Punters, tipus agregats i manipulació de cadenes de caràcters

Lluís Garrido – lluis.garrido@ub.edu

Setembre 2014

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).

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 "reserva" la memòria necessària pel vector. En concret, 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 mica 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;
}
```

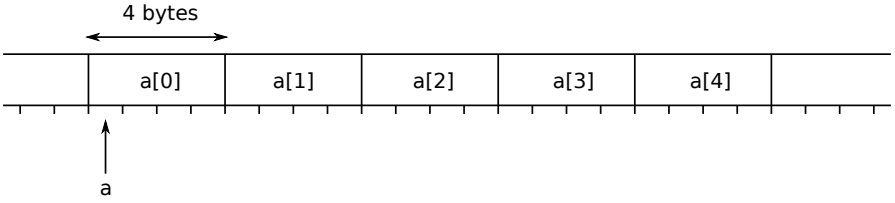
En aquest exemple estem declarant un vector estàtic de 5 elements o posicions. Cada element d'aquest vector és de tipus *struct camp*. Quin és el valor de la mida en bytes de l'estructura que s'imprimeix per pantalla? Coincideix amb el que esperàveu?

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 simplement un tipus de variable que apunta a una direcció de memòria. Dit d'altra forma, un punter és una variable que conté un nombre sencer sense signe. El punter en sí (el sencer sense signe) s'emmagatzema a la pila. El punter pot apuntar a qualsevol punt de la memòria.

La memòria no és més que un vector molt gran de bytes (char). Aquest vector té, en els ordinadors actuals, al voltant de 2147483648 elements (correspon a 2MB de memòria). Podem accedir una posició d'aquest vector mitjançant el seu índex. L'índex 0 correspon




```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`).

a la primera posició, 1 a la segona posició, i així successivament. Típicament els índexos es visualitzen (pels humans) en format hexadecimal. Així, en comptes de referir-nos a la posició 394838430 de memòria, ho expressem com $0x1788C19E$ (en general tot valor hexadecimal està precedit amb el símbol "$0x$" per indicar de forma explícita que es tracta d'un valor hexadecimal)

El punter, per tant, és simplement un índex a una posició d'aquest vector de bytes. Com hem dit abans un punter pot apuntar a qualsevol posició de memòria (sigui a una posició de la pila, o una posició del programa executable, etc.). Típicament, però, els punters es fan servir per gestionar els anomenats vectors dinàmics. De forma senzilla, un vector dinàmic és aquell pel qual no sabem la seva mida abans d'executar el programa, i que pot créixer o decréixer en mida al llarg de l'execució del programa. S'aconsella fer servir vectors dinàmics en comptes dels estàtics en cas que es prevegi que la mida del vector en bytes sigui "relativament gran" (uns 10KBytes, per exemple).

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 de 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

```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`).

l'índex 0 (també conegut amb NULL).

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. Per tant, 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, es convenient saber que 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 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"), doncs, 1000 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 cada cop més lent (l'ordinador aniria 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! A

De forma similar al cas estàtic, podem utilitzar la funció *malloc* amb estructures. Vegem

```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`).

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 aquí es pot interpretar com una matriu bi-dimensional. Un "punter triple" es pot interpretar com una matriu tri-dimensional, etc.

