TD/TP 8 — corrigés partiels

1 Expressions

```
int i1 = 3;
printf("r1 = %d\n", i1++);
```

3 ; après, `i1` vaut 4.

```
int i2 = 3;
printf("r2 = %d\n", ++i2);
```

4.

```
int i3 = 3;
printf("r3 = %d\n", i3++ + i3++);
```

Incorrect : entre deux modifications de la même variable, il doit y avoir un point de séquencement.

```
int i4 = 3;
printf("r4 = %d\n", i4++ + i4);
```

Incorrect : entre la modification d'une variable et sa lecture, il doit y avoir un point de séquencement.

```
int i5 = 3;
printf("r5 = %d\n", i5 ^ i5);
```

0 (le ou exclusif d'un nombre avec lui-même est toujours 0).

```
int i6;
i6 = 4 | 2;
printf("r6 = %d\n", i6);
```

6 (ou binaire de 4 et 2).

```
int i7;
i7 = 4 || 2;
printf("r7 = %d\n", i7);
```

1 (le ou logique de deux expressions vaut 1 dès lors que l'une au moins des deux expressions est non nulle.)

```
int i8;
i8 = (4 || 2);
printf("r8 = %d\n", i8);
```

1, comme précédemment.

```
int i9 = 3, j9 = 3, k9 = 3;
printf("r9 = %d\n", i9 == j9 == k9);
```

0 (`i9 == j9 == k9` est équivalent à `(i9 == j9) == k9` ; `i9 == j9` vaut 1, et `1 == k9` vaut 0).

```
int i10 = 3;
printf("r10 = %d\n", i10 == 4 | 2);
```

2 (la précédence des opérateurs fait que `i10 == 4 | 2` signifie `(i10 == 4) | 2`, donc le nombre affiché est `0 | 2` qui vaut 2).

Un *point de séquencement* est une « barrière » dans le programme, telle qu'on sait qu'une partie du programme est exécutée avant et le reste est exécuté après.

Le principal moyen d'introduire un point de séquencement est le point-virgule : par exemple, avec « `i++; i++;` », on est sûr que `i` est incrémenté deux fois. La virgule quand elle est opérateur de séquencement ainsi que les opérateurs logiques `&&`, `||` et `?:` introduisent des points de séquencement. Il y a également un point de séquencement à la fin de la condition d'un `if`, d'un `switch` ou d'un `while`, ainsi qu'au début de l'exécution du corps d'une fonction.

En revanche, la plupart des opérateurs et la virgule dans un appel de fonction *n'introduisent pas* de point de séquencement : dans « `i++ + i++` », par exemple, il n'y a pas de point de séquencement entre les deux `i++`, et il en est de même dans `f(i++, i++)`.

Si on modifie une variable (ou, plus généralement, une case mémoire, par exemple `*p`) entre deux points de séquencement, on n'a pas le droit d'accéder en plus à la valeur cette variable, ou de la modifier une deuxième fois.

Voici quelques exemples :

<table>
<tr><td>(i=1) + (i=2)</td><td>incorrect (deux modifications)</td></tr>
<tr><td>(i=1) + i</td><td>incorrect (une modification et un accès)</td></tr>
<tr><td>(i=1) || (i=2)</td><td>correct (|| est un point de séquencement)</td></tr>
<tr><td>i + i</td><td>correct (on ne fait que lire i)</td></tr>
<tr><td>f(&i) + f(&i) avec
int f(int *p){return p++;}</td><td>correct (le ; de f est un point de séquencement)</td></tr>
<tr><td>g(i++) + g(i)</td><td>incorrect (une modification et un accès)</td></tr>
<tr><td>h(i, g(i++))</td><td>incorrect (i et g(i++) ne sont pas séparés par un point de séquencement, même si i++)</td></tr>
<tr><td>e(i++) où i est une variable globale et void e(int k){i+=k;}</td><td>correct (il y a un point de séquencement entre l'évaluation de l'argument i++ et l'exécution du corps de la fonction e)</td></tr>
<tr><td>i=(++i)</td><td>incorrect (pas de point de séquencement entre les deux modifications de i)</td></tr>
</table>

