TP6 - Itérateurs

Ce qu'on voit lors de ce TP

- La notion d'itérateur
- Un retour sur toutes les structures de données de ce cours (à part Union Find)
- Des notions de C avancées (union, macros, `_Generic` ...)

Avant de commencer, **télécharger l'archive associée à ce TP dans un dossier, auquel vous rajouterez les fichiers codant les différentes structures vues précédemment dans ce cours (à part *union find*), à savoir `liste.c`, `liste.h`, `liste_chainee.c`, `liste_chainee.h`, `ensemble.h`, `ensemble.c`, `file_priorite.c`, `file_priorite.h`, `avl.c` et `avl.h`.**

I) Des itérateurs, en C !?

Normalement, la notion d'itérateur devrait vous être familière (ne serait-ce parce que vous êtes des pros de java et que vous avez vu ça dans la partie *Patrons de conceptions*). Un *itérateur* permet de parcourir facilement des éléments appartenant dans une structure de données (souvent appelée *"conteneur"*). C'est quelque chose qui n'existe pas nativement en C, et pourtant nous allons émuler ça dans ce TP ! Je vous propose **de zyeuter `main.c` et d'exécuter le programme.**

Premièrement, on observe ce bout de code :

```
    for( EACH(i,8) ){
        printf("%d ",i);
    }
```

qui affiche

```
0 1 2 3 4 5 6 7
```

Puis, dans un second temps :

```
    char* phrase_exemple = "Ca marche ?!";
    for( EACH(lettre,phrase_exemple) ){
```

```
        printf("%c%c",lettre,lettre);
    }
```

qui affiche

```
CCaa  mmaarrcchhee  ??!!
```

On constate plusieurs choses surprenantes :

1. La boucle `for` voit en son intérieur un `EACH`, qui plus est, sans aucun point virgule `;`. C'est normal, `EACH` est une macro, les deux points virgules sont cachées à l'intérieur, on reviendra dessus.

2. Une variable est automatiquement créée, locale à la boucle, et qui porte le même nom que le première paramètre du `EACH`. Dans le premier c'est `i`, dans l'autre c'est `lettre` ; ça marcherait même si on mettait des `i` partout.

3. La boucle réagit de manière différente selon le type du deuxième paramètre du `EACH`. Dans le premier cas, `8` étant un entier, `i` va prendre toutes les valeurs de 0 jusqu'à $8 - 1 = 7$ (comme un *range* en python). Dans le second cas, `phrase_exemple` est une chaîne de caractères, et alors `lettre` va successivement valoir chacun des caractères de `phrase_exemple`, de la gauche vers la droite.

Il y a beaucoup de magie noire cachée derrière ce `EACH`, je ne vous expliquerai que vers la toute fin (si vous êtes motivés) comment on peut faire ce genre de choses (qui sont vraiment non conventionnelles, pas sûr que vous pouvez trouver ça sur internet !)

Ce que je vous propose, c'est d'enrichir ce `EACH` en y incorporant les structures de données que vous avez codées tout le long de ce cours. Mais avant cela, il faut comprendre comme se code un itérateur.

II) Anatomie d'un itérateur

Un itérateur fonctionne classiquement sur 5 méthodes :

1. Un constructeur `init` qui va initialiser l'itérateur, qui va "pointer" sur le premier élément de la structure

2. Une méthode qui renvoie l'élément pointé par l'itérateur

3. Une méthode `next` qui va faire passer l'itérateur à l'élément suivant

4. Un méthode qui va déterminer si l'itérateur a parcouru tous les éléments ou s'il reste encore des éléments à lire.

5. Un destructeur, qui va libérer la mémoire liée à l'itérateur.

Regardons maintenant l'entête `iterateurs.h` et plus précisément la structure `struct iterateur_s` :

```
struct iterateur_s{
    type_base valeur;
    bool est_fini;
    // Oubliez ce bout-là pour l'instant d'ici ...
    union donnees_supplementaires{
        char* curseur_string;
    } data;
    // ... jusque là
};
typedef struct iterateur_s* iterateur;
```

Cette structure va stocker les informations nécessaires à la réalisation des cinq méthodes précédentes. Pour l'instant, oubliez l'*union*, nous reviendrons sur ça plus tard. Elle est là parce que toutes les structures de données sur lesquelles on veut itérer vont partager la même structure d'itérateur.

