Université d'Auvergne
IUT Département informatique
Première année 2012-2013
Période 2

Structures
Allocation dynamique Fichiers binaires
Tris Listes

Université d'Auvergne IUT Département informatique
1ère année période 2 Structures Tris Listes

BOURDIEN
Rémi
G3

1. Spécification d'un type synonyme : typedef

typedef permet de donner un nom à un type pour faciliter l'écriture des programmes .

On écrit :

typedef suivi d'une déclaration de variable

typedef float Note;

Le type ainsi défini porte le nom de la variable : Note

Par convention, un nom de type commencera par une majuscule.

On utilisera souvent typedef pour déclarer des types correspondant à des tableaux ou des structures.

Exemples:

- Déclarer un type appelé Entier pour les int

  typedef int Entier ;

  Entier i, j ; sera equivalent à: int i, j ;

- Déclarer un type appelé Nom pour les chaînes de 21 caractères ;

  typedef char Nom[21] ;

  Nom nom1, nom2 ;

  est équivalent à

  char nom1[21], nom2[21] ;

  Pour déclarer un tableau de 10 chaînes de 21 car, on écrira :

  Nom tnom[10] ;

  est équivalent à

  char tnom[10][21] ;

2. Enumération : enum

enum permet de définir des constantes entières.

enum {NomCste0, NomCste1, ,... } n, m ;

enum {NomCste0 = entier0, NomCste1 = entier1, ...} ;

Par défaut, les constantes de la liste ont les valeurs 0, 1, 2... ou la valeur précédente plus 1.

Nous utiliserons cette possibilité pour améliorer la lisibilité des programmes.

Exemple :

enum {BLEU, BLANC, ROUGE} ;

c = BLANC ; est équivalent à c = 1 ;

enum {BLEU=10, BLANC=20, ROUGE=40} ;

c = BLANC ; est équivalent à c = 20 ;

enum {JANVIER=1, FEVRIER, MARS...} m ;

m = MARS ; est équivalent à m = 3 ;

enum déclare un entier m qui peut prendre n'importe quelle valeur

On peut définir un type énuméré qui sera utilisé pour déclarer des variables :

typedef enum {JANVIER=1, FEVRIER, MARS...} TypeMois;

TypeMois m1, m2 ;

On indique ainsi que m1 et m2 sont des mois qui prendront les valeurs 1, 2,... ou JANVIER, FEVRIER, MARS...

Attention, le système n'interdit pas les autres valeurs

3. Les types composés : les structures

Nous n'avons utilisé jusque là que des variables élémentaires de type entier, réel ou caractère.

Il est possible de déclarer des types plus riches pour décrire un objet complexe .

Cela permettra de concevoir une fonction qui retourne un résultat composé de plusieurs valeurs.

Nous appellerons **structure** ou variable structurée une variable composée de plusieurs variables élémentaires et/ou structurées.

Exemple :

Structure pour un étudiant :

- numéro entier
- date de naissance 3 entiers : jour, mois, année
- note du bac réel

3.1. Déclaration d'une variable structurée ou structure

Déclaration de 2 variables structurées d1 et d2 composées de trois entiers

```
struct
    {
    int jour , mois, an ;
    }d1,d2 ;
```

3.2. Définition d'un type de structure

Déclaration d'un type Date :

```
typedef  struct
    {
    int jour , mois, an ;
    }Date ;
```

Un type défini s'utilise ensuite comme un type prédéfini.

On peut écrire :

Date d1, d2 ;

3.3. Déclaration et manipulation d'une variable structurée

On peut initialiser une variable structurée lors de sa déclaration

Date d1={5,12,2000}, d2 ;

d2=d1;

mais l'affectation : d2 ={05,12,1981}; est interdite

On désigne un champ d'une structure par: *nomDeLaStructure.nomDuChamp*

```
d2.an= 2001;
printf("%d  %d  %d",d1.jour, d1.mois, d1.an);
```

3.4. Structure et sous structure

Exemple :

Définition d'un type Etudiant composé des informations d'un étudiant : un entier, une structure de type Date, un flottant :

```
typedef struct
    {
    int         numero ;
    Date        dateNaissance ;
    float       noteBac ;
    }Etudiant ;
```

On déclare la structure etud1 par :

Etudiant etud1 ;

On désigne les champs de etud1 par :

etud1.numero
etud1.dateNaissance.jour
etud1.dateNaissance.mois
etud1.dateNaissance.annee
etud1.noteBac

On peut initialiser une variable structurée lors de sa déclaration :

```
Etudiant  etud2 ={007, {10,06,1980} , 12.5 } ;
Date date1 ;
```

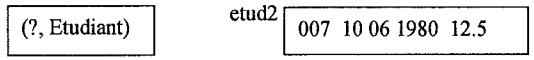
On peut utiliser chaque champ comme une variable élémentaire ou structurée:

```
etud1.numero = 008 ;
etud1.dateNaissance.annee = 1981 ;
date1 = etud1.dateNaissance;

printf("Taper le numéro de l'étudiant :");
scanf(" %d",&etud1.numero) ;
```

3.5. Pointeur vers une structure; l'opérateur ->

On peut déclarer un pointeur de structure

```
Etudiant *p1,  etud2 ={007, {10,06,1980} , 12.5 } ;
```




```
p1 = &etud2;        // place l'adresse de etud2 dans p1
```


Lorsqu'on dispose d'un pointeur vers une structure, l'accès à ses champs peut se faire avec l'opérateur ->.

