k són expressions el resultat de les quals ha de ser un valor enter. Habitualment, les expressions solen ser molt simples: una variable o una constant.

Per exemple, per a llegir de l'entrada estàndard les dades per a la matriu de reals dobles de 3×3 que s'ha declarat anteriorment, es podria fer el programa següent en el qual, sens dubte, les variables `fila` i `columna` són enters positius:

```
/* ... */
for( fila = 0; fila < 3; fila = fila + 1 ) {
   for( columna = 0; columna < 3; columna = columna + 1 ) {
    printf( "matriu[%u][%u]=?" fila, columna );
    scanf( "%lf" &dada );
    matriu[fila][columna] = dada;
   } /* for */
} /* for */
/* ... */
```

És molt important tenir present que el compilador de C no afegeix cap codi per comprovar la validesa dels índexs de les matrius. Per tant, no es comproven els límits de les matrius i es pot fer referència a qualsevol element, tant si pertany a la matriu com si no. Això és sempre responsabilitat del programador!

A més a més, en C, els claudàtors són operadors d'accés a estructures homogènies de dades (és a dir, matrius) que calculen la posició d'un element a partir de l'adreça de memòria base en què es troben i l'argument que se'ls dóna. Això implica que és possible, per exemple, accedir a una columna d'una matriu quadrada (per exemple: `int A[3][3];`) indicant només el seu primer índex (per exemple: `pcol = A[0];`). Encara més, és possible que es cometi l'error de referir-se a un element en la forma `A[1,2]` (comú en altres llenguatges de programació). En aquest cas, el compilador accepta la referència en tractar-se d'una manera vàlida d'accedir a l'última columna de la matriu, ja que la coma és un operador de concatenació d'expressions el resultat del qual és el de l'última expressió avaluada; és a dir, per a l'exemple donat, la referència `A[1,2]` seria, en realitat, `A[2]`.

ANOTACIONS

2.7.3. Exemples

Exemple

Un dels palíndroms més coneguts en català és el següent: "Senén té sis nens i set nenes".

En aquest primer exemple, el programa comprovarà si una paraula o frase curta és un palíndrom; és a dir, si es llegeix igual d'esquerra a dreta que de dreta a esquerra.

```
#include <stdio.h>
#define LONGITUD 81
#define NUL'\0'
main( )
{
   char            text[LONGITUD];
   unsigned int    longitud, esq, dreta;
   printf( "Comprovació de palíndroms.\n" );
   printf( "Introdueixi text: ");
   gets( text );
   longitud = 0;
   while( text[longitud] != NUL ) {
     longitud = longitud + 1;
   } /* while */
   esq = 0;
   dreta = longitud;
   while( ( text[esq] == text[dreta] ) && ( esq < dreta ) ) {
     esq = esq + 1;
     dreta = dreta - 1;
   } /* while */
   if( esq < dreta ) {
     printf( "No és palíndrom.\n" );
   } else {
     printf( "És palíndrom!\n" );
   } /* if */
} /* main */
```

Com que `gets` pren com a argument la referència de tota la cadena de caràcters, és a dir, l'adreça de la posició inicial de memòria que ocupa, no és necessari emprar l'operador d'"adreça de".

El programa següent que es mostra emmagatzema en un vector els coeficients d'un polinomi per després avaluar-lo en un punt determinat. El polinomi té la forma següent:

$$P(x) = a_{MAX\_GRAU-1}x^{MAX\_GRAU} + \ldots + a_2x^2 + a_1x + a_0$$

ANOTACIONS

Els polinomis serien emmagatzemats en un vector segons la correspondència següent:

$$\begin{aligned} a[MAX\_GRAU-1] &= a_{MAX\_GRAU-1} \\ &\vdots \\ a[2] &= a_2 \\ a[1] &= a_1 \\ a[0] &= a_0 \end{aligned}$$

El programa haurà d'avaluar el polinomi per a una x determinada segons el mètode de Horner, en el qual el polinomi es tracta com si estigués expressat en la forma:

$$P(x) = (\ldots (a_{MAX\_GRAU-1}x + a_{MAX\_GRAU-2})x + \ldots + a_1)x + a_0$$

D'aquesta manera, el coeficient de més grau es multiplica per x i hi suma el coeficient del grau precedent. El resultat es torna a multiplicar per x, sempre que en aquest procés no s'hagi arribat al terme independent. Si fos així, ja s'hauria obtingut el resultat final.

Nota

Amb aquest mètode es redueix el nombre d'operacions que caldrà fer, ja que no s'ha de calcular cap potència de *x*.

```
#include <stdio.h>
#define MAX_GRAU 16
main( )
{
   double a[MAX_GRAU];
   double x, resultat;
   int grau, i;
   printf( "Avaluació de polinomis.\n" );
   for( i = 0; i < MAX_GRAU; i = i + 1 ) {
    a[i] = 0.0;
   } /* for */
   printf( "grau màxim del polinomi = ? " );
   scanf( "%d", &grau );
   if( ( 0 <= grau ) && ( grau < MAX_GRAU ) ) {
    for( i = 0; i <= grau; i = i + 1 ) {
     printf( "a[%d]*x^%d = ? ", i, i );
     scanf( "%lf", &x);
     a[i] = x;
    } /* for */
```

ANOTACIONS

```
    printf( "x = ? " );
    scanf( "%lf", &x );
    resultat = 0.0;
    for( i = grau; i > 0; i = i - 1 ) {
     resultat = x * resultat + a[i-1];
    } /* for */
    printf( "P(%g) = %g\n", x, resultat );
   } else {
    printf( "El grau ha d'estar entre 0 i %d!\n",
     MAX_GRAU-1
    ); /* printf */
   } /* if */
} /* main */
```

És convenient, ara, programar aquests exemples a fi d'adquirir una mica de pràctica en la programació amb matrius.

2.8. Estructures heterogènies

Les estructures de dades heterogènies són aquelles capaces de contenir dades de diferent tipus. Generalment, són agrupacions de dades (tuples) que formen una unitat lògica respecte de la informació que processen els programes que les usen.

Exemple

Les dates (dia, mes i any), les dades personals (nom, cognoms, adreça, població, etc.), les entrades de les guies telefòniques (nombre, propietari, adreça), etc.

2.8.1. Tuples

Els tuples són conjunts de dades de diferent tipus. Cada element dins d'un tuple s'identifica amb un nom de camp específic. Aquests tuples, en C, es denominen *estructures* (`struct`).

De la mateixa manera que succeeix amb les matrius, són útils per a organitzar les dades des d'un punt de vista lògic. Aquesta organització lògica implica poder tractar conjunts de dades fortament relacionades les unes amb les altres com una entitat única. És a dir, que els programes que les emprin reflectiran la seva relació i, per tant, seran molt més intel·ligibles i menys propensos a errors.

ANOTACIONS

Nota

S'aconsegueix molta més claredat si s'empra un tuple per a una data que si s'empren tres enters diferents (dia, mes i any). D'altra banda, les referències als camps de la data inclouen una menció al fet que en són part; cosa que no succeeix si aquestes dades estan contingudes en variables independents.

Declaració

La declaració de les estructures heterogènies o tuples en C comença per la paraula clau `struct`, que ha d'anar seguida d'un bloc de declaracions de les variables que pertanyin a l'estructura i, a continuació, el nom de la variable o els d'una llista de variables que contindran dades del tipus que es declara.

Atès que el procediment que s'acaba de descriure s'ha de repetir per a declarar altres tuples idèntics, és convenient donar un nom (entre `struct` i el bloc de declaracions dels seus camps) a les estructures declarades. Amb això, només és necessari incloure la declaració dels camps de l'estructura en la de la primera variable d'aquest tipus. Per a les altres, n'hi haurà prou d'especificar el nom de l'estructura.

Els noms de les estructures heterogènies solen seguir algun conveni perquè sigui fàcil identificar-les. En aquest cas, en pren un dels més estesos: afegir "`_s`" com a postfix del nom.

L'exemple següent descriu com podria ser una estructura de dades relativa a un avió localitzat per un radar d'un centre de control d'aviació i la variable corresponent (`avio`). Com es pot observar, no es repeteix la declaració dels camps en la posterior declaració d'un vector d'aquestes estructures per contenir la informació de fins a `MAXNAU` avions (se suposa que és la màxima concentració d'avions possible a l'abast d'aquest punt de control i que ha estat definit prèviament):

ANOTACIONS

```
struct avio_s {
       double     radi, angle;
       double     altura;
       char       nombre[33];
       unsigned   codi;
} avio;
struct avio_s avions[MAXNAU];
```

També es poden donar valors inicials a les estructures emprant una assignació al final de la declaració. Els valors dels diferents camps s'han de separar mitjançant comes i anar inclosos entre claus:

```
struct persona_s {
   char           nom[ MAXLONG ];
   unsigned short edat;
} persona = { "Carme", 31 };
struct persona_s guanyadora ={ "desconeguda", 0 };
struct persona_s gent[] =     { { "Eva", 43 },
                                { "Pere", 51 },
                                { "Jesús", 32 },
                                { "Anna", 37 },
                                { "Joaquim", 42 }
                              }; /* struct persona_s gent */
```

Referència

La referència a un camp determinat d'un tuple es fa amb el nom del camp després del nom de la variable que el conté, separant tots dos mitjançant un operador d'accés a camp d'estructura (el punt).

Exemple

```
guanyadora.edat = 25;
inicial = gent[i].nom[0];
```

En el programa següent s'empren variables estructurades que contenen dos nombres reals per a indicar un punt en el pla de manera cartesiana (`struct cartesia_s`) i polar (`struct polar_s`). El programa demana les coordenades cartesianes d'un punt i les transforma en coordenades polars (angle i radi, o distància respecte de l'origen). Observeu que es declaren dues variables amb una inicialització directa: `prec` per a indicar la precisió amb què es treballarà i `pi` per a emmagatzemar el valor de la constant $\prod$ en la mateixa precisió.

ANOTACIONS

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

main( )
{
   struct cartesia_s { double x, y; }  c;
   struct polar_s { double radi, angle; } p;
   double prec = 1e-9;
   double pi = 3.141592654;
   printf( "De coordenades cartesianes a polars.\n" );
   printf( "x = ? " ); scanf( "%lf", &(c.x) );
   printf( " y = ? " ); scanf( "%lf", &(c.y) );
   p.radi = sqrt( c.x * c.x + c.y * c.y );
   if( p.radi < prec ) { /* Si el radi és zero ...       */
    p.angle = 0.0; /* ... l'angle és zero.               */
   } else {
    if( -prec<c.x && c.x<prec ) { /* si c.x és zero ...  */
     if( c.y > 0.0 ) p.angle = 0.5*pi;
     else p.angle = -0.5*pi;
    } else {
     p.angle = atan( c.y / c.x );
    } /* if */
   } /* if */
   printf( "radi = %g\n", p.radi );
   printf( "angle = %g (%g graus sexagesimals)\n",
    p.angle,
    p.angle*180.0/pi
   )          ; /* printf */
} /* main */
```

El programa anterior fa ús de les funcions matemàtiques estàndard `sqrt` i `atan` per a calcular l'arrel quadrada i l'arc tangent, respectivament. Per a això, és necessari que s'inclogui el fitxer de capçaleres (`#include <math.h>`) corresponent en el codi font.

2.8.2. Variables de tipus múltiple

Són variables el contingut de les quals pot variar entre dades de diferent tipus. El tipus de dades ha de ser algun dels que s'indiquen en la seva declaració i el compilador reserva espai per contenir el que ocupi més espai de tots. La seva declaració és semblant a la dels tuples.

ANOTACIONS

Exemple

```
union nombre_s {
  signed enter;
  unsigned natural;
  float real;
} nombre;
```

L'ús d'aquesta classe de variables pot representar un determinat estalvi d'espai. No obstant això, cal tenir present que per a gestionar aquests camps de tipus variable és necessari disposar d'informació (explícita o implícita) del tipus de dada que s'hi emmagatzema en un moment determinat.

Així doncs, solen anar combinats en tuples que disposin d'algun camp que permeti esbrinar el tipus de dades del contingut d'aquestes variables. Per exemple, vegeu la declaració de la variable següent (`asseguranca`), en què el camp `tipus_be` permet conèixer quina de les estructures del tipus múltiple està present en el seu contingut:

```
struct asseguranca_s {
   unsigned  polissa;
   char      prenedor[31];
   char      NIF[9];
   char      tipus_be;    /* 'C': habitatge,   */
                          /* 'F': vida,        */
                          /* 'M': vehicle.     */
   union {
     struct {
       char ref_cadastre[];
       float superficie;
     } habitatge;
     struct {
       struct data_s naixement;
       char beneficiari[31];
     } vida;
     struct {
       char matricula[7];
       struct data_s fabricacio;
       unsigned short sinistres;
     } vehicle;
   } dades;
```

ANOTACIONS

```
    unsigned valor;
    unsigned primera;
} asseguranca;
```

Amb això, també es pot tenir informació sobre una sèrie d'assegurances en una mateixa taula, independentment del tipus de pòlissa que tinguin associats:

```
struct asseguranca_s assegurats[ NUMASSEGURANCES ];
```

En tot cas, l'ús d'`union` resulta bastant poc freqüent.

2.9. Tipus de dades abstractes

Els tipus de dades abstractes són tipus de dades a les quals s'atribueix un significat amb relació al problema que es pretén resoldre amb el programa i, per tant, a un nivell d'abstracció superior al del model computacional. Es tracta, doncs, de tipus de dades transparents al compilador i, consegüentment, irrellevants en el codi executable corresponent.

En la pràctica, qualsevol estructura de dades que es defineixi és, de fet, una agrupació de dades en funció del problema i, per tant, un tipus abstracte de dades. De totes maneres, també ho podria ser un enter que s'utilitzi amb una finalitat diferent, per exemple, només per a emmagatzemar valors lògics ('cert' o 'fals').

En qualsevol cas, l'ús dels tipus abstractes de dades permet augmentar la llegibilitat del programa (entre altres coses que es veuran més endavant). A més a més, fa possible emprar declaracions de tipus anteriorment descrits sense haver de repetir part de la declaració, ja que n'hi ha prou d'indicar el nom que s'hi ha assignat.

2.9.1. Definició de tipus de dades abstractes

Per a definir un nou nom de tipus de dada n'hi ha prou de fer una declaració d'una variable precedida per `typedef`. En aquesta de-

ANOTACIONS

claració, el que seria el nom de la variable serà, de fet, el nom del nou tipus de dades. A continuació, es mostren diversos exemples.

```
typedef char boolean, logic;
#define MAXSTRLEN 81
typedef char cadena[MAXSTRLEN];
typedef struct persona_s {
   cadena     nom, adreca, poblacio;
   char       codi_postal[5];
   unsigned   telefon;
} persona_t;
```

En les definicions anteriors s'observa que la sintaxi no varia respecte de la de la declaració de les variables excepte per la inclusió de la paraula clau `typedef`, el significat de la qual és, precisament, una apòcope de "defineix tipus". Amb aquestes definicions, ja és possible declarar variables dels tipus corresponents:

```
boolean   correcte, ok;
cadena    nom_professor;
persona_t alumnes[MAX_GRUP];
```

És molt recomanable emprar sempre un nom de tipus que identifiqui els continguts de les variables de manera significativa dins del problema que ha de resoldre el programa que les utilitza.

És per això que, a partir d'aquest punt, tots els exemples empraran tipus abstractes de dades quan sigui necessari.

Pel que fa a aquest text, es preferirà que els noms de tipus acabin sempre en "`_t`".

2.9.2. Tipus enumerats

Els tipus de dades enumerats són un tipus de dades compatible amb enters en el qual es fa una correspondència entre un enter i un determinat símbol (la constant d'enumeració). En altres paraules, és un

ANOTACIONS

tipus de dades enter en què es dóna nom (s'enumeren) a un conjunt de valors. En certa manera, el seu ús substitueix l'ordre de definició de constants simbòliques del preprocessador (`#define`) quan són de tipus enter.

L'exemple següent il·lustra com es declaren i com es poden emprar:

```
/* ... */
enum { VERMELL, VERD, BLAU } rgb;
enum bool_e { FALSE = 0, TRUE = 1 } logic;
enum bool_e trobat;
int color;
/* ... */
rgb = VERD;
/* Es pot assignar un enumerat a un enter:     */
color = VERMELL;
logic = TRUE;
/* Es pot assignar un enter a un enumerat,     */
/* encara que no tingui cap símbol associat:   */
trobat = -1;
/* ... */
```

La variable enumerada `rgb` podrà contenir qualsevol valor sencer (de tipus `int`), però tindrà tres valors sencers identificats amb els noms `VERMELL`, `VERD` i `BLAU`. Si el valor associat als símbols importa, s'ha d'assignar a cada símbol el seu valor mitjançant el signe igual, tal com apareix en la declaració de `logic`.

El tipus enumerat pot tenir un nom específic (`bool_e` en l'exemple) que eviti la repetició de l'enumerat en una declaració posterior d'una variable del mateix tipus (en l'exemple: `trobat`).

També és possible i recomanable definir un tipus de dada associada:

```
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
```

En aquest cas particular, s'empra el nom `bool` en lloc de `bool_t` o `logic` per a coincidir amb el nom del tipus de dades primitiu de C++. Atesa la seva freqüència d'ús, la resta del text es considerarà

ANOTACIONS

definit. (No obstant això, caldrà tenir present que una variable d'aquest tipus pot adquirir valors diferents d'1 i ser, conceptualment, 'certa' o `TRUE`.)

2.9.3. Exemple

En els programes vistos abans s'utilitzen variables de tipus de dades estructurades i que sovint són (o podrien ser) utilitzades en altres programes de la mateixa índole. Així doncs, és convenient transformar les declaracions de tipus de les dades estructurades en definició de tipus.

En particular, el programa d'avaluació de polinomis pel mètode de Horner hauria d'haver tingut un tipus de dades estructurades que representés la informació d'un polinomi (grau màxim i coeficients).

El programa que es mostra a continuació conté una definició del tipus de dades `polinomi_t` per a identificar els seus components com a dades d'un mateix polinomi. El grau màxim s'empra també per a saber quins elements del vector contenen els coeficients per a cada grau i quins no. Aquest programa fa la derivació simbòlica d'un polinomi concret (la derivació simbòlica implica obtenir un altre polinomi que representa la derivada de la funció polinòmica donada com a entrada).

```
#include <stdio.h>
#define MAX_GRAU 16
typedef struct polinomi_s {
   int     grau;
   double  a[MAX_GRAU];
} polinomi_t;

main( )
{
   polinomi_t p;
   double     x, coef;
   int        i, grau;
```

ANOTACIONS

```
  p.grau = 0; /* inicialització de (polinomi_t) p       */
  p.a[0] = 0.0;
  printf( "Derivació simbòlica de polinomis.\n" );
  printf( "Grau del polinomi = ? " );
  scanf( "%d", &(p.grau) );
  if( ( 0 <= p.grau ) && ( p.grau < MAX_GRAU ) ) {
   for( i = 0; i <= p.grau; i = i + 1 ) { /* Lectura    */
    printf( "a[%d]*x^%d = ? ", i, i );
    scanf( "%lf", &coef );
    p.a[i] = coef;
   } /* for */
   for( i = 0; i < p.grau; i = i + 1 ) { /* Derivació   */
    p.a[i] = p.a[i+1]*(i+1);
   } /* for */
   if( p.grau >0 ) {
    p.grau = p.grau -1;
   } else {
    p.a[0] = 0.0;
   } /* if */
   printf( "Polinomi derivat:\n" );
   for( i =0; i < p.grau; i = i + 1 ) { /* Impressió    */
    printf( "%g*x^%d + ", p.a[i], i );
   } /* for */
   printf( "%g\n", p.a[i] );
  } else {
   printf( " El grau del polinomi ha d'estar" );
   printf( " entre 0 i %d!\n", MAX_GRAU-1 );
  } /* if */
} /* main */
```

2.10. Fitxers

Exemple

Unitats de disquet, de disc dur, de CD, de DVD, de targetes de memòria, etc.

Els fitxers són una estructura de dades homogènia que té la particularitat de tenir les dades emmagatzemades fora de la memòria principal. De fet són estructures de dades que es troben en l'anomenada *memòria externa* o *secundària* (no obstant això, és possible que alguns fitxers temporals es trobin només en la memòria principal).

Per a accedir a les dades d'un fitxer, l'ordinador ha de tenir els dispositius adequats que siguin capaços de llegir i, opcionalment, escriure en els suports adequats.

Pel fet de residir en suports d'informació permanents, poden mantenir informació entre diferents execucions d'un mateix programa o servir de font i dipòsit d'informació per a qualsevol programa.

Atesa la capacitat de molts d'aquests suports, la mida dels fitxers pot ser molt més gran fins i tot que l'espai de memòria principal disponible. Per aquest motiu, en la memòria principal només es disposa d'una part del contingut dels fitxers en ús i de la informació necessària per al seu maneig.

No menys important és que els fitxers són estructures de dades amb un nombre indefinit d'aquests.

En els pròxims apartats es comentaran els aspectes relacionats amb els fitxers en C, que s'anomenen *fitxers de flux de bytes* (en anglès, *byte streams*). Aquests fitxers són estructures homogènies de dades simples en què cada dada és un únic byte. Habitualment, n'hi ha de dos tipus: els fitxers de text ASCII (cada byte és un caràcter) i els fitxers binaris (cada byte coincideix amb algun byte que forma part d'alguna dada d'algun dels tipus de dades que hi ha).

2.10.1. Fitxers de flux de bytes

Els fitxers de tipus flux de bytes de C són seqüències de bytes que es poden considerar com una còpia del contingut de la memòria (binaris), o com una cadena de caràcters (textuals). En aquest apartat ens ocuparem especialment d'aquests últims perquè són els més habituals.

Com que estan emmagatzemats en un suport extern, és necessari disposar d'informació sobre els fitxers en la memòria principal. En aquest sentit, tota la informació de control d'un fitxer d'aquest tipus i una part de les dades que conté (o que haurà de contenir, en cas d'escriptura) es recull en una única variable de tipus `FILE`.

ANOTACIONS

El tipus de dades `FILE` és un tuple compost, entre altres camps, pel nom del fitxer, la longitud del fitxer, la posició de l'últim byte llegit o escrit i una memòria intermèdia (memòria temporal) que conté `BUFSIZ` bytes del fitxer. Aquest últim és necessari per a evitar accessos al dispositiu perifèric afectat i, ja que les operacions de lectura i escriptura es fan per blocs de bytes, perquè es facin més ràpidament.

Afortunadament, hi ha funcions estàndard per a fer totes les operacions que s'acaben d'insinuar. Igual com l'estructura `FILE` i la constant `BUFSIZ`, estan declarades en el fitxer `stdio.h`. En el pròxim apartat es comentaran les més comunes.

2.10.2. Funcions estàndard per a fitxers

Per a accedir a la informació d'un fitxer, primer s'ha d'"obrir". És a dir, cal localitzar-lo i crear una variable de tipus `FILE`. Per a això, la funció d'obertura dóna com a resultat de la seva execució la posició de memòria en què es troba la variable que crea o `NULL` si no ha aconseguit obrir el fitxer indicat.

Nota

En el tercer cas cal anar amb compte: si el fitxer ja existís, se'n perdria tot el contingut!

Quan s'obre un fitxer, és necessari especificar si es llegirà el seu contingut (`mode_obertura = "r"`), si s'hi volen afegir més dades (`mode_obertura = "a"`), o si es vol crear de nou (`mode_obertura = "w"`).

També és convenient indicar si el fitxer és un fitxer de text (els finals de línia es poden transformar lleugerament) o si és binari. Això s'aconsegueix afegint al mode d'obertura una `"t"` o una `"b"`, respectivament. Si s'omet aquesta informació, el fitxer s'obre en mode text.

Una vegada obert, es pot llegir el seu contingut o bé escriure-hi noves dades. Després d'haver operat amb el fitxer, cal tancar-lo. És a dir, s'ha de dir al sistema operatiu que ja no s'hi treballarà més i, sobretot, cal acabar d'escriure les dades que poguessin haver quedat pendents en la memòria itnermèdia corresponent. De tot això s'ocupa la funció estàndard de tancament de fitxer.

En el codi següent es reflecteix l'esquema algorítmic per al treball amb fitxers i es detalla, a més a més, una funció de reobertura de

ANOTACIONS

fitxers que aprofita la mateixa estructura de dades de control. Cal tenir en compte que això implica tancar el fitxer anteriorment obert:

```
/* ... */
/* Es declara una variable perque contingui                    */
/* la referència de l'estructura FILE:                         */
FILE* fitxer;
/* ... */
fitxer = fopen( nombre_fitxer, mode_obertura );
/* El mode d'obertura pot ser "r" per a lectura,               */
/* "w" per a escriptura, "a" per a afegir i                    */
/* "r+", "w+" o "a+" per a actualització (llegir/escriure).    */
/* S'hi pot afegir el sufix                                    */
/* "t" per a text o "b" per a binari.                          */
if( fitxer != NULL ) {
   /* Tractament de les dades del fitxer.                      */
   /* Possible reobertura del mateix fitxer:                   */
   fitxer = freopen(
          nom_fitxer,
          mode_obertura,
          fitxer
   ); /* freopen */
   /* Tractament de les dades del fitxer.                      */
   fclose( fitxer );
} /* if */
```

En els pròxims apartats es detallen les funcions estàndard per a treballar amb fitxers de flux o *streams*. Les variables que s'empren en els exemples són del tipus adequat per a allò que s'usen i, en particular, `flux` és de tipus `FILE*`; és a dir, referència a estructura de fitxer.

Funcions estàndard d'entrada de dades (lectura) de fitxers

Aquestes funcions són molt similars a les ja vistes per a la lectura de dades procedents de l'entrada estàndard. Tanmateix, serà molt important saber si ja s'ha arribat al final del fitxer i, per tant, ja no hi ha més dades per a la seva lectura.

ANOTACIONS

fscanf(flux, "format" [,llista_de_&variables **)**

De funcionament similar a `scanf()`, torna com a resultat el nombre d'arguments realment llegits. Per tant, ofereix una manera indirecta de determinar si s'ha arribat al final del fitxer. En aquest cas, de totes maneres, activa la condició de final de fitxer. De fet, un nombre més petit d'assignacions es pot deure, simplement, a una entrada inesperada, com per exemple, llegir un caràcter alfabètic per a una conversió `"%d"`.

D'altra banda, aquesta funció torna `EOF` (de l'anglès *end of file*) si s'ha arribat al final de fitxer i no s'ha pogut fer cap assignació. En tot cas, resulta molt més convenient emprar la funció que comprova aquesta condició abans que comprovar-la de manera indirecta mitjançant el nombre de paràmetres correctament llegits (pot ser que el fitxer contingui més dades) o pel retorn d'`EOF` (no es produeix si s'ha llegit, almenys, una dada).

Exemple

```
fscanf( flux, "%u%c", &num_dni, &lletra_nif );
fscanf( flux, "%d%d%d", &codi, &preu, &quantitat );
```

feof(flux **)**

Torna 0 en cas que no s'hagi arribat al final del fitxer. En cas contrari torna un valor diferent de zero, és a dir, que és certa la condició de final de fitxer.

fgetc(flux **)**

Llegeix un caràcter del flux. En cas de no poder efectuar la lectura perquè ha arribat al final, torna `EOF`. Aquesta constant ja està definida en el fitxer de capçaleres `stdio.h`; per tant, es pot utilitzar lliurement dins del codi.

Nota

És important tenir present que hi pot haver fitxers que tinguin un caràcter `EOF` al mig del seu flux, ja que el final del fitxer és determinat per la seva longitud.

ANOTACIONS

fgets(cadena, longitud_maxima, flux **)**

Llegeix una cadena de caràcters del fitxer fins a trobar un final de línia, fins a arribar a `longitud_maxima` (–1 per a la marca de final de cadena) de caràcters, o fins al final de fitxer. Torna `NULL` si troba el final de fitxer durant la lectura.

Exemple

```
if( fgets( cadena, 33, flux ) != NULL ) puts( cadena );
```

Funcions estàndard de sortida de dades (escriptura) de fitxers

Les funcions que s'inclouen aquí també tenen un comportament similar al de les funcions per a la sortida de dades pel dispositiu estàndard. Totes escriuen caràcters en el flux de sortida indicat:

fprintf(flux, "format" [llista_de_variables] **)**

La funció `fprintf()` escriu caràcters en el flux de sortida indicat amb format. Si hi ha hagut cap problema, aquesta funció torna l'últim caràcter escrit o la constant `EOF`.

fputc(caracter, flux **)**

La funció `fputc()` escriu caràcters en el flux de sortida indicat caràcter a caràcter. Si s'ha produït un error d'escriptura o bé el suport és ple, la funció `fputc()` activa un indicador d'error del fitxer. Aquest indicador es pot consultar amb la funció `ferror( flux )`, que retorna un zero (valor lògic fals) quan no hi ha error.

fputs(cadena, flux **)**

La funció `fputs()` escriu caràcters en el flux de sortida indicat i permet gravar cadenes completes. Si hi ha hagut cap problema, aquesta funció actua de manera semblant a `fprintf()`.

ANOTACIONS

Funcions estàndard de posicionament en fitxers de flux

En els fitxers de flux és possible determinar la posició de lectura o escriptura; és a dir, la posició de l'últim byte llegit o que s'ha escrit. Això es fa mitjançant la funció `ftell( flux )`, que torna un enter de tipus `long` que indica la posició, o –1 en cas d'error.

També hi ha funcions per a canviar aquesta posició de lectura (i escriptura, si es tracta de fitxers que cal actualitzar):

`fseek(` `flux, desplaçament, adreça` **`)`**

Desplaça el "capçal" lector/escriptor respecte de la posició actual amb el valor de l'enter llarg indicat a `desplaçament` si `adreça` és igual a `SEEK_CUR`. Si aquesta adreça és `SEEK_SET`, llavors `desplaçament` es converteix en un desplaçament respecte del principi i, per tant, indica la posició final. En canvi, si és `SEEK_END`, indicarà el desplaçament respecte de l'última posició del fitxer. Si el reposicionament és correcte, torna 0.

`rewind(` `flux` **`)`**

Situa el "capçal" al principi del fitxer. Aquesta funció és equivalent a:

```
seek( flux, 0L, SEEK_SET );
```

on la constant de tipus `long int` s'indica amb el sufix "L". Aquesta funció permetria, doncs, rellegir un fitxer des del principi.

Relació amb les funcions d'entrada/sortida per dispositius estàndard

Les entrades i sortides per terminal estàndard es poden dur a terme amb les funcions estàndard d'entrada/sortida, o també mitjançant les funcions de tractament de fitxers de flux. Per a això últim és necessari emprar les referències als fitxers dels dispositius estàndard, que s'obren en iniciar l'execució d'un programa. A C hi ha, com a mínim, tres fitxers predefinits: `stdin` per a l'entrada estàndard, `stdout` per a la sortida estàndard i `stderr` per a la sortida d'avisos d'error, que sol coincidir amb `stdout`.

ANOTACIONS

2.10.3. Exemple

Es mostra aquí un petit programa que compta el nombre de paraules i de línies que hi ha en un fitxer de text. El programa entén com a paraula tota mena de caràcters entre dos espais en blanc. Un espai en blanc és qualsevol caràcter que faci que `isspace()` es torni cert. El final de línia s'indica amb el caràcter de retorn de carro (ASCII número 13); és a dir, amb `'\n'`. És important observar l'ús de les funcions relacionades amb els fitxers de flux de bytes.

Com es veurà, l'estructura dels programes que treballen amb aquests fitxers inclou la codificació d'algun dels esquemes algorítmics per al tractament de seqüències de dades (de fet, els fitxers de flux són seqüències de bytes). En aquest cas, com que es fa un recompte de paraules i línies, cal recórrer tota la seqüència d'entrada. Per tant, es pot observar que el codi del programa segueix perfectament l'esquema algorítmic per al recorregut de seqüències.

```
/* Fitxer: nparaules.c                                          */
#include <stdio.h>
#include <ctype.h> /* Conté: isspace()                          */
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;

main()
{
   char nom_fitxer[FILENAME_MAX];
   FILE *flux;
   bool en_paraula;
   char c;
   unsigned long int nparaules, nlinies;

   printf( "Comptador de paraules i línies.\n" );
   printf( "Nom del fitxer: " );
   gets( nom_fitxer );
   flux = fopen( nom_fitxer, "rt" );
   if( flux != NULL ) {
    nparaules = 0;
    nlinies = 0;
    en_paraula = FALSE;
```

ANOTACIONS

```
    while(! feof( flux ) ) {
     c = fgetc( flux );
     if( c == '\n' ) nlinies = nlinies + 1;
     if( isspace( c ) ) {
      if( en_paraula ) {
       en_paraula = FALSE;
       nparaules = nparaules + 1;
      } /* if */
     } else { /* si el caràcter no és espai en blanc          */
      en_paraula = TRUE;
     } /* if */
    } /* while */
    printf( "Nombre de paraules = %lu\n", nparaules );
    printf( "Nombre de línies = %lu\n", nlinies );
   } else {
    printf( " No puc obrir el fitxer!\n" );
   } /* if */
} /* main */
```

Nota

La detecció de les paraules es fa comprovant els finals de paraula, que han d'estar formades per un caràcter diferent de l'espai en blanc, seguit d'un que ho sigui.

2.11. Principis de la programació modular

La lectura del codi font d'un programa implica fer el seguiment del flux d'execució de les seves instruccions (el flux de control). Evidentment, una execució en l'ordre seqüencial de les instruccions no necessita gaire atenció. Però ja hem vist que els programes contenen també instruccions condicionals o alternatives i iteratives. Tot i així, el seguiment del flux de control pot resultar complex si el codi font ocupa més del que es pot observar (per exemple, més d'una vintena de línies).

És per això que resulta convenient agrupar les parts del codi que fan una funció molt concreta en un subprograma identificat de manera individual. És més, això resulta fins i tot profitós quan es tracta de funcions que es fan en diversos moments de l'execució d'un programa.

ANOTACIONS

En els apartats següents veurem com es distribueix en diferents subprogrames un programa en C. L'organització és semblant en altres llenguatges de programació.

2.12. Funcions

En C, les agrupacions de codi en què es divideix un determinat programa s'anomenen, precisament, *funcions*. Més encara, en C, tot el codi ha d'estar distribuït en funcions i, de fet, el mateix programa principal és una funció: la funció principal (`main`).

Generalment, una funció inclourà en el seu codi, a tot estirar, la programació d'uns quants esquemes algorítmics de processament de seqüències de dades i algunes execucions condicionals o alternatives. És a dir, el necessari per a fer una tasca molt concreta.

2.12.1. Declaració i definició

La declaració de qualsevol entitat (variable o funció) implica la manifestació de la seva existència al compilador, mentre que definir-la representa descriure el seu contingut. Aquesta diferenciació ja s'ha vist per a les variables, però només s'ha insinuat per a les funcions.

La declaració consisteix exactament en el mateix que per a les variables: manifestar la seva existència. En aquest cas, de totes maneres, cal descriure els arguments que pren i el resultat que torna perquè el compilador pugui generar el codi, i poder-les emprar.

Els fitxers de capçalera contenen declaracions de funcions.

En canvi, la definició d'una funció es correspon amb el seu programa, que és el seu contingut. De fet, de manera similar a les variables,

ANOTACIONS

el contingut es pot identificar per la posició del primer dels seus bytes en la memòria principal. Aquest primer byte és el primer de la primera instrucció que s'executa per a dur a terme la tasca que tingui programada.

Declaracions

La declaració d'una funció consisteix a especificar el tipus de dada que torna, el nom de la funció, la llista de paràmetres que rep entre parèntesis i un punt i coma que acaba la declaració:

```
tipus_de_dada nom_funcio( llista_de_parametres );
```

Cal tenir present que no es pot fer referència a funcions que no s'hagin declarat prèviament. Per aquest motiu, és necessari incloure els fitxers de capçaleres de les funcions estàndard de la biblioteca de C com a `stdio.h`, per exemple.

Si una funció no ha estat declarada prèviament, el compilador suposarà que torna un enter. Igualment, si s'omet el tipus de dada que retorna, suposarà que és un enter.

La llista de paràmetres és opcional i consisteix en una llista de declaracions de variables que contindran les dades preses com a arguments de la funció. Cada declaració se separa de la següent per mitjà d'una coma. Per exemple:

```
float nota_mitjana( float teo, float prb, float pract );
bool aprova( float nota, float tolerancia );
```

Si la funció no torna cap valor o no necessita cap argument, s'ha d'indicar mitjançant el tipus de dades buit (`void`):

```
void avisa( char missatge[] );
bool si_o_no( void );
int llegeix_codi( void );
```

ANOTACIONS

Definicions

La definició d'una funció sempre està encapçalada per la seva declaració, que ara ha d'incloure forçosament la llista de paràmetres si els té. Aquesta capçalera no ha d'acabar amb punt i coma, sinó que anirà seguida del cos de la funció, delimitada entre claus d'obertura i tancament:

```
tipus_de_dada nom_funcio( llista_de_parametres )
{ /* cos de la funció:                        */
  /* 1) declaració de variables locals       */
  /* 2) instruccions de la funció            */
} /* nom_funció                               */
```

Tal com s'ha comentat anteriorment, la definició de la funció ja implica la seva declaració. Per tant, les funcions que fan tasques d'altres programes i, en particular, del programa principal (la funció `main`) es defineixen amb anterioritat.

Crides

El mecanisme d'ús d'una funció en el codi és el mateix que s'ha utilitzat per a les funcions de la biblioteca estàndard de C: n'hi ha prou de referir-s'hi pel seu nom, proporcionar-los els arguments necessaris perquè puguin dur a terme la tasca que els correspongui i, opcionalment, emprar la dada tornada dins d'una expressió, que serà, habitualment, de condició o d'assignació.

El procediment pel qual el flux d'execució d'instruccions passa a la primera instrucció d'una funció s'anomena *procediment de crida*. Així doncs, es parlarà de crides a funcions cada vegada que s'indiqui l'ús d'una funció en un programa.

A continuació es presenta la seqüència d'un procediment de crida:

1. Preparar l'entorn d'execució de la funció; és a dir, reservar espai per al valor de retorn, els paràmetres formals (les variables que s'identifiquen amb cada un dels arguments que té), i les variables locals.

ANOTACIONS

2. Fer el pas de paràmetres; és a dir, copiar els valors resultants d'avaluar les expressions en cada un dels arguments de la instrucció de crida als paràmetres formals.

3. Executar el programa corresponent.

4. Alliberar l'espai ocupat per l'entorn local i tornar el possible valor de retorn abans de tornar al flux d'execució d'instruccions on era la crida.

L'últim punt es fa mitjançant la instrucció de retorn que, és clar, és l'última instrucció que s'executarà en una funció:

```
return expressio;
```

Nota

Aquesta instrucció ha d'aparèixer buida o no aparèixer si la funció és de tipus `void`; és a dir, si ha estat declarada explícitament per a no tornar cap dada.

En el cos de la funció es pot fer una crida a ella mateixa. Aquesta crida s'anomena *crida recursiva,* ja que la definició de la funció es fa en els seus termes. Aquesta mena de crides no és incorrecta però cal vigilar que no es produeixin indefinidament; és a dir, que hi hagi algun cas on el flux d'execució de les instruccions no impliqui fer cap crida recursiva i, d'altra banda, que la transformació que s'aplica als seus paràmetres condueixi, en algun moment, a les condicions d'execució anterior. En particular, no es pot fer el següent:

```
/* ... */
void menu( void )
{
  /* Mostrar menú d'opcions,         */
  /* Executar opció seleccionada     */
  menu();
/* ... */
```

La funció anterior comporta fer un nombre indefinit de crides a `menu()` i, per tant, la contínua creació d'entorns locals sense el seu posterior alliberament. En aquesta situació, és possible que el pro-

ANOTACIONS

grama no es pugui executar correctament després d'un quant temps per falta de memòria per a crear nous entorns.

2.12.2. Àmbit de les variables

L'àmbit de les variables fa referència a les parts del programa que les poden utilitzar. Dit d'una altra manera, l'àmbit de les variables inclou totes aquelles instruccions que hi poden accedir.

En el codi d'una funció es poden emprar totes les variables globals (les que són "visibles" per qualsevol instrucció del programa), tots els paràmetres formals (les variables que equivalen als arguments de la funció), i totes les variables locals (les que es declaren dins del cos de la funció).

En alguns casos pot no ser convenient utilitzar variables globals, perquè fer-ho dificultaria la comprensió del codi font i, al seu torn, la depuració i el manteniment del programa posteriors. Per a il·lustrar-ho, vegem l'exemple següent:

```
#include <stdio.h>

unsigned int A, B;

void reduce( void )
{
   if( A < B ) B = B - A;
   else A = A - B;
} /* reduce */

void main( void )
{
   printf( "L'MCD de: " );
   scanf( "%u%u", &A, &B );
   while( A!=0 && B!=0 ) reduce();
   printf( "... és %u\n", A + B );
} /* main */
```

Encara que el programa mostrat té un funcionament correcte, no és possible deduir directament què fa la funció `reduce()`, ni tampoc

ANOTACIONS

determinar de quines variables depèn ni quines afecta. Per tant, cal adoptar com a norma que cap funció no depengui de variables globals o n'afecti. Per això, en C, tot el codi es distribueix en funcions i es dedueix fàcilment que no hi ha d'haver cap variable global.

Així doncs, totes les variables són d'àmbit local (paràmetres formals i variables locals). En altres paraules, es declaren en l'entorn local d'una funció i només poden ser utilitzades les instruccions dins d'aquesta.

Les variables locals es creen en el moment en què s'activa la funció corresponent, és a dir, després d'executar la instrucció de crida de la funció. Per aquest motiu, tenen una classe d'emmagatzematge anomenada *automàtica*, ja que es creen i es destrueixen de manera automàtica en el procediment de crida a funció. Aquesta classe d'emmagatzematge es pot fer explícita mitjançant la paraula clau `auto`:

```
int una_funcio_qualsevol( int a, int b )
{
   /* ... */
   auto int variable_local;
   /* resta de la funció */
} /* una_funcio_qualsevol */
```

De vegades és interessant que la variable local s'emmagatzemi temporalment en un dels registres del processador per a evitar haver-la d'actualitzar contínuament en la memòria principal i accelerar, amb això, l'execució de les instruccions involucrades (normalment, les iteratives). En aquests casos es pot aconsellar al compilador que generi codi màquina perquè es faci així; és a dir, perquè l'emmagatzematge d'una variable local es dugui a terme en un dels registres del processador. De totes maneres, molts compiladors són capaços de dur a terme aquestes optimitzacions de manera autònoma.

Aquesta classe d'emmagatzematge s'indica amb la paraula clau `register`:

```
int una_funcio_qualsevol( int a, int b )
{
   /* ... */
```

ANOTACIONS

```
   register int comptador;
   /* resta de la funció */
} /* una_funcio_qualsevol */
```

Per a aconseguir l'efecte contrari, es pot indicar que una variable local resideixi sempre en memòria mitjançant la indicació `volatile` com a classe d'emmagatzematge. Això només és convenient quan la variable es pot modificar de manera aliena al programa.

```
int una_funcio_qualsevol( int a, int b )
{
   /* ... */
   volatile float temperatura;
   /* resta de la funció */
} /* una_funcio_qualsevol */
```

En els casos anteriors, es tractava de variables automàtiques. Tanmateix, de vegades resulta interessant que una funció pugui mantenir la informació continguda en alguna variable local entre diferents crides. Això és, permetre que l'algoritme corresponent "recordi" alguna cosa del seu estat passat. Per a aconseguir-ho, cal indicar que la variable té una classe d'emmagatzematge `static`; és a dir, que es troba estàtica o inamovible en memòria:

```
int una_funcio_qualsevol ( int a, int b )
{
   /* ... */
   static unsigned nombre_crides = 0;
   nombre_crides = nombre_crides + 1;
   /* resta de la funció */
} /* una_funcio_qualsevol */
```

En el cas anterior és molt important inicialitzar les variables en la declaració; altrament, no es podria saber el contingut inicial, previ a qualsevol crida a la funció.

Com a nota final, us indiquem que les classes d'emmagatzematge s'utilitzen molt rarament en la programa-

ANOTACIONS

ció en C. De fet, a excepció de `static`, les altres pràcticament no tenen efecte en un compilador actual.

2.12.3. Paràmetres per valor i per referència

El pas de paràmetres es refereix a l'acció de transformar els paràmetres formals a paràmetres reals; és a dir, d'assignar un contingut a les variables que representen als arguments:

```
tipus funcio_cridada(
   parametre_formal_1,
   parametre_formal_2,
   ...
);
funcio_de_crida(... )
{
   /* ... */
   funcio_cridada( parametre_real_1, parametre_real_2... );
   /* ... */
} /* funcio_de_crida */
```

En aquest sentit, hi ha dues possibilitats: que els arguments rebin el resultat de l'avaluació de l'expressió corresponent o que se substitueixin per la variable que es va indicar en el paràmetre real de la mateixa posició. El primer cas es tracta d'un **pas de paràmetres per valor**, mentre que el segon es tracta d'un **pas de variable** (qualsevol canvi en l'argument és un canvi en la variable que consta com a paràmetre real).

El pas per valor consisteix a assignar a la variable del paràmetre formal que correspon el valor resultant del paràmetre real en la mateixa posició. El pas de variable consisteix a substituir la variable del paràmetre real per la del paràmetre formal corresponent i, conseqüentment, poder-la utilitzar dins de la mateixa funció amb el nom del *paràmetre formal*.

ANOTACIONS

En C, el pas de paràmetres només s'efectua per valor; és a dir, s'avaluen tots els paràmetres en la crida i s'assigna el resultat al paràmetre formal corresponent en la funció.

Per a modificar alguna variable que es vulgui passar com a argument en la crida d'una funció, és necessari passar l'adreça de memòria en què es troba. Per a això s'ha d'utilitzar l'operador d'obtenció d'adreça (&), que dóna com a resultat l'adreça de memòria en què es troba el seu argument (variable, camp de tuple o element de matriu, entre altres). Aquest és el mecanisme que s'empra perquè la funció `scanf` dipositi en les variables que s'hi passen com a argument els valors que llegeix.

D'altra banda, en les funcions cridades, els paràmetres formals que reben una referència d'una variable en lloc d'un valor s'han de declarar de manera especial, anteposant al seu nom un asterisc. L'asterisc, en aquest context, es pot llegir com el "contingut la posició inicial del qual es troba en la variable corresponent". Per tant, en una funció com la mostrada a continuació, es llegiria "el contingut la posició inicial del qual es troba en el paràmetre formal `numerador`" és de tipus enter. De la mateixa manera es llegiria per al denominador:

```
void simplifica( int *numerador, int *denominador )
{
   int mcd;
   mcd = maxim_comu_divisor( *numerador, *denominador );
   *numerador = *numerador / mcd;
   *denominador = *denominador / mcd;
} /* simplifica */
/* ... */
   simplifica( &a, &b );
/* ... */
```

Encara que s'hi insistirà més endavant, cal tenir present que l'asterisc, en la part del codi, s'hauria de llegir com "el contingut de la variable que

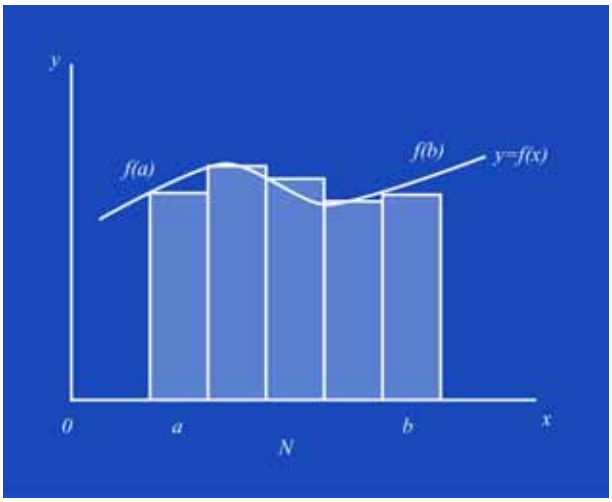
ANOTACIONS

està emmagatzemada en la posició de memòria de l'argument corresponent". Per tant, haurem d'utilitzar `*paràmetre_formal` cada vegada que es vulgui utilitzar la variable passada per referència.

2.12.4. Exemple

El programa següent calcula numèricament la integral d'una funció en un interval concret segons la regla de Simpson. Bàsicament, el mètode consisteix a dividir l'interval d'integració en un nombre determinat de segments de la mateixa longitud que constitueixen la base d'uns rectangles l'altura dels quals és determinada pel valor de la funció que s'ha d'integrar al punt inicial del segment. La suma de les àrees d'aquests rectangles donarà la superfície aproximada definida per la funció, l'eix de les X i les rectes perpendiculars a aquest que passen pels punts inicial i final del segment d'integració:

Figura 3.




```
/* Fitxer: simpson.c */

#include <stdio.h>
#include <math.h>

double f( double x )
{
   return 1.0 / (1.0 + x*x);
} /* f */
```

```
double integral_f( double a, double b, int n )
{
   double result;
   double x, dx;
   int i;

   result = 0.0;
   if( (a < b) && (n > 0) ) {
     x = a;
     dx = (b-a)/n;
     for( i = 0; i < n; i = i + 1 ) {
       result = result + f(x)* dx;
       x = x + dx;
     } /* for */
   } /* if */
   return result;
} /* integral_f */

void main( void )
{
   double a, b;
   int n;

   printf( "Integració numèrica de f(x).\n" );
   printf( "Punt inicial de l'interval, a = ? " );
   scanf( "%lf", &a );
   printf( "Punt final de l'interval, b = ? " );
   scanf( "%lf", &b );
   printf( "Nombre de divisions, n = ? " );
   scanf( "%d", &n );
   printf(
     "Resultat, integral(f)[%g,%g] = %g\n",
     a, b, integral_f( a, b, n )
   ); /* printf */
} /* main */
```

2.13. Macros del preprocessador de C

El preprocessador no solament fa substitucions de símbols simples com les que hem vist. També pot fer substitucions amb paràmetres.

ANOTACIONS

Les definicions de substitucions de símbols amb paràmetres s'anomenen *macros*:

```
#define simbol expressio_constant
#define macro( arguments ) expressio_const_amb_arguments
```

L'ús de les macros pot ajudar a l'aclariment de petites parts del codi mitjançant l'ús d'una sintaxi similar a la de les crides a les funcions.

D'aquesta manera, determinades operacions simples es poden beneficiar d'un nom significatiu en lloc d'utilitzar unes construccions en C que podrien dificultar la comprensió de la seva intencionalitat.

Exemple

```
#define absolut( x ) ( x < 0 ? -x : x )
#define arrodoneix( x ) ( (int) ( x + 0.5) )
#define trunca( x ) ( (int) x )
```

Cal tenir present que el nom de la macro i el parèntesi esquerre no poden anar separats i que la continuació de la línia, si l'ordre és massa llarga, es fa col·locant una barra invertida just abans del caràcter de salt de línia.

D'altra banda, cal advertir que les macros fan una substitució de cada nom de paràmetre aparegut en la definició per la part del codi font que s'indiqui com a argument. Així doncs:

```
absolut( 2*enter + 1 )
```

se substituiria per:

```
( 2*enter + 1 < 0 ? -2*enter + 1 : 2*enter + 1 )
```

aquesta expressió no seria correcta en el cas que fos negatiu.

ANOTACIONS

Nota

En aquest cas, seria possible evitar l'error si en la definició s'haguessin posat parèntesis al voltant de l'argument.

2.14. Resum

L'organització del codi font és essencial per a confeccionar programes llegibles que resultin fàcils de mantenir i d'actualitzar. Això és especialment cert per als programes de codi obert, és a dir, per al programari lliure.

En aquest capítol s'han repassat els aspectes fonamentals que intervenen en un codi organitzat. En essència, l'organització correcta del codi font d'un programa depèn tant de les instruccions com de les dades. Per aquest motiu, no solament s'ha tractat de com s'organitza el programa sinó que, a més a més, s'ha vist com s'empren estructures de dades.

L'organització correcta del programa comença perquè aquest tingui un flux d'execució clar de les seves instruccions. Com que el flux d'instruccions més simple és aquell en què s'executen de manera seqüencial segons apareixen en el codi, és fonamental que les instruccions de control de flux tinguin un únic punt d'entrada i un únic punt de sortida. En aquest principi es basa el mètode de la programació estructurada. En aquest mètode, només hi ha dos tipus d'instruccions de control de flux: les alternatives i les iteratives.

Les instruccions iteratives impliquen un altre repte en la determinació del flux de control, ja que és necessari determinar que la condició per la qual es deté la iteració es compleix alguna vegada. Per aquest motiu, s'han repassat els esquemes algorítmics per al tractament de seqüències de dades i s'han vist petits programes que, a més de servir d'exemple de programació estructurada, són útils per a fer operacions de filtre de dades en canonades (processos encadenats).

Per a organitzar correctament el codi i fer possible el tractament d'informació complexa, s'ha de recórrer a l'estructuració de les dades.

ANOTACIONS

En aquest aspecte, cal tenir present que el programa ha de reflectir les operacions que es fan en la informació i no tant el que representa dur a terme les dades elementals que la componen. Per aquest motiu, no solament s'ha explicat com es declaren i s'utilitzen dades estructurades, tant si són de tipus homogeni com heterogeni, sinó que també s'ha detallat com es defineixen nous tipus de dades a partir dels tipus de dades bàsiques i estructurats. Aquests nous tipus de dades s'anomenen *tipus abstractes de dades* ja que són transparents per al llenguatge de programació.

En parlar de les dades també s'ha tractat dels fitxers de flux de bytes. Aquestes estructures de dades homogènies es caracteritzen per tenir un nombre indefinit d'elements, per residir en memòria secundària –és a dir, en algun suport d'informació extern– i, finalment, per requerir funcions específiques per a accedir a les seves dades. Així doncs, s'han comentat les funcions estàndard en C per a operar amb aquest tipus de fitxers. Bàsicament, els programes que els utilitzen implementen esquemes algorítmics de recorregut o de cerca en què s'inclou una inicialització específica per a obrir els fitxers, una comprovació de final de fitxer per a la condició d'iteració, operacions de lectura i escriptura per al tractament de la seqüència de dades i, per acabar, un final que consisteix, entre altres coses, a tancar els fitxers emprats.

L'última part s'ha dedicat a la programació modular, que consisteix a agrupar les seqüències d'instruccions en subprogrames que facin una funció concreta i susceptible de ser utilitzada més d'una vegada en el mateix programa o en d'altres. Així doncs, se substitueix en el flux d'execució tot el subprograma per una instrucció que s'ocuparà d'executar el subprograma corresponent. Aquests subprogrames s'anomenen *funcions* en C i la instrucció que s'ocupa que les executi, *instrucció de crida*. S'ha vist com es duu a terme una crida a una funció i que, en aquest aspecte, el més important és el pas de paràmetres.

El pas de paràmetres consisteix a transmetre a una funció el conjunt de dades amb què haurà de fer la seva tasca. Com que la funció pot necessitar tornar resultats que no es puguin emmagatzemar en una variable simple, alguns d'aquests paràmetres s'utilitzen per a passar referències a variables que també podran contenir valors de retorn. Així doncs, també s'ha analitzat tota la problemàtica relacionada amb el pas de paràmetres per valor i per referència.

ANOTACIONS

2.15. Exercicis d'autoavaluació

1) Feu un programa per determinar el nombre de dígits necessaris per a representar un nombre enter donat. L'algoritme consisteix a fer divisions enteres per 10 del nombre fins que el resultat sigui un valor inferior a 10.

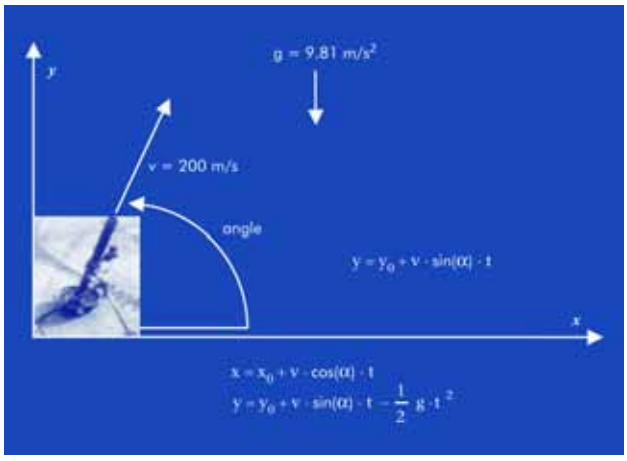
2) Feu un programa que determini a cada moment la posició d'un projectil llançat des d'un morter. S'haurà de mostrar l'altura i la distància a intervals regulars de temps fins que arribi al terra. Per a això, se suposarà que el terra és pla i que es proporcionen, com a dades d'entrada, l'angle del canó i l'interval de temps en què es mostraran les dades de sortida. Se suposa que la velocitat de sortida dels projectils és de 200 m/s.

Nota

Per a més detalls, es pot tenir en compte que el tub del morter és d'1 m de longitud i que els angles de tir varien entre 45 i 85 graus.

En l'esquema següent es resumeixen les diferents fórmules que són necessàries per a resoldre el problema:

Figura 4.



on x_0 i y_0 és la posició inicial (es pot considerar 0 per als dos), i α és l'angle en radiants (Π radiants = 180°).

3) Es vol calcular el capital final acumulat d'un pla de pensions sabent el capital inicial, l'edat de l'assegurat (se suposa que es jubi-

ANOTACIONS

larà als 65 anys) i les aportacions i els percentatges d'interès rendits de cada any. (Se suposa que les aportacions són de caràcter anual.)

4) Programeu un filtre que calculi la mitjana, el màxim i el mínim d'una sèrie de nombres reals d'entrada.

5) Implementeu els filtres de l'últim exemple de l'apartat 2.4.2 ("Filtres i canonades"), és a dir: calculeu els imports d'una seqüència de dades { codi d'article, preu, quantitat } que generi una altra seqüència de dades { codi d'article, import } i feu, posteriorment, la suma dels imports d'aquesta segona seqüència de dades.

6) Feu un programa que calculi la desviació típica que tenen les dades d'ocupació d'un aparcament públic al llarg de les vint-i-quatre hores d'un dia. Hi haurà, per tant, vint-i-quatre dades d'entrada.

   Aquestes dades es refereixen al percentatge d'ocupació (nombre de places ocupades amb relació al nombre total de places) calculat al final de cada hora. També haureu d'indicar les hores del dia que tinguin un percentatge d'ocupació inferior a la mitjana menys dues vegades la desviació típica, i les que el tinguin superior a la mitjana més dues vegades aquesta desviació.

   Nota

   La desviació típica es calcula com l'arrel quadrada de la suma dels quadrats de les diferències entre les dades i la mitjana, dividida pel nombre de mostres.

7) Esbrineu si la lletra d'un NIF concret és o no correcta. El procediment del seu càlcul consisteix a realitzar el mòdul 23 del nombre corresponent. El resultat dóna una posició en una seqüència de lletres (TRWAGMYFPDXBNJZSQVHLCKE). La lletra situada en aquesta posició serà la lletra del NIF.

   Nota

   Per a poder efectuar la comparació entre lletres, és convenient convertir la que proporcioni l'usuari en majúscula. Per a això s'ha d'emprar `toupper()`, la declaració del qual és a `ctype.h` i torna el caràcter

ANOTACIONS

corresponent a la lletra majúscula de la que ha rebut com a argument. Si no és un caràcter alfabètic o es tracta d'una lletra ja majúscula, torna el mateix caràcter.

8) Feu un programa que calculi el mínim nombre de monedes necessari per a tornar el canvi sabent l'import total que s'ha de cobrar i la quantitat rebuda com a pagament. La moneda d'import màxim és la de 2 euros i la més petita, d'1 cèntim.

Nota

És convenient tenir un vector amb els valors de les monedes ordenats per valor.

9) Resumiu l'activitat que hi ha en un terminal de venda per articles. El programa ha de mostrar, per a cada codi d'article, el nombre d'unitats venudes. Per a això, disposarà d'un fitxer generat pel terminal que consta de parells de nombres enters: el primer indica el codi de l'article i el segon, la quantitat venuda. En cas de devolució, la quantitat apareixerà com un valor negatiu. Se sap, a més a més, que no hi haurà mai més de cent codis d'articles diferents.

Nota

És convenient disposar d'un vector de 100 tuples per a emmagatzemar la informació del seu codi i les unitats venudes corresponents. Com que no se sap quants tuples seran necessaris, tingueu en compte que s'haurà de tenir una variable que indiqui els que s'hagin emmagatzemat en el vector (de 0 al nombre de codis diferents –1).

10) Reprogrameu l'exercici anterior de manera que les operacions que afectin el vector de dades es duguin a terme en el cos de funcions específiques.

Nota

Definiu un tipus de dada nova que contingui la informació dels productes. Se suggereix, per exemple, el que es mostra a continuació.

ANOTACIONS

```
typedef struct productes_s {
 unsigned int n; /* Nombre de productes. */
 venda_t producte[MAX_PRODUCTES];
} productes_t;
```

Recordeu que haurà de passar la variable d'aquest tipus per referència.

11) Busqueu una paraula en un fitxer de text. Per a això, feu un programa que demani tant el text de la paraula com el nom del fitxer. El resultat haurà de ser una llista de totes les línies en què hi hagi aquesta paraula.

Se suposarà que una paraula és una seqüència de caràcters alfanumèrics. És convenient emprar la macro `isalnum()`, que es troba declarada a `ctype.h`, per a determinar si un caràcter és o no alfanumèric.

Nota

En la solució proposada, s'empren les funcions que es declaren a continuació.