Tout d'abord le champ `valeur` va indiquer la valeur de l'élement sur lequel "pointe" l'itérateur actuellement. Autrement dit, la méthode n°2 peut être simplement simulé en écrivant `it->valeur`, si `it` est de type `iterateur`, c'est-à-dire un pointeur sur une structure `iterateur_s`.

Dans la même idée, le champ `est_fini` encode la méthode n°4 : cela vaut `true` s'il n'y a plus d'élément suivant ; `false` si l'itérateur a encore des éléments à parcourir.

Pour les trois autres méthodes, on va les coder dans trois fonctions, qui vont dépendre du type du conteneur. Leur noms seront de la forme `[type]_init` (méthode n°1), `[type]_suivant` (méthode n°3) et `[type]_termine` (méthode n°5), où `[type]` est le type du conteneur (entier, string, et pour vous : liste, ensemble...).

Illustrons chacune de ces fonctions à travers un exemple. **Ouvrez maintenant `iterateurs.c` et regarder le bout de code concernant les entiers.**

La méthode `init` est une fonction qui initialise un itérateur en fonction d'un entrée unique : le *conteneur* (ici `n`) :

```
iterateur entier_init(int n){
    iterateur it = malloc(sizeof(struct iterateur_s));
    it->valeur = 0; // Première valeur : 0
    it->est_fini = n <= 0; // L'itérateur se termine immédiatement si n<1
    return it;
}
```

La méthode `suivant` actualise l'itérateur de sorte à ce qu'il pointe sur l'élement suivant. Cette fonction prend en paramètres l'itérateur et l'objet sur lequel on itère (parfois ce dernier est inutile...) :

```
void entier_suivant(iterateur it, int n){
    (it->valeur)++;
    it->est_fini = (it->valeur == n);
}
```

Enfin le destructeur, simple, va être tout le temps le même (vous pourrez normalement le copier-coller).

```
void entier_termine(iterateur it){
    free(it);
}
```

Ici le destructeur libère la mémoire du seul *malloc* effectué lors de l'initialisation de l'itérateur. Si jamais d'autres allocations dynamiques ont été réalisées à la création ou lors de l'exécution de l'itérateur, c'est ici qu'il faudra toutes les libérer.

Une fois que ces trois fonctions écrites, on pourra simplement écrire un itérateur des entiers de `0` à `nb-1` comme :

```
int nb = 20; // Nombre "conteneur"
iterateur it = entier_init(nb);
while( it->est_fini == false ){
    int i = it->valeur; // i est le "compteur" de notre itérateur

    // Coeur de la boucle ici...

    entier_suivant(it,nb);
}
entier_termine(it);
```

C'est un peu lourd oui oui. Mais c'est pour ça que j'ai écrit la macro `EACH` qui fait exactement la même chose :

```
int nb = 20;
for( EACH(i,nb) ){
    // Coeur de la boucle ici...
}
```

(Petite différence : si on fait un break dans le `for EACH`, il y aura une fuite mémoire car le destructeur ne sera pas appelé.)

On a fait les entiers et c'est pas si dur. Les autres structures de données, comme les chaînes de caractères, vont demander de stocker des informations supplémentaires, comme la position dans le mot. C'est l'intérêt du dernier champ de `struct iterateur_s` : il se nomme `data` et c'est une union.

III) Union en C

Regardez maintenant la méthode `next` pour les chaînes de caractères.

```
void string_suivant(iterateur it, char* ch){
    char* ptr = it->data.curseur_string;

    ptr++;
    it->valeur = *ptr;
    it->est_fini = (*ptr == '\0');

    it->data.curseur_string = ptr;
}
```

En plus de sa valeur, on stocke dans l'itérateur une autre variable nommée `curseur_string` de type `char*` qui indique où on se trouve dans la chaîne de caractères.