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

```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`).

cadascun dels vectors de 5 posicions i el guardem al vector de punters (observeu també l'argument de la funció *sizeof*).

3. A les línies 20 i 21 accedim a les dades com si fossin una matriu. Com s'ho fa el compilador per generar el codi que permeti accedir a la posició correcta ? Suposem que volem accedir a *a[563][3]*. Primer de tot, el compilador genera codi per accedir al contingut de la variable *a*, que és una posició de memòria a un vector de punters. Aleshores s'accedeix a la posició 563 d'aquest vector, que a la seva vegada també conté una altra posició de memòria, en concret la posició inicial un vector de sencers de 5 posicions. Finalment, s'accedeix a la posició 3 d'aquest vector. A la secció 5.3 es donen més detalls al respecte dintre del context de manipulació de cadenes de caràcters.

4. Se suposa que entre les línies 24 i 27 hi ha tot el codi que manipula i accedeix a la matriu.

5. 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 ?

3 Programació amb punters

3.1 Operacions bàsiques

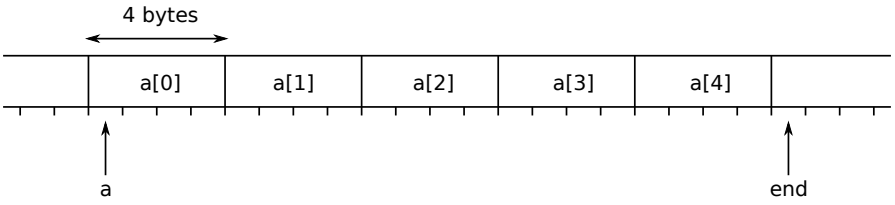
Tal com s'ha comentat a la secció 2.2, els punters són un tipus de variable que apunten a una direcció de memòria.

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 a l'índex de memòria de la variable *a*". Si, per exemple, la variable *a* apunta a l'índex de memòria 394838430, la variable *end* no serà $394838430 + 5 = 394838435$. No, és una mica més complex que això. 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)*. Per tant, accedir a un element fent servir *a*[i] és més costós que fer-ho amb **pos*. És habitual recórrer els elements d'un vector fent




```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.

servir punters. 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 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. A l'exemple d'abans *a*, *pos* i *end* estan declarats com a punters a sencers. En aquest exemple les variables *a*, *pos* i *end* estan declarats com a punters a *struct camp*. Per tant, si el punter *a* apunta a l'índex de memòria 394838430, l'operació *end = a + 5* farà que *end* apunti a $394838430 + 5 \times sizeof(struct\ camp)$. Per altra banda, recordeu que en C la instrucció *pos->identificador=0* és equivalent a fer *(*pos).identificador=0*. Són dues formes d'aconseguir el mateix.

3.2 Operacions aritmètiques amb punters

Com ja hem dit, 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.

```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.

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!

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. En el nostre exemple, esteu escrivint zeros fora del vector que heu reservat. Per exemple, suposem que en cridar a *malloc* es retorna el valor 394838430. Aquest valor és la posició de la memòria en què es troba emmagatzemat $a[0]$. Només hem reservat memòria per a 5 elements, és a dir, hem reservat

```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`.

memòria des de la posició 394838430 fins a la $394838430 + 5 * sizeof(int) - 1 = 394838449$, on hem suposat que un sencer ocupa 4 bytes. En aquest exemple esteu escrivint zeros des de la posició $394838430 + 1000 * sizeof(int)$ fins a la posició $394838430 + 1100 * sizeof(int) - 1$. Això es troba fora de la zona de memòria que nosaltres hem reservat!

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. 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, no cal canviar de llenguatge. Existeixen eines que permeten detectar aquests problemes. Una d'aquestes eines és el *valgrind*, que està instal·lat a les aules. El *valgrind* és una maquina virtual que permet executar un programa qualsevol. En temps d'execució *valgrind* comprova que els accessos a la memòria es realitza correctament. En cas que es realitzi alguna operació de forma incorrecta ho indicarà mitjançant un missatge per pantalla.

Vegem com funciona amb l'exemple que estem estudiant. Primer de tot, cal compilar el programa incloent informació per depuració.

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

A continuació executem *valgrind* i li indiquem quin és el programa que volem depurar

```
$ valgrind ./test
```

El resultat de l'execució al meu ordinador és el següent

```
==23957== Memcheck, a memory error detector
==23957== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==23957== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==23957== Command: ./test
==23957==
==23957==  Començo a escriure zeros...
==23957== Invalid write of size 4
==23957==    at 0x40066C: main (test.c:20)
==23957==  Address 0x519afe0 is not stack'd, malloc'd or (recently) free'd
==23957==  He escrit els zeros!
==23957==
==23957== HEAP SUMMARY:
==23957==     in use at exit: 0 bytes in 0 blocks
==23957==   total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==23957==
==23957== All heap blocks were freed -- no leaks are possible
==23957==
==23957== For counts of detected and suppressed errors, rerun with: -v
==23957== ERROR SUMMARY: 100 errors from 1 contexts (suppressed: 4 from 4)
```

