Programmation 1 – TD 3

Les questions de cette feuille qui ne sont pas dans le poly sont dues à Juliusz Chroboczek, que je remercie.

1 Pointeurs en C

1. Un truc pour obtenir le type d'une variable C est de regarder sa déclaration, par exemple `int i`, et d'enlever le nom de la variable (ici, `i`) : le type de `i` dans ce cas est juste `int`. 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.

3. (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 $(\mathtt{i}+1)$-ième objet stocké au tableau démarrant à l'adresse `a`. Justifier les égalités suivantes : (a) `a[i]=*(a+i)`, (b) `a+i=&a[i]`, (c) `&*p=p`, (d) `*&x=x`, (e) `p[i]=i[p]`, (f) `(&p[i])[j]=p[i+j]`.

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

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

   Dessiner schématiquement la liste des cases du tableau `p` en mémoire. On réalisera qu'il s'agit juste d'un tableau de 30 éléments, chaque élément étant un tableau de 40 éléments. 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])); /* des types de plus en plus fous!
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]));
```

5. 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 */
}
```

2 Tableaux en C et en assembleur

6. 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 à 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.

7. 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 ?

8. 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 ?