École de technologie supérieure

THÈME 10
Structures et bâtiments imaginaires

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

1. Mise en situation

A. Situation simple

Dans les exemples que nous avons rencontrées jusqu'à maintenant, nos variables simples suffisaient à la tâche et lorsque nous avions besoin d'en gérer plusieurs, nous utilisions les tableaux. Considérons maintenant le cas où nous aimerions stocker les informations suivantes sur un étudiant:

- son nom et son prénom
- son code permanent
- sa moyenne actuelle
- s'il étudie à temps plein ou à temps partiel
- le nombre de cours inscrits dans son dossier
- le nombre de cours réussis
- la liste de ses cours actuels

Avec les types que nous connaissons, nous devrions déclarer 8 variables:

```
char nom[80];
char prenom[80];
char codePerm[10];
double moyenne;
char tempsPlein; /* 0 = temps partiel, 1 = temps plein */
int nbCoursInscrits;
int nbCoursReussis;
char cours[10][6]; /* au maximum, 10 cours dont le sigle comporte 6 caractères */
```

Si nous devons ensuite passer cette information à une fonction, la tâche devient lourde. La fonction devrait avoir aussi 8 arguments!

A.1. DÉCLARATION D'UNE STRUCTURE

Malheureusement ici les tableaux ne peuvent nous venir en aide car le type de chacune des informations est différent. C'est là qu'entre en jeu les structures. Dans un premier temps, nous pourrions définir la structure d'un étudiant comme suit:

```
struct Etudiant{
   char nom[80];
   char prenom[80];
   char codePerm[10];
   double moyenne;
   char tempsPlein; /* 0 = temps partiel, 1 = temps plein */
   int nbCoursInscrits;
   int nbCoursReussis;
   char cours[10][7];
};
```

NOTEZ BIEN le point-virgule à la fin de la définition. Cette séquence déclare un nouveau type, le type struct Etudiant. Ce type comprend 7 champs. Il est important de noter que ces champs sont conservés dans un endroit distinct des variables. On peut donc avoir des champs portant le même nom qu'une de nos variables (locales ou globales).

Pour déclarer une variable de ce type, il suffit de faire comme d'habitude, le type "struct Etudiant" suivi du nom de la variable :

```
struct Etudiant vous;
```

Suite à la déclaration "vous" est maintenant une variable de type "struct Etudiant". Mais ce n'est pas la seule manière de procéder. Voici les possibilités qui s'offrent à vous:

1) Faire comme précédemment

(1) Faire comme précédemment

```
struct Etudiant{
      char nom[80];
      char prenom[80];
      char codePerm[10];
      double moyenne;
      char tempsPlein; /* 0 = temps partiel, 1 = temps plein */
      int nbCoursInscrits;
      int nbCoursReussis;
      char cours[10][7];
   };
int main(void){
  struct Etudiant vous;
  ...
}
```

(2) Faire un "struct" anonyme:

```
int main(void){
  struct {
      char nom[80];
      char prenom[80];
      char codePerm[10];
      double moyenne;
      char tempsPlein;
      int nbCoursInscrits;
      int nbCoursReussis;
      char cours[10][7];
   }  vous;
  ...
}
```

(3) Passer par une définition de type:

```
typedef struct {
         char nom[80];
         char prenom[80];
         char codePerm[10];
         double moyenne;
         char tempsPlein;
         int nbCoursInscrits;
         int nbCoursReussis;
         char cours[10][7];
       }                        tEtudiant;
int main(void){
  tEtudiant vous;
  ...
}
```

La troisième possibilité rend les choses beaucoup plus agréable et nous l'utiliserons partout dans le reste de ce document. Au lieu de dire que "vous" est une variable de type "struct Etudiant", on dira plus simplement, que "vous" est de type "tEtudiant". NOTE: les programmeurs utilisent souvent un "t" devant leur nom de type de manière à les distinguer plus facilement de leur variable et aussi, pour ne pas utiliser un bon nom de variable comme nom de type.

