POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S6) / 2013 **Programmation Avancée**

Jonathan Aceituno, Vincent Aranéga, Nathalie Devésa, Rachid Elkouri et Laure Gonnord

Bilan TPs 4 à 10 de PA

1 TP 4/6/7 : Listes chaînées

Les listes chaînées sont la SD de base et il faut savoir les programmer parfaitement en C.

1.1 Déclaration/ insertion

Il faut savoir par coeur la déclaration d'un type liste d'entiers, et savoir reconstruire rapidement pour une liste de n'importe quoi (notamment liste de struct) :

```
/*Cell and List types*/
typedef struct cell {
  struct cell *next ;
  int value; // entier. à remplacer par le type de base de votre liste
}  Cell;

typedef Cell *PtCell, *List;
```

Attention à bien initialiser la liste à NULL lors de la déclaration.

Insérer dans une liste triée n'est pas bien difficile si on le pense en récursif :

```
void insertInSortedList(int el,List* p_list){

  if (*p_list == NULL || ((*p_list)-> value).intval > el) {
    //insertion en tete
  }
  else { // sinon cas d'egalite ou insertion dans le reste
    if ((*p_list)-> value == el) {
      // depend du cas d'utilisation
    }
    else // insertion dans la suite - appel rec
      insertInSortedList(el,&((*p_list)->next));
  }
}
```

Attention à ne pas imbriquer trop de cas. Attention aussi à ne pas faire de `else if` pour le dernier cas, sinon l'erreur de clang vous incite à modifier votre code et vous ne savez pas trop quoi en faire.

1.2 Désallocation

La désallocation pose souvent problème. Si on la pense en récursif c'est plus simple. La solution du cours est :

```
void detruireliste(List *l){
  if (l==NULL) return;
  if (*l!=NULL){
    freeFirstCell(l);
    detruireliste(l);
  }
}
```

Alternative : on peut aussi ne pas passer un pointeur vers la liste (Cell** / List*) mais simplement un pointeur sur Cellule. Il ne faudra pas oublier ensuite de désallouer proprement le dernier pointeur ensuite.

```
void detruireliste2_aux (Cell* pc){
  if (pc !=NULL){
    detruireliste2_aux(pc-> next);
    free(pc);
  }
}

void detruireliste2(List* l){
  if (l!=NULL){
    detruireliste2_aux(*l);
  }
  *l=NULL;
}
```

Pensez à Valgrind pour valider vos désallocations (et pensez à désallouer le `FILE*` avec un appel à `fclose`).

1.3 Liste chaînées de chaînes de caractères (tp7)

Ce n'est pas plus dur qu'auparavant, si on se permet des chaînes de taille bornée :

```
/*Cell and List types*/
typedef struct cell {
  struct cell *next ;
  char value[42];
}  Cell;

typedef Cell *PtCell, *List;
```

Les fonctions de construction, de recherche, sont un peu modifiées, la principale chose à ne pas oublier est que l'on ne peut pas faire de copie de tableau, et il faut donc remplacer $var = value$ par `strcpy(var, value)`, ou encore mieux, `strncpy(var,value,42)` pour ne pas faire de débordement. De même, la comparaison de deux chaînes ne se fait pas avec $<$ mais avec `strcmp`.

Ce qu'il faut retenir sur les listes

- La déclaration, l'ajout en tête
- Algos de base : ajouts dans une liste triée, désallocation, selon le paradigme que vous voulez (itératif, récursif) ...
- Les avantages et les inconvénients par rapports aux tableaux et aux listes contiguës (voir le cours).
- Savoir déboguer ! et surtout, savoir tester au fur et à mesure.

1.4 Contrôle TP

Voici les erreurs/soucis couramment relevées, en vrac :

- Oubli de son nom (!)
- Mauvais cas de gestion de liste vide pour `update`.
- Oubli de la gestion de la virgule dans `load`.
- Oubli du `fclose` (à écrire tout de suite après `fopen`).
- Boucles `while` qui ne terminent pas car la condition porte sur quelque chose qui n'est pas modifié dans le corps de la boucle.
- Update itératif mal développé et inutilement compliqué. Prendre la version récursive est une bonne idée.
- Absence de tests.
- Absence de tests.
- Absence de ... vous avez compris.

