Itérateurs

Le but de cette séance de Travaux Pratiques est d'implémenter des itérateurs pour différentes structures de données.

Vous trouverez sur Moodle une archive contenant un certain nombre de fichiers pour vous aider à démarrer. Après l'avoir téléchargée :

```
$ tar -xf sda1-tp5.tar.gz
$ cd sda1-tp5
```

Les fichiers sont répartis sur plusieurs répertoires correspondant aux différents exercices décrits dans la suite de ce document. Le fichier `Makefile` à la racine permet de compiler l'ensemble des exercices. Des fichiers `Makefile` sont également disponibles dans chacun des répertoires.

```
$ make
$ ./test_iterateur
```

Les itérateurs sont des objets permettant de parcourir (d'*itérer* sur) l'ensemble des éléments contenus dans un autre objet. Le but est de séparer la notion de parcours des élements d'un objet du fonctionnement de l'objet.

On souhaite pouvoir parcourir les éléments d'un objet d'une manière généraliste sans avoir à se soucier du type de l'objet (tableau, liste, etc.) :

```
for (iterateur it = debut(objet); it != fin(objet); it = suivant(it))
  afficher(valeur(it));
```

1 Itérateur basique

Commencez par les itérateurs basiques pour des tableaux d'élements de sorte `S` et de taille fixe. Dans ce cas, l'itérateur est simplement un pointeur :

```
typedef S* iterateur;

iterateur iterateur_debut(size_t taille, S tableau[static taille]);
iterateur iterateur_fin(size_t taille, S tableau[static taille]);
iterateur iterateur_suivant(iterateur i);
iterateur iterateur_precedent(iterateur i);
S iterateur_valeur(iterateur i);
```

Proposez une définition des fonctions déclarées ci-dessus.

2 Première tentative d'itérateur générique

Les itérateurs présentés dans l'excercice précédent ne sont adaptés qu'aux tableaux de sorte `S` de taille fixe. On souhaite maintenant créer des itérateurs qui peuvent être utilisés indifférement avec des tableaux ou des listes d'éléments de sorte `S` :

```
typedef struct iterateur iterateur;

iterateur* liste_iterateur_debut(const liste* l);
iterateur* liste_iterateur_fin(const liste* l);
iterateur* tableau_iterateur_debut(const tableau* t);
iterateur* tableau_iterateur_fin(const tableau* t);
#define iterateur_debut(x) \
  _Generic((x), \
      liste*: liste_iterateur_debut, \
      tableau*: tableau_iterateur_debut \
      )(x)
#define iterateur_fin(x) \
  _Generic((x), \
      liste*: liste_iterateur_fin, \
      tableau*: tableau_iterateur_fin \
      )(x)
iterateur* iterateur_suivant(iterateur* i);
iterateur* iterateur_precedent(iterateur* i);
bool iterateur_egal(const iterateur* a, const iterateur* b);
S iterateur_valeur(const iterateur* it);
void iterateur_free(iterateur* i);
```

Voici la structure suggérée :

```
struct iterateur {
  iterateur_type type;
  union {
    const liste* l;
    struct {
      const tableau* t;
      size_t position;
    };
  };
};
```

Proposez une implémentation pour les itérateurs décrits précédemment en modifiant le fichier `ex2/iterateur.c`. L'implémentation des listes et des tableaux est fournie.

Note : constatez-vous un problème avec l'opération `iterateur_precendent()` pour les listes ?

3 Itérateurs génériques

Dans l'exercice précédent, les itérateurs ont été définis séparement et leur implémentation est intimement liée à l'implémentation des conteneurs visés. Normalement, c'est celui qui développe un nouvel objet qui définit l'itérateur correspondant.

Par exemple, si on définit un nouveau type `ensemble`, on pourrait souhaiter parcourir tous les élements de cet ensemble. Est-il possible de définir un itérateur selon la méthode précédente pour les ensembles dont l'interface est donnée ci-dessous ? Pourquoi ?

```
typedef struct ensemble ensemble;

ensemble* ensemble_vide(void);
void ensemble_free(ensemble* e);

ensemble* ensemble_insertion(ensemble* e, S x);
ensemble* ensemble_suppression(ensemble* e, S x);

bool ensemble_contient(const ensemble* e, S x);
size_t ensemble_cardinal(const ensemble * e);
bool ensemble_est_vide(const ensemble * e);
```

Contrairement aux listes et tableaux où il est possible d'obtenir très exactement tous les éléments, dans le cas des ensembles tels que décrits ci-dessus il est difficile de le faire sans connaître les détails d'implémentation : c'est donc la personne qui développe les ensembles qui doit fournir une implémentation spécifique des itérateurs pour les ensembles.

