Punters, tipus agregats i cadenes de caràcters

Lluís Garrido – lluis.garrido@ub.edu

Setembre 2017

El llenguatge C és el llenguatge per excel·lència en la programació de sistemes (programació a baix nivell). És un llenguatge que permet als programadors programar de forma molt propera al llenguatge màquina. El llenguatge C proveeix una gran varietat de tipus de dades, incloent punters, tipus numèrics i diverses formes per definir un tipus agregat (en el tipus agregats s'inclouen les estructures i els vectors). En aquest document es revisen tots aquests tipus de dades, incloent els punters a funcions.

Índex

1 Tipus escalars en C

Hi ha 7 tipus d'escalars al llenguatge C. Aquests es llisten a continuació. Per a cada tipus indiquem la mida en bytes en memòria.

- *char*, amb una mida d'1 byte.
- *short*, amb una mida de 2 bytes.
- *int*, amb una mida de 4 bytes.
- *long*, amb una mida de 4 bytes per a sistemes de 32 bits, i 8 bytes per sistemes de 64 bits.
- *float*, amb una mida de 4 bytes.
- *double*, amb una mida de 8 bytes.
- *punter*, amb una mida de 4 bytes per a sistemes de 32 bits, i 8 bytes per a sistemes de 64 bits.

Per obtenir la mida de cada tipus, podem fer servir la instrucció C *sizeof*, tal com s'indica en el següent exemple (correspon al fitxer `exemple1.c` del directori `src`)

```c
#include <stdio.h>

int main(void)
{
        printf("sizeof(long) = %d\n", sizeof(long));
        return 0;
}
```

Suposant que el codi està guardat com `exemple1.c`, podem compilar-lo i executar-lo amb la següent instrucció

```
$ gcc exemple1.c -o exemple1
$ ./test
```

2 Els tipus agregats en C

2.1 Vectors estàtics

Els vectors i les estructures són els anomenats tipus agregats en C. Aquests tipus són més complexes que els escalars. Podem, per exemple, declarar un vector de sencers de la següent forma (correspon al fitxer `exemple2.c` del directori `src`).

```c
#include <stdio.h>

int main(void)
{
  int i, a[5];

  for(i = 0; i < 5; i++)
    a[i] = 0;

  return 0;
}
```

En aquest exemple estem declarant un vector estàtic de sencers de 5 elements o posicions. Recordar que els índexos del vector, com a la majoria dels llenguatges, va de 0 a 4. Diem que és

un vector estàtic ja que la mida del vector es declara a l'hora d'escriure el programa. És a dir, el compilador sap quina és la mida del vector en el moment de crear l'executable i, per tant, inclou la "reserva" la memòria necessària pel vector. En concret, en aquest exemple el vector es guarda a la pila del programa. A la pila del programa s'emmagatzemen les variables locals definides a les funcions (llevat de per exemple, la memòria dinàmica reservada amb `malloc`, vegeu secció 2.2). La pila del programa té una mida relativament petita (al voltant de 1MBytes) i per tant no és gens aconsellable que hi emmagatzemeu vectors de mida gaire gran.

Podem declarar també vectors d'estructures. Vegeu el següent exemple (correspon al fitxer `exemple3.c` del directori `src`).

```c
#include <stdio.h>

struct camp
{
  int identificador;
  double valor;
};

int main(void)
{
  int i;
  struct camp a[5];

  printf("sizeof(struct camp) = %d\n",
      sizeof(struct camp));

  for(i = 0; i < 5; i++)
  {
    a[i].identificador = 0;
    a[i].valor = 0.0;
  }

  return 0;
}
```

En aquest exemple estem declarant un vector estàtic de 5 elements o posicions. Cada element d'aquest vector és de tipus *struct camp*. Quin és el valor de la mida en bytes de l'estructura que s'imprimeix per pantalla? Coincideix amb el que esperàveu?

