

Corrigé TP5: Structures

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

Exercice 1

Q1. En théorie, un élément de type `struct cours` prend $50 + 50 + 4 + 4 = 108$ octets. A priori, il n'y aura pas d'octets inutilisés à cause de l'alignement mémoire : la taille réelle renvoyée par `sizeof` devrait être égale à la taille théorique, 108. Si on exécute le programme de l'archive, on voit que c'est le cas.

Q2. On peut mélanger les deux manières proposées dans le code de l'archive, et créer le cours d'informatique avec quelques attributs initialisés, et d'autres non :

```
1  cours_t info = {
2      .intitule = "Informatique",
3      .salle = 0xC205
4  };
5  info.jour = 4;
```

Q3. Afin d'afficher le nom du jour, et pas l'entier 0-6 correspondant, on peut créer un tableau global contenant les noms des jours :

```
1  char* jours[7] = {"Lundi", "Mardi", "Mercredi",
2  "Jeudi", "Vendredi", "Samedi", "Dimanche"};
```

On peut alors facilement récupérer le nom du jour où un cours a lieu. Il ne faut pas oublier qu'avec des pointeurs de structures, on doit utiliser la notation `->` :

```
1  /* Affiche les informations de c */
2  void afficher_infos(cours_t* c){
3      printf("Cours: %s\n", c->intitule);
4      printf("Enseignant: %s\n", c->prof);
5      printf("Jour: %s\n", jours[c->jour]);
6      printf("Salle: %d\n\n", c->salle);
7  }
```

(Notons que si l'on veut afficher en hexadécimal, on peut utiliser le format `"%x"` dans `printf`.)

Q4. Dans la définition de la structure, on rajoute une ligne pour le nouvel attribut :

```
1  bool est_TP;
```

On doit alors modifier la fonction `afficher_infos`, par exemple en modifiant l'affichage de l'intitulé :

```
1 /* Affiche les informations de c */
2 void afficher_infos(cours_t* c){
3     if (c->est_TP){
4         printf("TP: %s\n", c->intitule);
5     } else {
6         printf("Cours: %s\n", c->intitule);
7     }
8     ...
```

Q5. On commence par allouer de la place dans le tas pour un `cours_t`, et on remplit les attributs un par un. On doit utiliser `strcpy` pour les chaînes de caractères, car on ne peut pas écrire, en C : `un_tableau = un_pointeur`.

```
1 /* Crée un cours alloué dans le tas, d'intitulé `intitule`,
2    enseigné par `prof`, dans la salle `salle`, le jour `jour` */
3 cours_t* creer_cours(char* intitule, char* prof, int salle, int jour){
4     cours_t* c = malloc(sizeof(cours_t));
5     c->salle = salle;
6     c->jour = jour;
7     strcpy(c->intitule, intitule);
8     strcpy(c->prof, prof);
9     return c;
10 }
```

Dans le main, pour tester :

```
1     cours_t* angl = creer_cours("Anglais", "P.P.", 0xDE02, 0);
2     afficher_infos(angl);
3     free(angl);
```

Exercice 2

Q1. OK

Q2. Dans la fonction test :

```
1     menu_elem_t oeuf = {
2         .nom = "Oeuf de girafe",
3         .prix_unite = 8.50,
4         .vegan = false
5     };
6     printf("Prix de l'oeuf: %f\n", oeuf.prix_unite);
```

Q3. Fonction demandée :

```
1 /* Affiche les informations de e */
2 void affiche_elem(menu_elem_t* e){
3     printf("%s", e->nom);
4     if (e->vegan){
5         printf(" (V)");
6     }
7     printf(": %.2f\n", e->prix_unite);
8 }
```

Q4. En utilisant un flottant, on risque d'avoir des erreurs d'arrondis. Si quelqu'un

commande 100 Hamburgers du chef, le prix total théorique est de 799 euros, mais les erreurs flottantes risquent de donner un résultat comme 799.01 ou 798.99. Pour régler ce problème, on pourrait stocker le prix en centimes, sous la forme d'un `int`, il faudrait alors un peu modifier la fonction d'affichage pour séparer les euros et les centimes mais on n'aurait plus d'erreurs possibles (tant que la commande ne dépasse pas $2^{31} - 1$ centimes...).