L'error "Invalid write of size 4" (apareix entre els missatges "Començo a escriure zeros" i "He escrit els zeros!"). Aquest error ens diu que estem intentant escriure 4 bytes (un sencer!) en una zona de memòria no vàlida. A més, ens indica que el problema es troba a la línia 20 del programa test.c. Fantàstic, no? El programa *valgrind* permet detectar molts errors típics degut a l'ús de memòria dinàmica: escriptura o lectura de dades en posicions de memòria no vàlides, memòria no alliberada, etc. És per tant molt recomanable utilitzar el *valgrind* per assegurar que el codi funciona correctament. L'aplicació *valgrind* no permet, en canvi, detectar problemes en utilitzar memòria estàtica (la de la pila). És convenient doncs utilitzar memòria dinàmica per facilitar la detecció d'aquest tipus de problemes.

Aquests tipus de problemes amb els punters no existeix en altres llenguatges com *Java* o *Python*. La raó és que aquests llenguatges, en realitzar una operació d'accés a un vector tipus a[i], comproven automàticament que el valor de la variable i estigui dintre del rang de valors vàlids. En aquest darrer exemple, el rang de valors vàlids és de *0* a *4*. Això fa que el programa sigui més lent. Per això, moltes vegades, si es vol aconseguir una aplicació eficient cal fer servir el llenguatge C.

4 Punters a funcions

4.1 Concepte i utilització

Ja que la memòria no és més que un vector gegant de bytes on es guarden dades i codi executable, és 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 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

```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`.

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 *));
```

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."

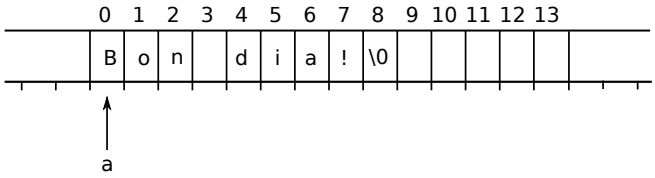
L'exemple de codi d'exemple donat al manual és enrevessat i complex. Anem a veure aquí com es pot aplicar la funció *qsort* per a ordenar un vector de sencers. Per simplificar, farem servir memòria estàtica (codi `exemple12.c`), veure figura

Analitzeu els paràmetres que se li passen a la funció *qsort*. De tots el 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 punters genèrics (tipus *void*) cap als elements que s'han de comparar. És a dir, a la funció de comparació se li passen els dos índexos de memòria dels elements que s'han de comparar. La funció *qsort* no sap pas si estem comparant sencers, *double* o algun tipus d'estructura més complexa. Per això treballa amb punters genèrics (però necessita saber la mida de cadascun dels elements del vector). 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

```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]);

  return 0;
}
```

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




```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`).

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.

Es proposa fer la següent modificació al programa. Suposeu que en comptes d'ordenar sencers volem ordenar cadenes de caràcters. La cadena de caràcters està formada, per exemple, per

```
char vector[4][9] = {"Hola", "Bon dia", "Adeu", "Sistemes"};
```

És a dir, estem definint 4 cadenes, cadascuna ocupant 9 bytes de memòria (tots els elements a ordenar ocupen el mateix espai). Podeu escriure un programa que permeti ordenar aquestes cadenes de caràcters? Per això ajudeu-vos de les funcions disponibles per a manipular cadenes de caràcters (en particular *strcmp* i *strncmp*).

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 (codi `exemple13.c`).

A la figura 11 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 C 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) poden tenir un valor qualsevol.

És important tenir aquest darrer punt en compte. 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

```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);

  return 0;
}
```

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

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. Observeu que en aquest cas no alliberem la cadena amb *free* (ho hauríem de fer!).

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