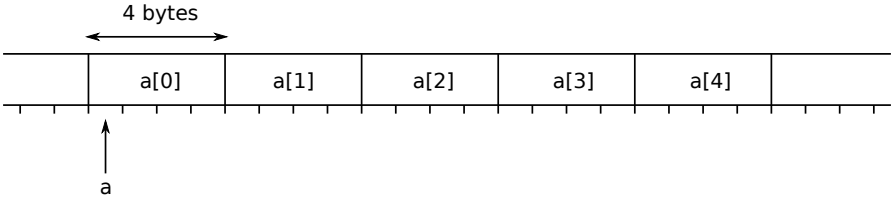
Moltes vegades, però, no sabem quina és la mida del vector fins el moment d'executar el programa. La mida del vector pot dependre, per exemple, de la mida de les dades que l'usuari li introdueix al programa. És per això que moltes vegades cal utilitzar vectors dinàmics.

2.2 Vectors dinàmics

Els vectors dinàmics i el seu tipus bàsic associat, el punter, és un dels temes en què la majoria de la gent té problemes. Un punter és un tipus de variable que apunta a una direcció de memòria. En altres paraules, un punter és una variable que emmagatzema un nombre sencer sense signe prou gran per poder apuntar a qualsevol direcció de memòria de la RAM.

La instrucció *malloc* és la que ens permetrà reservar memòria de forma dinàmica. Vegem-ne un exemple (correspon al fitxer `exemple4.c` del directori `src`) a la figura 1.

Aquest exemple correspon a l'exemple vist a la secció 2.1 però utilitzant vectors dinàmics. La funció *malloc* només requereix un paràmetre; el nombre de bytes a reservar. Quan reservem n bytes de memòria, reservarem n bytes contigus en memòria, veure figura 1. La funció *malloc* retorna l'índex de memòria a l'inici d'aquest vector (un sencer sense signe) i s'assigna al punter. En cas que no hi hagi prou memòria disponible, *malloc* retorna l'índex 0 (també conegut amb




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

int main(void)
{
  int i, *a;

  a = malloc(sizeof(int) * 5);

  if (a == NULL)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  for(i = 0; i < 5; i++)
    a[i] = 0;

  free(a);

  return 0;
}
```

Figura 1: Exemple de la funció *malloc*. Part superior, ubicació en memòria del vector de 5 sencers. Part inferior, codi font (fitxer `exemple4.c`).

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

int main(void)
{
  int i, j, *a;

  for(i = 0; i < 1000; i++)
  {
    a = malloc(sizeof(int) * 5);

    if (a == NULL)
    {
      printf("No he pogut reservar la memòria\n");
      exit(1);
    }

    for(j = 0; j < 5; j++)
      a[j] = 0;
  }

  free(a);

  return 0;
}
```

Figura 2: Exemple incorrecte de l'ús de la funció *free* (codi fitxer `exemple5.c`).

NULL).

El punter – en aquest exemple la variable "*a*" – s'emmagatzema a la pila i té una mida de 8 bytes. Un cop s'ha cridat a la funció malloc s'assigna a la variable "*a*" el valor retornat per aquesta funció, una direcció de memòria que el sistema operatiu ha reservat per poder emmagatzemnar-hi els bytes que s'han demanat.

És necessari alliberar la memòria un cop ja no ens faci falta accedir al vector. Per això cal cridar a la funció *free*. Aquesta funció només requereix un paràmetre: l'índex de memòria retornat per la funció *malloc*. El cas de no alliberar el vector es poden produir problemes de memòria: cada cop que es crida a *malloc* es "bloquegen" els bytes demanats perquè el procés que fa la crida a *malloc* els pugui utilitzar. El fet de cridar repetides vegades a *malloc* farà que es redueixi, poc a poc, la memòria disponible per altres processos. És molt important alliberar la memòria en el moment en què ja no faci falta de forma que la memòria quedi disponible per altres processos. En tot cas, la memòria no alliberada de forma manual s'alliberarà automàticament en sortir del procés (el sistema operatiu s'encarrega de fer-ho ja que sap quina és la memòria dinàmica assignada a cada procés).