A.2. ACCÈS AUX CHAMPS

La variable "vous" est complexe, aussi complexe qu'un tableau. Pour accéder à ses champs, les concepteurs du C auraient pu imaginer toutes sortes d'artifices. Ils ont choisi d'utiliser le point:

```
vous.nom
vous.prenom
vous.codePerm
vous.moyenne
vous.coursInscrits
vous.coursReussis
vous.cours
```

L'utilisation de ces champs respectent ensuite les règles normales d'utilisation des variables ordinaires. Ainsi, pour accéder à la première lettre du nom de famille, on devrait faire:

```
vous.nom[0]
```

et pour accéder au sigle du premier cours actuel, on devrait faire:

```
vous.cours[0]
```

B. Compléments sur les structs

Inventons la structure Date:

```
typedef struct{ int jour; int mois; int annee; } Date;
```

Ne nous arrêtons pas en si bon chemin. Pourquoi ne pas ajouter un champ "datenaissance" dans la structure de notre étudiant:

B.1. DES STRUCTS DANS DES STRUCTS

La structure de notre étudiant devient:

```
typedef    struct {
             char nom[80];
             char prenom[80];
             char codePerm[10];
             double moyenne;
             char tempsPlein; /* 0 = temps partiel, 1 = temps plein */
             int nbCoursInscrits;
             int nbCoursReussis;
             char cours[10][7];
             Date datenaissance;
           } tEtudiant;
```

Comment pourrons-nous accéder à l'année de naissance de notre étudiant? "vous.datenaissance" nous permet d'accéder au champ. Le reste, c'est de la routine puisque l'utilisation de ces champs respectent ensuite les règles que nous avons établies. Le champ "datenaissance" est lui-même un "struct". On utilisera donc le point:

```
vous.datenaissance.annee
```

B.2. INITIALISATION LORS DE LA DÉCLARATION

Comme pour les tableaux, les structs admettent un petit raccourci permettant de les initialiser. Commençons par le type "Date".

```
Date aujourdhui = {15, 11, 1999};
```

On peut également initialiser une variable de type "tedudiant":

```
tetudiant vous = { "Tartempion", "Nicolas", "TARN01019905", 3.7, 1, 56, 15,
                   {"INF125", "MAT115", "CHM101", "ATE050", "", "", "", "", "", ""},
                   {1, 1, 1999} };
```

B.3. CARACTÉRISTIQUES IMPORTANTES

Pour afficher une variable "struct", il faut absolument afficher chacun des champs simples, un à la fois. Ainsi, pour afficher la variable "aujourdhui" (de type Date), on devra faire:

```
printf("%i %i %i", aujourdhui.jour, aujourdhui.mois, aujourdhui.annee);
```

L'affichage de la variable "vous" sera évidemment plus complexe:

INSTRUCTION	RÉSULTAT
`printf("%s, %s\n", vous.nom, vous.prenom);`	`Tartempion, Nicolas`
`printf("Code perm. = %s\n", vous.codePerm);`	`TARN01019905`
`printf("Moyenne = %.1f\n", vous.moyenne);`	`Moyenne = 3.7`
`printf("Status : %s\n", vous.tempPlein ? "Temps plein" : "Temps partiel");`	`Status : Temps plein`
`printf("Cours réussis: %i\n, vous.nbCoursInscrits);`	`Cours inscrits: 56`
`printf("Cours réussis: %i\n, vous.nbCoursReussis);`	`Cours réussis: 15`

`for (i=0; i < 10; i++)`
`  if (strlen(cours[i]) > 0)`
`   printf("%s \n");`
	`INF125`
	`MAT115`
	`CHM101`
	`ATE050`

`printf("Date de naissance: %i %i %i\n", vous.datenaissance.jour, vous.datenaissance.mois, vous.datenaissance.annee);`	`Date de naissance: 1 1 1999`

Rien ne nous empêche de connaître la taille d'une variable "struct" en utilisant l'opérateur "sizeof":

```
        sizeof(tEtudiant)
ou
        sizeof(vous)
```

