Estructures i tipus de dades en C

Lluís Alsedà

Departament de Matemàtiques
Universitat Autònoma de Barcelona
http://www.mat.uab.cat/~alseda

Versió 3.2 (març de 2019)

Subjecte a una llicència *Creative Commons Internacional de Reconeixement-NoComercial-CompartirIgual 4.0* (http://creativecommons.org/licenses/by-nc-sa/4.0/)

Continguts

Part I: Nous tipus de dades: `typedef` 1
Part II: Estructures 5
Part III: Organitzacions d'estructures 19

Els tres exemples d'organització d'estructures s'il·lustren amb la implementació d'una llista d'alumnes.

Part I: Nous tipus de dades: `typedef`

Índex

1. Ús de `typedef`
2. Exemples

Nous tipus de dades — `typedef`

El llenguatge **C** disposa d'una declaració anomenada `typedef` que permet la creació de nous tipus de dades.

Declaració

```
typedef tipus_a_definir sinonim;
typedef tipus_a_definir sinonim [dim];
typedef tipus_a_definir sinonim [dim1][dim2];
```

Exemples

```
typedef unsigned int enter;
typedef float Vector [100];
typedef char * string;

enter a, b = 3;
Vector v1, vv[73];
string s;
string *t;
```

`enter` és un sinònim d'`unsigned int`. N'hem creat dues instàncies: `a` (sense inicialitzar) i `b` (inicialitzat a `3`).

`Vector` Els seus objectes són vectors `float` de dimensió 100. Així `v1` és un vector `float` de dimensió 100 sense inicialitzar i `vv` és un vector de mida 73 de vectors float de mida 100 (és a dir una matriu 73×100). `v1` i `vv` s'utilitzen de forma estàndard **Exemple:** `i = v1 [19]; v1 [20-j] = a; vv[19][97] = b * v1[14] ;`

`string` Els seus objectes són apuntadors a `char`. `t` és un apuntador a `string` i, per tant, és un apuntador a apuntadors *char*.

Exemple d'ús

```
#include <stdio.h>

typedef unsigned char loopindex; // per bucles de 0 a 255
#define VectorDIM 15
typedef float Vector [VectorDIM];

void main() {
   register loopindex i;
   Vector A;

   for(i=0; i<VectorDIM; i++) A[i] = 2*i;
   for(i=0; i<VectorDIM; i++) printf("%d %f\n", i, A[i]);
}
```


Part II: Estructures

Índex

1. Introducció a les estructures
2. Declaració d'estructures
3. Operacions d'estructures
4. Un exemple bàsic complet
5. Herència: Una estructura pot contenir altres estructures
6. Pas d'estructures a funcions
   - Pas per valor
   - Pas per referència amb apuntadors

Introducció a les estructures

Una estructura és un conjunt d'elements heterogenis (anomenats camps) unificats sota un mateix nom, tipus i espai de memòria.

Les estructures permeten agrupar elements heterogenis en un tipus de dada que els englobi i relacioni.

Declaració d'estructures

Declaració estàndard

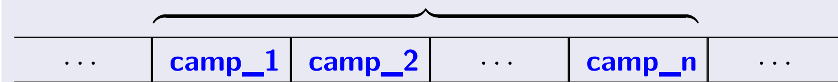
```
struct nom_estructura {
  tipus_1 camp_1;
  tipus_2 camp_2;
       ...
  tipus_n camp_n;
} ;
```

Millor creant un nou tipus de dades que correspon a aquesta estructura

```
typedef struct [nom_opcional] {
  tipus_1 camp_1;
  tipus_2 camp_2;
       ...
  tipus_n camp_n;
} nom_nou_tipus_de_dada ;
```

Internament, cada instància d'una estructura conté tots els camps de l'estructura.

una instància de `struct nom_estructura` o de `nom_nou_tipus_de_dada`



`sizeof(nom_estructura)` =
`sizeof(nom_nou_tipus_de_dada)` =
`sizeof(camp_1)` + `sizeof(camp_2)` + ⋯ + `sizeof(camp_n)`

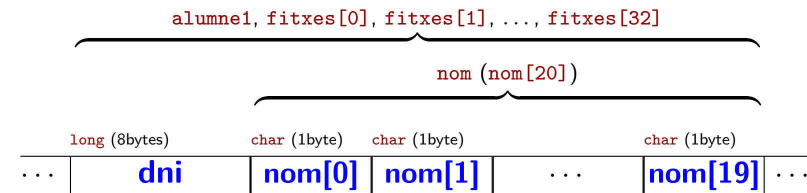
Exemple

```
struct fitxa_alumne {
   long dni;
   char nom [20];
} alumne1;

struct fitxa_alumne fitxes[33];
```

```
typedef struct [nom_opcional] {
   long dni;
   char nom [20];
} fitxa_alumne ;

fitxa_alumne alumne1, fitxes[33];
```



`sizeof(fitxa_alumne)` =
`sizeof(dni)` + `sizeof(nom[20])` =
`sizeof(long)` + 20 ∗ `sizeof(char)` = 28

Operacions d'estructures

Assignació

Les estructures es poden assignar entre elles, a l'igual que passa amb les variables normals (però no els vectors, ni les cadenes !).
Exemple: `fitxes[10]=fitxes[1];` copia la fitxa de l'alumne 1 al 10 i `fitxes[0]=alumne1;` copia la fitxa de `alumne1` al 0.

Inicialització

Es pot fer dintre el programa on es declara la variable estructura, posant tots els camps entre dues claus i separant-los per una coma.
Exemple:
`fitxa_alumne alumne1 = {35353535, "Pere Primer"};`

Operacions d'estructures

Accés a les dades internes d'una estructura

Per a accedir a una estructura de dades es pot emprar l'operador `.` ("punt").
La sintaxi és molt simple: `nom_estructura.nom_camp`
Exemple: `alumne1.dni = 35353535;`
Exemple: `fitxes[22].dni = 35353535;`

Mida d'una estructura

La mida de l'estructura s'obté emprant l'operador-funció ja conegut: `sizeof()`
Exemple: `sizeof(fitxa_alumne)` o també
Exemple: `int mida = sizeof(alumne1);`

Operacions d'estructures

Apuntadors i aritmètica d'apuntadors

Funcionen exactament igual que per els altres tipus de dades.

Exemple: `fitxa_alumne *f;` declara un apuntador `f` a dades (estructures) del tipus `fitxa_alumne`

Exemple: `f = fitxes + 22;` apunta a `fitxes[22]`. Així:

- `*f` és `fitxes[22]`.
- `f` apunta a la posició de memòria `fitxes + 22*sizeof(fitxa_alumne)`.

Operacions d'estructures

Accés a les dades internes d'un apuntador a una estructura

Per a accedir als camps d'una estructura des d'un apuntador es poden emprar els operadors `*` i `.` ja coneguts.

Exemple: `fitxes[22].dni = 35353535;`

és totalment equivalent a

`(*f).dni = 35353535;` i a

`(*(fitxes+22)).dni = 35353535;`

Hi ha una drecera per la construcció `* .`: l'operador `->`.