A la figura 1 es mostra un l'ús correcte de la funció *free*: s'allibera la memòria a què apunta la variable "a" un cop ja no fa falta utilitzar-la.

A la figura 2 se us mostra un exemple d'ús incorrecte de la funció *free* (codi `exemple5.c`). En aquest programa estem reservant 1000 cops un vector de sencers de 5 posicions. A cada iteració sobreescrivim el valor de la variable *a* i, per tant, la posició (el sencer sense signe) a la memòria reservada en una iteració es "perd" en sobreescriure la variable *a* a la següent iteració. A cada iteració es reserven (i es "bloquegen") $5 \times sizeof(int)$ bytes. La instrucció *free*, en aquest programa, només allibera la memòria reservada a la darrera iteració.

Aquest és un dels errors comuns a l'hora de programar. En aquest exemple la memòria que es "perd" a cada iteració no es massa gran, però penseu en el que pot passar amb un programa molt complexe. A cada iteració hi haurà menys memòria disponible i per tant l'ordinador anirà

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

struct camp
{
  int identificador;
  double valor;
};

int main(void)
{
  int i;
  struct camp *a;

  a = malloc(sizeof(struct camp) * 5);

  if (!a)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  for(i = 0; i < 5; i++)
  {
    a[i].identificador = 0;
    a[i].valor = 0.0;
  }

  free(a);

  return 0;
}
```

Figura 3: Exemple de l'ús de la funció *free* amb estructures (codi fitxer `exemple6.c`).

cada cop més lent (degut a la paginació amb la memòria d'intercanvi o *swap* al disc). Com s'hauria de modificar el programa perquè en sortir del programa s'hagi alliberat tota la memòria dinàmica? Penseu-hi!

De forma similar al cas estàtic, podem utilitzar la funció *malloc* amb estructures. Vegem l'exemple de la secció 2.1 re-escrit pel cas dinàmic, veure figura 3.

A la funció *malloc* li indiquem el nombre de bytes que volem reservar (en aquest cas 5 vegades el nombre de bytes que ocupa l'estructura). Sempre cal comprovar si la memòria ha pogut ser reservada. Observeu que en aquest cas s'allibera la memòria després de cada iteració i, per tant, s'arregla el problema que hi ha a l'exemple de la figura 2.

2.3 Matrius dinàmiques

En C, també es poden definir matrius estàtiques i dinàmiques. En aquesta secció ens centrem únicament en matrius dinàmiques ja que acostumen a ser també una font de confusió. A la figura 4 es mostra l'exemple de la figura 2 re-escrit per treballar amb matrius.

Observeu el següent al codi:

1. A la figura 2 la variable `a` es declara com un "punter simple" que es pot interpretar com un vector. A la línia 6 del codi de la figura 4 es declara la variable `a` com a "doble punter" que es pot interpretar com una matriu bi-dimensional (un vector de vectors). Un "punter triple" es pot interpretar com una matriu tri-dimensional, etc.

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

int main(void)
{
  int i, j, **a;

  a = malloc(sizeof(int *) * 1000);

  for(i = 0; i < 1000; i++)
  {
    a[i] = malloc(sizeof(int) * 5);

    if (a[i] == NULL)
    {
      printf("No he pogut reservar la memòria\n");
      exit(1);
    }

    for(j = 0; j < 5; j++)
      a[i][j] = 0;
  }

  /* Processem les dades .... */


  /* I alliberem en acabar de processar */

  for(i = 0; i < 1000; i++)
      free(a[i]);

  free(a);

  return 0;
}

```

Figura 4: Exemple d'ús de malloc i free amb matrius (veure fitxer `exemple5-matriu.c`).

2. A la línia 8 reservem memòria per a un vector que emmagatzema 1000 punters (observeu l'argument de la funció *sizeof*). A la línia 12 reservem memòria per a cadascun dels vectors de 5 posicions i el guardem al vector de punters (observeu també l'argument de la funció *sizeof*).