Toutefois, quand il s'agit d'écrire l'itérateur d'autres structures, il est probable qu'on utilise autre chose qu'un `char*` , comme un entier ou un pointeur de type différent . Donc plutôt que de faire une structure `iterateur_s` avec autant de champs que de types de "conteneur", on utilise une **union** pour condenser la mémoire.

Une union se déclare de manière similaire à une structure, avec listant plusieurs champs avec leurs types et leurs noms, sauf que ces champs vont partager le même espace mémoire. Leurs mémoires seront superposées ce qui fait qu'on ne manipulera qu'un seul de ces champs à la fois.

Prenons un exemple pour illustrer cela :

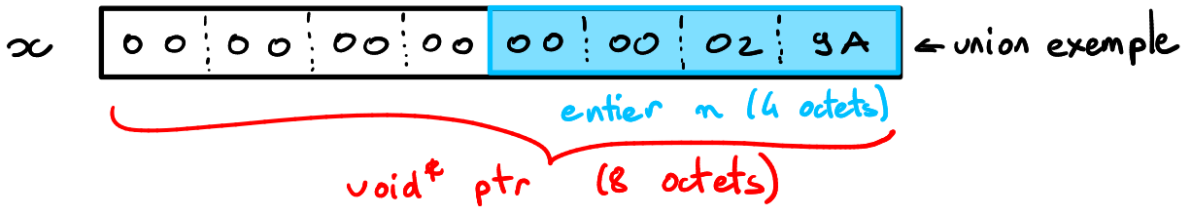
```
    // Déclaration de l'union :
    union exemple{
        void* ptr;
        int n;
    };

    // On déclare x qui est de type "union exemple"
    union exemple x;
    // On change la valeur du champ 'n' par 666
    x.n = 666;
    printf("x.ptr: %p, x.n : %d\n",x.ptr,x.n);
```

Ce bout de code affichera :

```
x.ptr: 0x29a, x.n : 666
```

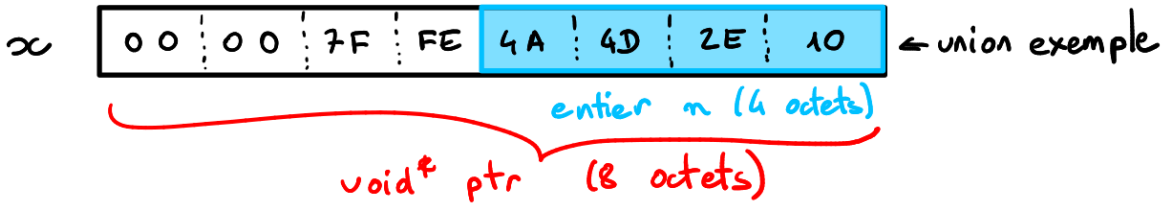
En effet, dans la mémoire, la variable `x` aura ses deux champs `n` et `ptr` au même endroit, donc en initialisant `x.n`, on initialise aussi `x.ptr` (attention les champs ne font pas tous la même taille en termes d'octets occupés, ça dépend du type).



Et bien sûr, si on change un des deux champs, l'autre changera aussi. Par exemple, si on écrit à la suite :

```
    x.ptr = &x;
    printf("x.ptr: %p, x.n : %d\n",x.ptr,x.n);
```

alors `x` ressemblera à



et on verra affiché quelque chose comme :

```
x.ptr: 0x7ffe4a4d2e10, x.n : 1246572048
```

(1246572048 s'écrit en hexadécimal `0x4A4D2E10`) Revenons à notre structure `iterateur_s` :

```
struct iterateur_s{
    type_base valeur;
    bool est_fini;
    // On déclare ici une union "données supplémentaire"
    union donnees_supplementaires{
        char* curseur_string;

        // À vous de rajouter des champs ici selon la structure de données !

    } data;
};
```

```
typedef struct iterateur_s* iterateur;
```

Si vous avez besoin de stocker une variable supplémentaire pour le bon fonctionnement de votre itérateur, vous n'avez qu'à modifier l'entête `iterateurs.h` insérer votre variable dans cette union. Ainsi vous pourrez y accéder en écrivant `nom_iterateur->data.nom_variable` .