Il est important de savoir que la taille d'un struct N'EST PAS NÉCESSAIREMENT égale à la somme des tailles de ses champs. Ainsi,

```
sizeof(vous) != sizeof(vous.nom) + sizeof(vous.prenom) + sizeof(vous.moyenne) +
                sizeof(vous.tempsplein) + sizeof(vous.nbCoursInscrits) +
                sizeof(vous.nbCoursReussis) + sizeof(vous.cours) +
                sizeof(vous.datenaissance.jour) + sizeof(vous.datenaissance.mois) +
                sizeof(vous.datenaissance.annee)
```

La raison de cette inégalité est que le système effectue parfois un réalignement des champs au début d'un mot mémoire. Si le champ précédent (ex. tempsPlein) ne prend pas tout un mot mémoire, le système laisse entre ce champ et le suivant, un ensemble d'octets vides. Ce réalignement a un impact majeur sur la comparaison de deux "struct".

Pour comparer deux "struct", mieux vaut effectuer une compaisons champ par champ et résister à la tentation d'utiliser "memcmp".

2. Tableaux et pointeurs de struct

A. Tableaux statiques de structs

A.1. DÉCLARATION ET ACCÈS AUX CHAMPS

Puisqu'il est possible de déclarer des tableaux à l'intérieur des "struct", vous pensez bien qu'il est possible de déclarer des tableaux de "struct". Supposons que nous désirions conserver les informations concernant deux classes de INF125. On pourrait alors faire:

```
tEtudiant groupe1[45];
tEtudiant groupe2[45];
```

Pour accéder au nom du 6e étudiant dans le groupe 1, on ferait alors:

```
groupe1[5].nom
```

et pour accéder au premier cours suivi par ce même étudiant, on ferait:

```
groupe1[5].cours[0]
```

dans notre exemple précédent.. maintenant plusieurs éléments

A.2. ASSIGNATION DE STRUCTS

Vous vous rappelez sans doute (voir le thème 9) qu'il n'est pas possible d'effectuer une assignation directe de deux tableaux:

```
int T[10];
int M[10];
```

L'assignation T = M est donc illégale.

Cependant, et c'est là que ça devient curieux, si les tableaux sont dans des structs, l'assignation devient possible. Ainsi,

```
typedef struct { int tableau[10]; } tablo;

int main(void){
  tablo T, M;
  ...
  T = M; /* tout à fait légal ! */
```

on peut donc, sans risque d'erreur, faire une assignation sur deux structs plus complexes:

```
tEtudiant e1, e2;
...
e2 = e1; /* tout à fait légal */
```

Par contre, l'assignation suivante n'est pas légale puisque "groupe1" et "groupe2" sont deux tableaux!

```
groupe1 = groupe2;
```

Mais, l'assignation suivante est légale puisque "groupe1[6]" et "groupe2[8]" sont des structs!

```
groupe1[6] = groupe2[8];
```

Il faut cependant être prudent car si le "struct" contient non pas un tableau statique mais un tableau dynamique, l'assignation n'aura pas l'effet désiré. Dans ce cas, le contenu des tableaux dynamiques ne sera pas copié.

B. Pointeurs

B.1. POINTEURS SUR DES STRUCTS

Considérez la séquence suivante:

```
tEtudiant * ptr;
ptr = (tetudiant) malloc(sizeof(tetudiant));
```

Récapitulons, `ptr` est de type `(tetudiant *)` et `*ptr` est donc de type `tetudiant`. Pour accéder au champ "nom", on devrait en toute logique faire:

```
*ptr.nom
```

Malheusement, cette séquence n'a pas l'effet désirée. L'opérateur point étant plus prioritaire que l'étoile, notre expression précédente est en fait équivalente à:

```
*(ptr.nom)
```

Ce qui est incorrect. On devrait donc faire:

```
(*ptr).nom
```

Puisque cette syntaxe est lourde, les concepteurs du C ont introduit une autre petite sucrerie, la "flèche". L'expression précédente peut donc s'écrire:

```
ptr->nom
```

C'est bien agréable d'autant plus que cela cache l'étoile.