```
#define LONG_PARAULA 81
typedef char paraula_t[LONG_PARAULA];
bool paraules_iguals( paraula_t p1, paraula_t p2 );
unsigned int llegeix_paraula( paraula_t paraula, FILE *entrada );
void primera_paraula( paraula_t paraula, char *frase );
```

2.15.1. Solucionari

1)

```
/* ------------------------------------------------------*/
/* Fitxer: ndigits.                                      */
/* ------------------------------------------------------*/

#include <stdio.h>

main()
{
 unsigned int nombre;
 unsigned int digits;
```

ANOTACIONS

```
   printf( "El nombre de dígits per a representar: " );
   scanf( "%u", &nombre );
   digits = 1;
   while( nombre > 10 ) {
    nombre = nombre / 10;
    digits = digits + 1;
   } /* while */
   printf( "... és %u.\n", digits );
} /* main */
```

2)

```
/* ------------------------------------------------- */
/* Fitxer: morter.c                                  */
/* ------------------------------------------------- */

#include <stdio.h>
#include < math.h >

#define V_INICIAL 200.00        /* m/s               */
#define L_TUB      1.0          /* m                 */
#define G          9.81         /* m/(s*s)           */
#define PI         3.14159265

main()
{
   doble angle, inc_temps, t;
   doble v_x, v_y; /* Velocitats horitzontal i vertical. */
   doble x0, x, y0, i;

   printf( "Tir amb morter.\n " );
   printf( "Angle de tir [graus sexagesimals] =? " );
   scanf( "%lf", &angle );
   angle = angle * PI / 180.0;
   printf( "Cicle de mostra [segons] =? " );
   scanf( "%lf", &inc_temps );
   x0 = L_TUB * cos( angle );
   y0 = L_TUB * sin( angle );
   t = 0.0;
   v_x = V_INICIAL * cos( angle );
   v_y = V_INICIAL * sin( angle );
```

ANOTACIONS

```
   do {
    x = x0 + v_x * t;
    y = y0 + v_y * t - 0.5 * G * t * t;
    printf( "%6.2lf s: ( %6.2lf, %6.2lf )\n", t, x, y );
    t = t + inc_temps;
   } while ( y > 0.0 );
} /* main */
```

3)

```
/* ------------------------------------------------- */
/* Fitxer: pensions.c                                */
/* ------------------------------------------------- */
#include <stdio.h>

#define EDAT_JUBILACIO 65

main()
{
   unsigned int edat;
   float capital, interes, aportacio;

   printf( "Pla de pensions.\n " );
   printf( "Edat =? " );
   scanf( "%u", &edat );
   printf( "Aportació inicial =? " );
   scanf( "%f", &capital );
   while( edat < EDAT_JUBILACIO ) {
    printf( "Interès rendit [en %%] =? " );
    scanf( "%f", &interes );
    capital = capital*( 1.0 + interes/100.0 );
    printf( "Nova aportació =? " );
    scanf( "%f", &aportacio );
    capital = capital + aportacio;
    edat = edat + 1;
    printf( "Capital acumulat a %u anys: %.2f\n",
    edat, capital
    ); /* printf */
   } /* while */
} /* main */
```

4)

```
/* ------------------------------------------------- */
/* Fitxer: estadistica.c                             */
/* ------------------------------------------------- */
```

ANOTACIONS

```
#include <stdio.h>
main()
{
   double       suma, minim, maxim;
   double       nombre;
   unsigned int quantitat, llegit_ok;

   printf( "Minim, mitjana i maxim.\n " );
   llegit_ok = scanf( "%lf", &nombre );
   if( llegit_ok == 1 ) {
    quantitat = 1;
    suma = nombre;
    minim = nombre;
    maxim = nombre;
    llegit_ok = scanf( "%lf", &nombre );
    while( llegit_ok == 1 ) {
     suma = suma + nombre;
     if( nombre > maxim ) maxim = nombre;
     if( nombre < minim ) minim = nombre;
     quantitat = quantitat + 1;
     llegit_ok = scanf( "%lf", &nombre );
    } /* while */
    printf( "Minim = %g\n", minim );
    printf( "Mitjana = %g\n", suma / (double) quantitat );
    printf( "Maxim = %g\n", maxim );
   } else {
    printf( "Entrada buida.\n" );
   } /* if */
} /* main */
```

5)

```
/* ------------------------------------------------- */
/* Fitxer: calc_imports.c                            */
/* ------------------------------------------------- */
#include <stdio.h>
main()
{
   unsigned int llegits_ok, codi;
   int          quantitat;
   float        preu, import;
```

ANOTACIONS

```
   llegits_ok = scanf( "%u%f%i",
   &codi, &preu, &quantitat
   ); /* scanf */
   while( llegits_ok == 3 ) {
    import = (float) quantitat * preu;
    printf( "%u %.2f\n", codi, import );
    llegits_ok = scanf( "%u%f%i",
     &codi, &preu, &quantitat
    ); /* scanf */
   } /* while */
} /* main */

/* ------------------------------------------------------*/
/* Fitxer: totalitza.c*/
/* ------------------------------------------------------*/
#include <stdio.h>
main()
{
   unsigned int llegits_ok, codi;
   int          quantitat;
   flota        import;
   double       total = 0.0;

   llegits_ok = scanf( "%u%f", &codi, &import );
   while( llegits_ok == 2 ) {
    total = total + import;
    llegits_ok = scanf( "%u%f", &codi, &import );
   } /* while */
   printf( "%.2lf\n", total );
} /* main */
```

6)

```
/* ------------------------------------------------  */
/* Fitxer: ocupa_pk.c                                */
/* ------------------------------------------------  */

#include <stdio.h>
#include < math.h >

main()
{
   unsigned int hora;
```

ANOTACIONS

```
float        percentatge;
float        ratio_acum[24];
double       mitjana, desv, dif;

printf( "Estadística d'ocupació diària:\n" );
/* Lectura de ratios d'ocupacio per hores:            */
for( hora = 0; hora < 24; hora = hora + 1 ) {
 printf(
  "Percentatge d'ocupació a les %02u hores =? ",
  hora
 ); /* printf */
 scanf( "%f", &percentatge );
 ratio_acum[hora] = percentatge;
} /* for */
/* Calcul de la mitjana:                              */
 mitjana = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
 mitjana = mitjana + ratio_acum[hora];
} /* for */
mitjana = mitjana / 24.0;
/* Calcul de la desviació típica:                     */
desv = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
 dif = ratio_acum[ hora ] - mitjana;
 desv = desv + dif * dif;
} /* for */
desv = sqrt( desv ) / 24.0;
/* Impressio dels resultats:                          */
printf( "Mitjana d'ocupació al dia: %.2lf\n", mitjana );
printf( "Desviació típica: %.2lf\n", desv );
printf( "Hores amb percentatge molt baix: " desv );
 dif = mitjana - 2*desv;
for( hora = 0; hora < 24; hora = hora + 1 ) {
 if( ratio_acum[ hora ] < dif ) printf( " %u " hora );
} /* for */
printf( "\nHores amb percentatge molt alt: " desv );
 dif = mitjana + 2*desv;
for( hora = 0; hora < 24; hora = hora + 1 ) {
 if( ratio_acum[ hora ] > dif ) printf( " %u " hora );
} /* for */
printf( "\nFi.\n" );
```

ANOTACIONS

```
} /* main */
```

7)

```
/* ------------------------------------------------- */
/* Fitxer: valida_nif.c                              */
/* ------------------------------------------------- */

#include <stdio.h>
#include <ctype.h>
main()
{
   char         NIF[ BUFSIZ ];
   unsigned int DNI;
   char         lletra;
   char         codi[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;

   printf( "Digues-me el NIF (DNI-lletra): " );
   gets( NIF );
   sscanf( NIF, "%u", &DNI );
   DNI = DNI % 23;
   lletra = NIF[ strlen(NIF)-1 ];
   lletra = toupper( lletra );
   if( lletra == codi[ DNI ] ) {
    printf( "NIF valid.\n" );
   } else {
    printf( " Lletra %c no vàlida!\n", lletra );
   } /* if */
} /* main */
```

8)

```
/* ------------------------------------------------- */
/* Fitxer: canvi.c                                   */
/* ------------------------------------------------- */

#include <stdio.h>
#define NRE_MONEDES 8

main()
{
   double       import, pagat;
   int          canvi_cents;
```

ANOTACIONS

```
   int          i, nmonedes;
   int          cents[NRE_MONEDES] = { 1, 2, 5, 10,
                20, 50, 100, 200 };

   printf( "Import : " );
   scanf( "%lf", &import );
   printf( "Pagat : " );
   scanf( "%lf", &pagat );
   canvi_cents = (int) 100.00 * ( pagat - import );
   if( canvi_cents < 0 ) {
    printf( "PAGAMENT INSUFICIENT\n");
   } else {
    printf( "Per tornar : %.2f\n",
      (float) canvi_cents / 100.0
    ); /* printf */
    i = NRE_MONEDES - 1;
    while( canvi_cents > 0 ) {
     nmonedes = canvi_cents / cents[i] ;
     if( nmonedes >0 ) {
      canvi_cents = canvi_cents - cents[i]*nmonedes;
      printf( "%u monedes de %.2f euros.\n",
       nmonedes,
       (float) cents[i] / 100.0
      ); /* printf */
     } /* if */
     i = i - 1;
    } /* while */
   } /* if */
} /* main */
```

9)

```
/* ------------------------------------------------------ */
/* Fitxer: resum_tpv.c                                    */
/* ------------------------------------------------------ */

#include <stdio.h>

typedef struct venda_s {
   unsigned int codi;
   int quantitat;
} venda_t;
```

ANOTACIONS

```
#define MAX_PRODUCTES 100

main()
{
   FILE          *entrada;
   char          nom[BUFSIZ];
   unsigned int  llegits_ok;
   int           num_prod, i;
   venda_t       venda, producte[MAX_PRODUCTES];

   printf( "Resum del dia en TPV.\n" );
   printf( "Fitxer: ");
   gets( nom );
   entrada = fopen( nom, "rt" );
   if( entrada != NULL ) {
    num_prod = 0;
    llegits_ok = fscanf( entrada, "%u%i",
     &(venda.codi), &(venda.quantitat)
    ); /* fscanf */
    while( llegits_ok == 2 ) {
     /* Cerca del codi en la taula:                          */
     i = 0;
     while( ( i < num_prod ) &&
            ( producte[i].codi != venda.codi )
     ) {
      i = i + 1;
     } /* while */
     if( i < num_prod ) { /* Codi trobat:                    */
      producte[i].quantitat =  producte[i].quantitat +
                               venda.quantitat;
     } else { /* Codi no trobat => nou producte:            */
      producte[num_prod].codi = venda.codi;
      producte[num_prod].quantitat = venda.quantitat;
      num_prod = num_prod + 1;
     } /* if */
     /* Lectura de venda següent:                            */
     llegits_ok = fscanf( entrada, "%u%i",
      &(venda.codi), &(venda.quantitat)
     ); /* fscanf */
    } /* while */
```

```
    printf( "Resum:\n" );
    for( i=0; i<num_prod; i = i + 1 ) {
     printf( "%u : %i\n",
       producte[i].codi, producte[i].quantitat
     ); /* printf */
    } /* for */
   } else {
    printf( "No puc obrir %s!\n", nom );
   } /* if */
} /* main */
```

10)

```
/* ... */

int busca( unsigned int codi, productes_t *pref )
{
   int i;

   i = 0;
   while( (i < (*pref).n ) &&
          ( (*pref).producte[i].codi != codi )
   ) {
      i = i + 1;
   } /* while */
   if( i == (*pref).n ) i = -1;
   return i;
} /* cerca */

void afegeix( venda_t venda, productes_t *pref )
{
   unsigned int n;

   n = (*pref).n;
   if( n < MAX_PRODUCTES ) {
     (*pref).producte[n].codi = venda.codi;
     (*pref).producte[n].quantitat = venda.quantitat;
     (*pref).n = n + 1;
   } /* if */
} /* afegeix */
```

ANOTACIONS

```
void actualitza( venda_t venda, unsigned int posicio,
productes_t *pref )
{
   (*pref).producte[posicio].quantitat =
    (*pref).producte[posicio].quantitat + venda.quantitat;
} /* actualitza */

void mostra( productes_t *pref )
{
   int i;

   for( i=0; i<(*pref).n; i = i +1 ) {
    printf( "%u : %i\n",
      (*pref).producte[i].codi,
      (*pref).producte[i].quantitat
    ); /* printf */
   } /* for */
} /* mostra */

void main( void )
{
   FILE         *entrada;
   char         nom[BUFSIZ];
   unsigned int llegits_ok;
   int          i;
   venda_t      venda;
   productes_t  productes;

   printf( "Resum del dia en TPV.\n" );
   printf( "Fitxer: ");
   gets( nom );
   entrada = fopen( nom, "rt" );
   if( entrada != NULL ) {
    productes.n = 0;
    llegits_ok = fscanf( entrada, "%u%i",
      &(venda.codi), &(venda.quantitat)
    ); /* scanf */
    while( llegits_ok == 2 ) {
      i = busca( venda.codi, &productes );
      if( i < 0 ) afegeix( venda, &productes );
      else actualitza( venda, i, &productes );
      llegits_ok = fscanf( entrada, "%u%i",
        &(venda.codi), &(venda.quantitat)
```

ANOTACIONS

```
      ); /* scanf */
     } /* while */
     printf( "Resum:\n" );
     mostra( &productes );
   } else {
     printf( "No puc obrir %s!\n", nom );
   } /* if */
} /* main */
```

11)

```
/* ------------------------------------------------------ */
/* Fitxer: busca.c                                        */
/* ------------------------------------------------------ */

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

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;

#define LONG_PARAULA 81
typedef char paraula_t[LONG_PARAULA];

bool paraules_iguals( paraula_t p1, paraula_t p2 )
{
   int i = 0;

   while( (p1[i]!='\0') && (p2[i]!='\0') && (p1[i]==p2[i])) {
     i = i +1;
   } /* while */
   return p1[i]==p2[i];
} /* paraules_iguals */

unsigned int llegeix_paraula( paraula_t p, FILE *entrada )
{
   unsigned int i, nlin;
   bool         terme;
   char         caracter;

   i = 0;
   nlin = 0;
   terme = FALSE;
```

ANOTACIONS

```
   caracter = fgetc( entrada );
   while( !feof( entrada ) && !terme ) {
    if( caracter == '\n' ) nlin = nlin + 1;
    if( isalnum( caracter ) ) {
     p[i] = caracter;
     i = i + 1;
     caracter = fgetc( entrada );
    } else {
     if( i > 0 ) {
      terme = TRUE;
     } else {
      caracter = fgetc( entrada );
     } /* if */
    } /* if */
   } /* while */
   p[i] = '\0';
   return nlin;
} /* llegeix_paraula */

void primera_paraula( paraula_t paraula, char *frase )
{
   int i, j;

   i = 0;
   while( frase[i]!='\0' && isspace( frase[i] ) ) {
    i = i + 1;
   } /* while */
   j = 0;
   while( frase[i]!='\0' && !isspace( frase[i] ) ) {
    paraula[j] = frase[i];
    i = i + 1;
    j = j + 1;
   } /* while */
   paraula[j] = '\0';
} /* primera_paraula */

void main( void )
{
   FILE         *entrada;
```

ANOTACIONS

```
   char          nom[BUFSIZ];
   palabra_t     paraula, paraula2;
   unsigned int numlin;

   printf( "Busca paraules.\n" );
   printf( "Fitxer: ");
   gets( nom );
   entrada = fopen( nom, "rt" );
   if( entrada != NULL ) {
    printf( "Paraula: " );
    gets( nom );
    primera_paraula( paraula, nom );
    printf( "Buscant %s en fitxer...\n", paraula );
    numlin = 1;
    while( !feof( entrada ) ) {
     numlin = numlin + llegeix_paraula( paraula2, entrada );
     if( paraules_iguals( paraula, paraula2 ) ) {
      printf( "... línia %lu\n", numlin );
     } /* if */
    } /* while */
   } else {
    printf( "No puc obrir %s!\n", nom );
   } /* if */
} /* main */
```

ANOTACIONS

3. Programació avançada en C. Desenvolupament eficient d'aplicacions

3.1. Introducció

La programació d'una aplicació informàtica sol donar com a resultat un codi font de mida considerable. Fins i tot aplicant tècniques de programació modular i ajudant-se de funcions estàndard de biblioteca, resulta complex organitzar eficaçment el codi. Molt més si es té en compte que, habitualment, es tracta d'un codi desenvolupat per un equip de programadors.

D'altra banda, cal tenir present que molta de la informació amb què es treballa no té una mida predefinida ni, la majoria de les vegades, es presenta de la manera més adequada per a ser processada. Això comporta haver de reestructurar les dades de manera que els algoritmes que els tracten puguin ser més eficients.

Com a últim element que s'ha de considerar, però no per això menys important, és que la majoria d'aplicacions estan constituïdes per més d'un programa. Per tant, resulta convenient organitzar-los aprofitant les facilitats que per a això ens ofereix tant el conjunt d'eines de desenvolupament de programari com el sistema operatiu en què s'executarà.

En aquest capítol es tractarà de diversos aspectes que alleugereixen els problemes abans esmentats. Així doncs, des del punt de vista d'un programa en el context d'una aplicació que el contingui (o des del punt de vista que sigui l'únic), és important l'adaptació a la mida real de les dades que s'han de processar i la seva disposició en estructures dinàmiques, l'organització del codi perquè reflecteixi l'algoritme que implementa i, finalment, tenir el suport del sistema operatiu per a la coordinació amb altres programes dins i fora de la mateixa aplicació i, també, per a la interacció amb l'usuari.

La representació d'informació en estructures dinàmiques de dades permet ajustar les necessitats de memòria del programa al mínim re-

ANOTACIONS

querit per a la resolució del problema i, d'altra banda, representar internament la informació de manera que el seu processament sigui més eficient. Una estructura dinàmica de dades no és més que una col·lecció de dades la relació de les quals no està establerta *a priori* i es pot modificar durant l'execució del programa. Això, per exemple, no és factible mitjançant un vector, perquè les dades que contenen estan relacionades per la seva posició dins d'aquest i, a més, la seva mida ha d'estar predefinida en el programa.

En la primera part es tractarà de les variables dinàmiques i del seu ús com a contenidors de dades que responen a les exigències d'adaptabilitat a la informació que s'ha de representar i acomodament respecte de l'algoritme que hi ha de tractar.

El codi font d'una aplicació, tant si és un conjunt de programes com un de sol, s'ha d'organitzar de manera que es mantinguin les característiques d'un bon codi (intel·ligible, de manteniment fàcil i cost d'execució òptim). Per a això, no solament cal emprar una programació estructurada i modular, sinó que cal distribuir el codi en diversos fitxers de manera que sigui més manejable i, d'altra banda, es mantingui la seva llegibilitat.

En l'apartat dedicat al disseny descendent de programes es tractarà, precisament, dels aspectes que afecten la programació més enllà de la programació modular. Es tractarà d'aspectes relacionats amb la divisió del programa en termes algorítmics i de l'agrupació de conjunts de funcions fortament relacionades. Com que el codi font es reparteix en diversos fitxers, es tractarà també d'aquells aspectes relacionats amb la seva compilació i, en especial, de l'eina *make*.

Atesa la complexitat del codi font de qualsevol aplicació, resulta convenient utilitzar totes aquelles funcions estàndard que aporta el sistema operatiu. Amb això s'aconsegueix, a més de reduir la seva complexitat en deixar determinades tasques com a simples instruccions de crida, que sigui més independent de la màquina. Així doncs, en l'última part es tracta de la relació entre els programes i els sistemes operatius, de manera que sigui possible comunicar els uns amb els altres i, a més, els programes entre ells.

Nota

Sobre aquest tema es va veure un exemple en parlar de les canonades en la unitat anterior.

Per acabar, es tractarà, encara que breument, de com es distribueix l'execució d'un programa en diversos fluxos (o fils) d'execució. És a dir, com es fa una programació concurrent de manera que diverses tasques es puguin resoldre en un mateix interval de temps.

Aquest capítol pretén mostrar aquells aspectes de la programació més involucrats amb el nivell més alt d'abstracció dels algoritmes. Així doncs, en finalitzar el seu estudi, el lector hauria d'haver assolit els objectius següents:

1) Emprar adequadament variables dinàmiques en un programa.

2) Conèixer les estructures dinàmiques de dades i, en especial, les llistes i les seves aplicacions.

3) Entendre el principi del disseny descendent de programes.

4) Ser capaç de desenvolupar una biblioteca de funcions.

5) Tenir coneixements bàsics per a la relació del programa amb el sistema operatiu.

6) Assimilar el concepte de fil d'execució i els rudiments de la programació concurrent.

3.2. Les variables dinàmiques

La informació processada per un programa està formada per un conjunt de dades que, molt sovint, no té ni una mida fixa, no es coneix la seva mida màxima, les dades no es relacionen les unes amb les altres de la mateixa manera, etc.

Exemple

Un programa que faci anàlisis sintàctiques (que, d'altra banda, podria formar part d'una aplicació de tractament de textos) haurà de processar frases de mides diferents en nombre i categoria de paraules. A més a més, les paraules poden estar relacionades de maneres molt diferents. Per exemple, els adverbis ho estan amb

ANOTACIONS

els verbs i tots dos es diferencien de les que formen el sintagma nominal, entre d'altres.

Les relacions existents entre els elements que formen una informació determinada es poden representar mitjançant dades addicionals que les reflecteixin i ens permetin, una vegada calculades, fer un algoritme més eficient. Així doncs, en l'exemple anterior resultaria molt millor efectuar les anàlisis sintàctiques necessàries a partir de l'arbre sintàctic que no pas de la simple successió de paraules de la frase que s'ha de tractar.

La mida de la informació –és a dir, el nombre de dades que la formen– afecta significativament el rendiment d'un programa. Més encara, l'ús de variables estàtiques per al seu emmagatzematge implica o bé conèixer *a priori* la seva mida màxima, o bé limitar la capacitat de tractament a només una porció de la informació. A més a més, encara que sigui possible determinar la mida màxima, es pot produir un malbaratament de memòria innecessari quan la informació que s'ha de tractar ocupi molt menys.

Les **variables estàtiques** són les que es declaren en el programa de manera que disposen d'un espai reservat de memòria durant tota l'execució del programa. En C, només són realment estàtiques les variables globals.

Les **variables locals**, en canvi, són automàtiques perquè se'ls reserva espai solament durant l'execució d'una part del programa i després es destrueixen de manera automàtica. Tot i així, són variables de caràcter estàtic respecte de l'àmbit en què s'utilitzen, ja que té un espai de memòria reservat i limitat.

Les **variables dinàmiques**, tanmateix, es poden crear i destruir durant l'execució d'un programa i tenen un caràcter global; és a dir, són "visibles" des de qualsevol punt del programa. Com que és possible crear un nombre indeterminat d'aquestes variables, permeten ajustar-se a la mida requerida per a representar la informació d'un problema particular sense que malgasti cap espai de memòria.

ANOTACIONS

Per a poder crear les variables dinàmiques durant l'execució del programa, és necessari comptar amb operacions que ho permetin. D'altra banda, les variables dinàmiques manquen de nom i, per tant, la seva única identificació es fa mitjançant l'adreça de la primera posició de memòria en la qual resideixen.

És per això que és necessari tenir dades que puguin contenir referències a variables dinàmiques de manera que permetin utilitzar-les. Com que les seves referències són adreces de memòria, el tipus d'aquestes dades serà, precisament, el d'adreça de memòria o *apuntador*, ja que el seu valor és l'indicador d'on és la variable referenciada.

En els apartats següents es tractarà, en definitiva, de tot allò que incumbeix a les variables dinàmiques i a les estructures de dades que s'hi poden construir.

3.3. Els apuntadors

Els apuntadors són variables que contenen adreces de memòria d'altres variables; és a dir, referències a altres variables. Evidentment, el tipus de dades és el de posicions de memòria i, com a tal, és un tipus compatible amb enters. Tot i així, té les seves particularitats, que veurem més endavant.

La declaració d'un apuntador consisteix a declarar el tipus de dades de les variables de les quals contindrà adreces. Així doncs, s'utilitza l'operador d'indirecció (un asterisc) o, el que és el mateix, el que es pot llegir com a "contingut de la adreça". En els exemples següents es declaren diferents tipus d'apuntadors:

```
int         *ref_enter;
char        *cadena;
altre_t     *apuntador;
node_t      *ap_node;
```

- A `ref_enter` el contingut de l'adreça de memòria emmagatzemada és una dada de tipus enter.

ANOTACIONS

- A `cadena` el contingut de l'adreça de memòria emmagatzemada és un caràcter.
- A `apuntador` el contingut de l'adreça de memòria emmagatzemada és de tipus `altre_t`.
- A `ap_node`, el contingut de l'adreça de memòria emmagatzemada és de tipus `node_t`.

La referència a les variables sense l'operador d'indirecció és, simplement, l'adreça de memòria que contenen.

El tipus d'apuntador és determinat pel tipus de dada de la qual té l'adreça. Per aquest motiu, per exemple, es dirà que `ref_enter` és un apuntador de tipus enter, o bé, que es tracta d'un apuntador "a" un tipus de dades enter. També és possible declarar apuntadors d'apuntadors, etc.

En l'exemple següent es pot observar que, per a fer referència al valor apuntat per l'adreça continguda en un apuntador, cal emprar l'operador de "contingut de l'adreça de":

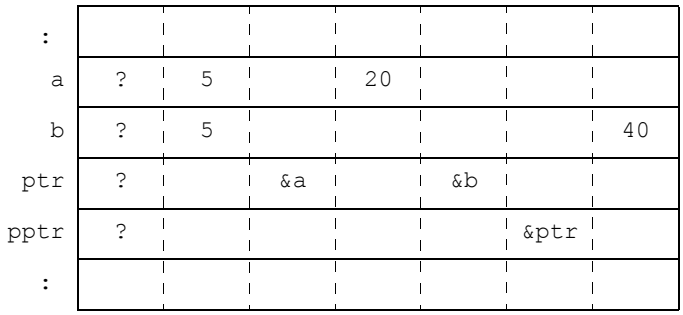
```
int  a, b;  /* Dues variables de tipus enter.          */
int  *ptr;  /* Un apuntador a enter.                   */
int  **pptr;/* Un apuntador a un apuntador d'enter.    */
/* ... */
a = b = 5;
ptr = &a;
*ptr = 20;  /* a == 20                                 */
ptr = &b;
pptr = &ptr;
**pptr = 40;/* b == 40                                 */
/* ... */
```

En la figura següent es pot apreciar com el programa anterior va modificant les variables a mesura que es va executant. Cada columna vertical representa les modificacions dutes a terme en l'entorn per

ANOTACIONS

una instrucció. Inicialment (la columna de més a l'esquerra), es desconeix el contingut de les variables:

Figura 5.



En el cas particular dels tuples, es pot recordar que l'accés també es fa mitjançant l'operador d'indirecció aplicat als apuntadors que contenen les seves adreces inicials. L'accés als seus camps no canvia:

```
struct alumne_s {
   cadena_t nom;
   unsigned short dni, nota;
} alumne;
struct alumne_s *ref_alumne;
/* ... */
alumne.nota = *ref_alumne.nota;
/* ... */
```

Perquè quedi clar l'accés a un camp d'un tuple l'adreça de la qual és en una variable, és preferible utilitzar el següent:

```
/* ... */
alumne.nota = (*ref_alumne).nota;
/* ... */
```

Si es vol emfasitzar la idea de l'apuntador, és possible utilitzar l'operador d'indirecció per a tuples que s'assembla a una fletxa que apunta al tuple corresponent:

```
/* ... */
alumne.nota = ref_alumne->nota;
/* ... */
```

ANOTACIONS

En els exemples anteriors, totes les variables eren de caràcter estàtic o automàtic, però el seu ús primordial és com a referència de les variables dinàmiques.

3.3.1. Relació entre apuntadors i vectors

Els vectors, en C, són entelèquies del programador, ja que es fa servir un operador (els claudàtors) per a calcular l'adreça inicial d'un element dins d'un vector. Per a això, cal tenir present que els noms amb què es declaren són, de fet, apuntadors a les primeres posicions dels primers elements de cada vector. Així doncs, les declaracions següents són pràcticament equivalents:

```
/* ... */
int vector_real[DIMENSIO];
int *vector_virtual;
/* ... */
```

En la primera, el vector té una dimensió determinada, mentre que en la segona, el `vector_virtual` és un apuntador a un enter; és a dir, una variable que conté l'adreça de dades de tipus enter. Tot i així, és possible emprar l'identificador de la primera com un apuntador a enter. De fet, conté l'adreça del primer enter del vector:

```
vector_real == &(vector_real[0])
```

És molt important no modificar el contingut de l'identificador, ja que es podria perdre la referència a tot el vector!

D'altra banda, els apuntadors es manegen amb una aritmètica especial: es permet la suma i la resta d'apuntadors d'un mateix tipus i enters. En l'últim cas, les sumes i restes amb enters són, en realitat, sumes i restes amb els múltiples dels enters, que es multipliquen per la mida en bytes d'allò al que apunten.

ANOTACIONS

Sumar o restar continguts d'apuntadors resulta una cosa poc habitual. El més freqüent és incrementar-los o fer decréixer-los perquè apuntin a algun element posterior o anterior, respectivament. En l'exemple següent es pot intuir el perquè d'aquesta aritmètica especial. Amb ella, s'allibera el programador d'haver de pensar quants bytes ocupa cada tipus de dada:

```
/* ... */
int vector[DIMENSIO], *ref_enter;
/* ... */
ref_enter = vector;
ref_enter = ref_enter + 3;
*ref_enter = 15; /* És equivalent a vector[3] = 15 */
/* ... */
```

En tot cas, hi ha l'operador `sizeof` ('mida de') que torna com a resultat la mida en bytes del tipus de dades del seu argument. Així doncs, el cas següent és, en realitat, un increment de `ref_enter`, de manera que s'afegeix `3*sizeof(int)` al seu contingut inicial:

```
/* ... */
ref_enter = ref_enter + 3;
/* ... */
```

Per tant, en el cas dels vectors, es compleix que:

```
vector[i] == *(vector+i)
```

És a dir, que l'element que és troba en la posició i-èsima és el que es troba en la posició de memòria que resulta de sumar `i*sizeof (*vector)` a la seva adreça inicial, indicada en vector.

Nota

L'operador `sizeof` s'aplica a l'element apuntat per `vector`, ja que, en cas d'aplicar-se a `vector`, s'obtindria la mida en bytes que ocupa un apuntador.

ANOTACIONS

En l'exemple següent, es podrà observar amb més detall la relació entre apuntadors i vectors. El programa presentat a continuació pren com entrada un nom complet i el separa en nom i cognoms:

```
#include <stdio.h>
#include <ctype.h>
typedef char frase_t[256];
char *copia_paraula( char *frase, char *paraula )
/* Copia la primera paraula de la frase en paraula.      */
/* Frase : apuntador a un vector de caràcters.           */
/* Paraula : apuntador a un vector de caràcters.         */
/* Torna l'adreça de l'ultim caracter llegit             */
/* en la frase.                                          */
{
   while( *frase!='\0' && isspace( *frase ) ) {
     frase = frase + 1;
   } /* while */
   while( *frase!='\0' && !isspace( *frase ) ) {
     *paraula = *frase;
     paraula = paraula + 1;
     frase = frase + 1;
   } /* while */
   *paraula = '\0';
   return frase;
} /* paraula */

main( void ) {
   frase_t nom_complet, nom, cognom1, cognom2;
   char *posicio;

   printf( "Nom i cognoms? " );
   gets( nom_complet );
   posicio = copia_paraula( nom_complet, nom );
   posicio = copia_paraula ( posicio, cognom1 );
   posicio = copia_paraula ( posicio, cognom2 );
   printf(
     "Gràcies per la seva amabilitat, Sr/a. %s.\n",
     cognom1
   ); /* printf */
} /* main */
```

ANOTACIONS

Nota

Més endavant, quan es tracti de les cadenes de caràcters, s'insistirà de nou en la relació entre apuntadors i vectors.

3.3.2. Referències de funcions

Les referències de funcions són, en realitat, l'adreça a la primera instrucció executable. Per tant, es poden emmagatzemar en apuntadors a funcions.

La declaració d'un apuntador a una funció es fa de manera semblant a la declaració dels apuntadors a variables: n'hi ha prou d'incloure en el seu nom un asterisc. Així doncs, un apuntador a una funció que torna un nombre real, producte de fer alguna operació amb l'argument, es declararia de la manera següent:

```
float (*ref_funcio)( double x );
```

Nota

El parèntesi que enclou el nom de l'apuntador i l'asterisc que el precedeix és necessari perquè no es confongui amb la declaració d'una funció el valor de la qual tornat sigui un apuntador a un nombre real.

Per exemple, un programa per a la integració numèrica d'un determinat conjunt de funcions. Aquest programa és semblant al que ja es va veure en la unitat anterior amb la modificació que la funció d'integració numèrica té com a nou argument la referència de la funció de la qual cal calcular la integral:

```
/* Programa: integrals.c */
#include <stdio.h>
#include < math.h >
double f0( double x ) { return x/2.0;           }
double f1( double x ) { return 1+2*log(x);      }
double f2( double x ) { return 1.0/ (1.0 + x*x);}
```

ANOTACIONS

```
double integral_f( double a, double b, int n,
                   double (*fref)( double x )
) {
   double result;
   double x, dx;
   int i;
   result = 0.0;
   if( (a < b) && (n > 0) ) {
    x = a;
    dx = (b-a)/n;
    for( i = 0; i < n; i = i + 1 ) {
     result = result + (*fref)(x)* dx ;
     x = x + dx;
    } /* for */
   } /* if */
   return result;
} /* integral_f */

void main( void )
{
   double   a, b;
   int      n, fnum;
   double   (*fref)( double x );
printf( "Integració numèrica de f(x).\n" );
printf( "Punt inicial de l'interval, a = ? " );
scanf( "%lf", &a );
printf( "Punt final de l'interval, b = ? " );
scanf( "%lf", &b );
printf( "Nombre de divisions, n = ? " );
scanf( "%d", &n );
printf( "Número de funció, fnum = ? ");
scanf( "%d", &fnum );
   switch( fnum ) {
    case 1 : fref = f1; break;
    case 2 : fref = f2; break;
    default: fref = f0;
   } /* switch */
   printf(
    "Resultat, integral(f)[%g,%g] = %g\n",
    a, b, integral_f( a, b, n, fref )
   ); /* printf */
} /* main */
```

ANOTACIONS

Com es pot observar, el programa principal podria substituir perfectament les assignacions de referències de funcions per crides. Amb això, el programa seria molt més clar. Tot i així, com es veurà més endavant, això permet que la funció que fa la integració numèrica pugui estar en alguna biblioteca i ser utilitzada per qualsevol programa.

3.4. Creació i destrucció de variables dinàmiques

Tal com s'ha dit al principi d'aquesta unitat, les variables dinàmiques són les que es creen i destrueixen durant l'execució del programa que les utilitza. Al contrari, les altres són variables estàtiques o automàtiques, que no necessiten accions especials per part del programa per a ser emprades.

Abans de poder emprar una variable dinàmica, cal reservar-hi espai mitjançant la funció estàndard (declarada a `stdlib.h`) que localitza i reserva en la memòria principal un espai de mida `nombre_bytes` perquè pugui contenir les diferents dades d'una variable:

Nota

El tipus de dades `size_t` és, simplement, un tipus d'enter sense signe que s'ha denominat així perquè representa mides.

```
void * malloc( size_t nombre_bytes );
```

Com que la funció desconeix el tipus de dades de la futura variable dinàmica, torna un apuntador a tipus buit, que cal corregir al tipus correcte de dades:

```
/* ... */
char *apuntador;
/* ... */
apuntador = (char *)malloc( 31 );
/* ... */
```

Si no pot reservar espai, torna `NULL`.

En general, resulta difícil conèixer exactament quants bytes ocupa cada tipus de dades i, d'altra banda, la seva mida pot dependre del

ANOTACIONS

compilador i de la màquina que s'utilitzi. Per aquest motiu és convenient utilitzar sempre l'operador `sizeof`. Així, l'exemple anterior hauria d'haver estat escrit de la manera següent:

```
/* ... */
apuntador = (char *)malloc( 31 * sizeof(char) );
/* ... */
```

`sizeof` torna el nombre de bytes necessari per a contenir el tipus de dades de la variable o del tipus de dades que té com a argument, excepte en el cas de les matrius, que torna el mateix valor que per a un apuntador.

De vegades, és necessari ajustar la mida reservada per a una variable dinàmica (sobretot, en el cas de les de tipus vector) perquè falta espai per a noves dades, o bé perquè es desaprofita gran part de l'àrea de memòria. Amb aquesta finalitat, és possible emprar la funció de "relocalització" d'una variable:

```
void * realloc( void *apuntador, size_t nova_mida );
```

El comportament de la funció anterior és similar a la de `malloc`: torna `NULL` si no ha pogut trobar un emplaçament nou per a la variable amb la mida indicada.

Quan una variable dinàmica ja no és necessària, s'ha de destruir; és a dir, alliberar l'espai que ocupa perquè altres variables dinàmiques el puguin emprar. Per a això cal utilitzar la funció `free`:

```
/* ... */
free( apuntador );
/* ... */
```

Com que aquesta funció només allibera l'espai ocupat però no modifica en absolut el contingut de l'apuntador, resulta que aquest encara té la referència a la variable dinàmica (la seva

ANOTACIONS

adreça) i, per tant, hi ha la possibilitat d'accedir a una variable inexistent. Per a evitar-ho, és molt convenient assignar l'apuntador a `NULL`:

```
/* ... */
free( apuntador );
apuntador = NULL;
/* ... */
```

D'aquesta manera, qualsevol referència errònia a la variable dinàmica destruïda causarà error, el qual es podrà corregir fàcilment.

3.5. Tipus de dades dinàmiques

Aquelles dades l'estructura de les quals pot variar al llarg de l'execució d'un programa s'anomenen *tipus de dades dinàmiques*.

L'estructura pot variar únicament en el nombre d'elements, com en el cas d'una cadena de caràcters, o també en la relació entre ells, com podria ser el cas d'un arbre sintàctic.

Els tipus de dades dinàmiques es poden emmagatzemar en estructures de dades estàtiques, però en tractar-se d'un conjunt de dades, han de ser vectors, o bé de manera menys habitual, matrius multidimensionals.

Nota

Les estructures de dades estàtiques s'oposen, per definició, a les estructures de dades dinàmiques. És a dir, aquelles en què tant el nombre de les dades com la seva interrelació no varien en tota l'execució del programa corresponent. Per exemple, un vector sempre tindrà una longitud determinada i tots els elements a excepció del primer i de l'últim tenen un element precedent i un altre de següent.

En cas d'emmagatzemar les estructures de dades dinàmiques en estructures estàtiques, és recomanable comprovar si es coneix el

ANOTACIONS

nombre màxim i la quantitat mitjana de dades que puguin tenir. Si tots dos valors són semblants, es pot emprar una variable de vector estàtica o automàtica. Si són molt diferents, o bé es desconeixen, és convenient ajustar la mida del vector al nombre d'elements que hi hagi en un moment determinat en l'estructura de dades i, per tant, emmagatzemar el vector en una variable de caràcter dinàmic.

Les estructures de dades dinàmiques s'emmagatzemen, en general, emprant variables dinàmiques. Així doncs, es pot veure una estructura de dades dinàmica com una col·lecció de variables dinàmiques la relació de les quals queda establerta mitjançant apuntadors. D'aquesta manera, es pot modificar fàcilment tant el nombre de dades de l'estructura (creant o destruint les variables que les contenen) com la mateixa estructura, canviant les adreces contingudes en els apuntadors dels seus elements. En aquest cas, és habitual que els elements siguin tuples i que s'anomenin *nodes*.

En els apartats següents es veuran els dos casos, és a dir, estructures de dades dinàmiques emmagatzemades en estructures de dades estàtiques i com a col·leccions de variables dinàmiques. En el primer cas, es tractarà de les cadenes de caràcters, ja que són, de llarg, les estructures de dades dinàmiques més emprades. En el segon, de les llistes i de les seves aplicacions.

3.5.1. Cadenes de caràcters

Les cadenes de caràcters són un cas particular de vectors en què els elements són caràcters. A més a més, s'utilitza una marca de final (el caràcter NUL o `'\0'`) que delimita la longitud real de la cadena representada en el vector.

La declaració següent seria una font de problemes derivada de la no-inclusió de la marca de final, ja que és una norma de C que cal respectar per a poder emprar totes les funcions estàndard en el procés de cadenes:

```
char cadena[20] = { 'H', 'o', 'l', 'a' } ;
```

ANOTACIONS

Per tant, caldria inicialitzar-la de la manera següent:

```
char cadena[20] = { 'H', 'o', 'l', 'a', '\0' } ;
```

Les declaracions de cadenes de caràcters inicialitzades mitjançant text impliquen que la marca de final s'afegeixi sempre. És per això que la declaració anterior és equivalent a:

```
char cadena[20] = "Hola" ;
```

Encara que el format de representació de cadenes de caràcters sigui estàndard en C, no es disposa d'instruccions ni d'operadors que treballin amb cadenes: no es poden fer assignacions ni comparacions de cadenes, és necessari recórrer a les funcions estàndard (declarades a `string.h`) per al maneig de cadenes:

```
int    strlen   ( char *cadena );
char * strcpy   ( char *destinacio, char *font );
char * strncpy  ( char * destinacio, char *font, int nre_car );
char * strcat   ( char * destinacio, char *font );
char * strncat  ( char * destinacio, char *font, int nre_car );
char * strdup   ( char *origen );
char * strcmp   ( char *cadena1, char *cadena2 );
char * strncmp  ( char *kdna1, char *kdna2, int nre_car );
char * strchr   ( char *cadena, char caracter );
char * strrchr  ( char *cadena, char caracter );
```

La longitud real d'una cadena de caràcters `kdna` en un moment determinat es pot obtenir mitjançant la funció següent:

```
strlen ( kdna )
```

El contingut de la cadena de caràcters apuntada per `kdna` es pot copiar a `kdna9`, amb `strcpy( kdna9, kdna )`. Si la cadena font pot ser més llarga que la capacitat del vector corresponent a la de destinació, amb `strncpy( kdna9, kdna, LONG_KDNA9 - 1 )`. En aquest últim cas, cal preveure que la cadena resultant no porti un `'\0'` al final. Per a solucionar-lo, s'ha de reservar l'últim caràcter de

ANOTACIONS

la còpia amb límit per a posar un '\0' de guàrdia. Si la cadena no tingués espai reservat, s'hauria de fer el següent:

```
/* ... */
char *kdna9, kdna[LONG_MAX];
/* ... */
kdna9 = (char *) malloc( strlen( kdna ) + 1 );
if( kdna9 != NULL ) strcpy( kdna9, kdna );
/* ... */
```

Nota

D'aquesta manera, `kdna9` és una cadena amb l'espai ajustat al nombre de caràcters de la cadena emmagatzemada a `kdna`, que s'haurà d'alliberar per mitjà d'un `free( kdna9 )` quan ja no sigui necessària. El procediment anterior es pot substituir pel següent:

```
/* ... */
kdna9 = strdup( kdna );
/* ... */
```

La comparació entre cadenes es fa caràcter per caràcter, començant pel primer de les dues cadenes que s'han de comparar i continuant pels següents mentre la diferència entre els codis ASCII sigui 0. La funció `strcmp()` retorna el valor de l'última diferència. És a dir, un valor negatiu si la segona cadena és alfabèticament més gran que la primera, positiu en cas oposat, i 0 si són iguals. Per a entendre-ho millor, s'adjunta un possible codi per a la funció de comparació de cadenes:

```
int strcmp( char *cadena1, char *cadena2 )
{
   while( (*cadena1 != '\0') &&
          (*cadena2 != '\0') &&
          (*cadena1 == *cadena2)
   ) {
     cadena1 = cadena1 + 1;
     cadena2 = cadena2 + 1;
   } /* while */
   return *cadena1 - *cadena2;
} /* strcmp */
```

ANOTACIONS

La funció `strncmp()` fa el mateix que `strcmp()` amb els primers `nre_car` caràcters.

Finalment, encara que n'hi ha més, es comenten les funcions de cerca de caràcters dins de cadenes. Aquestes funcions retornen l'apuntador al caràcter buscat o `NULL` si no es troba en la cadena:

- `strchr()` fa la cerca des del primer caràcter.
- `strrchr()` inspecciona la cadena començant per la dreta; és a dir, pel final.

Exemple

```
char *strchr( char *cadena, char caracter )
{
    while(  (*cadena != '\0') &&
            (*cadena2 != caracter
)   ) {
      cadena = cadena + 1;
    } /* while */
    return cadena;
} /* strchr */
```

Totes les funcions anteriors estan declarades a `string.h` i, per tant, per a utilitzar-les, cal incloure aquest fitxer en el codi font del programa corresponent.

A `stdio.h` també hi ha funcions estàndard per a operar amb cadenes, com `gets()` i `puts()`, que serveixen per a l'entrada i la sortida de dades que siguin cadenes de caràcters i que ja s'han descrit en el seu moment. També conté les declaracions de `sscanf()` i `sprintf()` per a la lectura i l'escriptura de cadenes amb format. Aquestes dues últimes funcions es comporten exactament igual que `scanf()` i `printf()`, excepte que la lectura o escriptura es fan en una cadena de caràcters en lloc de fer-ho en el dispositiu estàndard d'entrada o sortida:

```
sprintf(
   char *destinacio /* Cadena en la qual "imprimeix".*/
   char *format
   [, llista_de_variables]
); /* sprintf */
```

ANOTACIONS

```
int sscanf(          /* Torna el nombre de variables                    */
                     /* el contingut de les quals ha estat actualitzat.*/
   char *origen,     /* Cadena de la qual es "llegeix".                 */
   char *format
   [,llista_de_&variables]
); /* sscanf */
```

Quan s'utilitza `sprintf()` s'ha de comprovar que la cadena `destinació` tingui espai suficient per a contenir allò que resulti de la impressió amb el format donat.

Quan s'utilitza `sscanf()`, s'ha de comprovar sempre que s'han llegit tots els camps: la inspecció de la cadena `origen` es deté en trobar la marca de final de cadena independentment dels especificadors de camp indicats en el format.

En l'exemple següent, es pot observar el codi d'una funció de conversió d'una cadena que representi un valor hexadecimal (per exemple: "3D") a un enter positiu (seguint l'exemple anterior: $3D_{(16} = 61$) mitjançant l'ús de les funcions anteriorment esmentades. La primera s'empra per a prefixar la cadena amb "0x", ja que és el format dels nombres hexadecimals en C, i la segona, per a efectuar la lectura de la cadena obtinguda aprofitant que llegeix nombres en qualsevol format estàndard de C.