Pourquoi toutes ces complications ? Essentiellement pour laisser au compilateur suffisamment de latitude pour qu'il puisse produire du code efficace. Par exemple, étant donné le code `(*p)++ + (*q)++`, à compiler sur une machine où les accès mémoires prennent plusieurs cycles d'horloge, le compilateur peut générer du code qui obtient `*p` et `*q` en parallèle ou en léger différé, effectue chaque incrémentation dès que l'accès mémoire est achevé, et fait la modification de `*p` et `*q` quand il veut. Si `p` et `q` désignent la même case mémoire, on ne sait pas dans quel ordre les choses vont se passer ; et le langage C demande au programmeur de s'assurer que ceci n'arrive pas : si `p` et `q` peuvent désigner la même case mémoire, le code `(*p)++ + (*q)++` est incorrect.

2 Notation binaire

Écrire une fonction de conversion d'un nombre entier en écriture binaire. Écrire une fonction qui fait la conversion inverse.

Pour les fonctions `xmalloc` et `xrealloc`, voir le TP 7.

La fonction `ecriture_binaire` construit la représentation binaire de `x` en partant du chiffre le moins significatif, c'est-à-dire le chiffre de droite. Ainsi l'écriture binaire se retrouve dans la chaîne à partir de la position `p`. L'appel à `memmove` sert à déplacer les chiffres de la position `p` au début de `s`.

La taille initiale donnée à `s` est la taille maximale que peut occuper un nombre qui rentre dans le type `unsigned` : il y a `sizeof(unsigned)` `char`s dans un `unsigned`, et `CHAR_BIT` bits dans un `char`. On prévoit un caractère de plus pour le caractère nul final.

Une autre méthode serait de trouver la position du bit le plus à gauche de `x`, et d'en déduire la taille exacte nécessaire pour `s`.

La fonction `entier_de_binaire` est plus simple : on construit le nombre à partir de ses chiffres, de gauche à droite, sachant que rajouter un chiffre c à droite correspond à multiplier le nombre par 2 et ajouter c.

```
#include <limits.h>
#include <stdlib.h>
#include <string.h>
void *xmalloc(size_t);
void *xrealloc(void*, size_t);
#define UNSIGNED_BITS (sizeof(unsigned) * CHAR_BIT)
char *ecriture_binaire(unsigned x) {
    char *s = xmalloc(UNSIGNED_BITS + 1);
    char *p = s + UNSIGNED_BITS;
    *p = 0;
    for (; x != 0; x >>= 1)
        *--p = '0' + (x & 1);
    if (p == s + UNSIGNED_BITS) *--p = '0';
    memmove(s, p, s + UNSIGNED_BITS + 1 - p);
    return xrealloc(s, s + UNSIGNED_BITS + 1 - p);
}
unsigned entier_de_binaire(char *s) {
```

```
    unsigned x = 0;
    for (; *s != 0; s++) {x = (x << 1) | (*s - '0');}
    return x;
}
```

3 Opérations binaires

(a) Écrire une fonction qui compte le nombre de bits 1 dans un entier. Par exemple :

```
int nombre_de_bits(unsigned x) {
    int k;
    for (k = 0; x != 0; x >>= 1) if (x & 1) ++k;
    return k;
}
```

(b) Écrire une fonction qui prend un argument entier $n > 0$ et qui calcule l'entier k tel que $2^k \leq n < 2^{k+1}$. k est la position du plus grand bit de n qui est à 1. Pour l'argument 0, renvoyer -1.

```
int logarithme_entier(unsigned x) {
    int k;
    for (k = -1; x != 0; x >>= 1) ++k;
    return k;
}
```

4 Découpage d'une chaîne en mots

Écrire une fonction qui découpe une chaîne de caractères en mots (séparés par un ou plusieurs espaces, sauts de lignes, etc). Cette fonction devra renvoyer un pointeur vers un tableau de pointeurs vers des chaînes de caractères (les mots en question). Utiliser un pointeur nul pour marquer la fin du tableau.

