Département IMA / 3A (S6) / 2013 **Programmation Avancée**
Jonathan Aceituno, Vincent Aranéga, Nathalie Devésa, Rachid Elkouri et Laure Gonnord

Bilan sur les 3 premiers tp de PA

1 TP1 : Annuaire

1.1 À destination des étudiants des groupes 1 et 2

Vous avez eu l'énoncé de l'an dernier, ce qui ne pose pas de problème car c'est à peu près le même. Néanmoins, l'énoncé de cette année guidait vers une solution propre pour s'apercevoir qu'il faut arrêter de lire les personnes : utiliser une valeur de retour spéciale dans `lirePersonne`, et utiliser cette valeur dans `creeAnnuaire` pour stopper la boucle de construction.

```
int lirePersonneBis(Personne* p_p)
{
  if (scanf("%s",(*p_p).lastname)==EOF) return 1 ; // fin de fichier
  scanf("%s",(*p_p).firstname);
  (*p_p).birthdate=lireDate();
  scanf("%s",(*p_p).telnumber);
  return 0;
}
```

L'insertion dans l'annuaire:

```
void construitAnnuaire(Annuaire* thebase)
{
..
while ((lirePersonneBis(&oneP) == 0) && (*thebase).lastindex <MAX_PERSONNES - 1))
  {
// la lecture a fonctionné ET il reste de la place
    ... // insertion effective + augmentation de l'index de la liste contiguë
   }
...
}
```

1.2 Au sujet des listes contiguës

Certains n'ont déclaré qu'un tableau, on demandait bien une liste contiguë, c'est-à-dire une structure contenant un tableau statique et un marqueur de fin :

```
typedef struct Annuaire {
  int lastindex;
  Personne ptable[MAX_PERSONNES];
} Annuaire;
```

Attention à bien maintenir le marqueur de fin à jour (l'initialiser juste après la déclaration, et l'augmenter à chaque insertion dun élément). Lors de l'insertion d'un élément dans une liste contiguë, on fera bien attention à vérifier au préalable que la table n'est pas complètement remplie.

1.3 Au sujet des pointeurs sur les structures

Par défaut, une structure est passée par valeur dans les fonctions. Pour chaque fonction avec des structures, on se demandera donc si la structure passée en paramètre est modifiée, par exemple la fonction :

```
void imprimeAnnuaire(Annuaire thebase)
```

ne fait qu'imprimer les champs, donc si *suffit* de passer l'annuaire par valeur[1]. Par contre, il est *nécessaire* de passer par adresse l'annuaire pour le modifier, par exemple :

```
void construitAnnuaire(Annuaire* thebase)
```

pour la construction. Lors de l'appel, `construitAnnuaire` veut donc un pointeur sur un annuaire, donc on l'appelle de la façon suivante :

```
Annuaire thebase ;           //déclaration, la structure est allouée.
thebase.lastindex = -1;      // do not forget!
construitAnnuaire(&thebase); // appel
```

Ce qu'il faut retenir

- déclaration et utilisation de liste contiguë (et gestion du marqueur de fin).
- le passage de structure par adresse.
- l'utilisation de scanf et des redirections, et arrêter un scanf à la fin de fichier avec `EOF`

2 TP2 : Annuaire et fichier

2.1 Lire dans un fichier (question 1.1)

La lecture dans un fichier peut se faire uniquement après ouverture du-dit fichier avec fopen :

```
FILE* fp = fopen(nomdufichier,"r")
```

Ensuite, on utilise `fscanf` de la même façon que `scanf` en passant le pointeur de fichier en premier argument :

```
fscanf(fp,"%d",&i); // lit un entier dans le fichier
```

La question 1 du TP2 nécessitait donc que les fonctions de lecture clavier soient remplacées par des fonctions de lecture dans un fichier, il fallait donc passer le pointeur de fichier (ou *file descriptor*) en paramètre à toutes les fonctions de lecture :

```
Date lireDate(FILE* fp)
int lirePersonneBis(Personne* p_p, FILE* fp)
void construitAnnuaire(FILE* fp,Annuaire* thebase)
```

[1] On peut éventuellement "optimiser" en passant l'annuaire par adresse, ce qui évite une copie d'une structure qui peut être grosse.

Voici un exemple d'appel à partir du `main` :

```
FILE* fp = fopen("annu.txt","r")
construitAnnuaire(fp,&thebase);
```

Il existe une fonction qui permet de détecter la fin de fichier, c'est la fonction `feof()`, que l'on peut utiliser dans `lirePersonneBis` par exemple :

```
fscanf(fp,"%s",(*p_p).lastname); // première lecture
if (feof(fp)) return 1;          // fin de fichier atteinte
fscanf(fp,"%s",(*p_p).firstname);
```

2.2 Arguments de la ligne de commande

Comme le montre l'exemple fourni par l'énoncé, on a accès aux chaînes de caractères passées en argument à la ligne de commande (leur nombre, et leur valeur), en déclarant deux paramètres au `main` : un entier, *classiquement* appelé `argc` (argument counter) et un tableau de chaînes de caractères, classiquement appelé `argv` (argument vector) :

```
int main(int argc, char **argv){ ...
```

Attention, le nom du binaire est compté comme un argument, et est stocké dans `argv[0]`. Si votre binaire est lancé avec :

```
./tp2 annu.txt
```

`argc` vaudra 2, `argv` sera un tableau de taille 2 contenant "./tp2" et "annu.txt" (deux chaînes). Si vous voulez utiliser un argument de la ligne de commande, il faudra donc tester que l'utilisateur a bien donné le bon nombre d'arguments, donc dans le cas de l'énoncé :

```
if(argc != 2) {
  printf("wrong number of arguments\n");
  return 1; // on arrête avec un code différent de 0
} else {
  // suite "normale du programme"
  fp = fopen(argv[1],"r");
}
```