```
unsigned hexaVal( char *hexadecimal )
{
   unsigned nombre;
   char *hexaC;

   hexaC = (char *) malloc(
     ( strlen( hexadecimal ) +3 ) * sizeof( char )
   ); /* malloc */
   if( hexaC != NULL ) {
    sprintf( hexaC, "0x%s", hexadecimal );
    sscanf( hexaC, "%x", &nombre );
```

ANOTACIONS

```
    free( hexaC );
   } else {
    nombre = 0; /* La conversió no s'ha fet!*/
   } /* if */
   return nombre;
} /* hexaVal */
```

Nota

Recordeu la importància d'alliberar l'espai de les cadenes de caràcters creades dinàmicament que no s'utilitzaran. En el cas anterior, si no es fa un `free( hexaC )`, la variable continuaria ocupant espai, malgrat que ja no s'hi podria accedir, ja que l'adreça està continguda en l'apuntador `hexaC`, que és de tipus automàtic i, per tant, es destrueix en acabar l'execució de la funció. Aquest tipus d'errors pot arribar a causar un gran malbaratament de memòria.

3.5.2. Llistes i cues

Les llistes són un dels tipus dinàmics de dades més emprades i consisteixen en seqüències homogènies d'elements sense mida predeterminada. Igual com les cadenes de caràcters, es poden emmagatzemar en vectors sempre que se'n sàpiga la longitud màxima i la longitud mitjana durant l'execució. Si això no és així, s'utilitzaran variables dinàmiques "enllaçades" entre elles; és a dir, variables que contindran apuntadors a d'altres dins de la mateixa estructura dinàmica de dades.

L'avantatge que aporta representar una llista en un vector és que elimina la necessitat d'un camp apuntador al següent. De totes maneres, cal insistir que només és possible aprofitar-la si no es produeix un malbaratament de memòria excessiu i quan el programa no hagi de fer freqüents insercions i eliminacions d'elements en qualsevol posició de la llista.

En aquest apartat es veurà una manera possible de programar les operacions bàsiques amb una estructura de dades dinàmica de tipus

ANOTACIONS

llista mitjançant variables dinàmiques. En aquest cas, cada element de la llista serà un node del tipus següent:

```
typedef struct node_s {
   int           dada;
   struct node_s *seguent;
} node_t, *llista_t;
```

El tipus `node_t` es correspon amb un node de la llista i `llista_t` és un apuntador a un node el tuple del qual té un camp `seguent` que, alhora, és un apuntador a un altre node, i així repetidament fins que té enllaçada tota una llista de nodes. Aquest tipus de llistes s'anomenen ***llistes simplement enllaçades***, ja que només hi ha un enllaç entre un node i el següent.

Les llistes simplement enllaçades són adequades per a algoritmes que facin recorreguts seqüencials freqüents. En canvi, per aquells en què es fan recorreguts parcials en tots dos sentits (cap endavant i cap enrere en la seqüència de nodes) és recomanable utilitzar **llistes doblement enllaçades**; és a dir, llistes els nodes de les quals continguin apuntadors als elements següents i anteriors.

En tots dos casos, si l'algoritme fa insercions de nous elements i destruccions d'elements innecessaris de manera molt freqüent, pot ser convenient tenir el primer i l'últim element enllaçats. Aquestes estructures s'anomenen ***llistes circulars*** i, habitualment, els primers elements estan marcats amb alguna dada o camp especial.

Operacions elementals amb llistes

Una llista d'elements ha de permetre dur a terme les operacions següents:

- Accedir a un node determinat.
- Eliminar un node existent.
- Inserir un node nou.

ANOTACIONS

En els apartats següents es comentaran aquestes tres operacions i es donaran els programes de les funcions per a dur-les a terme sobre una llista simplement enllaçada.

Per a accedir a un node determinat és imprescindible obtenir-ne la adreça. Si se suposa que es tracta d'obtenir l'enèsim element en una llista i tornar la seva adreça, la funció corresponent necessita com a arguments tant la posició de l'element buscat com l'adreça del primer element de la llista, que pot ser `NULL` si aquesta és buida.

Evidentment, la funció retornarà l'adreça del node enèsim o `NULL` si no l'ha trobat:

```
node_t *enesim_node( llista_t llista, unsigned int n )
{
   while( ( llista != NULL ) && ( n != 0 ) ) {
     llista = llista -> seguent;
     n = n - 1;
   } /* while */
   return llista;
} /* enesim_node */
```

En aquest cas, es considera que la primera posició és la posició número 0, de manera semblant als vectors en C. És imprescindible comprovar que `( llista != NULL )` es compleixi, ja que, altrament, no es podria executar `llista = llista->seguent;`: no es pot accedir al camp següent d'un node que no existeix.

Per a eliminar un element en una llista simplement enllaçada, és necessari disposar de l'adreça de l'element anterior, ja que el camp següent d'aquest últim element s'ha d'actualitzar convenientment. Per a això, és necessari estendre la funció anterior de manera que torni tant l'adreça de l'element buscat com la de l'element anterior. Com que ha de tornar dues dades, cal fer-ho per mitjà d'un pas per referència: cal passar les adreces dels apuntadors a node que contindran les adreces dels nodes:

ANOTACIONS

```
void enesim_pq_node(
   llista_t llista,    /* Apuntador al primer node.      */
   unsigned int n,     /* Posicio del node buscat.       */
   node_t     **pref,  /* Ref. apuntador a node previ.   */
   node_t     **qref)  /* Ref. apuntador a node actual.  */
{
   node_t *p, *q;
   p = NULL;           /* L'anterior al primer no existeix.*/
   q = llista;
   while( ( q != NULL ) && ( n != 0 ) ) {
    p = q;
    q = q->seguent;
    n = n - 1;
   } /* while */
   *pref = p;
   *qref = q;
} /* enesim_pq_node */
```

El programa de la funció que destrueixi un node ha de partir de la idea que es coneix tant la seva adreça (emmagatzemada a `q`) com la del possible element anterior (desada a l'apuntador `p`). Dit d'una altra manera, es pretén eliminar l'element següent a l'apuntat per `p`.

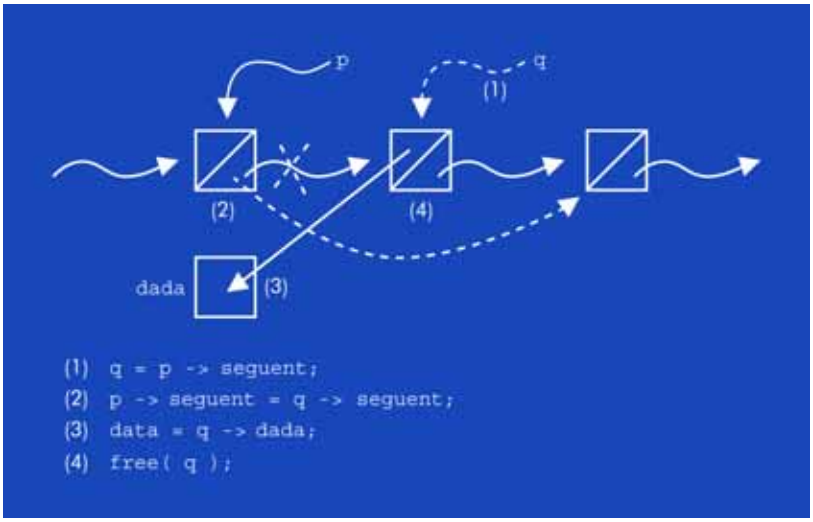
Quan es fan aquests programes, és més que convenient fer un esquema de l'estructura de dades dinàmica en el qual s'indiquin els efectes de les diferents modificacions d'aquesta.

Així doncs, per a programar aquesta funció, primer establirem el cas general sobre un esquema de l'estructura de la qual volem eliminar un node i, després, es programarà atenent les diferents excepcions que pot tenir el cas general. Habitualment, aquestes excepcions apareixen en tractar el primer o l'últim element, perquè són les que no tenen precedent o següent i, com a conseqüència, no segueixen la norma dels altres en la llista.

ANOTACIONS

En la imatge següent es resumeix el procediment general d'eliminació del node següent al node `p`:

Figura 6.



Nota

Evidentment, no es pot eliminar un node quan `p == NULL`. En aquest cas, tant (1) com (2) no es poden executar perquè impliquen referències a variables inexistents. Més encara, per a `p == NULL` es compleix que es vol eliminar el primer element, ja que és l'únic que no té cap element que el precedeixi. Així doncs, hem de "protegir" l'execució d'(1) i (2) amb una instrucció condicional que decideixi si es pot dur a terme el cas general o, al contrari, s'elimina el primer dels elements. Una cosa semblant passa amb (3) i (4), que no es poden executar si `q == NULL`.

La funció per a destruir el node podria ser com la següent:

```
int destrueix_node (
   llista_t *llistaref,/* Apuntador a referència 1r. node.      */
   node_t *p,          /* Apuntador a node previ.               */
   node_t *q)          /* Apuntador a node per destruir.        */
{
   int data = 0        * Valor per omissió de les dades.        */
   if( p != NULL ) {
    /* q = p->seguent; (No és necessari) */
```

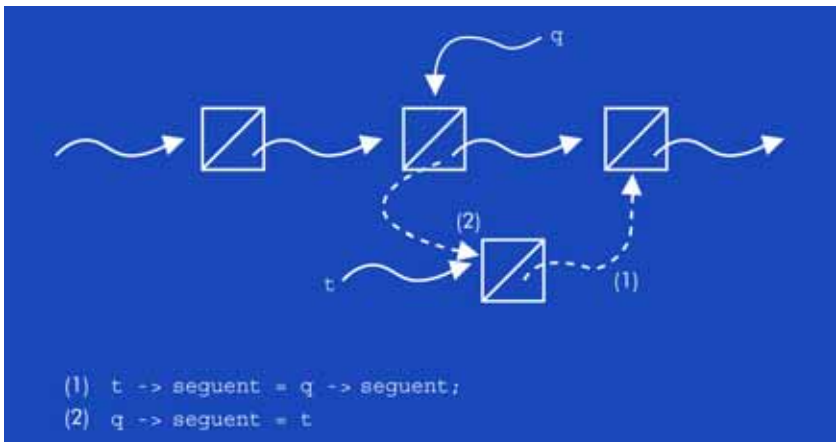
ANOTACIONS

```
    p->seguent = q->seguent;
   } else {
    if( q != NULL ) *llistaref = q->seguent;
   } /* if */
   if( q!= NULL ) {
    data = q->dada;
    free( q );
   } /* if */
   return data;
} /* destrueix_node */
```

En eliminar el primer element, és necessari canviar l'adreça continguda a la llista perquè apunti al nou primer element, llevat que `q` també sigui `NULL`).

Per a inserir un nou node, l'adreça del qual sigui a `t`, n'hi ha prou de fer les operacions indicades en la figura següent:

Figura 7.



Nota

En aquest cas, el node `t` quedarà inserit després del node `q`. Com es pot observar, és necessari que `q != NULL` per a poder fer les operacions d'inserció.

El codi de la funció corresponent seria com segueix:

```
void insereix_seguent_node(
   llista_t *llistaref /* Apuntador a referència 1r. node.      */
```

ANOTACIONS

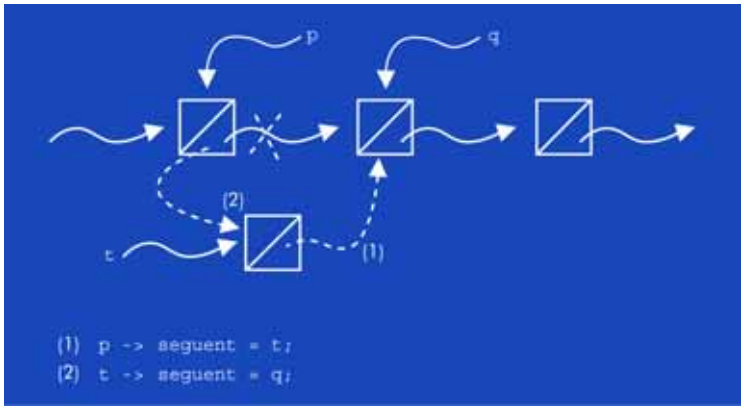
```
   node_t   *q,         /* Apuntador a node en la posició.       */
   node_t   *t)         /* Apuntador a node que s'ha d'inserir.  */
{
   if( q != NULL ) {
     t->seguent = q->seguent;
     q->seguent = t;
   } else { /* La llista és buida.                               */
     *llistaref = t;
   } /* if */
} /* insereix_seguent_node */
```

Perquè la funció anterior pugui ser útil, és necessari disposar d'una funció que permeti crear nodes de la llista. En aquest cas:

```
node_t *crea_node( int data )
{
   node_t *noderef;
   noderef = (node_t *)malloc( sizeof( node_t ) );
   if( noderef != NULL ) {
     noderef->dada = data;
     noderef->seguent = NULL;
   } /* if */
   return noderef;
} /* crea_node */
```

Si es tracta d'inserir en alguna posició determinada, el més freqüent sol ser inserir el nou element com a precedent de l'indicat; és a dir, que ocupi la posició del node referenciat i desplaci la resta de nodes una posició "a la dreta". Per a això les operacions que cal fer en el cas general es mostren en la figura següent:

Figura 8.



ANOTACIONS

Seguint les indicacions de la figura anterior, el codi de la funció corresponent seria el següent:

```
void insereix_node(
   llista_t *llistaref,/* Apuntador a referencia 1r. node.    */
   node_t *p,          /* Apuntador a node precedent.         */
   node_t *q,          /* Apuntador a node en la posicio.     */
   node_t *t)          /* Apuntador a node que s'ha d'inserir. */
{
   if( p != NULL ) {
    p->seguent = t;
   } else { /* S'insereix un nou primer element.              */
   *llistaref = t;
   } /* if */
   t->seguent = q;
} /* insereix_node */
```

Així doncs, la inserció d'un node en la posició enèsima es podria construir de la manera següent:

```
bool insereix_enesim_llista(
   llista_t  *llistaref,/* Apuntador a referència 1r. node.*/
   unsigned  int n,     /* Posició de la inserció.         */
   int       dada)      /* Dada que s'ha d'inserir.        */
{                       /* Torna FALSE si no es pot.       */
   node_t *p, *q, *t;
   bool retval;
   t = crea_node( dada );
   if( t != NULL ) {
    enesim_pq_node( *llistaref, n, &p, &q );
    insereix_node( llistaref, p, q, t );
    retval = TRUE;
   } else {
    retval = FALSE;
   } /* if */
   return retval;
} /* insereix_enesim_llista */
```

De la mateixa manera, es podria compondre el codi per a la funció de destrucció de l'enèsim element d'una llista.

ANOTACIONS

Normalment, a més a més, les llistes no seran d'elements tan simples com els enters i caldrà substituir la definició del tipus de dades `node_t` per un de més adequat.

Els criteris de recerca de nodes solen ser més sofisticats que la recerca d'una determinada posició. Per tant, cal prendre les funcions anteriors com un exemple d'ús a partir del qual es poden derivar casos reals d'aplicació.

Cues

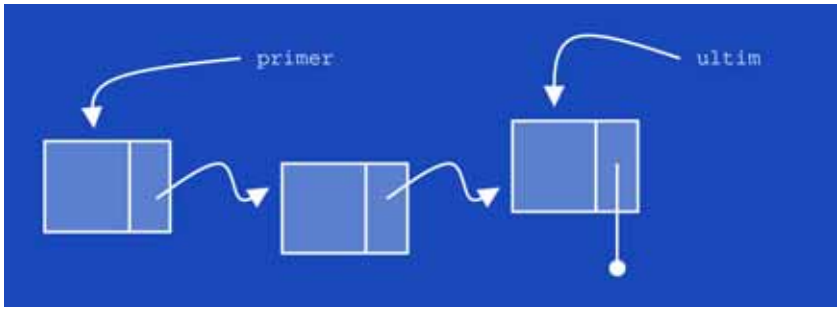
Les cues són, de fet, llistes en les quals s'insereix per un extrem i s'elimina per l'altre. És a dir, són llistes en les quals les operacions d'inserció i d'eliminació es restringeixen a uns casos molt determinats. Això permet gestionar-les d'una manera molt més eficaç. En aquest sentit, és convenient disposar d'un tuple que faciliti tant l'accés directe al primer element com a l'últim. D'aquesta manera, les operacions d'eliminació i inserció es poden resoldre sense necessitat de recerques en la llista.

Així doncs, aquesta classe de cues hauria de tenir un tuple de control com el següent:

```
typedef struct cua_s {
   node_t *primer;
   node_t *ultim;
} cua_t;
```

Gràficament:

Figura 9.



ANOTACIONS

Per a exemplificar les dues operacions, suposarem que els elements de la cua seran simples enters, és a dir, que la cua serà una llista de nodes del tipus de dades `node_t` que ja s'ha vist.

Amb això, la inserció seria així:

```
bool encua( cua_t *cuaref, int dada )
/* Torna FALSE si no es pot afegir la dada.*/
{
   node_t *q, *t;
   bool   retval;

   t = crea_node( dada );
   if( t != NULL ) {
     t->seguent = NULL;
     q = cuaref->ultim;
     if( q == NULL ) { /* Cua buida: */
       cuaref->primer = t;
     } else {
       q->seguent = t;
     } /* if */
     cuaref->ultim = t;
     retval = TRUE;
   } else {
     retval = FALSE;
   } /* if */
   return retval;
} /* encua */
```

I l'eliminació:

```
bool desencua( cua_t *cuaref, int *dadaref )
/* Torna FALSE si no es pot eliminar la dada.   */
{
   node_t *q;
   bool retval;
   q = cuaref->primer;
   if( q != NULL ) {
    cuaref->primer = q->seguent;
    *dadaref = destrueix_node( &q );
```

ANOTACIONS

```
    if( cuaref->primer == NULL ) { /* Cua buida:*/
      cuaref->ultim = NULL;
    } /* if */
    retval = TRUE;
  } else {
    retval = FALSE;
  } /* if */
  return retval;
} /* desencua */
```

La funció `destrueix_node` de l'eliminació anterior és com segueix:

```
int destrueix_node( node_t **pref )
{
  int dada = 0;
  if( *pref != NULL ) {
    dada = (*pref)-> dada;
    free( *pref );
    *pref = NULL;
  } /* if */
  return dada;
} /* destrueix_node */
```

Les cues s'utilitzen freqüentment quan hi ha recursos compartits entre molts usuaris.

Exemple

- Per a gestionar una impressora, el recurs és la mateixa impressora i els usuaris, els ordinadors que hi estan connectats.
- Per a controlar una màquina del tipus "el seu torn": el recurs és el venedor i els usuaris són els clients.

En general, quan es fan insercions en cua es té en compte quin element s'està inserint; és a dir, no es fan sempre pel final, sinó que l'element es col·loca segons els seus privilegis sobre els altres. En aquest cas, es parla de cues amb prioritat. En aquest tipus de cues, l'eliminació sempre es fa pel principi, però la inserció impli-

ANOTACIONS

ca situar l'element nou en l'última posició d'elements amb la mateixa prioritat.

Certament, hi ha altres tipus de gestions especialitzades amb llistes i, més enllà de les llistes, altres tipus d'estructures dinàmiques de dades com els arbres (per exemple, un arbre sintàctic) i els grafs (per exemple, una xarxa de carreteres). Malauradament, no es disposa de prou temps per a tractar-los, però cal tenir-los en compte quan les dades del problema ho puguin demanar.

3.6. Disseny descendent de programes

Recordem que la programació modular consisteix a dividir el codi en subprogrames que facin una funció concreta. En aquesta unitat es tractarà especialment de la manera com es poden agrupar aquests subprogrames d'acord amb les tasques que els són encomanades i, en definitiva, de com es poden organitzar per a una millor programació de l'algoritme corresponent.

Els algoritmes complexos se solen reflectir en programes amb moltes línies de codi. Per tant, s'imposa una programació molt curosa perquè el resultat sigui un codi llegible i de manteniment fàcil.

3.6.1. Descripció

El fruit d'una programació modular és un codi constituït per diversos subprogrames de poques línies relacionats entre ells mitjançant crides. Així doncs, cadascun pot ser de comprensió fàcil i, en conseqüència, de manteniment fàcil.

La tècnica de disseny descendent és, de fet, una tècnica de disseny d'algoritmes en què es resol l'algoritme principal abstraient els detalls, que després se solucionen mitjançant altres algoritmes de la mateixa manera. És a dir, es comença pel nivell d'abstracció més alt i, per a totes aquelles accions que no es puguin traslladar de manera directa a alguna instrucció del llenguatge de programació triat, es dissenyen els algoritmes corresponents de manera independent del principal seguint el mateix principi.

ANOTACIONS

El disseny descendent consisteix a escriure programes d'algoritmes d'unes quantes instruccions i implementar les instruccions no primitives amb funcions els programes de les quals segueixin les mateixes normes anteriors.

Nota

Una instrucció primitiva és la que es pot programar directament en un llenguatge de programació.

A la pràctica, això comporta dissenyar algoritmes de manera que es permeti programar-los d'una manera totalment modular.

3.6.2. Exemple

El disseny descendent de programes consisteix, doncs, a començar pel programa de l'algoritme principal i anar refinant les instruccions "gruixudes" convertint-les en subprogrames amb instruccions més "fines". D'aquí ve la idea de *refinar*. Evidentment, el procés acaba quan ja no hi ha més instruccions "gruixudes" per refinar.

En aquest apartat veurem un exemple simple de disseny descendent per a resoldre un problema bastant habitual en la programació: el de l'ordenació de dades per a facilitar-ne, per exemple, la consulta.

Aquest problema és un dels més estudiats en la ciència informàtica i hi ha diversos mètodes per a solucionar-lo. Un dels més simples consisteix a seleccionar l'element que hauria d'encapçalar la classificació (per exemple, el més petit o el més gran, si es tracta de nombres), posar-lo en la llista ordenada i repetir el procés amb la resta d'elements per ordenar. El programa principal d'aquest algoritme pot ser el següent:

```
/* ... */
llista_t  pendents, ordenats;
element_t element;
/* ... */
inicialitza_llista( &ordenats );
while(! es_buida_llista( pendents ) ) {
   element = extreu_minim_de_llista( &pendents );
   posa_al_final_de_llista( &ordenats, element );
} /* while */
/* ... */
```

ANOTACIONS

En el programa anterior hi ha poques instruccions primitives de C i, per tant, caldrà refinar-lo. Com a contrapartida, el seu funcionament resulta fàcil de comprendre. És important tenir en compte que els operadors d'"adreça de" (el signe `&`) en els paràmetres de les crides de les funcions indiquen que poden modificar el seu contingut.

La dificultat més gran del procés de refinament és, habitualment, identificar les parts que s'han de descriure amb instruccions primitives; és a dir, determinar els diferents nivells d'abstracció que haurà de tenir l'algoritme i, consegüentment, el programa corresponent. Generalment, es tracta d'aconseguir que el programa reflecteixi al màxim l'algoritme del qual prové.

És un error comú pensar que les operacions que només exigeixen una o dues instruccions primitives no es poden veure mai com una instrucció no primitiva.

Una norma d'adopció fàcil és que totes les operacions que es facin amb un tipus de dades abstracte siguin igualment abstractes; és a dir, es materialitzin en instruccions no primitives (funcions).

3.7. Tipus de dades abstractes i funcions associades

La millor manera d'implementar la programació descendent és programar totes les operacions que es puguin fer amb cada una dels tipus de dades abstractes que s'hagin d'emprar. De fet, es tracta de crear una màquina virtual per executar instruccions que s'ajusten a l'algoritme, com un llenguatge que disposa de totes les operacions necessàries per a la màquina que és capaç de processar-lo i, òbviament, després es trasllada a les operacions del llenguatge de la màquina real que farà el procés.

En l'exemple de l'algoritme d'ordenació anterior, apareixen dos tipus de dades abstractes (`llista_t` i `element_t`) per a les quals, com a mínim, són necessàries les operacions següents:

```
void inicialitza_llista( llista_t *ref_llista );
bool es_buida_llista( llista_t llista );
```

ANOTACIONS

```
element_t extreu_minim_de_llista( llista_t *ref_llista );
void posa_al_final_de_llista( llista_t *rlst, element_t i );
```

Com es pot observar, no hi ha operacions que afectin dades del tipus `element_t`. En tot cas, és segur que el programa en farà ús (lectura de dades, inserció en la llista, comparació entre elements, escriptura de resultats, etc.) en alguna altra secció. Per tant, també s'hauran de programar les operacions corresponents. En particular, si s'observa el codi següent es comprovarà que apareixen funcions per a tractar amb dades del tipus `element_t`:

```
element_t extreu_minim_de_llista( llista_t *ref_llista )
{
   ref_node_t actual, minim;
   bool es_menor;
   element_t petit;
   principi_de_llista( ref_llista );
   if( es_buida_llista( *ref_llista ) ) {
    inicialitza_element( &petit );
   } else {
    minim = ref_node_de_llista( *ref_llista );
    petit = element_en_ref_node( *ref_llista, minim );
    avanca_posicio_en_llista( ref_llista );
    while( !es_final_de_llista( *ref_llista ) ) {
     actual = ref_node_de_llista( *ref_llista );
     es_menor = compara_elements(
      element_en_ref_node( *ref_llista, actual ), petit
     ); /* compara_elements */
     if( es_menor ) {
      minim = actual;
      petit = element_en_ref_node( *ref_llista, minim );
     } /* if */
     avanca_posicio_en_llista( ref_llista );
    } /* while */
    elimina_de_llista( ref_llista, minim );
   } /* if */
   return petit;
} /* extreu_minim_de_llista */
```

ANOTACIONS

Com es pot deduir del codi anterior, almenys són necessàries dues operacions per a dades del tipus `element_t`:

```
inicialitza_element();
compara_elements();
```

A més a més, cal fer quatre operacions més per a llistes:

```
principi_de_llista();
es_final_de_llista();
avanca_posicio_en_llista();
elimina_de_llista();
```

S'ha afegit també el tipus de dades `ref_node_t` per a tenir les referències dels nodes en les llistes i dues operacions: `ref_node_de_llista` per a obtenir la referència d'un node determinat en la llista i `element_en_ref_node` per a obtenir l'element que es desa en el node indicat.

Amb això es demostra que el refinament progressiu serveix per a determinar quines operacions cal fer per a cada tipus de dades i, d'altra banda, es reflecteix que les llistes constitueixen un nivell d'abstracció diferent i més gran que el dels elements.

Per a completar la programació de l'algoritme d'ordenació, és necessari desenvolupar totes les funcions associades a les llistes, i després als elements. De totes maneres, el primer que cal fer és determinar els tipus de dades abstractes que s'empraran.

Les llistes es poden materialitzar amb vectors o amb variables dinàmiques, segons el tipus d'algoritmes que s'utilitzin. En el cas de l'exemple de l'ordenació, dependrà en part dels mateixos criteris generals que s'apliquen per a prendre aquesta decisió i, en part, de les característiques del mateix algoritme. Generalment, s'escollirà una implementació amb vectors si tenen un desaprofitament mitjà petit, i particularment es té en compte que l'algoritme que ens ocupa només es pot aplicar per a la classificació de quantitats modestes de dades (per exemple, uns quants centenars a tot estirar).

ANOTACIONS

Així doncs, en cas d'escollir la primera opció, el tipus de dades `llista_t` seria:

```
#define LONG_MAXIMA 100
typedef struct llista_e {
   element_t node[ LONG_MAXIMA ];
   unsigned short posicio;   /* Posició actual d'accés.*/
   unsigned short quantitat;/* Longitud de la llista.  */
} llista_t;
```

També faria falta definir el tipus de dades per a la referència dels nodes:

```
typedef unsigned short ref_node_t;
```

D'aquesta manera, la resta d'operacions que són necessàries es correspondrien amb les funcions següents:

```
void inicialitza_llista( llista_t *ref_llista )
{
   (*ref_llista).quantitat = 0;
   (*ref_llista).posicio = 0;
} /* inicialitza_llista */

bool es_buida_llista( llista_t llista )
{
   return ( llista.quantitat == 0 );
} /* es_buida_llista */

bool es_final_de_llista( llista_t llista )
{
   return ( llista_posicio == llista.quantitat );
} /* es_final_de_llista */

void principi_de_llista( llista_t *llista_ref )
{
   llista_ref->posicio = 0;
} /* principi_de_llista */
```

ANOTACIONS

```
ref_node_t ref_node_de_llista( llista_t llista )
{
   return llista.posicio;
} /* ref_node_de_llista */

element_t element_en_ref_node(
   llista_t llista,
   ref_node_t refnode)
{
   return llista.node[ refnode ];
} /* element_en_ref_node */

void avanca_posicio_en_llista( llista_t *llista_ref )
{
   if( !es_final_de_llista( *llista_ref ) ) {
    (*llista_ref).posicio = (*llista_ref).posicio + 1;
   } /* if */
} /* avanca_posicio_en_llista */

element_t elimina_de_llista(
   llista_t *ref_llista,
   ref_node_t refnode)
{
   element_t eliminat;
   ref_node_t pos, ultim;
   if( es_buida_llista( *ref_llista ) ) {
    inicialitza_element( &eliminat );
   } else {
    eliminat = (*ref_llista).node[ refnode ];
    ultim = (*ref_llista).quantitat - 1;
    for( pos = refnode; pos < ultim; pos = pos + 1 ) {
      (*ref_llista).node[pos] = (*ref_llista).node[pos+1];
    } /* for */
    (*ref_llista).quantitat = (*ref_llista).quantitat - 1;
   } /* if */
   return eliminat;
} /* elimina_de_llista */

element_t extreu_minim_de_llista( llista_t *ref_llista );
```

ANOTACIONS

```
void posa_al_final_de_llista(
   llista_t *ref_llista,
   element_t element)
{
   if( (*ref_llista).quantitat < LONG_MAXIMA ) {
     (*ref_llista).node[(*ref_llista).quantitat] = element;
     (*ref_llista).quantitat = (*ref_llista).quantitat + 1;
   } /* if */
} /* posa_al_final_de_llista */
```

Si s'examinen les funcions anteriors, totes associades al tipus de dades `llista_t`, veurem que només necessiten una operació amb el tipus de dades `element_t`: `compara_elements`. Per tant, per a completar el programa d'ordenació, ja només falta definir el tipus de dades i l'operació de comparació.

Si bé les operacions amb les llistes sobre vectors eren genèriques, tot allò que afecta els elements dependrà de la informació les dades de la qual es vulguin ordenar.

Per exemple, suposant que es vulguin ordenar per número de DNI les notes d'un examen, el tipus de dades dels elements podria ser així:

```
typedef struct element_s {
       unsigned int   DNI;
       flota          nota;
} dada_t, *element_t;
```

La funció de comparació seria:

```
bool compara_elements(
   element_t menor,
   element_t major )
{
   return ( menor->DNI < major->DNI );
} /* compara_elements */
```

La funció d'inicialització seria:

```
void inicialitza_element( element_t *ref_elem )
{
   *ref_elem = NULL;
} /* inicialitza_element */
```

ANOTACIONS

Noteu que els elements són, en realitat, apuntadors de variables dinàmiques. Això, tot i que requereix que el programa les construeixi i destrueixi, simplifica, i molt, la codificació de les operacions en nivells d'abstracció superiors. No obstant això, és important recalcar que sempre s'hauran de preparar funcions per a la creació, destrucció, còpia i duplicat de cada un dels tipus de dades per a les quals hi ha variables dinàmiques.

Les funcions de còpia i de duplicat són necessàries ja que la simple assignació constitueix una còpia de la adreça d'una variable a un altre apuntador, és a dir, es tindran dues referències a la mateixa variable en lloc de dues variables diferents amb el mateix contingut.

Exemple

Comproveu la diferència que hi ha entre aquestes dues funcions:

```
/* ... */
element_t original, un_altre, copia;
/* ... */
un_altre = original; /* Copia d'apuntadors. */
/* l'adreça desada en 'un_altre'
és la mateixa que la continguda en 'original'
*/

/* ... */
copia_element( original, copia );
un_altre = duplica_element( original );
/* les adreces emmagatzemades en 'copia' i en 'un_altre'
són diferents de la continguda en 'original'
*/
```

La còpia de continguts s'ha de fer, doncs, mitjançant una funció específica i, és clar, si la variable que ha de contenir la còpia no està creada, s'haurà de procedir primer a crear-la, és a dir, se'n farà un duplicat.

En resum, quan hàgim de programar un algoritme en disseny descendent, haurem de programar les funcions de creació, destrucció i còpia per a cada un dels tipus de dades abstractes que contingui. A més a més, es programarà com a funció tota operació que es faci

ANOTACIONS

amb cada tipus de dades perquè quedin reflectits els diferents nivells d'abstracció presents en l'algoritme. Així doncs, fins i tot a costa d'alguna línia de codi més, s'aconsegueix un programa intel·ligible i de manutenció fàcil.

3.8. Fitxers de capçalera

És comú que diversos equips de programadors col·laborin en la realització d'un mateix programa, si bé és cert que moltes vegades aquesta afirmació és més aviat la manifestació d'un desig que el reflex d'una realitat. En empreses petites i mitjanes se sol traduir en la circumstància que els equips siguin d'una persona i, fins i tot, que els diferents equips es redueixin a un de sol. De totes maneres, l'afirmació, en el fons, és certa: l'elaboració d'un programa ha d'aprofitar, en una bona pràctica de la programació, parts d'altres programes. El reaprofitament permet no solament reduir el temps de desenvolupament, sinó també tenir la garantia d'ocupar components de funcionalitat provada.

Tot això és, a més, especialment cert per al programari lliure, en què els programes són, per norma, producte d'un conjunt divers de programadors i no forçosament coordinat: un programador pot haver aprofitat codi elaborat prèviament per d'altres per a una aplicació diferent de la que ha motivat el seu desenvolupament original.

Per a permetre l'aprofitament d'un determinat codi, és convenient eliminar els detalls d'implementació i indicar només el tipus de dades per al qual està destinat i quines operacions es poden fer amb les variables corresponents. Així doncs, n'hi ha prou de disposar d'un fitxer on es defineixin el tipus de dades abstracte i es declarin les funcions que es proveeixen per a les seves variables. A aquests fitxers se'ls anomena *fitxers de capçalera* perquè són la part inicial del codi font de les funcions les declaracions o capçaleres del qual s'han inclòs.

En els fitxers de capçalera es dóna a conèixer tot allò relatiu a un tipus de dades abstracte i a les funcions per al seu maneig.

ANOTACIONS

3.8.1. Estructura

Els fitxers de capçalera porten l'extensió ".h" i el seu contingut s'ha d'organitzar de tal manera que sigui de lectura fàcil. Per a això, primer s'han de col·locar uns comentaris que indiquin la naturalesa del contingut i, sobretot, de la funcionalitat del codi en el fitxer ".c" corresponent. Després, n'hi ha prou de seguir l'estructura d'un programa típic de C: inclusió de fitxers de capçalera, definició de constants simbòliques i, finalment, declaració de les funcions.

La definició ha d'estar sempre al fitxer ".c".

El fitxer de capçalera següent serveix d'exemple per a operar amb nombres complexos:

```
/* Fitxer:     complexos.h                                    */
/* Contingut:  Funcions per a operar amb nombres              */
/*             complexos del tipus (X + iY) en els quals      */
/*             X és la part real i Y, la imaginaria.          */
/* Revisió:    0.0 (original)                                 */

#ifndef _NOMBRES_COMPLEXOS_H_
#define _NOMBRES_COMPLEXOS_H_

#include <stdio.h>
#define PRECISIO 1E-10
typedef struct complex_s {
      double real, imaginari;
} *complex_t;
complex_t nou_complex( double real, double imaginari );
void esborra_complex( complex_t complex );
void imprimeix_complex( FILE *fitxer, complex_t complex );
double modul_complex( complex_t complex );
complex_t oposat_complex( complex_t complex );
complex_t suma_complex( complex_t c1, complex_t c2 );
/* etc. */

#endif /* _NOMBRES_COMPLEXOS_H_ */
```

ANOTACIONS

Les definicions de tipus i constants i les declaracions de funcions s'han col·locat com el cos de l'ordre del preprocessador `#ifndef... #endif`. Aquesta ordre pregunta si una constant determinada està definida i, si no ho està, trasllada al compilador el que contingui el fitxer fins a la marca de final. La primera ordre del cos d'aquesta ordre condicional consisteix, precisament, a definir la constant `_NOMBRES_COMPLEXOS_H_` per a evitar que una altra inclusió del mateix fitxer generi el mateix codi font per al compilador (és innecessari si ja l'ha processat una vegada).

Les anomenades **ordres de compilació condicional** del preprocessador permeten decidir si un determinat tros de codi font se subministra al compilador o no. Els resumim en la taula següent:

Tabla 8.

Ordre	Significat
`#if expressió`	Les línies següents es compilen si `expressió ? 0`.
`#ifdef SÍMBOL`	Es compilen les línies següents si `SÍMBOL` està definit.
`#ifndef SÍMBOL`	Es compilen les línies següents si `SÍMBOL` no està definit.
`#else`	Finalitza el bloc compilat si es compleix la condició i inicia el bloc que s'ha de compilar en cas contrari.
`#elif expressió`	Encadena un `else` amb un `if`.
`#endif`	Indica el final del bloc de compilació condicional.

Nota

Un símbol definit (per exemple: `SÍMBOL`) es pot anul·lar mitjançant:

```
#undef SÍMBOL
```

i, a partir d'aquell moment, es considera com a no definit.

ANOTACIONS

Les formes:

```
#ifdef SÍMBOL
#ifndef SÍMBOL
```

són abreviacions del següent:

```
#if defined( SÍMBOL )
#if !defined( SÍMBOL )
```

respectivament. La funció `defined` es pot emprar en expressions lògiques més complexes.

Per acabar aquesta secció, es pot indicar que el fitxer de codi font associat ha d'incloure, evidentment, el fitxer de capçalera:

```
/* Fitxer:   complexos.c                    */
/* ... */
#include "complexos.h"
/* ... */
```

Nota

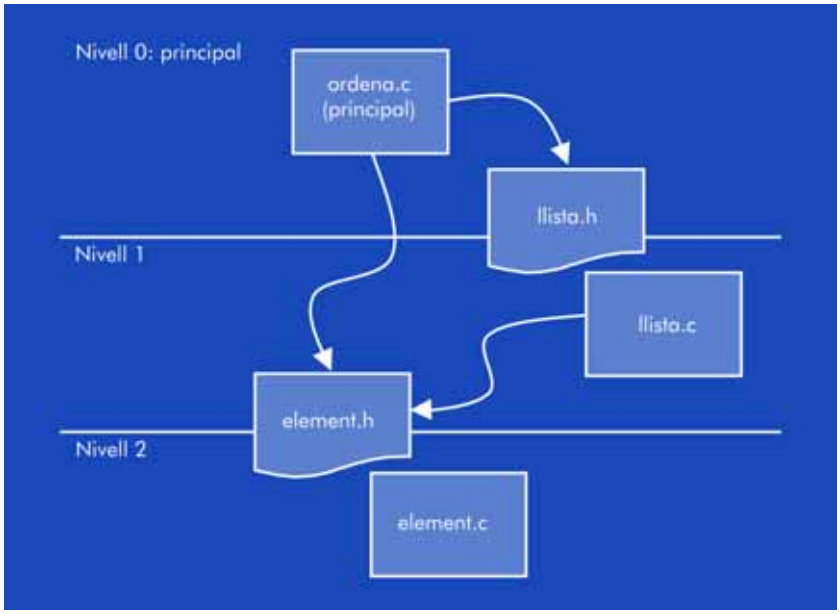
La inclusió es fa indicant el fitxer entre cometes dobles en lloc d'utilitzar angles (símbols de major i menor que) perquè se suposa que el fitxer que s'inclou es troba en el mateix directori que el fitxer que conté la directiva d'inclusió. Els angles s'han d'utilitzar si es requereix que el preprocessador examini el conjunt de camins d'accés a directoris estàndard de fitxers d'inclusió, com és el cas de `stdio.h`, per exemple.

3.8.2. Exemple

En l'exemple de l'ordenació per selecció, tindríem un fitxer de capçalera per a cada tipus de dades abstracte i, evidentment, els corresponents fitxers amb el codi en C. A més a més, disposaríem d'un tercer

fitxer que contindria el codi del programa principal. La figura següent reflecteix l'esquema de relacions entre aquests fitxers:

Figura 10.



Cada fitxer de codi font es pot compilar independentment. És a dir, en aquest exemple hi hauria tres unitats de compilació. Cadascuna es pot desenvolupar, doncs, independentment de les altres. L'únic que han de respectar les unitats que tenen un fitxer de capçalera és no modificar ni els tipus de dades ni les declaracions de les funcions. D'aquesta manera, les altres unitats que en facin ús no hauran de modificar les seves crides i, per tant, no requeriran cap canvi ni compilació.

És important tenir en compte que només hi pot haver una unitat amb una funció `main` (que és la que es correspon amb `ordena.c`, en l'exemple donat). Les altres hauran de ser compendis de funcions associades a un determinat tipus de dades abstracte.

L'ús de fitxers de capçalera permet, a més, canviar el codi d'alguna funció, sense haver de canviar, per això, res dels programes que l'utilitzen. És clar, sempre que el canvi no afecti el "contracte" establert en el fitxer de capçalera corresponent.

En el fitxer de capçalera es descriu tota la funcionalitat que es proveeix per a un determinat tipus de dades abstractes i,

ANOTACIONS

amb aquesta descripció, s'adquireix el compromís que es mantindrà independentment de com es materialitzi en el codi associat.

Per a il·lustrar aquesta idea es pot pensar que, en l'exemple, és possible canviar el codi de les funcions que treballen amb les llistes perquè siguin de tipus dinàmic sense canviar el contracte adquirit en el fitxer de capçalera.

A continuació es mostra el fitxer de capçalera per a les llistes en el cas de l'ordenació:

```
/* Fitxer: llista.h                                          */
#ifndef _LLISTA_VEC_H_
#define _LLISTA_VEC_H_

#include <stdio.h>
#include "bool.h"
#include "element.h"

#define LONG_MAXIMA 100
typedef struct llista_e {
   element_t      node[ LONG_MAXIMA ];
   unsigned short posicio;   /* Posició actual d'accés.     */
   unsigned short quantitat;/* Longitud de la llista.       */
} llista_t;

void inicialitza_llista( llista_t *ref_llista );
bool es_buida_llista( llista_t llista );
bool es_final_de_llista( llista_t llista );
void principi_de_llista( llista_t *llista_ref );
ref_node_t ref_node_de_llista( llista_t llista );
element_t element_en_ref_node(
   llista_t llista,
   ref_node_t refnode
);
void avanca_posicio_en_llista( llista_t *llista_ref );
element_t extreu_minim_de_llista( llista_t *ref_llista );
```

ANOTACIONS

```
void posa_al_final_de_llista(
   llista_t *ref_llista,
   element_t element
);

#endif /* _LLISTA_VEC_H_ */
```

Com es pot observar, es podria canviar la forma d'extracció de l'element mínim sense necessitat de modificar el fitxer de capçalera, i menys encara la crida en el programa principal. Tanmateix, un canvi en el tipus de dades, per exemple, per a implementar les llistes mitjançant variables dinàmiques, implicaria tornar a compilar totes les unitats que n'utilitzin, encara que no es modifiquin les capçaleres de les funcions. De fet, en aquests casos, és molt convenient mantenir el contracte, ja que evitarà modificacions en els codis font de les unitats que en facin ús.

3.9. Biblioteques

Les biblioteques de funcions són, de fet, unitats de compilació. Com a tals, cada una disposa d'un fitxer de codi font i un de capçalera. Per a evitar la compilació repetitiva de les biblioteques, el codi font ja ha estat compilat (fitxers d'extensió ".o") i només queden pendents del seu enllaç amb el programa principal.

Les biblioteques de funcions, de totes maneres, es distingeixen de les unitats de compilació perquè només s'incorporen dins del programa final aquelles funcions que són necessàries: les que no s'utilitzen, no s'inclouen. Els fitxers compilats que permeten aquesta opció tenen l'extensió ".l".

3.9.1. Creació

Per a obtenir una unitat de compilació de manera que només s'incloguin en el programa executable les funcions utilitzades, cal compilar-la i així s'obtindrà un fitxer de tipus objecte; és a dir, amb el codi executable sense enllaçar:

```
$ gcc -c -o biblioteca.o biblioteca.c
```

ANOTACIONS

Se suposa que en el fitxer de `biblioteca.c`, tal com s'ha indicat, hi ha una inclusió del fitxer de capçalera apropiat.

Una vegada generat el fitxer objecte, és necessari incloure'l en un arxiu (amb extensió ".a") de fitxers del mateix tipus (pot ser l'únic si la biblioteca consta d'una sola unitat de compilació). En aquest context, els "arxius" són col·leccions de fitxers objecte reunits en un únic fitxer amb un índex per a localitzar-los i, sobretot, per a determinar a quines parts del codi objecte corresponen determinades funcions. Per a crear un arxiu cal executar l'ordre següent:

```
$ ar biblioteca.a biblioteca.o
```

Per a construir l'índex (la taula de símbols i localitzacions) s'ha d'executar l'ordre:

```
$ ar - biblioteca.a
```

o bé:

```
$ ranlib biblioteca.a
```

L'ordre de gestió d'arxius `ar` permet, a més a més, crear llistes dels fitxers objecte que conté, afegir-ne o reemplaçar-ne d'altres de nous o modificats, actualitzar-ne (es duu a terme el reemplaçament si la data de modificació és posterior a la data d'inclusió en l'arxiu) i eliminar-ne si ja no són necessaris. Això es fa, respectivament, amb les ordres següents:

```
$ ar -t biblioteca.a
$ ar -r nou.o biblioteca.a
$ ar -u actualitzable.o biblioteca.a
$ ar -d obsolet.o biblioteca.a
```

Amb la informació de la taula de símbols, l'enllaçador munta un programa executable emprant només les funcions a les quals es refereix. Per a les altres qüestions, els arxius són similars als fitxers de codi objecte simples.

ANOTACIONS

3.9.2. Ús

L'ús de les funcions d'una biblioteca és exactament igual que el de les funcions de qualsevol altra unitat de compilació.

N'hi ha prou d'incloure el fitxer de capçalera apropiat i incorporar les crides a les funcions que es requereixin dins del codi font.

3.9.3. Exemple

En l'exemple de l'ordenació s'ha preparat una unitat de compilació per a les llistes. Com que les llistes són un tipus de dades dinàmic que s'utilitza molt, resulta convenient disposar d'una biblioteca de funcions per a operar-hi. D'aquesta manera, no se les ha de programar de nou en ocasions posteriors.

Per a transformar la unitat de llistes en una biblioteca de funcions de llistes cal fer que el tipus de dades no depengui en absolut de l'aplicació. En cas contrari, caldria compilar la unitat de llistes per a cada programa nou.

En l'exemple, les llistes contenien elements del tipus `element_t`, que era un apuntador a `dada_t`. En general, les llistes podran tenir elements que siguin apuntadors a qualsevol tipus de dades. En qualsevol cas, tot són adreces de memòria. Per aquest motiu, pel que fa a les llistes, els elements seran d'un tipus buit, que l'usuari de la biblioteca de funcions haurà de definir. Així doncs, en la unitat de compilació de les llistes s'inclou:

```
typedef void *element_t;
```

D'altra banda, per a fer la funció d'extracció de l'element més petit, ha de saber quina funció ha de cridar per fer-ne la comparació. Per tant, s'hi afegeix un paràmetre més que consisteix en l'adreça de la funció de comparació:

```
element_t extreu_minim_de_llista(
   llista_t *ref_llista,
   bool (*compara_elements)( element_t, element_t )
); /* extreu_minim_de_llista */
```

ANOTACIONS

El segon argument és un apuntador a una funció que pren com a paràmetres dos elements (no és necessari donar un nom als paràmetres formals) i torna un valor lògic de tipus `bool`. En el codi font de la definició de la funció, la crida s'efectuaria de la manera següent:

```
/* ... */
es_menor = (*compara_elements)(
       element_en_ref_node( llista, actual ),
       petit
); /* compara_elements */
/* ... */
```

Fora d'això, no s'hauria de fer cap canvi.

Així doncs, la decisió de transformar alguna unitat de compilació d'un programa en biblioteca dependrà fonamentalment de dos factors:

- El tipus de dades i les operacions s'han de poder emprar en altres programes.
- Rarament s'han d'emprar totes les funcions de la unitat.

3.10. Eina *make*

La compilació d'un programa representa, normalment, compilar algunes de les seves unitats i després enllaçar-les totes juntament amb les funcions de biblioteca que s'utilitzin per a muntar el programa executable final. Per tant, per a obtenir-lo, no solament cal dur a terme una sèrie d'ordres, sinó que també cal tenir en compte quins fitxers s'han modificat.

Les eines de tipus *make* permeten establir les relacions entre fitxers de manera que es pugui determinar quins depenen dels altres. Així, quan detecta que algun dels fitxers té una data i hora de modificació anterior a algun dels que depèn, s'executa l'ordre indicada per a generar-los de nou.

ANOTACIONS

D'aquesta manera, no cal preocupar-se de quins fitxers cal generar i quins no fa falta actualitzar. D'altra banda, evita haver d'executar individualment una sèrie d'ordres que, per a programes grans, pot ser considerable.

El propòsit de les eines de tipus *make* és determinar automàticament quines peces d'un programa s'han de recompilar i executar les ordres pertinents.

L'eina `gmake` (o, simplement, `make`) és una utilitat *make* de GNU (www.gnu.org/software/make) que s'ocupa del que s'ha esmentat abans. Per a poder-ho fer, necessita un fitxer que, per omissió, s'anomena `makefile`. És possible indicar-hi un fitxer amb un altre nom si se la invoca amb l'opció `-f`:

```
$ make -f fitxer_objectius
```

3.10.1. Fitxer *makefile*

En el fitxer *makefile* cal especificar els objectius (habitualment, fitxers per construir) i els fitxers dels quals depenen (els prerequisits per a complir els objectius). Per a cada objectiu és necessari construir una regla, l'estructura de la qual és la següent:

```
# Sintaxi d'una regla:
objectiu : fitxer1 fitxer2 ... fitxerN
   ordre1
   ordre2
   ...
   ordreK
```

El símbol # s'utilitza per a introduir una línia de comentaris. Tota regla requereix que s'indiqui quin és l'objectiu i, després dels dos punts, s'indiquin els fitxers dels quals depèn. En les línies següents s'indicaran les ordres que s'han d'executar. Qualsevol línia que

Nota

La primera d'aquestes línies ha de començar forçosament per un caràcter de tabulació.

es vulgui continuar haurà d'acabar amb un salt de línia precedit del caràcter d'*escape* o barra inversa (\).

Prenent com a exemple el programa d'ordenació de notes d'un examen per DNI, es podria construir un *makefile* com el mostrat a continuació per a simplificar l'actualització de l'executable final:

```
# Compilació del programa d'ordenació:
classifica : ordena.o nota.o llista.a
   gcc -g -o classifica ordena.o nota.o llista.a
ordena.o : ordena.c
   gcc -g -c -o ordena.o ordena.c
nota.o : nota.c nota.h
   gcc -g -c -o nota.o nota.c
llista.a : llista.o
   ar -r llista.a llista.o ;
   ranlib llista.a
llista.o : llista.c lista.h
   gcc -g -c -o llista.o llista.c
```

L'eina *make* processaria el fitxer anterior revisant els prerequisits del primer objectiu, i si són al seu torn objectius d'altres regles, es procedirà de la mateixa manera per a aquests últims. Qualsevol prerequisit terminal (que no sigui un objectiu d'alguna altra regla) modificat amb posterioritat a l'objectiu que afecta provoca l'execució en sèrie de les ordres especificades en les línies següents de la regla.

Es pot especificar més d'un objectiu, però `make` només examinarà les regles del primer objectiu final que trobi. Si es vol que processi altres objectius, serà necessari indicar-ho així quan s'invoqui.

És possible tenir objectius sense prerequisits, i en aquest cas sempre s'executaran les ordres associades a la regla en què apareixen.

És possible tenir un objectiu per a esborrar tots els fitxers objecte que ja no són necessaris.

ANOTACIONS

Exemple

Reprenent l'exemple anterior, podríem netejar el directori de fitxers innecessaris afegint el text següent al final del *makefile*:

```
# Neteja del directori de treball:
neteja :
   rm -f ordena.o nota.o llista.o
```

Per a aconseguir aquest objectiu, n'hi hauria prou d'introduir l'ordre:

```
$ make neteja
```

És possible indicar diversos objectius amb uns mateixos prerrequisitos:

```
# Compilació del programa d'ordenació:
tot depura optim : classifica
depura : CFLAGS := -g
optim : CFLAGS := -O
classifica : ordena.o nota.o llista.a
 gcc $(CFLAGS) -o classifica ordena.o nota.o llista.a
# resta del fitxer ...
```

Nota

Pel que fa al fragment anterior, podem veure el següent:

- Si s'invoca `make` sense argument, s'actualitzarà l'objectiu `tot` (el primer que troba), que depèn de l'actualització de l'objectiu secundari `classifica`.
- Si s'especifica l'objectiu `depura` o l'`optim`, haurà de comprovar la consistència de dues regles: en la primera s'indica que per a aconseguir-los cal actualitzar `classifica` i, en la segona, que cal assignar a la variable `CFLAGS` un valor.

ANOTACIONS

Atès que *make* primer analitza les dependències i després executa les ordres oportunes, el resultat és que el possible muntatge de *compila* es farà amb el contingut assignat a la variable `CFLAGS` en la regla precedent: l'accés al contingut d'una variable es fa mitjançant l'operador corresponent, que es representa pel símbol del dòlar.

En l'exemple anterior ja es pot apreciar que la utilitat de *make* va molt més enllà del que aquí hem explicat i, de la mateixa manera, els *makefile* tenen moltes maneres d'expressar regles de manera més potent. Tot i així, hem repassat les seves principals característiques des del punt de vista de la programació i n'hem vist alguns exemples significatius.

3.11. Relació amb el sistema operatiu. Pas de paràmetres a programes

Els programes es tradueixen a instruccions del llenguatge màquina per a ser executats. Tanmateix, moltes operacions relacionades amb els dispositius d'entrada, de sortida i d'emmagatzematge (discos i memòria, entre d'altres) es tradueixen en crides a funcions del sistema operatiu.

Una de les funcions del sistema operatiu és, precisament, permetre l'execució d'altres programes a la màquina i, en definitiva, del programari. Per a això, proveeix l'usuari de certs mecanismes perquè triï quines aplicacions vol executar. Actualment, la majoria són, de fet, interfícies d'usuari gràfiques. Tot i així, continuen existint els entorns textuals dels intèrprets d'ordres, anomenats de manera comuna *shells*.

En aquests intèrprets d'ordres, per a executar programes (alguns d'ells, utilitats del mateix sistema i d'altres, d'aplicacions) n'hi ha prou de subministrar-los el nom del fitxer de codi executable corresponent. Certament, també hi ha la possibilitat que un programa sigui invocat per un altre.

De fet, la funció `main` dels programes en C pot tenir diferents arguments. En particular, hi ha la convenció que el primer paràmetre si-

ANOTACIONS

gui el nombre d'arguments que s'hi passen a través del segon, que consisteix en un vector de cadenes de caràcters. Per a aclarir com funciona aquest procediment de pas de paràmetres, us posem com a exemple el programa següent, que ens descobrirà què succeeix quan l'invoquem des d'un intèrpret d'ordres amb un nombre determinat d'arguments:

```
/* Fitxer: args.c */
#include <stdio.h>
int main( int argc, char *argv[] )
{
   int i;
   printf( "Nre. arguments, argc = %i\n", argc );
   for( i = 0; i < argc; i = i + 1 ) {
     printf( "Argument argv[%i] = \"%s\"\n", i, argv[i] );
   } /* for */
   return argc;
} /* main */
```

Si es fa la invocació amb l'ordre següent, el resultat serà el que s'indica a continuació:

```
$ args -prova nombre 1

Núm. arguments, argc =4
Argument argv[0] = "args"
Argument argv[1] = "-prova"
Argument argv[2] = "nombre"
Argument argv[3] = "1"
$
```

És important tenir present que la mateixa invocació de programa pren com a argument 0 de l'orde. Així doncs, en l'exemple anterior, la invocació en què es donen tres paràmetres al programa es converteix, finalment, en una crida a la funció `main` en què s'adjuntarà també el text amb què ha estat invocat el mateix programa com a primera cadena de caràcters del vector d'arguments.

Es pot consultar el valor retornat per `main` per a determinar si hi ha hagut algun error durant l'execució o no. Generalment, es pren per

ANOTACIONS

conveni que el valor tornat ha de ser el codi de l'error corresponent o 0 en absència d'errors.

Nota

En el codi font de la funció és possible emprar les constants `EXIT_FAILURE` i `EXIT_SUCCESS`, que està definit a `stdlib.h`, per a indicar el retorn amb error o sense, respectivament.

En l'exemple següent, es mostra un programa que efectua la suma de tots els paràmetres presents en la invocació. Per a això, empra la funció `atof`, declarada a `stdlib.h`, que converteix cadenes de text en nombres reals. En cas que la cadena no representés un nombre, torna un zero:

Nota

En aquest cas, el programa torna sempre el valor 0, ja que no hi ha errors d'execució per indicar.

