PROGRAMACIÓ I

(1ª PART)

ENGINYERIA EN INFORMÀTICA
i
ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Universitat Pompeu Fabra
Curs 2003-2004

Toni Navarrete

Tutories: dijous de 10:30 a 11:30 i de 18:30 a 19:30
Despatx: 371
e-mail: toni.navarrete@upf.edu
Web assignatura: http://www.tecn.upf.es/~tnavarrete/programacio1

Índex

1. INTRODUCCIÓ I CONCEPTES GENERALS 2

2. ELS TIPUS DE DADES BÀSICS 6

2.1. VARIABLES I CONSTANTS.. 6
2.2. ELS DIFERENTS TIPUS 8
2.2.1. Tipus numèrics 8
2.2.2. Caràcters 9
2.2.2. Booleans 10

3. EXPRESSIONS, SENTÈNCIES I ESTRUCTURES DE CONTROL 11

3.1. LES EXPRESSIONS 11
3.1.1 Lògica booleana 12
3.2. SENTÈNCIES O INSTRUCCIONS 13
3.2.1. Assignacions 13
3.2.1. Operacions d'entrada i sortida 14
3.2.1. L'ordre de precedència dels operadors 15
3.3. LES ESTRUCTURES DE CONTROL 16
3.3.1. Les estructures condicionals 17
3.3.2. Les estructures iteratives 21

4. TIPUS DE DADES COMPOSTS ESTÀTICS: *ARRAYS*, *STRINGS* I *STRUCTS* 26

4.1. ARRAYS UNIDIMENSIONALS 26
4.1.1. Escriure tots els elements d'un vector 27
4.1.2. Calcular la mitjana de tots els elements d'un vector 28
4.1.3. Cercar l'element major d'un vector 28
4.1.4. Cercar si un valor concret és a un vector 29
4.1.5. Comptar quantes vegades apareix un valor concret a un vector 30
4.2. ARRAYS MULTIDIMENSIONALS 30
4.2.1. Escriure tots els elements d'una matriu 31
4.2.2. Sumar dues matrius 32
4.2.3. Multiplicar dues matrius 32
4.3. STRINGS O CADENES DE CARÀCTERS 33
4.4. ESTRUCTURES (STRUCTS) 35

5. ELS PUNTERS 37

5.1. DECLARACIÓ D'UN PUNTER 37
5.2. INDIRECCIÓ 37
5.3. ASSIGNACIÓ DE VALORS A UN PUNTER 38
5.4. EL PUNTER NUL 40
5.5. UN EXEMPLE AMB PUNTERS 40

BIBLIOGRAFIA 42

1. Introducció i conceptes generals

A qualsevol ordinador podem distingir dos elements essencials: el *hardware* o maquinari i el *software* o programari. El *hardware* és la part física, els ferros i cables, la màquina, mentre que el software són els programes que s'executen sobre aquesta màquina. En aquesta assignatura ens dediquem a estudiar aquest segon element, el software o programari, i més en concret en com es confeccionen els programes.

Un programa informàtic és el resultat final d'un procés que comença amb el plantejament d'un problema. Un programa és un conjunt d'instruccions escrites amb un llenguatge de programació i que seran executades a un ordinador.

Cada màquina reconeix un llenguatge propi, anomenat llenguatge màquina o ensamblador, que consta d'unes instruccions molt a baix nivell, en les quals s'accedeix directament als elements físics de la màquina com per exemple els registres. Per exemple són del tipus: copia el que hi ha a una posició concreta de la memòria a un registre concret o suma el que hi ha a dos registres o copia el que hi ha un registre a una altra posició de memòria. La unitat de control (a la unitat central de procés o CPU) és qui és capaç d'entendre aquestes instruccions senzilles i controlar la seva execució. Però noteu que cada màquina té una configuració física diferent, i també un llenguatge ensamblador diferent. Per exemple, és diferent l'ensamblador d'un PC que el d'un Mac.

A la primera etapa de la informàtica tots els programes es feien utilitzant el codi màquina, però a part de que és molt complicat de programar i molt difícil de llegir, una vegada tenim el programa només el podem executar a un tipus de màquina. Si canviem de màquina perdem tots els nostres programes.

És per aquests factors que apareixen els llenguatges de programació d'alt nivell, on es cerca per una banda un llenguatge més fàcil d'escriure i llegir, i per l'altra la portabilitat, és a dir que puguem portar els programes d'una màquina a una altra. Noteu però que el que realment la màquina executa continua essent codi màquina, però el programador escriu en un altre llenguatge. Cal, per tant, una etapa de *traducció* intermèdia. Aquesta *traducció* pot ser per dos mecanismes diferents: la compilació i la interpretació:

- Compilació: abans d'executar el nostre programa, el compilem. El que anomenem compilador (que és un altre programa) primer reconeix si el nostre codi està escrit seguint les regles del llenguatge i si no hi ha errors ho tradueix a codi màquina. El que posteriorment s'executa és el codi màquina que s'ha generat.

- Interpretació: el codi es va executant instrucció a instrucció, i una a una es van traduint a codi màquina. Qui ho fa és l'intèrpret (també un altre programa).

Hi ha un tercer esquema que és el de Java, que utilitza una solució mixta, primer amb un compilador que no genera codi màquina per la màquina física sinó pel que s'anomena una màquina virtual. A aquest codi màquina virtual se li anomena byte-code. La màquina virtual en realitat és un intèrpret que després va executant (traduint a codi màquina real) instrucció a instrucció el byte-code.

La majoria de llenguatges de programació, almenys els de la família dels llenguatges imperatius que són els més abundants (com C, Java, Pascal, Visual Basic,...), tenen unes característiques molt semblants. L'objectiu de l'assignatura és aprendre aquests mecanismes comuns que són presents a tots els llenguatges per facilitar l'aprenentatge de nous llenguatges. Després cada llenguatge també té les seves particularitats, i en veurem les específiques del C.

Però donat un problema concret, aquest es pot resoldre de diferents formes, i més encara a mesura que els problemes són majors. Imaginem, per exemple, que hem d'ordenar una llista de nombres. Una manera per solucionar el problema seria cercar el més petit de tots i posar-lo en la primera posició, després en cerquem el segon més petit i el col·loquem a la segona posició i així successivament. Noteu que el que hem explicat és una forma de resoldre un problema, independentment de quin llenguatge s'utilitzi després per escriure el programa. A aquesta explicació de la solució se li anomena algorisme. Així doncs, un algorisme és una especificació d'una solució a un problema. L'algorisme contarà d'una sèrie d'accions, a més dels objectes que necessitem per dur-la a terme. L'algorísmica existeix en matemàtiques molt abans del naixement dels programes informàtics. Serà molt habitual que abans d'escriure un programa donat un problema, per facilitar-nos la tasca, passem abans per la fase d'escriure l'algorisme, almenys en línies generals. Per problemes petits, serà freqüent que no ens calgui escriure un algorisme, però com més gran sigui el problema, més important serà aquesta fase intermèdia. Els algorismes s'escriuen en "llenguatge humà", en català en el nostre cas (a diferència del programa que està escrit en un llenguatge de programació). Per facilitar al programador el fet d'entendre l'algorisme, hi ha diferents normes que especifiquen unes regles sintàctiques per escriure un algorisme, amb el que s'anomena pseudo-codi. En aquesta assignatura no seguirem cap norma sinó que utilitzarem la sintaxi del propi C.

En realitat són moltes més les etapes, especialment en problemes grans, però això no ho tractarem en aquesta assignatura sinó en Enginyeria del Software.

Alguns exemples d'algorismes

Problema: calcular la mitjana de dos nombres amb una calculadora tradicional
Objectes: una calculadora (no científica)
Solució:

1. Posar en marxa la calculadora pitjant el botó "On"
2. Introduir el primer nombre
3. Pitjar la tecla "+"
4. Introduir el segon nombre
5. Pitjar la tecla "/"
6. Teclejar el nombre "2"
7. Pitjar la tecla "="

Aquesta seqüència de set accions ens descriu com ho hem de fer per resoldre un determinat problema. Com podem veure és molt similar a una recepta de cuina. A un algorisme hem de dir els ingredients i estris que necessitem (els objectes de què parlàvem) i com es preparen (les accions). El següent exemple és encara més culinari:

Problema: fer una truita

Objectes: un ou, una paella, oli, dos plats, una forquilla
Solució:
1. Rompre un ou dins un plat
2. Batre'l amb una forquilla
3. Escalfar oli a una paella
4. Tirar l'ou batut dins la paella
5. Treure l'ou i posar-lo a un plat net

Als dos exemples que hem vist les accions s'executen una darrera l'altra, en ordre. Però hi ha situacions en què necessitem alterar aquest ordre i repetir una certa acció un nombre determinat de vegades, o en funció de quelcom fer una acció o una altra. Profunditzarem sobre això en el capítol tercer, a l'apartat de les estructures de control.

El llenguatge C

C va se creat el 1972 per Dennis Ritchie en els Laboratories Bell de l'empresa AT&T amb el propòsit de desenvolupar el sistema operatiu UNIX. Durant alguns anys, l'estàndard de C era el que se suministrarva amb la versió 5 de UNIX. Però amb la creixent popularitat dels micro-ordinadors van aparèixer moltes implementacions amb diferències entre elles. Per això, al 1983 es va crear un comitè que va elaborar un document que defineix l'estàndard ANSI (l'organisme estandaritzador dels Estants Units) per C.