3. Entre les línies 24 i 27 hi ha tot el codi que manipula i accedeix a la matriu.

4. Finalment, un cop hem acabat de manipular la matriu, alliberem la memòria. Observeu que es fa entre les línies 29 i 32. És important notar que la memòria s'allibera (amb *free*) en ordre invers al qual s'ha reservat (amb *malloc*). En particular, no és correcte executar el codi de la línia 32 abans d'executar el bucle que hi ha a les línies 29 i 30. Us podeu imaginar per què ?

2.4 La funció *sizeof*

En els exemples que hem vist fins ara hem fet servir la funció *sizeof*: aquesta funció permet saber el nombre de bytes que ocupa un determinat tipus de dades o estructura.

Tornem a l'exemple de la figura 4. Observar que a la línia 8 i 12 del codi es fa ús d'aquest operador. A la línia 8 s'utilitza per saber quants bytes fan falta per emmagatzemar "un punter a un sencer" mentre que a la línia 12 es fa servir per saber quants bytes fan falta per emmagatzemar "un sencer". Als ordinadors de 64 bits, amb els quals estem habituats a treballar, el resultat d'aquest opeador és 8 i 4 respectivament. Qualsevol punter, generalment, ocupa 8 bytes a memòria (sigui a un sencer, un float, un double, una estructura, etc.). Un sencer n'ocupa 4, de bytes.

A vegades, però, l'operador *sizeof* s'utilitza de forma incorrecte. En particular, l'operador *sizeof* no permet pas saber la mida d'un vector. Al codi de la figura 1, quin serà el valor retornat en fer *sizeof(a)*? Alguns potser esperen que es retorni 5, la mida del vector reservat. Però no és així. En fer *sizeof(a)* es retornarà "la mida en bytes del que ocupa la variable *a* a memòria". Com que la variable *a* és un "punter a sencer", l'operador *sizeof(a)* retornarà sempre 8, que correspon al nombre de bytes que ocupa un punter a memòria.

No hi ha cap funció en C que permeti saber el nombre d'elements que hem reservat a l'hora de fer un *malloc*. En cas que ens interessi saber quina és la mida del vector, hem d'emmagatzemar aquesta informació en una variable a part. En llenguatges com Java o Python, per exemple, els vectors estan implementats internament com una estructura (o objecte) que conté, entre altres, dues variables membre: una amb el "punter a la dada" i una altra amb la longitud d'aquesta. És per això que no fa falta guardar la longitud d'un vector en aquests llenguatges.

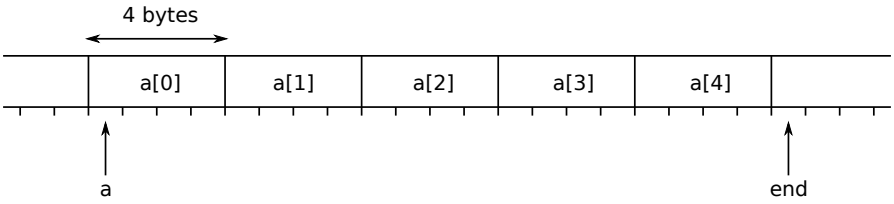
3 Programació amb punters

3.1 Operacions bàsiques

Tal com s'ha comentat a la secció 2.2, els punters són un tipus de variable que apunten a una direcció de memòria.

Els punters és fan servir típicament per recórrer els elements d'un vector de forma eficient. Per exemple, vegem un altre cop l'exemple vist de la secció 2.2 fent servir punters per recórrer els elements del vector, veure figura 5

Estudieu amb deteniment l'exemple de la figura 5. En aquest cas estem declarant dos punters addicionals, *pos* i *end*. La variable *pos* serveix per recórrer els elements del vector i la variable *end* apunta al primer índex de memòria després del vector, veure figura 5. En C la instrucció *end* = *a* + *5* no vol dir pas "suma 5 al valor de la variable *a*". Si, per exemple, la variable *a* apunta a




