TP1 : listes chaînées

1 Quelques rappels de base

Durant les TPs de cette année, vous aurez le choix de programmer en `C` pur ou en `C++`. Si les langages sont proches, ils ont tout de même leurs spécificités. Le `C++` est en majeure partie une surcouche du `C`, et une grosse partie (mais pas tout) de ce qui compile en `C` compilera avec un compilateur `C++`. Si vous avez suivi votre cursus à Lyon1, vous avez utilisé en LIF1 et LIF5 du `C` enrichi d'une petite couche de `C++` :

- les références (matérialisées par l'utilisation de l'esperluette `&` dans le type d'une variable) ;
- la surcharge de fonctions (possibilité de donner le même nom à des fonctions différentes) ;
- les entrées/sorties utilisant les classes `cin` et `cout` (utilisées en LIF1) ;
- la possibilité de commenter une ligne par `//` ;
- la possibilité de ne pas faire précéder le nom d'un type correspondant à une structure du mot clé `struct` ;
- l'allocation dynamique de mémoire par les opérateurs `new` et `delete`.

Nous récapitulerons dans cette section les éléments de langage que vous utiliserez selon que vous choisissez de programmer en `C` ou en `C++`.

1.1 Commentaires

Les commentaires sont des portions de fichier qui ne sont pas interprétées par le compilateur. Ils vous permettent de documenter votre code, et de rajouter des explications pour le rendre plus lisible. Prenez l'habitude d'écrire des commentaires dans votre code. Une bonne façon de développer consiste par exemple à commencer par remplir une fonction avec des commentaire indiquant ce qu'il faut ajouter, puis à rédiger le code sous les commentaires correspondants.

En `C` les commentaires sont délimités par `/*` et `*/`. Ils peuvent faire plusieurs lignes, mais il n'est pas possible de les imbriquer (et donc de placer des `/* ... */` dans un commentaire) :

```
/* commentaire sur une ligne */
/* commentaire
   sur deux lignes */
```

En `C++` ou en `C99` il est possible en plus de rajouter de courts commentaire en utilisant `//`. Une fois écrit `//`, le reste de la ligne n'est plus interprété.

```
int a ; //un commentaire de fin de ligne
```

1.2 Passage de paramètres, pointeurs, références

On parle de *passage de paramètres* lorsqu'on appelle une fonction en lui fournissant des paramètres. Selon les langages de programmation, il existe de types de passage de paramètre :

Le passage par valeur *recopie* la valeur fournie en paramètre à la fonction ou à la procédure. Ainsi, une fonction modifiant la valeur de ses paramètre ne modifiera pas la valeur des variables utilisées pour fournir ces paramètres dans la fonction appelante. C'est cette stratégie qui est appliquée en `C` et en `C++`. Par exemple en `C` :

```
void f(int a) {
  a = a+1 ;
}

int b = 0 ;
f(b) ;
/* b vaut toujours 0 */
```

Le nom *passage par valeur* vient du fait qu'on considère que c'est la *valeur* de la variable qui est fournie à la fonction lors de l'appel.

Le passage par nom *ne recopie pas* la valeur fournie en paramètre, mais considère que la fonction appelée peut modifier la variable fournie en paramètre. Cette modification impactera la valeur de cette variable dans le programme appelant. Par exemple en `JavaScript` :

```
function f(a) {
  a = a + 1 ;
}

var b = 0 ;
f(b) ;
// b vaut maintenant 1
```

Le nom *passage par nom* vient du fait qu'on considère que c'est le *nom* de la variable qui est fourni en paramètre, et qu'à partir du nom de la variable la donnée référencée est accessible.