La sintaxi és: `apuntador_estructura->nom_camp`

Així els dos exemples anteriors queden, més simples:

Exemple: `f->dni = 35353535;` i

`(fitxes+22)->dni = 35353535;`

Un exemple bàsic complet

```
/* Trobar la distància entre dos punts */

/* declarem l'estructura punt del pla */
typedef struct { float x, y; } puntpla ;

float dist( puntpla a, puntpla b ){
   float x = a.x-b.x, y = a.y-b.y;
   return sqrt(x*x + y*y);
}

void main() {
  puntpla a, b; //declarem variables de l'estructrua creada

  printf("\tOPERANT AMB COORDENADES\n\n");

  printf("\tEntra les coordenades x, y del punt A: ");
  scanf("%f %f", &(a.x), &(a.y)); //definim les coords del punt a

  printf("\tEntra les coordenades x, y del punt B: ");
  scanf("%f %f",&b.x, &b.y); //definim la variable b

  printf("\n\tLa distancia entre els dos punts es %3.2f\n\n", dist(a,b));
}
```


Herència d'estructures: Una estructura pot contenir altres estructures

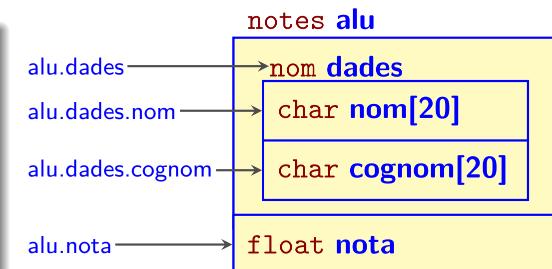
Les estructures es niuen declarant-les i incloent-les a dins d'una estructura com una dada més. Com amb els altres tipus de dades s'hi accedeix amb l'operador punt.

Declaracions

```
typedef struct {
  char nom[20], cognom[20];
} nom;

typedef struct {
  nom dades;
  float nota;
} notes;

notes alu;
```



Exemple d'herència d'estructures

```
#include <stdio.h>
#define NALUM 20

typedef struct { char nom[20], cognom[20]; } nom;
typedef struct { nom dades; float nota; } notes;

void main() { notes als[NALUM];
  printf("\tNOTES ALUMNES\n");

  for (int i=0; i < NALUM; i++) {
     printf("\tEntra nom, cognom i nota alumne %i: ", (i+1));
     fflush(stdin);
     scanf("%s%s%f", als[i].dades.nom,
                     als[i].dades.cognom,
                     &(als[i].nota)       );
   }

   printf("\n\tEL RESULTAT HA ESTAT:\n");
   for (i=0;i<NALUM;i++) {
       printf("\tAlumne %i:\n", (i+1));
       printf("\t\tNom: %s%s\n",als[i].dades.nom, als[i].dades.cognom);
       printf("\t\tNota: %3.1f\n", als[i].nota);
   }
}
```

 (CC BY-NC-SA)

Pas d'estructures a funcions

Com amb els tipus de dades bàsics el pas a funcions es pot fer per *valor* o per *referencia*.

En estructures grans és millor fer el pas per referència, ja que passar per valor podria omplir la memòria. Es veurà com es treballa amb dos exemples, que fan servir l'estructura `data` següent:

```
typedef struct {
   int dia, mes, any;
} data ;
```

Per a il·lustrar les dues maneres de passar estructures es definiran dues funcions:

- Calcular una nova data
- Imprimir data

És important entendre com es passen els paràmetres i quins són els resultats en cada cas.

 (CC BY-NC-SA)

Pas d'estructures a funcions: Pas per valor

No es modifica el contingut de les variables en les funcions.

El pas per valor

```
void main() {
   data d2, d1 = {21, 2, 2001};

   imprimir(d1, "d1");
   d2=calcular(d1);
   imprimir(d1, "d1");
   imprimir(d2, "d2");
}
```

Els resultats

```
d1 = 21/2/2001
d1 = 21/2/2001 // No s'ha modificat
d2 = 22/3/2002
```

El codi de les funcions

```
data calcular(data d) { d.dia ++; d.mes ++; d.any ++; return d; }

void imprimir (data d, char v[] ) {
    printf("%s = %i/%i/%i\n", v, d.dia, d.mes, d.any);
}
```

 (CC BY-NC-SA)

Pas d'estructures a funcions: Pas per referència amb apuntadors

Es modifica el contingut de les variables en les funcions.

El pas per referència amb apuntadors (es passa l'adreça de `d1`)

```
void main() {
   data d2, d1 = {21, 2, 2001};

   imprimir(d1, "d1");
   d2=calcular(&d1);
   imprimir(d1, "d1");
   imprimir(d2, "d2");
}
```

Els resultats

```
d1 = 21/2/2001
d1 = 22/3/2002 // Ara s'ha modificat
d2 = 22/3/2002
```

El codi de la funció `calcular`

```
data calcular(data * d) { d->dia ++; d->mes ++; d->any ++; return *d; }
```

 (CC BY-NC-SA)

Part III: Organitzacions d'estructures

Índex

1. Preliminars. Com s'ordena en **C**: la funció `qsort`
2. Vectors d'estructures.
3. Vectors d'apuntadors a estructures.
4. Cerques eficients per vectors i vectors d'apuntadors. Cerca binaria.
5. Una primera aproximació a les llistes enllaçades (estructures que apunten a altres estructures).

Els tres exemples d'organització d'estructures s'il·lustren amb la implementació d'una llista d'alumnes.

Plantejament

En el que segueix il·lustrarem la utilització d'organitzacions d'estructures i apuntadors a estructures per a gestionar dades. Com a mostra gestionarem una llista (d'alumnes) amb la que farem les següents operacions base:

En tots els cassos usarem la mateixa base (fitxer) de dades d'alumnes anomenada `dadesalumnes.dat`:

```
Pere;Barniol Serra;3.5;6.6
Joan;Lopez Garcia;8.4;5.5
Ramon;Marti Perez;6.3;7.4
Maria;Barniol Roca;9.5;7.4
```

- Creació del conjunt adequat d'estructures
- Càrrega de les estructures amb dades d'un fitxer
- Recorregut del conjunt d'estructures: Impressió seqüencial de les dades (per ordre de lectura) realitzant càlculs amb les estructures (per exemple la mitjana de les notes)
- Cerca d'estructures per cadenes de caràcters
- Cerca d'estructures amb criteris numèrics
- Ordenació de la llista usant diversos criteris (`qsort`)
- Eliminació d'un element

Organitzacions d'estructures

Usarem tres tipus d'organitzacions d'estructures: vectors, vectors d'apuntadors i llistes enllaçades.

Nota

La majoria d'operacions esmentades es poden fer sense usar cap estructura ni memòria, tractant les dades al llegir-les del fitxer. Ho fem així per il·lustrar els processos i operacions descrits anteriorment.