```
/* Fitxer: suma.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
   float    suma = 0.0;
   int      pos = 1;

   while( pos < argc ) {
     suma = suma + atof( argv[ pos ] );
     pos = pos + 1;
   } /* while */
   printf( " = %g\n", suma );
   return 0;
} /* main */
```

3.12. Execució de funcions del sistema operatiu

Un sistema operatiu és un programari que permet que les aplicacions que s'executen en un ordinador s'abstreguin dels detalls de la màquina. És a dir, les aplicacions poden treballar amb una màquina virtual que és capaç de fer operacions que la màquina real no pot entendre.

ANOTACIONS

A més a més, el fet que els programes es defineixin en termes de rutines (o funcions) subministrades per l'SO augmenta la seva portabilitat, la seva capacitat de ser executat en màquines diferents. En definitiva, els fa independents de la màquina, però no de l'SO, òbviament.

En C, moltes de les funcions de la biblioteca estàndard empren les rutines de l'SO per a dur a terme les seves tasques. Entre aquestes funcions hi ha les d'entrada i sortida de dades, de fitxers i de gestió de memòria (variables dinàmiques, sobretot).

Generalment, totes les funcions de la biblioteca estàndard de C tenen la mateixa capçalera i el mateix comportament, fins i tot amb independència del sistema operatiu; no obstant això, n'hi ha algunes que depenen del sistema operatiu: hi pot haver algunes diferències entre Linux i Microsoft. Afortunadament, són fàcilment perceptibles, ja que les funcions relacionades amb un determinat sistema operatiu estan declarades en fitxers de capçalera específics.

En tot cas, de vegades resulta convenient executar les ordres de l'intèrpret d'ordres del sistema operatiu en lloc d'executar directament les funcions per portar-les a terme. Això permet, entre altres coses, que el programa corresponent es pugui descriure a un nivell d'abstracció més alt i, amb això, aprofitar que sigui el mateix intèrpret d'ordres el que completi els detalls necessaris per a portar a terme la tasca encomanada. Generalment, es tracta d'executar ordres internes del mateix intèrpret d'ordres o aprofitar els seus recursos (camins de recerca i variables d'entorn, entre d'altres) per a executar altres programes.

Per a poder executar una ordre de l'intèrpret d'ordres, n'hi ha prou de subministrar a la funció `system` la cadena de caràcters que la descrigui. El valor que torna és el codi de retorn de l'ordre executada o –1 en cas d'error.

En Linux, `system( ordre )` executa `/bin/sh -c ordre`; és a dir, empra `sh` com a intèrpret d'ordres. Per tant, s'han d'ajustar a la seva sintaxi.

ANOTACIONS

En el programa següent mostra el codi tornat per l'execució d'una ordre, que cal introduir entre cometes com a argument del programa:

```
/* Fitxer: executa.c */
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] )
{
   int codi;
   if( argc == 2 ) {
     codi = system( argv[1] );
     printf( "%i = %s\n", codi, argv[1] );
   } else {
     printf( "Us: executa \"ordre\"\n" );
   } /* if */
   return 0;
} /* main */
```

Encara que simple, el programa anterior ens dóna una certa idea de com s'ha d'emprar la funció `system`.

El conjunt de rutines i serveis que ofereix el sistema operatiu va més enllà de donar suport a les funcions d'entrada i sortida de dades, de maneig de fitxers i de gestió de memòria, ja que també permet llançar l'execució de programes dins d'altres. En l'apartat següent, es tractarà amb més profunditat el tema de l'execució de programes.

3.13. Gestió de processos

Els sistemes operatius actuals són capaços, a més de tot el que hem vist, d'executar una diversitat de programes en la mateixa màquina en un mateix període de temps. Evidentment, això només és possible si s'executen en processadors diferents o si la seva execució es fa de manera seqüencial o intercalada en un mateix processador.

ANOTACIONS

Normalment, malgrat que el sistema operatiu sigui capaç de gestionar una màquina amb diversos processadors, hi haurà més programes per executar que recursos per a fer-ho. Per aquest motiu, sempre haurà de poder planificar l'execució d'un conjunt de programes determinat en un processador únic. La planificació pot ser:

- Seqüencial. Una vegada finalitzada l'execució d'un programa, s'inicia la del programa següent (també es coneix com a *execució per lots*).
- Intercalada. Cada programa disposa d'un temps determinat en què es duu a terme una part del seu flux d'execució d'instruccions i, al final del període de temps assignat, s'executa una part d'un altre programa.

D'aquesta manera, es poden executar diversos programes en un mateix interval de temps, cosa que fa la sensació que la seva execució progressa paral·lelament.

Recolzant-se en els serveis (funcions) que ofereix l'SO respecte a l'execució del programari, és possible, entre altres coses, executar-lo dissociat de l'entrada/sortida estàndard i/o partir el seu flux d'execució d'instruccions en diversos paral·lels.

3.13.1. Definició de procés

Pel que fa al sistema operatiu, cada flux d'instruccions que ha de gestionar és un procés. Per tant, repartirà la seva execució entre els diversos processadors de la màquina i en el temps perquè duguin a terme les seves tasques progressivament. Hi haurà, això sí, alguns processos que es dividiran en dos de paral·lels, és a dir, el programa consistirà, a partir d'aquell moment, en dos processos diferents.

Cada procés té associat, en iniciar-se, una entrada i una sortida estàndard de dades de la qual es pot dissociar per continuar l'execució en segon pla, una cosa habitual en els processos permanents, que tractem en l'apartat següent.

Els processos que comparteixen un mateix entorn o estat, amb excepció evident de la referència a la instrucció següent, són denominats

ANOTACIONS

fils o *brins* (en anglès, *threads*), mentre que els que tenen entorns diferents són denominats simplement *processos*.

Amb tot, un programa pot organitzar el seu codi de manera que porti a terme la seva tasca mitjançant diversos fluxos d'instruccions paral·lels, tant si són simples fils com processos complets.

3.13.2. Processos permanents

Un procés permanent és aquell que s'executa indefinidament en una màquina. Solen ser processos que s'ocupen de la gestió automatitzada d'entrades i sortides de dades i, per tant, amb escassa interacció amb els usuaris.

Així doncs, moltes de les aplicacions que funcionen amb el model client/servidor es construeixen amb processos permanents per al servidor i amb processos interactius per als clients. Un exemple clar d'aquestes aplicacions són les relacionades amb Internet: els clients són programes, com el gestor de correu electrònic o el navegador, i els servidors són programes que atenen les peticions dels clients corresponents.

Nota

Literalment, un dimoni és un esperit maligne, encara que se suposa que els processos anomenats com a tals no ho haurien de ser.

En Linux, als processos permanents se'ls anomena, gràficament, *dimonis* (*daemons*, en anglès) perquè encara que els usuaris no els poden veure, ja que no hi interactuen (en especial, no ho fan per mitjà del terminal estàndard), existeixen: els dimonis són "esperits" de la màquina que l'usuari no veu però dels quals percep els efectes.

Per a crear un dimoni, n'hi ha prou de cridar la funció `daemon`, declarada a `unistd.h`, amb els paràmetres adequats. El primer argument indica si no canvia de directori de treball i, el segon, si no es dissocia del terminal estàndard d'entrada/sortida; és a dir, una crida comuna hauria de ser com segueix:

```
/* ... */
if( daemon( FALSE, FALSE ) == 0 ) {
   /* cos */
}  /* if */
/* resta del programa, tant si s'ha creat com si no.    */
```

ANOTACIONS

Nota

Aquesta crida aconsegueix que el cos del programa sigui un dimoni que treballa en el directori arrel (com si hagués fet un `cd /`) i que està dissociat de les entrades i sortides estàndard (en realitat, redirigides al dispositiu buit: `/dev/null`). La funció torna un codi d'error, que és zero si tot ha anat bé.

Exemple

Per a il·lustrar el funcionament dels dimonis, es mostra un programa que avisa l'usuari que ha transcorregut un temps determinat. Per a això, caldrà invocar al programa amb dos paràmetres: un per a indicar les hores i minuts que han de transcórrer abans d'advertir a l'usuari i un altre que contingui el text de l'avís. En aquest cas, el programa es convertirà en un dimoni no dissociat del terminal d'entrada/sortida estàndard, ja que l'avís hi apareixerà.

```
/* Fitxer: alarma.c                                    */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
   unsigned int hores;
   unsigned int minuts;
   unsigned int segons;
   cha           *avis, *separador;
   if( argc == 3 ) {
     separador = strchr( argv[1], ': ' );
     if( separador != NULL ) {
       hores = atoi( argv[1] );
       minuts = atoi( separador+1 );
     } else {
       hores = 0;
       minuts = atoi( argv[1] );
     } /* if */
     segons = (hores*60 + minuts) * 60;
```

ANOTACIONS

```
    avis = argv[2];
    if( daemon( FALSE, TRUE ) ) {
     printf( "No es pot installar l'avisador :-(\n);
    } else {
     printf( "Alarma dins de %i hores i %i minuts.\n",
      hores, minuts
     ); /* printf */
     printf( "Fes $ kill %li per a desconnectar-la.\n",
      getpid()
     ); /* printf */
    } /* if */
    sleep( segons );
    printf( "%s\007\n", avis );
    printf( "Alarma desconectada.\n" );
   } else {
    printf( "Us: %s hores:minuts \"avís\"\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

La lectura dels paràmetres d'entrada ocupa bona part del codi. En particular, el que necessita més atenció és l'extracció de les hores i dels minuts; per a això, es busquen els dos punts (amb `strchr`, declarada a `string.h`) i després pren la cadena sencera per determinar el valor de les hores i la cadena a partir dels dos punts per als minuts.

L'espera es fa mitjançant una crida a la funció `sleep`, que té com a argument el nombre de segons en què el programa ha "de dormir", és a dir, suspendre la seva execució.

Finalment, per a donar a l'usuari la possibilitat de parar l'alarma, se l'informa de l'ordre que ha d'introduir en l'intèrpret d'ordres per "matar" el procés (és a dir, per finalitzar-ne l'execució). Amb aquesta finalitat, s'hi mostra el nombre de procés que es correspon amb el dimoni instal·lat. Aquest identificador s'aconsegueix cridant la funció `getpid()`, en la qual PID significa, precisament, identificador de procés.

Un dels usos fonamentals dels dimonis és el de la implementació de processos proveïdors de serveis.

ANOTACIONS

3.13.3. Processos concurrents

Els processos concurrents són els que s'executen simultàniament en un mateix sistema. En dir *simultàniament*, en aquest context, entenem que es duen a terme en un mateix període de temps o bé en processadors diferents, o bé repartits temporalment en un mateix processador, o bé en els dos casos anteriors.

El fet de repartir l'execució d'un programa en diversos fluxos d'instruccions concurrents pot perseguir algun dels objectius següents:

- Aprofitar els recursos en un sistema multiprocessador. En executar-se cada flux d'instruccions en un processador diferent s'aconsegueix una rapidesa d'execució més gran. De fet, només en aquest cas es tracta de processos d'execució veritablement simultània.

  Nota

  Quan dos o més processos comparteixen un mateix processador, no hi ha altre remei que executar-los per trams en un determinat període de temps dins del qual, efectivament, se'n pot observar una evolució progressiva.

- Augmentar el rendiment respecte de l'entrada/sortida de dades del programa. Per a aconseguir-ho, pot resultar convenient que un dels processos s'ocupi de la relació amb l'entrada de dades, un altre del càlcul que calgui fer amb ells i, finalment, un altre s'encarregui de la sortida de resultats. D'aquesta manera, es pot fer el càlcul sense aturar-se a donar sortida als resultats o esperar dades d'entrada. Certament, no sempre es pot fer aquesta partició i el nombre de processos pot variar molt segons les necessitats del programa. De fet, en aquest cas es tracta, fonamentalment, de separar processos de naturalesa lenta (per exemple, els que s'han de comunicar amb d'altres sia per rebre bé sia per transmetre dades) d'altres processos més ràpids, és a dir, amb més atenció al càlcul.

En els apartats següents comentem diversos casos de programació concurrent tant amb "processos lleugers" (els fils) com amb processos complets o "pesants" (per contraposició als lleugers).

ANOTACIONS

3.14. Fils

Un fil, bri o *thread* és un procés que comparteix l'entorn amb d'altres del mateix programa, cosa que comporta que l'espai de memòria sigui el mateix. Per tant, la creació d'un fil nou només implica disposar d'informació sobre l'estat del processador i la instrucció següent per a aquest. Precisament per aquest motiu s'anomenen *processos lleugers*.

Els fils són fluxos d'execució d'instruccions independents que tenen molta relació entre ells.

Per a emprar-los en C sobre Linux, és necessari fer crides a funcions de fils de l'estàndard de POSIX. Aquest estàndard defineix una interfície portable de sistemes operatius (originalment, Unix) per a entorns de computació, de l'expressió en anglès dels quals pren l'acrònim.

Les funcions de POSIX per a fils estan declarades en el fitxer `pthread.h` i s'ha d'enllaçar l'arxiu de la biblioteca corresponent amb el programa. Per a això, cal compilar amb l'ordre següent:

```
$ gcc -o executable codi.c -lpthread
```

Nota

L'opció `-lpthread` indica a l'enllaçador que ha d'incloure també la biblioteca de funcions POSIX per a fils.

3.14.1. Exemple

Mostrarem un programa per determinar si un nombre és primer o no com a exemple d'un programa desenfilat en dos fils. El fil principal s'ocuparà de buscar possibles divisors mentre que el secundari actuarà d'"observador" per a l'usuari: llegirà les dades que maneja el fil principal per mostrar-los pel terminal de sortida estàndard. Evidentment, això és possible perquè comparteixen el mateix espai de memòria.

La creació dels fils requereix que el seu codi sigui dins d'una funció que només admet un paràmetre del tipus `(void *)`. De fet, les funcions creades per a ser fils POSIX han d'obeir a la capçalera següent:

```
(void *)fil( void *referencia_parametres );
```

Així doncs, serà necessari col·locar en un tuple tota la informació que es vulgui fer visible a l'usuari i passar la seva adreça com a paràmetre seu. A continuació us definim el tuple d'elements que es mostrarà:

```
/* ... */
typedef struct s_visible {
   unsigned long nombre;
   unsigned long divisor;
   bool fi;
} t_visible;
/* ... */
```

Nota

El camp `fi` servirà per a indicar al fil fill que el fil principal (el pare) ha acabat la seva tasca. En aquest cas, ha determinat si el nombre és primer o no.

La funció del fil fill serà la següent:

```
/* ... */
void *observador( void *parametre )
{
   t_visible *ref_vista;

   ref_vista = (t_visible *)parametre;
   printf( "... provant %012lu", 0 );
   do {
     printf( "\b\b\b\b\b\b\b\b\b\b\b\b" );
     printf( "%12lu", ref_vista->divisor );
   } while( !(ref_vista->fi) );
   printf( "\n" );
   return NULL;
} /* observador */
/* ... */
```

ANOTACIONS

Nota

El caràcter '\b' es correspon amb un retrocés. Atès que els nombres s'imprimeixen amb dotze dígits (els zeros a l'esquerra es mostren com a espais), la impressió de dotze retrocessos implica esborrar el nombre que s'hagi escrit anteriorment.

Per a crear el fil de l'observador, n'hi ha prou de cridar `pthread_create()` amb els arguments adequats. A partir d'aquest moment, un nou fil s'executa concurrentement amb el codi del programa:

```
/* ... */
int main( int argc, char *argv[] )
{
   int         codi_error;     /* Codi d'error que s'ha de tornar. */
   pthread_t   id_fil;         /* Identificador del fil.           */
   t_visible   vista;          /* Dades observables.               */
   bool        resultat;       /* Indicador de si és primer.       */
{
   if( argc == 2 ) {
    vista.nombre = atol( argv[1] );
    vista.fi = FALSE;
    codi_error = pthread_create(
     &id_fil,           /* Referència en la qual posar l'ID.     */
     NULL,              /* Referència a possibles atributs.      */
     observador,        /* Funció que executarà el fil.          */
     (void *)&vista     /* Argument de la funció.                */
    ); /* pthread_create */
    if( codi_error == 0 ) {
     resultat = es_primer( &vista );
     vista.fi = TRUE;
     pthread_join( id_fil, NULL );
     if( resultat )  printf( "És primer.\n" );
     else            printf( "No és primer.\n" );
     codi_error = 0;
    } else {
     printf( "No he pogut crear un fil observador!\n" );
```

```
      codi_error = 1;
    } /* if */
  } else {
    printf( "Us: %s nombre\n", argv[0] );
    codi_error = -1;
  } /* if */
  return codi_error;
} /* main */
```

Nota

Després de crear el fil de l'observador, es comprova que el nombre sigui primer i, en tornar de la funció `es_primer()`, es posa el camp `fi` a `TRUE` perquè l'observador finalitzi la seva execució. Per a esperar que efectivament hagi acabat, es crida `pthread_join()`. Aquesta funció espera que el fil l'identificador del qual s'hagi donat com a primer argument arribi al final de la seva execució i, per tant, es produeixi una unió dels fils (d'aquí ve l'apel·latiu en anglès *join*). El segon argument s'utilitza per a recollir possibles dades tornades pel fil.

Per acabar l'exemple, seria necessari codificar la funció `es_primer()`, que tindria com a capçalera la següent:

```
/* ... */
bool es_primer( t_visible *ref_dades );
/* ... */
```

La programació es deixa com a exercici. Per a resoldre'l adequadament, es pot tenir present que cal emprar `ref_dades->divisor` com a tal, ja que la funció `observador()` la llegeix per mostrar-la a l'usuari.

En aquest cas, no hi ha cap problema que els fils d'un programa tinguin el mateix espai de memòria; és a dir, el mateix entorn o context. De totes maneres, sol ser habitual que l'accés a dades compartides per més d'un fil sigui sincronitzat. En altres paraules,

ANOTACIONS

que s'habiliti algun mecanisme per a impedir que dos fils accedeixin simultàniament a la mateixa dada, especialment per modificar-la, encara que també perquè els fils lectors llegeixin les dades degudament actualitzades. Aquests mecanismes d'exclusió mútua són, de fet, convenis de crides a funcions prèvies i posteriors a l'accés a les dades.

Nota

Es pot imaginar com a funcions de control d'un semàfor d'accés a una plaça d'aparcament: si està lliure, el semàfor estarà en verd i, si està ocupada, estarà en vermell fins que s'alliberi.

3.15. Processos

Un procés és un flux d'execució d'instruccions amb un entorn propi i, per tant, amb totes les atribucions d'un programa. Pot dividir, doncs, el flux d'execució d'instruccions en altres processos (lleugers o no) si així es considera convenient per raons d'eficiència.

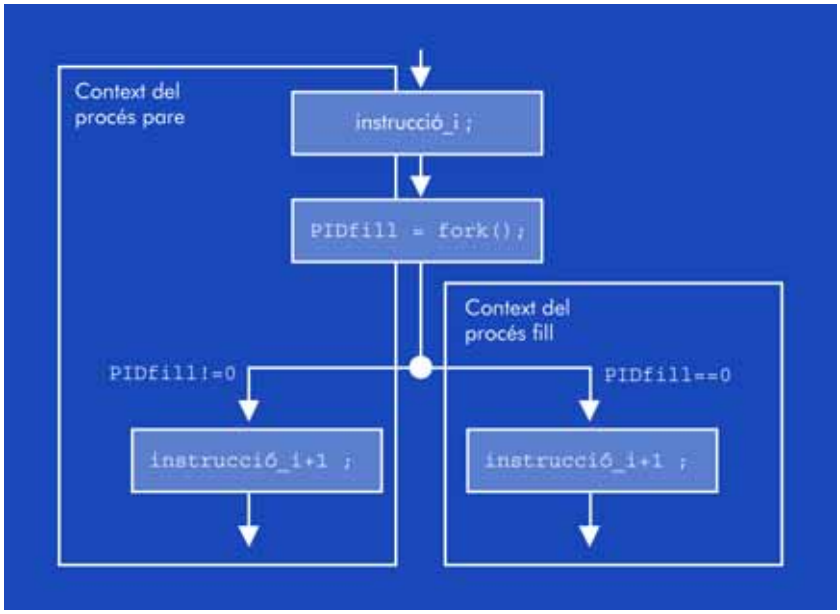
En aquest cas, la generació d'un nou procés a partir del procés principal implica fer la còpia de tot l'entorn d'aquest últim. Amb això, el procés fill té una còpia exacta de l'entorn del pare en el moment de la divisió. A partir d'aquí, tant els continguts de les variables com la indicació de la instrucció següent pot divergir. De fet, actuen com a dos processos diferents amb entorns evidentment diferents del mateix codi executable.

Atesa la separació estricta dels entorns dels processos, generalment es divideixen quan cal fer una mateixa tasca sobre dades diferents, que duu a terme cadascun dels processos fill de manera autònoma. D'altra banda, també hi ha mecanismes per a comunicar processos entre ells: les canonades, les cues de missatges, les variables compartides (en aquest cas, hi ha funcions per a implementar l'exclusió mútua) i qualsevol altre tipus de comunicació que es pugui establir entre processos diferents.

ANOTACIONS

Per a crear un procés nou, n'hi ha prou de cridar `fork()`, la declaració del qual es troba a `unistd.h`, i que torna l'identificador del procés fill en el pare i zero en el procés nou:

Figura 11.



Nota

El fet que `fork()` torni valors diferents en el procés pare i en el fill permet als fluxos d'instruccions següents que determinin si pertanyen a un o a un altre.

El programa següent és un exemple simple de divisió d'un procés en dos: l'original o pare i la còpia o fill. Per simplificar, se suposa que tant el fill com el pare fan una mateixa tasca. En aquest cas, el pare espera que el fill finalitzi l'execució amb `wait()`, que requereix la inclusió dels fitxers `sys/types.h` i `sys/wait.h`:

```
/* Fitxer: ex_fork.c                                    */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* ... */
```

ANOTACIONS

```
int main( void )
{
   pid_t  proces;
   int    estat;
   printf( "Procés pare (%li) iniciat.\n", getpid() );
   proces = fork();
   if( proces == 0 ) {
    printf( "Procés fill (%li) iniciat.\n", getpid() );
    tasca( "fill" );
    printf( "Final del procés fill.\n" );
   } else {
    tasca( "pare" );
    wait( &estat ); /* Espera l'acabament del fill.   */
    printf( "Final del procés pare.\n" );
   } /* if */
   return 0;
} /* main */
```

Per a posar de manifest que es tracta de dos processos que s'executen paral·lelament, resulta convenient que les tasques del pare i del fill siguin diferents o que aconsegueixin dades d'entrada diferents.

Per a il·lustrar aquest cas, es mostra una possible programació de la funció `tasca` en què es fa una repetició d'esperes. Per a poder observar que l'execució dels dos processos pot no estar intercalada sempre de la mateixa manera, tant el nombre de repeticions com el temps de les esperes es posa en funció de nombres aleatoris proveïts per la funció `random()`, que arrenca amb un valor "llavor" calculat mitjançant `srandom()` amb un argument que varïi amb les diferents execucions i amb el tipus de procés (pare o fill):

```
/* ... */
void tasca( char *nom )
{
   unsigned int comptador;

   srandom( getpid() % ( nom[0] * nom[2]) );
   comptador = random() % 11 + 1;
```

ANOTACIONS

```
   while( comptador > 0 ) {
    printf( "... pas %i del %s\n", comptador, nom );
    sleep( random() % 7 + 1 );
    comptador = comptador - 1;
   } /* while */
} /* tasca */
/* ... */
```

En l'exemple anterior, el pare espera un únic fill i fa una mateixa tasca. Això, evidentment, no és l'habitual. És molt més comú que el procés pare s'ocupi de generar un procés fill per a cada conjunt de dades que s'ha de processar. En aquest cas, el programa principal es complica lleugerament i cal seleccionar en funció del valor tornat per `fork()` si les instruccions pertanyen al pare o a un dels fills.

Per a il·lustrar la codificació d'aquests programes, es mostra un programa que pren com a arguments una quantitat indeterminada de nombres naturals per als quals esbrina si es tracta de nombres primers o no. En aquest cas, el programa principal crearà un procés fill per a cada nombre natural que s'ha de tractar:

```
/* ... */
int main( int argc, char *argv[] )
{
   int                comptador;
   unsigned long int  nombre, divisor;
   pid_t              proces;
   int                estat;

   if( argc > 1 ) {
    proces = getpid();
    printf( "Procés %li iniciat.\n", proces );
    comptador =1;
    while( proces != 0 && comptador < argc ) {
     /* Creació de processos fill:                */
     nombre = atol( argv[ comptador ] );
     comptador = comptador + 1;
     proces = fork();
     if( proces == 0 ) {
      printf( "Proces %li per a %lu\n",
       getpid(),
       nombre
```

ANOTACIONS

```
      ); /* printf */
      divisor = es_primer( nombre );
      if( divisor > 1 ) {
       printf( "%lu no es primer.\n", nombre );
       printf("El seu primer divisor és %lu\n", divisor );
      } else {
       printf( "%lu és primer.\n", nombre );
      } /* if */
     } /* if */
    } /* while */
    while( proces != 0 && comptador > 0 ) {
     /* Espera d'acabament de processos fill:          */
     wait( &estat );
     comptador = comptador - 1;
    } /* while */
    if( proces !=0 ) printf( "Fi.\n");
   } else {
    printf( "Us: %s natural_1 ... natural_N\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

Nota

El bucle de creació s'interromp si `proces==0` per evitar que els processos fill puguin crear "néts" amb les mateixes dades que alguns dels seus "germans".

Cal tenir present que el codi del programa és el mateix tant per al procés pare com per al fill.

D'altra banda, el bucle final d'espera només s'ha d'aplicar al pare, és a dir, al procés en què es compleixi que la variable `proces` sigui diferent de zero. En aquest cas, n'hi ha prou de descomptar del comptador de processos generats una unitat per a cada espera complerta.

Per a comprovar el seu funcionament, falta dissenyar la funció `es_primer()`, que queda com a exercici. Si es vol veure el funcionament d'aquest programa de manera exemplar, és convenient introduir algun nombre primer gran juntament amb altres de menors o no primers.

ANOTACIONS

3.15.1. Comunicació entre processos

Tal com s'ha comentat, els processos (tant si són d'un mateix programa com de programes diferents) es poden comunicar entre ells mitjançant mecanismes de canonades, cues de missatges i variables compartides, entre d'altres. Per tant, aquests mecanismes també es poden aplicar a la comunicació entre programes diferents d'una mateixa aplicació o, fins i tot, d'aplicacions diferents. En tot cas, sempre es tracta d'una comunicació poc intensa i que requereix exclusió mútua en l'accés a les dades per a evitar conflictes (tot i així, no sempre s'eviten tots).

Com a exemple, es mostrarà un programa que descompon en suma de potències de divisors primers qualsevol nombre natural donat. Per a això, disposa d'un procés de càlcul de divisors primers i un altre, el pare, que els mostra a mesura que es van calculant. Cada factor de la suma és una dada del tipus:

```
typedef struct factor_s {
   unsigned long int divisor;
   unsigned long int potencia;
} factor_t;
```

La comunicació entre ambdós processos es fa mitjançant una canonada.

Nota

Recordeu que en la unitat anterior ja s'ha definit *canonada*. Una canonada consisteix, de fet, en dos fitxers de flux de bytes, un d'entrada i un altre de sortida, pels quals es comuniquen dos processos diferents.

Com es pot apreciar en el codi següent, la funció per a obrir una canonada s'anomena `pipe()` i pren com a argument l'adreça d'un vector de dos enters on dipositarà els descriptors dels fitxers de tipus *stream* que hagi obert: en la posició 0 el de sortida, i en l'1, el d'entrada. Després del `fork()`, tots dos processos tenen una còpia dels descriptors i, per tant, poden accedir als mateixos fitxers tant per a l'entrada com per a la sortida de dades. En aquest cas, el procés fill

ANOTACIONS

tancarà el fitxer d'entrada i el pare, el de sortida; ja que la canonada només comunicarà els processos en un únic sentit: de fill a pare. (Si la comunicació es fes en tots dos sentits, seria necessari establir un protocol d'accés a les dades per a evitar conflictes.):

```
/* ... */
int main( int argc, char *argv[] )
{
   unsigned long int   nombre;
   pid_t               proces;
   int                 estat;
   int                 desc_canonada[2];

   if( argc == 2 ) {
    printf( "Divisors primers.\n" );
    nombre = atol( argv[ 1 ] );
    if( pipe( desc_canonada ) != -1 ) {
     proces = fork();
     if( proces == 0 ) { /* Proces fill:       */
      close( desc_canonada[0] );
      divisors_de( nombre, desc_canonada[1] );
      close( desc_canonada[1] );
     } else { /* Proces principal o pare:      */
      close( desc_canonada[1] );
      mostra_divisors( desc_canonada[0] );
      wait( &estat );
      close( desc_canonada[0] );
      printf( "Fi.\n" );
     } /* if */
    } else {
     printf( "No puc crear la canonada!\n" );
    } /* if */
   } else {
    printf( "Us: %s nombre_natural\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

Amb tot, el codi de la funció `mostra_divisors()` en el procés pare podria ser com el que es mostra a continuació. S'hi empra la funció de lectura `read()`, que intenta llegir un determinat nombre

ANOTACIONS

de bytes del fitxer el descriptor del qual s'hi passa com a primer argument. Torna el nombre de bytes efectivament llegits i el seu contingut el diposita a partir de l'adreça de memòria indicada:

```
/* ... */
void mostra_divisors( int desc_entrada )
{
   size_t     nbytes;
   factor_t   factor;

   do {
    nbytes = read( desc_entrada,
      (void *)&factor,
      sizeof( factor_t )
    ); /* read */
    if( nbytes > 0 ) {
     printf( "%lu ^ %lu\n",
       factor.divisor,
       factor.potencia
     ); /* printf */
   } while( nbytes > 0 );
} /* mostra_divisors */
/* ... */
```

Per a completar l'exemple, es mostra una possible programació de la funció `divisors_de()` en el procés fill. Aquesta funció empra `write()` per a dipositar els factors acabats de calculats en el fitxer de sortida de la canonada:

```
/* ... */
void divisors_de(
   unsigned long   int nombre,
   int             desc_sortida )
{
   factor_t f;

   f.divisor = 2;
   while( nombre > 1 ) {
    f.potencia = 0;
```

ANOTACIONS

```
    while( nombre % f.divisor == 0 ) {
      f.potencia = f.potencia + 1;
      nombre = nombre / f.divisor;
    } /* while */
    if( f.potencia > 0 ) {
      write( desc_sortida, (void *)&f, sizeof(factor_t) );
    } /* if */
    f.divisor = f.divisor + 1;
  } /* while */
} /* divisors_de */
/* ... */
```

Amb aquest exemple s'ha mostrat una de les possibles formes de comunicació entre processos. En general, cada mecanisme de comunicació té uns usos preferents.

Nota

Les canonades són adequades per al pas d'una quantitat relativament alta de dades entre processos, mentre que les cues de missatges s'adapten millor a processos que es comuniquen poc sovint o de manera irregular.

En tot cas, cal tenir present que repartir les tasques d'un programa en diversos processos comportarà un increment determinat de la complexitat per la necessària introducció de mecanismes de comunicació entre ells. Així doncs, és important valorar els beneficis que aquesta divisió pugui aportar al desenvolupament del programa corresponent.

3.16. Resum

Els algoritmes que s'empren per a processar la informació poden ser més o menys complexos segons la representació que s'esculli. Com a conseqüència, l'eficiència de la programació està directament relacionada amb les estructures de dades que s'hi emprin.

ANOTACIONS

Per aquest motiu s'han introduït les estructures dinàmiques de dades, que permeten, entre altres coses, aprofitar millor la memòria i canviar la relació entre ells com a part del processament de la informació.

Les estructures de dades dinàmiques són, doncs, aquelles en què el nombre de dades pot variar durant l'execució del programa i les relacions de les quals, evidentment, poden canviar. Per a això, es recolzen en la creació i destrucció de variables dinàmiques i en els mecanismes per accedir-hi. Fonamentalment, l'accés a aquestes variables s'ha de fer mitjançant apuntadors, ja que les variables dinàmiques no disposen de noms amb què identificar-les.

S'ha vist també un exemple comú d'estructures de dades dinàmiques com les cadenes de caràcters i les llistes de nodes. En particular, per a aquest últim cas s'ha revisat no solament la possible programació de les funcions de gestió dels nodes en una llista, sinó també una forma especial de tractament en què s'empren com a representacions de cues.

Atès l'ús habitual de moltes d'aquestes funcions per a estructures de dades dinàmiques comunes, resulta convenient agrupar-les en arxius de fitxers objecte: les biblioteques de funcions. D'aquesta manera, és possible emprar les mateixes funcions en programes diversos sense preocupar-se de la seva programació. Tot i així, és necessari incloure els fitxers de capçalera per a indicar al compilador la manera d'invocar aquestes funcions. Amb tot, es repassa el mecanisme de creació de biblioteques de funcions i, a més, s'introdueix l'ús de la utilitat *make* per a la generació d'executables que resulten de la compilació de diverses unitats del mateix programa i dels arxius de biblioteca requerits.

D'altra banda, també s'ha vist com la relació entre els diferents tipus de dades abstractes d'un programa faciliten la programació modular. De fet, aquests tipus es classifiquen segons nivells d'abstracció o, segons com, de dependència d'altres tipus de dades. Així doncs, el nivell més baix d'abstracció el tenen els tipus de dades abstractes que es defineixen en termes de tipus de dades primitives.

D'aquesta manera, el programa principal serà aquell que operi amb els tipus de dades de més nivell d'abstracció. La resta de mòduls del

ANOTACIONS

programa seran els que proveeixin el programa principal de les funcions necessàries per a executar aquestes operacions.

Per tant, el disseny descendent d'algoritmes, basat en la jerarquia que s'estableix entre els diferents tipus de dades que empren, és una tècnica amb la qual s'obté una programació modular eficient.

A la pràctica, cada tipus de dades abstracte s'haurà d'acompanyar de les funcions per a operacions elementals com creació, accés a dades, còpia, duplicat i destrucció de les variables dinàmiques corresponents. Més encara, haurà d'estar continguda en una unitat de compilació independent, juntament amb el fitxer de capçalera adequat.

Finalment, en l'última unitat, s'ha insistit en l'organització del codi, no tant amb relació a la informació que ha de processar, sinó més amb relació a la manera de fer-ho. En aquest sentit, resulta convenient aprofitar al màxim les facilitats que ens ofereix el llenguatge de programació C per a utilitzar les rutines de servei del sistema operatiu.

Quan la informació que s'ha de tractar hagi de ser processada per un altre programa, és possible executar-los des del flux d'execució d'instruccions del que s'està executant. En aquest cas, tanmateix, la comunicació entre el programa cridat i el cridador és mínima. Com a conseqüència, ha de ser el mateix programa cridat el que obtingui la major part de la informació que s'ha de tractar i el que generi el resultat.

S'ha tractat també de la possibilitat de dividir el flux d'execució d'instruccions en diversos fluxos diferents que s'executen concurrentment. D'aquesta manera, és possible que cada flux s'especialitzi en un aspecte determinat del tractament de la informació o, en altres casos, fer el mateix tractament sobre parts diferents de la informació.

Els fluxos d'execució d'instruccions es poden dividir en fils o processos. Als primers també se'ls anomena *processos lleugers*, ja que són processos que comparteixen el mateix context (entorn) d'execució. El tipus de tractament de la informació serà el que determini quina forma de divisió és la millor. Com a norma pot prendre la del grau de compartiment de la informació: si és alt, llavors és millor un fil, i si és baix, un procés (entre ells, no obstant això, hi ha diver-

ANOTACIONS

sos mecanismes de comunicació segons el grau particular de relació que tinguin).

En tot cas, part del contingut d'aquest capítol es veurà de nou en els propers, ja que tant C++ com Java faciliten la programació amb tipus de dades abstractes, el disseny modular i la distribució de l'execució en diversos fluxos d'instruccions.

3.17. Exercicis d'autoavaluació

1) Feu un cercador de paraules en fitxers, de manera similar a l'últim exercici del capítol anterior. El programa haurà de demanar el nom del fitxer i la paraula que s'ha de buscar. En aquest cas, la funció principal haurà de ser la següent:

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

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
typedef char *paraula_t;
paraula_t seguent_paraula(
   char *frase,
   unsigned int inici
) { /* ... */ }
int main( void )
{
   FILE          *entrada;
   char          nom[BUFSIZ];
   paraula_t     paraula, paraula2;
   unsigned int  numlin, pos;

   printf( "Busca paraules.\n" );
   printf( "Fitxer: " );
   gets( nom );
   entrada = fopen( nom, "rt" );
   if( entrada != NULL ) {
     printf( "Paraula: " );
     gets( nom );
```

ANOTACIONS

```
    paraula = seguent_paraula( nom, 0 );
    printf( "Buscant %s en fitxer...\n", paraula );
    numlin = 1;
    while( fgets( nom, BUFSIZ-1, entrada ) != NULL ) {
     numlin = numlin + 1;
     pos = 0;
     paraula2 = seguent_paraula( nom, pos );
     while( paraula2 != NULL ) {
      if( !strcmp( paraula, paraula2 ) ) {
       printf( "... linia %lu\n", numlin );
      } /* if */
      pos = pos + strlen( paraula2 );
      free( paraula2 );
      paraula2 = seguent_paraula( nom, pos );
     } /* while */
    } /* while */
    free( paraula );
    fclose( entrada );
    printf( "Fi.\n" );
   } else {
    printf( "No puc obrir %s!\n", nom );
   } /* if */
   return 0;
} /* main */
```

S'ha de programar, doncs, la funció `seguent_paraula()`.

2) Componeu, a partir de les funcions donades en l'apartat 3.5.2, la funció per eliminar l'element enèsim d'una llista d'enters. El programa principal haurà de ser el següent:

```
int main( void )
{
   llista_t     llista;
   char         opcio;
   int          dada;
   unsigned int n;

   printf( "Gestor de llistes d'enters.\n" );
   llista = NULL;
```

ANOTACIONS

```
  do {
   printf(
    " [I]nserir [E]liminar [M]ostrar o [S]ortir? "
   ); /* printf */
   do { opcio = getchar(); } while( isspace(opcio) );
   opcio = toupper( opcio );
   switch( opcio ) {
    case 'I':
     printf( "Dada =? ");
     scanf( "%i", &dada );
     printf( "Posicio =? " );
     scanf( "%u", &n );
     if( !insereix_enesim_llista( &llista, n, dada ) ) {
      printf( "No s'ha inserit.\n" );
     } /* if */
     break;
    case 'E':
     printf( "Posició =? " );
     scanf( "%u", &n );
     if( elimina_enesim_llista( &llista, n, &dada ) ) {
      printf( "Dada = %i\n", dada );
     } else {
      printf( "No s'ha eliminat.\n" );
     } /* if */
     break;
    case 'M':
     mostra_llista( llista );
     break;
   } /* switch */
  } while( opcio != 'S' );
  while( llista != NULL ) {
   elimina_enesim_llista( &llista, 0, &dada );
  } /* while */
  printf( "Fi.\n" );
  return 0;
} /* main */
```

També cal programar la **funció** `mostra_llista()` per a poder veure'n el contingut.

3) Feu un programa que permeti inserir i eliminar elements d'una cua d'enters. Les funcions que s'han d'emprar es troben en

ANOTACIONS

l'apartat 3.5.2 (en la part que parla de les cues). Per tant, només cal desenvolupar la funció principal del programa esmentat, que es pot inspirar en la mostrada en l'exercici anterior.

4) Programeu l'algoritme d'ordenació per selecció vist en l'apartat 3.6 per classificar un fitxer de text en què cada línia tingui el format següent:

```
DNI nota '\n'
```

Per tant, els elements seran del mateix tipus de dades que el que hem vist en l'exemple. El programa principal serà:

```
int main( void )
{
   FILE        *entrada;
   char        nom[ BUFSIZ ];
   llista_t    pendents, ordenats;
   ref_node_t  refnode;
   element_t   element;

   printf( "Ordena llista de noms.\n" );
   printf( "Fitxer =? " ); gets( nom );
   entrada = fopen( nom, "rt" );
   if( entrada != NULL ) {
    inicialitza_llista( &pendents );
    while( fgets( nom, BUFSIZ-1, entrada ) != NULL ) {
     element = llegeix_element( nom );
     posa_al_final_de_llista( &pendents, element );
    } /* if */
    inicialitza_llista( &ordenats );
    while(! es_buida_llista( pendents ) ) {
     element = extreu_minim_de_llista( &pendents );
     posa_al_final_de_llista( &ordenats, element );
    } /* while */
    printf( "Llista ordenada per DNI:\n" );
    principi_de_llista( &ordenats );
    while( !es_final_de_llista( ordenats ) ) {
     refnode = ref_node_de_llista( ordenats );
     element = element_en_ref_node(ordenats, refnode);
     mostra_element( element );
```

```
      avanca_posicio_en_llista( &ordenats );
    } /* while */
    printf( "Fi.\n" );
  } else {
    printf( "No puc obrir %s!\n", nom );
  } /* if */
  return 0;
} /* main */
```

Per tant, també serà necessari programar les funcions següents:

- `element_t crea_element( unsigned int DNI, float nota );`
- `element_t llegeix_element( char *frase );`
- `void mostra_element( element_t element );`

Nota

En aquest cas, els elements de la llista no es destrueixen abans que acabi l'execució del programa perquè resulta més simple i, a més, se sap que l'espai de memòria que ocupa s'alliberarà totalment. Tot i així, no deixa de ser una mala pràctica de la programació i, per tant, es proposa com a exercici lliure la incorporació d'una funció per a eliminar les variables dinàmiques corresponents a cada element abans d'acabar l'execució del programa.

5) Implementeu el programa anterior en tres unitats de compilació diferents: una per al programa principal, que també es pot dividir en funcions més manejables, una per als elements i una altra per a les llistes, que es pot transformar en biblioteca.

6) Feu un programa que accepti com a argument un NIF i validi la seva lletra. Per a això, preneu com a referència l'exercici d'autoavaluació número 7 del capítol anterior.

7) Transformeu la utilitat de cerca de paraules en fitxers de text del primer exercici perquè prengui com a arguments en la línia d'ordres tant la paraula que s'ha de buscar com el nom del fitxer de text en el qual hagi de fer la cerca.

ANOTACIONS

8) Creeu una ordre que mostri el contingut del directori com si es tractés d'un `ls -als | more`. Per a això, cal fer un programa que executi aquesta ordre i torni el codi d'error corresponent.

9) Programeu un "despertador" perquè mostri un avís cada cert temps o en una hora determinada. Per a això, preneu com a referència el programa exemple vist en la secció de processos permanents.

   El programa tindrà com a argument l'hora i minuts en què s'ha de mostrar l'avís indicat, que serà el segon argument. Si l'hora i minuts es precedeix amb el signe `'+'`, llavors es tractarà com en l'exemple, és a dir, com el lapse de temps que ha de passar abans de mostrar l'avís.

   Cal tenir present que la lectura del primer valor del primer argument es pot fer de la mateixa manera que en el programa "avisador" del tema, ja que el signe `'+'` s'interpreta com a indicador de signe del mateix nombre. Això sí, cal llegir específicament `argv[1][0]` per a saber si l'usuari ha introduït el signe o no.

   Per a saber l'hora actual, és necessari emprar les funcions de biblioteca estàndard de temps, que estan declarades a `time.h`, i l'ús del qual es mostra en el programa següent:

```
/* Fitxer: horamin.c                                    */
#include <stdio.h>
#include <time.h>
int main( void )
{
   time_t temps;
   struct tm *temps_desc;
   time( &temps );
   temps_desc = localtime( &temps );
   printf( "Són les %2d i %2d minuts.\n",
     temps_desc->tm_hour,
     temps_desc->tm_min
   ); /* printf */
   return 0;
} /* main */
```

ANOTACIONS

10) Proveu els programes de detecció de nombres primers mitjançant fils i processos. Per a això, és necessari definir la funció `es_primer()` de manera adequada. El següent programa és una mostra d'aquesta funció, que aprofita el fet que que cap divisor enter no serà més gran que l'arrel quadrada del nombre (s'aproxima per la potència de 2 més semblant):

```
/* Fitxer: es_primer.c                                          */
#include <stdio.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
   unsigned long int nombre, maxim, divisor;
   bool              primer;

   if( argc == 2 ) {
    nombre = atol( argv[1] );
    primer = nombre < 4; /* 0 ... 3, considerats primers.  */
    if( !primer ) {
     divisor = 2;
     primer = nombre % divisor != 0;
     if( primer ) {
      maxim = nombre / 2;
      while( maxim*maxim > nombre ) maxim = maxim/2;
      maxim = maxim * 2;
      divisor = 1;
      while( primer && (divisor < maxim) ) {
       divisor = divisor + 2;
       primer = nombre % divisor != 0;
      } /* while */
     } /* if */
    } /* if */
    printf( "... %s primer.\n", primer? "és" : "no és" );
   } else {
    printf( "Ús: %s nombre_natural\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

ANOTACIONS

3.17.1. Solucionari

1) Com que ja tenim el programa principal, n'hi ha prou de mostrar la funció `seguent_paraula`:

```
paraula_t seguent_paraula(
   char         *frase,
   unsigned int inici)
{
   unsigned int final, longitud;
   paraula_t    paraula;

   while( frase[inici]!= '\0' && !isalnum(frase[inici]) ) {
    inici = inici + 1;
   } /* while */
   final = inici;
   while( frase[fin]!= '\0' && isalnum( frase[fi] ) ) {
    fi = fi + 1;
   } /* while */
   longitud = fi - inici;
   if( longitud > 0 ) {
    paraula = (paraula_t)malloc((longitud+1)*sizeof(char));
    if( paraula != NULL ) {
     strncpy( paraula, &(frase[inici]), longitud );
     paraula[longitud] = '\0';
    } /* if */
   } else {
    paraula = NULL;
   } /* if */
   return paraula;
} /* seguent_paraula */
```

2)

```
bool elimina_enesim_llista(
   llista_t *llistaref,/* Apuntador a referència 1r. node. */
   unsigned int n,     /* Posició de l'eliminació.         */
   int *dadaref)       /* Referència de la dada eliminada. */
{                      /* Torna FALSE si no es pot.        */
   node_t *p, *q, *t;
   bool retval;
   enesim_pq_node( *llistaref, n, &p, &q );
   if( q != NULL ) {
    *dadaref = destrueix_node( llistaref, p, q );
```

ANOTACIONS

```
    retval = TRUE;
   } else {
    retval = FALSE;
   } /* if */
   return retval;
} /* elimina_enesim_llista */

void mostra_llista( llista_t llista )
{
   node_t *q;
   if( llista != NULL ) {
    q = llista;
    printf( "llista = " );
    while( q != NULL ) {
     printf( " %i " q->dada );
     q = q->seguent;
    } /* while */
    printf( "\n" );
   } else {
    printf( "Llista buida.\n" );
   } /* if */
} /* mostra_llista */
```

3)

```
int main( void )
{
   cua_t  cua;
   char   opcio;
   int    dada;

   printf( "Gestor de cues d'enters.\n" );
   cua.primer = NULL; cua.ultim = NULL;
   do {
    printf(
     " [E]ncuar [D]esencuar [M]ostrar o [S]ortir? "
    ); /* printf */
    do opcio = getchar(); while( isspace(opcio) );
    opcio = toupper( opcio );
    switch( opcio ) {
```

ANOTACIONS

```
      case 'E':
       printf( "Dada =? ");
       scanf( "%i", &dada );
       if( !encua( &cua, dada ) ) {
        printf( "No s'ha inserit.\n" );
       } /* if */
       break;
      case 'D':
       if( desencua( &cua, &dada ) ) {
        printf( "Dada = %i\n", dada );
       } else {
        printf( "No s'ha eliminat.\n" );
       } /* if */
       break;
      case 'M':
       mostra_llista( cua.primer );
       break;
     } /* switch */
   } while( opcio != 'S' );
   while( desencua( &cua, &dada ) ) { ; }
   printf( "Fi.\n" );
   return 0;
} /* main */
```

4)

```
element_t crea_element( unsigned int DNI, float nota )
{
   element_t element;
   element = (element_t)malloc( sizeof( dada_t ) );
   if( element != NULL ) {
    element->DNI = DNI;
    element->nota = nota;
   } /* if */
   return element;
} /* crea_element */

element_t llegeix_element( char *frase )
{
   unsigned int DNI;
   double       nota;
   int          llegit_ok;
   element_t    element;
   llegit_ok = sscanf( frase, "%u%lf", &DNI, &nota );
```

ANOTACIONS

```
   if( llegit_ok == 2 ) {
     element = crea_element( DNI, nota );
   } else {
     element = NULL;
   } /* if */
   return element;
} /* llegeix_element */

void mostra_element( element_t element )
{
   printf( "%10u %.2f\n", element->DNI, element->nota );
} /* mostra_element */
```

5) Vegeu l'apartat 3.6.2.

6)

```
char lletra_de( unsigned int DNI )
{
   char codi[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;
   return codi[ DNI % 23 ];
} /* lletra_de */

int main( int argc, char *argv[] )
{
   unsigned int DNI;
   char         lletra;
   int          codi_error;

   if( argc == 2 ) {
     sscanf( argv[1], "%u", &DNI );
     lletra = argv[1][ strlen(argv[1])-1 ];
     lletra = toupper( lletra );
     if( lletra == lletra_de( DNI ) ) {
       printf( "NIF vàlid.\n" );
       codi_error =0;
     } else {
       printf( " Lletra %c no vàlida!\n", lletra );
       codi_error = -1;
     } /* if */
   } else {
     printf( "Ús: %s DNI-lletra\n", argv[0] );
     codi_error = 1;
   } /* if */
   return codi_error;
} /* main */
```

ANOTACIONS

7)

```
int main( int argc, char *argv[] )
{
   FILE         *entrada;
   char         nom[BUFSIZ];
   paraula_t    paraula, paraula2;
   unsigned int numlin, pos;
   int          codi_error;
   if( argc == 3 ) {
    paraula = següent_paraula( argv[1], 0 );
    if( paraula != NULL ) {
     entrada = fopen( argv[2], "rt" );
     if( entrada != NULL ) {
      /* (Vegeu: enunciat de l'exercici 1.) */
     } else {
      printf( " No puc obrir %s!\n", argv[2] );
      codi_error = -1;
     } /* if */
    } else {
     printf( " Paraula %s invàlida!\n", argv[1] );
     codi_error = -1;
    } /* if */
   } else {
    printf( "Ús: %s paraula fitxer\n", argv[0] );
    codi_error = 0;
   } /* if */
   return codi_error;
} /* main */
```

8)

```
int main( int argc, char *argv[] )
{
   int codi_error;
   codi_error = system( "ls -als | more" );
   return codi_error;
} /* main */
```

ANOTACIONS

9)

```
int main( int argc, char *argv[] )
{
   unsigned int hores;
   unsigned int minuts;
   unsigned int segons;
   char         *avis, *separador;
   time_t       temps;
   struct tm    *temps_desc;

   if( argc == 3 ) {
    separador = strchr( argv[1], ':' );
    if( separador != NULL ) {
     hores = atoi( argv[1] ) % 24;
     minuts = atoi( separador+1 ) % 60;
    } else {
     hores = 0;
     minuts = atoi( argv[1] ) % 60;
    } /* if */
    if( argv[1][0]!= '+' ) {
     time( &temps );
     temps_desc = localtime( &temps );
     if( minuts < temps_desc->tm_min ) {
      minuts = minuts + 60;
      hores = hores - 1;
     } /* if */
     if( hores < temps_desc->tm_hour ) {
      hores = hores + 24;
     } /* if */
     minuts = minuts - temps_desc->tm_min;
     hores = hores - temps_desc->tm_hour;
    } /* if */
    segons = (hores*60 + minuts) * 60;
    avís = argv[2];
    if( daemon( FALSE, TRUE ) ) {
     printf( "No es pot instal·lar l'avisador : (\n" );
    } else {
     printf( "Alarma dins de %i hores i %i minuts.\n",
      hores, minuts
     ); /* printf */
```

ANOTACIONS

```
      printf( "Fes $ kill %li per a tancar-la.\n",
        getpid()
      ); /* printf */
     } /* if */
     sleep( segons );
     printf( "%s\007\n", avís );
     printf( "Alarma tancada.\n" );
    } else {
     printf( "Ús: %s [+]HH:MM \"avis\"\n, argv[0] );
     printf( " (amb + és respecte de l'hora actual)\n" );
     printf( " (sense + és l'hora del dia)\n" );
    } /* if */
    return 0;
} /* main */
```

10) Es tracta de repetir el codi donat dins d'una funció que tingui la capçalera adequada per a cada cas.

4. Programació orientada a objectes en C++

4.1. Introducció

Fins al moment s'ha estudiat com s'aborda un problema utilitzant els paradigmes de programació modular i el disseny descendent d'algoritmes. Amb ells s'aconsegueix afrontar un problema complex mitjançant la descomposició en problemes més simples, reduint-ne progressivament el nivell d'abstracció, fins a obtenir un nivell de detall manejable. Al final, el problema es redueix a estructures de dades i funcions o procediments.

Per a treballar de manera eficient, les bones pràctiques de programació ens aconsellen agrupar els conjunts de rutines i estructures relacionats entre els uns amb els altres en unitats de compilació, que després s'enllaçarien amb l'arxiu principal. Amb això, s'aconsegueix el següent:

- Localitzar amb rapidesa el codi font que fa una tasca determinada i limitar l'impacte de les modificacions a uns arxius determinats.
- Millorar la llegibilitat i comprensió del codi font en conjunt en no barrejar-se entre elles cada una de les parts.

No obstant això, aquesta recomanable organització dels documents del projecte només proporciona una separació dels diferents arxius i, d'altra banda, no reflecteix l'estreta relació que hi sol haver entre les dades i les funcions.

En la realitat moltes vegades es volen implementar entitats de manera que es compleixin unes propietats generals: conèixer les entrades que necessiten, una idea general del seu funcionament i les sortides que generen. Generalment, els detalls concrets de la implementació no són importants: segurament hi haurà desenes de maneres possibles de fer-ho.

ANOTACIONS

Es pot posar com a exemple un televisor. Les seves propietats poden ser la marca, model, mides, nombre de canals; i les accions que s'han d'implementar serien engegar o tancar el televisor, canviar de canal, sintonitzar un nou canal, etc. Quan utilitzem un aparell de televisió, el veiem com una caixa tancada, amb les seves propietats i les seves connexions. No ens interessa en absolut els seus mecanismes interns, només volem que actuï quan pitgem el botó adequat. A més a més, es pot utilitzar en diverses localitzacions i sempre tindrà la mateixa funció. D'altra banda, si s'espatlla es pot substituir per un altre i les seves característiques bàsiques (tenir una marca, engegar, tancar, canviar de canal, etc.) continuen essent iguals independentment que el nou aparell sigui més modern. El televisor es tracta com un objecte per si mateix i no com un conjunt de components.

Aquest mateix principi aplicat a la programació s'anomena *encapsulament*. L'**encapsulament** consisteix a implementar un element (els detalls d'aquesta acció es veuran més endavant) que actuarà com una "caixa negra", on s'especificaran unes entrades, una idea general del seu funcionament i unes sortides. D'aquesta manera es facilita el següent:

- La reutilització de codi. Si ja es disposa d'una "caixa negra" que tingui unes característiques coincidents amb les necessitats definides, es podrà incorporar sense interferir amb la resta del projecte.