p1->numero est équivalent à *etud2.numero*

On aura aussi:
*p1 équivalent à etud2
(*p1).numero est le numéro de etud2, soit 007

priorité des opérateurs :

le point (.) est prioritaire sur *

*p1.numero est équivalent à *(p1.numero)
et n'a aucun sens

3.6. *Mise en œuvre de structures : fonction de saisie d'un étudiant*

L'exemple ci dessous met en œuvre une fonction saisirEtudiant qui fait la saisie des informations pour un étudiant et retourne ces données dans une structure.

```
/*
programme:  etud06.c
auteur:
date:
finalité : saisie d'informations Etudiant dans une structure
*/
#include <stdio.h>

/************************************
      structure Date
*/
typedef struct
      {
int jour ;
int mois;
int annee ;
} Date;

/************************************
      structure Etudiant
numero            numero de l'étudiant
dateNaissance     jour,mois,annee
noteBac           moyenne obtenue au bac
*/
typedef struct
{
int         numero ;
Date      dateNaissance ;
float             noteBac ;
}Etudiant ;

/************************************/

Etudiant saisirEtudiant (void);
void test(void);

/************************************/

Etudiant saisirEtudiant (void)

/*
Finalité :
Description:
      saisit les differentes informations d'un etudiant
      et les retourne

Valeur retournée: etud  l'étudiant saisi

Variables:
      etud  structure Etudiant
                        numero
                        dateNaissance   (jour,mois,annee)
                        noteBac
*/
{
Etudiant etud;
printf("\n\nSaisie d'un étudiant");
printf("\nNum,ro:");
```



```
scanf("%d",&etud.numero);
printf("Date de naissance: taper jour mois an (séparés par un espace):");
scanf("%d%d%d",   &etud.dateNaissance.jour,
                  &etud.dateNaissance.mois,
                  &etud.dateNaissance.annee);
printf("Note au Bac:");
scanf("%f",&etud.noteBac);

return etud;
}
/****************************************/
void test(void)

/*
Finalité :
Description:      appelle la fonction saisirEtudiant
                        et affiche l'étudiant saisi

Variables:
etud  structure Etudiant
                        numero
                        dateNaissance   (jour,mois,annee)
                        noteBac
*/
{
Etudiant etud;

etud = saisirEtudiant();

printf("\n%d",etud.numero);
printf("\n%d/%d/%d",    etud.dateNaissance.jour,
                        etud.dateNaissance.mois,
                        etud.dateNaissance.annee);
printf("\n%5.2f",etud.noteBac);
return;
}

/****************************************/

int main(void)
{
test();
printf("\n Faire Entree"); scanf("%*c%*c");
return 0;
}
```

Essai du programme

```
Saisie d'un étudiant
Numéro:1234
Date de naissance: taper jour mois an (séparés par un espace):05 12 1980
Note au Bac:13.5

1234
5/12/1980
13.50
```

Remarque :
Les types de structure Date et Etudiant ont été définis au début du fichier en dehors des fonctions (définitions globales) afin d'être utilisés dans les deux fonctions.

4. Tableaux et structures en C

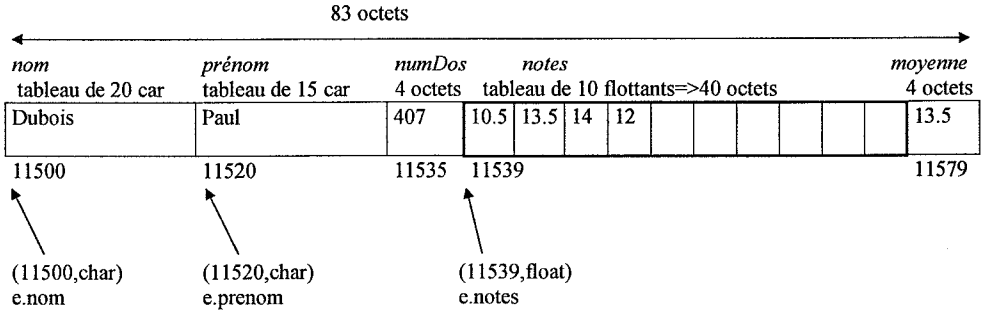
4.1. Représentation d'une structure

Dans les structures, on peut déclarer des chaînes de caractères telles que *nom, prenom, adresse, ...*
Prenons le cas où un étudiant est composé d'un nom (de moins de 20 caractères), d'un prénom (de moins de 15 caractères), d'un numéro de dossier (entier), d'un tableau de 10 notes et d'une moyenne. Nous aurons la déclaration du type Etudiant suivante :

```
typedef struct
{
   char nom[20] ;
   char prenom[15] ;
   int numDos ;
   float notes[10] ;
   float moyenne ;
} Etudiant ;
```

Les composants d'une structure de type étudiant seront stockés de façon contigüe .
Avec une taille de 4 octets pour un entier ou un flottant, la taille d' un élément de type *Etudiant* sera :
20 + 15 + 4 + 40 (10 x 4) +4 = 83 octets

Etudiant e;



Pour une variable e de type *Etudiant* commençant à l'octet 11500.

e.nom représente le doublet (11500, char) .
e.nom[2] est le caractère b
e.prenom représente le doublet (11520, char).
e.numDos est le nom de la zone contenant un int, qui débute à l'octet 11535.
e.notes représente le doublet (11539, float).
e.notes[3] est un flottant à l'adresse (11539 + 3 × 4), c'est-à-dire 11551.
e.moyenne est le nom de la zone contenant un flottant, qui débute à l'octet 11579.

4.2. Tableau de structures

Pour un tableau tet déclaré:
Etudiant tet[5];

on aura 5 structures tet[0], tet[1], tet[2], tet[3], tet[4]
tet est un pointeur constant vers un tableau d'étudiants

tet (adresse du tableau, Etudiant)

tet[0]	Dubois	Paul	407	10.5	13.5	14	12	...					13.5
tet[1]	Martin	Robert	412	15.5	12.5	13	12.8						13,7
tet[2]	Dupond	...											
...													
...													

tet[1].nom est le nom de la structure tet[1], càd est l'adresse de la chaîne Martin
tet[1].notes[3] vaut 12.8.