C és un llenguatge de més baix nivell que altres com Pascal. Certs aspectes més complexes del llenguatge s'han aparcat per l'assignatura de Programació II per facilitar l'assimilació dels conceptes bàsics d'algorísmica i programació.

Un primer programa en C

El següent és un primer programa en C, molt senzill que mostrarà el missatge "Hola" per pantalla.

```
# include <stdio.h>

main ()
{
      printf("Hola");
}
```

Hem d'acostumar-nos a introduir en el codi tots els comentaris que estimem que seran útils per desprès poder entendre'l, ja sigui per altres persones, o per nosaltres mateixos al cap d'un cert temps.

Per fer-ho, en C s'utilitza una doble barra (//) si el comentari té una única línia o delimitat per /* i */ si n'ocupa més d'una. Vegem de nou l'exemple anterior amb comentaris. L'execució serà exactament igual que abans:

```
# include <stdio.h>
/* la línia anterior ha d'estar sempre que fem operacions de
lectura o escriptura. Tot i això, hi ha compiladors que no ho
necessiten*/

main ()
```

```
{ //el codi del programa anirà entre les dues claus
     printf("Hola"); //això escriu Hola per pantalla
}
```

2. Els tipus de dades bàsics

2.1. Variables i constants

Quan confeccionem un programa necessitem d'uns elements que ens serveixen per emmagatzemar dades. En realitat aquests elements només són un espai reservat a la memòria de l'ordinador que es fa servir per guardar aquestes dades de forma temporal durant l'execució del programa.

Normalment aquests elements van canviant el seu valor al llarg de l'execució del programa, raó per la qual s'anomenen variables. Veurem que també podem definir elements que no puguin variar el seu valor (per exemple el nombre pi). Són les constants.

Tant una variable ve definida per dos trets:

- el seu nom (també dit identificador), que serveix per identificar-la: perquè sigui més fàcil fer referència a aquestes posicions de memòria els hi assignem un nom.

- el seu tipus, que descriu com és aquesta variable: d'aquesta manera es pot saber quant d'espai de memòria s'ha de reservar per emmagatzemar la variable o constant. Cal tenir en compte que no ocupen la mateixa memòria un caràcter que un número real, per posar un exemple.

A un programa abans d'emprar una variable hem de "declarar-la". Això vol dir que hem de definir prèviament els seus noms i tipus. Així el compilador, segons el tipus, sabrà l'espai necessari per emmagatzemar-la. Si utilitzen una variable sense haver-la declarada el compilador ens donarà un error. La forma de la declaració en C és:

```
tipus nom
```

Vegem un exemple:

```
#include <stdio.h>

main()
{
      int x;        //declaració de la variable x de tipus enter
      x=3;          //li donem el valor 3
      printf("El valor de la variable c es %d", x);
            //escribim el seu valor per pantalla
}
```

En l'exemple anterior s'ha declarat una variable x de tipus int (nombre enter) amb la línia `int x;` Després a aquesta variable se li dóna el valor 3 i s'escriu un missatge per pantalla. Veureu que és habitual posar la declaració de la variable i la inicialització (donar-li un valor inicial) en la mateixa instrucció.

```
int x = 3; //és el mateix que int x; x=3; que teníem abans
```

Quan tenim més d'una variable del mateix tipus, les podem declarar en la mateixa línia, separades per comes. Per exemple, x, y i z són enters:

```
int x,y,z;    //declara tres variables (x, y, z) de tipus enter
```

No obstant, hi ha alguns llenguatges on aquesta declaració prèvia no és necessària, i el tipus i en conseqüència l'espai de memòria es determina quan les variables es van utilitzant per primer cop. Així ho fa per exemple JavaScript o Visual Basic (tot i que es pot forçar a fer la declaració).

Com hem comentat abans podem declarar també constants, és a dir un valor que al contrari que les variables no variarà al llarg de l'execució del programa. Per exemple, si volem fer certs càlculs matemàtics ens pot interessar declarar una constant pi, que prendrà el valor 3.1416, i que no canviarà mai. Això en C es fa afegint la paraula reservada `const` a davant de la declaració i assignant-li el valor corresponent, com en l'exemple següent:

```
#include <stdio.h>

main()
{
      const float pi=3.1416;  //declarem la constant pi
      printf("El valor de la constant pi es %f", pi);
}
```

En l'exemple hem declarat una constant anomenada pi de tipus nombre real que pren el valor 3.1415927 i que no podrà variar al llarg del programa.

Hi ha una segona manera diferent de declarar una constant. És utilitzant la directiva del processador `#define`. En aquest cas no cal dir explícitament el tipus i la declaració va fóra del `main`. Internament, el pre-processador substituirà totes les aparicions de la constant en el codi pel seu valor abans de compilar.

```
#include <stdio.h>
#define pi 3.1416; //declarem la constant pi

main()
{
      printf("El valor de la constant pi es %f", pi);
}
```

Cada llenguatge pot tenir unes restriccions a l'hora de triar el nom d'una variable o constant. En el cas concret de C, tenim que:

- No poden haver dos elements (variables o constants en el nostre cas) amb el mateix nom. En posterior capítols tornarem a parlar d'un cas especial que és quan es defineixen funcions.

- El nom no pot ser cap paraula reservada del llenguatge. Les paraules reservades són les que el propi llenguatge fa servir per definir la seva gramàtica. Per exemple, no podem tenir una variable que es digui `int`, ja que `int` ja té un significat propi en C.

- No pot contenir caràcters "especials". Només pot tenir lletres, nombres i guions baixos (_), sempre que el primer sigui una lletra o un guió baix (no un nombre.

- Es fa distinció entre majúscules i minúscules (és el que s'anomena *case-sensitive*). Per tant, no hi ha cap problema en tenir una variable `A` i una altra `a`.

2.2. Els diferents tipus

N'hi ha diversos tipus diferents i fins i tot el propi programador en pot a arribar a crear de nous (en posteriors capítols veurem com), però tots parteixen dels tipus de dades bàsics, que en C són caràcters i nombres. En altres llenguatges existeix el que s'anomena tipus lògic o booleà, les variables del qual poden prendre només els valors vertader o fals. En C no existeix aquest tipus.

2.2.1. Tipus numèrics

Per representar els nombres hem de distingir dos tipus numèrics diferents, segons si volem representar nombres enters o nombres reals.

En principi un nombre enter pot estar entre els valors –32768 i +32767. Això és perquè per al seu emmagatzematge es fan servir 2 bytes (16 bits). Noteu que el primer bit es reserva per al signe i aleshores 2^{15} és igual a 32767. En C aquest tipus rep el nom d'`int`.

Quant als reals, es fan servir per representar els nombres amb decimals, ja sigui en notació ordinària o exponencial. En C hi ha dos tipus per nombres reals, el `float` que permet representar valors entre $3.4x10^{-38}$ i $3.4x10^{38}$ (positius i negatius) i el double, que té més precisió (el doble) i que permet representar valors entre $1.7x10^{-308}$ i $1.7x10^{308}$ (també positius i negatius).

És important notar que aquests valors són orientatius, ja que poden variar segons la plataforma. Per exemple, hi ha compiladors en certes plataformes que reserven 4 bytes per als enters, amb la qual cosa es poden representar més nombres.

Tant en enters com en reals, es poden afegir uns prefixos que permeten afinar més el tipus. Són:

- `short`
- `long`
- `unsigned`

De nou, variarà segons la plataforma la precisió dels tipus amb aquests prefixos, però bàsicament `unsigned` vol dir que només es consideren valors positius i per tant el bit de signe es pot utilitzar per donar més rang de valors, mentre que el `short` i `long`

reserven un espai menor o major respectivament, i per tant tenen un rang de valors menor per al `short` i major per al `long`.

Vegem a continuació un petit programa que fa servir el tipus `float` per calcular l'àrea d'una circumferència:

```
#include <stdio.h>

main()
{
      const float pi=3.1416;
      float radi;
      float area;

      printf("Introdueix el valor del radi: ");
      scanf("%f",&radi); //llegim el radi introduït per l'usuari

      area = pi * radi * radi; //calculem l'àrea

      printf("El valor de l'area es: %f",area);
            //escrivim el resultat
}
```

2.2.2. Caràcters

Emmagatzema un caràcter alfanumèric, és a dir no només les lletres sinó també els dígits numèrics, signes de puntuació i demés signes especials com $, %, & o d'altres.

Una variable o constant de tipus caràcter només tindrà per valor un d'aquests caràcters i no més, i per diferenciar-lo del nom d'una variable, aquest caràcter es posa entre cometes simples ('caràcter'). Vegem un exemple:

```
#include <stdio.h>

main()
{
      char c,d;   //declarem c i d com a caràcters

      c = 'A'; // Això fa que c contingui una A majúscula
      d = c; /* Això fa que d prengui el valor que
                  emmagatzema c, és a dir una A majúscula */
}
```

Noteu que una lletra en majúscula i en minúscula són dos valors diferents, és a dir que els caràcters `'a'` i `'A'` són diferents.

En realitat allò que es guarda a la memòria és un codi numèric que representa el caràcter. Així doncs, cada caràcter té assignat un número entre 0 i 255. Aquest número rep el nom de codi ASCII. Noteu que per representar un número entre 0 i 255 es necessiten 8 bits, és a dir un byte. Per tant, un caràcter ocupa un byte de memòria.

En realitat la taula ASCII original només contenia 128 caràcters, mentre que és l'ASCII estès el que conté 256. Moltes vegades apareixen problemes amb els caràcters "estesos", on són tots els caràcters accentuats, així com les lletres ç i ñ.