Preliminars. Com s'ordena en **C**: la funció `qsort`

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

La funció `qsort` ordena `num` blocs de mida `size` bytes del vector `base` de més petit a més gran fent servir l'algorisme *quicksort*. Exemple: Pàgina 27

Donats dos d'aquests blocs `A` i `B` cal decidir quin és el mes gran i, si estan en l'ordre incorrecte, intercanviar-los. La funció `qsort` determina quin dels dos blocs és més gran cridant:

```
compar(&A, &B);
```

Si la crida torna negatiu s'interpreta que $A < B$; si torna zero $A = B$ i, finalment, si la crida torna positiu $A > B$ i cal canviar `A` i `B` de lloc.

Un exemple de funció vàlida com a paràmetre `compar`

```
int compareMyType(const void * a, const void * b)
{
  if ( *(MyType*)a <  *(MyType*)b ) return -1;
  if ( *(MyType*)a == *(MyType*)b ) return 0;
  if ( *(MyType*)a >  *(MyType*)b ) return 1;
}
```

Exemple: Pàgina 25

Nota:

Dins de la funció `compar` cal forçar el tipus d'`a` i `b` (`(MyType*)a` i `(MyType*)b`) ja que ambdós paràmetres són apuntadors a `void` i, per tant, el **C** no sap com operar-los.

Observem que com més grans siguin `A` i `B` més tardara el procés de canvi. Per tant *cal evitar passar estructures a la funció* `qsort`.

Vectors d'estructures

Avantatges Simplicitat

Inconvenients
- Cada operació es realitza sobre tota la estructura
- Ralentitza el procés si les estructures són molt grans (`sizeof`)

Exemple: la llista d'alumnes amb vectors d'estructures

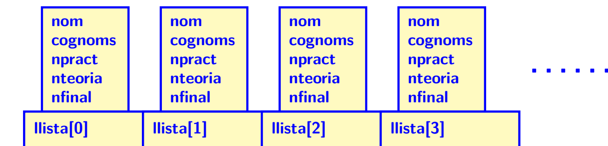
Declaració de l'estructura

```
typedef struct {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
} alumne;
```

El vector d'estructures

```
alumne *llista;

if ((llista = (alumne *) malloc (nalu*sizeof (alumne))) == NULL){
     fprintf (stderr, "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
     return 1;
}
```



Exemple: la llista d'alumnes amb vectors d'estructures

Funcions auxiliars; Declaracions i obertura del fitxer

```
#define NotaFinal(t,p) (0.6 * t + 0.4 * p)
void LlistaAlumne(alumne curr){
   printf("%s, %s. Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
         curr.cognoms, curr.nom, curr.nteoria, curr.npract, curr.nfinal );
}

int ComparaNota(const void * a, const void * b) {
    return (((alumne*)b)->nfinal - ((alumne*)a)->nfinal);
}

int main () {
  alumne *llista;
  FILE *fin;
  unsigned int i, nalu=0;
  char fitxer[]="dadesalumnes.dat";

// Obrim el fitxer
  if ((fin = fopen (fitxer, "r")) == NULL){
      fprintf (stderr, "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n", fitxer);
      return 1;
  }
```

D'entrada, ambdós paràmetres són apuntadors a `void` i llavors `*a` i `*b` no són cap estructura (en particular no són cap estructura del tipus `alumne`).
En conseqüència, `a->nfinal` i `b->nfinal` no tenen sentit.
Això s'arregla forçant el tipus d'`a` i `b` a `(alumne*)a` i `(alumne*)b`.

▸ Tornar a la pàgina 22

L'exemple amb vectors d'estructures (II)

```
// Contem el nombre d'alumnes. Caldria controlar que no és zero
   while (!feof (fin)) if(fgetc(fin) == '\n') nalu++;

// Creem llista d'estructures
   if ((llista = (alumne *) malloc (nalu*sizeof (alumne))) == NULL){
        fprintf (stderr, "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
        return 1;
   }

   rewind(fin);
   for(i=0 ; i < nalu ; i++) {
// Carreguem estructura amb dades de fitxer
     fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", llista[i].nom,
             llista[i].cognoms, &(llista[i].nteoria), &(llista[i].npract) );
     llista[i].nfinal = NotaFinal( llista[i].nteoria, llista[i].npract );
   }
   fclose (fin);

// Imprimim dades seqüencials i calculem les mitjanes
{ float m[3] = {0.0, 0.0, 0.0 };
  printf("\n***** Llistat d'alumnes segons l'ordre de lectura:\n");
  for(i=0 ; i < nalu ; i++) {
    printf("Alumne %u: ", i+1); LlistaAlumne(llista[i]);
    m[0] += llista[i].nteoria; m[1] += llista[i].npract; m[2] += llista[i].nfinal; }
  printf("Hi ha %u alumnes\n", nalu);
  printf("Mitjanes: Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",m[0]/nalu,m[1]/nalu,m[2]/nalu);
}
```

L'exemple amb vectors d'estructures (III)

```
/* Cerca d'estructures per cognom */
{  char cognom[]="Barniol";
   printf("\n***** Llistat d'alumnes amb cognom %s:\n", cognom);
   for(i=0 ; i < nalu ; i++)
       if(! strncmp(llista[i].cognoms, cognom, strlen(cognom))) LlistaAlumne(llista[i]);
}

/* Cerca d'estructures per nota */
{  float notamin=7.0;
   printf("\n***** Llistat d'alumnes amb nota mínima %f:\n", notamin);
   for(i=0 ; i < nalu ; i++) if(llista[i].nfinal >= notamin) LlistaAlumne(llista[i]);
}

// Ordenació. qsort. No recomanat: estem movent estructures.
  printf("\n***** Llistat d'alumnes segons l'ordre de nota final:\n");
  qsort(llista, nalu, sizeof(alumne), ComparaNota);
  for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);

/* Borrem l'alumne 2 per ordre de nota */
{  int borralu=1;
   nalu--; for(i=borralu ; i < nalu ; i++) llista[i]=llista[i+1];
   printf("\n***** Alumne 2 eliminat. Llistat sequencial dels alumnes que queden:\n");
   for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);
}
```

La instrucció `nalu-` fixa el rang del vector `llista` de `0` a `nalu_inicial-2` (sense alliberar memòria).
Així `llista[nalu]=llista[nalu_inicial-1]` quedarà fora de rang però ocupant memòria.
El bucle copia les estructures amb índex més gran o igual que `borralu+1` a la posició anterior; esborrant així la informació de `llista[borralu]`.
És crucial que `nalu` s'hagi disminuït abans del bucle per a evitar accedir al vector `llista` fora de rang.