```
$ 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".

```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`).

5.3 Manipulació de cadenes de caràcters

En aquesta secció es mostra com manipular una cadena de caràcters. En particular, l'objectiu és extreure les cadenes de caràcters "Bon dia", "Sistemes Operatius" i "Barcelona" sabent que aquestes estan separades pel caràcter ";". Hi ha diverses solucions per aquest problema. Aquests es discuteixen a continuació.

5.3.1 Manipulació directa de la cadena

Una primera forma d'abordar el problema es basa en manipular directament la cadena de caràcters. No es fa servir cap de les funcions de manipulació de cadenes de caràcters de què disposem a les llibreries estàndard. Es proposen dues solucions:

- Solució 1: aquí teniu una solució a l'exercici suposant que no podem modificar la cadena de caràcters original. Suposem també que desconeixem el nombre de subcadenes que s'han d'extreure. El codi declara una variable de punter doble anomenada *subcadena*. En el context d'aquest exemple, podem interpretar aquesta variable com un vector de cadenes de caràcters, vegeu secció 2.3. A la figura 15 podem veure la disposició en memòria de les variables *str* i *subcadena* després de processar la cadena de caràcters "Bon dia;Sist;Hola", veure codi `exemple16.c`.

- Solució 2: A continuació anem a fer el mateix exemple d'abans però suposant que podem modificar la cadena original *str*. Observeu com es manipula en aquest cas el vector de cadenes de caràcters. Vegeu figura 16. De forma similar, a la figura 17 podem veure la disposició en memòria de les variables *str* i *subcadena* després de processar la cadena "Bon dia;Sist;Hola".

És important entendre en aquests dos exemples la interpretació de la variable *subcadena*, per això compareu les figures 15 i 17.

5.3.2 Utilització de la funció d'entrada/sortida *sscanf*

La funció *sscanf* també serveix per a manipular cadenes de caràcters: la funció és equivalent a la funció *scanf* per introduir dades per teclat però pel cas en què l'entrada és una cadena (per això la "s" inicial a *sscanf*). En particular, aquesta funció és útil quan el nombre d'elements a extreure de la cadena és fix. Cal esmentar, però, que no es recomana utilitzar aquesta funció per manipular cadenes: és una funció "poc segura" ja que el programa pot "petar" fàcilment si les dades a manipular no segueixen el patró previst.

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

#define MAXLEN 100

int main(void)
{
  int i, j, k, npunticoma, nsubcadenes;
  char **subcadena;
  char str[MAXLEN];

  if (fgets(str, MAXLEN, stdin) == NULL) {
    printf("No s'ha entrat cap cadena.\n");
  }

  // Comencem comptant el nombre de ";" que apareixen

  npunticoma = 0;
  while (str[i] != '\0') {
    if (str[i] == ';')
      npunticoma++;
    i++;
  }

  // El nombre de subcadenes és ...

  nsubcadenes = npunticoma + 1;

  // Ara reservem memoria per nsubcadenes cadenes de
  // caràcters de longitud MAXLEN

  subcadena = malloc(sizeof(char *) * nsubcadenes);
  for(i = 0; i < nsubcadenes; i++)
    subcadena[i] = malloc(sizeof(char) * MAXLEN);

  // I ara tornem a passar per la cadena original per
  // generar les subcadenes

  j = 0;
  i = 0;
  while (j < nsubcadenes) {
    k = 0;
    while ((str[i] != '\0') && (str[i] != ';')) {
      subcadena[j][k] = str[i];
      i++;
      k++;
    }
    subcadena[j][k] = '\0';
    j++;
    i++;
  }

  // Les imprimimim

  printf("Les subcadenes trobades són:\n");

  for(i = 0; i < nsubcadenes; i++)
    printf("Sucadena %d: %s\n", i+1, subcadena[i]);

  // Alliberem memoria

  for(i = 0; i < nsubcadenes; i++)
    free(subcadena[i]);

  free(subcadena);

  return 0;
}
```

Figura 14: Manipulació directa de la cadena, solució 1 (codi `exemple16.c`).

