Programmation 1

TD n°4

Aliaume Lopez

8 octobre 2019

1 Retour sur le TD précédent

Exercise 1 : Introduction aux pointeurs

1. Quel est le type de `p`, de `f`, de `__sa_handler`, de `__sa_sigaction`, de `signal` dans les exemples ci-dessous ? (Non, ne cherchez pas à enlever des parenthèses ou quoi que ce soit d'autre, juste le nom de la variable. Et non, vous ne rêvez pas.)
   (a) `int *p;`
   (b) `int (*f) (char *s);`
   (c) `void (*__sa_handler)(int signo);`
   (d)
   ```
   void (*__sa_sigaction)(int signo,
                          struct __siginfo *si, void *data);
   ```
   (e) `void (*signal(int sig, void (*func)(int)))(int);`
2. Expliquer, en français, ce que sont donc les objets déclarés dans la question précédente.

Arithmétique de pointeurs

En C, pour `a` un pointeur vers des objets de taille n, et pour `i` un entier, `a+i` est l'adresse du $(\texttt{i}+1)$-ième objet stocké au tableau démarrant à l'adresse `a`.

Exercise 2 : Arithmétique de pointeurs

Justifier les égalités suivantes :

1. `a[i]=*(a+i)`,
2. `a+i=&a[i]`,
3. `&*p=p`,
4. `*&x=x`,
5. `p[i]=i[p]`,
6. `(&p[i])[j]=p[i+j]`.

2 Pointeurs et variantes

Exercise 3 : Pointeurs, références, objets

1. Combien valent `!a`, `!b`, `!c` après exécution du code Caml suivant ?
   ```
   let a = ref 2;;
   let b = ref (!a);;
   let c = a;;
   a:= 9;;
   ```
2. Même question avec `a^`, `b^`, `c^` et le code Pascal suivant :

```
var a, b, c : integer^;
new a;  a^ := 2;
new b;  b^ := a^;
c := a;
a^ := 9;
```

3. Pareil avec `*a`, `*b`, `*c` et le code C suivant :

```
int *a, *b, *c;
a = malloc (sizeof (int));  *a = 2;
b = malloc (sizeof (int));  *b = *a;
c = a;
*a = 9;
```

4. Pareil avec `a.value`, `b.value`, `c.value` et le code Java suivant :

```
public class MutableInteger{
  private int value;
  MutableInteger (int v) { value = v; }
  MutableInteger (MutableInteger a) { value = a.value; }
  public int getValue() { return value; }
  public void setValue(int v) { value = v; }
}
...
  MutableInteger a, b, c;
  a = new MutableInteger(2);
  b = new MutableInteger(a);
  c = a;
  a.setValue(9);
```

(Question subsidiaire : à quoi sert cette manie des programmeurs Java de programmer des accesseurs `getValue()` et `setValue()` plutôt que d'accéder aux champs eux-mêmes ?)

5. Pareil avec `a.get()`, `b.get()`, `c.get()` en Python :

```
class ref:
    def __init__(self, obj): self.obj = obj
    def get(self):    return self.obj
    def copy(self):   return ref(self.obj)
    def set(self, obj):      self.obj = obj
a = ref 2
b = a.copy()
c = a
a.set(9)
```

3 Introduction aux tableaux

Tableaux statiques

On peut créer des tableaux à plusieurs dimensions en C. Voici comment créer un tableau 30×40 d'entiers :

```
int p[30][40];
```

Exercise 4 : Tableaux C

1. Dessiner schématiquement la liste des cases du tableau `p` en mémoire obtenu à partir de l'instruction `int p[30][40]`. On réalisera qu'il s'agit juste d'un tableau de 30 éléments, chaque élément étant un tableau de 40 éléments.

2. Quelle est la différence avec les déclarations suivantes ?

(a)
```
int *p[30];
int i;
for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
```

(b)
```
int **p;
int i;
p = malloc(sizeof(int *[30]));
if (p==NULL) abort();
for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
```

(c)
```
int p[30*40];
```

(d)
```
int *p;
p = malloc(sizeof(int [30*40]));
```

Exercise 5 : Normes normes

Pourquoi d'après vous la plupart des compilateurs C conformes à la norme ANSI C 89 (pas `gcc`, ni les compilateurs plus récents, conformes à la norme ANSI C 99) refusent-t-ils le code suivant ?

```
int f(int n){
  int p[n];

  [...] /* reste non pertinent */
}
```

Exercise 6 : Les tableaux en C/ASM de long en large

1. On considère le programme C suivant :

```
int a[10];

int main(){
  int i, j;

  a[0] = 1;  a[1] = 1;
  for (i=2; i<10; i++)
    a[i] = a[i-2]+a[i-1];
  for (j=9; j>=0; j--)
    printf ("%d\n", a[j]);
  return 0;
}
```

Le tableau `a[]` est global (toutes les fonctions y ont accès) et *statique* (il est alloué une fois pour toutes : « statique » désigne en général tout ce qui se fait à l'écriture du programme ou lors de sa compilation, par opposition à « dynamique », qui se réfère à ce qui a lieu lors de l'exécution).