Per representar un caràcter també és pot fer directament amb una barra invertida seguida del seu codi en octal o amb una x i el seu codi en hexadecimal. Per exemple, el caràcter 'A' també es pot representar com '\101' o com '\x41'.

A més, hi ha uns caràcters especials com poden ser un tabulador o el que fa que una línia acabi. Es representen sempre amb una barra invertida i un caràcter. Són aquests:

\b	retrocés *(backspace)*
\t	tabulador
\n	nova línia
\"	cometes dobles
\'	cometa simple
\\	barra invertida

Per exemple, recuperant el primer exemple d'escriure "Hola", si volem que a continuació es passi a la següent línia es faria posant el caràcter de nova línia (\n):

```
printf("Hola\n");
```

Veurem en capítols posteriors com sovint, en lloc d'utilitzar el tipus `char`, es fa ús del tipus `int` (dels valors 0 a 255) per a representar caràcters.

2.2.2. Booleans

El tipus booleà existeix a la majoria de llenguatges i permeten representar els valors lògics, és a dir serveixen per representar quan una condició és vertadera o falsa. Per tant, una variable booleana pot tenir únicament dos valors: vertader o fals.

No obstant, en C no existeix aquest tipus i per a tal efecte s'utilitzen els enters, de forma que 0 indica el valor fals i qualsevol cosa diferent de 0 indica el valor vertader.

Al capítol següent veurem en més detall la lògica booleana i com s'analitzen les condicions.

3. Expressions, sentències i estructures de control

3.1. Les expressions

Una expressió està formada per operadors i operants i la seva avaluació ens dóna un valor concret. Els operants poden ser variables, constants o per exemple nombres o caràcters concrets. També una expressió (li diríem subexpressió) pot ser un operant amb la finalitat de formar altra expressió. Per exemple, l'expressió `a + (b*c)` té un operador + amb dos operants, la variable a i la subexpressió `(b*2)`, que a la vegada té un operador * amb dos operants, que són la variable `b` i el valor constant `2`. Noteu que `a`, `b` i `c`, també podrien ser constants. Una variable, constant o literal també formen per sí soles una expressió. Un literal és un valor concret, per exemple 5 o ‘A’.

Hi ha diferents tipus d'operadors:

Aritmètics:

Operador	Operació que fa
+	Suma
-	Resta
*	Multiplicació
/	Divisió
%	Resta de la divisió entera (mòdul)

Hi ha llenguatges, com Pascal, que tenen dos operadors per la divisió, un per la divisió real i un altre per la divisió entera (sense decimals).

Lògics:

Operador	Operació que fa
|| (dos signes |)	O lògica
&&	I lògica
!	Negació lògica

Vegeu el següent apartat sobre lògica booleana.

Comparatius o relacionals:

Operador	Operació que fa
== (dos signes =)	Igualtat
!=	Desigualtat
<	Menor que ...
<=	Menor o igual que ...
>	Major que ...
>=	Major o igual que ...

També existeixen operadors a nivell de bits, però no ens interessen en aquesta assignatura.

3.1.1 Lògica booleana

La lògica booleana es fa servir per construir condicions compostes. Per exemple la condició `(x < 5)` pot ser falsa o vertadera, però ens pot interessar construir condicions o predicats compostos a partir de vàries condicions simples. Per fer això utilitzem els operadors lògics que son: **o**, **i** i **no** (negació). Recordeu que en C no existeix el tipus booleà, i que 0 indica fals, mentre que qualsevol enter distint de 0 indica vertader. Vegem com funciona cada un d'ells:

1. O (*or*): ||

Donades dues condicions A i B, per què la condició (A o B) sigui vertadera ha d'ocórrer que almenys alguna de les dues sigui certa. És a dir que (A o B) serà certa si A és certa, si B és certa així com també si les dues son certes. En canvi serà falsa si les dues són falses. Vegem-m'ho representat amb una taula, que rep el nom de taula de veritat:

A	**B**	**A o B**
V	V	V
V	F	V
F	V	V
F	F	F

2. I (*and*): &&

Donades dues condicions A i B, per què la condició (A i B) sigui vertadera ha d'ocórrer que les dues siguin certes al mateix temps. És a dir que (A i B) serà certa si tant A com B són certes, però serà fals si qualsevol de les dues (o bé les dues) són falses. Vegem la taula de veritat:

A	**B**	**A i B**
V	V	V
V	F	F
F	V	F
F	F	F

3. No (*not*): !

La negació el que fa és canviar el valor d'una condició, és a dir si abans era vertadera aquesta passa a ser falsa i viceversa. Vegem la taula de veritat:

A	**no A**
V	F
F	V

3.2. Sentències o instruccions

Una sentència o instrucció és una ordre per donar un valor a una variable (típicament després de fer alguna operació matemàtica o lògica), o bé fer una operació d'entrada o sortida, és a dir, llegir el valor d'una variable o escriure el valor d'una expressió. Els programes estan formats per una sèrie d'instruccions que s'executen una darrera l'altra. En C, totes les instruccions acaben amb un punt i coma (;).

3.2.1. Assignacions

Assigna el valor de l'expressió del costat de la dreta a la variable de l'esquerra. Es representa amb el signe de =. Vegem alguns exemples d'assignacions:

```
a = 5 + 3;  //assignem a la variable a el valor 8
b = a;      //assignem a b el valor de a, és a dir, 8
```

En realitat, en C = és també un operador, l'operador d'assignació. Això vol dir, tot i que no ho farem servir en aquesta assignatura, que es pot utilitzar per formar expressions més llargues.

Cal anar amb compte de no confondre l'operador d'igualtat (==) amb l'assignació (=) ja que en moltes situacions el compilador no ens avisa de l'error i és una font habitual de problemes.

Operadors d'increment i decrement

En el llenguatge C existeixen uns operadors que permeten incrementar o decrementar en un el valor d'una variable de tipus enter. Són `++` i `--` respectivament. En realitat no és més que escriure-ho d'una forma abreujada. Així, la instrucció

```
x++;
```

és equivalent a:

```
x=x+1;
```

I igualment

```
x--;
```

és equivalent a:

```
x=x-1;
```

Aquests operadors es poden emprar en expressions més llargues. En aquest cas, cal destacar que aquests operadors poden anar abans o després de la variable amb efectes diferents. Vegem un cas:

```
y=x++;
```

Aquí primer es farà l'assignació i després d'increment. Per tant, si la x val inicialment 5, la y acabarà valent 5 i la x 6. En canvi, amb:

```
y=++x;
```

Aquí primer es farà l'increment i després l'assignació. Per tant, si la x val inicialment 5, tant la y com la x acabaran valent 6. El mateix s'aplica a l'operador --.

A més d'aquests operadors hi ha altres formes d'abreujar assignacions. Per exemple

```
x=x+5;
```

es pot escriure com

```
x+=5;
```

Aquesta forma reduïda és aplicable a qualsevol operador. Per exemple `x=x*3;` es pot escriure com `x*=3;`

3.2.1. Operacions d'entrada i sortida

Lectura

```
scanf("%format",&variable);
```

Aquesta instrucció llegeix i guarda el valor que introdueix l'usuari per teclat a la variable `variable`. El format indica com s'han de llegir les dades d'entrada. Per exemple, si volem llegir un nombre enter x, s'usa el caràcter d per especificar-ho. Així:

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

No oblideu posar el caràcter & abans del nom de la variable, perquè és un error típic. Altres formats són:

- d: un nombre enter en notació decimal
- o: un nombre enter en notació octal
- x: un nombre enter en notació hexadecimal
- c: un caràcter
- s: una tira de caràcters (per exemple, "hola"). En parlarem en capítols posteriors.
- f: un nombre real (tipus float)

Per utilitzar la instrucció `scanf`, cal posar la línia al principi del fitxer:

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

Això permet utilitzar totes les funcions estàndards per llegir i escriure que té C.

Escriptura

```
printf("%format",expressió);
```

Aquesta instrucció escriu el valor de l'expressió per pantalla. Per exemple, si volem escriure el valor d'un nombre enter x, s'usa el caràcter d per especificar el format. Així:

```
printf("%d",x);
```

Vegeu que ara la variable no porta la &. La raó d'això l'entendreu en temes posteriors.

Altres formats són:

- d: un nombre enter en notació decimal
- o: un nombre enter en notació octal
- x: un nombre enter en notació hexadecimal
- c: un caràcter
- s: una tira de caràcters (per exemple, "hola"). En parlarem en capítols posteriors.
- f: un nombre real (tipus float o double) en notació decimal
- e: un nombre real (tipus float o double) en notació exponencial (p.e. 3.1E-2)

Si només volem escriure un text, ho fem com en l'exemple inicial del curs:

```
printf("hola");
```

Es poden escriure més d'una variable a la vegada, i podem també intercalar text i valors. Per exemple:

```
printf("L'area de la cirumferencia de radi %f es %f",radi,area);
```

Per utilitzar la instrucció `printf` també cal incloure la línia:

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

3.2.1. L'ordre de precedència dels operadors

Sovint a una mateixa expressió o sentència s'empra més d'un operador. Per exemple podem escriure coses com:

```
c = a + b / c * d;
... a>5 && b<3 || c>0 ...
```

En aquests casos el resultat és molt diferent depenent de l'ordre en que executem les operacions. És per això que cal establir una regla de precedència o prioritat dels operadors, que ara discutirem.