▸ Tornar a la pàgina 22

El resultat de les operacions abans especificades és (per exemple):

```
***** Llistat d'alumnes segons l'ordre de lectura:
Alumne 1: Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
Alumne 2: Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Alumne 3: Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Alumne 4: Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Hi ha 4 alumnes
Mitjanes: Teor = 6.93; Pract = 6.72; Final = 6.84

***** Llistat d'alumnes amb cognom Barniol:
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66

***** Llistat d'alumnes amb nota mínima 7.000000:
Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66

***** Llistat d'alumnes segons l'ordre de nota final:
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74

***** Alumne 2 eliminat. Llistat sequencial dels alumnes que queden:
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
```


Vectors d'apuntadors a estructures

Avantatges

- Per realitzar operacions no cal moure tota l'estructura. Això és especialment útil a les ordenacions, en les que hi ha moviment massiu de dades
- Costa menys afegir (`realloc`) i esborrar elements
- Es pot usar per crear índexs addicionals de vectors d'estructures

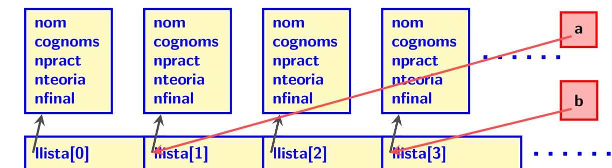
Inconvenients

- Una mica més complicat de programació
- Gasta una mica més de memòria

Exemple: la llista d'alumnes amb vectors d'apuntadors a estructures

Estructura (com abans)

```
typedef struct {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
} alumne;
```



Funcions auxiliars i declaracions (ometem l'obertura del fitxer)

```
#define NotaFinal(t,p) (0.6 * t + 0.4 * p)

void LlistaAlumne(alumne *curr){
   printf("%s, %s. Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
     curr->cognoms, curr->nom, curr->nteoria, curr->npract, curr->nfinal);
}

int ComparaCognoms(const void * a, const void * b) {
  return strcmp( (*((alumne**)a))->cognoms, (*((alumne**)b))->cognoms);
}

int main () {
  alumne **llista;
  FILE *fin;
  unsigned int i, nalu=0;
```

Ara usem l'operador `->` perquè `curr` és un apuntador.

`strcmp` compara dos "strings" i el seu valor de retorn coincideix amb el demanats per `qsort`: Negatiu (respectivament, zero o positiu) segons que el primer sigui més petit (respectivament, igual o més gran) que el segon.

Aquí `a` i `b` es declaren com apuntadors de tipus *void* però, en realitat, són apuntadors a algun element del vector `llista` (posem que `a` és un apuntador `void` a `llista[1]`).
Per accedir als cognoms de l'alumne apuntat per `llista[1]`, en primer lloc cal forçar el tipus d'`a` a un apuntador del mateix tipus que `llista` (que ha estat declarat com `alumne **llista`). Aquest apuntador és `(alumne**)a` (que apunta a `llista[1]` amb el tipus correcte).
Llavors, `*((alumne**)a) = llista[1]` i, per tant, s'accedeix al camp `cognoms` de l'alumne apuntat per `llista[1]` amb `(*((alumne**)a))->cognoms`.

L'exemple amb vectors d'apuntadors a estructures (II)

```
// Contem el nombre d'alumnes. Caldria controlar que no és zero
   while (!feof (fin)) if(fgetc(fin) == '\n') nalu++;

// Creem llista d'apuntadors a estructures
   if ((llista = (alumne **) malloc (nalu*sizeof (alumne *))) == NULL){
        fprintf (stderr, "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
        return 1;
   }

   rewind(fin);
   for(i=0 ; i < nalu ; i++) {
     if ((llista[i] = (alumne *) malloc(sizeof (alumne))) == NULL){
        fprintf (stderr, "\nERROR: No es possible crear alumne %u ...\n\n", i);
        return 1;
     }

// Carreguem estructura amb dades de fitxer
     fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", llista[i]->nom,
             llista[i]->cognoms, &(llista[i]->nteoria), &(llista[i]->npract) );
     llista[i]->nfinal = NotaFinal( llista[i]->nteoria, llista[i]->npract );
   }
   fclose (fin);

// Imprimim dades seqüencials i calculem les mitjanes
{ float m[3] = {0.0, 0.0, 0.0 };
  printf("\n***** Llistat d'alumnes segons l'ordre de lectura:\n");
  for(i=0 ; i < nalu ; i++) {
    printf("Alumne %u: ", i+1); LlistaAlumne(llista[i]);
    m[0] += llista[i]->nteoria; m[1] += llista[i]->npract; m[2] += llista[i]->nfinal; }
  printf("Hi ha %u alumnes\n", nalu);
  printf("Mitjanes: Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",m[0]/nalu,m[1]/nalu,m[2]/nalu);
}
```

Aquí `llista` és un vector d'elements `alumne *` (és a dir, d'apuntadors a `alumne`). Per això `llista` és un apuntador doble a `alumne`.

L'estructura `llista[i]`, ara, no existeix. L'hem de crear cada vegada.

`llista[i]` és un apuntador a `alumne`.

A partir d'ara usem l'operador `->` perquè `llista[i]` és un apuntador.

L'exemple amb vectors d'apuntadors a estructures (III)

```
/* Cerca d'estructures per cognom */
{  char cognom[]="Barniol";
   printf("\n***** Llistat d'alumnes amb cognom %s:\n", cognom);
   for(i=0 ; i < nalu ; i++)
        if(! strncmp(llista[i]->cognoms, cognom, strlen(cognom))) LlistaAlumne(llista[i]);
}

/* Cerca d'estructures per nota*/
{  float notamin=7.0;
   printf("\n***** Llistat d'alumnes amb nota mínima %f:\n", notamin);
   for(i=0 ; i < nalu ; i++) if(llista[i]->nfinal >= notamin) LlistaAlumne(llista[i]);
}

// Ordenació. qsort. Ara estem movent apuntadors (de mida molt més petita que les estructures)
  printf("\n***** Llistat d'alumnes per cognoms:\n");
  qsort(llista, nalu, sizeof(alumne *), ComparaCognoms);
  for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);

{  int borralu=1;
   free(llista[borralu]);
   nalu--; for(i=borralu ; i < nalu ; i++) llista[i]=llista[i+1];
   llista[nalu]=NULL;
   printf("\n***** Alumne 2 eliminat. Llistat alfabetic dels alumnes que queden:\n");
   for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);
}
```

Veure el diagrama de l'ordenació a la plana següent.

No és imprescindible. Per desapuntar la darrera posició del vector.

El procés d'esborrat és anàleg a l'anterior.
Ara l'estructura `*llista[borralu]` es pot eliminar ja que no s'usa; això millora el procés anterior (allà no podíem alliberar l'estructura que no s'utilitzava).
Per altra banda

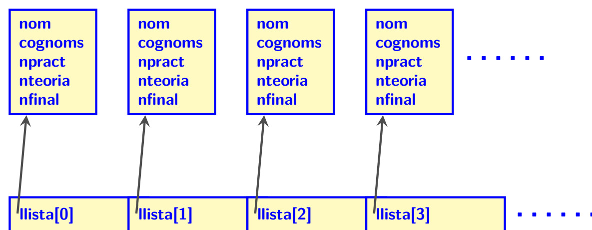
`llista[nalu] = llista[nalu_original-1]`

no s'utilitza però encara ocupa memòria (solament la d'un apuntador).