Q5. OK

Q6. Pour la fonction demandée :

```
1 /* Affiche la liste des plats de m */
2 void affiche_menu(menu_t* m){
3     printf("Menu:\n");
4     for (int i = 0; i < m->nb_elems; ++i){
5         printf("%d. ", i+1);
6         affiche_elem(&m->elements[i]);
7     }
8 }
```

Q7. Il suffit d'une boucle qui met tous les éléments du tableau à 0 :

```
1 /* Initialise les nb_elems premières cases de commande à 0 */
2 void init_commande(int* commande, int nb_elems){
3     for (int i = 0; i < nb_elems; ++i){
4         commande[i] = 0;
5     }
6 }
```

Q8. La fonction doit lire des entiers dans le terminal, jusqu'à échouer. On peut donc tirer avantage du fait que `scanf` renvoie le nombre d'éléments lus :

```
1 /* Lit dans le terminal les plats commandés. Pour chaque 0 <= i < nb_elems,
2    écrit dans commande[i] le nombre d'exemplaires du i-ème plat. */
3 void prendre_commande(int* commande, int nb_elems){
4     int plat_demande = 0;
5     while(scanf("%d", &plat_demande) == 1){
6         assert(1 <= plat_demande && plat_demande <= nb_elems);
7         commande[plat_demande-1]++;
8     }
9 }
```

Notons que cette fonction marcherait même si l'utilisateur finit sa commande par un symbole autre que `#`, comme `$` ou autre. Si on voulait absolument que la commande finisse par `#`, il faut lire un dernier caractère après être sorti de la boucle :

```
1     char c;
2     fscanf(f, "%c", &c);
3     assert(c == '#');
```

Q9. Rien de compliqué, il suffit d'aller chercher les informations dans les deux structures :

```

1  /* Affiche un résumé de la commande `commande` concernant les
2     plats du menu m */
3  void resume_commande(int* commande, menu_t* m){
4     printf("Vous avez commandé:\n");
5     for (int i = 0; i < m->nb_elems; ++i){
6         if (commande[i] > 0){
7             printf("%dx %s\n", commande[i], m->elements[i].nom);
8         }
9     }
10 }

11
12
13 /* Renvoie le prix total des plats commandés dans `commande` pour le
14     menu m */
15 float total_commande(int* commande, menu_t* m){
16     float res = 0;
17     for (int i = 0; i < m->nb_elems; ++i){
18         res += commande[i] * m->elements[i].prix_unite;
19     }
20     return res;
21 }

```

Q10. (Remarque : cette fonction est inutile dans la suite...) Puisqu'on connaît la taille réelle d'un menu, on sait dans quelle case écrire pour ajouter un élément. Dans la solution proposé, on commence par créer un pointeur vers la case à modifier, afin de ne pas devoir écrire `m->elements[m->nb_elems]` à chaque fois :

```

1  /* Ajoute à m l'élément décrit par nom, prix_unite et vegan */
2  void ajouter_element(menu_t* m, char* nom, float prix_unite, bool vegan){
3     menu_elem_t* e = &m->elements[m->nb_elems];
4     e->prix_unite = prix_unite;
5     e->vegan = vegan;
6     strcpy(e->nom, nom);
7     m->nb_elems++;
8 }

```

- Q11.** Erreur d'énoncé : le type de retour devrait être **bool**, et la fonction doit renvoyer un booléen indiquant si la fin du fichier a été atteinte ou non (sinon, la question suivante est dure !). Le principe est de lire le prix et le booléen vegan normalement via `fscanf`, puis de récupérer tout le reste de la ligne. Pour cela, on peut utiliser la fonction `getline` présentée au TP précédent, ou bien simplement lire caractère par caractère jusqu'à lire un retour ligne (ou tomber sur EOF) :

```

1  /* Lit dans f les informations d'un élément de menu, et les stocke dans e.
2     Renvoie true si la lecture a échoué en atteignant la fin du fichier. */
3  bool lire_element(FILE* f, menu_elem_t* e){
4      int nb_lus = fscanf(f, "%f", &e->prix_unite);
5      if (nb_lus == EOF){
6          return true;
7      }
8      int x;
9      fscanf(f, "%d", &x);
10     e->vegan = (x == 1);
11
12     // jeter l'espace suivant le booléen vegan
13     char poubelle;
14     fscanf(f, "%c", &poubelle);
15     // récupérer le reste de la ligne
16     int i = 0; // nb de caractères lus
17     while(fscanf(f, "%c", &e->nom[i]) != EOF && e->nom[i] != '\n'){
18         i++;
19     }
20     // sortie: il faut mettre un '\0' à la fin du nom.
21     // si on a atteint EOF, il va dans la case i, sinon
22     // il va dans la case i-1 pour remplacer le '\n' lu à la fin
23     if (e->nom[i-1] == '\n'){
24         e->nom[i-1] = '\0';
25     } else {
26         e->nom[i] = '\0';
27     }
28     return false;
29 }