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

int main(void)
{
  int *a, *pos, *end;

  a = malloc(sizeof(int) * 5);

  if (a == NULL)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  end = a + 5;

  for(pos = a; pos < end; pos++)
    *pos = 0;

  free(a);

  return 0;
}
```

Figura 5: Operacions bàsiques amb punters . Part superior, representació gràfica. Part inferior, codi associat (codi `exemple7.c`).

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

struct camp
{
  int identificador;
  double valor;
};

int main(void)
{
  int i;
  struct camp *a, *pos, *end;

  a = malloc(sizeof(struct camp) * 5);

  if (!a)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  end = a + 5;

  for(pos = a; pos < end; pos++)
  {
    pos->identificador = 0;
    pos->valor = 0.0;
  }

  free(a);

  return 0;
}
```

Figura 6: Exemple que mostra com recórrer un vector d'estructures amb punters (codi `exemple8.c`), vegeu secció 3.1.

l'índex de memòria 394838430, la variable *end* no serà $394838430 + 5 = 394838435$. És una mica més complex que això en cas que es tracti d'operacions amb punters. Quan es fan operacions amb punters es té en compte la mida en bytes de l'element associat al punter. És a dir, en aquest cas la variable a és un punter a un sencer. Cada sencer ocupa 4 bytes. L'operació *end = a + 5* farà, en el context el nostre exemple, la següent operació: $394838430 + 5 \times sizeof(int) = 394838450$. Aquest és l'índex de memòria associat al primer byte després del vector, veure figura 5.

De forma similar, a la instrucció *for* es realitza l'operació *pos++*. L'operació d'increment no incrementa en un l'índex de la posició de memòria, sinó que l'incrementa segons *sizeof(int)*. El bucle, per tant, permet recórrer tots els elements del vector. La instrucció **pos=0* permet escriure el valor zero en la posició de memòria a la qual apunta la variable *pos*.

L'exemple que hem vist és computacionalment més eficient que el de la secció 2.2. En particular, a la secció 2.2 accedim a un element amb $a[i]$, mentre que aquí hi accedim amb **pos*. L'operació $a[i]$ és equivalent a fer **(a+i)*. Accedir a un element fent servir $a[i]$ és doncs més costós que fer-ho amb **pos*. El guany computacional no es nota gaire si el vector és petit (com en aquest exemple), però sí que hi ha guany si el vector és molt gran (com per exemple si es processa una imatge digital).

De forma equivalent, podem re-escriure l'exemple de la figura 3 tal com es mostra a la figura

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

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

  a = malloc(sizeof(int) * 5);

  if (a == NULL)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  b = a + 1;

  for(i = -1; i < 4; i++)
    b[i] = 0;

  free(a);

  return 0;
}
```

Figura 7: Operacions aritmètiques amb punters (codi `exemple9.c`), vegeu secció 3.2.

6. Aquí tenim també la instrucció *end* = *a* + *5*, la mateixa que abans. Insistim en el fet que a l'hora de fer operacions amb punters es té en compte el tipus associat al punter.

3.2 Operacions aritmètiques amb punters

Un dels grans maldecaps del programador que s'inicia en el llenguatge C són els punters. La gran versatilitat del llenguatge C prové, de fet, dels punters. Un cop s'entenen bé es pot aprofitar la seva versatilitat per programar de forma eficient.

Analitzem el programa de la figura 7 (veure `exemple9.c`). La variable b és un punter al segon element del vector a, és a dir, que la variable a apunta a l'element $a[0]$, mentre que b apunta a $a[1]$. Per tant, accedir a $b[0]$ és equivalent a accedir a $a[1]$. Accedir a $b[1]$ és equivalent a accedir a $a[2]$, i així successivament. Per la mateixa raó, b[-1] és equivalent a accedir a $a[0]$! Recordeu que fer b[i] és equivalent a fer **(b+i)*, on la variable i pot ser qualsevol valor sencer (positiu o negatiu). Aquest truc permet definir, per exemple, vectors als quals s'accedeix fent servir índex negatius.

És important fer notar que a l'hora d'alliberar la memòria cal cridar la funció *free* amb l'índex de memòria retornat per la funció *malloc*. Si no fos així el programa segurament petarà en el moment d'executar-se.

3.3 Els maldecaps amb els punters

La gran versatilitat dels punters pot ser a la vegada font del maldecaps. Cal anar molt de compte a l'hora de fer operacions amb punters i memòria dinàmica. Vegem un exemple (codi `exemple10.c`), veure figura 8. Quin creieu que serà el resultat de l'execució d'aquest programa? S'adonarà el compilador que esteu escrivint zeros fora del vector que heu creat? O potser a l'hora d'executar el programa petarà quan intenti escriure fora del vector? Proveu-ho!

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

int main(void)
{
  int i;
  int *a;

  a = malloc(sizeof(int) * 5);

  if (a == NULL)
  {
    printf("No he pogut reservar la memòria\n");
    exit(1);
  }

  printf("Començo a escriure zeros...\n");

  for(i = 1000; i < 1100; i++)
    a[i] = 0;

  printf("He escrit els zeros!\n");

  free(a);

  return 0;
}
```