Prenons un exemple plus complexe. Supposons que notre type "tEtudiant" contienne un tableau "resultats" qui soit dynamique. Disons que ce tableau est destiné à recevoir toutes les notes obtenues par un étudiant durant son baccalauréat. Le type "tEtudiant" devient:

```
typedef    struct {
             char nom[80];
             char prenom[80];
             char codePerm[10];
```

```
    double moyenne;
    char tempsPlein; /* 0 = temps partiel, 1 = temps plein */
    int nbCoursInscrits;
    int nbCoursReussis;
    char cours[10][7];
    double * resultats;
    Date datenaissance;
} tEtudiant;
```

Supposons également que nous en sommes à l'étape où nous devrions créer le tableau "resultats" assez grand pour y placer 80 notes (nous supposons ici que la variable "vous" a déjà été déclarée):

```
vous.resultats = (double *) malloc(80 * sizeof(double));
```

Évidemment, pour accéder au 3e résultat de cet étudiant, on fera:

```
vous.resultats[2]
```

Par contre, imaginons que nous ayons déclaré la variable "vous" de la manière suivante::

```
tEtudiant * vous;
```

Pour créer la variable "vous", on fera d'abord:

```
vous = (tEtudiant *)malloc(sizeof(tEtudiant));
```

puis, pour créer le tableau "resultats", on fera donc:

```
vous->resultats (double *) malloc(80 * sizeof(double));
```

B.2. TABLEAUX DYNAMIQUES DE STRUCTS

En conséquence, au lieu d'avoir une variable statique pour contenir nos étudiants, on peut également bâtir un tableau dynamique. Pour déclarer "groupe1", on fait:

```
tEtudiant * groupe1;
```

Pour créer assez de place pour mettre 150 étudiants, on fera ensuite:

```
groupe1 = (tEtudiant *)malloc(150 * sizeof(tEtudiant));
```

Pour créer chacun des tableaux dynamiques "resultats" à l'intérieur de chaque "struct", on devra faire une boucle:

```
for (i = 0; i < 150; i++)
   groupe1[i].resultats = (double *) malloc(80 * sizeof(double));
```

ou encore:

```
for (i = 0; i < 150; i++)
  (groupe1 + i)->resultats = (double *) malloc(80 * sizeof(double));
```

ou même:

```
for (i = 0; i < 150; i++)
  (*(groupe1 + i)).resultats = (double *) malloc(80 * sizeof(double));
```

Examinons maintenant l'étude de cas.

ÉTUDE DE CAS

UN RÉPERTOIRE D'ADRESSES DE COURRIEL

Vous en avez ras-le-bol d'avoir à dépendre des logiciels énormes pour simplement consulter votre répertoire d'adresses de courriel. Vous avez donc décidé de faire un petit programme C qui vous permettra de consulter votre répertoire facilement et n'importe où. Le programme ainsi que le fichier d'adresses tiendra facilement sur une disquette et pourra être exécuté peu importe la taille de la mémoire de l'ordinateur utilisé. Le programme complet peut être récupéré ici: AGENDA.C.

Voici les informations associés à chaque adresse de courriel:

- un numéro d'identification
- le nom et le prénom de la personne
- l'adresse de courriel de cette personne

Notre petit programme fonctionnera avec un menu qui mettra ainsi à notre disposition les possibilités suivantes:

Création d'un nouveau répertoire d'adresse
: On pourra ainsi créer autant de répertoires qu'on le désire.

Activation d'un répertoire existant:
: Cet option nous permettra d'ouvrir un fichier afin d'y effectuer diverses opérations sans avoir à ouvrir et fermer le fichier à chaque fois.

Recherche d'une adresse dans le répertoire actif:
: On pourra chercher une adresse en utilisant soit le numéro d'identification associé à l'adresse, soit en utilisant une sous-chaîne du nom, du prénom ou du courriel.

Modification d'une adresse dans le répertoire actif:
: On pourra ainsi modifier l'un des trois champs de base (nom, prénom et le courriel).