```

- Q12.** L'idée est de lire dans le fichier, élément par élément, en s'arrêtant quand la fin du fichier est atteinte. Avec la bonne version de la fonction `lire_element`, le code est assez simple :

```

1  void lire_menu(char* nom_fichier, menu_t* m){
2      FILE* f = fopen(nom_fichier, "r");
3      assert(f != NULL);
4      int i = 0; // nb d'éléments lus
5      while(!lire_element(f, &m->elements[i])){
6          i++;
7      }
8      m->nb_elems = i;
9      fclose(f);
10 }

```

Q13. Dans le main, on enchaîne toutes les fonctions que l'on a codé précédemment :

```
1  int main(int argc, char** argv){
2      //test();
3      if (argc < 2){
4          printf("Veuillez préciser le fichier du menu\n");
5          exit(1);
6      }
7      char* fn = argv[1];
8      menu_t m;
9      lire_menu(fn, &m);
10     affiche_menu(&m);
11     int commande[MAX_ELEMS];
12     init_command(commande, m.nb_elems);
13     prendre_commande(commande, m.nb_elems);
14     resume_commande(commande, &m);
15     printf("Total: %.2f\n", total_commande(commande, &m));
16     return 0;
17 }
```

Exercice 3

Q1. Mettons la création de l'équipe dans une fonction à part, ce qui permettra de bien compter les free/mallocs plus tard. Dans la solution proposée, on choisit de **tout** stocker dans le tas, y compris les prénoms des membres de l'équipe :

```
1  /* Crée une équipe de 3 membres */
2  equipe_t* mon_equipe1(){
3      equipe_t* e1 = malloc(sizeof(equipe_t));
4      e1->nb_membres = 3;
5      e1->membres = malloc(3*sizeof(etu_t));
6      for (int i = 0; i < 3; ++i){
7          e1->membres[i] = malloc(sizeof(etu_t));
8      }
9
10     e1->membres[0]->age = 23;
11     e1->membres[0]->prenom = malloc(10*sizeof(char));
12     strcpy(e1->membres[0]->prenom, "Camille");
13
14     e1->membres[1]->age = 20;
15     e1->membres[1]->prenom = malloc(10*sizeof(char));
16     strcpy(e1->membres[1]->prenom, "Leila");
17     e1->indice_cpt = 1;
18
19     e1->membres[2]->age = 22;
20     e1->membres[2]->prenom = malloc(10*sizeof(char));
21     strcpy(e1->membres[2]->prenom, "Thibault");
22
23     return e1;
24 }
```

Q2. Fonctions d'affichage :

```

1  /* Affiche les informations de l'étudiant e */
2  void print_etu(etu_t* e){
3      printf("%s (%d ans)\n", e-> prenom, e->age);
4  }
5
6  /* Affiche les membres de l'équipe e */
7  void print_equipe(equipe_t* e){
8      for (int i = 0; i < e->nb_membres; ++i){
9          if (i == e->indice_cpt){
10             printf("Capitaine: ");
11         } else {
12             printf("Membre   : ");
13         }
14         print_etu(e->membres[i]);
15     }
16 }

```

Q3. Pour la libération de mémoire, on voit qu'un étudiant demande deux malloco : un pour la structure même, et un pour le prénom :

```

1  /* Libère la mémoire allouée pour e */
2  void free_etu(etu_t* e){
3      free(e->prenom);
4      free(e);
5  }

```

Pour les équipes, il faut libérer chaque étudiant, puis le tableau des étudiants, puis la structure elle-même :

```

1  /* Libère la mémoire allouée pour e */
2  void free_equipe(equipe_t* e){
3      for (int i = 0; i < e->nb_membres; ++i){
4          free_etu(e->membres[i]);
5      }
6      free(e->membres);
7      free(e);
8  }

```

Q4. Supposons que la fonction `agrandir_equipe` copie les pointeurs des étudiants uniquement, sans recopier les données. Alors, tous les étudiants sont free par l'appel à `free_equipe(e1)`, et l'appel à `free_equipe(e2)` cause une erreur de double free. Il faudrait donc, dans ce cas, que la fonction `agrandir_equipe` s'occupe de vider l'équipe source, par exemple en mettant `nb_membres` à 0.**Q5.** Il y a plusieurs solutions possibles, l'important est de bien spécifier le comportement, et d'utiliser la fonction de manière adéquate. Deux exemples de solutions :

- On fait une copie intégrale de chaque étudiant de `src`, que l'on rajoute à `dst`. Cela permet aux deux équipes `src` et `dst` de continuer à exister en même temps, et il faudra les libérer toutes les deux séparément
- On détruit l'équipe `src`, et l'on ajoute les pointeurs de ses membres au tableau des membres de `dst`. Alors, après agrandissement, seule l'équipe `dst` existe. L'équipe `src` est soit vide, soit inutilisable (et libérée).