Voici la nouvelle interface proposée pour des itérateurs génériques :

```
typedef struct iterateur iterateur;

iterateur* iterateur_nouveau(
    void* infos_conteneur, void (*suivant)(void*), void (*precedent)(void*),
    S (*valeur)(const void*), bool (*egal)(const void*, const void*));

iterateur* iterateur_suivant(iterateur* i);
iterateur* iterateur_precedent(iterateur* i);
bool iterateur_egal(const iterateur* a, const iterateur* b);
S iterateur_valeur(const iterateur* it);
void iterateur_free(iterateur* i);
```

Et pour l'implémentation, Voici la nouvelle structure suggérée pour les itérateurs :

```
struct iterateur {
  void* infos_conteneur;
  void (*suivant)(void*);
  void (*precedent)(void*);
  S (*valeur)(const void*);
  bool (*egal)(const void*, const void*);
};
```

Le champ `infos_conteneur` peut contenir diverses informations concernant un conteneur. Lors de la construction d'un itérateur, on fournit des fonctions spéciales `suivant`, `precedent`, `valeur` et `egal` prévues pour modifier ou analyser ce champ `infos_conteneur`.

Pour chacun des conteneurs il faut donc stocker les informations que l'on juge pertinentes dans une *autre* structure dont l'adresse sera stockée dans le champ `infos_conteneur`. Il faut alors écrire différentes fonctions capables d'interpréter ces informations pour pouvoir avancer ou reculer l'itérateur et enfin remplir les champs de la structure `struct iterateur`.

Ensuite, les fonctions prenant en paramètre un pointeur vers une structure `struct iterateur` utiliseront les fonctions pointées par les différents champs en y passant en paramètre le champ `infos_conteneur`.

A Le mot-clé `_Generic`

Le mot-clé `_Generic` a été introduit dans C11 pour l'écriture d'expressions génériques. La syntaxe est proche de celle employée pour la structure de contrôle `switch`. Un exemple d'utilisation est disponible en Section 2 ou dans le code fourni.

Ce mot-clé est utilisé, par exemple, dans l'en-tête `tgmath.h` pour définir des versions génériques des utilitaires fournis par l'en-tête `math.h`. Voici un exemple de code que l'on écrirait en utilisant l'en-tête `math.h` :

```c
#include <stdio.h>
#include <math.h>
int main(int argc, char* argv[argc + 1]) {
  const float f[2] = { -1.0f, 1.0f, };
  const double d[2] = { -1.0, 1.0, };
  const long double l[2] = { -1.0L, 1.0L, };
  fprintf(stdout, "abs(%f) = %f, abs(%f) = %f\n", f[0], fabsf(f[0]), f[1], fabsf(f[1]));
  fprintf(stdout, "abs(%f) = %f, abs(%f) = %f\n", d[0], fabs(d[0]), d[1], fabs(d[1]));
  fprintf(stdout, "abs(%Lf) = %Lf, abs(%Lf) = %Lf\n", l[0], fabsl(l[0]), l[1], fabsl(l[1]));
  return 0;
}
```

Voici un code équivalent qui utilise `tgmath.h` en lieu et place de `math.h` (cette fois-ci, on utilise toujours `fabs()`) :

```c
#include <stdio.h>
#include <tgmath.h>
int main(int argc, char* argv[argc + 1]) {
  const float f[2] = { -1.0f, 1.0f, };
  const double d[2] = { -1.0, 1.0, };
  const long double l[2] = { -1.0L, 1.0L, };
  fprintf(stdout, "abs(%f) = %f, abs(%f) = %f\n", f[0], fabs(f[0]), f[1], fabs(f[1]));
  fprintf(stdout, "abs(%f) = %f, abs(%f) = %f\n", d[0], fabs(d[0]), d[1], fabs(d[1]));
  fprintf(stdout, "abs(%Lf) = %Lf, abs(%Lf) = %Lf\n", l[0], fabs(l[0]), l[1], fabs(l[1]));
  return 0;
}
```