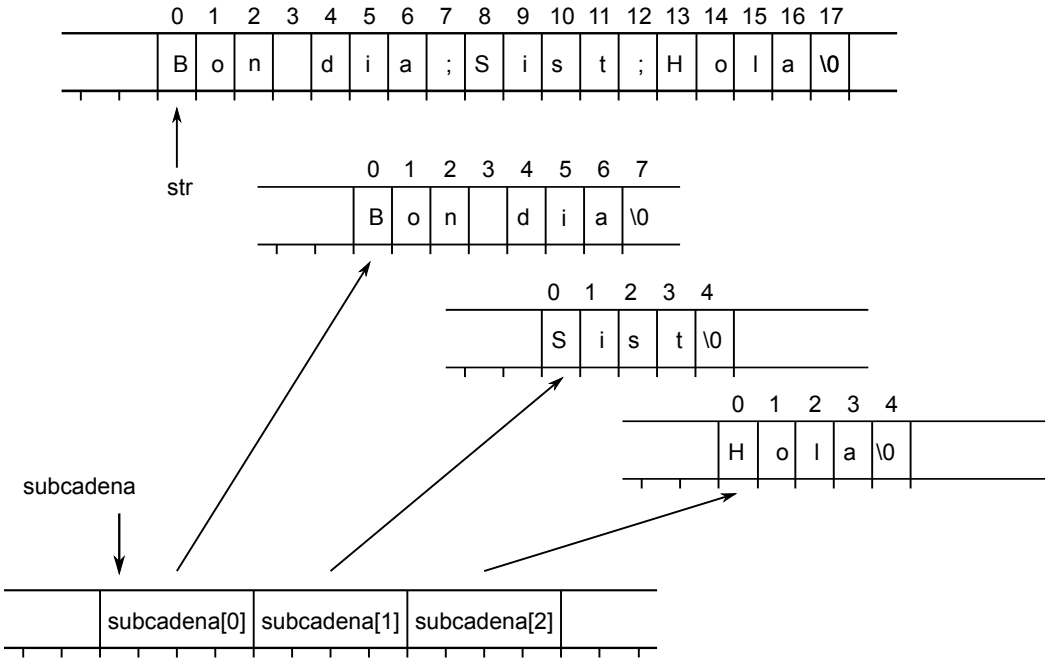


Figura 15: Disposició en memòria de les variables str i subcadena després de processar la cadena "Bon dia;Sist;Hola" utilitzant la solució 1 de la figura 14.

Per exemple, suposem que sabem que l'entrada té tres subcadenes. Una forma d'aconseguir separar l'entrada en tres subcadenes és la mostrada a la figura 18. En aquest exemple s'utilitza la directiva `%[^;]` per separar les cadenes. Sabeu quina funcionalitat té ? Podeu trobar la resposta fent `man sscanf` a un terminal. Serà difícil trobar una resposta directament al google.

Cal comentar que aquest exemple té una funcionalitat similar a la del primer exemple de la secció 5.3.1. En particular, la funció *sscanf* copia a les variables *cad1*, *cad2* i *cad3* les subcadenes que troba a *str*. No realitza pas una modificació de la cadena *str* original tal com es fa al segon exemple de la secció 5.3.1.

5.3.3 Utilització de la funció *strtok*

Una darrera funció que serveix per a extreure elements d'una cadena de caràcters és la funció *strtok*. En particular, aquesta funció és útil quan el nombre de elements a extreure de la cadena és variable. Aquesta funció s'aconsella molt més que la funció *sscanf*: és, de fet, més útil en el cas que les dades d'entrada puguin no seguir el patró previst.

Aquesta funció és equivalent al segon exemple de la secció 5.3.1, ja que modifica la cadena original. El prototip de la funció és

```
char *strtok(char *str, char *delim)
```

El funcionament de strtok és el següent:

1. La primera vegada que cridem a *strtok* per extreure'n els seus elements, la cridem passant a *str* la cadena a analitzar i a *delim* els delimitadors que separen el primer element del segon. En cridar-la la primera vegada la funció ens retorna un punter al primer element que ha extret de la cadena *str*.

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

#define MAXLEN 100

int main(void)
{
  int i, j, npunticoma, nsubcadenes;
  char **subcadena;
  char str[MAXLEN];

  if (fgets(str, MAXLEN, stdin) == NULL) {
    printf("No s'ha entrat cap cadena.\n");
  }

  // Comencem comptant el nombre de ";" que apareixen

  npunticoma = 0;
  while (str[i] != '\0') {
    if (str[i] == ';')
      npunticoma++;
    i++;
  }

  // El nombre de subcadenes és ...

  nsubcadenes = npunticoma + 1;

  // Ara reservem memoria per nsubcadenes punters a
  // cadenes de caràcters

  subcadena = malloc(sizeof(char *) * nsubcadenes);

  // I ara tornem a passar per la cadena original per
  // generar les subcadenes

  j = 0;
  i = 0;
  while (j < nsubcadenes) {
    subcadena[j] = str + i;
    while ((str[i] != '\0') && (str[i] != ';'))
      i++;
    str[i] = '\0';
    i++;
    j++;
  }