Pour plus de détails, vous pouvez utiliser les mots clé `call by name` et `call by value` dans vos recherches. Pour vous la conclusion à retenir est la suivante :

> En `C` et en `C++` le passage de paramètre est un *passage par valeur.*

Vous pouvez cependant avoir de temps besoin d'avoir le même comportement que le passage par nom, si vous souhaitez qu'une fonction modifie des données en dehors de sa portée, ou si vous voulez éviter la copie de données volumineuses passées en paramètre. Nous allons donc voir comment simuler ce comportement en `C` et en `C++`.

En `C`

Le `C` associe à chaque objet manipulé une *adresse mémoire*. Vous pouvez littéralement considérer que chaque octet de la mémoire associée à votre programme est numéroté, et que cette adresse est le numéro du premier octet où votre objet est stocké. Il est donc possible de donner la possibilité à une fonction de modifier des données en dehors de sa portée normale en lui fournissant l'adresse mémoire des données qu'elle peut modifier. Ces adresses sont communément appelées des *pointeurs*. Étant donné une variable, vous pouvez obtenir son adresse en utilisant le carractère *esperluette* (`&`). Si `type` est le nom du type des données, `type *` est le nom du type d'une adresse sur ces données. Par exemple :

```
int a = 5 ; /* declaration d'une variable int*/
int * adresse_a = &a ; /* enregistrement de sont adresse dans une autre variable */
```

L'accès aux données étant donnée une adresse se fait également avec le carractère `*` :

```
int b = *adresse_a ; /* recuperation des donnees de a depuis l'adresse, b vaut 5 */
*adresse_a = 12 ; /* a vaut maintenant 12, b vaut toujours 5 */
```

Vous pouvez ainsi simuler un passage par nom via un *passage par adresse* :

```
void f(int * a) {
  *a = *a + 1 ;
}

f(adresse_a) ; /* a vaut maintenant 13 */
f(&a) ; /* a vaut maintenant 14 */
```

En `C++`

Le `C++` permet d'utiliser les mêmes mécanismes que le `C` pour manipuler les adresses mémoire, et permet donc également la syntaxe du `C`. En plus de cette méthode, il est également possible d'utiliser des *références*. Une référence consiste à donner un autre nom à une donnée. Ainsi plusieurs variables peuvent partager la même donnée. Une référence se définit en utilisant une esperluette (`&`) au niveau du type de la nouvelle variable déclarée :

```
int a = 0 ;
int & b = a ; /* b et a sont maintenant deux noms pour une donnee qui vaut 0 */
a = 2 ; /* b vaut maintenant aussi 2 */
b = 1 ; /* a vaut maintenant aussi 1 */
```

Copier une référence dans une variable qui n'en est pas une réalise une copie des données :

```
int c = b ; /* c est une copie de a et b et vaut 1 */
c = 4 ; /* c vaut 4, a et b valent toujours 1 */
```

Contrairement à une variable classique, une référence ne provoque donc pas la création ou la copie de données, et partage la même adresse que la variable à partir de laquelle elle a été initialisée :

```
int * adresse_a = &a ; /* enregistrement de l'adresse de la variable a */
int * adresse_b = &b ; /* enregistrement de l'adresse de la reference b */
/* adresse_a et adresse_b ont la meme valeur */
```

Pour réaliser un passage par nom en C++, il suffit donc de définir une fonction prenant en paramètre une référence sur une donnée, et cette donnée ne sera ainsi pas copiée, mais simplement associée au nouveau nom :

```
void f(int & ref) {
  ref = ref + 1 ;
}

f(a) ; /* a vaut maintenant 2, b aussi */
f(b) ; /* b vaut maintenant 3, a aussi */
```

1.3 Gestion de la mémoire

En C et en C++, il existe plusieurs zones mémoire. Vous utiliserez généralement la *pile* et le *tas*.