4.3. Exemple: mise en oeuvre d'un tableau de noms : les mois de l'année

Solution avec tableau de type "structure de chaîne"

```
#include <stdio.h>

typedef struct
                {
                char nom[20];
                } Nom;

void afficherMois1( Nom mois[ ], int nbmois)
{
int i ;
for (i=0; i<nbmois; i++)        printf("%s\n", mois[i].nom);
return ;
}

int main()
{
Nom   mois1[3]={"janvier","février","décembre"};

afficherMois1( mois1,3) ;

return 0;
}
```

Solution avec tableau de caractères à deux dimensions

```
#include <stdio.h>

void afficherMois2( char mois[ ][20], int nbmois)
{
int i ;
for (i=0; i<nbmois; i++)        printf("%s\n", mois[i]);
return ;
}

int main()
{
char    mois2[3][20]={"janvier","février","décembre"};

afficherMois2( mois2,3) ;

return 0;
}
```

5. Arithmétique des pointeurs

Soit : Etudiant tet[5] ;

La notation *tet[n]* désigne l'objet se trouvant à l'adresse *tet + n* calculée par l'expression:
adresse de tet + n x taille d'un Etudiant

Si tet= (10000, Etudiant),
alors tet[2] désigne l'étudiant débutant à l'adresse *tet + 2* , soit (10000 + 2 × 83) c'est-à-dire 10166

*(tet + 2) et tet[2] ont la même valeur, celle de l'étudiant en 3ème position du tableau.

Dessinez les variables correspondant aux déclarations suivantes:

```
Etudiant  *i, *j, tet[5];
i=tet;
j= i+2;
```

Constatez que tet[2] , i[2] , *j , *(tet+2) et *(i+2) désignent le même objet Etudiant

Attention,
- on peut ajouter ou retrancher un entier à un pointeur, mais l'addition de 2 pointeurs n'a pas de sens.
- la différence entre deux pointeurs de même type donne le nombre d'objets placés entre les deux adresses.

6. Allocation dynamique de mémoire: malloc, calloc, realloc, free

L'allocation dynamique permet de réserver un espace mémoire en cours de traitement:
- au départ, on ne déclare qu'un pointeur
- ensuite on alloue un espace mémoire vers lequel on fait pointer le pointeur
- enfin on libère l'espace mémoire

Cette technique sera utilisée notamment quand on ne souhaite pas figer la taille d'un tableau au départ.
Un tel espace ne disparaît pas automatiquement à la fin de la fonction qui l'a créé; il faut le détruire explicitement.
Les fonctions malloc et calloc permettent d'allouer de l'espace.
La fonction free libère un espace alloué.
Ces fonctions font partie de la bibliothèque stdlib.h

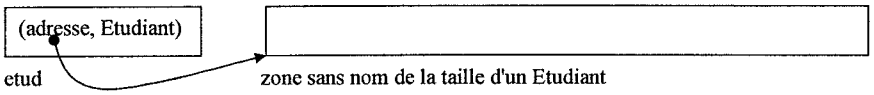
6.1. *Réservation: malloc*

void *malloc (unsigned long *taille*)

réserve une zone de *taille* octets et rend l'adresse de la zone.
C'est un pointeur de type void: on lui donne un type particulier en plaçant le type souhaité entre parenthèses avant la donnée dont on veut modifier le type.
Le pointeur est NULL si la réservation n'a pu se faire.

Exemple: déclaration d'un pointeur d'étudiant et allocation de mémoire pour un étudiant

```
#include <stdlib.h>
...
Etudiant * etud;
etud=(Etudiant*) malloc(sizeof(Etudiant));
if (etud==NULL) {printf("\n Pb sur malloc de etud"); exit(1);}
```



6.2. *Libération: free*

void free(void *adresse)

libère l'espace alloué par un malloc ou un calloc à l'adresse indiquée
Cette libération peut être faite en dehors de la fonction qui a fait l'allocation

```
free(etud);
```

6.3. *Allocation dynamique pour un tableau*

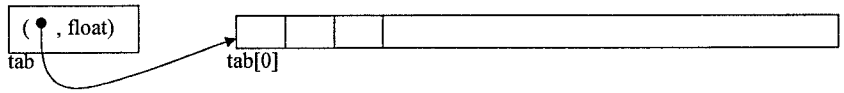
Pour un tableau, la place à réserver sera: *nb d'étudiants * sizeof(Etudiant)*

On pourra aussi utiliser
*void *calloc(unsigned long nbElements, unsigned long tailleElements)*

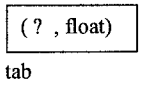
calloc initialise l'espace réservé à 0 (contrairement à malloc)

Exemple: tableau de flottants; la taille est entrée au clavier

```
float *tab;                                          tab  ( ? , float)
int  tailleTab;
printf("Nb d'éléments du tableau:");
scanf("%d",&tailleTab);

tab = (float*) malloc(tailleTab *  sizeof(float));
...
```




```
free(tab);
```



6.4. Modification de la taille: realloc

La fonction realloc permet de modifier la taille d'un espace réservé avec malloc ou calloc en allouant un nouvel espace contigu et en y recopiant les données de l'espace initial.

*void *realloc(void * adresseEspace, unsigned long nouvelleTaille)*

realloc retourne la nouvelle adresse de l'espace réservé

Exemple:

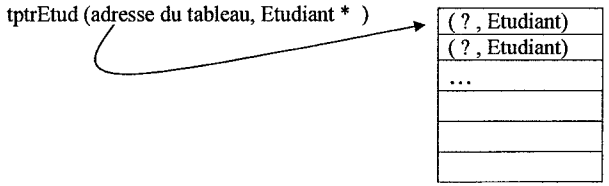
```
float * nouvTab ;
tailleTab = tailleTab + 100;
nouvTab = (float*)realloc( tab, tailleTab * sizeof(float) );
if (nouvTab==NULL)        {printf("\n  Pb sur realloc de tab");  exit(1);}
else                      tab = nouvTab;
```

6.5. Allocation d'un tableau de pointeurs

Pour éviter « realloc », on va surdimensionner la taille des tableaux : cela peut être très pénalisant pour des tableaux d'objets de grande taille : tableau de personnes…
La mise en œuvre d'un tableau de pointeurs est une réponse à ce problème et permet d'éviter de déplacer des données volumineuses.