L'exemple amb vectors d'apuntadors a estructures: Idea de l'ordenació

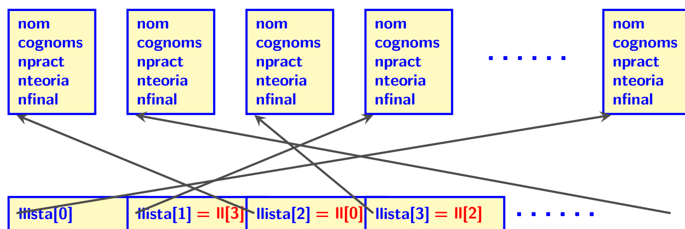
Amb l'ajut de `qsort` passem de:

situació de `llista` abans d'ordenar



a

situació de `llista` després d'ordenar



Cerques eficients per vectors i vectors d'apuntadors

Avantatges

- La cerca seqüencial és *terriblement* ineficient amb grans quantitats de dades ($\mathcal{O}(n)$).
- Les cerques binàries incrementen espectacularment l'eficiència ($\mathcal{O}(\log_2(n))$).

Inconvenients

- Programació més complicada
- Solament funciona en dades ordenades amb el mateix criteri de busca

Cerca binària

En informàtica, una *cerca binària* o *algorisme de busca de bisecció* troba la posició d'un valor especificat d'entrada (la *clau* de cerca) dins d'un *vector ordenat en ordre creixent respecte dels valors de la clau* És un *algorisme dicotòmic del tipus divideix i venceràs*. Consisteix a aplicar l'algorisme de bisecció a la cerca de valors. Cal tenir les dades ordenades i indexades.

La cerca binària redueix a la meitat el nombre d'elements a comprovar a cada iteració, de manera que la localització d'una clau (o la determinació de la seva absència) es fa en temps logarítmic ($\log_2(n)$).

continua...

Cerca binària (cont.)

A cada pas, l'algoritme compara el valor de la clau de cerca amb el valor de de l'element mitjà del vector. Si les claus coincideixen, llavors s'ha trobat un dels elements buscat i es retorna el seu índex o posició dins del vector. Altrament, si la clau de cerca és menor que la clau de l'element mitjà, llavors l'algorisme repeteix la seva acció sobre el sub-vector a l'esquerra de l'element mitjà o, si la clau de cerca és més gran, en el sub-vector a la dreta de la clau de cerca.

Si després d'algunes iteracions l'interval de vector que queda per buscar és buit, llavors la clau no pertany al vector i es retorna una indicació especial `no_trobat`.

continua...

Cerca binària (cont.)

Exemple:

- La llista que volem cercar és `L = {1 3 4 6 8 9 11}`
- El valor que volem trobar és: `clau = 4`
- Iteracions de l'algorisme:
  1. Inicialitzem l'interval de cerca:
     `inici = 1` (`L[1] = 1`); `final = 7` (`L[7] = 11`).
  2. `punt mig = 4` (`L[4] = 6`).
     Comparar `clau` amb `L[4] = 6`: `clau` **és menor**.
     En funció d'això redefinim l'interval de cerca:
     `inici = 1` (`L[1] = 1`); `final = 3` (`L[3] = 4`).
  3. `punt mig = 2` (`L[2] = 3`).
     Comparar `clau` amb `L[2] = 3`: `clau` **és més gran**.
     En funció d'això tornem a redefinir l'interval de cerca:
     `inici = 3`; `final = 3` (`L[3] = 4`).
  4. `punt mig = 3` (`L[3] = 4`).
     Comparar `clau` amb `L[3] = 4`: **Són iguals**.
     Ja hem acabat i retornem `index = 3`.

continua...

Cerca binària (cont.)

Algorisme (de cerca binària amb interval com a rang de cerca)

```
procedure BINARY__SEARCH(Vector, Vlen, clau)
    imin ← 0; imax ← Vlen − 1;        ▷ Inicialització de l'interval de cerca (tot el vector)
    while imax ≥ imin do              ▷ Iteració: continuar buscant mentre [imin, imax] no està buit
        imid ← (imin+imax)/2;         ▷ Punt mig
        if Vector[imid] = clau then
            return imid;              ▷ imid és l'element buscat. Retornem el seu índex
        else if Vector[imid] < clau then   ▷ clau és a la dreta d'imid
            imin ← imid + 1           ▷ El nou interval serà [imid+1,imax]
        else                          ▷ clau és a l'esquerra d'imid
            imax ← imid − 1           ▷ El nou interval serà [imin,imid-1]
        end if
    end while
    return −1;                        ▷ No s'ha trobat la clau. Retornem -1 que no és un index vàlid.
end procedure
```

continua...

Cerca binària (cont.)

Observacions

- *Solament funciona amb dades ordenades amb el mateix criteri que la busca*.
- A cada iteració de l'algorisme `imax` és més petit o igual que a la iteració anterior i `imin` és més gran o igual que a la iteració anterior. Per tant, dins del bucle (`imax` $\geq$ `imin`), $0 \leq$ `imin` $\leq$ `imax` $\leq$ `Vlen` $- 1$. En particular, mentre estem dins del bucle, ni `imin` ni `imax` poden sortir del rang del vector. Això justifica la sortida la sortida d'error fora del bucle.

Exemples d'implementació

La funció de cerca binaria numèrica per un vector `unsigned long` ordenat de petit a gran

```
unsigned long BinarySearch(unsigned long key, unsigned long *list, unsigned long lenlist){
  register unsigned long start=0UL, afterend=lenlist, middle;
  register unsigned long try;

  while(afterend > start){ middle = start + ((afterend-start-1)>>1); try = list[middle];
    if (key == try) return middle;
    else if ( key > try ) start=middle+1;
    else afterend=middle;
  }
  return ULONG_MAX;
}
```

Com es crida la funció

```
unsigned long index = BinarySearch(key, llista, numElements);
if(index == ULONG_MAX) printf("ERROR: Alumne no trobat\n");
else printf("Key %lu has index %lu\n", key, index);
```

continua...

Exemples d'implementació (cont.)

Observacions

- **`afterend` en lloc d'`end`**
  Mantenir la posició següent al final de l'interval en lloc de la posició final té diversos avantatges:
  - **`while(afterend > start)`**
    Permet canviar una comparació `>=` complicada per una de més simple: `>`.
  - **`afterend=middle;`**
    Evita fer `end = middle-1`, que pot donar negatiu (desastrós donat que usem `unsigned`'s).
- **`middle = start + ((afterend-start-1)»1);`**
  Si `start`,`afterend` $>$ `ULONG_MAX/2` tenim un overflow al calcular
  `middle = (start + afterend -1)/2` (`start+afterend-1` $>$ `ULONG_MAX`).
  Notem que la fórmula `middle = start + ((afterend-start-1)/2)` evita aquest problema (`(afterend-start-1)»1` és solament un forma ràpida de dividir un enter per 2).
- **`try = list[middle];`**
  Per evitar accedir a la memòria dues vegades
- **`return ULONG_MAX;`**
  No podem tornar `-1` com a codi d'error ja que retornem `unsigned`'s. En lloc de `-1` triem l'`unsigned long` més gran. Això implica que no podem usar vectors d'aquesta mida (no és gaire preocupant donat que `ULONG_MAX` $= 2^{64} - 1 = 18.446.744.073.709.551.615$).

continua...

Exemples d'implementació (cont.)