1.3.1 La pile

C'est la zone mémoire dans laquelle sont allouées les données locales aux fonctions : en C et en C++, toute variable n'est valable qu'à l'intérieur de sa *portée*, matérialisée par les accolades ({, }). Les paramètres d'une fonction sont également limités à la portée de cette fonction :

```
int f(int a) {
  int b = 10 ;
  int c = 0 ;

  for(int i = 0; i < b, ++i) {
    c = c * a ;
  } /* i n'est plus definie */

  if(c < 0) {
    int d = -c ;
    c = c * d ;
  } /* d n'est plus definie */

  {
    int d = 2 * c ;
    c = c / d ;
  } /* d n'est plus definie */

  return c ;
} /* a, b et c ne sont plus definies */
```

Toute donnée que vous créez sans utiliser les fonctions `new` ou `malloc` est stockée sur la pile, et aura donc une durée de vie limitée.

1.3.2 Le tas

C'est une zone mémoire pour créer des données *persistantes*. À moins que vous ne donniez explicitement l'instruction de détruire les données présentes, elles persisteront tant que le programme fonctionnera. L'outil

`valgrind` vous permettra de détecter de nombreuses erreurs liées à la gestion de la mémoire, et nous vous encourageons à l'utiliser systématiquement.

En `C` la gestion du tas en se fait avec le couple de fonctions `malloc` et `free`. La fonction `malloc` prend en paramètre *le nombre d'octets* à réserver. Le langage `C` fournit la directive `sizeof` qui permet de connaître le nombre d'octets nécessaire pour un type atomique ou une structure. La valeur de retour de `malloc` est une adresse *générique* (de type `void *`), qui indexe le premier octet alloué dans le tas. Il convient ensuite de convertir cette adresse générique en une adresse typée correctement via un *cast*. Il est **indispensable** de récupérer la valeur de retour de `malloc`, sans quoi l'adresse de la zone allouée est perdue, et vous ne pourrez plus accéder à la zone, ou la libérer pour faire de la place en mémoire. Une allocation typique d'un objet sur le tas a donc la forme :

```
int * a = (int *) malloc(sizeof(int)) ;
```

Notez le type `int` qui est ici mentionné trois fois : la première pour définir le type de la variable déclarée (l'adresse d'un entier), la seconde pour *caster* l'adresse générique renvoyée par `malloc` en l'adresse d'un entier, et la troisième pour déterminer le nombre d'octets à réserver pour un entier via la directive `sizeof`.

En `C` et en `C++`, la norme du langage assure qu'un tableau de données est une zone mémoire *contiguë* (où les données sont rangées les unes à côté des autres, il n'y a pas d'espace vide entre les données). Il est donc possible de réserver un tableau dans le tas en allouant simplement le nombre d'octets nécessaire pour *l'ensemble* des données du tableau :

```
/* allocation d'un tableau de 10 entiers */
int * tab = (int *) malloc(10*sizeof(int)) ;
```

Les données sont ensuite accessibles comme d'habitude en utilisant les crochets (`[`, `]`) :

```
for(int i = 0; i < 10; ++i) {
  tab[i] = 2*i ;
}
```

De la même façon qu'une parenthèse ouverte à un moment donné doit être refermée plus tard, une donnée allouée avec `malloc` doit être libérée plus tard avec `free`. Dans le cas contraire, on parle de *fuite de mémoire*. Prenez donc l'habitude lorsque vous écrivez `malloc` quelque part d'écrire `free` ailleurs (ou de noter en commentaire où vous comptez le faire). La fonction `free` prend en paramètre une adresse *qui doit avoir été fournie par `malloc`*. C'est parce que `malloc` a réalisé l'allocation que vous n'avez pas à préciser le nombre d'octets à libérer. Quelque part, en sous main, le nombre d'octets correspondants à l'adresse a été enregistré. Vous pouvez ainsi libérer la mémoire allouée par les deux instructions précédentes via :

```
free(a) ;
free(tab) ;
```

Si vous utilisez `valgrind`, les octets qui ont été alloués durant l'exécution du programme et n'ont pas été libérés à la sortie du programme vous seront signalés.