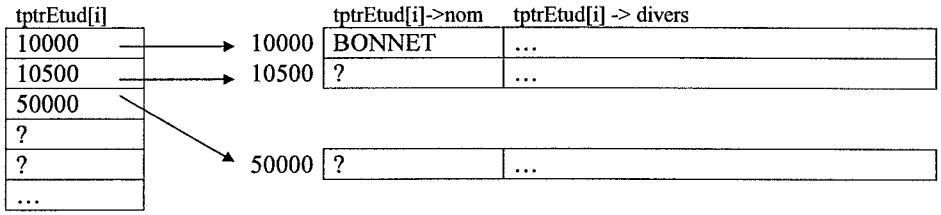
Exemple:

```
Typedef struct
                {
                  char nom[21] ;
                  char divers[500] ;
                } Etudiant ;

Etudiant * tptrEtud[1000];          // déclare un tableau de 1000 pointeurs d' Etudiant
                                    // tptrEtud est un pointeur constant vers le tableau
```



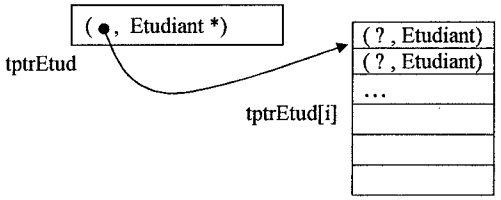
L'allocation de l'espace pour les données se fait à mesure; il n'est pas nécessairement contigu.

```
tptrEtud [0] =( Etudiant *) malloc ( sizeof(Etudiant) );

strcpy(tptrEtud [0] ->nom, "BONNET  " ) ;

tptrEtud [i] =( Etudiant *) malloc ( sizeof(Etudiant) );
```



L'allocation du tableau de pointeurs peut aussi être faite par *malloc*.

```
Etudiant * *  tptrEtud ;      // déclare un pointeur vers un pointeur d'étudiant

tptrEtud = (Etudiant * * ) malloc ( 1000 * sizeof( Etudiant *) ) ;
```



7. Les fichiers binaires

Un fichier binaire contient des données sous une forme identique à leur codage en mémoire centrale; les opérations de lecture et d'écriture sont très rapides; par contre, ils ne sont pas imprimables ni éditables.

7.1. Ouverture: fopen

On ouvre un fichier binaire comme un fichier texte; on précise qu'il s'agit d'un fichier binaire en ajoutant la lettre b au mode d'ouverture :

"rb" mode lecture à partir du début du fichier
"wb" mode écriture; si le fichier existait, il est écrasé, sinon, il est créé
"ab" mode ajout à la fin d'un fichier existant ou non
"r+b" mode lecture et écriture

```
FILE *fetud;
fetud = fopen("fetud.don","wb");     //ouverture d'un fichier binaire en écriture
```

7.2. Ecriture: fwrite

fwrite écrit nb objets consécutifs dans un fichier binaire de pointeur flot:

*int fwrite(const void *source, int taille, int nb, FILE *flot)*
source est l'adresse du premier objet
taille est la taille d'un objet

Exemple: création d'un fichier binaire des étudiants

```
typedef struct
{
int          numero ;
Date      dateNaissance ;
float        noteBac ;
}Etudiant ;

Etudiant etud;

FILE *fetud;
fetud = fopen("fetud.don","wb");     //ouverture d'un fichier binaire en écriture
...
etud=saisirEtudiant();
fwrite (&etud, sizeof(Etudiant), 1, fetud);
...
```

7.3. Lecture: fread

fread tente de lire *nb* objets sur le flot indiqué et les copie à l'adresse *destination*

*int fread(void *destination, int taille, int nb, FILE *flot)*
taille est la taille d'un objet
fread renvoie le nombre d'objets lus (sera inférieur à nb si la fin du fichier a été atteinte)

Exemple: lecture d'un fichier binaire des étudiants

```
Etudiant etud;

FILE *fetud;
fetud = fopen("fetud.don","rb");     //ouverture d'un fichier binaire en lecture
if (fetud==NULL) {printf("\n Pb fopen "); exit(1);}
...
fread (&etud, sizeof(Etudiant), 1, fetud);
                ↑
...             └── adresse de début
```

7.4. Positionnement: fseek

L'ouverture d'un fichier positionne son pointeur de flot au début du fichier.
Après une lecture ou une écriture, ce pointeur se trouve positionné sur l'emplacement où se fera la prochaine lecture ou écriture: c'est la *position courante*.
fseek permet de se positionner dans un fichier (texte ou binaire) en indiquant le déplacement à effectuer par rapport au début, à la position courante, ou à la fin.
Cela n'a de sens que pour des fichiers dont on maîtrise parfaitement la taille des contenus.
On l'utilisera aussi pour modifier un objet du fichier que l'on vient de lire.

*int fseek(FILE *flot, long deplacement, int origine)*

deplacement est exprimé en nombre d'octets
origine a pour valeur SEEK_SET (début), SEEK_CUR (position courante) ou SEEK_END
La fonction renvoie 0 en cas de succès, non nul en cas d'erreur.