  // Les imprimimim

  printf("Les subcadenes trobades són:\n");

  for(i = 0; i < nsubcadenes; i++)
    printf("Sucadena %d: %s\n", i+1, subcadena[i]);

  // Alliberem memoria

  free(subcadena);

  return 0;
}
```

Figura 16: Manipulació directa de la cadena, solució 2 (codi `exemple17.c`).

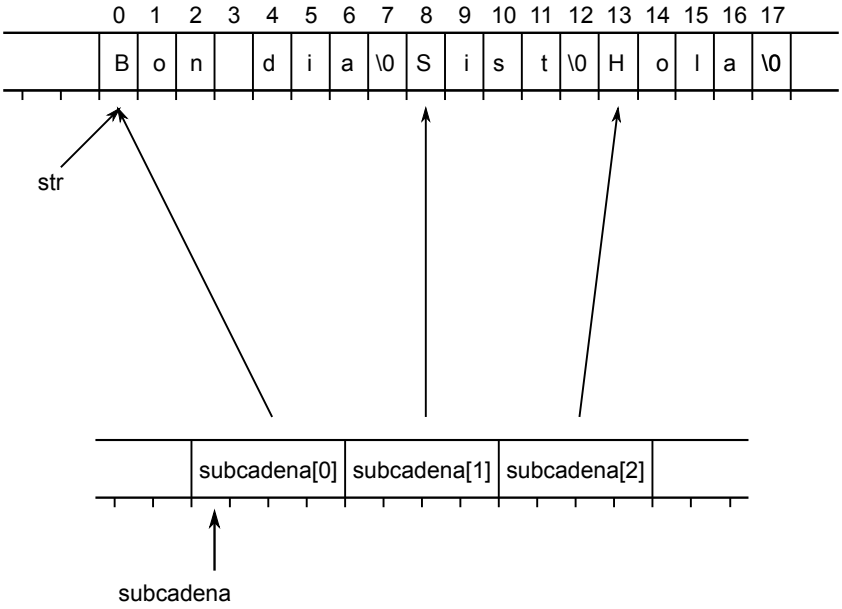


Figura 17: Disposició en memòria de les variables str i subcadena després de processar la cadena "Bon dia;Sist;Hola" utilitzant la solució 2 de la figura 16.

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

#define MAXLEN 100

int main(void)
{
  int nargs;
  char str[MAXLEN];
  char cad1[MAXLEN], cad2[MAXLEN], cad3[MAXLEN];

  if (fgets(str, MAXLEN, stdin) == NULL) {
    printf("No s'ha entrat cap cadena.\n");
  }

  nargs = sscanf(str, "%[^;];%[^;];%[^\n]",
      cad1, cad2, cad3);

  printf("%d\n", nargs);

  if (nargs != 3) {
    printf("Format incorrecte.\n");
    exit(1);
  }

  printf("Cadena 1: %s\n", cad1);
  printf("Cadena 2: %s\n", cad2);
  printf("Cadena 3: %s\n", cad3);

  return 0;
}
```

Figura 18: Manipulació de cadenes amb *sscanf* (veure codi `exemple18.c`).

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

#define MAXLEN 100

int main(void)
{
  int i;
  char str[MAXLEN];
  char *cad;

  if (fgets(str, MAXLEN, stdin) == NULL) {
    printf("No s'ha entrat cap cadena.\n");
  }

  i = 1;
  cad = strtok(str, ";");
  while (cad) {
    printf("Cadena %d: %s\n", i, cad);
    i++;
    cad = strtok(NULL, ";");
  }

  return 0;
}
```

Figura 19: Utilització de la funció strtok per separar cadenes (veure codi `exemple19.c`).

2. Per extreure el segon element de la cadena *str*, hem de tornar a cridar la funció *strtok* però amb *str* a NULL. A *delim* hem d'indicar els delimitadors que separen el segon element del tercer. En cridar-la la funció ens retornarà un punter al segon element que ha extret de la cadena *str*.

3. Així successivament podem treure tots els elements d'una cadena. La funció *strtok* retornarà NULL quan ja ho hi hagi més elements a extreure.

Vegem com s'escriu l'exemple anterior amb *strtok*. En aquest cas el programa és capaç d'extreure un nombre variable d'elements separats pel delimitador ";", veure figura 19.

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.