En `C++` vous pouvez utiliser `malloc` et `free` mais ces fonctions sont généralement déconseillées. Il vous est conseillé d'utiliser les directives `new` et `delete`, qui initialisent et détruisent correctement les objets, et perturbent moins le système de types :

```
int * a = new int ;
int * tab = new int[10] ;
```

Notez ici que vous n'avez pas à mentionner le nombre d'octets, le type fourni à new suffit. Comme précédemment, chaque utilisation de `new` doit être un jour compensée par l'utilisation de `delete`. La suppression des variables précédentes se fait via :

```
delete a ;
delete[] tab ;
```

Notez bien l'utilisation de `delete[]` pour faire écho à `new <type>[]`.

1.4 Structures

Les structures sont à la base de l'élaboration de structures de données complexes. Une structure permet d'assembler plusieurs types existants pour former de nouveaux types de données. Les structures se déclarent de la même façon en `C` et en `C++` :

```
struct <nom de la structure> {
  <type 1> <nom du champ 1> ;
  <type 2> <nom du champ 2> ;
  ...
} ;
```

Par exemple :

```
struct rendez_vous {
  int debut ;
  int fin ;
  char* description ;
} ;
```

Le compilateur se chargera de déterminer le nombre d'octets occupés par la structure, et vous pourrez l'obtenir via `sizeof(struct rendez_vous)`. Il est ensuite possible de déclarer un objet de type structure et d'accéder à ses champs via la syntaxe :

```
/* allocation sur la pile */
struct rendez_vous rdv ;
/* le . permet d'acceder aux champs de la structure */
rdv.debut = 12 ;
rdv.fin = 13 ;

/* allocation sur le tas de type C */
struct rendez_vous * trdv1 = (struct rendez_vous*) malloc(sizeof(struct rendez_vous)) ;
(*trdv).debut = 12 ;
trdv->fin = 13 ; /* a-> est un raccourci pour (*a). */

/* allocation sur le tas de type C++ */
struct rendez_vous * trdv2 = new struct rendez_vous ;
```

Une structure peut être définie récursivement, tant qu'elle ne contient que des *adresses* sur des structures similaires. Sinon, il serait bien impossible de déterminer le `sizeof` de la structure pour cause de récursion infinie.

```
struct personne {
  int num_secu ;
  struct personne * parent1 ;
  struct personne * parent2 ;
} ;
```

En `C++` il est possible d'omettre le mot clé `struct` pour faire référence à un objet de type structure. Vous pourrez ainsi utiliser le code suivant :

```
personne p ;
personne * ptr_p = new personne ;
```

En `C` vous pouvez imiter ce comportement en utilisant le mot clé `typedef` qui permet de renommer des types. Par exemple :

```
typedef struct personne personne ;
personne p ;
personne * ptr_p = new personne ;
```

2 A vous de jouer !

2.1 Liste chaînée

On se propose de réaliser l'implantation du type abstrait liste, en `C++` ou en `C` pur. Vous pouvez récupérer les archives correspondantes sur la page du cours :

`http://liris.cnrs.fr/~vnivolie/lifap6`

Ces archives contiennent les fichiers d'interface de base à respecter. Une liste chaînée est une structure de données permettant de stocker une séquence de valeurs. Chaque cellule de la liste contient une valeur, et permet d'accéder à la cellule suivante :

Commencez par écrire le code de la procédure d'initialisation d'une liste en une liste vide, la procédure d'ajout en tête, la procédure d'affichage, puis la procédure testament (nettoyage de la mémoire associée). Le reste des autres procédures n'est à écrire que si vous avez fini le reste du TP. Testez bien votre code au fur et à mesure en utilisant le programme de test.

2.2 Liste chaînée circulaire

De manière à enrichir votre bibliothèque de modules, reprenez à présent votre module `liste` avec une implantation différente, chaînée circulaire et utilisant une cellule bidon (sentinelle). Voyez vous l'intérêt d'une implantation circulaire en terme de complexité ?