Exemple: modifier la note du bac d'un étudiant du fichier

```
Etudiant etud;
FILE *fetud;
fetud = fopen("fetud.don","r+b");    //ouverture d'un fichier binaire en lecture et écriture
if (fetud==NULL) {printf("\n Pb fopen "); exit(1);}
...
/*      L'étudiant à traiter est lu par fread
 ce qui positionnera le pointeur de flot sur l'étudiant suivant        */
fread (&etud, sizeof(Etudiant), 1, fetud);

printf("Etudiant N° %d    Note du bac actuelle: %.2f \n",etud.numero, etud.noteBac);
printf("Taper la nouvelle note:"); scanf("%f",&etud.noteBac );

/* on se repositionne sur l'étudiant et on réécrit ses données dans le fichier */
fseek(fetud, -sizeof(Etudiant), SEEK_CUR);
fwrite (&etud, sizeof(Etudiant), 1, fetud);
```

8. Récursivité

Une fonction récursive est une fonction qui s'appelle elle même.

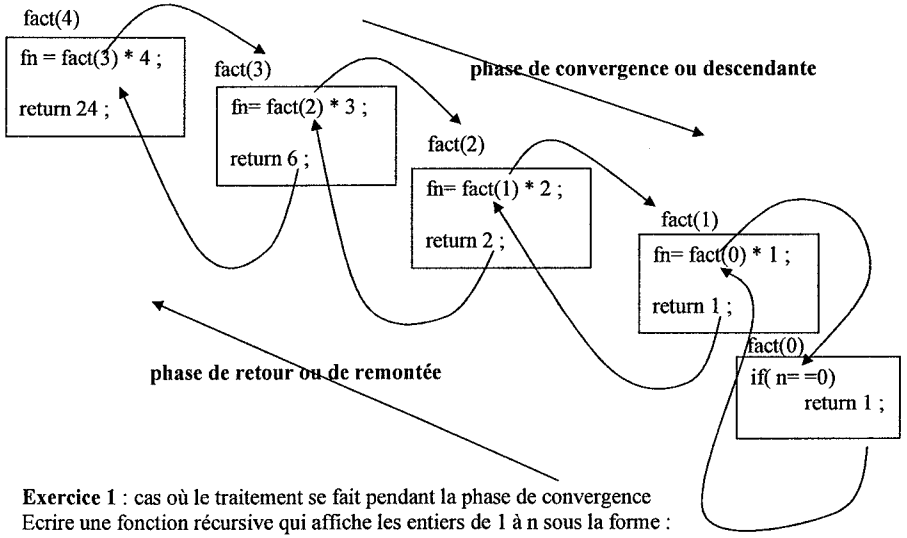
Exemple : calcul de factorielle : n ! = 1 x 2 x 3 ...x (n-1) x n
On peut définir n ! de façon récursive :

- 0 ! = 1
- n ! = (n-1) ! x n

et écrire le programme récursif suivant :

```
int fact (int n)
{
        int fn;
        if ( n = = 0) return 1;   // cas d'arrêt
        fn = fact ( n-1) * n;     // appel récursif convergeant vers le cas d'arrêt
        return fn ;
}
```

Trace d'exécution :schémas sémantiques pour fact(4)



Exercice 1 : cas où le traitement se fait pendant la phase de convergence
Ecrire une fonction récursive qui affiche les entiers de 1 à n sous la forme :
n n-1 ... 3 2 1

Exercice 2 : cas où le traitement se fait pendant la phase de retour
Ecrire une fonction récursive qui affiche les entiers de 1 à n sous la forme :
1 2 3 ... n-1 n

Exercice 3
Ecrire une fonction récursive qui affiche les entiers de 1 à n sous la forme :
n n-1 ... 3 2 1 0 1 2 3 ... n-1 n

9. Recherche dans un tableau trié

1.0	1.0	2.7	3.0	4.2	5.0	6.0	8.0	8.0	10.0

int rech(float *tf, int nbf, float v)

9.1. Parcours séquentiel

On compare la valeur cherchée aux valeurs du tableau.
On s'arrête quand on atteint une valeur du tableau supérieure ou égale à la valeur cherchée (les valeurs suivantes seront encore plus grandes) ou quand on atteint la fin du tableau.

9.2. Recherche dichotomique

On compare la valeur cherchée v à celle du milieu du tableau
On recommence dans la partie où elle se trouve
On retourne la position d'insertion de v dans le tableau

Version itérative

```
int rechDichoIter (float *tf, int nbf, int v)
{
   int deb=0, m, fin=nbf-1;  //debut, milieu, fin de la partie où on recherche v

  while(deb<=fin)        // tant que la partie  à examiner contient au moins 1 élément
    {
        m=(deb+fin)/2;

        if(v<=tf[m])

                fin=m-1;        // on va examiner les valeurs à gauche de m

         else
                deb=m+1;        // on va examiner les valeurs à droite de m
    }
return deb;
}
```


Version récursive

```
int rechDichoRec (float *tf, int nbf, int v)
{
 int  pos,m;

 if (nbf==0 ) return 0;                                  // cas d'un tableau vide
 if (nbf==1) if (v<=tf[0]) return 0; else return 1;     // tableau à 1 élément

 m=(nbf - 1)/2;

 if(v<=tf[m])
        pos= rechDichoRec (tf,m+1,v);                    // partie gauche où se trouve v
 else
        pos= m+1 + rechDichoRec (tf+m+1,nbf-(m+1),v);   // partie droite où se trouve v

 return pos;
}
```

10. Complexité d'un algorithme

C'est l'estimation du nombre d'opérations de base (affectation, test, calcul...) en fonction de la taille des données en entrée

10.1. Exemple : recherche séquentielle dans un tableau non trié

```
/*
 recherche séquentielle de la valeur v
 dans un tableau tf non trie de nbf flottants
 retourne la position ou -1 si non trouve
*/
int rech0(float *tf, int nbf, int v)
{
   int i;
   i=0;
   while(i<nbf)
            {  if (v==tf[i]) return i;
               i=i+1;
            }
   return -1;
}
```