Et que faire si vous avez besoin de plusieurs variables ? Vous pouvez les regrouper en une seule variable via une structure ! Il faudra juste la déclarer au sein de l'union.

IV) Bon on code ?

Maintenant que vous avez toutes les informations nécessaires, **écrivez dans `iterateurs.c` les 5 itérateurs associés :**

1. aux listes
2. aux listes simplement chaînées,
3. aux ensembles,
4. aux files de priorité (les éléments ne sont pas forcément parcourus de manière croissante)
5. aux AVL dans l'ordre croissant (c'est "juste" un parcours infixe dans un arbre binaire de recherche)

N'oubliez pas de tester vos itérateurs dans le fichier `main.c` . La macro `for(EACH)` est bien pratique pour cela !

V) Bonus : qu'y a-t-il caché derrière la macro ?

Si vous êtes de nature curieuse, je peux tenter de vous expliquer ce qui est caché derrière les macros de `iterateurs.h` .

J'utilise plus précisément des macros avec paramètres. Ca ressemble à des fonctions, mais attention ça n'en est pas. Rappelez-vous que les macros ne sont que des remplacements textuels qui ont lieu **avant** la compilation, lors de l'étape de prétraitement. Les fonctions elles sont appelées au cours de l'exécution du programme. Les macros à paramètres sont donc des "*rechercher remplacer*" un peu évolués où on module comment remplacer certaines expressions.

Tout d'abord regardons la macro `ACCOLE_TYPE` :

```
#define ACCOLE_TYPE(nom_fonction, x) _Generic((x), \
                int: entier_##nom_fonction, \
                char*: string_##nom_fonction, \
```

```
                Liste: liste_##nom_fonction, \
                ListeChainee: liste_chainee_##nom_fonction, \
                Ensemble: ensemble_##nom_fonction, \
                FilePriorite: file_priorite_##nom_fonction, \
                avl: avl_##nom_fonction \
                )
```

Cette macro utilise `_Generic` un outil très puissant qui effectue des opérations différentes en fonction du type de son premier argument. Ici `x` va être le type de notre conteneur. Le second argument de `_Generic` indique que faire selon le type de `x`. Par exemple, si `x` est une variable entière, alors notre macro va être associée à `entier_##nom_fonction`, où `nom_fonction` est le premier argument de la macro `ACCOLE_TYPE`.

Que fait le double dièse ? C'est quelque chose utilisé dans les macros pour concaténer des symboles ou des chaînes de caractères. En reprenant toujours l'exemple où `x` est une variable entière, alors `ACCOLE_TYPE(init,x)` est une macro qui sera remplacée textuellement par `entier_init` (qui est une fonction qu'on a écrite) ! Autre exemple pour être sûr que vous avez compris, si `ch` est une chaîne de caractères, alors `ACCOLE_TYPE(suivant,ch)` désigne `string_suivant`.

Ca c'était la partie facile ! Cette macro sert assez régulièrement (je l'utilise beaucoup dans mes tests *caseine* par exemple). Il n'est pas inutile de la retenir !

Maintenant passons à la macro `EACH` elle-même :

```
#define ITER(x) ( (iterateur) __iter__ ## x )

#define EACH(x,l) type_base* __iter__ ## x = (type_base*) ACCOLE_TYPE(init,l
                !ITER(x)->est_fini || ACCOLE_TYPE(termine,l) ( ITER(x) ) ;\
                ACCOLE_TYPE(suivant,l) (ITER(x),l), x = ITER(x)->valeur
```

`ITER` est une macro à paramètres assez simple ; `ITER(ma_var)` est juste un raccourci textuel pour " `(iterateur) __iter__ma_var` ".