(a) Qu'affiche ce programme ?

(b) On supposera qu'on tourne sur une architecture MIPS 32 bits, auquel cas le tableau `a[]` occupe 40 octets. Écrire les directives assembleur nécessaires à l'allocation (statique) de 40 octets de mémoire dans la section `.data` et lui donne l'étiquette `a`. On pourra soit utiliser la pseudo-instruction `.word` suivie de 10 zéros, soit la pseudo-instruction `.space` suivie du nombre d'octets à réserver. Il faudra en outre penser à utiliser `.align`.

(c) Quelle est la différence entre les instructions `.word` et `.space` ?

(d) Même exercice en assembleur x86 32 bits, en remplaçant `.word` par `dd`, `.space` par `.fill`.

(e) On rappelle que la donnée `a[i]` réside á l'adresse `a+4i` (toujours sur une machine 32 bits). Convertir le programme ci-dessus en code à trois valeurs et le traduire en assembleur MIPS.

(f) Pareil en assembleur x86.

2. On considère le même programme que ci-dessus, à part le changement suivant dans les premières lignes :

```
int main(){
  int a[10];
  int i, j;

  [...]
```

Dans ces conditions, le tableau `a[]` est alloué *dynamiquement*, chaque fois que l'on rentre dans `main`. Il est aussi désalloué à la sortie de `main`

En pratique, il est alloué en décrémentant le registre de pile (`sp` sur MIPS, `%esp` sur x86) de 40.

(a) Si l'on doit modifier le registre de pile, il vaut mieux le sauvegarder à l'entrée de `main` (dans `fp` sur MIPS, dans `%ebp` sur x86)... et sauvegarder aussi le registre de sauvegarde. Comment le fera-t-on en assembleur MIPS ?

(b) Modifier votre programme assembleur MIPS de la question précédente pour qu'il effectue cette sauvegarde, ainsi que la restauration correspondante en fin de procédure. Il devra aussi allouer 40 octets sur la pile, comme indiqué plus haut, et accéder au tableau `a[]` non plus via le label `a` (qui n'existe plus dans notre nouvelle version) mais aux endroits adéquats de la pile.

(c) Et en assembleur x86 ?

3. On pourrait vouloir optimiser le programme assembleur précédent de sorte à ce qu'il laisse le registre de pile inchangé, et en retrouvant le tableau `a[]` en utilisant des déplacements négatifs à partir de `sp` (resp., `%esp`).

(a) Pourquoi cette stratégie de compilation ne serait-elle pas acceptable si `main()` appelait des fonctions auxiliaires ?

(b) Pourquoi n'est-elle en fait pas acceptable en l'état ? À titre d'indication, savez-vous comment sont gérées les interruptions asynchrones sur les processus modernes, et les signaux sous Unix ?

4 Jouons avec la mémoire

Exercise 7 : Copie Profonde

On considère l'extrait de code Java suivant :

```
public class Node{
  private int value;
  private Node left, right;
  public Node(v, l, r){
    value = v; left = l; right = r;
  }
  public Node(Node n){
    value = n.value; left = n.left; right = n.right;
  }
```

```
  // accesseurs, mutateurs... omis
}
...
  Node a = new Node(1, new Node(2, null, null),
                       new Node(3,
                           new Node(4, null, null), null));
```

1. Dessiner un diagramme boîtes-flèches décrivant la mémoire après l'exécution du code ci-dessus.
2. Combien vaut `a.left.value` après exécution ? `a.right.left.value` ?
3. Combien vaut `a.left.value` après exécution du code suivant, exécuté à la suite du code ci-dessus ?

   ```
   Node b = new Node (5, new Node (a), null);
   b.left.left = new Node (6, null, null);
   ```

4. Pareil mais avec le code suivant :

   ```
   Node b = new Node (7, new Node (a), null);
   b.left.left.value = 8;
   ```

5. Écrire une méthode `deepCopy()` qui effectue une *copie profonde*, c'est-à-dire qui copie tous les nœuds de l'arbre, et pas seulement la racine.
6. Que se passe-t-il si l'on lance le code suivant ?

