Architecture des Ordinateurs

2ème Année Licence Informatique

Des bases en C

Travaux Dirigés #2

1. Contrôle préalable

Cette partie est conçue comme une vérification pour vous permettre de déterminer si vous comprenez les concepts abordés en cours ou non. Veuillez répondre aux questions suivantes et inclure une explication :

1.1. Vrai ou Faux : En langage C, le passage de paramètre se fait par valeur.
Vrai. Une copie des arguments réels est transmise aux arguments formels respectifs. Si le passage par référence est désiré, alors il faut passer l'emplacement (adresse) des arguments réels (c.-à-d. utiliser des pointeurs).

1.2. C'est quoi un pointeur en C ? Qu'a-t-il en commun avec un tableau ?
Un pointeur n'est qu'une séquence de bits interprétés comme une adresse mémoire. Un tableau agit comme un pointeur vers le premier élément de la mémoire allouée pour ce tableau.

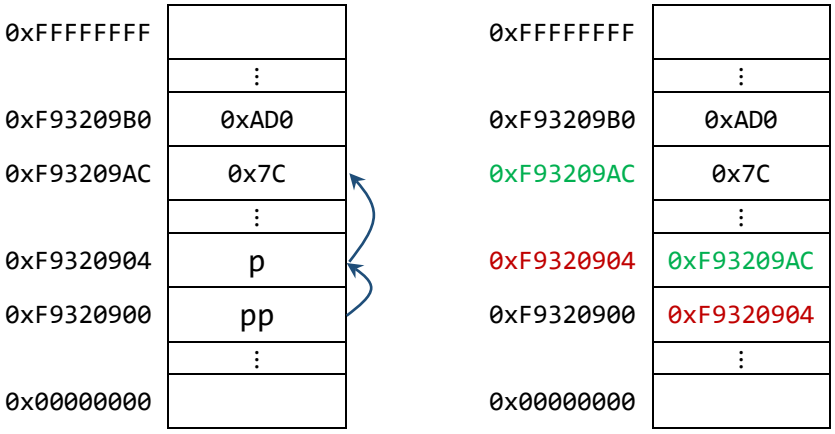
1.3. Si vous essayez de déréférencer une variable qui n'est pas un pointeur (c.-à-d. lui accoler un astérisque), que se passe-t-il ? Quid lorsque vous en libérez (*i.e.* `free(…)`) une ?
Le C traitera les bits de cette variable comme s'il s'agissait d'un pointeur et tentera d'accéder aux données « pointées ». Votre programme se terminera très probablement avec l'erreur « `memory segfault` ». Si vous libérez une variable qui a été libérée auparavant ou qui n'a pas été allouée, votre programme aura un comportement non défini et s'achèvera avec l'erreur « `invalid free` ».

2. Mémoire en C

Le langage C est syntactiquement très similaire à Java, mais il y'a quelques différences clés :

- Le C est « orienté fonction » et non « orienté objet ». Il n'y a donc pas d'objets.
- Il n'y a pas de « *garbage collector* » ou gestionnaire automatique de mémoire dans le langage C. Les allocations et libérations de mémoire dynamique sont explicitement gérées par le programmeur (`malloc()`, … , `free()`).
- Les pointeurs sont utilisés d'une manière explicite dans le langage C. Si « `p` » est un pointeur, alors « `*p` » indique (c.-à-d. *pointe vers*) la donnée à utiliser et non la valeur de « `p` » (*i.e.* l'adresse mémoire). Si « `x` » est une variable, alors « `&x` » renvoie l'adresse (*i.e.* un pointeur) de « `x` » et non la valeur de « `x` ».

Dans l'exemple suivant, à gauche, la mémoire est représentée par un diagramme boîte-et-pointeur. À droite, on voit comment la mémoire est réellement représentée dans l'ordinateur (les adresses ont été choisies arbitrairement).



Supposons un pointeur sur un entier (*i.e.* `int* p;`) alloué à l'adresse `0xF9320904`. Supposons également une variable de type entier « `int x` » allouée à l'adresse `0xF93209B0`. Comme on peut l'observer sur les diagrammes ci-dessus :

- `*p` doit retourner la valeur `0x7C`.
- `p` donnera la valeur `0xF93209AC` (l'adresse où la valeur `0x7C` est stockée).
- `x` retournera la valeur `0xAD0`.
- `&x` retournera la valeur `0xF93209B0` (l'adresse où la valeur `0xAD0` est stockée).

Supposons maintenant un pointeur sur un pointeur sur un entier (*i.e.* `int** p;`) alloué à l'adresse `0xF9320900` (voir le diagramme de gauche ci-dessus).

2.1. Quelle sera la valeur retournée par `pp` ? quid de `*pp` ? et de `**pp` ?
`pp` retournera la valeur `0xF9320904`. `*pp` retournera `0xF93209AC`. et `**pp` donnera `0x7C`.