Nb maximum d'opérations (cas où on parcourt tout le tableau) :

Le nombre maximum d'opérations sera : 3 + 3 x nbf
C'est la **complexité au pire**.
On peut aussi calculer la **complexité moyenne** : si les valeurs du tableau sont uniformément réparties, on effectuera la boucle nbf / 2 fois en moyenne. La complexité moyenne sera 3 + 3 x nbf / 2

Le nombre d'opérations est proportionnel à la taille du tableau : on dit que cet algorithme est linéaire en la taille du tableau

On dit que l'algorithme est en $O(n)$ (ou linéaire) si le nombre d'opérations est proportionnel à n, en $O(n^2)$ ou quadratique, en $O(2^n)$ ou exponentiel

10.2. Complexité de la recherche dichotomique

Soit n la taille du tableau, k le nombre d'itérations.
Le nombre d'opérations sera :
3 + 4 * k + 1

Estimation du nombre d'itération :
à la $1^{ère}$ on a n éléments
à la $2^{ème}$, on a n / 2 éléments
...
à la kième, on a $n / 2^{k-1}$ éléments
à la fin, on a 1 élément : $1 = n / 2^{k-1} \Rightarrow n = 2^{k-1} \Rightarrow k-1 = \text{Log}_2(n)$

Le nombre d'opération sera donc : $4 + 4 *(\text{Log}_2(n) + 1) = 8 + 4 * \text{Log}_2(n)$
La complexité est en $O(\text{Log}_2(n))$

Pour $n = 1024 = 2^{10}$, $\text{Log}_2(n) = 10$ => Nb d'opérations = 48
Par recherche séquentielle dans un tableau non trié, on aurait 3 + 3 * 1024 3075 opérations

11. TRI d'un tableau

4.2	3.0	1.0	2.7	5.0	1.0

void tri (float *tf, int nb)

1.0	1.0	2.7	3.0	4.2	5.0

11.1. Tri par sélection / échange (ou permutation)

Tri quadratique : complexité en $O(n^2)$

On recherche la plus petite valeur et on la place au début du tableau en l'échangeant avec l'élément de début du tableau (variante: rechercher la plus grande valeur que l'on place à la fin)
On déplace le début du tableau d'une position vers la droite et on recommence.
On arrête quand on est à la fin

11.2. Tri par insertion

Tri quadratique

On trie les 2 premiers éléments
On trie les 3 premiers : pour cela,

- On compare le 3ème élément à ceux qui le précèdent
- On décale vers la droite ceux qui le précèdent et qui lui sont supérieurs
- On place le 3eme élément dans la position créée par le décalage

On continue avec un élément supplémentaire

11.3. Tri bulle

Tri quadratique

On parcourt le tableau en comparant les éléments consécutifs (0 et 1, 1 et 2,...): s'ils sont mal ordonnés, on les échange, ce qui conduit à déplacer le plus grand élément vers la fin.
On recommence (avec le dernier élément en moins à chaque itération) tant qu'on trouve des éléments mal ordonnés

11.4. Tri dichotomique par fusion (approche récursive)

Complexité en $O(n \text{ Log}_2(n))$

On coupe le tableau en 2 parties
On trie chaque partie (appel récursif : les parties à trier sont de plus en plus petites jusqu'à ne contenir qu'un élément))
On copie les 2 parties triées dans 2 tableaux
On interclasse (ou fusionne) ces 2 tableaux triés tf1 et tf2 dans un tableau tf
*void fusion(float * tf1, int nb1, float * tf2, int nb2, float * tf)*

11.5. Tri rapide (Quick sort) ; méthode dichotomique avec pivot

Quadratique au pire, en $O(n \text{ Log}_2(n))$ en moyenne
En moyenne, il est meilleur que le tri par fusion mais ses performances sont moins régulières.

On prend la valeur du dernier élément comme « pivot »
On cherche à placer au début du tableau les éléments inférieurs au pivot, et à la fin les éléments supérieurs ; entre les deux se trouve la place qu'occupera le pivot dans le tableau trié.
Ce traitement s'appelle le partitionnement du tableau :

- on parcourt le tableau en partant du début jusqu'à trouver un élément i supérieur au pivot
- on parcourt le tableau en partant de la fin jusqu'à trouver un élément j inférieur au pivot
- si i<j, on échange les éléments i et j
- on continue les parcours jusqu'à ce qu'ils se croisent : i>j
- on échange le pivot avec le point i de croisement

Le pivot est alors à sa place
On trie de cette façon chaque partie du tableau de part et d'autre du pivot (appel récursif)

```
void trirapide(float *f, int nbf)
{
   int i,j,k;
   float w,d;

   if(nbf<=1) return;
   i=0;
   j=nbf-2;
   d=f[nbf-1];          // d valeur du dernier element du tableau= pivot
   while (i<=j)
   {
       while (f[i]<=d &&i<nbf) i++;  // on cherche un element à echanger a partir du debut
       while (f[j]>=d &&  0<=j) j--;  // on cherche un element à echanger a partir de la fin
       if (i<j){w=f[i]; f[i]=f[j]; f[j]=w;}          // si i<j, on echange
   }
   f[nbf-1]=f[i];f[i]=d;        // i est la place de d: on echange l'element i avec la fin

   trirapide(f,i);              // 1ere partie de 0 à i-1 => i elements
   trirapide(f+i+1,nbf-i-1); // 2eme de i+1 à nbf-1 => nbf - i -1

   return;
   }
```

11.6. Comparaison des performances des tris

en millisecondes (cf http://www.dailly.info/algorithmes-de-tri)

Nb de valeurs	4 10^3	32 10^3	131 10^3	2100 10^3	4200 10^3	8000 10^3
Tri bulle	94	5594	168 10^3			
Tri sélection	31	2281	49 10^3			
Tri insertion	31	2000	44 10^3			
Tri dichotomique fusion	0	31	62	1172	2391	4992
Tri dichotomique rapide	0	15	31	922	2610	8235

12. Listes

Une liste est un ensemble d'objets "qui se suivent"
On parle de la ***tête*** de la liste, du ***suivant*** ou du ***précédent*** d'un objet, de la ***queue*** de la liste

12.1. Stockage dans un tableau

Les objets sont stockés de façon contiguë

Exemple: liste de personnes

```
typedef struct { char nom[21];  char divers[479] } Personne;

Personne tpers[1000];
```

Liste de personnes

		nom	divers
tête	10000	BONNET	...
	10500	MARTIN	
	11000	DUBOIS	
queue	11500	DUPONT	
	12000	...	
	...		