Insertion d'une nouvelle adresse:
: Cette fonctionnalité est évidemment essentielle pour réussir à insérer des adresses dans notre répertoire.

Impression du contenu du répertoire:
: Pour nous permettre d'avoir une copie texte de notre répertoire.

1. LA STRUCTURE D'UNE ENTRÉE et DU RÉPERTOIRE

Une entrée est donc un "struct" défini comme suit:

```
typedef struct{
        long noId;
        char nom[80];
        char prenom[80];
        char courriel[80];
       }                    tAdresse;
```

Le fichier actif est accessible par une variable "static" de type FILE * déclarée comme suit:

```
static FILE * REPERTOIRE = 0;
```

Le répertoire lui-même est un fichier binaire composé d'éléments "tAdresse".

2. LA FONCTION "main" ET QUELQUES FONCTIONS AUXILIAIRES

L'affichage du menu et la saisie du choix de l'utilisateur sont effectuées par une petite fonction simple. La fonction "saisirCaractereEnEcho" effectue un simple "getch" et produit un écho du choix de l'utilisateur. Celui-ci n'a donc pas à appuyer sur la touche <enter> pour valider son choix. Pratique. La fonction "separateur" ne fait qu'afficher des tirets...pour faire joli.

```
int afficherMenu(void){
  int choix;
  printf("\n\n");
  separateur();
  printf("(A)ctiver un répertoire existant\n");
  printf("(C)réer un nouveau répertoire\n");
  printf("(R)echercher une entrée dans le répertoire actif\n");
  printf("(M)odifier une entrée dans le répertoire actif\n");
  printf("(N)ouvelle entrée dans le répertoire actif\n");
  printf("(I)mprimer le répertoire actif\n");
  printf("(Q)uitter le programme\n");
  separateur();
  printf("Votre choix ? ");
  choix = saisirCaractereEnEcho();
  separateur();
  return choix;
}
```

```
int saisirCaractereEnEcho(void){
  int choix;
  choix = toupper(getch());
  printf("%c\n", choix);
  return choix;
}
```

```
void separateur(void){
  printf("-----------------------------------------------\n");
}
```

La fonction "main" effectue donc, à partir du choix de l'utilisateur, les appels généraux aux fonctions appropriées. La fonction "desactiverRepertoire" a pour fonction de fermer le fichier présentemen actif.

```
int main(void){
  char choix;

  while((choix = afficherMenu()) != 'Q'){
    switch (choix){
      case 'C': creerRepertoire(); break;
      case 'R': rechercherAdresse(); break;
      case 'M': modifierAdresse(); break;
      case 'N': nouvelleAdresse(); break;
      case 'A': activerRepertoire(); break;
      case 'I': impressionRepertoire(); break;
    }
  }
  desactiverRepertoire();
  return EXIT_SUCCESS;
}
```

```
void desactiverRepertoire(void){
  if (REPERTOIRE)
    fclose(REPERTOIRE);
  REPERTOIRE = 0;
}
```

3. CRÉATION ET ACTIVATION D'UN RÉPERTOIRE

`int creerRepertoire(void){`
`  char nom[80];`
`  FILE * REP;`
	variable temporaire qui n'affecte pas le répertoire actif.

`  printf("\n\n"); separateur();`
`  printf("CREATION D'UN NOUVEAU REPERTOIRE\n");`
`  separateur();`	

`  printf("Nom du nouveau répertoire (sans extension): ");`
`  gets(nom);`
`  strcat(nom, ".AGD");`
	Récupération et création du nom du fichier

`  REP = fopen(nom, "wb");`	Ouverture
`  if (!REP) return erreur(0, nom);`	validation
`  fclose(REP);`	Fermeture (le fichier est créé)

`  printf("Répertoire %s est créé et peut être activé.\n", nom);`
`  separateur();`
`  return 1;`
`}`
	Finale

`int activerRepertoire(void){`
`  char nom[80];`
`  FILE * newREP;`
	variable temporaire afin de ne pas désactiver notre répertoire actuel si le répertoire fourni par l'utilisateur n'est pas bon.