Aquesta taula resumeix les prioritats que marquen en quin ordre executem.

Associativitat	Operadors
-->	( ) [ ] . ->
<--	! ~ ++ -- (cast) *[1] &[2] sizeof
-->	* / %
-->	+ -

-->	<< >>
-->	**< <= > >=**
-->	**== !=**
-->	&
-->	^
-->	|
-->	**&&**
-->	**||**
<--	?:
<--	**= += -= (i altres operadors d'assignació)**
-->	,

[1] Operador d'indirecció (contingut de)
[2] Operador de direcció (direcció de)

Alguns d'aquests operadors encara no els coneixem, però els anirem veient al llarg del curs. En negreta apareixen els que ja s'han introduït.

A partir d'aquesta taula s'apliquen les següents tres regles:

1. Els operadors de major prioritat (els de més amunt a la taula) s'executen abans dels de més baixa prioritat.

2. Els operadors que tenen la mateixa prioritat s'executen en ordre d'esquerra a dreta de l'expressió o de dreta a esquerra segons la fletxa de la columna associativitat.

3. Si una expressió conté subexpressions tancades entre parèntesis, aquestes s'avaluen primer (usant per tal efecte les regles 1 i 2). Si tenim parèntesis dins d'altres parèntesis, s'avaluarà primer la subexpressió més interna.

3.3. Les estructures de control

Les sentències o instruccions que hem vist fins ara (l'assignació, la lectura i l'escriptura) eren molt senzilles. Aquestes es poden executar seqüencialment, una darrera l'altre. Però per poder afrontar problemes amb un mínim de complexitat necessitem introduir el que anomenem les estructures de control que ens permeten definir una seqüència d'execució que no sigui només lineal. Així doncs, podem fer que en funció d'una certa condició, s'executin unes sentències o unes altres; parlarem aleshores d'estructures condicionals. O bé podem indicar que un determinat grup de sentències s'executin repetidament fins que una condició s'acompleixi (o es deixi d'acomplir); parlarem en aquests casos d'estructures iteratives. Resumint, allò que ambdues construccions ens permeten és predeterminar quin serà l'ordre d'execució de les instruccions d'un programa.

3.3.1. Les estructures condicionals

Utilitzem una estructura condicional quan hem d'especificar si una acció es fa depenent de si passa una certa condició. Això es representa de la següent manera:

```
if (condició)
{
      Conjunt d'accions a fer si la condició és certa;
}
```

Veureu que en C, les claus sempre emmarquen un bloc (també en altres estructures de control). Si només hi fos una instrucció, no cal posar-les. Les condicions en C sempre van entre parèntesis.

Vegem un exemple que es tracta de calcular el valor absolut d'un nombre sencer que l'usuari hagi introduït. En primer lloc hem de llegir aquest valor i després en cas de que el nombre sigui negatiu li hem de canviar de signe. Noteu que per canviar de signe el que hem de fer és multiplicar per -1. Després només queda escriure el valor.

```
#include <stdio.h>

main()
{
      int nombre;

      printf("Introdueix un nombre enter: ");
      scanf("%d",&nombre);
      if (nombre <0)
      {
            nombre = nombre * (-1);
            // això només s'executa si nombre és negatiu
      }
      printf("El seu valor absolut es: %d", nombre);
}
```

En aquest cas l'acció `nombre = nombre * (-1);` només s'executa en el cas de que nombre sigui menor que 0, és a dir de que sigui negatiu. Si per exemple, l'usuari introdueix el nombre 3, no s'executarà i s'imprimirà el valor 3. Per contra si el valor introduït és -5, aleshores sí que s'executarà aquesta línia, ja que nombre és menor que 0, i aleshores el que s'imprimeix és 5, positiu. Noteu que tot allò que hi ha després del final del if (després de }), s'executa sempre, tant si la condició és vertadera com si no.

També es pot especificar un grup d'accions a ser executades en el cas de que la condició sigui falsa. Es representa així:

```
if (condició)
{
      Conjunt d'accions a fer si la condició és certa;
}
else
{
      Conjunt d'accions a fer si la condició és falsa;
}
```

Vegem-ho amb un altre exemple, on s'ha de calcular la nota mitjana entre tres exàmens i després dir si l'alumne ha aprovat o no una assignatura. Ho podríem fer així:

```
#include <stdio.h>

main()
{
      float nota1,nota2,nota3,notafinal;

      printf("Introdueix la primera nota: ");
      scanf("%f",&nota1);
      printf("Introdueix la segona nota: ");
      scanf("%f",&nota2);
      printf("Introdueix la tercera nota: ");
      scanf("%f",&nota3);

      notafinal = (nota1+nota2+nota3)/3;
      if (notafinal>=5)
            printf("La nota final es APROVAT");
                  //només s'executa si la notafinal >= 5
      else
            printf("La nota final es SUSPENS");
                  //només s'executa si la notafinal < 5
}
```

Noteu que el programa s'executa seqüencialment fins a la instrucció `notafinal = (nota1 + nota2 + nota3) / 3;` . Aleshores, si `notafinal` és major o igual a 5, s'executa la instrucció `printf("La nota final es APROVAT");` En cas contrari, és a dir si és menor que 5, s'executa la instrucció `printf("La nota final es SUSPENS");`

Vegem un exemple de les dues seqüències d'execució:

<table>
<tr><th>Valors 6, 4 i 9</th><th>Valors 4, 5 i 3</th></tr>
<tr><td>printf("Introdueix la primera nota: ");
scanf("%f",¬a1);
printf("Introdueix la segona nota: ");
scanf("%f",¬a2);
printf("Introdueix la tercera nota: ");
scanf("%f",¬a3);

notafinal = (nota1 + nota2 + nota3)/3;

printf("La nota final es APROVAT");</td><td>printf("Introdueix la primera nota: ");
scanf("%f",¬a1);
printf("Introdueix la segona nota: ");
scanf("%f",¬a2);
printf("Introdueix la tercera nota: ");
scanf("%f",¬a3);

notafinal = (nota1 + nota2 + nota3)/3;

printf("La nota final es SUSPENS");</td></tr>
</table>

Aquests casos són força senzills i només ha estat una instrucció que ha estat executada dins de la estructura condicional, però el cas més habitual és que en siguin més. Fins i tot poden haver més estructures condicionals (o també iteratives) a dins.

Vegem un altre exemple senzill, on s'ha de dir quin és el major de entre dos nombres:

```
#include <stdio.h>

main()
{
      int n1,n2;

      printf("Introdueix el primer nombre enter: ");
      scanf("%d",&n1);
      printf("Introdueix el primer nombre enter: ");
      scanf("%d",&n2);

      if (n1>n2)
            printf("El major es %d",n1);
      else
            printf("El major es %d",n2);
}
```

En el següent programa s'introdueix el lloc en que un esportista ha finalitzat la seva prova en unes olimpíades i es determina quin premi li correspon, que pot ser medalla d'or, d'argent o de bronze, o un diploma olímpic si ha finalitzat entre el quart i el vuitè. Aquesta és la solució:

```
#include <stdio.h>

main()
{
      int lloc;

      printf("Introdueix el lloc on ha acabat l'esportista: ");
      scanf("%d",&lloc);
      if (lloc==1)
      {
            printf("Medalla d'or");
      }
      else
      {
            if (lloc==2)
            {
                  printf("Medalla d'argent");
            }
            else
            {
                  if (lloc==3)
                  {
                        printf("Medalla de bronze");
                  }
                  else
                  {
                        if ((lloc>3) && (lloc<9))
                        {
                              printf("Diploma olimpic");
                        }
                        else
                        {
                              printf("Cap premi");
                        }
```

```
                }
            }
        }
    }
```

Vegeu que totes les claus dels `if` i `else` es podrien eliminar. Com podem veure s'han de posar tot un seguit d'estructures `if-else` per a poder saber el resultat. És important saber que si no hi ha claus que indiquin el contrari, un else sempre fa referència al if més proper. Per exemple, en:

```
if (condició1) ...
if (condició2) ...
else ...
```

L'`else` fa referència al segon `if`, és a dir que s'executa si la `condició2` és falsa. Si ho volguéssim expressar de l'altra forma, és a dir que l'`else` s'hagués d'executar quan la `condició1` fos falsa (i per tant el segon `if` seria intern al primer), caldria emprar claus:

```
if (condició1)
{
      if (condició2) ...
}
else ...
```

Existeix una altra instrucció de tipus condicional, la selecció múltiple que ens permet expressar més fàcilment el mateix que un seguit d'estructures `if-else`, ja que permet prendre una decisió basada en un major nombre d'opcions (més que les dues de l'`if`). Es tracta de la instrucció `switch`. Vegem com seria el programa de les medalles utilitzant la selecció múltiple i després ja la estudiarem en més detall:

```
#include <stdio.h>

main()
{
      int lloc;

      printf("Introdueix el lloc on ha acabat l'esportista: ");
      scanf("%d",&lloc);
      switch (lloc)
      {
            case 1:
                  printf("Medalla d'or");
                  break;
            case 2:
                  printf("Medalla d'argent");
                  break;
            case 3:
                  printf("Medalla de bronze");
                  break;
            case 4:
            case 5:
            case 6:
            case 7:
            case 8:
                  printf("Diploma olimpic");
                  break;
```

```
            default:
                  printf("Cap premi");
                  break;
      }
}
```

Veiem que si `lloc` val 1, escriu `Medalla d'or`, si val 2 escriu `Medalla d'argent`, si val 3, `Medalla de bronze`, si val 4, 5 o 6 escriu `Diploma olímpic` i si no és cap de tots aquest valors escriu `Cap premi`; això fa, aleshores, exactament el mateix que l'anterior programa.