2.2. Quelque chose ne va pas avec le code C ci-dessous ! Pouvez-vous repérer le problème ?

```
1  int* get_money(int cash) {
2    int* money = malloc(2017 * sizeof(int));
3    if(!cash)
4        money = malloc(1 * sizeof(int)); // fuite de mémoire si !cache
5    return money;
6  }
```

Soit la liste chaînée `ll_noeud` définie ci-dessous. Supposons que le pointeur `*lst` indique le premier élément de la liste chaînée (tête de liste) ou contient la valeur `NULL` si la liste est vide.

```
struct ll_noeud {
    int valeur;
    struct ll_noeud* suivant;
}
```

2.3. Ecrivez le code pour insérer un élément au début de la liste chaînée.

```
void inserer(struct ll_noeud** lst, int val ) {
   struct ll_noeud* elem = (struct ll_noeud*) malloc(sizeof(struct ll_noeud));
   elem->valeur = val;
   elem->suivant = *lst;
   *lst = elem;
}
```

2.4. Implémentez la fonction `liberer_ll` pour libérer/vider toute la liste

```
void liberer_ll(struct ll_noeud * lst) {
// solution recursive
   if(lst) {
      liberer_ll(lst->suivant);
      free(lst);
   }

// ou solution séquentielle
   while (lst) {
      struct ll_noeud *temp = lst->suivant;
      free(lst);
      lst = temp;
   }
}
```

3. Programmer avec les pointeurs

Implémentez les fonctions suivantes afin qu'elles fonctionnent comme décrit.

3.1. Une fonction qui permet de permuter les valeurs de deux entiers donnés en paramètres.

```
void swap(int* x, int* y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

3.2. Une fonction qui retourne le nombre d'octets dans une chaîne de caractères (similaire à la fonction standard de la bibliothèque C `strlen()`).

```
int mastrlen(char* str) {
  int compte = 0;
  while(*str++) {
    compte++;
  }
  return compte;
}
```

Examinez les fonctions suivantes et corriger *éventuellement* les problèmes

3.3. Retournez le total de tous les éléments dans le tableau `summands`

```
1  int sum(int* summands) {  // int sum(int* summands, unsigned int n) {
2   int sum = 0;
3   for(int i = 0; i <sizeof(summands); i++)  // for(int i = 0; i < n; i++)
4       sum += *(summands + i);
5   return sum;
6  }
```

3.4. Incémentez les caractères de la chaîne `string` stockée au début d'un tableau de longueur `n >= strlen(string)`. NE DOIT PAS modifier les zones de mémoire en dehors de la chaîne de caractères.

```
1  void increment(char* string, int n) {  // le paramètre n est inutile
2   for(int i = 0; i < n; i++)  // for(int i = 0; string[i] != 0; i++)
3       *(string + i)++;   // string[i]++;
4                          // ou (*(string + i))++;
5  }
```

3.5. Copie de la chaine de caractère `src` dans `dst`.

```
1  void copy(char* src, char* dst) {
2     while(*dst++ = *src++);
3     // Il n'y a pas d'erreur avec ce code
4  }
```

3.6. Remplacez, s'il y'a assez d'espace dans une chaîne de caractères donnée en paramètre, par la chaîne `"Le cours ADO est fantastique !"`. La fonction ne doit rien faire si la condition n'est pas vérifiée. Vous pouvez supposer que le paramètre `length` donne la longueur correcte de la chaine de caractères `src`.

```
1  void ado(char* src, unsigned int length) {
2   char *srcptr, replacteptr;    // char *srcptr, *replaceptr;
3   char remplacement[31] = "Le cours ADO est fantastique !";
4   srcptr = src ;
5   replaceptr = replacement;
6   if(length >= 31) {
7     for(int i=0; i<31; i++)
8        *srcptr++ = *replaceptr++;
9   }
10 }
```

4. Données en mémoire

Soit le type de structure de données définie ci-dessous.

```
typedef struct _donnees {
    char nom[13];          /* Nom Prénom  */
    unsigned short age;    /* En années    */
    char sexe;             /* M : Masculin, F : Féminin   */
    int matricule[4];      /* exemple : 1994 408 010 7212 */
} donnees;
```

Supposons qu'une structure « `employe` » de type « `donnees` » soit allouée à l'adresse mémoire « `0x8040` » avec les initialisations suivantes :

```
donnees employe = {
    .nom = "Tintin Lupin",
    .age = 23,
    .sexe = 'M',
    .matricule = {1994,408,10,7212}
};
```

4.1. Si on considère que « `sizeof(char) == 1`, `sizeof(short) == 2`, et `sizeof(int) == 4` », et si l'on considère aussi une organisation de la mémoire en mode « little-endian », donnez la représentation en hexadécimal des octets de la structure « `employe` » dans la mémoire.

Adresses	Données (octets)							
0x8040	54	69	6E	74	69	6E	20	4C
0x8048	75	6E	69	6E	00	-	17	00
0x8050	4D	-	-	-	CA	07	00	00
0x8058	98	01	00	00	0A	00	00	00
0x8060	2C	1C	00	00				

4.2. Utilisez maintenant le mode « big-endian »

Adresses	Données (octets)							
0x8040	54	69	6E	74	69	6E	20	4C
0x8048	75	6E	69	6E	00	-	00	17
0x8050	4D	-	-	-	00	00	07	CA
0x8058	00	00	01	98	00	00	00	0A
0x8060	00	00	1C	2C				

Rappelons que, pour des raisons de performance que nous verrons plus tard dans le cours, les données en mémoire sont (toujours) stockées à des adresses qui sont multiples de leurs tailles. D'autre part, une chaine de caractères en C est toujours achevée par un zéro implicite – N'oubliez pas de le compter quand vous allouez de l'espace mémoire !