Figura 8: Els maldecaps amb els punters, veure secció 3.3. Codi `exemple10.c`.

Si tot va segons el previst podreu compilar i executar sense problemes. Apareixeran per pantalla els dos missatges! El llenguatge C no comprova, en temps d'execució, si estem escrivint fora dels límits del vector. A altres llenguatges com el *Java* o el *Python* sí que es fa aquesta comprovació, cosa que fa el programa més lent.

Què és el que esteu sobreescrivint amb zeros? És difícil de dir. En qualsevol cas, esteu sobreescrivint amb zeros una zona de memòria que no s'hauria de sobreescriure. Es pot donar el cas que sobreescriviu amb zeros altres variables del vostre procés. Potser sobreescriviu amb zeros una zona de memòria d'un altre procés (amb la qual cosa es produeix el temut "Segmentation fault"). Potser no té cap efecte nociu el que esteu fent. La majoria de vegades, però, no és així. Sovint no és fins molt més tard en l'execució del programa que us adoneu que alguna variable no té el valor que toca o que el programa peta (amb un "Segmentation fault") en un punt que és evident que no hauria de petar. I tot és degut a que no heu treballat correctament amb els punters!

Què es pot fer en aquest cas? No cal canviar de llenguatge de programació. Existeixen eines que permeten detectar aquests problemes. Una d'aquestes eines és el *valgrind*, que està instal·lat a les aules, i que està descrit a una altra fitxa. Aquí ja no es donen més detalls al respecte.

4 Punters a funcions

4.1 Concepte i utilització

La memòria RAM es pot interpretar com un vector de bytes on es guarden dades i codi executable. És doncs natural pensar que els punters no només poden apuntar a dades (vector de sencers, vector d'estructures, etc.) sinó que també poden apuntar a codi executable. En particular, els

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

double func1(int a)
{
  return sqrt((double) a);
}

double func2(int b)
{
  return log((double) b);
}

int main(void)
{
  double ret;
  int selecciona = 1;
  double (*myfunc)(int);

  if (selecciona == 1)
    myfunc = func1;
  else
    myfunc = func2;

  ret = myfunc(5);

  printf("Resultat: %e", ret);

  return 0;
}
```

Figura 9: Punters a funcions, veure codi `exemple11.c`.

punters poden apuntar al punt d'entrada d'una funció.

Vegem-ne un exemple (codi `exemple11.c`), veure figura 9. En aquest programa declarem una variable *myfunc* que és un punter a una funció (la declaració és una mica estranya!). Aquesta declaració diu que la funció admet com a paràmetre un sencer i retorna un *double*. La instrucció *myfunc = func1* fa que el la variable *myfunc* apunti a la funció *func1*. En el moment de fer *ret = myfunc(5)* cridem a la funció a la que apunta *myfunc*, que és *func1*. És a dir, que fer *myfunc(5)* és equivalent, en aquest programa, a fer *ret = func1(5)*.

Per a compilar el programa feu servir la instrucció