La funció de cerca binaria numèrica per un vector d'estructures ordenat de gran a petit

```
long BuscaNotaCercaBinaria(alumne *llista, unsigned int nalu, const float nota){
  register unsigned int start=0UL, afterend=nalu, middle;
  register float try;

  while(afterend > start){ middle = start + ((afterend-start-1)>>1); try = llista[middle].nfinal;
    if (fabs(try - nota) <= 5.0e-3) return (long) middle;
    else if ( nota < try ) start=middle+1;
    else afterend=middle;
  }
  return -1L;
}
```

Com es crida la funció

```
long index = BuscaNotaCercaBinaria(llista, nalu, 7.24);
if(index == -1) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(llista[index]);
```

Observacions

- **`fabs(try - nota) <= 5.0e-3`**
  Recordem que en punt flotant no té sentit fer comparacions `try == nota`.
- **`return -1L;`**
  Noteu que ara retornem `long` (amb signe — `long BuscaNotaCercaBinaria(alumne ...)`). Per tant podem tornar valors negatius (fora del rang d'índexs d'un vector). No hi ha cap problema en fer això ja que `UINT_MAX` = 4294967295U i `LONG_MAX` = 9223372036854775807L. Per tant, una variable `long` pot contenir qualsevol `unsigned int`.

continua...

Exemples d'implementació (cont.)

La funció de cerca binaria alfabètica per un vector d'apuntadors a estructures ordenat de petit a gran

```
long BuscaNomCercaBinaria(alumne **llista, unsigned int nalu, const char *cognom){
    register unsigned int start=0UL, afterend=nalu, middle;
    register unsigned short cognomlen = strlen(cognom);
    register int rescom;

  while(afterend > start){ middle = start + ((afterend-start-1)>>1);
    rescom = strncmp(llista[middle]->cognoms, cognom, cognomlen);
    if (rescom == 0) return middle;
    else if ( rescom < 0 ) start=middle+1;
    else afterend=middle;
  }
  return -1;
}
```

Per estalviar *moltes vegades* aquest calcul dins del bucle.

Per evitar accedir a la memòria dues vegades.

La crida a la funció

```
long index = BuscaNomCercaBinaria(llista, nalu, "Lopez");
if(index == -1) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(llista[index]);
```


El C també té la cerca binària de llibreria: bsearch

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

La funció `bsearch` busca al vector `base` que està format per `nmemb` blocs de mida `size` bytes *ja ordenat amb el mateix criteri de la busca* fent servir l'algorisme de *cerca binària*. La funció torna un apuntador a l'element buscat (*no el seu índex dins el vector*) o `NULL` si no l'ha trobat.

Donats l'apuntador a `key` i un bloc `B` del vector, cal decidir si són iguals (i per tant si hem trobat l'element buscat) o, en cas contrari, quin és el mes gran. La funció `bsearch` aconsegueix aquesta informació cridant:

`compar(key, &B);`

Si la crida torna negatiu s'interpreta que `*key` < `B`; si torna zero `*key` = `B` i, finalment, si la crida torna positiu `*key` > `B`.

Exemple de funció compar

```
int comparMyWay(const void * key, const void * b)
{
  if ( *(MyType*)key <  *(MyType*)b ) return -1;
  if ( *(MyType*)key == *(MyType*)b ) return 0;
  if ( *(MyType*)key >  *(MyType*)b ) return 1;
}
```

Nota:

Dins de la funció `compar` cal forçar el tipus de `key` i `b` (`(MyType*)key` i `(MyType*)b`) ja que ambdós paràmetres són apuntadors a `void` i, per tant, el C no sap com operar-los.

continua...

El C també té la cerca binària de llibreria: bsearch (cont.)

Ús de la funció bsearch per un vector d'estructures ordenat numèricament de gran a petit

```
float nota=7.24;
alumne *alu = bsearch(&nota, llista, nalu, sizeof(alumne), ComparaKeyNota);
if(alu == NULL) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(*alu);
```

La funció de comparació

```
int ComparaKeyNota(const void * key, const void * b) {
  if ( fabs(((alumne*)b)->nfinal - *(float*)key) <= 5.0e-3 ) return 0;
    else if ( *(float*)key <  ((alumne*)b)->nfinal) return 1;
    else return -1;
}
```

Ús de la funció bsearch per un vector d'apuntadors a estructures ordenat alfabèticament de petit a gran

```
char cognom[]="Lopez";
alumne **alu = bsearch(cognom, llista, nalu, sizeof(alumne*), ComparaKeyCognom);
if(alu == NULL) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(*alu);
```

La versió de `LlistaAlumne` per vectors d'apuntadors accepta un apuntador a alumne (`alumne *`).

La funció de comparació

```
int ComparaKeyCognom(const void * key, const void * b) {
    return strncmp(key, (*((alumne**)b))->cognoms, strlen(key));
}
```


Comparativa d'eficiència i optimització del codi

Temps de CPU de 26.000.000 de cerques a una llista d'unsigned long

Mida Llista	BS	BS(O)	LS	LS(O)	CBS	CBS(O)
1024	1.37	1.20	8.63	0.00	1.97	0.98
2048	1.52	1.33	16.91	0.00	2.14	1.09
4096	1.67	1.44	33.71	0.00	2.34	1.19
8192	1.92	1.65	69.06	0.00	2.62	1.41
16384	2.12	1.86	137.65	0.00	2.91	1.60
32768	2.42	2.16	277.00	0.00	3.26	1.85
65536	3.07	2.72	551.69	0.00	4.00	2.41
131072	3.66	3.30	1105.65	0.00	4.61	2.91
262144	3.98	3.97	2211.65	0.00	6.40	3.43
2097152		12.24		0.00		12.05
2147483648		31.05		0.00		30.93

Llegenda

- **BS** cerca binària
- **BS(O)** cerca binària amb compilació optimitzada (-O3)
- **LS** cerca seqüencial
- **LS(O)** cerca seqüencial amb compilació optimitzada (-O3)
- **CBS** cerca binària amb la funció `bsearch`
- **CBS(O)** cerca binària amb la funció `bsearch` amb compilació optimitzada (-O3)

Algorisme de cerca seqüencial (LS) usat

```
unsigned long LinearSearch(unsigned long key, unsigned long *list, unsigned long lenlist){
   register unsigned long s;
   for(s=0; s < lenlist; s++) if (key == list[s]) return s;
   return ULONG_MAX;
}
```

Observació

Notem que la cerca seqüencial és molt menys eficient que la binària (com està previst) *amb el codi no optimitzat*. Això dóna un paper crucial a la optimització de codi del `gcc`, que canvia completament l'estratègia de cerca

continua...

Comparativa d'eficiència i optimització del codi (cont.)