Avantages:
- recherche rapide
- le rang de l'objet permet de connaître immédiatement sa place

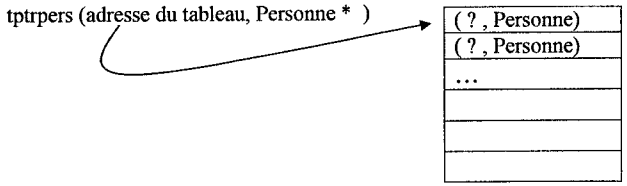
Inconvénients
- on doit réserver à l'avance un espace mémoire contigu
- pour augmenter l'espace, il faut allouer un autre espace contigu et recopier les données du précédent
- la place occupée par chaque objet est la même quelque soit sa taille (par exemple pour une liste d'adresses)
- l'insertion est complexe quand on veut maintenir les objets triés: nécessité de décalage
- le tri nécessite de manipuler des volumes importants de données

12.2. Tableau de pointeurs

Pour éviter de déplacer des données volumineuses, on peut introduire un tableau de pointeurs.

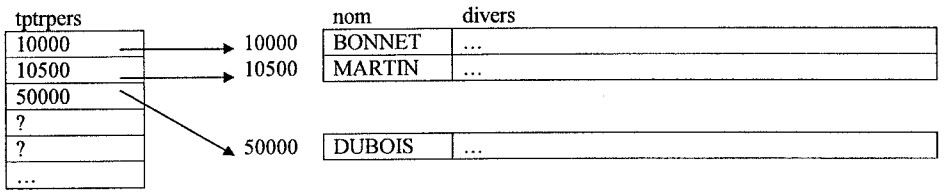
Exemple:

```
Personne * tptrpers[1000];        // déclare un tableau de 1000 pointeurs de Personnes
                                  // tptrpers est un pointeur constant vers le tableau
```

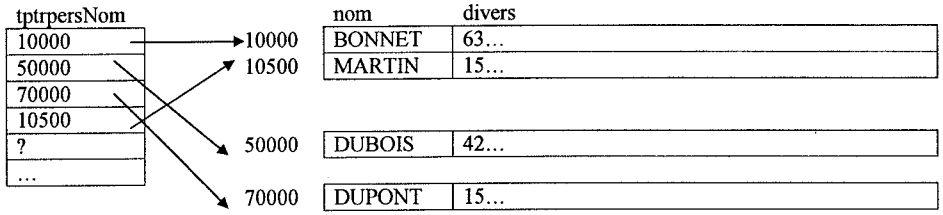


L'allocation de l'espace pour les données se fait à mesure; il n'est pas nécessairement contigu.

```
tptrpers[0] =( Personne *) malloc ( sizeof(Personne) );
...
tptrpers[i] =( Personne *) malloc ( sizeof(Personne) );
```



Pour obtenir une liste triée sur le nom, on réorganise le tableau des pointeurs
Pour insérer une personne dans la liste triée, on alloue son espace et on insère son pointeur dans le tableau des pointeurs. Les décalages à effectuer ne concerneront que le tableau des pointeurs.
<u>On ne déplacera pas les données des personnes</u>



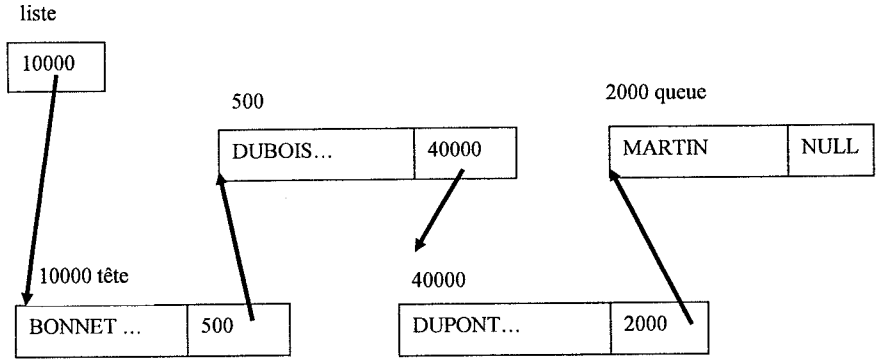
Ex :
dessiner un 2ème tableau de pointeurs présentant les personnes triées sur le département
dessiner l'insertion de DURAND 42
étudier l'algorithme d'insertion d'une personne

12.3. Liste chaînée

Une liste chaînée est un ensemble de maillons (ou cellules) liés entre eux par des pointeurs
Chaque maillon est une structure composée de:
- les données d'un objet de la liste
- un pointeur vers le maillon suivant

La queue de la liste présente un pointeur NULL
Une liste est déterminée par un pointeur qui pointe vers la tête (le 1er maillon)

Exemple: liste de personnes, triée

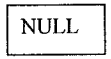


Une telle liste permet d'insérer un nouvel élément en conservant l'ordre, sans déplacer les autres