```
$ gcc test.c -o test -lm
```

4.2 Exemple d'utilització: Quicksort

La llibreria estàndard de C conté, a més de les funcions típiques (*malloc*, *free*, *printf*, etc.), la funció *qsort* que permet aplicar l'algorisme de quick sort a qualsevol tipus de dades d'entrada. Si feu

```
$ man qsort
```

obtindreu ajut respecte els paràmetres que s'han de passar en aquesta funció. La pàgina del manual us indica que aquests són

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

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

int compara_sencers(const void *p1, const void *p2)
{
  int *num1, *num2;

  num1 = (int *) p1;
  num2 = (int *) p2;

  if (*num1 < *num2)
    return -1;
  if (*num1 > *num2)
    return 1;
  return 0;
}

int main(void)
{
  int i;
  int vector[8] = {8, 4, 2, 3, 5, 6, 8, 5};

  qsort(vector, 8, sizeof(int), compara_sencers);

  printf("El vector ordenat és ");

  for(i = 0; i < 8; i++)
    printf("%d ", vector[i]);

  printf("\n");

  return 0;
}
```

Figura 10: Exemple d'ús de la funció *qsort*, codi `exemple12.c`.

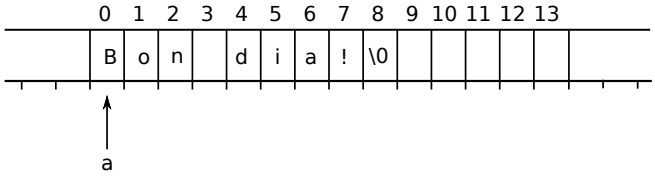
El primer argument (*base*) és un punter cap a l'inici del vector que conté les dades a ordenar. Amb el segon argument (*nmemb*) indiquem el nombre d'elements que té el vector, mentre que amb el tercer (*size*) indiquem la mida de cadascun dels elements (recordeu que la mida d'un element es pot obtenir amb *sizeof*). Finalment, el quart argument és un punter cap a la funció que permet comparar dos elements qualssevol del vector. Aquest quart argument és una funció que l'usuari (nosaltres) ha d'escriure i passar com a argument a la funció *qsort*. És a dir, que la llibreria del sistema proporciona l'algorisme de QuickSort però necessita que nosaltres li proporcionem la funció que permeti comparar dos elements del vector qualssevol. Els elements poden ser sencers, cadenes de caràcters, o inclús una estructura amb múltiples membres.

Si llegiu la pàgina del manual, us indica que "The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted vector is undefined."

Anem a veure aquí com es pot aplicar la funció *qsort* per a ordenar un vector de sencers. Per simplificar, farem servir memòria estàtica (codi `exemple12.c`), veure figura 10.

Analitzeu els paràmetres que se li passen a la funció *qsort*. De tots els paràmetres el darrer és el que interessa aquí: és un punter a una funció que permet comparar dos elements del vector. Cada cop que l'algorisme de *qsort* necessiti comparar dos elements, cridarà a la funció *compara_sencers* i li passarà per argument els dos elements a comparar.

Analitzem la funció de comparació *compara_sencers*. La funció obté, per paràmetre, dos




```c
#include <stdio.h>

int main(void)
{
  char a[14] = "Bon dia!";

  printf("%s", a);

  return 0;
}
```

Figura 11: Cadenes i caràcters, concepte. Part superior, representació gràfica. Part inferior, codi associat (veure codi `exemple13.c`).

punters genèrics (tipus *void*) cap als elements que s'han de comparar. Un punter genèric és un punter a una direcció de memòria sense especificar el tipus de dades que s'hi emmagatzema. La funció *qsort* no sap pas si estem comparant sencers, *double* o algun tipus d'estructura més complexa. Nosaltres, com a programadors de la funció, sabem que els elements que es comparen són sencers. Sabem que els dos índexos de memòria que obté la funció *compara_sencers* són en realitat punters a sencers. És per això que es fa el *casting* a les variables *num1* i *num2*. Després, només cal comparar els dos sencers i retornar un -1,0 o 1 segons indica el manual.