2.3 Modification du fichier annuaire (question 1.3)

L'écriture est symétrique à la lecture, en utilisant `fprintf` et un fichier ouvert en écriture. Modifier une unique ligne d'un fichier existant est possible (en utilisant `fseek`), mais l'énoncé vous guidait vers une solution plus simple qui consiste à créer un nouveau fichier dans lequel sauver l'annuaire entier :

```
FILE* fpsave = fopen(svfilename,"w");
sauveAnnuaire(thebase,fp)
```

Ce qu'il faut retenir

- l'ouverture et fermeture de fichier, en lecture et en écriture, avec `fopen` et `fclose`. Ne pas oublier de tester que tout s'est bien passé.
- Lire dans un fichier avec `fscanf` et écrire avec `fprintf`.
- L'utilisation des arguments de la ligne de commande.
- La recherche dichotomique (algo classique à savoir implémenter).

3 Tp 3 : Allocation dynamique

3.1 Allocation d'une matrice

Dans le cours il y a un exemple d'allocation d'un tableau de N entiers :

```
int* m = (int*) malloc(N * sizeof(int));
```

Ensuite, on peut utiliser M comme avant, par exemple pour stocker 2 à la case 9 (si N¿9) :

```
m[9] = 2;
```

Il convient par contre de s'assurer que l'allocation dynamique a bien réussi :

```
int* m = (int*) malloc(N * sizeof(int));
if (m != NULL){
//continuer ici
...
}
```

Pour créer une matrice, on commence par faire un dessin, puis on en déduit qu'on essaie de créer un tableau de N "tableaux d'entiers de taille N", c'est à dire un tableau de N pointeurs d'entiers. On commence donc par allouer un tableau de pointeurs d'entiers, en remplaçant les (int) par des (int*) dans le code précédent :

```
int** m = (int**) malloc(N * sizeof(int*));
```

ensuite, chaque $M[i]$ (pour i de 0 à $N-1$) doit lui même contenir un tableau d'entiers de taille N, donc :

```
for (i = 0; i < N; i++)
  m[i] = (int*) malloc(N * sizeof(int));
```

à ce stade là, si *et seulement si* tous les `malloc` ont réussi, on peut utiliser M comme une matrice d'entiers, c'est à dire parler de $M[i][j]$ (au passage $M[i]$ est toujours un `int *`).

3.2 Libération de mémoire et utilisation de valgrind

Toute mémoire allouée doit être libérée, après la dernière utilisation, bien sûr. Dans le cas de notre matrice, cela donne :

```
/* désallocation mémoire de chaque sous-tableau*/
for (i = 0; i < N; i++)
  free(m[i]);

/* désallocation mémoire finale de m */
free(m);
```

Pour effectuer un diagnostic mémoire de votre programme, vous pouvez lancer l'utilitaire valgrind qui va inspecter la mémoire durant son exécution :

```
valgrind ./tp3
```

Dans le cas du code précédemment cité, avec N = 10, voici le diagnostic :

```
==2876==   total heap usage: 11 allocs, 11 frees, 440 bytes allocated
==2876==
==2876== All heap blocks were freed -- no leaks are possible
```

Le programme contient bien 11 allocations et 11 désallocations, pour un total de 10 pointeurs d'entiers (donc 40 octets), et 10 tableaux de 10 entiers (400 octets). Il n'a pas de fuite mémoire.

Maintenant, si l'on oublie de désallouer le dernier pointeur (deuxième `free` plus haut, on obtient :

```
==2896==    definitely lost: 40 bytes in 1 blocks
```

effectivement, le dernier free libère un tableau de 10 pointeurs d'entiers. Valgrind est donc un très bon outil pour effectuer des diagnostics mémoire. On s'en servira le plus souvent possible.

3.3 Utilisation de realloc

Le cas d'utilisation de l'exercice 2 est vraiment adapté à l'utilisation de l'utilitaire `realloc`, dont voici la signature et une partie de la documentation :

```
void *realloc(void *ptr, size_t size);
```

> The realloc function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If ptr is NULL, then the call is equivalent to malloc(size) [...] Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

En effet, le développeur a prévu un tableau de taille 5 de caractères pour stocker l'information :

```
s = (char*) malloc (5 * sizeof (char))
```

et l'utilisateur lui donne 6 caractères, donc il y a dépassement de capacité si rien n'est fait. Le développeur dans ce cas peut faire un `realloc`, qui en plus garde intactes les 5 premières cases :

```
if (nblus ==  5) {
s = (char*) realloc (s, 10 * sizeof (char));
}
```

à adapter pour traiter l'exo dans tous les cas (`realloc` d'un multiple de 5 cases). Attention, avant d'imprimer la chaîne, à bien insérer le marqueur de fin de chaîne dans le tableau.

La désallocation s'effectue ensuite tout simplement avec :

```
free (s)
```

Ce qu'il faut retenir

- Les variables locales, les variables globales sont allouées dans un espace restreint imposé par le compilateur. Si on veut pousser les limites, il faut faire de l'allocation dynamique, c'est à dire de l'allocation à l'exécution.
- Comment allouer un tableau d'entiers de N cases, et adapter pour allouer un tableau de ... de ... cases.
- Toujours récupérer la valeur de retour de malloc pour vérifier que l'on n'a pas atteint la limite de la mémoire.
- Savoir utiliser `realloc`
- Valgrind est un utilitaire sympa pour vérifier que l'on a bien libéré la mémoire.