- El manteniment del codi. Es poden fer modificacions sense que afectin el projecte en conjunt, sempre que es continuïn complint les especificacions de la "caixa negra" esmentada.

A cada un d'aquests elements l'anomenarem *objecte* (quant als objectes de la vida real que representa). En treballar amb objectes, cosa que representa un nivell d'abstracció més gran, s'afronta el disseny d'una aplicació no pensant en la seqüència d'instruccions que s'han de fer, sinó en la definició dels objectes que hi intervenen i les relacions que s'hi estableixen.

En aquesta unitat estudiarem un llenguatge nou que ens permet implementar aquesta nova visió que implica el paradigma de la programació orientada a objectes: C++.

ANOTACIONS

Aquest llenguatge nou es basa en el llenguatge C al qual es dota de característiques noves. Per aquest motiu, en primer lloc, s'estableix una comparació entre tots dos llenguatges en els àmbits comuns que ens permet un aprenentatge ràpid de les seves bases. A continuació, se'ns presenta el paradigma nou i les eines que el nou llenguatge proporciona per a la implementació dels objectes i les seves relacions. Finalment, es mostra com aquest canvi de filosofia afecta el disseny d'aplicacions.

En aquesta unitat es pretén que els lectors, partint dels seus coneixements del llenguatge C, puguin conèixer els principis bàsics de la programació orientada a objectes utilitzant el llenguatge C++ i del disseny d'aplicacions seguint aquest paradigma. En concret, en finalitzar l'estudi d'aquesta unitat, el lector haurà assolit els objectius següents:

1) Conèixer les diferències principals entre C i C++, inicialment sense explorar encara la tecnologia d'objectes.
2) Comprendre el paradigma de la programació orientada a objectes.
3) Saber implementar classes i objectes en C++.
4) Conèixer les propietats principals dels objectes: l'herència, l'homonímia i el polimorfisme.
5) Poder dissenyar una aplicació simple en C++ aplicant els principis del disseny orientat a objectes.

4.2. De C a C++

4.2.1. El primer programa en C++

Escollir l'entorn de programació C++ per a la implementació del nou paradigma de la programació orientada a objectes implica un gran avantatge per les nombroses similituds existents amb el llenguatge C. No obstant això, es pot convertir en una limitació si el programador no explora les característiques addicionals que ens proporciona el nou llenguatge i que aporten una sèrie de millores bastant interessants.

ANOTACIONS

Tradicionalment, en el món de la programació, la primera presa de contacte amb un llenguatge de programació es fa a partir del clàssic missatge d'"hola món" i, en aquest cas, no farem una excepció.

Nota

L'extensió ".cpp" indica al compilador que el tipus de codi font és C++.

Per tant, en primer lloc, escriviu en el vostre editor el text següent i deseu-lo amb el nom *exemple01.cpp*:

```
#include <iostream>
int main()
{
   cout << "hola mon \n" ;
   return 0;
}
```

Comparant aquest programa amb el primer programa en C, observem que l'estructura és similar. De fet, com s'ha comentat, el C++ es pot veure com una evolució del C per a implementar la programació orientada a objectes i, com a tal, manté la compatibilitat en un percentatge molt alt del llenguatge.

L'única diferència observable la trobem en la manera de gestionar la sortida que es fa per mitjà d'un objecte anomenat `cout`. La naturalesa dels objectes i de les classes s'estudiarà en profunditat més endavant, però, de moment, ens podrem fer una idea considerant la classe com un tipus de dades nou que inclou atributs i funcions associades, i l'objecte com una variable d'aquest tipus de dades.

La definició de l'objecte `cout` es troba dins de la llibreria <iostream>, que s'inclou en la primera línia del codi font. També crida l'atenció la forma d'ús, mitjançant el direccionament (amb el símbol <<) del text d'"hola món" sobre l'objecte `cout`, que genera la sortida d'aquest missatge a la pantalla.

A causa que el tema del tractament de les funcions d'entrada/sortida és una de les principals novetats del C++, començarem per aquest per desenvolupar les diferències entre un llenguatge i l'altre.

ANOTACIONS

4.2.2. Entrada i sortida de dades

Encara que ni el llenguatge C ni el C++ no defineixen les operacions d'entrada/sortida dins del llenguatge en si mateix, és evident que és indispensable el seu tractament per al funcionament dels programes. Les operacions que permeten la comunicació entre els usuaris i els programes són en biblioteques que proveeix el compilador. D'aquesta manera, podrem traslladar un codi font escrit per a un entorn Sun al nostre PC a casa i així obtindrem una independència de la plataforma. Almenys, en teoria.

Tal com s'ha comentat en unitats anteriors, el funcionament de l'entrada/sortida en C es produeix per mitjà de llibreries de funcions, la més important de les quals és la <stdio.h> o <cstdio> (entrada/sortida estàndard). Les funcions esmentades (`printf`, `scanf`, `fprint`, `fscanf`, etc.) continuen essent operatives en C++, encara que no se'n recomana l'ús en no aprofitar els beneficis que proporciona l'entorn de programació nou.

Nota

Totes dues maneres d'expressar el nom de la llibreria <xxxx.h> o <cxxxx> són correctes, encara que la segona es considera actualment la manera estàndard d'incorporar llibreries C dins del llenguatge C++ i l'única recomanada per al seu ús en aplicacions noves.

C++, igual com C, entén la comunicació de dades entre el programa i la pantalla com un flux de dades: el programa va enviant dades i la pantalla en va rebent i mostrant. De la mateixa manera s'entén la comunicació entre el teclat (o altres dispositius d'entrada) i el programa.

Per a gestionar aquests fluxos de dades, C++ inclou la classe iostream, que crea i inicialitza quatre objectes:

- `cin`. Maneja fluxos d'entrada de dades.
- `cout`. Maneja fluxos de sortida de dades.

ANOTACIONS

- `cerr`. Maneja la sortida cap al dispositiu d'error estàndard, la pantalla.
- `clog`. Maneja els missatges d'error.

A continuació, presentem alguns exemples simples del seu ús.

```
#include <iostream>
int main()
{
   int nombre;
   cout << "Escriu un nombre";
   cin >> nombre;
}
```

En aquest bloc de codi s'observa el següent:

- La declaració d'una variable sencera amb la qual es vol treballar.
- El text "Escriu un nombre" (que podem considerar com un flux de dades literal) que volem enviar al nostre dispositiu de sortida.

  Per a aconseguir el nostre objectiu, es direcciona el text cap a l'objecte `cout` mitjançant l'operador >>. El resultat serà que el missatge sortirà per pantalla.

- Una variable on es vol desar l'entrada de teclat. Una altra vegada el funcionament desitjat consistirà a direccionar el flux d'entrada rebut al teclat (representat/gestionat per l'objecte `cin`) sobre aquesta variable.

La primera sorpresa per als programadors en C, acostumats al `printf` i a l'`scanf`, és que no s'indica en la instrucció el format de les dades que es vol imprimir o rebre. De fet, aquest és un dels principals avantatges de C++: el compilador reconeix el tipus de dades de les variables i tracta el flux de dades de manera conseqüent. Per tant, simplificant una mica, es podria considerar que els objectes `cin` i `cout` s'adapten al tipus de dades. Aquesta característica ens permetrà adaptar els objectes `cin` i `cout` per al tractament de nous tipus de dades (per exemple, `structs`), cosa impensable amb el sistema anterior.

ANOTACIONS

Si es vol mostrar o recollir diverses variables, simplement, s'encadenen fluxos de dades:

```
#include <iostream>
int main()
{
   int i, j, k;
   cout << "Introduir tres nombres";
   cin >> i >> j >> k;
   cout << "Els nombres són: "
   cout << i << ", " << j << " i " << k;
}
```

En l'última línia es veu com en primer lloc s'envia al `cout` el flux de dades corresponent al text "Els nombres són:"; després, el flux de dades corresponent a la variable i; posteriorment, el text literal " , ", i així fins al final.

En el cas de l'entrada de dades per teclat, `cin` llegirà caràcters fins a la introducció d'un caràcter de salt de línia (retorn o "\n"). Després, anirà extraient del flux de dades introduït caràcters fins a trobar el primer espai i deixarà el resultat en la variable *i*. El resultat d'aquesta operació també serà un flux de dades (sense el primer nombre que ja ha estat extret) que rebrà el mateix tractament: anar extraient caràcters del flux de dades fins al separador següent per enviar-lo a la variable següent. El procés es repetirà fins a llegir les tres variables.

Per tant, la línia de lectura es podria haver escrit de la manera següent i hauria estat equivalent, però menys clara:

```
( ( ( cin >> i ) >> j ) >> k )
```

Si es vol mostrar la variable en un format determinat, s'ha d'enviar un manipulador de l'objecte que li indiqui el format desitjat. En l'exemple següent, es podrà observar la seva mecànica de funcionament:

```
#include <iostream>
#include <iomanip>
// S'ha d'incloure per a la definició dels
// manipuladors d'objecte cout amb parametres
```

ANOTACIONS

```
int main()
{
   int i = 5;
   float j = 4.1234;

   cout << setw(4) << i << endl;
   //mostra i amb amplada de 4 car.
   cout << setprecisio(3) << j << endl;
   // mostra j amb 3 decimals
}
```

Hi ha moltes altres possibilitats de format, però no és l'objectiu d'aquest curs. Aquesta informació addicional està disponible en l'ajuda del compilador.

4.2.3. Utilitzant C++ com C

Com s'ha comentat, el llenguatge C++ va néixer com una evolució del C. Per aquest motiu, per als programadors en C és bastant simple adaptar-se al nou entorn. No obstant això, a més d'introduir tot el tractament per a la programació orientada a objectes, C++ aporta algunes millores respecte a la programació clàssica que és interessant conèixer i que posteriorment, en la programació orientada a objectes, adquireixen tota la seva dimensió.

A continuació, analitzarem diferents aspectes del llenguatge.

4.2.4. Les instruccions bàsiques

En aquest aspecte, C++ es manté fidel al llenguatge C: les instruccions mantenen el seu aspecte general (acabades en punt i coma, els blocs de codi entre claus, etc.) i les instruccions bàsiques de control de flux, tant les de selecció com les iteratives, conserven la seva sintaxi (`if`, `switch`, `for`, `while`, `do ... while`). Aquestes característiques garanteixen una aproximació ràpida al llenguatge nou.

ANOTACIONS

Dins de les instruccions bàsiques, podríem incloure les d'entrada/sortida. En aquest cas, C i C++ sí que presenten diferències significatives, diferències que ja hem comentat en l'apartat anterior.

A més a més, és important destacar que s'ha inclòs una manera nova d'afegir comentaris dins del codi font per contribuir a millorar la seva lectura i el seu manteniment. Es conserva la forma clàssica dels comentaris en C com el text inclòs entre les seqüències de caràcters /* (inici de comentari) i */ (final de comentari), però s'afegeix una forma nova que ens permet un comentari fins a final de línia: la seqüència //.

Exemple

```
/*
Aquest text està comentat utilitzant la forma clàssica de C.
Pot contenir la quantitat de línies que es vulgui.
*/

//
// Aquest text utilitza el nou format de comentaris
// fins a final de línia, que incorpora C++
//
```

4.2.5. Els tipus de dades

Els tipus de dades fonamentals de C (`char`, `int`, `long int`, `float` i `double`) es conserven en C++, i s'incorpora el nou tipus `bool` (tipus booleà o lògic), que pot adquirir dos valors possibles: fals (*false*) o vertader (*true*), ara definits dins del llenguatge.

```
// ...
{
   int i = 0, nre;
   bool continuar;

   continuar = true;
   do
   {
    i++;
    cout <<"Per a acabar aquest bucle que ha passat";
    cout << i << "vegades, escriu un 0";
    cin >> nre;
    if (nre == 0) continuar = false;
```

ANOTACIONS

```
   } while (continuar);
}
```

Encara que l'ús de variables de tipus lògic o booleà ja era comú en C, utilitzant com a suport els nombres enters (el 0 com a valor fals i qualsevol altre valor com a verdader), la nova implementació simplifica el seu ús i ajuda a reduir errors. A més a més, el nou tipus de dades només ocupa un byte de memòria, en lloc dels dos bytes que utilitzava quan se simulava amb el tipus `int`.

D'altra banda, cal destacar les novetats respecte dels tipus estructurats (`struct`, `enum` o `unio`). En C++ passen a ser considerats descriptors de tipus de dades completes, amb la qual cosa s'evita la necessitat de l'ús de la instrucció `typedef` per a definir tipus de dades nous.

En l'exemple que segueix, es pot comprovar que la part de la definició del nou tipus no varia:

```
struct data {
   int dia;
   int mes;
   int any;
};
enum diesSetmana{DILLUNS, DIMARTS, DIMECRES, DIJOUS,
                 DIVENDRES, DISSABTE, DIUMENGE};
```

El que se simplifica és la declaració d'una variable del tipus esmentat, ja que no s'han de tornar a utilitzar els termes `struct`, `enum` o `unio`, o la definició de nous tipus mitjançant la instrucció `typedef`:

```
data aniversari;
diesSetmana festiu;
```

D'altra banda, la referència a les dades tampoc no varia.

```
// ...
aniversari.dia = 2;
aniversari.mes = 6;
aniversari.any = 2001;
festiu = DILLUNS;
```

ANOTACIONS

En el cas de la declaració de variables tipus `enum` es compleixen dues funcions:

– Declarar `diesSetmana` com un tipus nou.

– Fer que el `DILLUNS` correspongui a la constant 0, `DIMARTS` a la constant 1 i així successivament.

Per tant, cada constant enumerada correspon a un valor sencer. Si no s'especifica res, el primer valor prendrà el valor de 0 i les constants següents aniran incrementant el seu valor en una unitat. No obstant això, C++ permet canviar aquest criteri i assignar un valor determinat a cada constant:

```
enum comportament {HORRIBLE = 0, DOLENT, REGULAR = 100,
            BO = 200, MOLT_BO, EXCEL·LENT};
```

D'aquesta manera, `HORRIBLE` prendria el valor de 0, `DOLENT` el valor d'1, `REGULAR` el valor de 100, `BO` el de 200, `MOLT_BO` el de 201 i `EXCEL·LENT`, el de 202.

Un altre aspecte que s'ha de tenir en compte és la recomanació en aquesta nova versió que es refereix a fer les coercions de tipus de manera explícita. La versió C++ es recomana per la seva llegibilitat:

```
int i = 0;
long v = (long) i; // coerció de tipus en C
long v = long (i); // coerció de tipus en C++
```

4.2.6. La declaració de variables i constants

La declaració de variables en C++ continua tenint el mateix format que en C, però s'introdueix un element nou que aportarà més seguretat en la nostra manera de treballar. Es tracta de l'especificador `const` per a la definició de constants.

En programació s'utilitza una constant quan es coneix amb certesa que aquest valor no ha de variar durant el procés d'execució de l'aplicació:

```
const float PI = 3.14159;
```

ANOTACIONS

Una vegada definida aquesta constant, no s'hi pot assignar cap valor i, per tant, sempre estarà en la part dreta de les expressions. D'altra banda, una constant sempre ha d'estar inicialitzada:

```
const float radi;// ERROR!!!!!!!!
```

L'ús de constants no és nou en C. La manera clàssica de definir-les és mitjançant la instrucció de prepocessador `#define`.

```
#define PI 3.14159
```

En aquest cas, el comportament real és substituir cada aparició del text `PI` pel seu valor en la fase de preprocessament del text. Per tant, quan analitza el text, el compilador només veu el nombre 3.14159 en lloc de `PI`.

No obstant això, mentre que el segon cas correspon a un tractament especial durant el procés previ a la compilació, l'ús de `const` permet un ús normalitzat i similar al d'una variable però amb capacitats limitades. En canvi, rep el mateix tractament que les variables respecte a l'àmbit d'actuació (només en el fitxer de treball, llevat que s'hi indiqui el contrari mitjançant la paraula reservada `extern`) i té un tipus assignat, amb el qual es podran fer totes les comprovacions de tipus en fase de compilació fent el codi font resultant més robust.

4.2.7. La gestió de variables dinàmiques

La gestió directa de la memòria és una de les armes més poderoses de què disposa el C, i una de les més perilloses: qualsevol accés inadequat a zones de memòria no corresponent a les dades desitjades pot provocar resultats imprevisibles en el millor dels casos, si no desastrosos.

En els capítols anteriors, s'ha vist que les operacions amb adreces de memòria en C es basen en els apuntadors (`*apunt`), usats per a accedir a una variable a partir de la seva adreça de memòria. Utilitzen l'operador d'indirecció o desreferència (`*`), i les seves característiques són:

- `apuntador` conté una adreça de memòria.

ANOTACIONS

- `*apuntador` indica el contingut existent en l'adreça de memòria esmentada.
- Per a accedir a l'adreça de memòria d'una variable s'utilitza l'operador adreça (&), que precedeix el nom de la variable.

```
// Exemple d'ús d'apuntadors
int i = 10;
int *apunt_i = &i;
      // apunt_i pren l'adreça
      // de la variable i de tipus sencer
      // si no l'assignessim aquí, seria
      // recomanable inicialitzar-lo en NULL

*apunt_i = 3;
      // S'assigna el valor 3 a la posició
      // de memòria apunt_i
      //Per tant, es modifica el valor de i.

cout << "Valor original           : " << i << endl ;
cout << "Per mitjà de l'apuntador : " ;
cout << *apunt_i << endl ;
          // La sortida mostrarà:
          // Valor original: 3
          // Per mitjà de l'apuntador: 3
```

Operadors `new` i `delete`

El principal ús dels apuntadors està relacionat amb les variables dinàmiques. Les dues principals funcions definides en C que fan aquestes operacions són `malloc`() i `free`() per a reservar memòria i alliberar-la, respectivament. Totes dues funcions continuen essent vàlides en C++. No obstant això, C++ proporciona dos nous operadors que permeten un control més robust per a aquesta gestió. Són `new` i `delete`. En estar inclosos en el llenguatge, no es necessita la inclusió de cap llibreria específica.

L'operador `new` fa la reserva de memòria. El seu format és `new` i un tipus de dades. A diferència del `malloc`, en aquest cas no cal indicar-hi la mida de la memòria que s'ha de reservar, ja que el compilador el calcula a partir del tipus de dades utilitzat.

ANOTACIONS

Per a alliberar la memòria es disposa de l'operador `delete`, que també ofereix més robustesa que la funció `free`(), ja que protegeix internament el fet d'intentar alliberar memòria apuntada per un apuntador nul.

```
data * aniversari = new data;
...
delete aniversari;
```

Si es vol crear diversos elements, n'hi ha prou d'especificar-ho en forma de vector o matriu. El resultat és declarar la variable com a apuntador al primer element del vector.

D'una manera semblant, es pot alliberar tota la memòria reservada pel vector (cada un dels objectes creats i el vector mateix) utilitzant la forma `delete []`.

```
data * llunesPlenes = new data[12];
...
delete [] llunesPlenes;
```

Si s'ometen els claudàtors, el resultat serà eliminar únicament el primer objecte del vector –però no la resta– i crear una fuga de memòria; és a dir, memòria reservada a la qual no es pot accedir en el futur.

Apuntadors `const`

En la declaració d'apuntadors en C++ permet l'ús de la paraula reservada `const`. I a més hi ha diverses possibilitats.

```
const int * ap_i;
      // El valor *ap_i roman constant
      // però no la seva adreça ap_i
int * const ap_j;
        // L'adreça ap_j és constant
        // però no el seu valor *ap_j
const int * const ap_k;
        // Tant l'adreça ap_k
        // com el seu valor *ap_k són constants
```

ANOTACIONS

És a dir, en el cas d'apuntadors es pot fer constant el seu valor (`*ap_i`) o la seva adreça de memòria (`ap_i`), o totes dues coses. Per a no confondre's, n'hi ha prou de fixar-se en el text posterior a la paraula reservada `const`.

Amb la declaració d'apuntadors constants el programador indica al compilador quan es vol que el valor o l'adreça que conté un apuntador no pateixin modificacions. Per tant, qualsevol intent d'assignació no prevista es detecta en temps de compilació. D'aquesta manera es redueix el risc d'errors de programació.

Referències

Per a la gestió de variables dinàmiques, C++ afegeix un element nou que facilitarà el seu ús: les referències. Una referència és un àlies o un sinònim. Quan es crea una referència, s'inicialitza amb el nom d'una altra variable i actua com un nom alternatiu seu.

Per a crear-la s'escriu el tipus de la variable destinació, seguit de l'operador de referència (`&`) i del nom de la referència. Per exemple,

```
int i;
int & ref_i = i;
```

En l'expressió anterior es llegeix: la variable `ref_i` és una referència a la variable `i`. Les referències sempre s'han d'inicialitzar en el moment de la declaració (com si fos una `const`).

Cal destacar que encara que l'operador de referència i el d'adreça es representen de la mateixa manera (`&`), corresponen a operacions diferents, encara que estan relacionats entre elles. De fet, la característica principal de les referències és que si es demana la seva adreça, tornen la de la variable destinació.

```
#include <iostream>
int main()
{
   int i = 10;
   int & ref_i = i;
```

ANOTACIONS

```
    ref_i = 3; //S'assigna el valor 3 a la posició

    cout << "valor de i      " << i << endl;
    cout << "adreça de i     " << &i << endl;
    cout << "adreça de ref_i " << &ref_i <<endl;
}
```

Amb aquest exemple es pot comprovar que totes dues adreces són idèntiques, i que l'assignació sobre `ref_i` té el mateix efecte que si hagués estat sobre `i`.

Altres característiques de les referències són les següents:

- No es poden reassignar. L'intent de reassignació es converteix en una assignació en la variable sinònima.
- No se'ls pot assignar un valor nul.

L'ús principal de les referències és el de la crida a funcions, que veurem a continuació.

4.2.8. Les funcions i els seus paràmetres

L'ús de les funcions, element bàsic de la programació modular, continua tenint el mateix format: un tipus del valor de retorn, el nom de la funció i un nombre de paràmetres precedits pel seu tipus. A aquesta llista de paràmetres d'una funció també se la coneix com la **signatura d'una funció**.

Ús de paràmetres per valor o per variable

Nota

Aquí utilitzarem el terme *pas de paràmetres per variable* en lloc de *per referència* per a no induir a confusió amb les referències de C++.

Com s'ha comentat en unitats anteriors, en C hi ha dues maneres de passar paràmetres a una funció: per valor o per variable. En el primer cas, la funció rep una còpia del valor original del paràmetre, mentre que en el segon es rep l'adreça d'aquesta variable. D'aquesta manera, es pot accedir directament a la variable original, que es pot modificar. Per a fer-ho, la manera tradicional en C és passar a la funció com a paràmetre l'apuntador a una variable.

A continuació, veurem una funció que permet intercanviar el contingut de dues variables:

```
#include <iostream>

void intercanviar(int *i, int *j);

int main()
{
   int x = 2, y = 3;
   cout << " Abans. x = " << x << " y = " << y << endl;
   intercanviar(&x , &y);
   cout << " Despres. x = " << x << " y = " << y << endl;
}
void intercanviar(int *i, int *j)
{
   int k;
   k = *i;
   *i = *j;
   *j = k;
}
```

Es pot comprovar que l'ús de les desreferències (*) dificulta la seva comprensió. Però en C++ es disposa d'un nou concepte comentat anteriorment: les referències. La nova proposta consisteix a rebre el paràmetre com a referència en lloc d'apuntador:

```
#include <iostream>

void intercanviar(int &i, int &j);

int main()
{
   int x = 2, y = 3;
   cout << " Abans. x = " << x << " y = " << y << endl;
   intercanviar(x, y); //No intercanviar(&x, &y);
   cout << " Despres. x = " << x << " y = " << y << endl;
}
void intercanviar(int & i, int & j)
{
   int k;
   k = i;
   i = j;
   j = k;
}
```

ANOTACIONS

El funcionament d'aquesta nova proposta és idèntic al de l'anterior, però la lectura del codi font és molt més simple perquè utilitza l'operador de referència (`&`) per a recollir les adreces de memòria dels paràmetres.

No obstant això, cal recordar que les referències tenen limitacions (no poden prendre mai un valor nul i no es poden reassignar). Per tant, no es podran utilitzar les referències en el pas de paràmetres quan es vulgui passar un apuntador com a paràmetre i que aquest pugui ser modificat (per exemple, obtenir l'últim element en una estructura de dades de cua). Tampoc no es podran utilitzar referències per als paràmetres que vulguem considerar com a opcionals, ja que hi ha la possibilitat que no puguin ser assignats a cap paràmetre de la funció que els crida; per aquest motiu haurien de prendre el valor `null` (cosa que no és possible).

En aquests casos, el pas de paràmetres per variable s'ha de continuar fent mitjançant l'ús d'apuntadors.

Ús de paràmetres `const`

A la pràctica, en programar en C, de vegades s'utilitza el pas per variable com una forma d'eficiència en evitar haver de fer una còpia de les dades dins de la funció. Amb estructures de dades grans (estructures, etc.) aquesta operació interna de salvaguarda dels valors originals pot ocupar un temps considerable i, a més, es corre el risc d'una modificació de les dades per error.

Per a reduir aquests riscos, C++ permet col·locar l'especificador `const` just abans del paràmetre (tal com hem comentat en l'apartat d'apuntadors `const`).

Si en l'exemple anterior de la funció intercanviar s'haguessin definit els paràmetres `i` i `j` com a `const` (que no té cap sentit pràctic i només es considera a efectes explicatius), ens donaria errors de compilació:

```
void intercanviar(const int & i, const int & j);
{
   int k;
   k = i;
   i = j; // Error de compilació. Valor i constant.
```

ANOTACIONS

```
   j = k; // Error de compilació. Valor j constant.
}
```

D'aquesta manera es poden aconseguir els avantatges d'eficiència perquè s'eviten els processos de còpia no desitjats sense tenir l'inconvenient d'estar desprotegit davant de modificacions no desitjades.

Sobrecàrrega de funcions

C admet una mica de flexibilitat en les crides a funcions en permetre l'ús d'un nombre de paràmetres variables en la crida a una funció, sempre que siguin els paràmetres finals i en la definició d'aquesta funció se'ls hagi assignat un valor per al cas que no s'arribi a utilitzar aquest paràmetre.

C++ ha incorporat una opció molt més flexible, que és una de les novetats respecte al C més destacades: permet l'ús de diferents funcions amb el mateix nom (*homonímia de funcions*). Aquesta propietat també s'anomena *sobrecàrrega de funcions*.

Les funcions poden tenir el mateix nom però han de tenir diferències en la seva llista de paràmetres, sia en el nombre de paràmetres o bé en variacions en el seu tipus.

Cal destacar que el tipus del valor de retorn de la funció no es considera un element diferencial de la funció i, per tant, el compilador mostra error si s'intenten definir dues funcions amb el mateix nom i idèntic nombre i tipus de paràmetres que es vol que retornin valors de diferents tipus. El motiu és que el compilador no pot distingir quina de les funcions definides es vol cridar.

A continuació es proposa un programa que eleva nombres de diferent tipus al quadrat:

```
#include <iostream>

int elevarAlQuadrat (int);
float elevarAlQuadrat (float);
```

ANOTACIONS

```
int main()
{
   int nreEnter = 123;
   float nreReal = 12.3;
   int nreEnterAlQuadrat;
   float nreRealAlQuadrat;

   cout << "Exemple per a elevar nombres al quadrat\n";
   cout << "Nombres originals \n";
   cout << "Nombre enter: " << nreEnter << "\n";
   cout << "Nombre real: " << nreReal << "\n";

   nreEnterAlQuadrat = elevarAlQuadrat (nreEnter);
   nreRealAlQuadrat = elevarAlQuadrat (nreReal);

   cout << "Nombres elevats al quadrat \n";
   cout << "Nombre enter:"<< nreEnterAlQuadrat << "\n";
   cout << "Nombre real: " << nreRealAlQuadrat << "\n";

   return 0;
}

int elevarAlQuadrat (int nre)
{
   cout << "Elevant un nombre enter al quadrat \n";
   return ( nre * nre);
}
float elevarAlQuadrat (float nre)
{
   cout << "Elevant un nombre real al quadrat \n";
   return ( nre * nre);
}
```

ANOTACIONS

El fet de sobrecarregar la funció `ElevarAlQuadrat` ha permès que amb el mateix nom de funció es pugui fer el que és intrínsicament la mateixa operació. Amb això, hem evitat haver de definir dos noms de funció diferents:

- `ElevarAlQuadratNombresEnters`
- `ElevarAlQuadratNombresReals`

D'aquesta manera, el mateix compilador identifica la funció que es vol executar pel tipus dels seus paràmetres i fa la crida correcta.

4.3. El paradigma de la programació orientada a objectes

En les unitats anteriors, s'han analitzat una sèrie de paradigmes de programació (modular i descendent) que es basen en la progressiva organització de les dades i la resolució dels problemes a partir de la seva divisió en un conjunt d'instruccions seqüencials. L'execució d'aquestes instruccions només es recolza en les dades definides prèviament.

Aquest enfocament, que ens permet afrontar múltiples problemes, també mostra les seves limitacions:

– L'ús compartit de les dades provoca que sigui difícil modificar i ampliar els programes per les seves interrelacions.

– El manteniment dels grans programes es torna realment complicat en no poder assegurar el control de totes les implicacions que comporten els canvis en el codi.

– La reutilització de codi també pot provocar sorpreses, una altra vegada perquè no es pot conèixer totes les implicacions que comporta.

Nota

Com és possible que es tinguin tantes dificultats si les persones són capaces de fer accions complexes en la seva vida quotidiana? La raó és molt senzilla: en la nostra vida quotidiana no es procedeix amb els mateixos criteris. La descripció del nostre entorn es fa a partir d'objectes –portes, ordinadors, automòbils, ascensors, persones, edificis, etc.–, els quals compleixen unes relacions més o menys simples: si una porta és oberta es pot passar i si és tancada no es pot. Si un automòbil té una roda punxada, se substitueix i es pot tornar a circular. I no necessitem conèixer tota la mecànica de l'automòbil per a poder fer aquesta operació! Ara bé,

ANOTACIONS

> ens podríem imaginar el nostre món si en canviar el pneumàtic ens deixés de funcionar el parabrisa? Seria un caos. Això és gairebé el que succeeix, o almenys no podem estar completament segurs que no succeeixi, amb els paradigmes anteriors.

El paradigma de l'orientació a objectes ens proposa una manera diferent d'enfocar la programació sobre la base de la definició d'objectes i de les relacions entre ells.

Cada objecte es representa mitjançant una **abstracció** que conté la seva informació essencial, sense preocupar-se de les seves altres característiques.

Aquesta informació es compon de dades (variables) i accions (funcions) i, llevat que s'indiqui específicament el contrari, el seu àmbit d'actuació es limita a l'objecte en qüestió (**ocultació de la informació**). D'aquesta manera es limita l'abast del seu codi de programació, i per tant la seva repercussió, sobre l'entorn que l'envolta. Aquesta característica s'anomena **encapsulament**.

Exemple

Exemple d'accions sobre objectes en la vida quotidiana: cridar per telèfon, despenjar el telèfon, parlar, contestar, etc.

Les relacions entre els diferents objectes poden ser diverses, i normalment comporten accions d'un objecte sobre l'altre que s'implementen mitjançant missatges entre els objectes.

Exemple

Un gos és un animal, un camió és un vehicle, etc.

Una de les relacions més important entre els objectes és la pertinença a un tipus més general. En aquest cas, l'objecte més específic comparteix una sèrie de trets (informació i accions) amb els més genèrics que són determinats per aquesta relació d'inclusió. El nou paradigma proporciona una eina per poder reutilitzar tots aquests trets de manera simple: **l'herència**.

Finalment, una característica addicional és el fet de poder-se comportar de manera diferent segons el context que l'envolta. És coneguda com a **polimorfisme** (un objecte, moltes formes). A més a més, aquesta propietat adquireix tota la seva potència en ser capaç d'adaptar aquest comportament en el moment de l'execució i no en temps de compilació.

ANOTACIONS

El paradigma de programació orientada a objectes es basa en aquests quatre pilars: abstracció, encapsulament, herència i polimorfisme.

4.3.1. Classes i objectes

En l'àmbit d'implementació, una classe correspon a un nou tipus de dades que conté una col·lecció de dades i de funcions que ens permeten la seva manipulació.

Exemple

Volem descriure un aparell de vídeo.

La descripció es pot fer a partir de les seves característiques com marca, model, nombre de capçals, etc., o per mitjà de les seves funcions com reproducció de cintes de vídeo, enregistrament, rebobinatge, etc. És a dir, tenim dues visions diferents per a tractar el mateix aparell.

El primer enfocament correspondria a una col·lecció de variables, mentre que el segon correspondria a una col·lecció de funcions.

L'ús de les classes ens permet integrar dades i funcions dins de la mateixa entitat.

El fet de reunir el conjunt de característiques i funcions en el mateix contenidor facilita la seva interrelació, i també el seu aïllament de la resta del codi font. En l'exemple de l'aparell de vídeo, la reproducció d'una cinta implica, una vegada posada, l'acció d'uns motors que fan que la cinta es vagi desplaçant per davant d'uns capçals que llegeixen la informació.

ANOTACIONS

En realitat, als usuaris el detall del funcionament ens és intrascendent, senzillament, veiem l'aparell de vídeo com una caixa que té una ranura i uns botons, i sabem que al seu interior conté uns mecanismes que ens imaginem bastant complexos. Però també sabem que en tenim prou de pitjar el botó "Play".

A aquest concepte se l'anomena **encapsulament de dades i funcions en una classe**. Les variables que són dins de la classe reben el nom de *variables membres* o *dades membres*, i a les funcions se les anomena *funcions membres* o *mètodes de la classe*.

Però les classes corresponen a elements abstractes; és a dir, a idees genèriques i a casa nostra no disposem d'un aparell de vídeo en forma d'idea, sinó d'un element real amb unes característiques i funcions determinades. Igualment, en C++ necessitarem treballar amb els elements concrets. A aquests elements els anomenarem *objectes*.

Declaració d'una classe

La sintaxi per a la declaració d'una classe és utilitzar la paraula reservada `class` seguida del nom de la classe i, entre claus, la llista de les variables membre i de les funcions membre.

```
class Gos
{
   // llista de variables membre
   int edat;
   int alcada;

   // llista de funcions membre
   void bordar();
};
```

La declaració de la classe no implica cap reserva de memòria, encara que informa de la quantitat de memòria que necessitarà cadascun dels objectes d'aquesta classe.

Exemple

En la classe Gos presentada, cada un dels objectes ocuparà 8 bytes de memòria: 4 bytes per a la variable

ANOTACIONS

membre `edat` de tipus sencer i 4 bytes per a la variable membre `alçada`. Les definicions de les funcions membre, en el nostre cas `bordar`, no impliquen reserva d'espai.

Implementació de les funcions membre d'una classe

Fins al moment, en la classe Gos hem declarat com a membres dues variables (`edat` i `alcada`) i una funció (`bordar`). Però no s'ha especificat la implementació de la funció.

La definició d'una funció membre es fa mitjançant el nom de la classe seguit per l'operador d'àmbit (`::`) , el nom de la funció membre i els seus paràmetres.

```
class Gos
{
   // llista de variables membre
   int edat;
   int alcada;

   // llista de funcions membre
   void bordar();
};

Gos:: bordar()
{
   cout << "Bub";
}
```

Nota

Encara que aquesta és la manera habitual, també és possible implementar les funcions membres en línia. Per a això, després de la declaració del mètode i abans del punt i coma (;) s'introdueix el codi font de la funció:

```
class Gos
{
```

ANOTACIONS

```
    // llista de variables membre
    int edat;
    int alcada;

    // llista de funcions membre
    void bordar()
      { cout << "Bup"; };
};
```

Aquest tipus de crides només és útil quan el cos de la funció és molt reduït (una o dues instruccions).

Funcions membre `const`

En el capítol anterior es va comentar la utilitat de considerar les variables que no haurien de patir modificacions en el transcurs de l'execució del programa, i la seva declaració mitjançant l'especificador `const`. També es va comentar la seguretat que aportaven els apuntadors `const`. Doncs de manera similar es podrà definir una funció membre com a `const`.

```
void bordar() const;
```

Per a indicar que una funció membre és `const`, només cal posar la paraula reservada `const` entre el símbol de tancar parèntesi després del pas de paràmetres i el punt i coma (;) final.

En fer-ho, s'indica al compilador que aquesta funció membre no pot modificar l'objecte. Qualsevol intent en el seu interior d'assignar una variable membre o cridar alguna funció no constant generarà un error per part del compilador. Per tant, és una mesura més a disposició del programador per a assegurar la coherència de les línies de codi font.

Declaració d'un objecte

Així com una classe es pot assimilar com un nou tipus de dades, un objecte només correspon a la definició d'un element d'aquest

ANOTACIONS

tipus. Per tant, la declaració d'un objecte segueix el mateix model:

```
Gos sulta;              // Objecte de la classe Gos.
```

Un objecte és una instància individual d'una classe.

4.3.2. Accés a objectes

Per a accedir a les variables i funcions membres d'un objecte s'utilitza l'operador punt (`.`): es posa el nom de l'objecte seguit de punt i del nom de la variable o funció que es vol.

En l'apartat anterior s'ha definit un objecte `sulta` de la classe Gos. Si es vol inicialitzar l'edat de `sulta` a 4 o cridar la seva funció `bordar()`, només cal fer el següent:

```
sulta.edat = 4;
sulta.bordar();
```

No obstant això, un dels principals avantatges proporcionats per les classes és que només són visibles aquells membres (tant dades com funcions) que ens interessa mostrar. Per aquest motiu, si no s'indica el contrari, els membres d'una classe només són visibles des de les funcions d'aquesta classe. En aquest cas, direm que els membres són **privats**.

Per a poder controlar l'àmbit dels membres d'una classe, es disposa de les paraules reservades següents: `public`, `private` i `protected`.

Nota

`protected` correspon a un cas més específic que s'estudiarà en l'apartat "Herència".

Quan es declara un membre (tant variables com funcions) com a `private` s'està indicant al compilador que el seu ús és privat i restringit a l'interior d'aquesta classe. En canvi, si es declara com a `public`, és accessible des de qualsevol lloc on s'utilitzin objectes d'aquesta classe.

ANOTACIONS

En el codi font, aquestes paraules reservades s'apliquen amb forma d'etiqueta davant dels membres del mateix àmbit:

```
class Gos
{
   public:
     void bordar();
   private:
     int edat;
     int alcada;
};
```

Nota

S'ha declarat la funció `bordar()` com a pública –cosa que permet l'accés des de fora de la classe–, però s'han mantingut ocults els valors d'`edat` i `alcada` en declarar-los com a privats.

Vegem-ho en la implementació d'un programa de manera completa:

```
#include <iostream>
class Gos
{
   public:
     void bordar() const
     { cout << "Bup"; };

   private:
     int edat;
     int alcada;
};

int main()
{
   Gos sulta;

   //Error de compilació. Ús de variable privada
   sulta.edat = 4;
   cout << sulta.edat;
}
```

En el bloc `main` s'ha declarat un objecte `sulta` de la classe Gos. Posteriorment, s'intenta assignar a la variable membre edat el valor de 4. Com que aquesta variable no és pública, el compilador dóna error i indica que no s'hi té accés. Igualment, ens mostraria un error de compilació similar per a la fila següent. Una solució en aquest cas seria declarar la variable membre `edat` com a `public`.

Confidencialitat de les dades membres

El fet de declarar una variable membre com a pública limita la flexibilitat de les classes, ja que una modificació del tipus de la variable afectarà els diferents llocs del codi on s'utilitzin aquests valors.

Una regla general de disseny recomana mantenir les dades membres com a privades, i manipular-les mitjançant funcions públiques d'accés on s'obté o s'hi assigna un valor.

En el nostre exemple, es podria utilitzar les funcions `obtenirEdat()` i `assignarEdat()` com a mètodes d'accés a les dades. Declarar la variable edat com a privada ens permetria canviar el tipus en enter a byte o fins i tot substituir la dada per la data de naixement. La modificació es limitaria a canviar el codi font en els mètodes d'accés, però continuaria de manera transparent fora de la classe, ja que es pot calcular l'edat a partir de la data actual i la data de naixement o assignar una data de naixement aproximada a partir d'un valor per a l'edat.

```
class Gos
{
   public:
     // Mètodes constructors i destructors
     Gos (int, int);
     Gos (int);
     Gos ();
     ~Gos ();
```

ANOTACIONS

```
    // Mètodes d'accés
    void assignarEdat(int);
    int obtenirEdat();
    void assignarAlcada(int);
    int obtenirAlcada();

    Mètodes de la classe
    void bordar();//
  private:
    int edat;
    int alcada;
};

Gos:: bordar()
{ cout << "Bup"; }

void Gos:: assignarAlcada (int nAlcada)
{ alcada = nAlcada; }

int Gos:: obtenirAlcada (int nAlcada)
{ return (alcada); }

void Gos:: assignarEdat (int nEdat)
{ edat = nEdat; }

int Gos:: obtenirEdat ()
{ return (edat); }
```

L'apuntador `this`

Un altre aspecte que es pot destacar de les funcions membres és que sempre tenen accés al mateix objecte per mitjà de l'apuntador `this`.

De fet, la funció membre `obtenirEdat` també es podria expressar de la manera següent:

```
int Gos:: obtenirEdat ()
{ return (this->edat); }
```

Vist d'aquesta manera, sembla que tingui poca importància. No obstant això, poder-se referir a l'objecte com a apuntador `this` o en la seva forma desreferenciada (`*this`) hi dóna molta potència. Veu-

ANOTACIONS

rem exemples més avançats sobre això quan estudiem la sobrecàrrega d'operadors.

4.3.3. Constructors i destructors d'objectes

Generalment, cada vegada que es defineix una variable, després s'inicialitza. Aquesta és una pràctica correcta amb què s'intenta prevenir resultats imprevisibles en utilitzar una variable sense haver-hi assignat cap valor previ.

Les classes també es poden inicialitzar. Cada vegada que es crea un objecte nou, el compilador crida un mètode específic de la classe per inicialitzar els seus valors que rep el nom de *constructor*.

El constructor sempre rep el nom de la classe, sense valor de retorn (ni tan sols `void`) i pot tenir paràmetres d'inicialització.

```
Gos::Gos()
{
   edat = 0;
}
```

o

```
Gos::Gos(int nEdat)// Nova edat del gos
{
   edat = nEdat;
}
```

En cas que no es defineixi específicament el constructor en la classe, el compilador utilitza el constructor predeterminat, que consisteix en el nom de la classe, sense cap paràmetre, i té un bloc d'instruccions buit. Per tant, no fa res.

```
Gos::Gos()
{ }
```

ANOTACIONS

Aquesta característica sona desconcertant, però permet mantenir el mateix criteri per a la creació de tots els objectes.

En el nostre cas, si volem inicialitzar un objecte de la classe Gos amb una edat inicial d'un any, utilitzem la definició del constructor següent:

```
Gos(int novaEdat);
```

D'aquesta manera, la crida al constructor seria de la manera següent:

```
Gos sulta(1);
```

Si no hi hagués paràmetres, la declaració del constructor que hauríem d'utilitzar dins de la classe seria la següent:

```
Gos();
```

La declaració del constructor a `main` o qualsevol altra funció del cos del programa quedaria de la manera següent:

```
Gos sulta();
```

Però en aquest cas especial es pot aplicar una excepció a la regla que indica que totes les crides a funcions van seguides de parèntesis encara que no tinguin paràmetres. El resultat final seria el següent:

```
Gos sulta;
```

El fragment anterior és una crida al constructor `Gos()` i coincideix amb la forma de declaració presentada inicialment i ja coneguda.

De la mateixa manera, sempre que es declari un mètode constructor, s'hauria de declarar un mètode **destructor** que s'encarregués de netejar quan ja no s'usarà més l'objecte i que alliberés la memòria utilitzada.

ANOTACIONS

El destructor sempre va precedit per una titlla (~), té el nom de la classe, i no té paràmetres ni valor de retorn.

```
~Gos();
```

En cas que no definim cap destructor, el compilador defineix un **destructor predeterminat**. La definició és exactament la mateixa, però sempre tindrà un cos d'instruccions buit:

```
Gos::~Gos()
{ }
```

Incorporant les noves definicions en el programa, el resultat final és el següent:

```
#include <iostream>

class Gos
{
   public:
     Gos(int edat); // Constructor amb un paràmetre
     Gos();         // Constructor predeterminat
     ~ Gos();       // Destructor
     void bordar();

   private:
     int edat;
     int alcada;
};

Gos:: bordar()
{ cout << "Bup"; }

int main()
{
   Gos sulta(4);    // Inicialitzant l'objecte
                    // amb una edat de 4.
```


ANOTACIONS

```
    sulta.bordar();
}
```

El constructor de còpia

El compilador, a més de proporcionar de manera predeterminada a les classes un mètode constructor i un mètode destructor, també forneix un mètode constructor de còpia.

Cada vegada que es crea una còpia d'un objecte es crida el constructor de còpia. Això inclou els casos en què un objecte es passa com a paràmetre per valor a una funció o es torna aquest objecte com a retorn de la funció. El propòsit del constructor de còpia és fer una còpia de les dades membres de l'objecte en un de nou. A aquest procés també se l'anomena *còpia superficial*.

Aquest procés, que generalment és correcte, pot provocar forts conflictes si entre les variables membres que s'han de copiar hi ha apuntadors. El resultat de la còpia superficial faria que dos apuntadors (el de l'objecte original i el de l'objecte còpia) apuntin cap a la mateixa adreça de memòria: si algun n'alliberés la memòria, provocaria que l'altre apuntador, en no poder-se adonar de l'operació, es quedés apuntant a una posició de memòria perduda, cosa que generaria una situació de resultats impredictibles.

La solució en aquests casos passa per substituir la còpia superficial per una **còpia profunda** en la qual es reserven noves posicions de memòria per als elements tipus apuntador i se'ls assigna el contingut de les variables apuntades pels apuntadors originals.

La forma que declari aquest constructor és la següent:

```
Gos :: Gos (const Gos & ungos);
```

En aquesta declaració s'observa la conveniència de passar el paràmetre com a referència constant ja que el constructor no ha d'alterar l'objecte.

La utilitat del constructor de còpia s'observa millor quan algun dels atributs és un apuntador. Per aquest motiu i per a aquesta prova

ANOTACIONS

canviarem el tipus d'edat a apuntador a enter. El resultat final seria el següent:

```
class Gos
{
   public:
     Gos();                        // Constructor predeterminat
     ~Gos();                       // Destructor
     Gos(const Gos & rhs);         // Constructor de còpia
     int obtenirEdat();            // Mètode d'accés

   private:                        // Prova apuntador
   int *edat;
};

Gos :: obtenirEdat()
{ return (*edat) }

Gos :: Gos ()                      // Constructor
{
   edat = new int;
   * edat = 3;
}

Gos :: ~Gos ()                     // Destructor
{
   delete edat;
   edat = NULL;
}
Gos :: Gos (const Gos & rhs)       // Constructor de còpia
{
   edat = new int;                 // Es reserva memòria nova
   *edat = rhs.obtenirEdat();      // Còpia el valor edat
                                   // en la nova posició

}
int main()
{
   Gos sulta(4);                   // Inicialitzant amb edat 4
}
```

Inicialitzar valors en els mètodes constructors

Es poden inicialitzar els valors en un mètode constructor d'una manera més neta i eficient, que consisteix a interposar aquesta inicialització entre la definició dels paràmetres del mètode i la clau que indica l'inici del bloc de codi.

```
Gos:: Gos () :
   edat (0),
   alcada (0)
{ }

Gos:: Gos (int nEdat, int nAlcada):
   edat (nEdat),
   alcada(nAlcada)
{ }
```

Tal com es veu en el fragment anterior, la inicialització consisteix en un símbol de dos punts (`:`) seguit de la variable que s'ha d'inicialitzar i, entre parèntesis, el valor que s'hi vol assignar. Aquest valor pot correspondre tant a una constant com a un paràmetre del constructor esmentat. Si hi ha més d'una variable que s'ha d'inicialitzar, se separen per comes (,).

Variables membres i funcions membres estàtiques

Fins al moment, quan ens hem referit a les classes i als objectes els hem situat en plans diferents: les classes descrivien ens abstractes, i els objectes descrivien elements creats i amb valors concrets.

No obstant això, hi ha moments en què els objectes necessiten referir-se a un atribut o a un mètode comú amb els altres objectes de la mateixa classe.

Exemple

Si estem creant una classe Animals, ens pot interessar conservar en alguna variable el nombre total de gossos que s'han creat fins al moment, o fer una funció que ens permeti comptar els gossos tot i que encara no se n'hagi creat cap.

ANOTACIONS

La solució és precedir la declaració de les variables membres o de les funcions membres amb la paraula reservada `static`. Amb això estem indicant al compilador que aquesta variable, o funció, es refereix a la classe en general i no a cap objecte en concret. També es pot considerar que s'està compartint aquesta dada o funció amb totes les instàncies de l'objecte.

En l'exemple següent es defineix una variable membre i una funció membre estàtiques:

```
class Gos {
// ...
static int nombreDeGossos; //Normalment serà privada
static int quantsGossos() { return nombreDeGossos; }
};
```

S'hi pot accedir de dues maneres:

- Des d'un objecte de la classe Gos.

```
Gos sulta = new Gos();
sulta.nombreDeGossos;
sulta.quantsGossos();
```

- Utilitzant l'identificador de la classe sense definir cap objecte.

```
Gos::nombreDeGossos;
Gos::quantsGossos();
```

Però cal tenir present un aspecte important: les variables i les funcions membres `static` es refereixen sempre a la classe i no a cap objecte determinat, per la qual cosa l'objecte `this` no existeix.

Com a conseqüència, en les funcions membres estàtiques no es podrà fer referència ni directament ni indirectament a l'objecte `this` i:

- Només podran cridar funcions estàtiques, ja que les funcions no estàtiques sempre esperen implícitament aquest objecte com a paràmetre.

ANOTACIONS

- Només podran tenir accés a variables estàtiques, perquè a les variables no estàtiques sempre s'hi accedeix per mitjà de l'objecte esmentat.
- No es podran declarar aquestes funcions com a `const` perquè ja no té sentit.

4.3.4. Organització i ús de biblioteques en C++

Fins al moment s'ha inclòs en el mateix arxiu la definició de la classe i el codi que la utilitza, però aquesta no és l'organització aconsellada si es vol reutilitzar la informació.

Nota

Les extensions que es fan servir de manera més estàndard són ".hpp" (més utilitzades en entorns Windows), ".H" i ".hxx" (més utilitzades en entorns Unix) o fins i tot ".h" (igual com en C).

Es recomana dividir el codi font de la classe en dos fitxers separant la definició de la classe i la seva implementació:

- El fitxer de capçalera incorpora la definició de la classe. L'extensió d'aquests fitxers pot variar entre diverses possibilitats, i la decisió final és arbitrària.
- El fitxer d'implementació dels mètodes de la classe i que conté un include del fitxer de capçalera. L'extensió utilitzada per a aquests fitxers també pot variar.

Nota

Les extensions que es fan servir de manera més estàndard són ".cpp" (més freqüents en entorns Windows), ".C" i ".cxx" (més utilitzades en entorns Unix).

Posteriorment, quan es vulgui reutilitzar aquesta classe, n'hi haurà prou d'incloure en el codi font una crida al fitxer de capçalera de la classe.

En el nostre exemple, la implementació es troba al fitxer gos.cpp, que fa un `include` del fitxer de capçalera gos.hpp.

Fitxer gos.hpp (fitxer de capçalera de la classe)

```
class Gos
{
   public:

   Gos(int edat);          //Mètodes constructors
   Gos();
   ~Gos();                 //Mètode destructor
```

ANOTACIONS

```
   int obtenirEdat();     // Mètodes d'accés
   int assignarEdat(int);
   int assignarAlcada(int);
   int obtenirAlcada();
   void bordar()          // Mètodes propis

   private:
   int edat;
   int alcada;
};
```

Fitxer gos.cpp (fitxer d'implementació de la classe)

```
#include <iostream>      //Necessària per al cout
#include <gos.hpp>

Gos:: bordar()
{ cout << "Bup"; }

void Gos:: assignarAlcada (int nAlcada)
{ alcada = nAlcada; }

int Gos:: obtenirAlcada (int nAlcada)
{ return (alcada); }

void Gos:: assignarEdat (int nEdat)
{ edat = nEdat; }

int Gos:: obtenirEdat ()
{ return (edat); }
```

Fitxer exemple.cpp

```
#include <gos.hpp>
int main()
{

    //Inicialitzant l'objecte amb edat 4.
    Gos sulta (4);
    sulta.bordar();
}
```

ANOTACIONS

Les biblioteques estàndard

Els compiladors solen incloure una sèrie de funcions addicionals perquè el programador les utilitzi. En el cas de GNU, proporciona una biblioteca estàndard de funcions i objectes per als programadors de C++ anomenada libstdc++.

Aquesta llibreria proporciona les operacions d'entrada/sortida amb corrents de dades o *streams*, cadenes o *strings*, vectors, llistes, algoritmes de comparació, operacions matemàtiques i algoritmes d'ordenació entre moltes altres.

Ús de la biblioteca STL

C++ ha incorporat un nivell més d'abstracció en introduir les **plantilles**, que també es coneixen com a **tipus parametritzats**. No és objecte d'aquest curs desenvolupar aquest tema, però la potència que proporciona al C++ la inclusió d'STL (biblioteca estàndard de plantilles) ens obliga a fer una breu ressenya de com s'ha d'utilitzar.

La idea bàsica de les plantilles és simple: quan s'implementa una operació general amb un objecte (per exemple, una llista de gossos) definirem les diferents operacions de manipulació d'una llista sobre la base de la classe Gos. Si posteriorment es vol fer una operació similar amb altres objectes (una llista de gats), el codi resultant per al manteniment de la llista és semblant però amb la diferència que defineix els elements partint de la classe Gat.

Segurament, la nostra manera d'actuar seria fer un copiar i enganxar i modificar el bloc copiat per treballar amb la nova classe que volem. No obstant això, aquest procés es repeteix cada vegada que es vol implementar una nova llista d'un altre tipus d'objectes (per exemple, una llista de cavalls). A més, cada vegada que es volgués modificar una operació de les llistes, s'hauria de canviar cada una de les seves personalitzacions. Per tant, ràpidament s'intueix que aquesta implementació no seria eficient.

La manera eficient de fer-ho és generar un codi genèric que faci les operacions de les llistes per a un tipus que s'hi pot indicar amb posterioritat. Aquest codi genèric és el de les plantilles o tipus parametritzats.

ANOTACIONS

Després d'aquest breu comentari, s'intueix l'eficiència i la potència d'aquesta nova característica i també la seva complexitat, que, com ja hem comentat, sobrepassa l'objectiu d'aquest curs. No obstant això, per a un domini avançat del C++ aquest tema és imprescindible i es recomana la consulta d'altres fonts bibliogràfiques per a completar aquests coneixements.

Malgrat tot, mentre que la definició i implementació d'una llibreria de plantilles pot arribar a adquirir una gran complexitat, l'ús de la llibreria estàndard de plantilles (STL) és accessible.