Així doncs, la sentència de selecció múltiple té aquesta forma:

```
switch (selector)
{
      case valor_1:
            accions de la primera opció;
            break;
      case valor_2:
            accions de la segona opció;
            break;
      ...
      default:
            accions de resta d'opcions;
            break;
}
```

On selector és una expressió de tipus caràcter o enter (no pot ser un real!). Els distints valors només s'expressen amb valors literals (és a dir lletres o nombres), però no és possible amb altres variables. Noteu que al final de les accions de cada opció hi ha un `break`. Si aquest `break` no hi és (com en els cas del 4, 5, 6 i 7) vol dir que es continua fins que se'n troba un i per això en qualsevol d'aquests casos acaba escrivint "Diploma olímpic" i després ja si surt del `switch` perquè troba el `break`. La clàusula `default` serveix per especificar què fer si la variable no té cap dels valors abans exposats.

3.3.2. Les estructures iteratives

En aquest cas el que volem és que una mateixa instrucció o grup d'instruccions es repeteixin un nombre determinat de vegades. Aquest nombre determinat de vegades en què el grup d'instrucció s'executa ve determinat per una condició: mentre aquesta condició és vertadera, es va repetint l'execució; fins que deixa de ser-ho. Es representa així:

```
while (condició)
{
      Conjunt d'accions a fer si la condició és certa;
}
```

El primer exemple es tracta de sumar tots els nombres des d'1 fins a N, on N pot ser un nombre sencer positiu introduït per l'usuari. No val fer la solució ràpida de dir que la

suma és `(N*(N-1))/2`, sinó que hem d'aplicar les estructures iteratives. A part de N, necessitarem dues variables de tipus enter:

- una per controlar la condició, és a dir que només es sumin els nombres fins a N i no més. Començant per zero, a cada iteració li incrementarem en 1 el seu valor i ens aturarem quan arribi a N. Li direm `passa`.

- l'altre que conté la suma parcial, des d'1 fins a la passa actual. Li direm `suma`.

```
#include <stdio.h>

main()
{
      int suma,passa,N;

      suma=0;
      passa=1;
      printf("Introdueix el nombre enter:");
      scanf("%d",&N);
      while (passa <= N)
      {
            suma = suma + passa;
            passa = passa + 1;
      }
      printf("La suma de 1 a %d val %d", N, suma);
}
```

Suposant que l'usuari introdueixi el valor N=5, vegem com van evolucionant les variables fins al final del programa:

	suma	passa
N=5	0	1
1<=5 ? : SÍ	0+1 = 1	2
2<=5 ? : SÍ	1+2 = 3	3
3<=5 ? : SÍ	3+3 = 6	4
4<=5 ? : SÍ	6+4 = 10	5
5<=5 ? : SÍ	10+5 = 15	6
6<=5 ? : NO: escriure(15)		

És molt important veure que abans de fer la comparació (passa<=N) hem hagut d'haver donat un valor a la variable passa (1 en el nostre cas). És el que se'n diu inicialitzar la variable. En cas contrari es produiria un error, ja que passa no té cap valor. Noteu que quan declarem una variable només reservem un espai a memòria, però encara no li donem cap valor.

Vegem un cas molt semblant però un poc més complicat, on es combinen les estructures iteratives amb les condicionals. El que ara farem és sumar d'1 fins a N, però només els nombres parells. Noteu que parell vol dir que és divisible per 2, o el que ens és útil a nosaltres, que la resta de dividir el nombre x entre dos val zero, és a dir que `x % 2 == 0`. El programa és molt semblant a l'anterior però abans d'incrementar el valor de la suma parcial (`suma = suma + passa;`), hem de comprovar si `passa` és parell o no:

```
#include <stdio.h>

main()
{
      int suma,passa,N;

      suma=0;
      passa=1;
      printf("Introdueix el nombre enter:");
      scanf("%d",&N);
      while (passa <= N)
      {
            if (passa % 2 == 0)
                  suma = suma + passa;
            passa = passa + 1;
      }
      printf("La suma dels parells de 1 a %d val %d", N, suma);
}
```

Si N torna a ser 5, l'execució és així:

	suma	passa
N=5	0	1
1<=5 ? : SÍ	1 parell ? NO: 0	2
2<=5 ? : SÍ	2 parell ? SI: 0+2=2	3
3<=5 ? : SÍ	3 parell ? NO: 2	4
4<=5 ? : SÍ	4 parell ? SI: 2+4=6	5
5<=5 ? : SÍ	5 parell ? NO: 6	6
6<=5 ? : NO: escriure(6)		

Noteu que tot i que això és correcte hi havia una altra manera de fer-ho. Ja que sabem que només hem de sumar els valor parells, el que podem fer és que passa valgui en principi 2 i en lloc d'incrementar-se d'un en un, ho faci de dos en dos, ja que només hem de sumar els valors parells:

```
#include <stdio.h>

main()
{
      int suma,passa,N;

      suma=0;
      passa=2;
      printf("Introdueix el nombre enter:");
      scanf("%d",&N);
      while (passa <= N)
      {
            suma = suma + passa;
            passa = passa + 2;
      }
      printf("La suma dels parells de 1 a %d val %d", N, suma);
}
```

	suma	passa
N=5	0	2
2<=5 ? : SÍ	0+2=2	4

4<=5 ? : SÍ	2+4=6	6
6<=5 ? : NO: escriure(6)		

Fixeu-vos que amb aquesta solució fem menys iteracions i a més a més ens estalviem totes les comparacions que fèiem per saber si el nombre era parell o no. Per tant aquest programa és més eficient que l'anterior.

Les estructures iteratives es poden expressar d'altres dues maneres diferents:

1) do ... while:

```
do
{
      Conjunt d'accions a fer mentre s'acompleixi la condició
} while (condició);
```

Noteu que a diferència de l'estructura anterior, ara la primera iteració es fa abans d'avaluar la condició, mentre que abans, el primer que es feia era avaluar la condició. Així doncs, les accions de dins del bucle sempre s'executaran almenys un cop.

Escriurem el primer programa, el sumador, utilitzant ara el do...while:

```
main()
{
      int suma,passa,N;
      passa=1;
      suma=0;
      printf("Introdueix el nombre enter:");
      scanf("%d",&N);
      do
      {
            suma = suma + passa;
            passa = passa +1;
      } while (passa<=N);
      printf("La suma de 1 a %d val %d", N, suma);
}
```

I vegem com evoluciona l'execució del programa:

suma	passa	
		N=5
0	1	1<=5 ? : SÍ
0+1 = 1	2	2<=5 ? : SÍ
1+2 = 3	3	3<=5 ? : SÍ
3+3 = 6	4	4<=5 ? : SÍ
6+4 = 10	5	5<=5 ? : SÍ
10+5 = 15	6	6<=5 ? : NO: escriure(15)

2) For:

```
for (variable=valor_inicial;condició;acció_a_cada_iteració)
{
      Conjunt d'accions;
}
```

Aquí, donada una variable de control que sempre és un nombre enter, se li assigna un `valor_inicial` i el bucle es repeteix mentre la `condició` sigui vertadera. En acabar cada iteració s'executa la `accio_a_cada_iteració`, que típicament suposa l'increment de la variable de control del bucle. Així, en el nostre exemple, veureu que `passa=passa+1;` ja no apareix dins del bucle, ja que està com l'`acció_a_cada_iteració`. Cal anar amb molt de compte si modifiquem la variable de control dins del bucle. Veureu també com passa s'inicialitza en arribar al bucle i no cal posar el `passa=1;` al començament.

```
#include <stdio.h>

main()
{
      int suma,passa,N;
      suma=0;
      printf("Introdueix el nombre enter:");
      scanf("%d",&N);
      for (passa=1;passa<=N;passa=passa+1)
            suma = suma + passa;
      printf("La suma de 1 a %d val %d", N, suma);
}
```

L'execució és així:

	suma	passa (s'incrementa automàticament a cada iteració)
N=5	0	1
1<=5 ? : SÍ	0+1 = 1	2
2<=5 ? : SÍ	1+2 = 3	3
3<=5 ? : SÍ	3+3 = 6	4
4<=5 ? : SÍ	6+4 = 10	5
5<=5 ? : SÍ	10+5 = 15	6
6<=5 ? : NO: escriure(15)		

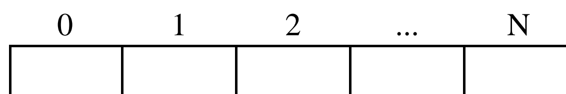
4. Tipus de dades composts estàtics: *arrays*, *strings* i *structs*

Els tipus de dades simples que hem estudiat fins ara representen valors simples, com ara nombres enters i reals o caràcters. No obstant això, molt sovint necessitarem agrupar una col·lecció de valors en un únic element. I és per això pel que es fan servir els tipus de dades composts: per agrupar en un únic element una col·lecció d'elements simples. Per exemple podem representar així una llista de preus, o una taula de temperatures, ...

Tant els *arrays* (o vectors o matrius) com els *strings* (o cadenes de caràcters) es poden manegar de forma dinàmica. No obstant en aquest tema només tractarem la forma estàtica, és a dir, indicant abans de compilar (en el codi) quin és el tamany de memòria necessari per a una variable d'aquests tipus.