Notons que dans tous les cas, il faudra agrandir le tableau des membres de `dst`, donc créer un nouveau tableau des membres. Voici une implémentation qui détruit l'équipe source :

```
1  /* Fusionne les équipes dst et src. Le pointeur src devient
2     inutilisable après appel, la nouvelle équipe est stockée
3     dans dst. */
4  void agrandir_equipe(equipe_t* dst, equipe_t* src){
5      int n1 = dst->nb_membres;
6      int n2 = src->nb_membres;
7
8      int n = n1 + n2;
9      // ajouts des membres des deux équipes:
10     // les membres de l'équipe dst seront entre 0 et n1 - 1
11     // les membres de l'équipe src seront entre n1 et n1 + n2 - 1
12     etu_t** nouveaux_membres = malloc(n * sizeof(etu_t*));
13     for (int i = 0; i < n1; ++i){
14         nouveaux_membres[i] = dst->membres[i];
15     }
16     for (int i = 0; i < n2; ++i){
17         nouveaux_membres[n1+i] = src->membres[i];
18     }
19     // choix du/de la capitaine
20     int nouv_cpt;
21     if (n1 > n2){
22         nouv_cpt = dst->indice_cpt;
23     } else {
24         nouv_cpt = n1 + src->indice_cpt;
25     }
26
27     // libérer l'équipe src. Il ne faut pas utiliser la fonction free_equipe
28     // sinon on libère la moitié des étudiants de la nouvelle équipe
29     free(src->membres);
30     free(src);
31
32     // plus besoin du tableau des membres pour dst, on le remplace par
33     // le nouveau tableau
34     free(dst->membres);
35     dst->membres = nouveaux_membres;
36     dst->nb_membres = n;
37     dst->indice_cpt = nouv_cpt;
38
39 }
```

Vu la spécification, lorsque l'on utilise cette fonction, on libère l'équipe source, qui n'a donc plus besoin d'être libérée :

```
1  e1 = mon_equipe1();
2  equipe_t* e2 = mon_equipe2();
3  agrandir_equipe(e2, e1);
4  printf("Fusion des deux équipes:\n");
5  print_equipe(e2);
6  free_equipe(e2);
7  return 0;
```

En exécutant avec valgrind, on constate qu'il n'y a aucune fuite mémoire.

Exercice 4

Q1. Dans le main :

```
1 int main(){
2     srand(time(NULL));
3     printf("Liste aléatoire:\n");
4     liste_t* L = liste_aleatoire(10);
```

Q2. En faisant le dessin, on voit que la variable `courant` (qui est dans la pile), pointe vers le tas. Initialement, elle pointe vers le premier maillon de la liste. et au fil de l'exécution, elle pointe systématiquement vers le dernier maillon créé.

Q3. Il suffit de reprendre la fonction `liste_aleatoire` et de la modifier pour qu'elle écrive 0, 1, ... sur les maillons créés :

```
1 /* Renvoie une liste contenant 0, 1, ..., n-1 */
2 liste_t* liste_range(int n){
3     assert(n >= 0);
4     liste_t* res = malloc(sizeof(liste_t));
5     res->taille = n;
6     if (n == 0){
7         res->tete = NULL;
8         return res;
9     }
10
11     res->tete = malloc(sizeof(maillon_t));
12
13     maillon_t* courant = res->tete;
14     courant->val = 0;
15     for (int i = 1; i < n; ++i){
16         courant->suivant = malloc(sizeof(maillon_t));
17         courant = courant->suivant;
18         courant->val = i;
19     }
20     courant->suivant = NULL;
21     return res;
22 }
```

Q4. Il y a plusieurs manières de répondre à cette question. On peut faire une boucle qui parcourt les maillons (comme `liste_print`) en les libérant au fur et à mesure. Il faut faire attention au fait que l'on ne peut pas faire :

```
1     maillon_t* m = l->tete;
2     while(m != NULL){
3         free(m);
4         m = m->suivant;
5     }
```

car après avoir free un maillon, on ne peut plus accéder à ses attributs. On va donc utiliser une variable permettant de stocker temporairement l'adresse du maillon à libérer, et le libérer **après** être passé au suivant :

```

1  /* Libère la liste l */
2  liste_t* liste_free(liste_t* l){
3      maillon_t* m = l->tete;
4      while (m != NULL){
5          // libérer m et passer au maillon suivant
6          maillon_t* p = m;
7          m = m->suivant;
8          free(p);
9      }
10     free(l);
11 }