En l'exemple següent, es treballa amb la classe `set`, que defineix un conjunt d'elements. Per a això, s'ha inclòs la llibreria `set` recollida en l'STL.

```
#include <iostream>
#include <set>

int main()
{
// Defineix un conjunt d'enters <int>
set<int> setNombres;

//Afegeix tres nombres al conjunt de nombres
setNombres.insert(123);
setNombres.insert(789);
setNombres.insert(456);

// Mostra quants nombres té
// el conjunt de nombres
cout << "Conjunt nombres: ";
cout << setNombres.size() << endl;

// Es repeteix el procés amb un conjunt de lletres
//Defineix conjunt de caràcters <char>
set<char> setLletres;

setLletres.insert('a');
setLletres.insert('z');
cout  << "Conjunt lletres: ";
      << setLletres.size() << endl;
return 0;
}
```

ANOTACIONS

En l'exemple, s'han definit un conjunt de nombres i un conjunt de lletres. Per al conjunt de nombres, s'ha definit la variable `setNombres` utilitzant la plantilla `set` amb elements de tipus `<int>`. Aquest conjunt defineix diversos mètodes, entre els quals hi ha el d'inserir un element al conjunt (`.insert`) o explicar el nombre d'elements (`.size`). Per al segon s'ha definit la variable `setLletres` també utilitzant la mateixa plantilla `set` però ara amb elements tipus `<char>`.

La sortida del programa ens mostra el nombre d'elements introduïts en el conjunt de nombres i, posteriorment, el nombre d'elements introduïts en el conjunt de lletres.

4.4. Disseny de programes orientats a objectes

La potència del paradigma de la programació orientada a objectes no resideix només en la definició de classes i objectes, sinó en totes les conseqüències que impliquen i que es poden implementar en el llenguatge de programació.

En aquesta unitat s'estudiaran les principals propietats d'aquest paradigma:

- L'homonímia
- L'herència
- El polimorfisme

Una vegada assimilat l'abast del canvi de paradigma, es proporcionaran noves regles per al disseny d'aplicacions.

4.4.1. L'homonímia

Tal com la seva definició indica, homonímia es refereix a l'ús de dos o més sentits (en el nostre cas, llegiu operacions) amb el mateix nom.

En programació orientada a objectes, parlarem d'homonímia en utilitzar el mateix nom per a definir la mateixa operació diverses vegades en diferents situacions encara que, normalment, amb la mateixa

ANOTACIONS

idea de fons. L'exemple més clar és definir amb el mateix nom les operacions que bàsicament compleixen el mateix objectiu però per a objectes diferents.

En el nostre cas, diferenciarem entre les seves dues formes principals: l'**homonímia** (o **sobrecàrrega**) **de funcions** i l'**homonímia d'operadors**.

Sobrecàrrega de funcions i mètodes

La sobrecàrrega de funcions es va estudiar anteriorment com una de les millores que proporciona més flexibilitat a C++ respecte de C, i és una de les característiques més utilitzades dins de la definició de les classes.

Els constructors són un cas pràctic de sobrecàrrega de mètodes. Per a cada classe, es disposa d'un constructor per defecte que no té paràmetres i que inicialitza els objectes d'aquesta classe.

En el nostre exemple,

```
Gos::Gos()
{ }
```

O, tal com es va veure, es pot donar el cas que sempre es vulgui inicialitzar aquesta classe a partir d'una edat determinada, o d'una edat i una alçada determinades:

```
Gos::Gos(int nEdat) // Nova edat del gos
{ edat = nEdat; }

Gos::Gos(int nEdat, int nAlcada) // Nova defin.
{
   edat = nEdat;
   alcada = nAlcada;
}
```

En tots tres casos, es crea una instància de l'objecte `Gos`. Per tant, bàsicament estan fent la mateixa operació encara que el resultat final sigui lleugerament diferent.

ANOTACIONS

De la mateixa manera, qualsevol altre mètode o funció d'una classe pot ser sobrecarregat.

Sobrecàrrega d'operadors

En el fons, un operador no és més que una manera simple d'expressar una operació entre un o més operands, mentre que una funció ens permet fer operacions més complexes.

Per tant, la sobrecàrrega d'operadors ens permet simplificar les expressions en les operacions entre objectes.

En el nostre exemple, es podria definir una funció per a incrementar l'edat d'un objecte `Gos`.

```
Gos Gos::incrementarEdat()
{
   ++edat;
   return (*this);
}
// la crida resultant seria sulta.IncrementarEdat()
```

Encara que la funció és molt simple, podria resultar una mica incòmoda d'utilitzar. En aquest cas, podríem considerar sobrecarregar l'operador `++` perquè, en aplicar-lo sobre un objecte `Gos`, s'incrementés automàticament la seva edat.

La sobrecàrrega de l'operador es declara de la mateixa manera que una funció. S'utilitza la paraula reservada `operator`, seguida de l'operador que se sobrecarregarà. En el cas de les funcions d'operadors unitaris, no porten paràmetres (a excepció de l'increment o decrement postfix que utilitzen un enter com a diferenciador).

```
#include <iostream>
class Gos
{
   public:
     Gos();
     Gos(nEdat);
```

ANOTACIONS

```
    ~Gos();
    int obtenirEdat();
    const Gos & operator++();// Operador ++i
    const Gos & operator++(int);// Operador i++

  private:
    int edat;
};
Gos::Gos():
   edat(0)
{ }

Gos::Gos(int nEdat):
   edat(nEdat)
{ }

int Gos::obtenirEdat()
{ return (edat); }

const Gos & Gos::operator++()
{
   ++edat;
   return (*this);
}

const Gos & Gos::operator++(int x)
   {
   Gos temp = *this;
   ++edat;
   return (temp);
}

int main()
{
    Gos sulta(3);
    cout << "Edat en començar el programa \n " ;
    cout << sulta.obtenirEdat() << endl;

    ++sulta;
    cout << "Edat després d'un aniversari \n ";
```

ANOTACIONS

```
    cout << sulta.obtenirEdat() << endl;

    sulta++;
    cout << "Edat després d'un altre aniversari\n";
    cout << sulta.obtenirEdat();
}
```

En la declaració de sobrecàrrega dels operadors, s'observa com es torna una referència `const` a un objecte tipus `Gos`. D'aquesta manera, es protegeix l'adreça de l'objecte original de qualsevol canvi no volgut.

També és possible observar que les declaracions per a l'operador postfix i prefix són pràcticament iguals, i només canvia el tipus d'argument. Per a diferenciar tots dos casos, es va establir la convenció que l'operador postfix tingués en la declaració un paràmetre tipus `int` (encara que aquest paràmetre no s'usa).

```
const Gos & operator++();    // Operador ++i
const Gos & operator++(int); // Operador i++
```

En la implementació de totes dues funcions, també hi ha diferències significatives:

- En el cas de l'operador prefix, es procedeix a incrementar el valor de l'edat de l'objecte i es retorna l'objecte modificat per mitjà de l'apuntador `this`.

```
const Gos & Gos::operator++()
{
   ++edat;
   return (*this);
}
```

- En el cas de l'operador postfix, es requereix tornar el valor de l'objecte anterior a la seva modificació. Per aquest motiu, s'estableix una variable temporal que recull l'objecte original, es procedeix a la seva modificació i es retorna la variable temporal.

ANOTACIONS

```
const Gos & Gos::operator++(int x)
{
   Gos temp = *this;
   ++edat;
   return (temp);
}
```

La definició de la sobrecàrrega de l'operador suma, que és un operador binari, seria com segueix:

```
// En aquest cas, la suma de dos objectes tipus Gos
// no té CAP SENTIT LÒGIC.
// NOMÉS a l'efecte de mostrar com seria
// la declaració de l'operador, s'ha considerat
// com a resultat "possible"
// retornar l'objecte Gos de l'esquerra
// de l'operador suma amb l'edat corresponent
// a la suma de les edats dels dos gossos.

const Gos & Gos::operator+(const Gos & rhs)
{
   Gos temp = *this;
   temp.edat = temp.edat + rhs.edat;
   return (temp);
```

Nota

Atès el desconcertant de la lògica emprada en l'exemple anterior, també queda clar que no s'ha d'abusar de la sobrecàrrega d'operadors. Només s'ha d'utilitzar en aquells casos en què el seu ús sigui intuïtiu i ajudi a una llegibilitat més gran del programa.

4.4.2. L'herència simple

Els objectes no són elements aïllats. Quan s'estudien els objectes, s'estableixen relacions entre ells que ens ajuden a comprendre'ls millor.

ANOTACIONS

Exemple

Un gos i un gat són objectes diferents, però tenen una cosa en comú: tots dos són mamífers. També ho són els dofins i les balenes, encara que es moguin en un entorn molt diferent, encara que no els taurons, que entrarien dins de la categoria de peixos. Què tenen tots aquests objectes en comú? Tots són animals.

Es pot establir una jerarquia d'objectes, en què un gos és un mamífer, un mamífer és un animal, un animal és un ésser viu, etc. Entre ells s'estableix la relació *és un*. Aquest tipus de relació és molt habitual: un pèsol és una verdura, que és un tipus de vegetal; un disc dur és una unitat d'emmagatzematge, que al seu torn és un tipus de component d'un ordinador.

En dir que un element és un tipus de l'altre, establim una especialització: diem que l'element té unes característiques generals compartides i d'altres de pròpies.

L'herència és una manera de representar les característiques que es reben del nivell més general.

La idea de gos hereta totes les característiques de mamífer –és a dir, mama, respira amb pulmons, es mou, etc.–, però també presenta unes característiques concretes pròpies com bordar o moure la cua. Al seu torn, els gossos també es poden dividir segons la seva raça: pastor alemany, *caniche*, dòberman, etc. Cada un té les seves particularitats, però hereta totes les característiques dels gossos.

Per a representar aquestes relacions, C++ permet que una classe derivi d'una altra. En el nostre cas, la classe Gos deriva de la classe Mamífer. Per tant, en la classe Gos no caldrà indicar que mama, ni que respira amb pulmons, ni que es mou. En ser un mamífer, hereta aquestes propietats a més d'aportar dades o funcions noves.

De la mateixa manera, un Mamífer es pot implementar com una classe derivada de la classe Animal, i al seu torn hereta informació d'aquesta classe com la de pertànyer als éssers vius i moure's.

ANOTACIONS

Atesa la relació entre la classe Gos i la classe Mamífer, i entre la classe Mamífer i la classe Animal, la classe Gos també heretarà la informació de la classe Animal: un gos és un ésser viu que es mou!

Implementació de l'herència

Per a expressar aquesta relació en C++, en la declaració d'una classe després del seu nom es posen dos punts (:), el tipus de derivació (pública o privada) i el nom de la classe de la qual deriva:

```
class Gos : public Mamifer
```

El tipus de derivació, que de moment considerarem pública, s'estudiarà amb posterioritat. Ara enfocarem la nostra atenció sobre com queda la implementació nova:

```
#include <iostream>

enum RACES { PASTOR_ALEMANY, CANICHE,
             DOBERMAN, YORKSHIRE };

class Mamifer

{
   public:
    Mamifer();              // Mètode constructor

    ~Mamifer();             // Mètode destructor

   void assignarEdat(int nEdat)
    { edat = nEdat } ;      // Mètodes d'accés
   int obtenirEdat() const
    { return (edat) };

   protected:
    int edat;
};

class Gos : public Mamifer
{
   public:
    Gos();                  // Mètode constructor
```

ANOTACIONS

```
    ~Gos();                    // Mètode destructor
    void assignarRaca(RACES);// Mètodes d'accés
    int obtenirRaca() const;
    void bordar() const
      { cout << " Bup "; };    // Mètodes propis
  private:
    RACES raca;
};

class Gat : public Mamifer
{
  public:
    Gat();                    // Mètode constructor
    ~Gat();                   // Mètode destructor
    void miolar() const
      { cout << "Meu"; }      // Mètodes propis
};
```

En la implementació de la classe Mamífer, en primer lloc s'han definit el seu constructor i destructor per defecte. Atès que la dada membre `edat` de què disposàvem en la classe Gos no és una característica exclusiva d'aquesta classe, sinó que tots els mamífers tenen una edat, s'ha traslladat la dada membre `edat` i els seus mètodes d'accés (`obtenirEdat` i `assignarEdat`) a la nova classe.

Nota

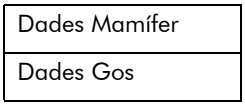
Es pot destacar que la declaració de tipus `protected` per a la dada membre `edat` permet que sigui accessible des de les classes derivades. Si s'hagués mantingut la declaració de `private`, no l'haurien pogut veure ni utilitzar altres classes, ni tan sols les derivades. Si s'hagués declarat `public`, hauria estat visible des de qualsevol objecte, però es recomana evitar aquesta situació.

Dins de la classe Gos, hem afegit com a dada nova la seva raça, i hem definit els seus mètodes d'accés (`obtenirRaca` i `assignarRaca`), i també el seu constructor i destructor predefinit. Es continua mantenint el mètode `bordar` com una funció de la classe Gos: els altres mamífers no borden.

ANOTACIONS

Constructors i destructors de classes derivades

En haver fet la classe Gos derivada de la classe Mamífer, en essència, els objectes `Gos` són objectes `Mamifer`. Per tant, en primer lloc crida el seu constructor base, amb la qual cosa es crea un Mamífer, i posteriorment es completa la informació cridant el constructor de Gos.



A l'hora de destruir un objecte de la classe Gos, el procés és l'invers: en primer lloc es crida el destructor de Gos ja que així s'allibera la informació específica i, posteriorment, es crida el destructor de Mamífer.

Fins al moment, sabem inicialitzar les dades d'un objecte de la classe que estem definint, però també és habitual que en el constructor d'una classe es vulgui inicialitzar dades pertanyents a la seva classe base.

Exemple

Seguint amb l'exemple, amb la classe Gos, a més d'inicialitzar la seva raça, també podem inicialitzar la seva edat (com que és un mamífer, té una edat).

El constructor de la classe Mamífer ja fa aquesta tasca, però és possible que només ens interessi fer aquesta operació per als gossos i no per a tots els animals. En aquest cas, podem fer la inicialització següent en el constructor de la classe Gos:

```
Gos :: Gos()
{
   assignarRaça(CANICHE);// Accés a raça
   assignarEdat(3);      // Accés a edat
};
```

En ser `assignarEdat` un mètode públic que pertany a la seva classe base, ja el reconeix automàticament.

En l'exemple anterior, hem definit dos mètodes –`bordar` i `miolar`– per a les classes Gos i Gat respectivament. La gran majoria dels animals tenen la capacitat d'emetre sons per a comunicar-se; per tant, es podria crear un mètode comú que podríem anomenar `emetreSo`, al

ANOTACIONS

qual podríem donar un valor general per a tots els animals, excepte per als gossos i els gats, cas en què es podria personalitzar:

```
#include <iostream>

enum RACES { PASTOR_ALEMANY, CANICHE,
             DOBERMAN, YORKSHIRE };

class Mamifer
{
   public:
    Mamifer();                  // Mètode constructor
                                // Mètode destructor
    ~Mamifer();
    void assignarEdat(int nEdat)
      { edat = nEdat; } ;       // Mètodes d'accés
    int obtenirEdat() const
      { return (edat); };
    void emetreSo() const
      { cout << "So"; };
    protected:
      int edat;
};

class Gos : public Mamifer
{
   public:
    Gos();                      // Mètode constructor
    ~Gos();                     // Mètode destructor
    void assignarRaca(RACES);// Mètodes d'accés
    int obtenirRaca() const;
    void emetreSo() const
    { cout << "Bup"; };        // Mètodes propis
   private:
    RACES raca;
};

class Gat : public Mamifer
{
   public:
    Gat();                      // Mètode constructor
```

ANOTACIONS

```
    ~Gat();                    // Mètode destructor
    void emetreSo () const
    { cout << "Meu"; }         // Mètodes propis
};

int main()
{
   Gos ungos;
   Gat ungat;
   Mamifer unmamifer;

   unmamifer.emetreSo;         // Resultat: "So"
   ungos.emetreSo;             // Resultat: Bup
   ungat.emetreSo;             // Resultat: Meu
}
```

El mètode `emetreSo` tindrà un resultat final segons si crida un Mamífer, un Gos o un Gat. En el cas de les classes derivades (Gos i Gat) es diu que s'ha **redefinit la funció** membre de la classe base. Per a això, la classe derivada ha de definir la funció base amb la mateixa signatura (tipus de retorn, paràmetres i els seus tipus, i l'especificador `const`).

Cal diferenciar la redefinició de funcions de la sobrecàrrega de funcions. En el primer cas, es tracta de funcions amb el mateix nom i la mateixa signatura en classes diferents (la classe base i la classe derivada). En el segon cas, són funcions amb el mateix nom i diferent signatura, que són dins de la mateixa classe.

El resultat de la redefinició de funcions és l'**ocultació** de la funció base per a les classes derivades. En aquest aspecte, cal tenir en compte que si es redefineix una funció en una classe derivada, quedaran ocultes també totes les sobrecàrregues d'aquesta funció de la classe base. Un intent d'usar alguna funció ocultada generarà un error de compilació. La solució consistirà a fer en la classe derivada les mateixes sobrecàrregues de la funció existents en la classe base.

ANOTACIONS

No obstant això, si es vol, encara es pot accedir al mètode ocultat anteposant al nom de la funció el nom de la classe base seguit de l'operador d'àmbit (::).

```
ungos.Mamifer::emetreSo();
```

4.4.3. El polimorfisme

En l'exemple que s'ha utilitzat fins al moment, només s'ha considerat que la classe Gos (classe derivada) hereta les dades i mètodes de la classe Mamífer (classe base). De fet, la relació existent és més forta.

C++ permet el tipus d'expressions següent:

```
Mamifer *ap_unmamifer = new Gos;
```

En aquestes expressions d'un apuntador a una classe Mamífer no hi assignem directament cap objecte de la classe Mamífer, sinó que hi assignem un objecte d'una classe diferent, la classe Gos, encara que es compleix que Gos és una classe derivada de Mamífer.

De fet, aquesta és la naturalesa del polimorfisme: un mateix objecte pot adquirir diferents formes. A un apuntador a un objecte mamífer s'hi pot assignar un objecte mamífer o un objecte de qualsevol de les seves classes derivades.

A més a més, com es podrà comprovar més endavant, aquesta assignació es podrà fer en temps d'execució.

Funcions virtuals

Amb l'apuntador que es presenta a continuació, es podrà cridar qualsevol mètode de la classe Mamífer. Però l'interessant seria que, en aquest cas concret, cridés els mètodes corresponents de la classe Gos. Això ens ho permeten les **funcions o mètodes virtuals**:

```
#include <iostream>

enum RACES { PASTOR_ALEMANY, CANICHE,
             DOBERMAN, YORKSHIRE };
```

ANOTACIONS

```
class Mamifer
{
   public:
    Mamifer();                    // Mètode constructor
    virtual ~Mamifer();           // Mètode destructor
    virtual void emetreSo() const
      { cout << "emetre un so" << endl; };
   protected:
    int edat;
};

class Gos : public Mamifer
{
   public:
    Gos();                        // Mètode constructor
    virtual ~Gos();               // Mètode destructor
    int obtenirRaca() const;
    virtual void emetreSo() const
      { cout << "Bup" << endl; };    // Mètodes propis
   private:
    RACES raca;
};

class Gat : public Mamifer
{
   public:
    Gat();                        // Mètode constructor
    virtual ~Gat();               // Mètode destructor
    virtual void emetreSo() const
      { cout << "Meu" << endl; };// Mètodes propis
};

class Vaca : public Mamifer
{
   public:
    Vaca();                       // Mètode constructor
    virtual ~Vaca();              // Mètode destructor
    virtual void emetreSo() const
    { cout << "Mu" << endl; };  //Mètodes propis
};
```

ANOTACIONS

```
int main()
{
   Mamifer * ap_Mamifers[3];
   int i;

   ap_Mamifers [0] = new Gat;
   ap_Mamifers [1] = new Vaca;
   ap_Mamifers [2] = new Gos;

   for (i=0; i<3 ; i++)
    ap_Mamifers[i]-> emetreSo();
return 0;
}
```

En executar el programa, en primer lloc es declara un vector de tres elements tipus apuntador a Mamífer, i s'inicialitza a diferents tipus de Mamífer (Gat, Vaca i Gos).

Posteriorment, es recorre aquest vector i es procedeix a cridar el mètode `emetreSo` per a cada un dels elements. La sortida obtinguda serà:

- `Meu`
- `Mu`
- `Bup`

El programa ha detectat en cada moment el tipus d'objecte que s'ha creat per mitjà del `new` i ha cridat la funció `emetreSo` corresponent.

Això hauria funcionat igualment encara que s'hagués demanat a l'usuari que indiqués al programa l'ordre dels animals. El funcionament intern es basa a detectar en temps d'execució el tipus de l'objecte a què s'apunta i aquest, en certa manera, substitueix les funcions virtuals de l'objecte de la classe base per les quals corresponguin a l'objecte derivat.

Per tot això, s'ha definit la funció membre `emetreSo` de la classe Mamífer com a funció virtual.

ANOTACIONS

Declaració de les funcions virtuals

En declarar una funció de la classe base com a virtual, implícitament s'estan declarant com a virtuals les funcions de les classes derivades, per la qual cosa no és necessària la seva declaració explícita com a tals. No obstant això, perquè el codi sigui més clar, es recomana fer-ho.

Si la funció no s'hagués declarat com a virtual, el programa entendria que ha de cridar la funció de la classe base, independentment del tipus d'apuntador que fos.

També és important destacar que, en tot moment, es tracta d'apuntadors a la classe base (encara que s'hagi inicialitzat amb un objecte de la classe derivada), amb la qual cosa només poden accedir a funcions de la classe base. Si un d'aquests apuntadors intentés accedir a una funció específica de la classe derivada, com per exemple `obtenirRaca()`, provocaria un error de funció desconeguda. A aquest tipus de funcions només s'hi pot accedir directament des d'apuntadors a objectes de la classe derivada.

Constructors i destructors virtuals

Per definició, els constructors no poden ser virtuals. En el nostre cas, en inicialitzar `new Gos` es crida el constructor de la classe Gos i el de la classe Mamífer, per la qual cosa ja crea un apuntador a la classe derivada.

En treballar amb aquests apuntadors, una operació possible és la seva eliminació. Per tant, ens interessarà que, per a la seva destrucció, primer es cridi el destructor de la classe derivada i després el de la classe base. Per a això, en tenim prou de declarar el destructor de la classe base com a virtual.

La regla pràctica que s'ha de seguir és declarar un destructor com a virtual en el cas que hi hagi alguna funció virtual dins de la classe.

ANOTACIONS

Tipus abstractes de dades i funcions virtuals pures

Ja hem comentat que les classes corresponen al nivell de les idees mentre que els objectes corresponen a elements concrets.

De fet, ens podem trobar classes en què no tingui sentit instanciar cap objecte, encara que sí que el tindria instanciar-los de classes derivades. És a dir, classes que volguéssim mantenir exclusivament en el món de les idees mentre les seves classes derivades generessin els nostres objectes.

Un exemple podria ser una classe `ObraDArt`, de la qual derivaran les subclasses Pintura, Escultura, Literatura, Arquitectura, etc. Es podria considerar la classe `ObraDArt` com un concepte abstracte i quan es tractés d'obres concretes, referir-nos-hi per mitjà d'una de les varietats d'art (les seves classes derivades). El criteri per a declarar una classe com a tipus de dades abstracte sempre és subjectiu i dependrà de l'ús que es vol que tinguin les classes en l'aplicació.

```
class ObraDArt
{
public:
   ObraDArt();
   virtual ~ObraDArt ();
   virtual void mostrarObraDArt() = 0; //virtual pura
   void assignarAutor(String autor);
   String obtenirAutor();
   String autor;
};

class Pintura : public ObraDArt
{
   public:
   Pintura();
   Pintura ( const Pintura & );
   virtual ~Pintura ();
   virtual void mostrarObraDArt();
   void assignarTitol(String titol);
   String obtenirTitol();
   private:
   String titol;
};
```

ANOTACIONS

```
Pintura :: mostrarObraDArt()
{ cout << "Fotografia Pintura \n" }
class Escultura : public ObraDArt
{
public:
   Escultura();
   Escultura ( const Escultura & );
   virtual ~Escultura ();
   virtual void mostrarObraDArt();
   void assignarTitol(String titol);
   String obtenirTitol();
private:
   String titol;
};

Escultura :: mostrarObraDArt()
{ cout << "Fotografia Escultura \n" }
```

Dins d'aquesta classe abstracta, s'ha definit una funció virtual que ens mostra una reproducció de l'obra d'art. Aquesta reproducció varia segons el tipus d'obra. Podria ser en forma de fotografia, de vídeo, d'una lectura d'un text literari o teatral, etc.

Per a declarar la classe `ObraDArt` com un tipus de dades abstracte, i per tant, no instanciable per cap objecte, només és necessari declarar una **funció virtual pura**.

Per a assignar una funció virtual pura, n'hi ha prou de prendre una funció virtual i assignar-la a 0:

```
virtual void mostrarObraDArt() = 0;
```

Ara, en intentar instanciar un objecte `ObraDArt` (mitjançant `new ObraDArt`) donaria error de compilació.

En declarar funcions virtuals pures s'ha de tenir en compte que aquesta funció membre també s'hereta. Per tant, en les classes derivades s'ha de redefinir aquesta funció. Si no es redefineix, la classe derivada es converteix automàticament en una altra classe abstracta.

ANOTACIONS

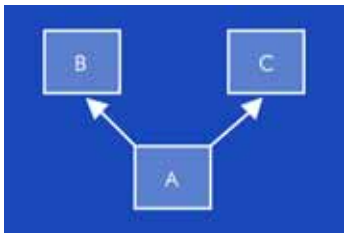
4.4.4. Operacions avançades amb herència

Malgrat la potència que s'entreveu en l'herència simple, hi ha situacions en què no és suficient. A continuació, es presenta una breu introducció als conceptes més avançats relatius a l'herència.

Herència múltiple

L'herència múltiple permet que una classe es derivi de més d'una classe base.

Figura 12.




```
class A : public B, public C
```

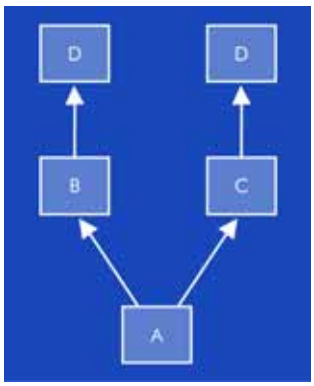
En aquest exemple, la classe A es deriva de la classe B i de la classe C. Davant d'aquesta situació, sorgeixen algunes preguntes:

- Què succeeix quan les dues classes derivades tenen una funció amb el mateix nom? Es podria produir un conflicte d'ambigüitat per al compilador que es pot resoldre afegint a la classe A una funció virtual que redefineixi aquesta funció, amb la qual cosa es resol explícitament l'ambigüitat.

- Què succeeix si les classes deriven d'una classe base comuna? Com que la classe A deriva de la classe D per part de B i per part de C, es produeixen dues còpies de la classe D (vegeu la il·lustració), la qual cosa pot provocar ambigüitats. La solució en aquest cas la proporciona l'**herència virtual**.

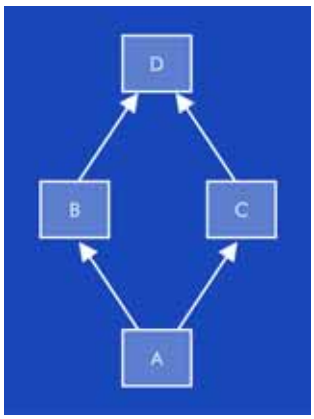
ANOTACIONS

Figura 13.



Mitjançant l'herència virtual s'indica al compilador que només es vol una classe base D compartida; per a això, les classes B i C es defineixen com a virtuals.

Figura 14.




```
class B: virtual D
class C: virtual D
class A : public B, public C
```

Generalment, una classe inicialitza només les seves variables i la seva classe base. En declarar una classe com a virtual, el constructor que inicialitza les variables correspon al de la classe més derivada.

Herència privada

De vegades no és necessari, o fins i tot no es vol, que les classes derivades tinguin accés a les dades o funcions de la classe base. En aquest cas s'utilitza l'herència privada.

ANOTACIONS

Amb l'herència privada les variables i funcions membres de la classe base es consideren com a privades, independentment de l'accessibilitat declarada en la classe base. Per tant, per a qualsevol funció que no sigui membre de la classe derivada són inaccessibles les funcions heretades de la classe base.

4.4.5. Orientacions per a l'anàlisi i disseny de programes

La complexitat dels projectes de programari actuals fa que s'utilitzi l'estratègia "divideix i venceràs" per a afrontar l'anàlisi d'un problema, descomponent el problema original en tasques més reduïdes i més fàcilment abordables.

La manera tradicional per a realitzar aquest procés es basa a descompondre el problema en funcions o processos més simples (disseny descendent), de manera que s'obtingui una estructura jeràrquica de processos i subprocessos.

Amb la programació orientada a objectes, la descomposició es fa de manera alternativa enfocant el problema cap als objectes que el componen i les seves relacions, i no cap a les funcions.

El procés que s'ha de seguir és el següent:

- **Conceptualitzar**. Els projectes normalment sorgeixen d'una idea que en guia el desenvolupament complet. És molt útil identificar l'objectiu general que es persegueix i vetllar perquè es mantingui en les diferents fases del projecte.
- **Analitzar**. És a dir, determinar les necessitats (requeriments) que ha de cobrir el projecte. En aquesta fase, l'esforç se centra a comprendre el domini (l'entorn) del problema en el món real (quins elements hi intervenen i com es relacionen) i capturar els requeriments.

  El primer pas per a aconseguir l'anàlisi de requeriments és identificar els **casos d'ús** que són descripcions en llenguatge natural dels diferents processos del domini. Cada cas d'ús descriu la interacció entre un actor (sia persona o element) i el sistema. L'actor

ANOTACIONS

Exemple

Si es dissenya una aplicació per a caixers automàtics, un cas d'ús podria ser retirar diners del compte.

envia un missatge al sistema i aquest actua conseqüentment (responent, cancel·lant, actuant sobre un altre element, etc.).

A partir d'un conjunt complet de casos d'ús, es pot començar a desenvolupar el **model del domini**, el document on es reflecteix tot el que es coneix sobre el domini. Com a part d'aquesta modelització, es descriuen tots els objectes que hi intervenen (que al final podran arribar a correspondre a les classes del disseny).

Exemple

En el cas del projecte del caixer automàtic, serien objectes el client, el caixer automàtic, el banc, el rebut, els diners, la targeta de crèdit, etc.

El model se sol expressar en **UML** (llenguatge de modelatge unificat), l'explicació del qual no és objectiu d'aquesta unitat.

Nota

UML només és una convenció comunament establerta per a representar la informació d'un model.

A partir dels casos d'ús, podem descriure diferents **escenaris**, circumstàncies concretes en què es desenvolupa el cas d'ús. D'aquesta manera es pot anar completant el conjunt d'interaccions possibles que ha de complir el nostre model. Cada escenari es caracteritza també en un entorn, amb unes condicions prèvies i elements que l'activen.

Exemple

En el cas del projecte del caixer automàtic, un possible escenari seria que el client volgués retirar els diners del compte i no hi hagués fons.

Tots aquests elements es poden representar gràficament mitjançant diagrames que mostrin aquestes interaccions.

A més a més, s'hauran de tenir en compte les restriccions que representen l'entorn en què funcionarà o altres requeriments proporcionats pel client.

- **Dissenyar.** A partir de la informació de l'anàlisi, s'enfoca el problema a crear la solució. Podem considerar el disseny com la conversió dels requeriments obtinguts en un model implementable en programari. El resultat és un document que conté el pla del disseny.

  En primer lloc, s'han d'identificar les classes que hi intervenen. Una primera (i simple) aproximació a la solució del problema consisteix a escriure els diferents escenaris, i crear una classe per a cada substantiu. Posteriorment, es pot reduir aquest nombre mitjançant l'agrupació dels sinònims.

  Una vegada definides les classes del model, podem afegir les classes que ens seran útils per a la implementació del projecte (les vistes, els

ANOTACIONS

informes, classes per a conversions o manipulacions de dades, ús de dispositius, etc.).

Una vegada establert el conjunt inicial de classes, que posteriorment es pot anar modificant, es pot procedir a modelar les relacions i interaccions entre elles. Un dels punts més importants en la definició d'una classe és determinar-ne les **responsabilitats**: un principi bàsic és que cada classe ha de ser responsable d'alguna cosa. Si s'identifica clarament aquesta responsabilitat única, el codi resultant serà més fàcil de mantenir. Les responsabilitats que no corresponen a una classe, les delega a les classes relacionades.

En aquesta fase també s'estableixen les **relacions** entre els objectes del disseny que poden coincidir, o no, amb els objectes de l'anàlisi. Poden ser de diferents tipus. El tipus de relació que més s'ha comentat en aquesta unitat són les relacions de generalització que posteriorment s'han implementat a partir de l'herència pública, però n'hi ha d'altres, cadascuna amb les seves formes d'implementació.

La informació del disseny es completa amb la inclusió de la **dinàmica** entre les classes: el modelatge de la interacció de les classes entre elles per mitjà de diversos tipus de diagrames gràfics.

El document que recull tota la informació sobre el disseny d'un progama s'anomena *pla de disseny*.

- **Implementar**. Perquè el projecte es pugui aplicar, s'ha de convertir el pla de disseny a un codi font, en el nostre cas, a C++. El llenguatge triat proporciona les eines i mecàniques de treball per poder traslladar totes les definicions de les classes, els seus requeriments, els seus atributs i les seves relacions des del món del disseny al nostre entorn real. En aquesta fase ens centrarem a codificar de manera eficient cada un dels elements del disseny.
- **Provar**. En aquesta fase es comprova que el sistema fa el que se n'espera. Si no és així, s'han de revisar les diferents especificacions d'anàlisi, disseny o implementació. El disseny d'un

ANOTACIONS

bon conjunt de proves basat en els casos d'ús ens pot evitar molts disgustos en el producte acabat. Sempre és preferible disposar d'un bon conjunt de proves que provoqui molts errors en les fases d'anàlisi, disseny o implementació (i per tant es poden corregir), que no pas trobar aquests errors en la fase de distribució.

- **Distribuir**. Es lliura al client una implementació del projecte perquè l'avaluï (del prototip) o perquè l'nstal·li definitivament.

Formes de desenvolupament d'un projecte

Normalment, el procés descrit es duu a terme mitjançant un **procés de cascada**: es va completant successivament cada una de les etapes i quan s'ha finalitzat i revisat, es passa a la següent sense possibilitat de retrocedir a l'etapa anterior.

Aquest mètode, que en teoria sembla perfecte, a la pràctica és fatal. Per això, en l'anàlisi i disseny orientat a objectes se sol utilitzar un **procés iteratiu**. En aquest procés, a mesura que es va desenvolupant el programari, es van repetint les etapes i cada vegada es procedeix a un refinament superior, la qual cosa permet adaptar-lo als canvis produïts pel coneixement més profund del projecte per part dels dissenyadors, dels desenvolupadors i del mateix client.

Aquest mètode també proporciona un altre avantatge en la vida real: es facilita el lliurament en la data prevista de versions completes encara que en un estat més o menys refinat. D'alguna manera, permet introduir la idea de versions "prou bones", i que posteriorment es poden anar refinant segons les necessitats del client.

4.5. Resum

En aquesta unitat, hem evolucionat des d'un entorn de programació C que segueix el model imperatiu –on pren com a base d'actuació les instruccions i la seva seqüència– a un model orientat a objectes –on la unitat base són els objectes i la seva interrelació.

ANOTACIONS

Per a això, hem hagut de comprendre els avantatges que comporta utilitzar un model de treball més abstracte, però més proper a la descripció de les entitats utilitzades en el món real i les seves relacions, i que ens permet enfocar la nostra atenció en els conceptes que es vol implementar més que en el detall final que impliquen les línies de codi.

Després hem estudiat les eines que proporciona C++ per implementar-se: les classes i els objectes. A més a més de la seva definició, també s'han revisat les propietats principals relatives al model d'orientació a objectes que proporciona C++. En concret, s'han estudiat l'herència entre classes, l'homonímia i el polimorfisme.

Finalment, hem vist que, a causa del canvi de filosofia en el nou paradigma de programació, no es poden aplicar els mateixos principis per al disseny dels programes i és per això que s'han introduït noves regles de disseny coherents amb ell.

4.6. Exercicis d'autoavaluació

1) Dissenyeu una aplicació que simuli el funcionament d'un ascensor. Inicialment, l'aplicació ha de definir tres operacions:

ASCENSOR: [1] Entrar [2] Sortir [0] Finalitzar

Després de cada operació, ha de mostrar l'ocupació de l'ascensor.

[1] Entrar correspon al fet que una persona entra a l'ascensor.
[2] Sortir correspon al fet que una persona surt de l'ascensor.

2) Amplieu l'exercici anterior per incorporar els requeriments següents:

- Una operació que sigui donar l'estat de l'ascensor

ASCENSOR: [1] Entrar [2] Sortir [3] Estat [0] Finalitzar

- Limitació de capacitat de l'ascensor a 6 places.
- Limitació de càrrega de l'ascensor a 500 kg.
- Petició de codi i pes dels usuaris per a permetre'ls accedir a l'ascensor segons els límits establerts.

ANOTACIONS

Si no es permet l'accés de l'usuari a l'ascensor, se li presenta un missatge amb els motius:

– < L'ascensor és complet >
– < L'ascensor superaria la càrrega màxima autoritzada >

Si es permet l'accés de l'usuari a l'ascensor, se li presenta el missatge següent: < #Codi# entra a l'ascensor>.

En entrar, com a mostra d'educació, saluda en general les persones de l'ascensor (< #codi# diu> Hola) si n'hi hagués, i elles li corresponen la salutació individualment (< #codi# respon> Hola).

En sortir un usuari de l'ascensor, s'ha de sol·licitar el seu codi i actualitzar la càrrega de l'ascensor mentre es presenta el següent missatge: < #codi# surt de l'ascensor>.

En sortir, l'usuari s'acomiada de les persones de l'ascensor (<#codi# diu> Adéu) si n'hi hagués, i elles li corresponen la salutació individualment (<#codi# respon> Adéu).

Per simplificar, considerarem que no hi pot haver mai dos passatgers amb el mateix codi.

Després de cada operació, s'ha de poder mostrar l'estat de l'ascensor (ocupació i càrrega).

3) Amplieu l'exercici anterior incorporant tres possibles idiomes en què els usuaris puguin saludar.

En entrar, s'ha de sol·licitar també quin és l'idioma de la persona:

IDIOMA: [1] Català [2] Castellà [3] Anglès

- En català, la salutació és "Bon dia" i el comiat, "Adéu".
- En castellà, la salutació és "Buenos días" i el comiat, "Adiós".
- En anglès, la salutació és "Hello" i el comiat, "Bye".

ANOTACIONS

4.6.1. Solucionari

1)

ascensor01.hpp

```
class Ascensor {
   private:
     int ocupacio;
   public:
     // Mètodes constructors i destructors
     Ascensor();
     ~Ascensor();

     // Funcions d'accés
     void mostrarOcupacio();
     int obtenirOcupacio();
     void modificarOcupacio(int difOcupacio);

     // Funcions del mètode
     bool persona_potEntrar();
     bool persona_potSortir();

     void persona_entrar();
     void persona_sortir();
};
```

ascensor01.cpp

```
#include <iostream>
#include "ascensor01.hpp"

// Mètodes constructors i destructors
Ascensor::Ascensor():
   ocupacio(0)
{ }

Ascensor::~Ascensor()
{ }

// Funcions d'accés
int Ascensor::obtenirOcupacio()
{ return (ocupacio); }
```

```
void Ascensor::modificarOcupacio(int difOcupacio)
{ ocupacio += difOcupacio; }

void Ascensor::mostrarOcupacio()
{ cout << "Ocupacio actual: " << ocupacio << endl;}

bool Ascensor::persona_potEntrar()
{ return (true); }

bool Ascensor::persona_potSortir()
{
   bool hihaOcupacio;
   if (obtenirOcupacio() > 0) hiHaOcupacio = true;
   else hiHaOcupacio = false;

   return (hihaOcupacio);
}

void Ascensor::persona_entrar()
{ modificarOcupacio(1); }

void Ascensor::persona_sortir()
{
   int ocupacioActual = obtenirOcupacio();
   if (ocupacioActual>0)
    modificarOcupacio(-1);
}
```

exerc01.cpp

```
#include <iostream>
#include "ascensor01.hpp"

int main(int argc, char *argv[])
{
   char opc;
   bool sortir = false;
   Ascensor unAscensor;
   do
   {
    cout << endl
    cout << "ASCENSOR: [1]Entrar [2]Sortir [0]Finalitzar ";
```

ANOTACIONS

```
    cin >> opc;
    switch (opc)
    {
    case '1':
     cout << "opc Entrar" << endl;
     unAscensor.persona_entrar();
     break;
    case '2':
     cout << "opc Sortir " << endl;
     if (unAscensor.persona_potSortir())
      unAscensor.persona_sortir();
     else
     cout << "Ascensor buit " << endl;
     break;
     case '0':
      sortir = true;
      break;
    }
    unAscensor.mostrarOcupacio();
  } while (! sortir);
  return 0;
}
```

2)

ascensor02.hpp

```
#ifndef _ASCENSOR02
#define _ASCENSOR02
#include "persona02.hpp"

class Ascensor {
  private:
    int ocupacio;
    int carrega;
    int ocupacioMaxima;
    int carregaMaxima;
    Persona *passatgers[6];
  public:
    // Constructors i destructors
    Ascensor();
    ~Ascensor();
```

ANOTACIONS

```
    // Funcions d'accés
    void mostrarOcupacio();
    int obtenirOcupacio();
    void modificarOcupacio(int difOcupacio);
    void mostrarCarrega();
    int obtenirCarrega();
    void modificarCarrega(int difCarrega);

    void mostrarLlistaPassatgers();

    // Funcions del mètode
    bool persona_potEntrar(Persona *);
    bool persona_seleccionar(Persona *localitzarPersona,
                             Persona **unaPersona);

    void persona_entrar(Persona *);
    void persona_sortir(Persona *);

    void persona_saludarRestaAscensor(Persona *);
    void persona_acomiadarseRestaAscensor(Persona *);
};
#endif
```

ascensor 02.cpp

```
#include <iostream>
#include "ascensor02.hpp"

//
// Mètodes constructors i destructors
//

// En el constructor, inicialitzem els valors màxims
// d'ocupació i càrrega màxima d'ascensor
// i el vector de passatgers a apuntadors NULL

Ascensor::Ascensor():
   ocupacio(0), carrega(0),
   ocupacioMaxima(6), carregaMaxima(500)
{ for (int i=0;i<=5;++i) {passatgers[i]=NULL;} }
```

ANOTACIONS

```
Ascensor::~Ascensor()
{ // Alliberar codis dels passatgers
   for (int i=0;i<=5;++i)
   { if (!(passatgers[i]==NULL)) {delete(passatgers[i]);} }
}

// Funcions d'accés
int Ascensor::obtenirOcupacio()
{ return (ocupacio); }

void Ascensor::modificarOcupacio(int difOcupacio)
{ ocupacio += difOcupacio; }

void Ascensor::mostrarOcupacio()
{ cout << "Ocupacio actual: " << ocupacio ; }

int Ascensor::obtenirCarrega()
{ return (carrega); }

void Ascensor::modificarCarrega(int difCarrega)
{ carrega += difCarrega; }

void Ascensor::mostrarCarrega()
{ cout << "Carrega actual: " << carrega ; }

bool Ascensor::persona_potEntrar(Persona *unaPersona)
{
   bool tmpPotEntrar;

   // Si l'ocupació no sobrepassa el límit d'ocupació i
   // si la càrrega no sobrepassa el límit de càrrega
   //-> pot entrar

   if (ocupacio + 1 > ocupacioMaxima)
   {
    cout << " Avís: Ascensor complet. No pot entrar. "
    cout << endl;
    return (false);
   }
```

ANOTACIONS

```
   if (unaPersona->obtenirPes() + carrega > carregaMaxima)
   {
    cout << "Avís: L'ascensor supera la seva càrrega màxima. ";
    cout << " No pot entrar. " << endl;
    return (false);
   }
   return (true);
}

bool Ascensor::persona_seleccionar(Persona *localitzarPersona,
                                   Persona **unaPersona)
{
   int comptador;
   // S'ha de seleccionar un passatger de l'ascensor.
   bool personaTrobada = false;
   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do
    {
     if (passatgers[comptador]!=NULL)
     {
      if ((passatgers[comptador]->obtenirCodi()==
         localitzarPersona->obtenirCodi() ))
      {
       *unaPersona=passatgers[comptador];
       personaTrobada=true;
       break;
      }
     }
     comptador++;
    } while (comptador<ocupacioMaxima);
    if (comptador>=ocupacioMaxima) {*unaPersona=NULL;}
   }
   return (personaTrobada);
}

void Ascensor::persona_entrar(Persona *unaPersona)
{
   int comptador;
   modificarOcupacio(1);
```

ANOTACIONS

```
   modificarCarrega(unaPersona->obtenirPes());
   cout << unaPersona->obtenirCodi();
   cout << " entra a l'ascensor " << endl;
   comptador=0;
   // Hem verificat anteriorment que hi ha places lliures
   do
   {
    if (passatgers[comptador]==NULL )
    {
     passatgers[comptador]=unaPersona;
     break;
    }
    comptador++;
   } while (comptador<ocupacioMaxima);
}

void Ascensor::persona_sortir(Persona *unaPersona)
{
int comptador;
   comptador=0;
   do
   {
    if ((passatgers[comptador]==unaPersona ))
    {
     cout << unaPersona->obtenirCodi();
     cout << " Surt de l'ascensor " << endl;
     passatgers[comptador]=NULL;
     // Modifiquem l'ocupació i la càrrega
     modificarOcupacio(-1);
     modificarCarrega(-1 * (unaPersona->obtenirPes()));
     break;
    }
    comptador++;
   } while (comptador<ocupacioMaxima);
   if (comptador == ocupacioMaxima)
   {
    cout << "Cap persona amb aquest codi. ";
    cout << "Ningú no surt de l'ascensor" << endl;}
}

void Ascensor::mostrarLlistaPassatgers()
{
   int comptador;
   Persona *unaPersona;
```

ANOTACIONS

```
   if (obtenirOcupacio() > 0)
   {
    cout << "Llista de passatgers de l'ascensor: " << endl;
    comptador=0;
    do
    {
     if (!(passatgers[comptador]==NULL ))
     {
      unaPersona=passatgers[comptador];
      cout << unaPersona->obtenirCodi() << "; ";
     }
     comptador++;
    } while (comptador<ocupacioMaxima);
    cout << endl;
   }
   else
   { cout << "L'ascensor es buit" << endl; }
}

void Ascensor::persona_saludarRestaAscensor( Persona *unaPersona)
{
   int comptador;
   Persona *unaAltraPersona;
   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do
    {
     if (!(passatgers[comptador]==NULL ))
     {
      unaAltraPersona=passatgers[comptador];
      if (!(unaPersona->obtenirCodi()==
         unaAltraPersona->obtenirCodi()))
      {
       cout << unaAltraPersona->obtenirCodi();
       cout << " respon: " ;
       unaAltraPersona->saludar();
       cout <<endl;
      }
```

ANOTACIONS

```
      }
     comptador++;
     } while (comptador<ocupacioMaxima);
   }
}

void Ascensor::persona_acomiadarseRestaAscensor( Persona *unaPersona)
{
   int comptador;
   Persona *unaAltraPersona;

   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do
    {
     if (!(passatgers[comptador]==NULL ))
     {
      unaAltraPersona=passatgers[comptador];
      if (!(unaPersona->obtenirCodi()==
         unaAltratraPersona->obtenirCodi()))
      {
       cout << unaAltraPersona->obtenirCodi();
       cout << " respon: " ;
       unaAltraPersona->acomiadarse();
       cout << endl;
      }
     }
     comptador++;
    } while (comptador<ocupacioMaxima);
   }
}
```

persona02.hpp

```
#ifndef _PERSONA02
#define _PERSONA02

class Persona
{
   private:
    int codi;
```

```
    int pes;
  public:
    // Mètodes constructors
    Persona();
    Persona(int codi, int pes);
    Persona(const persona &);
    ~Persona();

    // Funcions d'accés
    int obtenirCodi();
    void assignarCodi(int);
    int obtenirPes() const;
    void assignarPes(int nPes);
    void assignarPersona(int);
    void assignarPersona(int,int);
    void sollicitarCodi();

    void saludar();
    void acomiadarse();
};
#endif
```

persona02.cpp

```
#include <iostream>
#include "persona02.hpp"

Persona::Persona()
{ }
Persona::Persona(int nCodi, int nPes)
{
   codi = nCodi;
   pes = nPes;
}

Persona::~Persona()
{ }

int Persona::obtenirPes() const
{ return (pes); }
```

ANOTACIONS

```
void Persona::assignarPes(int nPes)
{ pes = nPes; }

int Persona::obtenirCodi()
{ return (codi); }

void Persona::assignarCodi(int nCodi)
{ codi= nCodi;}

void Persona::assignarPersona(int nCodi)
{ assignarCodi (nCodi);}

void Persona::assignarPersona(int nCodi, int nPes)
{
   assignarCodi(nCodi);
   assignarPes(nPes);
}

void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: acomiadarse ()
{ cout << "Adéu \n" ; };

void Persona::sollicitarCodi()
{
   int nCodi;
   cout << "Codi: ";
   cin >> nCodi;
   cout << endl;
   codi = nCodi;
}
```

exerc02.cpp

```
#include <iostream>
#include "ascensor02.hpp"
#include "persona02.hpp"

void sollicitarDades(int *nCodi, int *nPes)
```

ANOTACIONS

```
{
   cout << endl;
   cout << "Codi: ";
   cin >> *nCodi;
   cout << endl;
   cout << "Pes: ";
   cin >> *nPes;
   cout << endl;
}

int main(int argc, char *argv[])
{
   char opc;
bool sortir = false;
Ascensor unAscensor;
Persona * unaPersona;
Persona * localitzarPersona;

do
{
   cout << endl << "ASCENSOR: ";
   cout << " [1]Entrar [2]Sortir [3]Estat [0]Finalitzar ";
   cin >> opc;
   switch (opc)
   {
    case '1': // opcio Entrar
    {
     int nPes;
     int nCodi;

     sollicitarDades(&nCodi, &nPes);
     unaPersona = new Persona(nCodi, nPes);
     if (unAscensor.persona_potEntrar(unaPersona))
     {
      unAscensor.persona_entrar(unaPersona);
      if (unAscensor.obtenirOcupacio()>1)
      {
       cout << unaPersona->obtenirCodi();
       cout << " diu: " ;
       unaPersona->saludar();
       cout << endl; // Ara responen les altres
```

ANOTACIONS

```
        unAscensor.persona_saludarRestaAscensor(unaPersona);
      }
     }
     break;
    }
    case '2': // opció Sortir
    {
     unaPersona = NULL;
     localitzarPersona = new Persona;
     localitzarPersona->sollicitarCodi();
     if (unAscensor.persona_seleccionar (localitzarPersona,
                                         &unaPersona))
     {
        unAscensor.persona_sortir(unaPersona);
        if (unAscensor.obtenirOcupacio()>0)
        {
         cout << unaPersona->obtenirCodi()
         cout << " diu: " ;
         unaPersona->acomiadarse();
         cout << endl; // Ara responen les altres
         unAscensor.persona_acomiadarseRestaAscensor(unaPersona);
         delete (unaPersona);
        }
      }
      else
      {
       cout<<"No hi ha cap persona amb aquest codi";
       cout << endl;
      }
      delete localitzarPersona;
      break;
     }
     case '3': //Estat
     {
      unAscensor.mostrarOcupacio();
      cout <<"; // Per a separar ocupació de càrrega
      unAscensor.mostrarCarrega();
      cout << endl;
      unAscensor.mostrarLlistaPassatgers();
      break;
     }
     case '0':
```

ANOTACIONS

```
      {
       sortir = true;
       break;
      }
     }
   } while (! sortir);
   return 0;
}
```

3) `ascensor03.hpp` i `ascensor03.cpp` coincideixen amb `ascensor02.hpp` i `ascensor02.cpp` de l'exercici 2.

persona03.hpp

```
#ifndef _PERSONA03
#define _PERSONA03

class Persona
{
   private:
     int codi;
     int pes;
   public:
     // Mètodes constructors
     Persona();
     Persona(int codi, int pes);
     Persona(const persona &);
     ~Persona();

     // Funcions d'accés
     int obtenirCodi();
     void assignarCodi(int);
     int obtenirPes() const;
     void assignarPes(int nPes);
     void assignarPersona(int,int);
     void sollicitarCodi();
     virtual void saludar();
     virtual void acomiadarse();
};

class Catala: public Persona
{
```

ANOTACIONS

```
   public:
    Catala()
    {
     assignarCodi (0);
     assignarPes (0);
    };

    Catala(int nCodi, int nPes)
    {
     assignarCodi (nCodi);
     assignarPes (nPes);
    };

    virtual void saludar()
    { cout << "Bon dia"; };

    virtual void acomiadarse()
    { cout << "Adéu"; };
};

class Castella: public Persona
{
   public:
    Castella()
    {
     assignarCodi (0);
     assignarPes (0);
    };

    Castella(int nCodi, int nPes)
    {
     assignarCodi (nCodi);
     assignarPes (nPes);
    };

    virtual void saludar()
    { cout << "Buenos días"; };

    virtual void acomiadarse()
    { cout << "Adiós"; };
};
```

ANOTACIONS

```
class Angles : public Persona
{
   public:
    Angles()
    {
     assignarCodi (0);
     assignarPes (0);
    };

    Angles(int nCodi, int nPes)
    {
     assignarCodi (nCodi);
     assignarPes (nPes);
    };

    virtual void saludar()
    { cout << "Hello"; };

    virtual void acomiadarse()
    { cout << "Bye"; };
};
#endif
```

persona03.cpp

```
#include <iostream>
#include "persona03.hpp"

Persona::Persona()
{ }

Persona::Persona(int nCodi, int nPes)
{
   codi = nCodi;
   pes = nPes;
}

Persona::~Persona()
{ }
```

ANOTACIONS

```
int Persona::obtenirPes() const
{ return (pes); }

void Persona::assignarPes(int nPes)
{ pes = nPes; }

int Persona::obtenirCodi()
{ return (codi); }

void Persona::assignarCodi(int nCodi)
{ this->Codi = nCodi; }

void Persona::assignarPersona(int nCodi, int nPes)
{
   assignarCodi(nCodi);
   assignarPes(nPes);
}

void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: acomiadarse ()
{ cout << "Adéu \n" ; };

void Persona::sollicitarCodi
{
   int nCodi;

   cout << "Codi: ";
   cin >> nCodi;
   cout << endl;

   assignarCodi (nCodi);
}
```

exerc03.cpp

```
#include <iostream>
#include "ascensor03.hpp"
#include "persona03.hpp"
```

ANOTACIONS

```
void sollicitarDades(int *nCodi, int *nPes, int *nIdioma)
{
   cout << endl;
   cout << "codi: ";
   cin >> *nCodi;
   cout << endl;
   cout << "Pes: ";
   cin >> *nPes;
   cout << endl;
   cout << "Idioma: [1] Català [2] Castellà [3] Anglès ";
   cin >> *nIdioma;
   cout << endl;
}

int main(int argc, char *argv[])
{
   char opc;
   bool sortir = false;
   Ascensor unAscensor;
   Persona * unaPersona;
   Persona * localitzarPersona;

do
{
   cout << endl << "ASCENSOR: ";
   cout << " [1]Entrar [2]Sortir [3]Estat [0]Finalitzar";
   cin >> opc;
   switch (opc)
   {
    case '1': // Opcio Entrar
    {
     int nPes;
     int nCodi;
     int nIdioma;
     sollicitarDades(&nCodi, &nPes, &nIdioma);
     switch (nIdioma)
     {
      case 1:
      {
       unaPersona = new Catala(nCodi, nPes);
       break;
```

ANOTACIONS

```
  }
  case 2:
  {
   unaPersona = new Castella(nCodi, nPes);
   break;
  }
  case 3:
  {
   unaPersona = new Angles(nCodi, nPes);
   break;
  }
}
if (unAscensor.persona_potEntrar(unaPersona))
{
 unAscensor.persona_entrar(unaPersona);
 if (unAscensor.obtenirOcupacio()>1)
 {
  cout << unaPersona->obtenirCodi();
  cout << " diu: " ;
  unaPersona->saludar();
  cout << endl; // Ara responen les altres
  unAscensor.persona_saludarRestaAscensor (unaPersona);
 }
}
 break;
}
case '2': //Opció Sortir
{
 localitzarPersona = new Persona;
 unaPersona = NULL;

 localizarPersona->sollicitarCodi();
 if (unAscensor.persona_seleccionar(localitzarPersona,
     & unaPersona))
 {
  unAscensor.persona_sortir(unaPersona);
  if (unAscensor.obtenirOcupacio()>0)
  {
   cout << unaPersona->obtenirCodi();
   cout << " diu: " ;
   unaPersona->acomiadarse();
```

ANOTACIONS

```
          cout << endl; // Ara responen les altres
          unAscensor.persona_acomiadarseRestaAscensor (unaPersona);
          delete (unaPersona) ;
        }
       }
       else
       {
        cout<<"No hi ha cap persona amb aquest codi";
        cout << endl;
       }
       delete localitzarPersona;
       break;
      }
      case '3': //Estat
      {
       unAscensor.mostrarOcupacio();
       cout " - "; // Per a separar Ocupació de Càrrega
                   unAscensor.mostrarCarrega();
                   cout << endl;
                   unAscensor.mostrarLlistaPassatgers();
                   break;
                  }
                  case '0':
                  {
                    sortir = true;
                    break;
                  }
                 }
   } while (! sortir);
   return 0;
}
```

5. Programació en Java

5.1. Introducció

En els capítols anteriors, s'ha mostrat l'evolució que han experimentat els llenguatges de programació en la història i que han anat desembocant en els diferents paradigmes de programació.