`  printf("\n\n"); separateur();`
`  printf("ACTIVATION D'UN REPERTOIRE\n");`
`  separateur();`	

`  printf("Nom du répertoire (sans extension): ");`
`  gets(nom);`
`  strcat(nom, ".AGD");`
	Récupération et création du nom du fichier

`  newREP = fopen(nom, "r+b");` `  if (!newREP) return erreur(1, nom);`	Ouverture en lecture/écriture et validation usuelle
`  desactiverRepertoire();` `  REPERTOIRE = newREP;`	désactivation (fermeture) de l'ancien répertoire et initialisation de la variable globale

`  printf("Répertoire %s est activé.\n", nom);`
`  separateur();`
`  return 1;`
	c'est un succès.

```
}
```

4. INSERTION D'UNE NOUVELLE ADRESSE

```
int nouvelleAdresse(void){
  tAdresse adresse;
  printf("\n\n"); separateur();
  if (!REPERTOIRE) return erreur(3, "");
```

s'il n'y a aucun répertoire actif, erreur

```
  printf("AJOUTER UNE ADRESSE\n");
  separateur();
  adresse = creerAdresse();
```

création d'une adresse complète

```
  fseek(REPERTOIRE, 0, SEEK_END);
  if (!fwrite(&adresse, sizeof(tAdresse), 1, REPERTOIRE))
    return erreur(2, "");
```

on se rend à la fin du fichier et on inscrit l'adresse en question.

```
  printf("Nouvelle adresse insérée dans le répertoire actif.\n");
  return 1;
}
```

c'est un succès

```
tAdresse creerAdresse(void){
  long oldPosition;
  tAdresse adresse;

  oldPosition = ftell(REPERTOIRE);
```

On conserve la position actuelle dans le fichier (au cas où celle-ci pourrait être utilisée par la fonction qui appelle...)

```
  fseek(REPERTOIRE, 0, SEEK_END);
  adresse.noId = ftell(REPERTOIRE)/sizeof(tAdresse);
```

On se rend à la fin et en divisant la taille du fichier par la taille d'une entrée, on obtient ainsi notre no d'identification.

```
  fseek(REPERTOIRE, oldPosition, SEEK_SET);
```

On se replace à la position originale

```
  strcpy(adresse.nom, saisirSequence("Nom", ""));
  strcpy(adresse.prenom, saisirSequence("Prenom", ""));
  strcpy(adresse.courriel, saisirSequence("Courriel", ""));

  return adresse;
}
```

Chaque séquence de caractères saisie au clavier sera copiée dans le champ correspondant de la structure.

```
char * saisirSequence(char msg[], char old[]){
  char sequence[255];
  printf("%s : %s%s", msg, old,
         strlen(old) > 0 ? " (<enter> pour valider) " : "");
  gets(sequence);
  return strlen(sequence) > 0 ? sequence : old;
}
```

Si la fonction reçoit une chaîne de caractères dans "old", cette chaîne sera affichée suivie de (<enter> pour valider). cela nous permet d'utiliser cette fonction lorsque l'on désire modifier une adresse (voir plus bas). Lors de la création, seul le message dans "msg" est visible puisque le paramètre "old" contient une chaîne vide.

5. IMPRESSION DU RÉPERTOIRE

`int impressionRepertoire(void){`
`  tAdresse adresse;`
`  long n;`
`  printf("\n\n"); separateur();`
`  if (!REPERTOIRE) return erreur(3, "");`
	validation de base

`  fseek(REPERTOIRE, 0, SEEK_SET);` `  n = 0;`	On se rend au début du fichier...

`  while (fread(&adresse, sizeof(tAdresse), 1, REPERTOIRE)) {`
`  imprimerAdresse(adresse);`
`  n++;`
	On lit successivement chaque adresse pour ensuite l'afficher.

`  if (n % 4 == 0) petitePause();` `  }`	À toutes les 4 entrées, on effectue une petite pause.