4.1. Arrays *unidimensionals*

Un *array* unidimensional és un tipus de dades que representa un vector de n elements, cada un del mateix tipus. La traducció més acurada d'*array* en català seria vector. En castellà ho veureu traduït sovint com a *arreglo*.



En C tots els *arrays* comencen per la posició 0 (no així en altres llenguatges que permeten especificar tant el límit inferior com el superior).

Per exemple, podem crear un *array* de 12 posicions que contingui les temperatures mitjanes de cada mes durant un any:

0	1	2	3	4	5	6	7	8	9	10	11
12.8	13.7	14.9	15.5	16.0	17.1	19.3	20.1	19.0	17.2	15.4	13.1

Per declarar aquest *array* ho fem així:

```
float temperatures[12];
```

On:

- `temperatures` és el nom que li donem a l'*array*,
- `[12]` indica el tamany de l'*array*, és a dir que tindrà 12 posicions indexades del 0 a l'11. Recordeu que sempre la posició inicial serà 0.
- `float` indica el tipus de cada un dels valors de cada posició. En aquest cas es tracta de nombres reals (les temperatures poden tenir decimals).

En general, es declara:

```
Tipus nom_array[tamany];
```

L'*array* permet accedir a cada una de les posicions mitjançant un índex; ho fem indicant el nom de l'*array* seguit del nombre de posició entre claudàtors. Així doncs, per accedir a la temperatura del sisè mes: `temperatures[5]` (recordeu que comencem per zero!), que té el valor de 16.0. És important no intentar accedir a una posició fora del rang de l'*array*. Per exemple, si fem `temperatures[13]` el programa donaria un error quan s'executa i o bé s'aturaria o bé donaria una dada errònia que trobés a una altra posició de memòria.

Físicament és també un espai que es reserva a la memòria, evidentment més gran, que permet emmagatzemar més d'un valor (els que s'especifiquen quan es declara l'*array*) i recuperar el valor d'una posició concreta del vector, és a dir permet el que s'anomena un direccionament directe.

Una vegada vist això, analitzarem una sèrie d'exemples que fan servir *arrays*:

4.1.1. Escriure tots els elements d'un vector

Es tracta d'escriure per pantalla tots els elements consecutivament d'un *array*. Farem servir de nou l'*array* temperatures que abans havíem mostrat. L'únic que necessitem és un bucle que començant per 0 i acabant per 11, escrigui cada una de les dotze posicions de l'*array*, respectivament.

```
#include <stdio.h>

main()
{
      float temperatures[12];
      int index;

      /* aquí aniria la part d'omplir l'array */

      index=0;
      while (index<12)
      {
            printf("%f\n",temperatures[index]);
            index++;
      }
}
```

Noteu que també ho haguéssim pogut fer utilitzant qualsevol altre de les estructures iteratives, és a dir amb repetir o amb per a:

- Amb do ... while:

```
#include <stdio.h>

main()
{
      float temperatures[12];
      int index;

      /* aquí aniria la part d'omplir l'array */
```

```
        index=0;
        do {
              printf("%f\n",temperatures[index]);
              index++;
        } while (index<12);
}
```

- Amb for:

```
#include <stdio.h>

main()
{
        float temperatures[12];
        int index;

        /* aquí aniria la part d'omplir l'array */

        for (index=0;index<12;index++)
              printf("%f\n",temperatures[index]);
}
```

4.1.2. Calcular la mitjana de tots els elements d'un vector

Ara volem calcular la temperatura mitjana anual a partir del nostre *array* de temperatures mitjanes mensuals. El programa serà molt semblant a l'anterior: tindrem un bucle i la variable `index` que com a variable del bucle ens permetrà accedir a totes les posicions de l'*array*. Ara necessitarem també una altra variable que contingui la suma parcial de les posicions a cada iteració del bucle, i a la qual li direm `suma`. Vegem-ho:

```
#include <stdio.h>

main()
{
        float temperatures[12];
        int index;
        float suma;

        /* aquí aniria la part d'omplir l'array */

        index=0;
        suma=0;
        while (index<12)
        {
              suma=suma+temperatures[index];
              index++;
        }
        suma=suma/12; //dividim pel nombre de mesos
        printf("La suma de l'array es %f",suma);
}
```

4.1.3. Cercar l'element major d'un vector

Ara el que farem és cercar la temperatura mitjana màxima de l'any, és a dir, el mes més calorós. De nou haurem de recórrer tot el vector amb un bucle i amb una variable

emmagatzemarem la posició màxima de les examinades fins a la iteració present. A aquesta variable li direm `maxima`:

```
#include <stdio.h>

      main()
      {
            float temperatures[12];
            int index;
            float maxima;

            /* aquí aniria la part d'omplir l'array */

            index=0;
            maxima=0;
            while (index<12)
            {
                  if (maxima<temperatures[index])
                        maxima=temperatures[index];
                  index++;
            }
            printf("La maxima anual es %f",maxima);
      }
```

4.1.4. Cercar si un valor concret és a un vector

Ara el que farem és comprovar si una certa temperatura, per exemple 15.0, apareix com a mitjana d'algun mes concret. El que farem és un bucle, però diferent als anteriors, ja que noteu que si la primera posició ja conté el valor 15.0, no és necessari continuar mirant la resta de posicions. Per això emprarem una variable que guardi si el valor cercat ja ha estat trobat o encara no. A aquesta variable li direm `trobat`. També farem que sigui l'usuari qui digui el nombre a cercar (variable `valor_cercat`).

```
      #include <stdio.h>

      main()
      {
            float temperatures[12];
            int index,trobat;
            float valor_cercat;

            /* aquí aniria la part d'omplir l'array */

            index=0;
            trobat=0;
            printf("Introdueix el valor que vols cercar: ");
            scanf("%f",&valor_cercat);

            while ( (index<12) && (trobat==0) )
            {
                  if (temperatures[index]==valor_cercat)
                        trobat=1;
                  index++;
            }
            if (trobat==0)
                  printf("El valor no hi es a l'array");
            else
                  printf("El valor si es a l'array");
```

```
}
```

Estudiem la condició del bucle: sabem que es surt de la iteració quan la condició és falsa, i en cas contrari es continua la iteració. Com que és una i (*and*), la condició serà falsa quan una de les dues sigui falsa, és a dir que ja haguem arribat al final i `index` sigui igual que 12, o bé que `trobat` valgui 1, la qual cosa vol dir que ja hem trobat l'element que cercàvem. Noteu que una o (*or*) hagués estat incorrecta perquè, per exemple si no és trobés cap, no acabaria mai, ja que `trobat` sempre valdria 0.

4.1.5. Comptar quantes vegades apareix un valor concret a un vector

Aquest exemple és molt semblant als anteriors. De nou hem de recórrer el vector complet (no ens hem d'aturar si trobem el valor). Farem servir una variable que indica el nombre (sencer) d'ocasions en que aquest valor és present al vector; li direm `vegades`.

```
#include <stdio.h>

main()
{
      float temperatures[12];
      int index,vegades;
      float valor_cercat;

      /* aquí aniria la part d’omplir l’array */

      index=0;
      vegades=0;
      printf("Introdueix el valor que vols cercar: ");
      scanf("%f",&valor_cercat);

      while (index<12)
      {
            if (temperatures[index]==valor_cercat)
                  vegades++;
            index++;
      }
      printf("El valor %f s'ha trobat %d vegada/es\n",
                                    valor_cercat,vegades);
}
```

4.2. **Arrays** ***multidimensionals***

Fins ara hem parlat d'*arrays* unidimensionals, és a dir de vectors. Però també podem crear *arrays* de més dimensions. En tal cas el que tenim és la representació d'una matriu. Vegem com es declara una matriu de dues dimensions, N x M:

```
tipus nom_matriu[N][M];
```

Per accedir a un element concret d'una matriu ho fem utilitzant no només un índex com abans, sinó tants com dimensions tingui la matriu. Cada índex anirà entre claudàtors,

per exemple `nom_matriu[3][4]` fa referència a la quarta fila (fila 3), cinquena columna (columna 4) si ho pensem com una matriu típica de dues dimensions.

Cal tenir en compte que les matrius no només poden tenir elements numèrics, sinó que qualsevol altre tipus, ja sigui simple (caràcters, a més a més dels numèrics) o compost.

Veurem ara uns exemples que ens ajudaran a comprendre el funcionament de les matrius. Per a que sigui més fàcil d'entendre ho farem amb matrius bidimensionals i així l'alumne podrà representar-ho en un paper. Però cal tenir present que poden haver matrius de tantes dimensions com es vulgui.