```

Q5. Cette fonction reprend le même principe que `liste_print` et `liste_free` : on parcourt les maillons un par un en traitant chacun successivement. Lorsque l'on manipule des listes chaînées, il est courant d'utiliser des boucles for de la forme suivante :

```

1  for (maillon_t* m = l->tete; m != NULL; m = m->suiv){
2      ...
3  }

```

Ce qui est une manière condensée d'écrire :

```

1  maillon_t* m = l->tete;
2  while(m != NULL){
3      ...
4      m = m->suiv;
5  }

```

Par exemple pour la fonction `liste_recherche` :

```

1  /* Renvoie true si l contient x, false sinon */
2  bool liste_recherche(int x, liste_t* l){
3      for(maillon_t* m = l->tete; m != NULL; m = m->suivant){
4          if (m->val == x){
5              return true;
6          }
7      }
8      return false;
9  }

```

Q6. Pour ajouter un nouveau maillon au début de la liste on peut décomposer le problème en trois étapes :

- Récupérer la tête de liste actuelle A (qui peut être NULL);
- Créer un nouveau maillon M , dont le maillon suivant est A ;
- Marquer que M est la nouvelle tête de liste :

```

1  /* Ajoute x au début de l */
2  void ajouter_debut(liste_t* l, int x){
3      maillon_t* ancienne_tete = l->tete;
4
5      maillon_t* nouvelle_tete = malloc(sizeof(maillon_t));
6      nouvelle_tete->val = x;
7      nouvelle_tete->suivant = ancienne_tete;
8
9      l->tete = nouvelle_tete;
10 }

```

Pour ajouter un nouveau maillon en **fin** de liste, il faut commencer par trouver le dernier maillon actuel de la liste. Pour cela, on parcourt la liste, en s'arrêtant juste avant de trouver NULL :

```

1  maillon_t* m = l->tete;
2  while(m->suiv != NULL){
3      m = m->suiv;
4  }
5  // en sortie, m->suiv == NULL: c'est la queue de liste

```

Notons que ce code essaie d'accéder à l'attribut `m->suiv` en entrant dans la boucle. Il faut donc que `m` ne soit pas NULL! Autrement dit, le code ci-dessus ne fonctionnera pas pour une liste vide : ce cas doit être traité à part. D'où la fonction :

```

1  /* Ajoute x à la fin de l */
2  void ajouter_fin(liste_t* l, int x){
3      // liste vide
4      if (l->tete == NULL){
5          l->tete = malloc(sizeof(maillon_t));
6          l->tete->val = x;
7          l->tete->suivant = NULL;
8          return;
9      }
10     // liste non vide: la tete existe
11     maillon_t* m = l->tete;
12     while(m->suivant != NULL){
13         m = m->suivant;
14     }
15     // en sortie: m est la queue actuelle
16     m->suivant = malloc(sizeof(maillon_t));
17     m->suivant->val = x;
18     m->suivant->suivant = NULL;

```

- Q7.** Deux phases : d'abord trouver le maillon à supprimer, puis le supprimer en raccordant les liaisons. En faisant un dessin, on se rend compte que pour supprimer un maillon M , on a besoin de connaître le maillon précédent P afin de pouvoir relier P au suivant de M , notons S . Si M n'a pas de maillon précédent, i.e. si c'est la tête, on a juste à marquer que S est la nouvelle tête. D'où la fonction :

```
1  /* Supprime la première occurrence de x dans l */
2  void supprimer_valeur(liste_t* l, int x){
3      // Trouver le premier maillon contenant x
4      maillon_t* m = l->tete;
5      maillon_t* p = NULL;
6      // Invariant: p est le maillon précédant m
7      while (m != NULL && m->val != x){
8          p = m;
9          m = m->suivant;
10     }
11     if (m == NULL){
12         // x pas trouvé
13         return;
14     }
15
16     if (p == NULL){
17         // m est la tête
18         l->tete = m->suivant;
19         free(m);
20     } else {
21         // p existe
22         p->suivant = m->suivant;
23         free(m);
24     }
25 }
```

Exercice 5

Q1.

Q2.

Q3.

Exercice 6

Q1.

Q2.

Q3.

Q4.

Q5.