`EACH` est une macro complexe faisant intervenir `ITER` et `ACCOLE_TYPE`. Afin d'y voir plus clair, je vous propose de l'illustrer pour un `x` qui est entier. On va reprendre la boucle décrite plus haut :

```
int nb = 20;
for( EACH(i,nb) ){
    // Coeur de la boucle ici...
}
```

Lors du prétraitement, cette boucle va être remplacée par :

```
int nb = 20;
for( type_base* __iter__i = (type_base*) entier_init(nb), \
                    i = ((iterateur)__iter__i)->valeur;  \
                   !((iterateur)__iter__i)->est_fini || \
                   (entier_termine((iterateur)__iter__i), false); \
                   entier_suivant((iterateur)__iter__i,nb), \
                   i = ((iterateur)__iter__i)->valeur \
         ){
    // Coeur de la boucle ici...
}
```

OK c'est pas forcément plus clair mais on va décortiquer cela. Tout d'abord, la première difficulté est qu'il faut déclarer deux variables à l'intérieur des parenthèses du `for` : une pour l'itérateur (de nom `__iter__i`) et une pour la variable elle-même (de nom `i`). Or en C, on ne peut pas déclarer plus de deux variables dans une parenthèses de la boucle `for` **sauf** si ce sont de variables de même type.

Par exemple, ça on a le droit :

```
for(int i = 0, j = 0; i + j < 18; i++, j++)
```

mais ça c'est interdit :

```
for(int i = 0, float k = 0.0; i + k < 18; i++, j++)
```

Or notre iterateur est de type `iterateur` et notre `i` de type `type_base` . L'astuce c'est qu'on va faire croire que ce sont deux objets de même type : en castant (ou transtypant) notre itérateur en tant que `type_base*` ! C'est pour ça qu'on commence par écrire `type_base* __iter__i = (type_base*) entier_init(nb)` . Ainsi `__iter__i` est en réalité de type `type_base*` et notre vrai itérateur de type `iterateur` n'est d'autre que `(iterateur)__iter__i` !

Imaginons qu'on puisse en réalité déclarer deux variables différentes au sein d'une même boucle `for` et je vais réécrire la boucle précedente ; on va y voir plus clair :

```
int nb = 20;
for( iterateur it = entier_init(nb), type_base i = it->valeur;  \
                  !it->est_fini || (entier_termine(it), false); \
                    entier_suivant(it,nb), i = it->valeur ){
    // Coeur de la boucle ici...
}
```

La première ligne sert à déclarer donc l'itérateur et la variable "compteur".

Pour les deux autres lignes, j'ai utilisé un opérateur particulier : l'**opérateur virgule**. Il est utilisé pour séparer les expressions, exécutant chacune d'elles de gauche à droite et renvoyant la valeur de la dernière expression évaluée.

Regardons la dernière ligne `entier_suivant(it,nb), i = it->valeur` . Elle est relativement simple ; c'est l'équivalent du `i++` dans les boucles `for` classiques. `entier_suivant(it,nb)` fait bouger l'itérateur vers l'élément suivant et `i = it->valeur` actualise la valeur de `i` .

Il reste maintenant que la deuxième ligne à comprendre. Le booléen `!it->est_fini` sert juste à s'assurer que la boucle s'arrête quand l'itérateur est fini. Seulement une fois que la boucle est stoppée, il faut libérer la mémoire pour éviter les fuites !

On utilise alors le fait que l'évaluation du 'ou', représenté par les deux majestueuses barres verticales `||` , soit paresseuse. Si `bool1` vaut `true` , alors le programme ne va pas s'embêter à vérifier ce que vaut `bool2` pour savoir si `bool1 || bool2` vaut `true` ! Il sait déjà que c'est `true` et va passer à la suite.

J'ai utilisé ça à mon avantage de sorte que `(entier_termine(it), false)` ne soit évalué que si `!it->est_fini` vaut `false` , autrement dit quand la boucle est finie ! L'opérateur virgule s'assure ici que la valeur finale est bien égale à `false` .

Bravo en tout cas si vous avez tout lu ! (Vous n'aurez pas de points en plus malheureusement.)