Temps de CPU de cerques a un vector d'estructures que conté el mapa de carreteres de Catalunya o d'Espanya

Mapa	# nodes	# cerques	Clau	BS	BS(O)	LS	LS(O)
Catalunya	3.472.620	3.984.679	`unsigned long`	0.90	0.68	13326.00	0.15
Catalunya	3.472.620	3.984.679	`char id[11]`	1.85	1.33	29675.60	0.09
Espanya	23.895.681	26.060.004	`unsigned long`	5.16	3.87	no es pot	0.99
Espanya	23.895.681	26.060.004	`char id[11]`	12.59	8.15	no es pot	0.58

Llegenda: **BS** = cerca binària; **BS(O)** = cerca binària amb compilació optimitzada (`-O3`); **LS** = cerca seqüencial; **LS(O)** = cerca seqüencial amb compilació optimitzada (`-O3`).

Nota: L'estructura amb l'`id` de tipus `unsigned long` té 32`Bytes`; L'estructura amb l'`id` de tipus `char [11]` té 40`Bytes`.

Algorismes de cerca seqüencial (LS) usats en aquest cas

```
unsigned long LinearSearch(unsigned long id, node *nodvect, unsigned long nnod){
   register unsigned long s;
   for(s=0; s < nnod; s++) { if (id == nodvect[s].id) return s; }
   return ULONG_MAX;
}
unsigned long LinearSearchStr(char *id, node *nodvect, unsigned long nnod){
   register unsigned long s;
   for(s=0; s < nnod; s++) { if (strcmp(id, nodvect[s].id) == 0) return s; }
   return ULONG_MAX;
}
```


Conclusions

- Per un nombre de dades gran (diguem més gran que 64) i el codi sense optimitzar cal imperativament usar cerca binària.
- Per altra banda és millor optimitzar i, en aquest cas, la cerca seqüencial és més senzilla i ràpida.

Observació

La comparativa s'ha fet amb un ordinador amb processador Intel i7-4770K @ 3.50GHz QuadCore amb 32Gb de RAM DDR3 @ 1867MHz.

Una primera aproximació a les llistes enllaçades; estructures que apunten a altres estructures

Avantatges

- Usen el mínim de memòria
- Organització de dades estructurada
- La programació és independent del número de dades
- Eficiència
- És fàcil afegir i esborrar elements

Inconvenients

- Programació més complicada
- Cerca eficient molt difícil. És gairebé obligada la cerca seqüencial.
- Ordenació més complicada. Es pot simplificar afegint un "vector index" auxiliar d'apuntadors a les estructures de la llista.

Que és una llista enllaçada?

Una llista enllaçada és una col·lecció d'elements disposats seqüencialment que permet la inserció i eliminació d'elements en qualsevol lloc de la seqüència.

En una llista enllaçada podem inserir i eliminar nodes sense haver de conèixer la mida de la llista i de les dades. L'eina que permet el funcionament d'aquest mecanisme és la de reservar i alliberar els espais de memòria dels nodes mitjançant l'assignació dinàmica.

En l'us de les llistes dinàmiques cal sempre tenir present fer la reserva de memòria com a pas previ a la creació d'un node i d'alliberar-la en el moment en què un node desapareix.

Les llistes enllaçades poden ser simples i dobles. En una llista simple, de cada node solament podem saltar al següent. Per tant, solament podem recórrer la llista endavant. En una llista doblement enllaçada, de cada node podem saltar al posterior i a l'anterior. Per tant, podem recórrer la llista tan endavant com endarrere.

En el que segueix, per simplicitat, solament considerarem les llistes enllaçades simples.

Introducció a les llistes enllaçades

Considerem la següent millora de l'exemple `alumne` anterior. Volem crear una llista ordenada del 4 alumnes B, D, C, A. Suposem que l'ordre és `B -> D -> C -> A` i que les estructures ja tenen les dades (llegides d'un fitxer o de teclat).

Declaració de l'estructura

```
typedef struct ALUDATA {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
  struct ALUDATA *seg;
} alumne;
```

Nota

Si en una estructura volem incloure un apuntador a estructures del mateix tipus no es pot usar el tipus creat per `typedef`. Cal usar la declaració `struct 'namespace'` com hem fet a la declaració anterior.

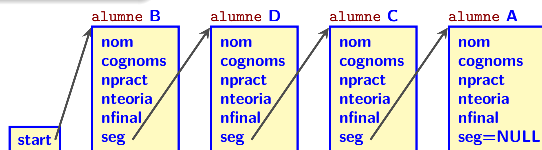
El codi

```
alumne A, B, C, D, *start=NULL;

start = &B;
B.seg = &D;
D.seg = &C;
C.seg = &A;
A.seg = NULL;
```

nota

`start` és fonamental. Necessitem l'adreça del primer element (un punt d'entrada).



Recorregut seqüencial d'una llista

Volem llistar el nom i cognoms dels alumnes en ordre alfabètic.

Amb un bucle `while`

```
alumne *curr; /* No ens volem 'carregar' l'apuntador 'start' */
curr=start;
while(curr){
   printf("Nom: %s ; Cognoms: %s\n", curr->nom, curr->cognoms);
   curr=curr->seg;
}
```

Amb un bucle `for`

```
alumne *curr; /* No ens volem 'carregar' l'apuntador 'start' */
for( curr=start ; curr ; curr=curr->seg )
    printf("Nom: %s ; Cognoms: %s\n", curr->nom, curr->cognoms);
```

Recordem que

```
start = &B; B.seg = &D; D.seg = &C; C.seg = &A; A.seg = NULL;
```


Exemple: la llista d'alumnes amb llistes enllaçades

Declaració de l'estructura

```
typedef struct ALUDATA {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
  struct ALUDATA *seg;
} alumne;
```

Funcions auxiliars i declaracions (ometem l'obertura del fitxer)

```
#define NotaFinal(t,p) (0.6 * t + 0.4 * p)

void LlistaAlumne( alumne *curr){
   printf("%s, %s. Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
          curr->cognoms, curr->nom,
          curr->nteoria, curr->npract, curr->nfinal );
}
int ComparaCognoms(const void * a, const void * b) {
  return strcmp( (*((alumne**)a))->cognoms, (*((alumne**)b))->cognoms );
}

int main () {
  alumne *start = NULL,
         *curr=NULL,
         *aux;
  FILE *fin;
  unsigned int nalu=0;
```

Aqui valen els mateixos comentaris que en el cas dels vectors d'apuntadors a estructures (veure la pàgina 30/59).

L'exemple amb llistes enllaçades (II)

```
   while (!feof (fin)) {
     if ((aux = (alumne *) malloc (sizeof (alumne))) == NULL){
        fprintf (stderr, "\nERROR: No es possible crear alumne ...\n\n");
        return 1; }

// Carreguem estructura amb dades de fitxer
     fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", aux->nom,
             aux->cognoms, &(aux->nteoria), &(aux->npract) );
     aux->nfinal = NotaFinal( aux->nteoria, aux->npract );
     aux->seg = NULL;
     if (curr) curr->seg = aux; else start = aux;
     curr = aux;
  } ; fclose (fin);

// Imprimim dades seqüencials i calculem les mitjanes
{ float m[3] = {0.0, 0.0, 0.0 };
  printf("\n***** Llistat d'alumnes segons l'ordre de lectur
  for( curr=start; curr ; curr=curr->seg ){ nalu++;
    printf("Alumne %u: ", nalu); LlistaAlumne(curr);
    m[0] += curr->nteoria; m[1] += curr->npract; m[2] += curr->nfinal;
  }
  printf("Hi ha %u alumnes\n", nalu);
  printf("Mitjanes: Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",m[0]/nalu,m[1]/nalu,m[2]/nalu);
}

/* Cerca d'estructures per cognom */
{  char cognom[]="Barniol";
   printf("\n***** Llistat d'alumnes amb cognom %s:\n", cognom);
   for(curr=start ; curr ; curr=curr->seg)
       if(! strncmp(curr->cognoms, cognom, strlen(cognom))) LlistaAlumne(curr);
}
```

Creem l'estructura a carregar. `aux` és un apuntador a `alumne`.