`  petitePause();`
`  return 1;`
`}`
	On termine par une petite pause...

`void imprimerAdresse(tAdresse adresse){`
`  printf("\n");`
`  printf("  ID: %li\n", adresse.noId);`
`  printf("  %s, %s\n", adresse.nom, adresse.prenom);`
`  printf("  %s\n\n", adresse.courriel);`
`}`
	Affichage d'une entrée à l'écran.

`void petitePause(void){`
`  printf("Appuyez sur une touche pour continuer...\n");`
`  getch();`
`}`
	Petite fonction utilitaire pour ne pas encombrer le code.

6. RECHERCHE D'UNE ADRESSE

```
int rechercherAdresse(void){
  long id;
  char sequence[80];
  long position;
  tAdresse adresse;
  printf("\n\n"); separateur();
  if (!REPERTOIRE) return erreur(3, "");
  printf("RECHERCHE D'UNE ADRESSE\n");
```

Validation de base

```
  do {
    separateur();

    switch(menuRecherche()){
```

Selon le choix (1 ou 2) effectué par l'utilisateur (et retourné par la fonction "menuRecherche"), on exécute deux blocs distincts

```
      case '2': id = getId();
                position = id * sizeof(tAdresse);
                break;
```

L'utilisateur veut une recherche par numéro d'identification. On passe à la récupération d'un numéro d'identification et de la position dans le fichier (exprimée en nombre d'octets).

```
      case '1' : getSequence(sequence);
                 position = rechercherSequence(sequence);
                 break;
    }
```

L'utilisateur veut une recherche par contenu. Il faut donc récupérer d'abord la séquence de recherche pour ensuite rechercher la position de l'entrée (dans le fichier) qui contient cette séquence.

```
    if (position < 0)
      printf("Aucune entrée ne correspond ...\n");
    else {
      adresse = getAdresse(position);
      imprimerAdresse(adresse);
    }
```

Si la position est valide, on récupère l'entrée à la position indiquée puis on affiche l'entrée à l'écran.

```
    printf("Autre recherche ? ");
  }while(saisirCaractereEnEcho() == 'O');
  separateur();
  return 1;
}
```

On peut évidemment faire plusieurs recherches...

```
int menuRecherche(void){
  int typeRecherche;
  do{
    printf("\nRecherche par:\n");
    printf("(1) nom (prenom ou courriel)\n");
    printf("(2) numéro d'identification\n");
    printf(" ?  ");
    typeRecherche = saisirCaractereEnEcho();
    printf("\n");
  }while(typeRecherche != '1' && typeRecherche != '2');
  return typeRecherche;
}
```

Petite fonction utilitaire pour gérer les choix dans un petit menu de recherche.

```
tAdresse getAdresse(long position){
  tAdresse adresse;
  fseek(REPERTOIRE, position, SEEK_SET);
  fread(&adresse, sizeof(tAdresse), 1, REPERTOIRE);
  return adresse;
}
```

Fonction toute simple qui récupère l'entrée située à la position indiquée.

```
long getId(void){
  long id;
  id = -1;
  do {
      fflush(stdin);
      printf("No d'identification recherché : ");
  } while(!scanf("%li", &id));
  fflush(stdin);
  return id;
}
```

fonction utilitaire permettant de saisir un nombre entier valide.

`void getSequence(char nom[80]){`
`  printf("Saisissez le nom ou le prénom ou le courriel au complet\n");`
`  printf("ou une séquence de caractères devant s'y trouver.\n");`
`  printf("? ");`
`  gets(nom);`
`}`
	fonction utilitaire permettant de saisir une séquence de caractères

`long rechercherSequence(char sequence[]){`
`  tAdresse adresse;`
`  fseek(REPERTOIRE, 0, SEEK_SET);`
	On se rend au début du fichier

`  while (fread(&adresse, sizeof(tAdresse), 1, REPERTOIRE)){`
`  if (strstr(adresse.nom, sequence) ||`
`  strstr(adresse.prenom, sequence) ||`
`  strstr(adresse.courriel, sequence) )`
	On cherche séquentiellement dans le fichier une entrée dont le nom, le prénom ou le courriel contienne la sous-chaîne "sequence".