Je donne ici deux solutions. La fonction `decoupe1` fait une copie de chaque mot, tandis que `decoupe2` fait une copie de toute la chaîne et remplace les espaces par des caractères nuls.

Les deux solutions font appel à une fonction `compte_mots`, qui calcule le nombre de mots présents dans la chaîne `s`. Cette fonction repère les débuts de mots : ce sont les caractères autres que les espaces qui ne font pas partie d'un mot en cours (`dans_mot` est vrai si un mot est en cours).

La boucle principale de la fonction `decoupe1` ressemble beaucoup à `compte_mots`. Cette fois, quand on arrive en fin de mot, en plus de compter, on place une copie du mot dans `t` ; et si on est dans un mot, on se souvient de l'adresse du début (`(debut != NULL)` remplace `dans_mot`).

Le fonctionnement de `decoupe2` est un peu plus compliqué. Ici, comme on modifie `s` au vol, on ne peut plus utiliser le test `*s != 0` pour s'arreter : à la place, on s'arrete quand on a trouvé tous les mots.

On fournit également les fonctions `free_decoupe1` et `free_decoupe2`, qui libèrent la mémoire allouée par la fonction `decoupe`*n* correspondante. Noter que `decoupe2` stocke l'adresse de la copie de la chaîne originale dans `t[n0+1]`, afin que `decoupe2` l'y retrouve (on n'a `t[n0+1] == t[0]` que si `s` commence par un non-espace).

Selon les cas, on peut préférer `decoupe1` ou `decoupe2`. S'il y a rarement plus d'un espace entre deux mots, `decoupe2` consomme un peu moins de mémoire (autant de mémoire allouée explicitement, mais seulement deux `malloc` au lieu de $n + 1$, donc on y gagne sur les structures de données internes de `malloc`). S'il y a de longues suites d'espaces dans la chaîne, `decoupe1` est préférable, puisqu'elle ne les stocke pas. En outre, `decoupe1` permet de libérer la mémoire associée à certains mots tout en en conservant d'autres.

```c
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void *xmalloc(size_t);
char *dupstr(char *);
size_t compte_mots(char *s) {
  int dans_mot = 0;
  size_t compte = 0;
  do {
    if (isspace((unsigned char) *s) ||
        *s == 0)
      dans_mot = 0;
    else if (!dans_mot) {
      dans_mot = 1;
      ++compte;
    }
  } while (*s++ != 0);
  return compte;
}
char **decoupe1(char *s) {
  size_t n = compte_mots(s);
  char **t = xmalloc((n+1) * sizeof(*t));
  char *debut = NULL;
  t[n] = NULL; n = 0;
  do {
    if (isspace((unsigned char) *s) ||
        *s == 0) {
      if (debut != NULL) {
        t[n] = xmalloc(s - debut + 1);
        t[n][s - debut] = 0;
        memcpy(t[n], debut, s - debut);
        debut = NULL;
        ++n;
      }
    } else if (debut == NULL) {
      debut = s;
    }
  } while (*s++ != 0);
  return t;
}
void free_decoupe1(char **t) {
  char **p;
  for (p = t; *p; p++) free(p);
  free(t);
}
char **decoupe2(char *s0) {
  char *s = dupstr(s0);
  size_t n0 = compte_mots(s);
  size_t n = n0;
  char **t = xmalloc((n+2) * sizeof(*t));
  char *debut = NULL;
  t[n0+1] = s; t[n0] = NULL; n = 0;
  do {
    if (isspace((unsigned char) *s) ||
        *s == 0) {
      *s = 0;
      if (debut != NULL) {
        t[n] = debut;
        debut = NULL;
        ++n;
      }
    } else if (debut == NULL) {
      debut = s;
    }
    ++s;
  } while (n < n0);
  return t;
}
void free_decoupe2(char **t) {
  char **p;
  for (p = t; *p; p++);
  free(*(p+1));
  free(t);
}
```