Inicialment, el cost d'un sistema informàtic estava marcat principalment pel maquinari: els components interns dels ordinadors eren voluminosos, lents i cars. En comparació, el cost que generaven les persones que intervenien en el seu manteniment i en el tractament de la informació era gairebé menyspreable. A més a més, per limitacions físiques, la mena d'aplicacions que es podien manejar era més aviat simple. L'èmfasi en la investigació en informàtica se centrava bàsicament a aconseguir sistemes més petits, més ràpids i més barats.

Amb el temps, aquesta situació ha canviat radicalment. La revolució produïda en el món del maquinari ha permès la fabricació d'ordinadors que no es podia ni somiar fa vint-i-cinc anys, però aquesta revolució no ha tingut la seva correspondència en el món del programari. En aquest aspecte, els costos materials s'han reduït considerablement, mentre que els relatius a personal han augmentat progressivament. També s'ha incrementat la complexitat en l'ús del programari, entre altres coses a causa de l'augment d'interactivitat amb l'usuari.

En l'actualitat moltes de les línies d'investigació busquen millorar el rendiment en la fase de desenvolupament de programari on, de moment, la intervenció humana és fonamental. Molt d'aquest esforç se centra en la generació de codi correcte i en la reutilització de la feina feta.

En aquest camí, el paradigma de la programació orientada a objectes ha representat una gran aproximació entre el procés de desenvolupament d'aplicacions i la realitat que intenten representar. D'altra

ANOTACIONS

banda, la incorporació de la informàtica a molts components que ens envolten també ha augmentat molt el nombre de plataformes diverses sobre les quals es pot desenvolupar programes.

Java és un llenguatge modern que ha nascut per donar solució a aquest entorn nou. Bàsicament, és un llenguatge orientat a objectes pensat per a treballar en múltiples plataformes. El seu plantejament consisteix a crear una plataforma comuna intermèdia per a la qual es desenvolupen les aplicacions i, després, traslladar el resultat generat per a aquesta plataforma comuna a cada màquina final.

Aquest pas intermedi permet:

– Escriure l'aplicació **només una vegada**. Una vegada compilada cap a aquesta plataforma comuna, l'aplicació es podrà executar amb tots els sistemes que tinguin aquesta plataforma intermèdia.

– Escriure la plataforma comuna **només una vegada**. En aconseguir que una màquina real sigui capaç d'executar les instruccions de la plataforma comuna –és a dir, que sigui capaç de traslladar-les al sistema subjacent–, s'hi podran executar totes les aplicacions desenvolupades per a aquesta plataforma.

Per tant, s'aconsegueix el nivell de reutilització màxim. El preu és el sacrifici d'una part de la velocitat.

En l'ordre de la generació de codi correcte, Java disposa de diverses característiques que anirem veient al llarg d'aquesta unitat. En tot cas, de moment es vol destacar que Java es basa en C++, amb la qual cosa s'aconsegueix més facilitat d'aprenentatge per a un gran nombre de desenvolupadors (reutilització del coneixement), però se l'allibera de moltes de les cadenes que C++ arrossegava per la seva compatibilitat amb el C.

Aquesta "netedat" té conseqüències positives:

– El llenguatge és més simple, ja que s'eliminen conceptes complexos que rares vegades s'utilitzen.

ANOTACIONS

– El llenguatge és més directe. S'ha estimat que Java permet reduir el nombre de línies de codi a la quarta part.

– El llenguatge és més pur, ja que només permet treballar en el paradigma de l'orientació a objectes.

A més, la joventut del llenguatge hi ha permès incorporar dins del seu nucli algunes característiques que senzillament no existien quan es van crear altres llenguatges, com les següents:

– La programació de fils d'execució, que permet aprofitar les arquitectures amb multiprocessadors.

– La programació de comunicacions (TCP/IP, etc.), que facilita el treball en xarxa, sia local o Internet.

– La programació de miniaplicacions (o *applets*) pensades per a ser executades per un navegador web.

– El suport per a crear interfícies gràfiques d'usuari i un sistema de gestió d'incidències, que faciliten la creació d'aplicacions seguint el paradigma de la programació dirigida per incidències.

En aquesta unitat es vol introduir el lector en aquest nou entorn de programació i presentar les seves principals característiques, i es pretén que, partint dels seus coneixements del llenguatge C++, aconsegueixi els objectius següents:

1) Conèixer l'entorn de desenvolupament de Java.

2) Ser capaç de programar en Java.

3) Entendre els conceptes de l'ús dels fils d'execució i la seva aplicació en l'entorn Java.

4) Comprendre les bases de la programació dirigida per incidències i ser capaç de desenvolupar exemples simples.

5) Poder crear miniaplicacions simples.

ANOTACIONS

5.2. Origen de Java

El 1991, enginyers de Sun Microsystems intentaven introduir-se en el desenvolupament de programes per a electrodomèstics i petits equips electrònics on la potència de càlcul i memòria era reduïda. Això requeria un llenguatge de programació que, principalment, aportés fiabilitat del codi i facilitat de desenvolupament, i es pogués adaptar a múltiples dispositius electrònics.

Nota

Per la varietat de dispositius i processadors existents en el mercat i els seus continus canvis buscaven un entorn de treball que no depengués de la màquina en què s'executés.

Per a això van dissenyar un esquema basat en una plataforma intermèdia sobre la qual funcionaria un nou codi màquina executable, i aquesta plataforma s'encarregaria de la translació al sistema subjacent. Aquest codi màquina genèric estaria molt orientat a la manera de funcionar de la majoria d'aquests dispositius i processadors, per la qual cosa la translació final havia de ser ràpida.

El procés complet consistiria, doncs, a escriure el programa en un llenguatge d'alt nivell i compilar-lo per generar codi genèric (els *bytecodes*) preparat per a ser executat per aquesta plataforma (la "màquina virtual"). D'aquesta manera s'aconseguiria l'objectiu de poder escriure el codi una sola vegada i poder-lo executar a tot arreu on estigués disponible la plataforma (*write once, run everywhere*).

Tenint aquestes referències, el seu primer intent va ser utilitzar C++, però per la seva complexitat van sorgir nombroses dificultats, per la qual cosa van decidir dissenyar un nou llenguatge basant-se en C++ per facilitar el seu aprenentatge. Aquest llenguatge nou havia de recollir, a més, les propietats dels llenguatges moderns i reduir la seva complexitat eliminant les funcions no absolutament imprescindibles.

ANOTACIONS

El projecte de creació d'aquest llenguatge nou va rebre el nom inicial d'*Oak*, però com que el nom estava registrat, es va rebatejar amb el nom final de Java. Conseqüentment, la màquina virtual capaç d'executar aquest codi en qualsevol plataforma va rebre el nom de *màquina virtual de Java* (JVM o *Java virtual machine*).

Els primers intents d'aplicació comercial no van fructificar, però el desenvolupament d'Internet va fomentar tecnologies multiplataforma, per la qual cosa Java es va revelar com una possibilitat interessant per a la companyia. Després d'una sèrie de modificacions de disseny per a adaptar-lo, Java es va presentar per primera vegada com a llenguatge per a ordinadors l'any 1995, i el gener de 1996, Sun va formar l'empresa Java Soft per desenvolupar productes nous en aquest entorn nou i facilitar la col·laboració amb terceres parts. El mateix mes es va donar a conèixer una primera versió, bastant rudimentària, de l'equip de desenvolupament de Java, el JDK 1.0.

Al començament de 1997 va aparèixer la primera revisió Java, la versió 1.1, que millorava considerablement les prestacions del llenguatge, i al final de 1998 va aparèixer la revisió Java 1.2, que va introduir canvis significatius. Per aquest motiu, a aquesta versió i posteriors se les coneix com a plataformes Java 2. El desembre de 2003, l'última versió de la plataforma Java2 disponible per a la seva baixada a la pàgina de Sun és Java 1.4.2.

Nota

Podeu trobar aquesta versió en l'adreça següent: http://java.sun.com.

La vertadera revolució que va impulsar definitivament l'expansió del llenguatge la va causar la incorporació el 1997 d'un intèrpret de Java al navegador Netscape.

5.3. Característiques generals de Java

Sun Microsystems descriu Java com un llenguatge simple, orientat a objectes, distribuït, robust, segur, d'arquitectura neutra, portable, interpretat, d'alt rendiment, multitasca i dinàmic.

ANOTACIONS

Analitzem aquesta descripció:

- **Simple**. Per a facilitar l'aprenentatge, es va considerar que els llenguatges més utilitzats pels programadors eren el C i el C++.

  Descartat el C++, es va dissenyar un nou llenguatge que els fos molt proper per a facilitar la seva comprensió.

  Amb aquest objectiu, Java elimina una sèrie de característiques poc utilitzades i de difícil comprensió del C++, com, per exemple, l'herència múltiple, les coercions automàtiques i la sobrecàrrega d'operadors.

- **Orientat a objectes**. En poques paraules, el disseny orientat a objectes enfoca el disseny cap a les dades (objectes), les seves funcions i interrelacions (mètodes). En aquest punt, se segueixen essencialment els mateixos criteris que C++.

- **Distribuït**. Java inclou una àmplia llibreria de rutines que permeten treballar fàcilment amb els protocols de TCP/IP com HTTP o FTP. Es poden crear connexions per mitjà de la xarxa a partir d'adreces URL amb la mateixa facilitat que treballant de manera local.

- **Robust**. Un dels propòsits de Java és buscar la fiabilitat dels programes. Per a això, es va posar l'èmfasi en tres fronts:

– Estricte control en temps de compilació amb l'objectiu de detectar els problemes com més aviat millor. Per a això, utilitza una estratègia de fort control de tipus, com en C++, encara que evita alguns dels seus forats normalment deguts a la seva compatibilitat amb C. També permet el control de tipus en temps d'enllaç.

– Revisió en temps d'execució dels possibles errors dinàmics que es poden produir.

– Eliminació de situacions propenses a generar errors. El cas més significatiu és el control dels apuntadors. Per a això, els tracta com a vectors vertaders, controlant els valors possibles d'índexs. En evitar l'aritmètica d'apuntadors (sumar desplaçament a una posició de memòria sense controlar els seus límits), s'evita la possibilitat de sobreescriptura de memòria i corrupció de dades.

ANOTACIONS

- **Segur**. Java està orientat a entorns distribuïts en xarxa i, per aquest motiu, s'ha posat molt d'èmfasi en la seguretat contra virus i intrusions, i en l'autenticació.

- **Arquitectura neutra**. Per a poder funcionar sobre una gran varietat de processadors i arquitectures de sistemes operatius, el compilador de Java proporciona un codi comú executable des de qualsevol sistema que tingui la presència d'un sistema en temps d'execució de Java.

  Això evita que els autors d'aplicacions hagin de produir versions per a sistemes diferents (com PC, Apple Macintosh, etc.). Amb Java, el mateix codi compilat funciona per a tots.

  Per a això, Java genera instruccions *bytecodes* dissenyades per a ser interpretades fàcilment per una plataforma intermèdia (la màquina virtual de Java) i traduïdes a qualsevol codi màquina natiu al vol.

- **Portable.** L'arquitectura neutra ja proporciona un gran avenç respecte a la portabilitat, però no és l'únic aspecte que s'ha cuidat.

  Exemple

  En Java no hi ha detalls que depenguin de la implementació, com podria ser la mida dels tipus primitius. En Java, a diferència de C o C++, el tipus `int` sempre es refereix a un nombre enter de 32 bits amb complement a 2 i el tipus `float`, a un nombre de 32 bits seguint la norma IEEE 754.

  La portabilitat també és determinada per les llibreries. Per exemple, hi ha una classe Windows abstracta i les seves implementacions per a Windows, Unix o Macintosh.

- **Interpretat**. Els *bytecodes* en Java es tradueixen en temps d'execució a instruccions de la màquina nativa (s'interpreten) i no s'emmagatzemen en cap lloc.

- **Alt rendiment**. De vegades es requereix millorar el rendiment produït per la interpretació dels *bytecodes*, que ja és bastant bo per si mateix. En aquests casos, és possible traduir-los en temps d'execució al codi natiu de la màquina on l'aplicació s'està executant. És a dir, compilar el llenguatge de la JVM al llenguatge de la màquina en què s'hagi d'executar el programa.

ANOTACIONS

D'altra banda, els *bytecodes* s'han dissenyat pensant en el codi màquina, i per això el procés final de la generació de codi màquina és molt simple. A més a més, la generació dels *bytecodes* és eficient i s'hi apliquen diversos processos d'optimització.

- **Multitasca**. Java proporciona, dins del mateix llenguatge, eines per construir aplicacions amb múltiples fils d'execució, cosa que simplifica el seu ús i el fa més robust.
- **Dinàmic**. Java es va dissenyar per a adaptar-se a un entorn canviant. Per exemple, un efecte lateral del C++ es produeix a causa de la manera com s'ha implementat el codi. Si un programa utilitza una llibreria de classes i aquesta canvia, cal recompilar tot el projecte i tornar-lo a redistribuir. Java evita aquests problemes en fer les interconnexions entre els mòduls més tard, cosa que permet afegir nous mètodes i instàncies sense tenir cap efecte sobre els seus clients.

  Mitjançant les interfícies s'especifiquen un conjunt de mètodes que un objecte pot fer, però deixa oberta la manera com els objectes poden implementar aquests mètodes. Una classe Java pot implementar múltiples interfícies, encara que només pot heretar d'una única classe. Les interfícies proporcionen flexibilitat i reusabilitat connectant objectes segons el que volem que facin i no pel que fan.

  Les classes en Java són representades en temps d'execució per una classe anomenada *Class*, que conté les definicions de les classes en temps d'execució. D'aquesta manera, es poden fer comprovacions de tipus en temps d'execució i es pot confiar en els tipus en Java, mentre que en C++ el compilador només confia que el programador faci el que és correcte.

5.4. L'entorn de desenvolupament de Java

Per a desenvolupar un programa en Java, en el mercat hi ha diverses opcions comercials. No obstant això, la companyia Sun distribueix de manera gratuïta el Java Development Kit (JDK), que és un conjunt de programes i llibreries que permeten el desenvolupament, la

ANOTACIONS

compilació i l'execució d'aplicacions en Java a més de proporcionar un depurador o *debugger* per al control d'errors.

També hi ha eines que permeten la integració de tots els components anteriors (IDE - *integrated development environment*), d'utilització més agradable, encara que poden presentar errors de compatibilitat entre plataformes o fitxers resultants no tan optimitzats. Per aquest motiu, i per a familiaritzar-se millor amb tots els processos de la creació de programari, s'ha optat en aquest material per desenvolupar les aplicacions directament amb les eines proporcionades per Sun.

Nota

Entre els IDE disponibles actualment es pot destacar el projecte Eclipse, que, seguint la filosofia de codi obert, ha aconseguit un paquet de desenvolupament molt complet (SDK o *standard development kit*) per a diversos sistemes operatius (Linux, Windows, Sun, Apple, etc.).

Aquest paquet està disponible per a baixar-lo a: http://www.eclipse.org.

Un altre IDE interessant és JCreator, que a més de desenvolupar una versió comercial, disposa d'una versió limitada, de fàcil maneig.

Aquest paquet està disponible per a baixar-lo a: http://www.jcreator.com.

Una altra característica particular de Java és que es poden generar diversos tipus d'aplicacions:

- **Aplicacions independents**. Un fitxer que s'executa directament sobre la màquina virtual de la plataforma.
- **Miniaplicacions**. Miniaplicacions que no es poden executar directament sobre la màquina virtual, sinó que estan pensades per a ser carregades i executades des d'un navegador web. Per aquest motiu, incorpora unes limitacions de seguretat extremes.

ANOTACIONS

- **Miniaplicacions de servidor**. Miniaplicacions sense interfície d'usuari per a executar-se des d'un servidor i que té la funció de donar resposta a les accions de navegadors remots (petició de pàgines HTML, tramesa de dades d'un formulari, etc.). La seva sortida generalment és per mitjà de fitxers, com per exemple, fitxers HTML.

Per a generar qualsevol dels tipus d'aplicacions anteriors, només es necessita el següent:

- Un editor de textos on escriure el codi font en llenguatge Java.

- La plataforma Java, que permet la compilació, depuració, execució i documentació d'aquests programes.

5.4.1. La plataforma Java

Entenem com a plataforma l'entorn maquinari o programari que necessita un programa per a executar-se. Encara que la majoria de plataformes es poden descriure com una combinació de sistema operatiu i maquinari, la plataforma Java es diferencia d'altres en el fet que es compon d'una plataforma programari que funciona sobre altres plataformes basades en el maquinari (GNU/Linux, Solaris, Windows, Macintosh, etc.).

La plataforma Java té dos components:

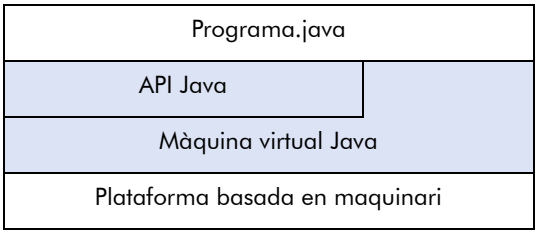
- Màquina virtual (MV). Com ja hem comentat, una de les principals característiques que proporciona Java és la independència de la plataforma maquinari: una vegada compilats, els programes s'han de poder executar en qualsevol plataforma.

  L'estratègia utilitzada per a aconseguir-ho és generar un codi executable "neutre" (*bytecode*) com a resultat de la compilació. Aquest codi neutre, que està molt orientat al codi màquina, s'executa des d'una "màquina hipotètica" o "màquina virtual". Per a executar un programa en una plataforma determinada n'hi ha prou de disposar d'una "màquina virtual" per a aquesta plataforma.

ANOTACIONS

- *Application programming interface* (API). L'API de Java és una gran col·lecció de programari ja desenvolupat que proporciona múltiples capacitats com entorns gràfics, comunicacions, multiprocés, etc. És organitzat en llibreries de classes relacionades i interfícies. Les llibreries reben el nom de *paquets informàtics* (*packages*).

En l'esquema següent, es pot observar l'estructura de la plataforma Java i com la màquina virtual aïlla el codi font (.java) del maquinari de la màquina:



5.4.2. El nostre primer programa en Java

Una altra vegada el nostre primer contacte amb el llenguatge serà mostrant una salutació al món. El desenvolupament del programa es dividirà en tres fases:

1) **Crear un fitxer font**. Mitjançant l'editor de textos escollit, escriurem el text i el desarem amb el nom *HolaMon.java*.

HolaMon.java

```
/**
 * La classe HolaMon mostra el missatge
 * "Hola Món" en la sortida estàndard.
 */
public class HolaMon {
    public static void main(String[] args) {
      // Mostra "Hola Món! "
      System.out.println("Hola Mon! ");
    }
}
```

2) **Compilar el programa** i generar un fitxer *bytecode*. Per a això, utilitzarem el compilador **javac**, que ens proporciona l'entorn de

ANOTACIONS

desenvolupament i que tradueix el codi font a instruccions que la JVM pugui interpretar.

Si després d'escriure "`javac HolaMon.java`" en l'intèrpret d'ordres, no es produeix cap error, obtenim el nostre primer programa en Java: un fitxer *HolaMon.class*.

3) **Executar el programa** en la màquina virtual de Java. Una vegada generat el fitxer de *bytecodes*, per a executar-lo en la JVM només haurem d'escriure la instrucció següent, perquè el nostre ordinador el pugui interpretar, i ens apareixerà en pantalla el missatge de benvinguda Hola món!:

```
java HolaMon
```

5.4.3. Les instruccions bàsiques i els comentaris

En aquest punt, Java es continua mantenint fidel a C++ i C i conserva la seva sintaxi.

L'única consideració que s'ha de tenir en compte és que, en Java, les expressions condicionals (per exemple, la condició `if`) han de retornar un valor de tipus *boolean*, mentre que C++, per compatibilitat amb C, permetia el retorn de valors numèrics i assimilava 0 a *false* i els valors diferents de 0 a *true*.

Respecte als comentaris, Java admet les formes provinents de C++ (/*... */ i // ...) i n'afegeix una de nova: incloure el text entre les seqüències /* * (inici de comentari) i */ (final de comentari).

De fet, la utilitat d'aquesta nova forma no és tant la de comentar com la de documentar. Java proporciona eines (per exemple, *javadoc*) per generar documentació a partir dels codis fonts que extreuen el contingut dels comentaris fets seguint aquest model.

Exemple

```
/**
* Text comentat amb la nova forma de Java per a la seva
* inclusió en documentació generada automàticament
*/
```

ANOTACIONS

5.5. Diferències entre C++ i Java

Com hem comentat, el llenguatge Java es basa en C++ per proporcionar un entorn de programació orientat a objectes que resultarà molt familiar a un gran nombre de programadors. Tanmateix, Java intenta millorar C++ en molts aspectes i, sobretot, elimina els que permetien que C++ treballés de manera "no orientada a objectes" i que es van incorporar per compatibilitat amb el llenguatge C.

5.5.1. Entrada/sortida

Com que Java està pensat principalment per a treballar de manera gràfica, les classes que gestionen l'entrada/sortida en mode text s'han desenvolupat de manera molt bàsica. Estan regulades per la classe System, que es troba en la llibreria java.lang, i d'aquesta classe es destaquen tres objectes estàtics que són els següents:

- **`System.in`**. Rep les dades des de l'entrada estàndard (normalment el teclat) en un objecte de la classe InputStream (flux d'entrada).

- **`System.out`**. Imprimeix les dades en la sortida estàndard (normalment la pantalla) un objecte de la classe OutputStream (flux de sortida).

- **`System.err`**. Imprimeix els missatges d'error en pantalla.

Els mètodes bàsics de què disposen aquests objectes són els següents:

- **`System.in.read()`**. Llegeix un caràcter i el torna en forma d'enter.

- **`System.out.print(var)`**. Imprimeix una variable de qualsevol tipus primitiu.

- **`System.out.println(var)`**. Igual que l'anterior però afegeix un salt de línia final.

ANOTACIONS

Per tant, per a escriure un missatge n'hi ha prou d'utilitzar les instruccions `System.out.print()` i `System.out.println()`:

```
int unEnter = 35;
double unDouble = 3.1415;

System.out.println("Mostrant un text");
System.out.print("Mostrant un enter ");
System.out.println (unEnter);
System.out.print("Mostrant un double ");
System.out.println (unDouble);
```

Mentre que la sortida de dades és bastant natural, l'entrada de dades és molt menys accessible ja que l'element bàsic de lectura és el caràcter. A continuació es presenta un exemple en què es pot observar el procés necessari per a la lectura d'una cadena de caràcters:

```
String laMevaVar;
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// L'entrada acaba en prémer la tecla Entrar
laMevaVar = br.readLine();
```

Si es volen llegir línies completes, es pot fer per mitjà de l'objecte `BufferedReader`, el mètode del qual `readLine()` crida un lector de caràcters (un objecte `Reader`) fins a trobar un símbol de final de línia ("\n" o "\r"). Però en aquest cas, el flux d'entrada és un objecte `InputStream`, i no tipus `Reader`. Llavors, necessitem una classe que actuï com a lectora per a un flux de dades `InputStream`. Serà la classe `InputStreamReader`.

No obstant això, l'exemple anterior és vàlid per a *Strings*. Quan es vulgui llegir un nombre enter o altres tipus de dades, una vegada feta la lectura, s'ha de fer la conversió. Tanmateix, aquesta conversió pot arribar a generar un error fatal en el sistema si el text introduït no coincideix amb el tipus esperat. En aquest cas, Java ens obliga a considerar sempre aquest control d'errors. La gestió d'errors (que provoquen les anomenades *excepcions*) es fa, igual que en C++, per mitjà de la sentència `try {...} catch {...} finally {...}`.

ANOTACIONS

A continuació, veurem com es pot dissenyar una classe perquè torni un nombre enter llegit des del teclat:

Llegir.java

```
import java.io.*;
public class Llegir
{
   public static String getString()
   {
    String str = "";
    try
    {
     InputStreamReader isr = new
      InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(isr);
     str = br.readLine();
    }
    catch(IOException i)
    {
     System.err.println("Error: " + e.getMessage());
    }
    return str; // Tornar la dada escrita
   }

   public static int getInt()
    {
     try
     {
      return Integer.parseInt(getString());
     }
     catch(NumberFormatException i)
     {
      return Integer.MIN_VALUE; // valor mes petit
     }
   }
   // getInt
   // Es pot definir una funció per a cada tipus...
   public static double getDouble() {} // getDouble

}
 // Llegir
```

ANOTACIONS

En el bloc `try {...}` s'inclou el tros de codi susceptible de patir un error. En cas de produir-se, es llança una excepció que és recollida pel bloc `catch {...}`.

En el cas de la conversió de tipus `string` a nombres, l'excepció que es pot produir és del tipus `NumberFormatException`. Hi podria haver més blocs `catch` per a tractar diferents tipus d'excepció. En l'exemple, si es produeix un error el valor numèric tornat correspon al mínim valor possible que pot tenir un nombre enter.

El bloc `finally {...}` correspon a un tros de codi que s'ha d'executar tant si hi ha hagut error com si no (per exemple, tancar fitxers), encara que el seu ús és opcional.

De manera similar, es poden desenvolupar funcions per a cada un dels tipus primitius de Java. Finalment, la lectura d'un nombre enter seria com segueix:

```
int i;
...
i = Llegir.getInt( );
```

5.5.2. El preprocessador

Java no disposa de preprocessador, perquè s'eliminen diferents ordres (generalment, originàries de C). Les més conegudes són:

- `define`. Aquestes ordres serveixen per a la definició de constants. Ja en C++ havien perdut gran part del seu sentit en poder declarar variables `const`, i ara s'implementen a partir de les variables `final`.

- `include`. Aquesta ordre, que s'utilitzava per a incloure el contingut d'un fitxer, era molt útil en C++, principalment per a la reutilització dels fitxers de capçaleres. En Java, no hi ha fitxers de capçalera i les llibreries (o paquets) s'inclouen mitjançant la instrucció `import`.

ANOTACIONS

5.5.3. La declaració de variables i constants

La declaració de variables es manté igual, però la definició de constants canvia de forma: en Java, la variable és precedida per la paraula reservada `final`; no és necessari assignar-hi un valor en el moment de la declaració. No obstant això, en el moment en què s'hi assigni un valor per primera vegada, ja no es pot modificar.

```
final int i;
int j = 2;
...
i = j + 2; // assignat el valor, no es podra modificar
```

5.5.4. Els tipus de dades

Java classifica els tipus de dades en dues categories: primitives i referències. Mentre que el primer conté el valor, el segon només conté l'adreça de memòria on s'ha emmagatzemat la informació.

Els tipus primitius de dades de Java (`byte`, `short`, `int`, `long`, `float`, `double`, `char` i `boolean`) bàsicament coincideixen amb els de C++, encara que amb algunes modificacions, que presentem a continuació:

- Els tipus numèrics tenen la mateixa mida independentment de la plataforma en què s'executi.
- Per als tipus numèrics no existeix l'especificador `unsigned`.
- El tipus `char` utilitza el conjunt de caràcters Unicode, que té 16 bits. Els caràcters del 0 al 127 coincideixen amb els codis ASCII.
- Si no s'inicialitzen les variables explícitament, Java inicialitza les dades a zero (o al seu equivalent) automàticament i així elimina els valors escombraries que poguessin contenir.

Els tipus referència en Java són els vectors, classes i interfícies. Les variables d'aquests tipus desen la seva adreça de memòria, la qual cosa es podria assimilar als apuntadors en altres llenguatges. No

ANOTACIONS

obstant això, en no permetre les operacions explícites amb les adreces de memòria, per a accedir-hi n'hi haurà prou d'utilitzar el nom de la variable.

D'altra banda, Java elimina els tipus `struct` i `unio` que es poden implementar amb `class` i que es mantenien en C++ per compatibilitat amb C. També elimina el tipus `enum`, encara que es pot emular utilitzant constants numèriques amb la paraula clau `final`.

També s'eliminen definitivament els `typedefs` per a la definició de tipus, que en C++ ja havien perdut gran part del seu sentit en fer que les classes –Structs, Union i Enum– fossin tipus propis.

Finalment, només admet les coercions de tipus automàtiques (*type càsting*) en el cas de conversions segures; és a dir, on no hi hagi risc de perdre cap informació. Per exemple, admet les conversions automàtiques de tipus `int` a `float`, però no en sentit invers, ja que es perdrien els decimals. En cas de possible pèrdua d'informació, cal indicar-hi explícitament que es vol fer la conversió de tipus.

Una altra característica molt destacable de Java és la implementació que fa dels vectors. Els tracta com a objectes reals i genera una excepció (error) quan se superen els seus límits. També disposa d'un membre anomenat `length` per a indicar la seva longitud, la qual cosa proporciona un increment de seguretat del llenguatge perquè evita accessos indesitjats a la memòria.

Per a treballar amb cadenes de caràcters, Java disposa dels tipus `String` i `StringBuffer`. Les cadenes definides entre cometes dobles es converteixen automàticament en objectes `String`, i no es poden modificar. El tipus `StringBuffer` és similar, però permet la modificació del seu valor i proporciona mètodes per a la seva manipulació.

5.5.5. La gestió de variables dinàmiques

Tal com hem comentat en explicar C++, la gestió directa de la memòria és una arma molt potent però també molt perillosa: qualsevol error en la seva gestió pot portar problemes molt greus en l'aplicació i, potser, en el sistema.

ANOTACIONS

De fet, la presència dels apuntadors en C i C++ es devia a l'ús de cadenes i de vectors. Java proporciona objectes tant per a les cadenes com per als vectors, per la qual cosa, per a aquests casos, els apuntadors ja no són necessaris. L'altra gran necessitat, els passos de paràmetres per variable, queda coberta per l'ús de referències.

En Java el tema de la seguretat és primordial i per aquest motiu es va optar per no permetre l'ús d'apuntadors, almenys en el sentit en què s'entenien en C i C++.

En C++, es preveien dues maneres de treballar amb apuntadors:

- Amb la seva adreça, permetent-hi, fins i tot, operacions aritmètiques (apuntador).

- Amb el seu contingut (* apuntador).

En Java s'eliminen totes les operacions sobre les adreces de memòria. Quan es parla de *referències* aconsegueix un sentit diferent de C++. Una variable dinàmica correspon a la referència a l'objecte (apuntador):

- Per a veure el contingut de la variable dinàmica, n'hi ha prou d'utilitzar la forma (apuntador).

- Per a crear un nou element, es manté l'operador `new`.

- Si s'assigna una variable tipus referència (per exemple, un objecte) a una altra variable del mateix tipus (un altre objecte de la mateixa classe), el contingut no es duplica, sinó que la primera variable apunta a la mateixa posició de la segona variable. El resultat final és que el contingut de totes dues és el mateix.

Java no permet operar directament amb les adreces de memòria, cosa que simplifica l'accés al seu contingut: es fa per mitjà del nom de la variable (en lloc d'utilitzar la forma desreferenciada *nom_variable).

ANOTACIONS

Un altre dels principals riscos que comporta la gestió directa de la memòria és que alliberi correctament l'espai ocupat per les variables dinàmiques quan es deixen d'utilitzar. Java resol aquesta problemàtica proporcionant una eina que allibera automàticament aquest espai quan detecta que ja no es torna a utilitzar. Aquesta eina, coneguda com a *recol·lector d'escombraries* (*garbage collector*), forma part del Java durant l'execució dels seus programes. Per tant, no és necessària cap instrucció `delete`, sinó que n'hi ha prou d'assignar l'apuntador a `null`, i el recol·lector de memòria detecta que la zona de memòria ja no s'utilitza i l'allibera.

Si volem, en lloc d'esperar que es produeixi la recollida d'escombraries automàticament, podem invocar el procés per mitjà de la funció `gc()`. No obstant això, per a la JVM aquesta crida només es considera com un suggeriment.

5.5.6. Les funcions i el pas de paràmetres

Com ja sabem, Java només permet la programació orientada a objectes. Per tant, no s'admeten les funcions independents (sempre s'han d'incloure en classes) ni les funcions globals. A més, la implementació dels mètodes s'ha de fer dins de la definició de la classe. D'aquesta manera, també s'elimina la necessitat dels fitxers de capçaleres. El mateix compilador detecta si una classe ja ha estat carregada per a evitar la seva duplicació. Malgrat la seva similitud amb les funcions `inline`, només és formal perquè internament tenen comportaments diferents: en Java no s'implementen les funcions `inline`.

D'altra banda, Java continua suportant la sobrecàrrega de funcions, encara que no permet al programador la sobrecàrrega d'operadors, tot i que el compilador utilitza aquesta característica internament.

En Java tots els paràmetres es passen per valor.

En el cas dels tipus de dades primitives, els mètodes sempre reben una còpia del valor original, que no es pot modificar.

ANOTACIONS

En el cas de tipus de dades de referència, també es copia el valor d'aquesta referència. No obstant això, per la naturalesa de les referències, els canvis fets en la variable rebuda per paràmetre també afecten la variable original.

Per a modificar les variables que passa per paràmetre a la funció, les hem d'incloure com a variables membres de la classe i passar com a argument la referència a un objecte d'aquesta classe.

5.6. Les classes en Java

Com ja hem comentat, un dels objectius que van motivar la creació de Java va ser disposar d'un llenguatge orientat a objectes "pur", en el sentit que sempre s'hauria de complir aquest paradigma de programació. Això, per la seva compatibilitat amb C, no passava en C++. Per tant, les classes són el component fonamental de Java: tot hi és inclòs. La manera de definir les classes en Java és similar a l'utilitzada en C++, encara que es presenten algunes diferències:

Punt2D.java

```
class Punt2D
{
   int x, y;

// Inicialitzant l'origen de coordenades
   Punt2D()
   {
    x = 0;
    y = 0;
   }

// Inicialitzant una coordenada x,y determinada
   Punt2D(int coordx, int coordy)
   {
    x = coordx;
    y = coordy;
   }
```

ANOTACIONS

```
    // Calcula la distància a un altre punt
    float distancia(Punt2D npunt)
    {
      int dx = x npunt.x;
      int dy = y npunt.y;
      return ( Math.sqrt(dx * dx + dy * dy));
    }
}
```

- La primera diferència és la inclusió de la definició dels mètodes en l'interior de la classe i no separada com en C++. En seguir aquest criteri, ja no és necessari l'operador d'àmbit (::).
- La segona diferència és que en Java no cal el punt i coma (;) final.
- Les classes es desen en un fitxer amb el mateix nom i amb l'extensió ".java" (Punt2.java).

Una característica comuna a C i C++ és que Java també és sensible a les majúscules, per la qual cosa la classe `Punt2D` és diferent de `punt2d` o `pUnT2d`.

Java permet desar més d'una classe en un fitxer però només permet que una sigui pública. Aquesta classe serà la que donarà el nom a l'arxiu. Per tant, llevat de rares excepcions, se sol utilitzar un arxiu independent per a cada classe.

En la definició de la classe, de manera similar a C++, es declaren els atributs (o variables membres) i els mètodes (o funcions membres) tal com es pot observar en l'exemple anterior.

5.6.1. Declaració d'objectes

Una vegada definida una classe, per a declarar un objecte d'aquesta classe n'hi ha prou d'anteposar el nom de la classe (com un tipus més) al de l'objecte.

```
Punt2D puntU;
```

El resultat és que `puntU` és una referència a un objecte de la classe `Punt2D`. Inicialment, aquesta referència té valor `null` i no ha fet cap

ANOTACIONS

reserva de memòria. Si el que es vol és poder utilitzar aquesta variable per a desar informació, és necessari crear una instància mitjançant l'operador `new`. En utilitzar-lo, es crida el constructor de l'objecte `Punt2D` definit.

```
puntU = new Punt2D(2,2); // inicialitzant (2,2)
```

Una diferència important en Java respecte a C++ és l'ús de referències per a manipular els objectes. Com s'ha comentat anteriorment, l'assignació de dues variables declarades com a objectes només implica l'assignació de la seva referència:

```
Punt2D    puntDos;
puntDos = puntU;
```

Si s'afegeix la instrucció anterior, no s'ha fet cap reserva específica de memòria per a la referència a objecte `puntDos`. Per tant, en fer l'assignació, `puntDos` farà referència al mateix objecte apuntat per `puntU`, i no a una còpia. Per tant, qualsevol canvi sobre els atributs de `puntU` es veuran reflectits en `puntDos`.

5.6.2. Accés als objectes

Una vegada creat un objecte, s'accedeix a qualsevol dels seus atributs i mètodes per mitjà de l'operador punt (.) tal com fèiem en C++.

```
int i;
float dist;

i = puntU.x;
dist = puntU.distancia(5,1);
```

En C++ es podia accedir a l'objecte per mitjà de la desreferència d'un apuntador a aquest objecte (`*apuntador`); en aquest cas, l'accés als seus atributs o mètodes es podia fer per mitjà de l'operador punt (`*apuntador.atribut`) o per la seva forma d'accés abreujada mitjançant l'operador –> (`apuntador->atribut`). En Java, en no existir la forma desreferenciada *apuntador, tampoc no existeix l'operador –>.

ANOTACIONS

Finalment Java, igual que C++, permet l'accés a l'objecte dins dels mètodes de la classe amb l'objecte `this`.

5.6.3. Destrucció d'objectes

Cada vegada que es crea un objecte, quan es deixa d'utilitzar ha de ser destruït. La manera d'operar que té la gestió de memòria en Java permet evitar molts dels conflictes que apareixen en altres llenguatges i és possible delegar aquesta responsabilitat a un procés automàtic: el recol·lector d'escombraries, que detecta quan una zona de memòria no està referenciada; quan el sistema disposa d'un moment de menys intensitat de processador, l'allibera.

Algunes vegades, en treballar amb una classe s'utilitzen altres recursos addicionals, com els fitxers. Sovint, en acabar l'activitat de la classe, també s'ha de poder tancar l'activitat d'aquests recursos addicionals. En aquests casos, cal fer un procés manual semblant als destructors en C++. Per a això, Java permet la implementació d'un mètode anomenat `finalize()` que, en cas d'existir, és cridat pel mateix recol·lector. En l'interior d'aquest mètode, s'escriu el codi que allibera explícitament els recursos addicionals utilitzats. El mètode `finalize` sempre és del tipus `static void`.

```
class LaMevaClasse
{
   LaMevaClasse() //Mètode constructor
   {
     ...          //Instruccions d'inicialització
   }

   static void finalize() //Mètode destructor
   {
     ... //Instruccions d'alliberament de recursos
   }
}
```

5.6.4. Constructors de còpia

C++ disposa dels constructors de còpia per a assegurar que es fa una còpia completa de les dades en el moment de fer una as-

signació, o d'assignar un paràmetre o un valor de retorn d'una funció.

Tal com s'ha comprovat, Java té una filosofia diferent. Les assignacions entre objectes no impliquen una còpia del seu contingut, sinó que la segona referència passa a referenciar el primer objecte. Per tant, sempre s'accedeix al mateix contingut i no és necessària cap operació de reserva de memòria addicional. Com a conseqüència d'aquest canvi de filosofia, Java no necessita constructors de còpia.

5.6.5. Herència simple i herència múltiple

En Java, per a indicar que una classe deriva d'una altra (és a dir, hereta totalment o parcialment els seus atributs i mètodes) es fa per mitjà del terme `extends`. Reprendrem l'exemple dels gossos i els mamífers.

```
class Mamifer
{
   int edat;

   Mamifer()
   { edat = 0; }

   void assignarEdat(int nEdat)
   { edat = nEdat; }

   int obtenirEdat()
   { return (edat); }

   void emetreSo()
   { System.out.println("So "); }
}

class Gos extends Mamifer
{
   void emetreSo()
   { System.out.println("Bup "); }
}
```

ANOTACIONS

En l'exemple anterior, hem dit que la classe Gos és una classe derivada de la classe Mamífer. També és possible llegir la relació en el sentit contrari indicant que la classe Mamífer és una superclasse de la classe Gos.

En C++ era possible l'herència múltiple, és a dir, rebre els atributs i mètodes de diverses classes. Java no admet aquesta possibilitat, encara que d'alguna manera permet una funcionalitat semblant per mitjà de les interfícies.

5.7. Herència i polimorfisme

L'herència i el polimorfisme són propietats essencials dins del paradigma del disseny orientat a objectes. Aquests conceptes ja han estat comentats en la unitat dedicada a C++ i continuen essent vigents en Java. No obstant això, hi ha variacions en la seva implementació que comentem a continuació.

5.7.1. Les referències this i super

Algunes vegades, és necessari accedir als atributs o mètodes de l'objecte que serveix de base a l'objecte en què s'està. Tal com s'ha vist, tant Java com C++ proporcionen aquest accés per mitjà de la referència `this`.

La novetat que presenta Java és poder accedir també als atributs o mètodes de l'objecte de la superclasse mitjançant la referència `super`.

5.7.2. La classe Object

Una altra de les diferències de Java respecte a C++ és que tots els objectes pertanyen al mateix arbre de jerarquies, l'arrel del qual és la classe Object i de la qual hereten totes les altres: si una classe, en la seva definició, no té el terme `Extends`, es considera que hereta directament d'Object.

Podem dir que la classe Object és la superclasse de la qual deriven directament o indirectament totes les altres classes en Java.

La classe Object proporciona una sèrie de mètodes comuns, entre els quals hi ha:

- `public boolean equals(Object)`. S'utilitza per a comparar el contingut de dos objectes i torna *true* si l'objecte rebut coincideix amb l'objecte que el crida. Si només es volen comparar dues referències a objecte, es poden utilitzar els operadors de comparació == i `!=`.
- `protected Object Clone()`. Retorna una còpia de l'objecte.

5.7.3. Polimorfisme

C++ implementava la capacitat d'una variable de poder prendre diverses formes mitjançant apuntadors a objectes. Com s'ha comentat, Java no disposa d'apuntadors i cobreix aquesta funció amb referències, però el funcionament és similar.

```
Mamifer MamiferU = new Gos;
Mamifer MamiferDos = new Mamifer;
```

Recordem que, en Java, la declaració d'un objecte sempre correspon a una referència a aquest.

5.7.4. Classes i mètodes abstractes

En C++ es va comentar que, en alguns casos, les classes corresponen a elements teòrics dels quals no té cap sentit instanciar objectes, sinó que sempre s'havia de crear objectes de les seves classes derivades.

ANOTACIONS

La implementació en C++ es fa per mitjà de les funcions virtuals pures i la manera de representar-la és, com a mínim, una mica peculiar: es declaraven assignant la funció virtual a 0.

La implementació de Java per a aquests casos és molt més senzilla: anteposar la paraula reservada `abstract` al nom de la funció. En declarar una funció com a `abstract`, ja s'indica que la classe també ho és. No obstant això, és recomanable explicitar-ho en la declaració anteposant la paraula `abstract` a la paraula reservada `class`.

El fet de definir una funció com a abstract obliga al fet que les classes derivades que puguin rebre aquest mètode la redefineixin. Si no ho fan, hereten la funció com a abstracta i, com a conseqüència, elles també ho seran, la qual cosa impedirà instanciar objectes d'aquestes classes.

```
abstract class ObraDArt
{
   String autor;

   ObraDArt(){} //Mètode constructor

   abstract void mostrarObraDArt(); //abstract
   void assignarAutor(String nAutor)
     { autor = nAutor; }
   String obtenirAutor();
   { return (autor); }
};
```

En l'exemple anterior, s'ha declarat com a abstracta la funció `mostrarObraDArt()`, cosa que obliga a redefinir-la en les classes derivades. Per tant, no inclou cap definició. Cal tenir present que, en ser una classe abstracta, no es podrà fer un `new ObraDArt`.

5.7.5. Classes i mètodes finals

En la definició de variables, ja s'ha tractat el concepte de variables finals. Hem dit que les variables finals, una vegada inicialitzades,

ANOTACIONS

no es poden modificar. El mateix concepte es pot aplicar a classes i mètodes:

- Les classes finals no tenen ni poden tenir classes derivades.
- Els mètodes finals no es poden redefinir en les classes derivades.

L'ús de la paraula reservada `final` es converteix en una mesura de seguretat per a evitar usos incorrectes o maliciosos de les propietats de l'herència que pogues-sin suplantar funcions establertes.

5.7.6. Interfícies

Una interfície és una col·lecció de definicions de mètodes (sense les seves implementacions), la funció de la qual és definir un protocol de comportament que pot ser implementat per qualsevol classe independentment del seu lloc en la jerarquia de classes.

En indicar que una classe implementa una interfície, se l'obliga a redefinir tots els mètodes definits. En aquest aspecte, les interfícies s'assemblen a les classes abstractes. No obstant això, mentre que una classe només pot heretar d'una superclasse (només permet herència simple), pot implementar diverses interfícies. Això només indica que compleix cada un dels protocols definits en cada interfície.

A continuació presentem un exemple de declaració d'interfície:

```
public interface NomInterficie
Extends SuperInterficie1, SuperInterficie2
   { cos interficie }
```

Si una interfície no s'especifica com a pública, només serà accessible per a les classes definides en el seu mateix paquet.

ANOTACIONS

El cos de la interfície conté les declaracions de tots els mètodes que s'hi inclouen. Cada declaració s'acaba en punt i coma (;) ja que no tenen implementacions i implícitament es consideren `public` i `abstract`.

El cos també pot incloure constants en el cas que es considerin `public`, `static` i `final`.

Per a indicar que una classe implementa una `interface`, n'hi ha prou d'afegir la paraula clau `implements` en la seva declaració. Java permet l'herència múltiple d'interfícies:

```
class LaMevaClasse extends SuperClasse
implements Interficie1, interficie2
{ ... }
```

Quan una classe declara una interfície, és com si signés un contracte pel qual es compromet a implementar els mètodes de la interfície i de les seves superinterfícies. L'única manera de no fer-ho és declarar la classe com a `abstract`, amb la qual cosa no es podrà instanciar objectes i es transmetrà aquesta obligació a les seves classes derivades.

De fet, a primera vista sembla que hi ha moltes semblances entre les classes abstractes i les interfícies però les diferències són significatives:

- Una interfície no pot implementar mètodes, mentre que les classes abstractes sí que ho fan.
- Una classe pot tenir diverses interfícies, però només una superclasse.
- Les interfícies no formen part de la jerarquia de classes i, per tant, classes no relacionades poden implementar la mateixa interfície.

Una altra característica rellevant de les interfícies és que en definir-les s'està declarant un nou tipus de dades referència. Una variable d'aquest tipus de dades podrà ser instanciada per qualsevol classe que implementi aquesta interfície. Això proporciona una altra manera d'aplicar el polimorfisme.

ANOTACIONS

5.7.7. Paquets

Per a organitzar les classes, Java proporciona els paquets. Un paquet (*package*) és una col·lecció de classes i interfícies relacionades que proporcionen protecció d'accés i gestió de l'espai de noms. Les classes i interfícies sempre pertanyen a un paquet.

Nota

De fet, les classes i interfícies que formen part de la plataforma de Java pertanyen a diversos paquets organitzats per la seva funció: `java.lang` inclou les classes fonamentals, `java.io` les classes per a entrada/sortida, etc.

El fet d'organitzar les classes en paquets evita en gran manera que hi pugui haver una col·lisió en l'elecció del nom.

Per a definir una classe o una interfície en un paquet, n'hi ha prou d'incloure en la primera línia de l'arxiu l'expressió següent:

```
package elMeuPaquet;
```

Si no es defineix cap paquet, s'inclou dins del paquet per defecte (`default package`), la qual cosa és una bona solució per a petites aplicacions o quan es comença a treballar en Java.

Per a accedir al nom de la classe, es pot fer per mitjà del nom llarg:

```
elMeuPaquet.LaMevaClasse
```

Una altra possibilitat és la importació de les classes públiques del paquet mitjançant la paraula clau `import`. Després, és possible utilitzar el nom de la classe o de la interfície en el programa sense el prefix:

```
import elMeuPaquet.LaMevaClasse;//Importa només la classe
import elMeuPaquet.*            // Importa tot el paquet
```

ANOTACIONS

Exemple

La importació de `java.awt` no inclou les classes del subpaquet `java.awt.event`.

Cal tenir en compte que importar un paquet no implica importar els diferents subpaquets que pugui contenir.

Per convenció, Java sempre importa per defecte del paquet `java.lang`.

Per a organitzar totes les classes i paquets possibles, es crea un subdirectori per a cada paquet on s'inclouen les seves diferents classes. Al seu torn, cada paquet pot tenir els seus subpaquets, que es trobaran en un subdirectori. Amb aquesta organització de directoris i arxius, tant el compilador com l'intèrpret tenen un mecanisme automàtic per a localitzar les classes que necessiten altres aplicacions.

Exemple

La classe `grafics.figures.rectangle` es trobaria dins del paquet `grafics.figures` i l'arxiu estaria localitzat a grafics\figures\rectangle.java.

5.7.8. L'API (*applications programming interface*) de Java

La multitud de biblioteques de funcions que proporciona el mateix llenguatge és una de les bases primordials de Java; biblioteques, que estan ben documentades, són estàndard i funcionen per a les diferents plataformes.

Aquest conjunt de biblioteques està organitzat en paquets i inclòs en l'API de Java. Les principals classes són les següents:

Taula 9.

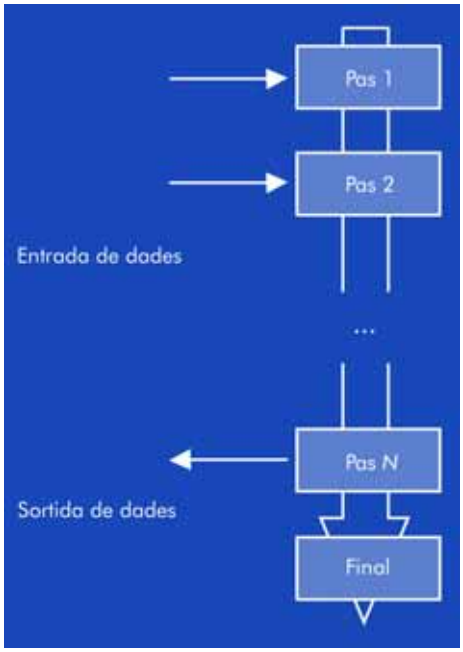
Paquet	Classes incorporades
java.lang.	Classes fonamentals per al llenguatge com la classe String i d'altres.
java.io	Classes per a l'entrada i sortida per mitjà de fluxos de dades, i fitxers del sistema.
java.util	Classes d'utilitat com col·leccions de dades i classes, el model d'incidències, facilitats horàries, generació aleatòria de nombres, i d'altres.
java.math	Classe que agrupa totes les funcions matemàtiques.

Paquet	Classes incorporades
java.applet	Classes amb utilitats per a crear miniaplicacions i classes que utilitzen les miniaplicacions per a comunicar-se amb el seu context.
java.awt	Classes que permeten la creació d'interfícies gràfiques amb l'usuari, i dibuixar imatges i gràfics.
javax.swing	Classes amb components gràfics que funcionen igual en totes les plataformes Java.
java.security	Classes responsables de la seguretat en Java (xifratge, etc.).
java.net	Classes amb funcions per a aplicacions en xarxa.
java.sql	Classe que incorpora el JDBC per a la connexió de Java amb bases de dades.

5.8. El paradigma de la programació orientada a incidències

Els diversos paradigmes de programació que s'han revisat fins al moment es caracteritzen per tenir un flux d'instruccions seqüencial i considerar les dades com el complement necessari per al desenvolupament de l'aplicació. El seu funcionament implica normalment un inici, una seqüència d'accions i un final de programa:

Figura 15.



Flux de programa en programació imperativa

Dins d'aquest funcionament seqüencial, el procés rep incidències externes que es poden esperar (entrades de dades de l'usuari per te-

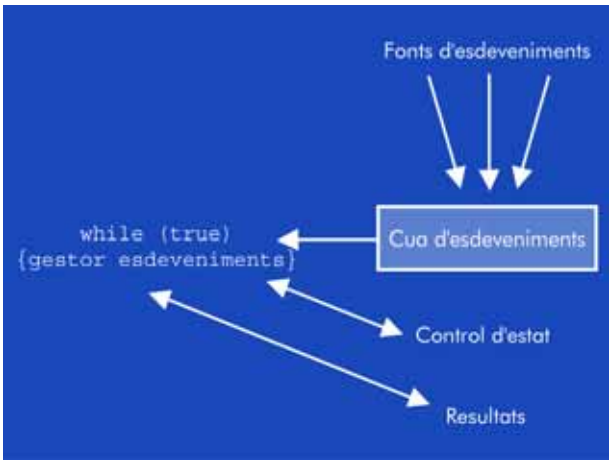
ANOTACIONS

clat, ratolí o altres formes, lectures d'informació del sistema, etc.) o inesperats (errors de sistema, etc.). A cada un d'aquestes esdeveniments externs l'anomenarem **incidència** (en anglès, *event*).