   ```
   a.left = a;
   Node b = a.deepCopy();
   ```

7. Comment pourrait-on corriger le problème ? Le jeu en vaut-il la chandelle ?

Exercise 8 : Structures en C, pointeurs sur sturctures

On définit les structures suivantes en C.

```
struct s1 { int i; };
struct s2 { struct s1 s; };
struct s3 { struct s1 *p; };
```

1. Écrire deux fragments de code C qui créent des structures de type `struct s2`, resp. `struct s3`, contenant une structure contenant la valeur 42.
2. Dessiner un diagramme boîtes-flèches de la mémoire après l'exécution du code suivant :

   ```
   struct s1 s1; // oui, on peut donner le meme nom a une
   struct s2 s2; // variable et a un type struct...
   struct s3 s3, ss3;
   s1.i = 42; s2.s = s1;
   s3.p = &s1;
   ss3.p = malloc (sizeof (struct s1)); ss3.p->i = 54;
   ```

   On différenciera bien le tas de la pile.

3. On définit la fonction suivante :

   ```
   struct s1 *f(void) { // (void) = ne prend pas d'argument
     struct s1 s;
     s.i = 42;
     return &s;
   }
   ```

   Que se passe-t-il lorsqu'on exécute cette fonction ? En profiter pour répondre à la question : à quoi sert `malloc()` ?

4. On définit la fonction suivante :

```
struct s3 *f(void) {
  struct s1 s;   s.i = 9;
  struct s3 *p3 = malloc (sizeof (struct s3));
  p3->p = &s;
  return p3;
}
```

Que se passe-t-il lorsqu'on exécute cette fonction ? Que doit-on faire pour corriger le problème ?

5. Quelle est la différence entre les deux déclarations de structures suivantes ? La question ne porte pas tant sur l'absence ou la présence d'un `*`, mais sur la façon dont sont représentés les objets des deux types en mémoire.

```
struct info1 { int value; struct s1 s1; };
struct info2 { int value; struct s1 *p1; };
```

6. L'une des deux déclarations suivantes sera rejetée par le compilateur C :

```
struct tree1 { int value; struct tree1 left, right; };
struct tree2 { int value; struct tree2 *left, *right; };
```

Laquelle ? Pourquoi ?

7. En Java, la seule déclaration autorisée d'un type équivalent sera celle qui ne mentionne pas « `*` », et pour cause : `*` n'existe pas en Java. La voici :

```
public class Tree {
  int value;
  Tree left, right;
  // methodes omises
}
```

De laquelle des définitions de la question précédente se rapproche-t-elle, en termes de représentation en mémoire ?

8. Pourquoi les deux formes d'« inclusions » de structures (voir la question 5) sont-elles autorisées en C ? (Seule la premiàre est effectivement appelée inclusion, au passage.)
9. Pourquoi Java et Caml n'autorisent-ils qu'une seule des formes d'« inclusion » ?

Exercise 9 : Blocs mémoire et chaînes

1. Écrire la fonction :

```
void *memchr(const void *s, int c, size_t n);
```

dont voici un extrait de la page man :

```
DESCRIPTION
     The memchr() function locates the first occurrence
     of c (converted to an unsigned char) in string s.

RETURN VALUES
     The memchr() function returns a pointer to the byte
     located, or NULL if no such byte exists within n bytes.
```

Le type `size_t` est un type entier, réservé aux specifications de longueurs. Le type `void *` est le type des pointeurs vers n'importe quel type. (Ce n'est *pas* le type des pointeurs vers les objets de type `void`... il n'y a pas d'objet de type `void`.) Le modificateur `const` promet au compilateur que la fonction ne modifiera rien de ce qui est pointé par `s` (mais la signification de `const` est, à mon avis, des plus obscures).

2. Écrire la fonction :

```
char *strchr (const char *s, int c);
```

dont la page man dit :

```
DESCRIPTION
     The strchr() function locates the first occurrence of c
 (converted to a char) in the string pointed to by s.
 The terminating null character is considered to be part
 of the string; therefore if c is '\0', the function locates
 the terminating '\0'.

RETURN VALUES
     The function strchr() returns a pointer to the located
 character, or NULL if the character does not appear in the
 string.
```

3. Écrire les fonctions :

```
char *strstr(const char *haystack, const char *needle);
void *memmem(const void *haystack, size_t haystacklen,
             const void *needle, size_t needlelen);
```

   qui retournent la première position de `needle` dans `haystack`. On ne demande pas d'algorithme subtil (Boyer-Moore, Knuth-Morris-Pratt, automate de Simon, ou autre...).

4. Que pensez-vous de la façon d'implémenter les chaînes de caractères en C ? Qu'y aurait-il comme autre façon de faire ? Comment d'autres langages représentent-ils les chaînes de caractères, d'après vous ?