Je ne parle même pas de personnes qui n'ont jamais compilé avec le Makefile fourni, qui n'ont jamais exécuté (!), ou qui n'ont pas rempli le `main` avec les appels à leurs fonctions. Non, je n'en parle même pas.

1.5 Méthodologie de développement

Les erreurs du contrôle TP montrent un sérieux problème dans la méthodologie de développement.

- **Tests** : lorsqu'on ne demande pas explicitement de tester une fonction, ce test n'est pas fait. Quasiment personne ne teste `compare_points` (3 cas à faire). Quasiment personne ne teste `update` avant d'écrire / d'appeler `load`.

- **Cas d'arrêt** : les cas de bases des fonctions de listes (liste vide ou réduite à un seul élément) doivent être traités *en premier !* et testés dès leur écriture. Attention à bien distinguer le cas liste vide du reste, et à bien vérifier que la liste n'est pas vide avant d'accéder à un champ ! Les 3/4 des `Segmentation Fault` viennent de là.

- **Récupération d'erreurs** : l'ouverture correcte d'un fichier doit être vérifiée, et on doit faire en sorte de poursuivre le programme uniquement lorsque cela s'est bien passé (c'était *écrit* dans le main!). De même l'appel à `fclose(fp)` ne doit être fait que dans le cas où le fichier est bien ouvert. En passant, la valeur de retour de `malloc` doit aussi être récupérée.

- **Débug** : il faut absolument savoir se servir de GDB au moins pour connaître la liste d'un éventuel `Segmentation Fault`, et imprimer les pointeurs du contexte courant. Débuguer au `printf` est un choix, mais à ce moment là ne pas oublier `\n` qui permet de forcer l'impression au moment précis où vous invoquez `printf`.

- **Tactique** : faire les fonctions "faciles" au début peut être une bonne idée (points donnés sur le `print_list` qui n'ont pas été récupérés par tout le monde).

2 TP5 : GDB / Valgrind

2.1 GDB

Pas grand chose à dire sur ce TP, à part qu'il faut pratiquer GDB pour être efficace avec. Les environnements de développement proposent souvent un support de GDB intégré, pensez y! De plus, GDB est un outil formidable pour déboguer en cas de SEGFAULT, mais aussi en cas de bug algorithmique (on n'y pense pas assez !).

2.2 Valgrind

Super utile pour savoir si la désallocation a bien marché. Permet de voir le nombre de pointeurs qui ne sont pas désalloués et d'autres choses sur la mémoire à l'exécution.

Ce qu'il faut retenir sur GDB/Valgrind

- La compilation avec `-g` pour obtenir les infos de ligne.
- La survie : savoir détecter le numéro de ligne de son SEGFAULT.
- L'ajout de breakpoints, les impressions des infos, next, et step.
- Savoir lire une doc gdb pour retrouver la syntaxe.
- L'utilisation de base de valgrind et se réjouir si `All heap blocks were freed`.

3 TP 7/8 : Librairies

Pas de difficulté majeure dans ce couple de TP. Il faut bien comprendre la différence entre compilation (création des fichiers objets) et édition de lien (création d'un unique binaire comprenant toutes les fonctions *statiques* du code, et des liens vers les fonctions des librairies présentes dans le système (dites *dynamiques*)

Il faut connaître la compilation de plusieurs fichiers :

```
clang -c fichier1.c
clang -c fichier2.c
clang -o nombinaire fichier1.o fichier2.o
```

et aussi la compilation d'un/plusieurs fichiers avec une librairie existante.

```
clang -c fichier.c -Ichemin
clang -o nombinaire fichier.o -Lchemin2 -ltoto
```

(chemin donne le chemin du/des .h de la librairie, chemin2 l'adresse de la librairie `libtoto.a` ou `libtoto.so`)

Ce qu'il faut retenir sur la compilation séparée

- La compilation séparée permet de construire un logiciel contenant plusieurs .c et de les compiler pour réaliser un unique binaire
- On peut créer des librairies, et les utiliser, ou utiliser des librairies existantes.
- Le .h ne comporte que les déclarations de type et les signatures des fonctions. Le .c comporte l'inclusion du .h et l'implémentation des fonctions.
- Un makefile permet d'automatiser ce processus, il faut savoir lire/concevoir un Makefile

4 TP9 : Arbres

Rien à signaler non plus sur ce TP. La SD Arbre est très simple d'utilisation. Les algorithmes de base (insertion, recherche) s'implémentent facilement récursivement, lorsqu'on a compris le schéma récursif pour les arbres :

- traitement ou pas sur la racine
- traitement ou pas sur le sous-arbre gauche (récursion, appel de la même fonction avec ce sous-arbre)
- traitement ou pas sur le sous-arbre droit.