`  return ftell(REPERTOIRE) - sizeof(tAdresse);` `  }`	Si tel est le cas on retourne la position de l'entrée en question. La soustraction est nécessaire parce qu'après le "fread" nous sommes situés à l'entrée suivante...
`  return -1;` `}`	Si nous n'avons rien trouvé, on retourne un négatif.

7. MODIFICATION D'UNE ADRESSE EXISTANTE

`int modifierAdresse(void){`
`  long id;`
`  long position;`
`  tAdresse adresse;`
`  printf("\n\n"); separateur();`
`  if (!REPERTOIRE) return erreur(3, "");`
`  printf("MODIFICATION D'UNE ADRESSE\n");`
`  separateur();`
	validation de base

`  id = getId();` `  position = id * sizeof(tAdresse);`	Récupération d'un numéro d'identification et de la position dans le fichier (exprimée en nombre d'octets).

`  if (position < 0)`
`  printf("Entrée %li inexistante ....\n", id);`
`  else {`
`  adresse = getAdresse(position);`
`  imprimerAdresse(adresse);`
`  modifier(&adresse);`
`  fseek(REPERTOIRE, position, SEEK_SET);`
`  fwrite(&adresse, sizeof(tAdresse), 1, REPERTOIRE);`
`  }`
	On lit l'entrée tel qu'on la trouve actuellement dans le fichier, on l'imprime et on demande ensuite à l'utilisateur de la modifier. Finalement, l'entrée modifiée est replacée dans le fichier.

`  separateur();`
`  return 1;`
`}`
	c'est un succès

`void modifier(tAdresse * ptrAdr){`
`  strcpy(ptrAdr->nom,`
`  saisirSequence("Nom",ptrAdr->nom));`
`  strcpy(ptrAdr->prenom,`
`  saisirSequence("Prenom", ptrAdr->prenom));`
`  strcpy(ptrAdr->courriel,`
`  saisirSequence("Courriel", ptrAdr->courriel));`
`}`
	Chaque séquence de caractères saisie au clavier sera copiée dans le champ correspondant de la structure. Lors du saise de la séquence, si l'utilisateur appuie simplement sur <enter>, l'ancienne valeur du champ sera utilisée. Cela permet à l'utilisateur de ne saisir que les champs qu'il désire modifier et lui évite d'avoir à retaper toutes les informations de l'adresse.

8. AFFICHAGE DES ERREURS

Un tableau constant permet de prévoir quelques messages d'erreurs utiles sans pour autant encombrer le code. Ce tableau est ensuite utilisé par la fonction "erreur".

`const char * ERREURS[] =`	Le tableau de message

`{"Impossible de créer le fichier %s",`
`"Impossible d'activer le fichier %s",`
`"Problème lors de l'écriture%s.",`
`"Il n'y a pas de répertoire actif%s."};`
	Les 4 messages d'erreurs sont en fait des chaînes de formattage. Le %s des deux premiers messages seront remplacés par une chaîne de caractères significative tandis que le %s des deux derniers seront remplacés par une chaîne vide. Nous aurions pu, à la place, utiliser une deuxième fonction "erreur" qui n'aurait pas inséré un "ajout" dans le message et nous aurions alors appelé cette fonction lors de l'affichage des deux derniers messages...cela nous aurait permis d'éviter tout ce blabla!

`int erreur(int no, char * addendum){` `char message[255];`	La chaîne "addendum" sera ajoutée au message d'erreur afin de personnaliser celui-ci.
`sprintf(message, ERREURS[no], addendum);`	Le message est créé en utilisant la chaîne de formattage contenue dans le tableau à l'endroit indiqué en y ajoutant l'information supplémentaire qui vient remplacer le %s dans le message d'erreur.

`printf("Erreur %i: %s\n", no, message);`
`return 0;`
`}`
	Le message complètement formé est finalement affiché.