En els paradigmes anteriors, les incidències no alteren l'ordre del flux d'instruccions previst: se les atén per a resoldre-les o, si no és possible, es produeix un acabament del programa.

En el paradigma de programació dirigida per incidències no es fixa una seqüència única d'accions, sinó que prepara reaccions a les incidències que puguin anar succeint una vegada iniciada l'execució del programa. Per tant, en aquest model les dades introduïdes són les que regulen la seqüència de control de l'aplicació. També es pot observar que les aplicacions difereixen en el seu disseny respecte dels paradigmes anteriors: estan preparades per a romandre en funcionament un temps indefinit, rebent i gestionant incidències.

Figura 16.



Flux de programa en programació dirigida per incidències

5.8.1. Les incidències en Java

Per a la gestió de les incidències, Java proposa utilitzar **el model de delegació d'incidències**. En aquest model, un component rep una incidència i la transmet al gestor d'incidències que té assignat perquè la gestioni (*event listener*). Per tant, tindrem una separació del codi entre la generació de la incidència i la seva manipulació que ens facilitarà la seva programació.

ANOTACIONS

Diferenciarem els quatre tipus d'elements que hi intervenen:

- La incidència (què es rep). En la gran majoria dels casos, és el sistema operatiu qui proporciona la incidència i gestiona finalment totes les operacions de comunicacions amb l'usuari i l'entorn. S'emmagatzema en un objecte derivat de la classe Event i que depèn del tipus d'incidència que s'ha produït. Les principals tenen relació amb l'entorn gràfic i són les següents: `ActionEvent`, `KeyEvent`, `MouseEvent`, `AdjustmentEvent`, `WindowEvent`, `TextEvent`, `ItemEvent`, `FocusEvent`, `ComponentEvent`, `ContainerEvent`.

  Cada una d'aquestes classes té els seus atributs i els seus mètodes d'accés.

- La font de la incidència (on es rep). Correspon a l'element on s'ha generat la incidència i, per tant, recull la informació per tractar-la o, en el nostre cas, per traspassar-la al seu gestor d'incidències. En entorns gràfics, sol correspondre a l'element amb què l'usuari ha interactuat (un botó, un quadre de text, etc.).

- El gestor d'incidències (qui el gestiona). És la classe especialitzada que indica, per a cada incidència, quina és la resposta que es vol. Cada gestor pot actuar davant de diferents tipus d'incidències només assignant-hi els perfils adequats.

- El perfil del gestor (quines operacions ha d'implementar el gestor). Per a facilitar aquesta tasca hi ha interfícies que indiquen els mètodes que s'han d'implementar per a cada tipus d'incidència. Normalment, el nom d'aquesta interfície és de la forma <nomIncidencia>Listener (literalment, 'qui escolta la incidència').

Exemple

`KeyListener` és la interfície per a les incidències de teclat i considera els tres mètodes següents: `keyPressed`, `keyReleased` i `keyTyped`. Algunes vegades, l'obligació d'implementar tots els mètodes implica una càrrega inútil. Per a aquestes situacions, Java proporciona adaptadors <nomIncidència>Adapter que implementen els diferents mètodes buits i així permeten redefinir només els mètodes que ens interessen.

ANOTACIONS

Els principals perfils (o interfícies) definits per Java són els següents: `ActionListener`, `KeyListener`, `MouseListener`, `TextListener`, `ItemListener`, `WindowListener`, `AdjustmentListener`, `FocusListener`, `ComponentListener` i `ContainerListener`. Tots ells derivats de la interfície `EventListener`.

Finalment, n'hi ha prou d'establir la relació entre la font de la incidència i el seu gestor. Per a això, en la classe font afegirem un mètode del tipus `add<nomIncidencia>Listener`.

Exemple

Si a un objecte botó de la classe Button hi volem afegir un *Listener* de les incidències de ratolí farem:

```
boton.
addMouseListener
(gestorIncidencies).
```

De fet, es podria considerar que les incidències no són realment enviades al gestor d'incidències, sinó que és el mateix gestor qui s'assigna a la incidència.

Nota

Comprendrem més fàcilment el funcionament de les incidències per mitjà d'un exemple pràctic, com el que mostra la creació d'una miniaplicació mitjançant la llibreria gràfica Swing que es veurà més endavant en aquest capítol.

5.9. Fils d'execució

Els sistemes operatius actuals permeten la multitasca, almenys en aparença, ja que si l'ordinador disposa d'un únic processador, només podrà fer una activitat alhora. No obstant això, es pot organitzar el funcionament d'aquest processador perquè reparteixi el seu temps entre diverses activitats o perquè aprofiti el temps que li deixa lliure una activitat per a continuar l'execució d'una altra.

A cada una d'aquestes activitats se l'anomena *procés*. Un procés és un programa que s'executa de manera independent i amb un espai propi de memòria. Per tant, els sistemes operatius multitasca permeten l'execució de diversos processos alhora.

Cada un d'aquests processos pot tenir un o diversos fils d'execució, cadascun dels quals correspon a un flux seqüencial d'instruccions. En aquest cas, tots els fils d'execució comparteixen el mateix espai de

ANOTACIONS

memòria i s'utilitza el mateix context i els mateixos recursos assignats al procés.

Java incorpora la possibilitat que un procés tingui múltiples fils d'execució simultanis. El coneixement complet de la seva implementació en Java supera els objectius del curs i, a continuació, ens limitarem a conèixer les bases per a la creació dels fils i el seu cicle de vida.

5.9.1. Creació de fils d'execució

En Java, hi ha dues maneres de crear fils d'execució:

- Crear una nova classe que hereti de `java.lang.Thread` i sobrecarregar el mètode `run()` d'aquesta classe.

- Crear una nova classe amb la interfície `java.lang.Runnable` on s'implementarà el mètode `run()`, i després crear un objecte de tipus *Thread* al qual es passa com a argument un objecte de la nova classe.

Sempre que sigui possible s'utilitzarà la primera forma, per la seva simplicitat. No obstant això, si la classe ja hereta d'alguna una altra superclasse, no serà possible derivar també de la classe *Thread* (Java no permet l'herència múltiple), amb la qual cosa s'haurà d'escollir la segona manera.

Vegem un exemple de cadascuna de les maneres de crear fils d'execució:

Creació de fils d'execució derivant de la classe *Thread*

ProvarThread.java

```
class ProvarThread
{
   public static void main(String args[] )
   {
   AThread a = new AThread();
```

ANOTACIONS

```
   BThread b = new BThread();
   a.start();
   b.start();
   }
}

class AThread extends Thread
{
   public void run()
   {
   int i;
   for (i=1;i<=10; i++)
    System.out.print(" A"+i);
   }
}

class BThread extends Thread
{
   public void run()
   {
    int i;
    for (i=1;i<=10; i++)
     System.out.print(" B"+i);
   }
}
```

En l'exemple anterior, es creen dues noves classes que deriven de la classe *Thread*: les classes `AThread` i `BThread`. Cadascuna mostra en pantalla un comptador precedit per la inicial del procés.

En la classe `ProvarThreads`, on tenim el mètode `main()`, es procedeix a la instanciació d'un objecte per a cada una de les classes *Thread* i s'hi inicia l'execució. El resultat final serà del tipus següent, encara que no per força en aquest ordre:

A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A10 B10

Noteu que en l'execució `ProvarThreads` s'executen tres fils: el principal i els dos creats.

ANOTACIONS

Creació de fils d'execució implementant la interfície `Runnable`

Provar2Thread.java

```
class Provar2Thread
{
   public static void main(String args[])
   {
    AThread a = new AThread();
    BThread b = new BThread();
    a.start();
    b.start();
   }
}

class AThread implements Runnable
{
   Thread t;
   public void start()
   {
    t = new Thread(this);
    t.start();
   }

   public void run()
   {
    int i;
    for (i=1;i<=50; i++)
      System.out.print(" A"+i);
   }
}

class BThread implements Runnable
{
   Thread t;

   public void start()
   {
    t = new Thread(this);
    t.start();
   }
```

ANOTACIONS

```
    public void run()
    {
      int i;
      for (i=1;i<=50; i++)
      System.out.print(" B"+i);
    }
}
```

En aquest exemple, es pot observar que la classe principal main () no ha canviat, però sí que ho ha fet la implementació de cada una de les *classes AThread* i *BThread*. En cadascuna, a més d'implementar la interfície `Runnable`, s'ha de definir un objecte de la classe `Thread` i redefinir el mètode `start()` perquè cridi l'`start()` de l'objecte de la classe `Thread` passant-hi l'objecte actual `this`.

Per acabar, dues coses: es pot passar un nom a cada fil d'execució per a identificar-lo, ja que la classe *Thread* té el constructor sobrecarregat per a admetre aquesta opció:

```
public Thread (String nom);
public Thread (Runnable destinacio, String nom);
```

Sempre es pot recuperar el nom per mitjà del mètode:

```
public final String getName();
```

5.9.2. Cicle de vida dels fils d'execució

El cicle de vida dels fils d'execució es pot representar a partir dels estats pels quals poden passar:

- Nou (*new*): el fil s'acaba de crear però encara no s'ha inicialitzat, és a dir, encara no s'ha executat el mètode `start()`.

- Executable (*runnable*): el fil s'està executant o està preparat per a executar-se.

- Bloquejat (*blocked* o *not runnable*): el fil està bloquejat per un missatge intern `sleep()`, `suspend()` o `wait()` o per alguna ac-

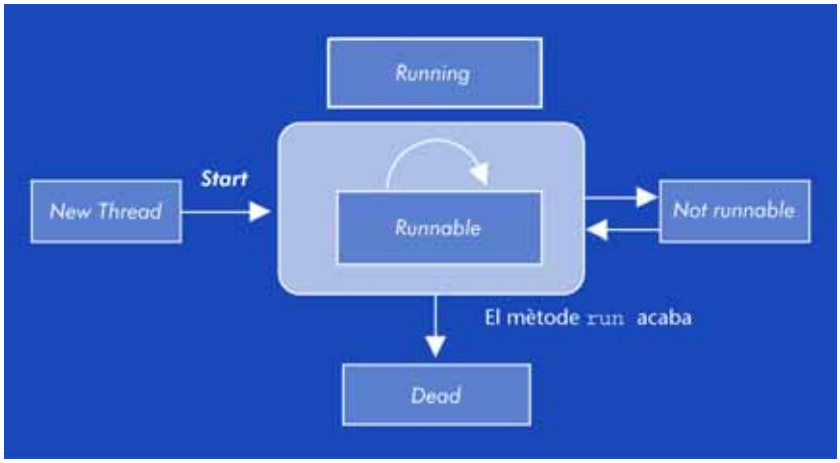
ANOTACIONS

tivitat interna, per exemple, en espera d'una entrada de dades. Si està en aquest estat, no entra dins de la llista de tasques que ha d'executar el processador.

Per a tornar a l'estat d'Executable, ha de rebre un missatge intern `resume()` o `notify()` o acabar la situació que provocava el bloqueig.

- Mort (*dead*): el mètode habitual d'acabar una cadena és que hagi acabat d'executar les instruccions del mètode `run()`. També es podria utilitzar el mètode `stop()`, però és una opció considerada "perillosa" i no recomanada.

Figura 17.



5.10. Les miniaplicacions

Una miniaplicació de Java està preparada per a ser executada en un navegador d'Internet. Per a incloure una miniaplicació en una pàgina HTML, n'hi ha prou d'incloure la seva informació per mitjà de les etiquetes <APPLET> ... </APPLET>.

La majoria de navegadors d'Internet funcionen en un entorn gràfic. Per tant, les miniaplicacions s'hi han d'adaptar per mitjà de biblioteques gràfiques. En aquest apartat, s'utilitzarà la biblioteca `java.awt`, que és la biblioteca proporcionada originalment des de les seves primeres versions. Una discussió més profunda entre les diferents biblioteques disponibles es veurà més endavant en aquest capítol.

ANOTACIONS

Les característiques principals de les miniaplicacions són les següents:

- Els fitxers ".class" es baixen per la xarxa des d'un servidor HTTP fins al navegador, on la JVM els executa.
- Atès que s'utilitzen per mitjà d'Internet, s'ha establert que tinguin unes restriccions de seguretat molt fortes, com per exemple, que només puguin llegir i escriure fitxers des del seu servidor (i no des de l'ordinador local), que només puguin accedir a informació limitada a l'ordinador on s'executen, etc.
- Les miniaplicacions no tenen finestra pròpia, sinó que s'executen en una finestra del navegador.

  Des del punt de vista del programador, destaquen els aspectes següents:

- No necessiten mètode `main`. La seva execució s'inicia per altres mecanismes.
- Deriven sempre de la classe java.applet.Applet i, per tant, han de redefinir alguns dels seus mètodes com `init()`, `start()`, `stop()` i `destroy()`.
- També solen redefinir altres mètodes com `paint()`, `update()` i `repaint()`, heretats de classes superiors per a tasques gràfiques.
- Disposen d'una sèrie de mètodes per a obtenir informació sobre la miniaplicació o sobre altres miniaplicacions en execució en la mateixa pàgina com `getAppletInfo()`, `getAppletContext()`, `getParameter()`, etc.

5.10.1. Cicle de vida de les miniaplicacions

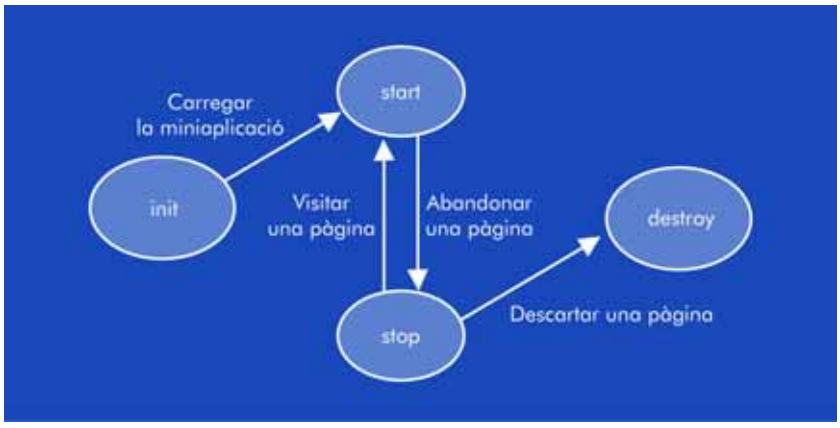
Per la seva naturalesa, el cicle de vida d'una miniaplicació és una mica més complex que el d'una aplicació normal. Cada una de les fases del cicle de vida està marcada amb una crida a un mètode de la miniaplicació:

- `void init()`. Es crida quan es carrega la miniaplicació, i conté les inicialitzacions que necessita.

ANOTACIONS

- `void start()`. Es crida quan la pàgina s'ha carregat, aturat (per minimització de la finestra, canvi de pàgina web, etc.) i s'ha tornat a activar.
- `void stop()`. Es crida de manera automàtica en ocultar la miniaplicació. En aquest mètode, se solen aturar els fils que s'estan executant per a no consumir recursos innecessaris.
- `void destroy()`. Es crida aquest mètode per a alliberar els recursos (menys la memòria) de la miniaplicació.

Figura 18.



En ser aplicacions gràfiques que apareixen en una finestra del navegador, també és útil redefinir el mètode següent:

- `voidpaint(Graphics g)`. En aquesta funció s'han d'incloure totes les operacions amb gràfics, perquè aquest mètode es crida quan la miniaplicació es dibuixa per primera vegada i quan es redibuixa.

5.10.2. Manera d'incloure miniaplicacions en una pàgina HTML

Com ja hem comentat, per a cridar una miniaplicació des d'una pàgina HTML utilitzem les etiquetes `<APPLET> ... <\APPLET>`, entre les quals, com a mínim, incloem la informació següent:

- `CODE` = nom de la miniaplicació (per exemple, elmeuApplet.class)
- `WIDTH` = amplada de la finestra
- `HEIGHT` = altura de la finestra

ANOTACIONS

I, opcionalment els atributs següents:

- `NAME` = "unnom", que li permet comunicar-se amb altres miniaplicacions
- `ARCHIVE` = "unarxiu", on es desen les classes en un ".zip" o un ".jar"
- `PARAM NAME` = "param1" VALUE = "valor1" per poder passar paràmetres a la miniaplicació

5.10.3. La nostra primera miniaplicació en Java

La millor manera de comprendre el funcionament de les miniaplicacions és per mitjà d'un exemple pràctic. Per a crear la nostra primera miniaplicació seguirem aquests passos:

1) Crear un fitxer font. Mitjançant l'editor escollit, escriurem el text i el desarem amb el nom `HolaMonApplet.java`.

HolaMonApplet.java

```
import java.applet.*;
import java.awt.*;
/**
 * La classe HolaMonApplet mostra el missatge
 * "Hola Món" en la sortida estàndard.
 */
public class HolaMonApplet extends Applet{
   public void paint(Graphics g)
   {
     // Mostra "Hola Món! "
     g.drawString("Hola Món! " 75, 30 );
   }
}
```

2) Crear un fitxer HTML. Mitjançant l'editor escollit, escriurem el text.

HolaMonApplet.html

```
<HTML>
<HEAD>
```

```
<TITLE>La meva primera miniaplicacio</TITLE >
</HEAD>
<BODY>
Us vull donar un missatge:
<APPLET CODE="HolaMonApplet.class" WIDTH=150 HEIGHT=25> </APPLET>
</BODY>
</HTML>
```

3) Compilar el programa i generar un fitxer *bytecode*.

```
javac HolaMonApplet.java
```

4) Visualitzar la pàgina HolaMonApplet.html des d'un navegador.

5.11. Programació d'interfícies gràfiques en Java

L'aparició de les interfícies gràfiques va implicar una gran evolució en el desenvolupament de sistemes i aplicacions. Fins a la seva aparició, els programes es basaven en el mode text (o consola) i, generalment, el flux d'informació d'aquests programes era seqüencial i es dirigia per mitjà de les diferents opcions que s'anaven introduint a mesura que l'aplicació ho sol·licitava.

Les interfícies gràfiques permeten una comunicació molt més àgil amb l'usuari i faciliten la seva interacció amb el sistema en múltiples punts de la pantalla. Es pot triar en un moment determinat entre múltiples operacions disponibles de naturalesa molt variada (per exemple, introducció de dades, selecció d'opcions de menú, canvis de formularis actius, canvis d'aplicació, etc.) i, per tant, entre múltiples fluxos d'instruccions, cadascun dels quals és la resposta a incidències diferenciades.

Els programes que utilitzen aquestes interfícies són un clar exemple del paradigma de programació dirigit per incidències.

ANOTACIONS

Amb el temps, les interfícies gràfiques han anat evolucionant i han anat sorgint components nous (botons, llistes desplegables, botons d'opcions, etc.) que s'adapten millor a la comunicació entre els usuaris i els ordinadors. La interacció amb cada un d'aquests components genera una sèrie de canvis d'estat i cada canvi d'estat és un esdeveniment susceptible de necessitar o provocar una acció determinada. És a dir, una possible incidència.

La programació de les aplicacions amb interfícies gràfiques s'elabora a partir d'una sèrie de components gràfics (des de formularis fins a controls, com els botons o les etiquetes), que es defineixen com a objectes propis, amb les seves variables i els seus mètodes.

Mentre que les variables corresponen a les diferents propietats necessàries per a la descripció de l'objecte (longituds, colors, bloquejos, etc.), els mètodes permeten la codificació d'una resposta a cada una de les diferents incidències que poden passar a aquest component.

5.11.1. Les interfícies d'usuari en Java

Java, des del seu origen en la versió 1.0, va implementar un paquet de rutines gràfiques anomenades *AWT* (*abstract windows toolkit*) incloses en el paquet `java.awt`, en què s'inclouen tots els components per a construir una interfície gràfica d'usuari (*GUI-graphic user interface*) i per a la gestió d'incidències. Aquest fet fa que les interfícies generades amb aquesta biblioteca funcionin en tots els entorns Java, inclosos els diferents navegadors.

Aquest paquet va patir una revisió que va millorar molts aspectes en la versió 1.1, però continuava presentant un inconvenient: AWT inclou components que depenen de la plataforma, la qual cosa ataca de manera frontal un dels pilars fonamentals en la filosofia de Java.

En la versió 1.2 (o Java 2) s'ha implementat una nova versió d'interfície gràfica que soluciona aquests problemes: el paquet Swing. Aquest paquet presenta, a més, una sèrie d'avantatges addicionals respecte a l'AWT com a aspecte modificable (diversos *look and feel,* com Metall –que és la presentació pròpia de Java–, Motif –pròpia de

ANOTACIONS

Unix–, Windows) i una àmplia varietat de components, que es poden identificar ràpidament perquè el seu nom comença per *J*.

Swing conserva la gestió d'incidències d'AWT, encara que l'enriqueix amb el paquet javax.swing.event.

El seu principal inconvenient és que alguns navegadors actuals no la inclouen inicialment, amb la qual cosa el seu ús en les miniaplicacions queda limitat.

Encara que l'objectiu d'aquest material no inclou el desenvolupament d'aplicacions amb interfícies gràfiques, un petit exemple de l'ús de la biblioteca Swing ens permetrà presentar les seves idees bàsiques i també l'ús de les incidències.

5.11.2. Exemple de miniaplicació de Swing

En el següent exemple, es defineix una miniaplicació que segueix la interfície Swing. La primera diferència respecte a la miniaplicació explicada anteriorment correspon a la inclusió del paquet `javax.swing.*`.

Es defineix la classe `HelloSwing` que hereta de la classe `Japplet` (que correspon a les miniaplicacions en Swing). En aquesta classe, es defineix el mètode `init`, on es defineix un botó nou (`new Jbutton`) i s'afegeix al tauler de la pantalla (`.add`).

Els botons reben incidències de la classe `ActionEvent` i, per al seu tractament, la classe que gestiona les seves incidències ha d'implementar la interfície `ActionListener`.

Per a aquesta funció s'ha declarat la classe `GestorIncidencies`, que al seu interior redefineix el mètode `actionPerformed` (l'únic mètode definit en la interfície `ActionListener`) de manera que obre una finestra nova per mitjà del mètode showMessageDialog.

Finalment, només falta indicar a la classe `HelloSwing` que la classe `GestorIncidencies` és la que gestiona els missatges del botó. Per a això, usem el mètode `.addActionListener(GestorIncidencies)`.

ANOTACIONS

HelloSwing.java

```
import javax.swing.*;
import java.awt.event.*;
public class HelloSwing extends Japplet
{
 public void init()
 { //Metodeconstructor
  JButton boto = new JButton("Pitja aquí! ");
  GestorIncidencies elmeuGestor = new GestorIncidencies();
   boto.addActionListener(elmeuGestor); //Gestor del boto
   getContentPane().add(boto);
  } // init
} // HelloSwing

class GestorIncidencies implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
   String titol = "Felicitats";
   String missatge = "Hola món, des de Swing";
   JOptionPane.showMessageDialog(null, missatge,
       titol, JOptionPane.INFORMATION_MESSAGE);

  } // actionPerformed
} // classe GestorIncidencies
```

5.12. Introducció a la informació visual

Encara que per motius de simplicitat s'han utilitzat entorns de desenvolupament en mode text, a la pràctica es fan servir entorns de desenvolupament integrats (IDE) que incorporen les diferents eines que faciliten al programador el procés de generació de codi font i executable (editor, compilador, depurador, etc.).

En ambdós casos, no hi ha cap diferència en el llenguatge de programació: es considera que el llenguatge és "textual", ja que les instruccions primitives s'expressen mitjançant text.

ANOTACIONS

Actualment, aquests entorns de desenvolupament proporcionen la possibilitat de treballar d'una manera més visual ja que permeten confeccionar les pantalles de treball (formularis, informes, etc.) mitjançant l'arrossegament dels diferents controls a la seva posició final, i després amb la introducció de valors per als seus atributs (colors, mesures, etc.) i el codi per a cada una de les incidències que pot provocar. No obstant això, la naturalesa del llenguatge no varia i es continua considerant "textual".

Un altre paradigma diferent correspondria a la programació mitjançant llenguatges "visuals". Parlem de *llenguatge visual* quan el llenguatge manipula informació visual, suporta interaccions visuals o permet la programació mitjançant expressions visuals. Per tant, les seves primitives són gràfics, animacions, dibuixos o icones.

En altres paraules, els programes es constitueixen com una relació entre diferents instruccions que es representen gràficament. Si la relació fos seqüencial i les instruccions s'expressessin mitjançant paraules, aquesta programació seria fàcilment recognoscible. En tot cas, ja s'ha comentat que la programació concurrent i la que es dirigeix per incidències no presenten una relació seqüencial entre les seves instruccions que, a més, solen ser d'alt nivell d'abstracció.

Així doncs, la programació visual resultaria ser una tècnica per a descriure programes els fluxos d'execució dels quals s'adaptin als paradigmes anteriorment esmentats.

Per tant, malgrat la possible confusió aportada pels noms de diversos entorns de programació com la família Visual de Microsoft (Visual C++, Visual Basic, etc.), a aquests llenguatges se'ls ha de continuar classificant com a llenguatges "textuals", encara que el seu entorn gràfic de desenvolupament sí que pot representar una aproximació cap a la programació visual.

5.13. Resum

En aquest capítol s'ha presentat un llenguatge de programació nou orientat a objectes que ens proporciona independència de la plata-

ANOTACIONS

forma sobre la qual s'executa. Per a això, proporciona una màquina virtual sobre cada plataforma. D'aquesta manera, el desenvolupador d'aplicacions només ha d'escriure el seu codi font una única vegada i compilar-lo per generar un codi "executable" comú, amb la qual cosa aconsegueix que l'aplicació pugui funcionar en entorns dispars com sistemes Unix, sistemes PC o Apple Macintosh. Aquesta filosofia és la que es coneix com a "write once, run everywhere".

Java va néixer com a evolució del C++ i es va adaptar a les condicions anteriorment descrites. S'aprofita el coneixement previ dels programadors en els llenguatges C i C++ per a facilitar una aproximació ràpida al llenguatge.

Atès que Java necessita un entorn de mida petita, permet incorporar el seu ús a navegadors web. D'altra banda, com que l'ús d'aquests navegadors implica, normalment, l'existència d'un entorn gràfic, s'ha aprofitat aquesta situació per a introduir breument l'ús de biblioteques gràfiques i el model de programació dirigit per incidències.

Així, Java inclou de manera estàndard dins del seu llenguatge operacions avançades que en altres llenguatges fan el sistema operatiu o biblioteques addicionals. Una d'aquestes característiques és la programació de diversos fils d'execució dins del mateix procés. Finalment ens hem pogut introduir breument en el tema.

5.14. Exercicis d'autoavaluació

1. Amplieu la classe Llegir.java per implementar la lectura de variables tipus `double`.

2. Introduïu la data (sol·licitant una cadena per a la població i tres nombres per a la data) i torneu-lo en forma de text.

Exemple

Entrada: Barcelona, 15 02 2003

Sortida: Barcelona, 15 de febrer de 2003

ANOTACIONS

3. Implementeu una aplicació que pugui diferenciar si una figura de quatre vèrtexs és un quadrat, un rectangle, un rombe o un altre tipus de polígon.

Es defineixen els casos de la manera següent:

- Quadrat: costats 1, 2,3 i 4 iguals; 2 diagonals iguals
- Rectangle: costats 1 i 3, 2 i 4 iguals; 2 diagonals iguals
- Rombe: costats 1, 2, 3 i 4 iguals, diagonals diferents
- Polígon: els altres casos

Per a això, es defineix la classe Punt2D definint les coordenades *x*,*y*, i el mètode "distància a un altre punt".

Exemple

(0,0) (1,0) (1,1) (0,1) Quadrat
(0,1) (1,0) (2,1) (1,2) Quadrat
(0,0) (2,0) (2,1) (0,1) Rectangle
(0,2) (1,0) (2,2) (1,4) Rombe

4. Convertiu el codi de l'exercici de l'ascensor (exercici 3 del capítol 4) a Java.

5.14.1. Solucionari

1.

Llegir.java

```
import java.io.*;

public class Llegir
{
   public static String getString()
   {
     String str = '';
     try
     {
       InputStreamReader isr = new
```

ANOTACIONS

```
      InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      str = br.readLine();
     }
     catch(IOException e)
  { System.err.println("Error: " + e.getMessage());
}
     return str; // tornar la dada escrita
   }

   public static int getInt()
   {
    try
    { return Integer.parseInt(getString()); }
    catch(NumberFormatException e)
    { return 0; // Integer.MIN_VALUE }
   } // getInt

   public static double getDouble()
   {
    try
    {
     return Double.parseDouble(getString());
    }
    catch(NumberFormatException e)
    {
     return 0; // Double.MIN_VALUE
    }
   } // getDouble
} // Llegir
```

2.

construirData.java

```
import java.io.*;
public class construirData
{
   static String nomMes(int nmes)
   {
    String strmes;
```

```
    switch (nmes)
    {
     case 1: { strmes = "gener"; break; }
     case 2: { strmes = "febrer"; break; }
     case 3: { strmes = "març"; break; }
     case 4: { strmes = "abril"; break; }
     case 5: { strmes = "maig"; break; }
     case 6: { strmes = "juny"; break; }
     case 7: { strmes = "juliol"; break; }
     case 8: { strmes = "agost"; break; }
     case 9: { strmes = "setembre"; break; }
     case 10: { strmes = "octubre"; break; }
     case 11: { strmes = "novembre"; break; }
     case 12: { strmes = "desembre"; break; }
     default: { strmes = " -- "; break; }
    } // switch nmes
    return (strmes);
   } // nomMes

   public static void main(String args[])
   {
    String poblacio;
    int dia, mes, any;
    String laMevaData, strmes;
    System.out.print(" Població: ");
    població = Llegir.getString();
    System.out.print(" Dia: ");
    dia = Llegir.getInt();
    System.out.print(" Mes: ");
    mes = Llegir.getInt();
    System.out.print(" Any: ");
    any = Llegir.getInt();
    laMevaData = poblacio + ", " + dia;
   laMevaData = laMevaData + "de " + nomMes(mes);
   laMevaData = laMevaData + "de " + any;
   System.out.print(" la data introduïda és: ");
   System.out.println(laMevaData);
   } // main
} // class
```

ANOTACIONS

3.

Punt2D.java

```
class Punt2D
{
   public int x, y;
   // Inicialitzant l'origen de coordenades
   Punt2D()
   { x = 0; y = 0; }

  // Inicialitzant una coordenada x,y determinada
   Punt2D(int coordx, int coordy)
   { x = coordx; y = coordy; }

   // Calcula la distància en un altre punt
   double distancia(Punt2D elMeuPunt)
   {
    int dx = x - elMeuPunt.x;
    int dy = y - elMeuPunt.y;
    return ( Math.sqrt(dx * dx + dy * dy));
   }
}
```

AppReconeixerFigura.java

```
class AppReconeixerFigura
{
static public void main(String args[])
   {
   int i;
   int coordx, coordy;

   // Introduir 4 punts i
   // indicar quin es el més proper a l'origen.

   Punt2D llistaPunts[];
   llistaPunts = new Punt2D[4];

   // Entrar dades
   for (i=0; i<4; i++)
   {
```

ANOTACIONS

```
 System.out.println("Entrar el punt (" + i + ") " );
 System.out.print("Coordenada x " );
 coordx = Leer.getInt();
 System.out.print("Coordenada y " );
 coordy = Llegir.getInt();
 llistaPunts[i] = new Punt2D(coordx, coordy);
} //for

// Indicar si els 4 punts formen un
// quadrat: dist1 = dist2 = dist3 = dist4
// diag1 = diag2
// rombe: dist1 = dist2 = dist3 = dist4
// diag1 <> diag2
// rectangle: dist1 = dist3, dist2 = dist4
// diag1 = diag2
// poligon: altres casos

double dist[] = new double[4];
double diag[] = new double[3];

// Càlcul de distàncies
for (i=0; i<3; i++)
{
 dist[i] = llistaPunts[i].distancia(llistaPunts[i+1]);
 System.out.print("Distancia "+i + " + dist[i] );
} //for
dist[3] = llistaPunts[3].distancia(llistaPunts[0]);
System.out.println("Distancia "+i + " + dist[3] );

// Càlcul de diagonals
for (i=0; i<2; i++)
{
 diag[i] = llistaPunts[i].distancia(llistaPunts[i+2]);
} //for
if ( (dist[0] == dist[2]) && (dist[1] == dist[3]) )
{

// És quadrat, rectangle o rombe
 if (dist[1] == dist[2]) {
 // És quadrat o rombe
```

ANOTACIONS

```
      if (diag[0] == diag[1])
        { System.out.println("És un quadrat"); }
        else
        { System.out.println("És un rombe"); } // if
        }
        else
        {

      // És rectangle
      if (diag[0] == diag[1])
        { System.out.println("És un rectangle"); }
        } else
        { System.out.println("És un polígon"); } // if
      }
     } else
      { System.out.println("És un polígon"); } // if
      } // main
  } // class
```

4.

AppAscensor.java

```
import java.io.*;

public class AppAscensor{

   static int n_Codi, n_Pes, n_idioma;

   public static void sollicitarDades()
   {
    System.out.print ("codi: ");
    n_Codi = Llegir.getInt();

    System.out.print("Pes: ");
    n_Pes = Llegir.getInt();

    System.out.print(
          "Idioma: [1] Català [2] Castellà [3] Anglès ");
    n_idioma = Llegir.getInt();
   } // sol·licitarDades

   public static void mostrarEstatAscensor(Ascensor nA)
   {
```

```
    nA.mostrarOcupacio();
    System.out.print(" - ");
    nA.mostrarCarrega();
    System.out.println(" ");
    nA.mostrarLlistaPassatgers();
  } // mostrarEstatAscensor

  public static void main( String[] args)
  {
    int opc;
    boolean sortir = false;
    Ascensor unAscensor;
    Persona unaPersona;
    Persona localitzarPersona;

    opc=0;

  unAscensor = new Ascensor();// Inicialitzem ascensor
  unaPersona = null;// Inicialitzem unaPersona

  do {

      System.out.print(
"ASCENSOR: [1]Entrar [2]Sortir [3]Estat [0]Finalitzar ");
    opc = Llegir.getInt();

    switch (opc)
    {
    case 1: { // Opcio Entrar
     sollicitarDades();
     switch (n_idioma)
     {
      case 1: { //"Catala"
       unaPersona = new Catala (n_Codi, n_Pes);
       break;
      }
      case 2: { //"Castella"
       unaPersona = new Castella (n_Codi, n_Pes);
       break;
      }
```

ANOTACIONS

```
    case 3: { //"Angles"
     unaPersona = new Angles(n_Codi, n_Pes);
     break;
    }
     default: { //"Angles"
     unaPersona = new Angles(n_Codi, n_Pes);
     break;
    }
   } //switch n_idioma

   if (unAscensor.persona_PotEntrar(unaPersona))
   {
    unAscensor.persona_Entrar(unaPersona);
    if (unAscensor.obtenirOcupacio()>1)
    {
     System.out.print(unaPersona.obtenirCodi());
     System.out.print(" diu: ");
     unaPersona.saludar();
     System.out.println(" "); // Responen les altres
     unAscensor.restaAscensor_Saludar(unaPersona);
    }
   } //pot entrar
   break;
  }
  case 2: { //Opcio Sortir

   localitzarPersona = new Persona(); //Per exemple
   unaPersona = null;

   localitzarPersona.sollicitarCodi();
   if (unAscensor.persona_Seleccionar(localitzarPersona))
   {
    unaPersona = unAscensor.obtenirRefPersona();
    unAscensor.persona_Sortir( unaPersona );
    if (unAscensor.obtenirOcupacio()>0)
    {
     System.out.print(unaPersona.obtenirCodi());
     System.out.print(" diu: ");
     unaPersona.acomiadarse();
     System.out.println(" "); // Responen les altres
     unAscensor.restaAscensor_Acomiadarse(unaPersona);
```

ANOTACIONS

```
          unaPersona=null;
        }
       } else {
          System.out.println(
            "No hi ha cap persona amb aquest codi");
       } // seleccionar
       localitzarPersona=null;
       break;
      }
      case 3: { //Estat
       mostrarEstat(unAscensor);
       break;
      }

      case 0: { //Finalitzar
       System.out.println("Finalitzar");
       sortir = true;
       break;
      }

      } //switch opc
    } while (! sortir);
    } // main
} //AppAscensor
```

Ascensor.java

```
import java.io.*;

class Ascensor {
   private int ocupacio;
   private int carrega;
   private int ocupacioMaxima;
   private int carregaMaxima;
   private Persona passatgers[];
   private Persona refPersonaSeleccionada;
//
// Mètodes constructors i destructors //
Ascensor()
{
```

ANOTACIONS

```
   ocupacio = 0;
   carrega=0;
   ocupacioMaxima=6;
   carregaMaxima=500;
   passatgers = new Persona[6];
   refPersonaSeleccionada = null;
} //Ascensor ()
// Funcions d'accés
int ObtenirOcupacio()
{ return (ocupacio); }

void ModificarOcupacio(int dif_ocupacio)
{ ocupacio += dif_ocupacio; }

void MostrarOcupacio()
{
   System.out.print("Ocupacio actual: ");
   System.out.print(ocupacio );
}

int obtenirCarrega()
{ return (carrega); }

void modificarCarrega(int dif_carrega)
{ carrega += dif_carrega; }

void mostrarCarrega()
{  System.out.print("Carrega actual: ");
   System.out.print(carrega) ;
}

Persona obtenirRefPersona()
{return (refPersonaSeleccionada);}

boolean persona_PotEntrar(Persona unaPersona)
{
   // Si l'ocupació no sobrepassa el límit d'ocupació i
   // si la càrrega no sobrepassa el límit de càrrega
   // ->pot entrar}
```

ANOTACIONS

```
   boolean tmpPotEntrar;
   if (ocupacio + 1 > ocupacioMaxima)
   {
    System.out.println(
   "Avís: L'ascensor és complet. No pot entrar");
    return (false);
   }

   if (unaPersona.obtenirPes() + carrega > carregaMaxima )
   {
    System.out.println(
"Avís: L'ascensor supera la seva càrrega màxima. No pot entrar");
    return (false);
   }
   return (true);
}
boolean persona_Seleccionar(Persona localitzarPersona) {
   int comptador;

   // Se selecciona persona entre passatgers de l'ascensor.
   boolean personaTrobada = false;
   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do {
     if (passatgers[comptador] != null)
     {
      if(passatgers[comptador].igualCodi(localitzarPersona)
      {
       refPersonaSeleccionada=passatgers[comptador];
       personaTrobada=true;
       break;
      }
     comptador++;
    } while (comptador<ocupacioMaxima);
    if (comptador>=ocupacioMaxima)
     {refPersonaSeleccionada=null;}
   }
   return (personaTrobada);
}
```

ANOTACIONS

```
void persona_Entrar(Persona unaPersona)
{
   int comptador;
   modificarOcupacio(1);
   modificarCarrega(unaPersona.obtenirPes());
   System.out.print(unaPersona.obtenirCodi());
   System.out.println(" entra a l'ascensor ");

   comptador=0;
   // Hem verificat anteriorment que hi ha places lliures
   do {
    if (passatgers[comptador]==null )
    {
     passatgers[comptador]=unaPersona;
     break;
    }
     comptador++;
   } while (comptador<ocupacioMaxima);
}

void persona_Sortir(Persona unaPersona)
{
   int comptador;

   comptador=0;
   do {
    if ((passatgers[comptador]==unaPersona ))
    {
     System.out.print(unaPersona.obtenirCodi());
     System.out.println(" surt de l'ascensor ");
     passatgers[comptador]=null;

     // Modifiquem l'ocupació i la càrrega
     modificarOcupacio(-1);
     modificarCarrega(-1 * (unaPersona.obtenirPes()));
     break;
    }
    comptador++;
   } while (comptador<ocupacioMaxima);
   if (comptador==ocupacioMaxima)
    {System.out.println(
```

ANOTACIONS

```
    "No hi ha cap persona amb aquest codi. No surt ningú ");}
}

void mostrarLlistaPassatgers()
{
   int comptador;
   Persona unaPersona;

   if (obtenirOcupacio() > 0)
   {
    System.out.println("Llista de passatgers de l'ascensor: ");
    comptador=0;
    do {
     if (!(passatgers[comptador]==null ))
     {
      unaPersona=passatgers[comptador];
      System.out.print(unaPersona.obtenirCodi());
      System.out.print("; ");
     }
     comptador++;
    } while (comptador<ocupacioMaxima);
    System.out.println(" ");
   } else {
    System.out.println("L'ascensor és buit");
   }
}

void restaAscensor_Saludar(Persona unaPersona)
{
   int comptador;
   Persona unaAltraPersona;

   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do {
     if (!(passatgers[comptador]==null ))
     {
      unaAltraPersona=passatgers[comptador];
      if (!unaPersona.igualCodi(unaAltraPersona) )
```

ANOTACIONS

```
        {
         System.out.print(unaAltraPersona.obtenirCodi());
         System.out.print(" respon: ");
         unaAltraPersona.saludar();
         System.out.println(" ");
        }
       }
       comptador++;
     } while (comptador<ocupacioMaxima);
   }
}

void restaAscensor_Acomiadarse(Persona unaPersona)
{
   int comptador;
   Persona unaAltraPersona;

   if (obtenirOcupacio() > 0)
   {
    comptador=0;
    do {
      if (!(passatgers[comptador]==null ))
      {
       unaAltraPersona=passatgers[comptador];
       if (!(unaPersona.igualCodi(unaAltraPersona))
       {
         System.out.print(unaaltraPersona.obtenirCodi());
         System.out.print(" respon: ");
         unaaltraPersona.acomiadarse();
         System.out.print(" ");
       }
      }
      comptador++;
    } while (comptador<ocupacioMaxima);
   }
}

} // class Ascensor
```

ANOTACIONS

Persona.java

```
import java.io.*;

class Persona {
   private int codi;
   private int pes;

   Persona()
   { }

   Persona(int n_Codi, int n_Pes)
   {
    codi = n_Codi;
    pes = n_Pes;
   }

   public int obtenirPes()
    { return (pes); }

   public void assignarPes(int n_Pes)
    { pes = n_Pes; }

   public int obtenirCodi()
   { return (codi); }

   public void assignarCodi(int n_Codi)
   { this.Codi = n_Codi; }

   public void assignarPersona(int n_Codi, int n_Pes)
   {
    assignarCodi( n_Codi );
    assignarPes( n_Pes );
   }
   void saludar() {};
   void acomiadarse() {};

   public void sollicitarCodi()
   {
    int n_codi=0;
```

ANOTACIONS

```
    System.out.print ("codi: ");
    n_codi = Llegir.getInt();
    assignarCodi (n_Codi);
   }

   public boolean igualCodi(Persona unaAltraPersona)
   {
    return (this.obtenirCodi()==unaAltraPersona.obtenirCodi());
   }
} //class Persona
```

Catala.java

```
class Catala extends Persona
{

   Catala()
   { Persona(0, 0); };

   Catala(int n_Codi, int n_Pes)
   { Persona (n_Codi, n_Pes); };

   void saludar()
   { System.out.println("Bon dia"); };

   void acomiadarse()
   { System.out.println("Adéu"); };
}
```

Castella.java

```
class Castella extends Persona
{

   Castella()
   { Persona(0, 0); };

   Castella(int n_Codi, int n_Pes)
   { Persona (n_Codi, n_Pes); };
```

```
   void saludar()
   { System.out.println("Buenos dias"); };

   void acomiadarse()
   { System.out.println("Adios"); };
}
```

Angles.java

```
class Angles extends Persona
{

   Angles ()
   { Persona(0, 0); };

   Angles (int n_Codi, int n_Pes)
   { Persona (n_Codi, n_Pes); };

   void saludar()
   { System.out.println("Hello"); };

   void acomiadarse()
   { System.out.println("Bye"); };
}
```

ANOTACIONS

Glossari

abstract windows toolkit

m Paquet de rutines gràfiques incloses en el paquet *java.awt*, en què s'inclouen tots els components per a proporcionar una interfície gràfica d'usuari (*GUI-graphic user interface*) a les aplicacions Java.

sigla: AWT

API

Vegeu application programming interface.

aplicació

f Programari amb una determinada finalitat: processament de textos, gestió comptable, control de processos industrials, entreteniment, multimèdia, etc.

applet Java

Vegeu miniaplicació de Java.

application programming interface

f Col·lecció de programari ja desenvolupat que proporciona múltiples capacitats com entorns gràfics, comunicacions, multiprocés, etc. i que està incorporat a la plataforma Java.

sigla: API

apuntador

m Variable que conté una adreça de memòria, que és, habitualment, l'adreça d'alguna altra variable.

AWT

Vegeu abstract windows toolkit.

ANOTACIONS

biblioteca

f Arxiu d'unitats de compilació ja compilades i llestes per a ser enllaçades amb altres programes. Conjunts de funcions precompilats.

bri

Vegeu fil.

bytecode

m Codi generat pel compilador de Java preparat per a ser executat per la màquina virtual de Java (JVM).

cerca

f Procés de tractament parcial d'una seqüència de dades. És un recorregut que s'atura quan es compleix el criteri de la cerca.

classe

f Tipus de dades abstracte que inclou dades i funcions, i que representa un model d'una entitat genèrica.

compilador

m Programa que tradueix el text en un llenguatge de programació a un codi en un altre llenguatge de programació, especialment a codi en llenguatge màquina.

constructor

m Funció membre especial d'una classe que es crida automàticament cada vegada que s'instancia un objecte d'aquesta classe.

cua

f Llista en què la inserció dels elements es fa per un extrem i l'eliminació per l'altre.

debugger

Vegeu depurador.

depurador

m Eina que permet observar l'estat d'un programa i controlar la seva execució. S'utilitza per a localitzar errors.

en debugger.

ANOTACIONS

destructor

m Funció membre especial d'una classe que s'anomena automàticament cada vegada que es destrueix un objecte.

eina

f Programa d'utilitat per a alguna aplicació determinada. Les eines de desenvolupament de programari inclouen, entre d'altres, un editor de text, un compilador, un enllaçador i un depurador.

encapsulament

m Propietat per la qual un element actua com una "caixa negra", on s'especifiquen unes entrades, una idea general del seu funcionament i unes sortides.

enllaçador

m Programa que enllaça diversos codis en llenguatge màquina per muntar un únic fitxer que contingui el codi d'un programa executable.

entorn d'un programa

m Conjunt de variables i la seqüència d'instruccions que les utilitzen i modifiquen.

esquema algorítmic

m Algoritme genèric les instruccions del qual s'han de reemplaçar per les requerides per a un cas particular.

estat d'un programa

m Contingut de les variables i la indicació de la instrucció en curs d'execució en un moment determinat.

estructura de dades

f Agrupació de dades organitzada amb mètodes d'accés específics.

fil

m Seqüència d'instruccions que s'executa paral·lelament a d'altres dins del mateix programa. També s'anomena *procés lleuger* perquè comparteix l'entorn d'execució amb els altres fils del mateix programa.

en thread.

ANOTACIONS

fil d'execució

m Flux d'instruccions que s'executa de manera simultània dins d'un mateix procés.

filtre

m Programa per al recorregut de seqüències de dades la sortida del qual és producte del tractament individual de les dades d'entrada (el nom deriva de les sortides que són producte de la còpia parcial de les dades d'entrada).

fitxer

m Estructura de dades habitualment emmagatzemada en memòria secundària que es caracteritza per no tenir una mida fixa.

funció

f Subprograma que duu a terme una tasca a partir d'uns paràmetres determinats el resultat del qual torna al programa que en fa ús.

funció membre (o mètode)

f Funció definida dins d'una classe i que actua o modifica les variables membres d'aquesta classe.

graphic user interface

Vegeu interfície gràfica d'usuari.

herència

f Propietat per la qual un objecte d'una classe pot rebre les propietats (dades i funcions) d'una classe més general.

homonímia

f Propietat per la qual dos o més elements que tenen el mateix nom fan operacions diferents.

IDE

Vegeu integrated development environment.

incidència

f Esdeveniment o procés que té lloc durant el funcionament d'un programa informàtic, automàticament o a requeriment de l'usuari per mitjà del teclat o el ratolí.

en event.

ANOTACIONS

integrated development environment

m Entorn de desenvolupament que integra les diferents eines per al desenvolupament de programari (edició, compilació, correcció d'errors, etc.).

sigla: IDE

interface

Vegeu interfície.

interfície

f Col·lecció de definicions de mètodes (sense les seves implementacions), la funció de les quals és definir un protocol de comportament per a una classe.

en interface.

interfície gràfica d'usuari

f Col·lecció d'objectes i eines gràfiques utilitzades per a gestionar la comunicació entre usuaris i ordinadors.

en graphic user interface

Java development kit

m Conjunt de programes i llibreries que permet el desenvolupament, compilació i execució d'aplicacions en Java.

sigla: JDK

JDK

Vegeu Java development kit.

JVM

Vegeu màquina virtual de Java.

llenguatge acoblador

m Llenguatge que s'utilitza per a programar directament en codi executable, ja que les seves instruccions es corresponen amb instruccions del llenguatge màquina corresponent.

llenguatge de programació

m Llenguatge que s'utilitza per a descriure un programa d'ordinador.

ANOTACIONS

llenguatge màquina

m Llenguatge que pot interpretar el processador d'un ordinador.

llenguatge unificat de modelatge

m Llenguatge unificat per al desenvolupament de programari, que permet la descripció de manera visual d'un model, els elements que el componen i les relacions que hi ha entre ells.

sigla: UML

llista

f Tipus de dada dinàmica en què la relació entre els seus diferents elements o nodes és seqüencial. És a dir, cada element en té un altre que el precedeix i un altre que el succeeix a excepció del primer i de l'últim.

màquina virtual de Java

f Plataforma de programari que permet l'execució d'un codi genèric (*bytecodes*) sobre una plataforma subjacent.

sigla: JVM

en Java virtual machine.

miniaplicació de Java

f Miniaplicació pensada per a ser executada des de navegadors web.

en applet Java.

node

m Cadascuna de les variables en una estructura de dades dinàmica.

objecte

m Instanciació d'una classe; és a dir, un element concret d'aquesta classe.

ocultació de dades

f Propietat per la qual es limita la visibilitat de les variables o mètodes d'una classe a una altra perquè li és aliena o ha redefinit els seus valors.

paquet

m Col·lecció de classes i interfícies relacionades que s'agrupen sota un mateix nom i que proporcionen protecció d'accés i gestió de l'espai de noms.

en package.

ANOTACIONS

polimorfisme

m Propietat per la qual un objecte pot adquirir diverses formes.

procés

m Flux seqüencial d'execució d'instruccions.

procés lleuger

Vegeu fil.

programa

m Seqüència d'instruccions.

programació

f Acció de transformar un algoritme en una seqüència d'instruccions que pugui executar un ordinador.

programació concurrent

f Programació en què es prepara el codi per a executar-se en diversos fluxos d'instruccions de manera concurrent o paral·lela, tant per a aprofitar la capacitat dels sistemes multiprocessador com per a evitar esperes conjuntes a processos d'entrada/sortida.

programació estructurada

f Programació que es fa mitjançant composició seqüencial d'instruccions (avaluació d'expressions, execució condicional o alternativa i iteració).

programació imperativa

f Programació que es fa mitjançant llenguatges les instruccions dels quals són ordres per a canviar l'estat de l'entorn del programa.

programació modular

f Programació el codi resultant del qual es divideix en subprogrames.

programari

m Conjunt de programes.

ANOTACIONS

programari lliure

m Conjunt de programes que s'ofereixen amb el codi font perquè es puguin modificar lliurement.

recorregut

m Procés de tractament d'una seqüència de dades en què participen tots.

signatura (d'una funció)

f Llista de característiques que, unides al nom de la funció, completen la seva definició: el seu tipus de retorn, el nombre de paràmetres i els seus tipus, i l'especificador `const` en cas que n'hi hagi.

Swing

m Paquet de rutines gràfiques que permeten crear una GUI implementades a partir de la versió Java 1.2 que substitueix les AWT.

thread

Vegeu fil d'execució.

tipus de dada abstracta

m Tipus de dada definida en el programa, inexistent en el llenguatge de programació.

tipus de dada dinàmica

m Tipus estructurat de dades en què pot variar tant el nombre d'elements, com la relació entre ells durant l'execució d'un programa.

variable dinàmica

f Variable creada durant l'execució d'un programa.

variable membre

f Variable definida dins d'una classe.

ANOTACIONS

Bibliografia

Antonakos, J.L.; Mansfield, K.C. (Jr.) (1997). *Programación estructurada en C*. Madrid: Prentice-Hall.

Eck, D.J. (2002). *Introduction to Programming Using Java* (4a. ed.). <http://math.hws.edu/eck/cs124/downloads/javanotes4.pdf>

Eckel, B. (2002). *Thinking in Java* (3a. ed.). <http://www.BruceEckel.com>

Eckel, B. (2003). *Thinking in C++* (2a. ed.). <http://www.BruceEckel.com>

Gottfried, B. (1997). *Programación en C* (2a. ed.). Madrid: McGraw-Hill.

Joyanes, L.; Castillo, A.; Sánchez, L.; Zahonero, I. (2002). *Programación en C (Libro de problemas)*. Madrid: McGraw-Hill.

Joyanes, L.; Zahonero, I. (2002). *Programación en C. Metodología, estructura de datos y objetos*. Madrid: McGraw-Hill.

Kernighan, B.W.; Pike, R. (2000). *La práctica de la programación*. Madrid: Prentice Hall.

Kernighan, B.W.; Ritchie, D.M. (1991). *El lenguaje de programación C* (2a.ed.). Mèxic: Prentice-Hall.

Liberty, J.; Horvath, D.B. (2001). *Aprendiendo C++ para Linux en 21 Días*. Mèxic: Pearson Educación.

Palma, J.T.; Garrido, M.C.; Sánchez, F.; Santos, J.M. (2003). *Progrramación concurrente*. Madrid: Thomson.

Pressman, R. (1998). *Ingeniería del Software. Un Enfoque Práctico* (4a. ed.). Madrid: McGraw Hill.

Quero, E.; López, J. (1997). *Programación en lenguajes estructurados*. Madrid: Int. Thomson Publishing Co.

Ribas Xirgo, Ll. (2000). *Programació en C per a matemàtics*. Bellaterra (Cerdanyola): Servei de Publicacions de la UAB.

ANOTACIONS

Sun Microsystems, Inc. (2003). *The Java Tutorial: A Practical Guide for Programmers*. <http://java.sun.com/docs/books/tutorial>

Tucker, A.; Noonan, R. (2003). *Lenguajes de programación. Principios y paradigmas*. Madrid: McGraw-Hill.

ANOTACIONS

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

ANOTACIONS

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

ANOTACIONS

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

ANOTACIONS

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

ANOTACIONS

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

ANOTACIONS

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

ANOTACIONS

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

ANOTACIONS

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

ANOTACIONS

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of

ANOTACIONS

those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

ANOTACIONS

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

ANOTACIONS