4.2.1. Escriure tots els elements d'una matriu

Tenim una matriu 3x4 de sencers i volem escriure un a un tots els seus elements. Ho farem de dues maneres diferents, bé ordenant per files o bé ordenant per columnes.

```
#include <stdio.h>

main()
{
      int m[3][4];
      int i,j;

      /* aquí aniria la part d'omplir la matriu */

      for (i=0;i<3;i++)
            for (j=0;j<4;j++)
                  printf(" - %d - ",m[i][j]);
}
```

L'ordre d'impressió d'aquest algorisme és:

```
M[0][0] - M[0,1] - M[0,2] - M[0,3] - M[1,0] - M[1,1] - M[1,2] -
M[1,3] - M[2,0] - M[2,1] - M[2,2] - M[2,3]
```

És a dir, que primer escriu tota la primera fila, després tota la segona i per acabar la tercera. Si canviem l'ordre dels bucles, podrem fer que ho imprimeixi ordenant per columnes:

```
#include <stdio.h>

main()
{
      int m[3][4];
      int i,j;

      /* aquí aniria la part d'omplir la matriu */

      for (j=0;j<4;j++)
            for (i=0;i<3;i++)
                  printf(" - %d - ",m[i][j]);
}
```

L'ordre d'impressió ara és:

```
M[0,0] - M[1,0] - M[2,0] - M[0,1] - M[1,1] - M[2,1] - M[0,2] -
M[1,2] - M[2,2] - M[0,3] - M[1,3] - M[2,3]
```

4.2.2. Sumar dues matrius

El que ara farem és sumar dues matrius de la mateixa dimensió, per exemple també de 3x4. Les matrius origen són A i B, i el resultat ho guardarem a C.

```
#include <stdio.h>

main()
{
      int a[3][4],b[3][4],c[3][4];
      int i,j;

      /* aquí aniria la part d'omplir la matriu */

      for (i=0;i<3;i++)
            for (j=0;j<4;j++)
                  c[i][j]=a[i][j]+b[i][j];
}
```

4.2.3. Multiplicar dues matrius

Ara multiplicarem les matrius A(MxP), B(PxN) i el resultat serà la matriu C(MxN). Com sabeu, la posició (i,j) de la matriu resultant C es calcula sumant el producte de la primera posició de la fila i de la matriu A per la primera posició de la columna j de la matriu B, més la segona posició de la fila i d'A per la segona posició de la columna j de B, i així successivament fins a P-1 (perquè comencem per zero!). En resum, que la posició C[i,j] és la suma de tots els productes C[i,k]*B[k,j], on k va des de 0 fins a P-1. Noteu que cal omplir la matriu C amb zeros abans d'operar.

```
#include <stdio.h>

main()
{
      int a[M][P],b[P][N],c[M][N];
      int i,j,k;

      /* aquí aniria la part d'omplir les matrius A i B */

      for (i=0;i<M;i++)
            for (j=0;j<N;j++)
                  c[i][j]=0;  //omplim C amb zeros

      for (i=0;i<M;i++)
            for (j=0;j<N;j++)
                  for (k=0;k<P;k++)
                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
```

Si voleu executar aquest codi definiu primer M, N i P com a constants i doneu-los valors.

4.3. Strings *o cadenes de caràcters*

Un *string* és una cadena de caràcters. En realitat és molt semblant a un array unidimensional de caràcters. La diferència principal és que una cadena de caràcters sempre ha d'acabar amb el caràcter '\0'. C té una llibreria `<string.h>` amb unes funcions que ens permeten treballar amb les cadenes en bloc, i no només caràcter a caràcter, com ens passaria amb els arrays.

Per declarar un string ho fem igual que si fos un array de caràcters. Per exemple declarem la variable cadena com una cadena de caràcters de fins a 5 caràcters com a màxim:

```
char cadena[5];
```

Recordeu que sempre hi ha d'haver el caràcter '\0' per indicar el final de la cadena, així que amb aquesta declaració, cadena només podrà contenir 4 caràcters com a màxim, més aquest caràcter '\0'.

Quan fem la declaració ja podem donar-li un valor inicial a la cadena. Per exemple:

```
char cadena1[5]="hola";
char cadena2[5]="ab";
```

Recordeu que un caràcter està delimitat per comentes simples (per exemple 'a') i una cadena per cometes dobles (per exemple "hola"). Noteu que internament `cadena1` conté els caràcters:

0	1	2	3	4
h	o	l	a	\0

I `cadena2`:

0	1	2	3	4
a	b	\0		

I que, per tant, la següent instrucció ens donaria un error de desbordament dels límits de l'array:

```
char cadena1[5]="abcde";
```

Tret de la declaració inicial només podem accedir a la cadena caràcter a caràcter, si no és amb les funcions de `<string.h>`. És a dir, que no podem fer el següent:

```
char cadena1[5];
cadena1="hola";
```

Així doncs, per assignar la cadena “hola” a `cadena1`, ho hauríem de fer amb la funció `strcpy` (*string copy*), que està definida a `<string.h>`. Així:

```
strcpy(cadena1,"hola");
```

Com que estem utilitzant una funció que està a la llibreria <string.h>, cal posar la següent línia al principi del fitxer:

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

Vegem un exemple:

```
#include <string.h>
main()
{
      char a[5],b[5],c[5];
      strcpy(a,"ab");
      strcpy(b,"cd");

      strcpy(c,a);
      strcat(c,b);

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

El resultat que s’imprimirà és “abcd”.

A continuació teniu una llista de funcions definides en <string.h> útils per treballar amb cadenes de caràcters. En realitat n’hi ha més, però encara no les veurem.

Funció	**Explicació**
`strlen(cadena)`	Retorna la longitud de la cadena
`strcpy(cadena1,cadena2)`	Copia el contingut de la cadena2 a la cadena1
`strncpy(cadena1,cadena2,n)`	Copia els n primers caràcters de la cadena2 a la cadena1
`strcat(cadena1,cadena2)`	Concatena cadena2 al final de cadena1
`strncat(cadena1,cadena2,n)`	Concatena els n primers caràcters de cadena2 al final de cadena1
`strcmp(cadena1,cadena2)`	Compara les dues cadenes i ens retorna 0 si són iguals, un enter <0 si cadena1 és alfabèticament menor que cadena2, i un enter >0 altrament
`strncmp(cadena1,cadena2,n)`	Igual que strcmp, però utilitzant només els n primers caràcters de les dues cadenes

A més d’aquestes funcions, podem llegir i escriure cadenes completes amb `scanf` i `printf` (recordeu que haureu d’afegir la línia `#include <string.h>`). El control per a strings és `%s`. Però en el cas del scanf no hem de posar el & a davant del nom de la cadena. Ho faríem així:

```
scanf("%s",cadena);     //llegeix una cadena escrita per l'usuari
```

```
printf("%s",cadena);    //imprimeix una cadena per pantalla
```

*4.4. Estructures (*structs)

Hem vist que els *arrays* i les cadena de caràcters ens permeten agrupar diferents elements d'un mateix tipus (caràcters en el segon cas). Però sovint les dades que volem agrupar no tenen el mateix tipus. És aleshores quan utilitzem el que anomenem estructures o registres, que són un tipus de dades compost per elements diferents tipus de dades. És a dir, les estructures ens permeten estructurar diferents dades en una única "peça", per tal de que després sigui més fàcil i intuïtiva la seva manipulació.

Per exemple, anem a declarar una variable que serà una estructura i que anomenarem `usuari` i que tindrà un codi numèric més un tipus d'usuari, que serà un caràcter que indicarà si l'usuari és un estudiant ('E'), professor ('P') o personal d'administració ('S'). Ho farem així:

```
struct {
      int codi;
      char tipus_usuari;
} usuari;
```

Així es reservarà per a la variable usuari l'espai d'un enter més el d'un caràcter. Un altre exemple és el codi d'un vol, on s'identifica amb dues lletres la companyia aèrea i amb un enter el nombre de vol (per exemple "IB1717"). La declaració de la variable `codi_vol` seria així:

```
struct {
      char companyia[2];
      int nombre;
} codi_vol;
```

Cada un dels components de l'estructura s'anomenen camps. No és obligatori que tots els camps siguin de tipus diferents. Per exemple, ens pot interessar declarar una variable per a representar un color en RGB (valors de vermell, verd i blau respectivament). Ho faríem així:

```
struct {
      short int R;
      short int G;
      short int B;
} color;
```

A una estructura no se li pot assignar un valor de cop, sinó que sempre es fa camp a camp. La manera d'accedir a un camp de l'estructura és:

```
nom_estructura.nom_camp
```

Per exemple:

```
color.R=255;
color.G=0;
color.B=100;
```

En posteriors temes veurem com el programador pot definir els seus propis tipus. Quan es treballa amb estructures el més habitual és definir un tipus per a l'estructura i declarar les variables a partir del nou tipus.

5. Els punters

Fins ara totes les estructures de dades que hem vist eren estructures estàtiques, és a dir que quan escrivim el programa i el compilem ja sabem l'espai de memòria que s'ha de reservar per emmagatzemar-les. Ara bé, també és possible crear estructures dinàmiques, la grandària de les quals no és coneguda en temps de compilació, sinó que va variant durant l'execució del programa. Aprofundirem aquest concepte en el tema de llistes seqüencials.

La base per definir estructures dinàmiques són els punters o apuntadors. Un punter o apuntador és un tipus de dades que el seu valor és una adreça de memòria, és a dir que "apunta" a una posició de memòria. C és un llenguatge que fa un ús molt intens dels punters.

5.1. Declaració d'un punter

Quan es declara un punter cal dir a quin tipus de dades hi apuntarà. Així doncs, podem definir punters a enters, a caràcters, a estructures,... Això vol dir que l'adreça que guarda el punter (a la que "apunta") només hi pot haver el tipus que s'ha declarat. Per declarar un punter es fa així:

```
tipus *punter;
```

Per exemple:

```
int *pi;
char *pc;
struct {
      short int R;
      short int G;
      short int B;
} *pcolor;
```

Però això només hi reserva l'espai per guardar una adreça (i diu a quin tipus apunta), però no li dóna cap valor. En la secció 6.3 veurem com es fa per donar un valor a un punter.

5.2. Indirecció

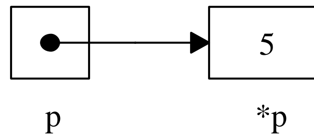
En aquest apartat veurem quina és la manera per accedir a l'adreça allà on apunta un punter: amb l'operador *, que rep el nom d'indirecció. Així, si tenim la declaració:

```
int *p;
```

`p` és del tipus punter (una adreça), mentre que `*p` és del tipus enter. Així, si ja haguéssim donat un valor a p (ho veurem a la secció 6.3), podríem fer:

```
*p = 5;
```

ja que com hem dit `*p` és un enter. Vegem-ho gràficament:



El que no podeu escriure és:

```
p = 5;
```

En altres llenguatges això un donaria un error de compilació, ja que p és una adreça i 5 un enter. Però C no, ja que si escrivim això entén que p ha d'apuntar a la posició 5 de la memòria, que evidentment no és del nostre programa (potser hi és el sistema operatiu). Així, quan després intentem accedir amb `*p` ens donarà un error en temps d'execució del tipus "violació d'accés", ja que estem accedint a una posició que ens està prohibida. Però noteu que és un error en temps d'execució i no de compilació i recordeu que els errors en temps d'execució sempre són més difícils de depurar que els que ens dóna el compilador.

5.3. Assignació de valors a un punter

Per donar-li un valor a un punter es pot fer de diverses maneres, depenent del cas:

1) Podem fer que apunti a una variable ja existent, és a dir, que el valor del punter serà l'adreça on està aquella variable. Hi ha un operador per obtenir l'adreça d'una variable que és &, anomenat operador de direcció. Per exemple:

```
int *p; //p és punter a int
int x;

x=3;
p=&x; //p apunta a l'adreça on es troba la variable x
```

I així tenim que p apunta a x, i per tant els enter *p i x són en realitat el mateix nombre. Si fem:

```
*p = 5;
printf("%d",x);
```

el valor que s'escriu és 5 i no 3.

2) Podem fer que apunti a un altre punter que ja tingui una adreça assignada. Per exemple:

```
int *p,*q; //p i q són tots dos punters a int
int x;

x=3;
```

```
p=&x;
q=p; //q apunta a la mateixa adreça que p
```

Així, p apunta a la mateixa adreça a la qual hi apunta p, que en el nostre cas és allà on està la variable x. Noteu que igual que abans, si modifiquem *q, en realitat estem modificant també *p i x.

```
*q = 5;
printf("%d",x);
printf("%d",*p);
```

En els dos casos imprimeix 5.

És molt important que vegeu que no és el mateix la instrucció

```
q=p;
```

que

```
*q=*p;
```

La primera fa que q apunti al mateix lloc que p, la segona que el valor enter d'allà on q està apuntant val el mateix del que p està apuntant, però l'adreça no canvia.

Vegem-ho amb dos exemples:

```
int *p,*q;                    int *p,*q;
int x,y;                      int x,y;


x=3;                          x=3;
y=5;                          y=5;
p=&x;                         p=&x;
q=&y;                         q=&y;
q=p;                          *q=*p;
printf("%d",y);               printf("%d",y);
```

Vegeu que el resultat que en el primer cas s'escriu un 5 i en el segon un 3.

3) Reservant dinàmicament espai a la memòria per a un nou element del tipus del punter. Això en C es fa amb la funció `malloc`. A aquesta funció li hem de dir quin espai cal reservar en bytes. Per exemple si tenim un punter a caràcters, seria un byte. Però com que podem haver diferències en el nombre de bytes que utilitza cada plataforma per a un tipus de dades, tenim un operador `sizeof` que calcula el tamany necessari en la plataforma on s'executa el programa. Així, `sizeof(int)` valdrà 2 si en la plataforma actual un enter es guarda amb 16 bits. A partir d'això, si volguéssim reservar l'espai per a un enter ho faríem amb `malloc(sizeof(int))` . Vegem un exemple:

```
int *p;
p=(int *)malloc(sizeof(int));
      // reserva l'espai per a un enter i fa que p hi apunti
*p=3; // escriu un 3 a l'espai que hem reservat (i p hi apunta)
```

Ja vegeu que abans del `malloc` hem afegit `(int *)`. Això és el que se'n diu un cast, i en parlarem en capítols posteriors. De moment, heu de saber que sempre heu de posar `(tipus *)(malloc(sizeof(tipus));`

Un cop que ja no hem d'utilitzar més el punter, ens hem d'encarregar nosaltres mateixos de tornar a alliberar la memòria que hem reservat. En cas contrari, aquesta memòria no podrà ser usada per altres programes, encara que el nostre programa hagi acabat. Per fer-ho, hem d'emprar la funció `free`, que allibera la memòria on el punter apunta i deixa al punter sense cap valor. En el nostre exemple anterior hauríem d'afegir la línia quan haguem acabat d'emprar `p`:

```
free(p);
```

Veurem més aspectes de gestió de memòria en temes posteriors. Entre d'altres coses entendrem millor com funciona el `malloc` i com comprovar que hi hagi suficient memòria per la reserva que volem fer (potser la memòria és plena i no hi cap).

Les funcions `free` i `malloc` estàn definides a la llibreria `stdlib.h`. Per tant, cal posar sempre que les utilitzem la línia:

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

5.4. El punter nul

Hi ha un valor especial que és NULL que podem assignar a tot punter de qualsevol tipus. NULL indica que el punter no apunta a cap adreça, però que ja ha estat inicialitzat. Recordeu que quan declarem un punter encara no té cap valor; si li assignem el valor NULL, encara no apunta enlloc però ja sí que té un valor. Així mai no es produiran errors quan mirem el valor del punter. Per exemple podem tenir:

```
int *p,*q;
p=NULL;
q=NULL;
...
if (p==q) ...
```

Vegeu que en cap cas no es produirà un error en fer la comparació `p==q`.

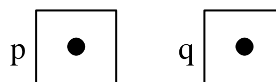
Veurem la seva utilitat al capítol de llistes seqüencials.

5.5. Un exemple amb punters

Anem a veure un senzill programa que treballa amb punters, comentat passa a passa.

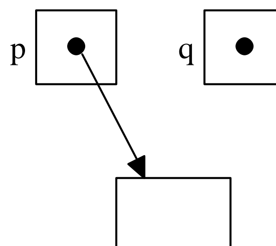
```
#include <stdlib.h>

main()
{
      int *p, *q;
```

Això crea dos punters a enters, p i q:



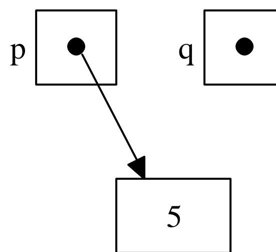

```
p=(int *)malloc(sizeof(int));
```

Això crea un enter a memòria el qual és apuntat per p:



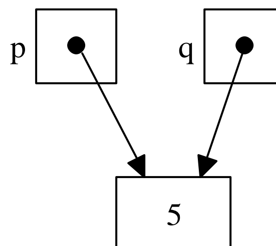

```
*p = 5;
```

Això fa que l'enter al qual apunta p prengui el valor 5:



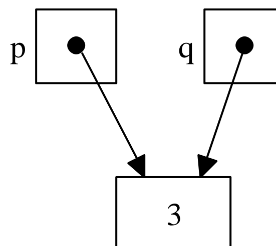

```
q = p;
```

Ara q apunta a la mateixa direcció que p:




```
*q = 3;
```

Per últim, l'enter al qual apunta q passa a valer 3 (la qual cosa fa que també ho valgui *p, ja que són el mateix):




```
}
```

Bibliografia

Llibre recomanat:

Brian W. Kernighan, Dennis M. Ritchie: *El lenguaje de programación C*. Segunda edición. Prentice-Hall. ISBN: 968-880-205-0

Altres llibres sobre programació en C:

Herbert Schildt: *C Manual de referencia*. Mc Graw Hill. 84-481-0335-1

James L. Antonakos, Kenneth C. Mansfield Jr.: *Programación estructurada en C*. Prentice-Hall. ISBN: 84-89660-23-9

Marco A. Peña, José M. Cela: *Introducción a la programación en C*. Edicions UPF. ISBN: 84-8301-429-7

Luis Joyanes, Ignacio Zahonero: *Programación en C*. Mc Graw Hill. ISBN: 84-481-3013-8

Félix García, Jesús Carretero, Javier Fernández, Alejandro Calderón: *El lenguaje de programación C. Diseño e implementación de programas*. Prentice-Hall. ISBN: 84-205-3178-2

P.J. Plauger, Jim Brodie: *C Estándar. Guía de referencia rápida para programadores*. Anaya. ISBN: 84-7614-264-1

Altres "clàssics" sobre algorísmica (més avançats):

Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft: *Estructuras de datos y algoritmos*. Addison Wesley, 1988. ISBN: 968-444-345-5

Niklaus Wirth: *Algoritmos + estructuras de datos = programas*. Ediciones del Castillo, 1980. ISBN: 84-219-0172-9

Niklaus Wirth: *Algoritmia y estructuras de datos*. Prentice Hall, 1987. ISBN: 968-880-113-5

D.E. Knuth: *El arte de programar ordenadores* (3 volums). Editoria Reverté. ISBN: 84-291-2661-9

Terrence W. Pratt, Marvin V. Zelkowitz: *Lenguajes de programación. Diseño e implementación*. Prentice Hall, 3ª edic., 1997. ISBN: 970-17-0046-5

G. Brassard, P. Bratley: *Fundamentos de algoritmia*. Prentice-Hall. ISBN: 84-89660-00-X