5 Cadenes de caràcters

5.1 Concepte i utilització

Un cas particular de tipus agregat són els cadenes de caràcters (*strings*). Vegem-ne un exemple de declaració estàtica d'una cadena.

A la figura 11 (codi `exemple13.c`) es mostra la representació en memòria d'aquesta cadena de caràcters. El punter a apunta al primer caràcter de la cadena. Per tant, $a[0]$ correspon a la lletra B, $a[1]$ a la lletra o, i així successivament. La representació en memòria de la cadena de caràcters termina amb un caràcter especial: el caràcter *0* (zero o nul). Aquest és el caràcter que a C s'utilitza per saber on termina una cadena de caràcters. Per exemple, en cridar la funció *printf* s'imprimeixen per pantalla tots els caràcters fins que es troba el caràcter nul. Les posicions de memòria després del caràcter nul fins arribar a la mida de la cadena (en aquest exemple fins a la posició 13) són ignorades per la funció *printf*.

La cadena de caràcters "Bon dia!" té una longitud de 8 caràcters, però ocupa un mínim de 9 bytes de memòria (ja que tota cadena de caràcters ha de tenir el caràcter nul al final). Suposem que estem interessats en declarar una cadena de forma dinàmica, tal com es mostra a la figura 12. En aquest exemple estem reservant 100 bytes per a una cadena de caràcters. Això vol dir que a tot estirar la cadena podrà tenir a 99 caràcters imprimibles, ja que la darrera posició haurà d'estar ocupada pel caràcter nul. La cadena, naturalment, pot tenir una longitud inferior.

El llenguatge C disposa de múltiples funcions per manipular caràcters. Podeu obtenir un llistat d'elles executant la següent instrucció en la línia de comandes

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

int main(void)
{
  int i;
  char *a;

  a = malloc(sizeof(char) * 100);

  strcpy(a, "Bon dia!\n");
  printf("%s",a);

  free(a);

  return 0;
}
```

Figura 12: Cadena de caràcters dinàmica (codi `exemple14.c`).

```
$ man string
```

Entre les instruccions disponibles, hi ha, per exemple, *strlen* (que permet obtenir la longitud d'una cadena de caràcters), *strcpy* (que permet copiar una cadena una zona de memòria a una altra). Executeu les següents instruccions en un terminal.

```
$ man strlen
$ man strcpy
```

Noteu que les funcions tenen comportaments diferents respecte el caràcter nul.

5.2 Pas d'arguments per línia de comandes

En entorns Linux/Unix es comú realitzar programes que accepten arguments per línia de comandes. Per exemple,

```
wc test.c
```

L'aplicació *wc* permet comptar el nombre de línies, paraules i caràcters que conté un fitxer de text. A l'exemple li passem un argument a l'aplicació *wc*; el fitxer a analitzar test.c. En C (i molts altres llenguatges) es poden capturar fàcilment aquests arguments.

Veiem aquí un exemple que imprimeix per pantalla els arguments que se li passen per línia de comandes, veure figura 13. La variable *argc* emmagatzema el nombre d'arguments que se li passen a l'aplicació, i *argv* (declarat com un vector de cadenes de caràcters) emmagatzema els arguments que se li passen a la línia de comandes. Per exemple, per a la línia de comandes

```
./test fitxer
```

el valor de *argc* serà 2, i *argv*[0] contindrà la cadena "./test", i *argv*[1] contindrà la cadena "fitxer".

6 Bibliografia

1. Rochkind, M.J. Advanced Unix Programming. Addison Wesley, 2004. Aquest llibre està disponible a la biblioteca.

2. Stevens, W.R. Advanced Programming in the Unix Environment. Addison Wesley, 2005.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;

  printf("Nombre d'arguments: %d\n", argc);

  for(i = 0; i < argc; i++)
    printf("Argument %d: %s\n", i, argv[i]);

  return 0;
}
```

Figura 13: Pas d'arguments per línia de comandes (codi `exemple15.c`).