Usem l'operador -> perquè `aux` és un apuntador.

El següent del darrer node no existeix. L'apuntador *s'ha d'inicialitzar* a `NULL`

Aquest codi afegeix el node `*aux` a la llista. La instrucció `curr = aux;` defineix `curr` com un apuntador al darrer node en el que s'han carregat dades. Si `curr=NULL` estem carregant dades al primer node. En aquest cas l'`if` es fals i s'executa `start = aux;` que inicialitza correctament `start`. Si `curr != NULL`, l'`if` es veritat i s'executa `curr->seg = aux;` que defineix `*aux` com a node següent de `*curr`.

Recorregut seqüencial de la llista; vist abans. Podríem usar `while`.

Recorregut seqüencial de la llista; vist abans. Podríem usar `while`.

L'exemple amb llistes enllaçades (III)

```
/* Cerca d'estructures per nota*/
{  float notamin=7.0;
   printf("\n***** Llistat d'alumnes amb nota mínima %f:\n", notamin);
   for(curr=start ; curr ; curr=curr->seg) if(curr->nfinal >= notamin) LlistaAlumne(curr);
}

// Ordenació. qsort. Amb l'ajut d'un vector axiliar d'index.
{  alumne **llista; unsigned int i;
   if ((llista = (alumne **) malloc (nalu*sizeof (alumne *))) == NULL){
       fprintf (stderr, "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
       return 1;
   }
   for(llista[0]=start, i=1 ; i < nalu ; llista[i]=llista[i-1]->seg, i++ )
   qsort(llista, nalu, sizeof(alumne *), ComparaCognoms);
   start = llista[0]; llista[nalu-1]->seg=NULL;
   for(i=1 ; i < nalu ; i++) llista[i-1]->seg = llista[i];
   free(llista);
}
   printf("\n***** Llistat d'alumnes per cognoms:\n");
   for(curr=start ; curr ; curr=curr->seg) LlistaAlumne(curr);

/* Borrem l'alumne 3 per ordre alfabetic. Volem que aux apunti a l'alumne que es vol borrar */
{  int i, borralu=2;
   if(borralu == 0) { aux=start; start=aux->seg; }
   else { borralu--;
          for(i=0,curr=start; i < borralu ; i++,curr=curr->seg) ;
          aux = curr->seg ; curr->seg=aux->seg;
   } ; free(aux);
   printf("\n***** Alumne 3 eliminat. Llistat alfabetic dels alumnes que queden:\n");
   for(curr=start ; curr ; curr=curr->seg) LlistaAlumne(curr);
}
```

Per a crear un vector d'apuntadors als elements de la llista (com abans). Servirà de vector auxiliar d'índex.

Veure els diagrames de la plana següent

El vector `llista` ja no és necessari. Ja hem reconstruït els enllaços.

Obvi

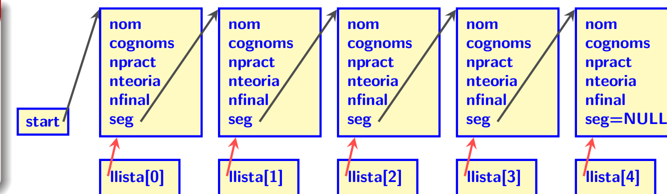
Veure la plana 57: Borrat d'un element

Obvi

La llista enllaçada al moment de la ordenació

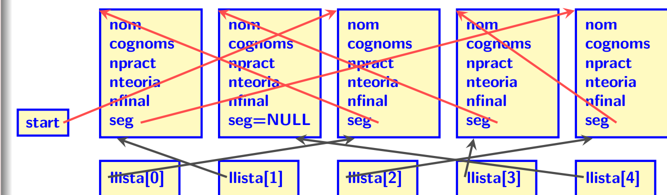
Abans de l'ordenació

La informació sobre la posició en memòria de cada estructura ve donada per l'apuntador `seg` de l'estructura anterior (o `start`). Per inicialitzar `llista` cal fer `llista[0] := start` i, seqüencialment, `llista[i+1] := llista[i]->seg`



Després de l'ordenació

La posició en memòria de cada estructura (en l'ordre desitjat) ve donada per `llista[i]`. Per a recuperar els enllaços seqüencials de la llista cal redefinir: `start := llista[0]` i, seqüencialment, `llista[i]->seg := llista[i+1]` per $i = 0, 1, \ldots, \text{nalu} - 2$. Observem que `llista[nalu-1]` és la darrera estructura i per tant. `llista[nalu-1]->seg=NULL`.



Borrat d'un element

Pas 1:

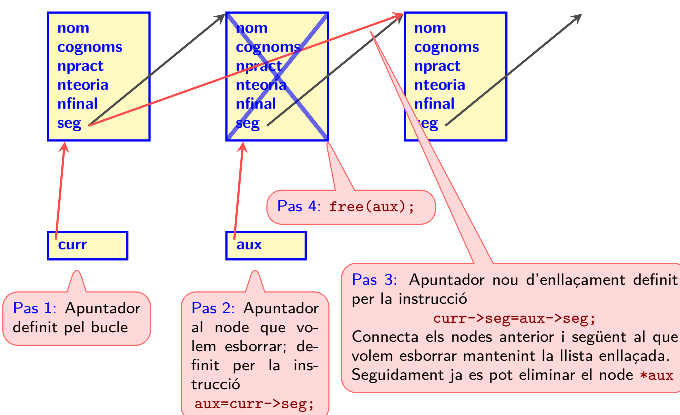
Al final del bucle, l'apuntador `curr` apunta al node anterior al que volem esborrar (gracies a que hem fet `borralu-`)

Quan `borralu` no és zero

```
borralu--;
for(i=0,curr=start; i < borralu ; i++,curr=curr->seg) ;
```

El procés de re-enllaçament de la llista i esborrat del node

```
aux = curr->seg ;
curr->seg=aux->seg;
free(aux);
```



Més sobre llistes

Exercici

Que cal fer per a poder llistar la llista en ordre invers?

Exercici

Declarar un nou apuntador a `alumne` anomenat `end` i modificar el codi anterior de manera que `end` apunti sempre al darrer node de la llista (solament és útil si s'ha fet l'exercici anterior).

Un estudi més sistemàtic de les operacions bàsiques amb llistes enllaçades com: inserció de nodes, reordenacions, recorregut de la llista endarrere i altres es farà en el context de les llistes com a *Tipus Abstracte de Dades (TAD)*.

Fi. Bona sort amb les estructures