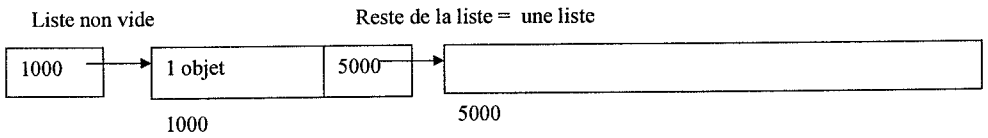
Exemple: insérer ALBERT, DURAND , ROBERT

Définition récursive :

Une liste peut être définie par les deux cas suivants :

- la liste est vide — Liste vide 

- la liste est constituée d'un objet et d'une liste (le reste de la liste qui peut être vide)



12.4. Mise en oeuvre d'une liste chaînée en C

- déclaration des types correspondant à un maillon (structure récursive: elle comporte un pointeur vers elle même) et à un pointeur de liste (c'est un pointeur de maillon)
- création d'une liste vide
- remplissage par insertions en tête: chaque nouveau maillon est alloué dynamiquement; il devient tête de liste et pointe vers l'ancienne tête.
- affichage de la liste: on affiche les données d'un maillon; le pointeur nous donne l'adresse du reste de la liste
- suppression des maillons d'une liste

Les fonctions sont abordées de façon récursive en exploitant le fait qu'une liste se compose d'un maillon pointant vers une sous liste : on décompose la liste en sous listes jusqu'au maillon de queue qui pointe vers une sous liste vide.

On notera que toutes les opérations sur la liste sont « encapsulées » dans des fonctions ; la structure de liste chaînée n'apparaît pas dans la fonction test ; l'utilisateur n'a pas à connaître l'implantation interne de la structure : on pourra la modifier sans qu'il ait à modifier les programmes qui l'utilisent.
Les opérations de base classiques seront :

- créer une liste vide
- ajouter un élément (en tête ou dans l'ordre si la liste est triée)
- rechercher un élément
- supprimer un élément
- afficher la liste depuis la tête, depuis la queue

...

```
/*************listfloat_recurs.c ********************
Liste chainee
Approche récursive
*/
#include <stdio.h>

/*-----------------------------------------
      Liste de flottants
*/

typedef struct maillonf
        {       float    f;                     // un flottant
                struct maillonf * restlist;     // pointeur vers le reste de la liste
        } Maillonf;                             // type maillon

typedef Maillonf  * Listef;                     //  type pointeur de maillon = type liste

/*-----------------------------------------
    cree une liste vide
*/
Listef creeListeVide(void)
{
     return NULL;
}
/*-----------------------------------------
  insertion d'une donnee en tête de liste
  retourne la nouvelle liste
*/
Listef  insereEnTete(Listef lf, float donnee)
{
        Listef  sauvelf = lf;    // sauve l'adresse du maillon de tête
        lf= (Listef ) malloc(sizeof(Maillonf));         // alloue un maillon
                                                        // et place son adresse dans lf
        lf->f = donnee;             // renseigne le maillon
        lf-> restlist= sauvelf;     // fait pointer le maillon  sur l'ancienne tête
        return lf;
}

/*-----------------------------------
     affichage de la liste
      affichage de la queue vers la tête
*/

void affich (Listef lf)
{
    if (lf==NULL) {printf("\n--------- liste----------\n"); return;}

    affich(lf->restlist);           // affiche le reste depuis la queue
    printf("%.2f\n",lf->f);      //  affiche la tête
}
```



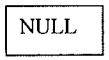
```
/*************************************************
  supprimer une liste
  il faut libérer la mémoire pour chaque maillon
  on retourne une liste vide
*/
Listef  supprime( Listef lf)
{
   Listef tete=lf ;                 // maillon à supprimer
   if (lf==NULL) return NULL;   // cas d'une liste vide
   lf=supprime(lf-> restlist);   //supprime le reste de la liste
   free(tete);                      // supprime le maillon de tête
   return lf;     // =NULL
}
/*-----------------------------------------------*/
void test(void)
{
        Listef lf;
        lf=creeListeVide();
        lf= insereEnTete (lf,17);
        lf= insereEnTete (lf,16);
        lf= insereEnTete (lf,15);
        lf= insereEnTete (lf,12);
        affich(lf);
        lf=supprime(lf);
        affich(lf);
}

int main(void)
{
   test();
   scanf("%*c%*c");
}
```

Faire les schémas correspondant au déroulement du programme
Donner le résultat.

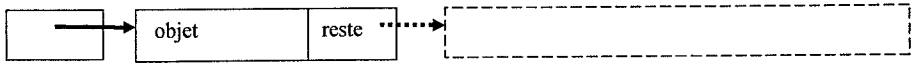
12.5. Traitement récursif d'une liste

Une liste se présente suivant l'un des deux shémas suivants:

Liste vide



Liste non vide Maillon de tête



list0 list0->objet list0->reste

list0->reste est un pointeur vers le reste de la liste list0: c'est une liste

Une liste étant une structure récursive, il sera souvent très intéressant de faire une approche récursive du traitement d'une liste
.Pour concevoir une fonction qui nécessite de parcourir les différents maillons d'une liste, on va de façon générale:

- décrire le traitement dans le cas d'une liste vide
- décrire le traitement concernant le maillon de tête (insertion, suppression, affichage...) avant de traiter le reste de la liste
- faire un appel récursif à la fonction pour traiter le reste de la liste (qui peut être vide ou non)
- décrire le traitement du maillon de tête après avoir traité le reste de la liste

Les appels récursifs pour traiter le reste de la liste vont converger vers une liste à un maillon, puis vers une liste vide : les traitements de ces cas simples ont été décrits au début de la fonction

Exemple: repérez ces différentes étapes dans les fonctions du programme listfloat_recurs.c

12.6. Liste de pointeurs; multilistes

On peut introduire des objets dans une liste en mettant en œuvre une liste de pointeurs vers les objets.
On peut introduire des objets dans différentes listes

Exemple: liste de personnes par ordre alphabétique et liste par des personnes du Cantal