Ce qu'il faut retenir sur les Arbres
- Déclaration de la SD.
- Les principaux algorithmes : insertion recherche parcours, et leur complexité.

5 TP10 : void*

5.1 fonctions en paramètre

Ce TP a été l'occasion de voir comment passer des fonctions en paramètre à d'autres fonctions. Sans surprise, il faut passer un type "fonction" en paramètre.

```
void appliquer_tableau(int f(int), int t[],  int size){
  for (int i=0;i<size;i++)
    t[i] = f(t[i]);
}
```

Ici on passe en paramètre la fonction f de type $int \to int$, et on réalise l'application de f sur chacune des cases du tableau. Le tableau est donc modifié après l'appel.

5.2 qsort

Pour l'utilisation de `qsort`, dont je rappelle ici la signature :

```
void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));
```

il y a eu un peu plus de difficultés, dues à l'usage du type `void*` (pointeur "générique").

Un objet de type `void*` peut contenir n'importe quel pointeur, un pointeur d'entier (`int*`, de caractère (`char*`), de chaîne de caractère (`char**`), ... On se sert donc de ce type par exemple pour :

- passer en paramètre (adresse) un objet quelconque (ici le tableau à trier comme premier paramètre de qsort).
- créer des tableaux génériques, des listes génériques (mais ce n'est pas si simple, en fait).

On a déjà vu une fonction qui retourne un objet de type `void*`, c'est malloc :

```
void *malloc(size_t size);
```

Lors de nos utilisations de malloc, on utilisait un *cast* (interprétation du pointeur quelconque en un pointeur typé ici), par exemple en faisant :

```
Cell* mycell = (Cell*) malloc(sizeof(Cell));
               ^^^^^^^^
```

En faisant cela, le pointeur générique `void*` retourné par `malloc` devient un pointeur sur une Cellule. Il est donc spécialité.

Revenons à `qsort`. En lisant la doc, il est évident que l'on va l'appeler de la façon suivante :

```
qsort(t,4,sizeof(int),compareInt); // tri tableau d'entiers de taille4
qsort(tab2,5,sizeof(char*),compareString); // tri tableau de chaînes
```

Il ne reste plus qu'à écrire les deux fonction de comparaison, qui doivent toutes les deux avoir comme signature :

```
int compare (const void* pav, const void* pbv){
..
}
```

pour correspondre à la signature de `qsort`.

Commençons par la fonction de comparaison sur les entiers. Attention la fonction de comparaison ne prend pas deux entiers en paramètre, mais deux pointeurs constants `void*`, à interpréter comme des pointeurs constants vers des entiers. Il faut donc réaliser le cast avant la comparaison des deux entiers :

```
int compareInt (const void* pav, const void* pbv){
  int *pa = (int*) pav;
  int *pb = (int*) pbv;
  return (*pa-*pb); // ne pas oublier de déréférencer !
}
```

Une fois que l'on a compris pour les entiers, il n'y a plus qu'à faire de même pour les chaînes :

```
int compareString(const void* pav, const void* pbv)
{
        char **char_pav, **char_pbv; // pointeurs sur des chaines de char
        char_pav = (char**) pav;
        char_pbv = (char**) pbv;

       return strcmp(*char_pav,*char_pbv);
}
```

Au passage, voici comment on peut rapidement déclarer et initialiser un tableau de chaînes :

```
  char* tab2[] = {"toto","aa","a","zebulon","tata"};
```

Ce qu'il faut retenir sur la généricité en C

- On peut passer des fonctions en paramètre (et la syntaxe est simple)
- Les pointeurs génériques peuvent servir à réaliser des fonctions génériques (qui opèrent sur n'importe quel type). Dans ce cas, on fait attention à bien caster correctement les `void*` en `foo*` dès que l'on connaît `foo`. Attention à bien caster les pointeurs et non les types sous-jacents.