TD/TP 5 et 6 — corrigés partiels

1 Vrai ou faux

Dans chacun des fragments de programme suivants, indiquer si le test à la fin est vrai ou faux. Il y a des pièges...

`char *a = "coucou"; char *b = a; if (a == b)` ... Vrai. Après `b = a`, `a` est égal à `b`.

`char *a = "coucou"; char *b = "coucou"; if (a == b)` ... Ça dépend des compilateurs. Une chaîne littérale n'est pas modifiable (le compilateur peut la placer dans une zone de mémoire en lecture seule). Du coup, le compilateur a le droit de stocker une seule fois la chaîne `"coucou"` pour tout le programme. Mais ce n'est pas une obligation.

`char a[7] = "coucou"; char b[7] = a; if (a == b)` ... Ceci n'est pas un programme valide. En effet, on ne peut pas faire d'affectation sur un tableau.

`char a[7] = "coucou"; char b[7] = "coucou"; if (a == b)` ... Faux. Les noms `a` et `b` désignent deux tableaux distincts, et l'opérateur == compare leurs adresses.

`char a[7] = "coucou"; char *b = a; if (a == b)` ... Vrai. La valeur de la variable pointeur `b` est l'adresse du tableau `a`, et dans l'expression `a == b`, le tableau `a` se dégrade en son adresse.

`char a[7] = "coucou"; char *b = a + 3; if (a[1] == b[1])` ... Vrai, il s'agit dans les deux cas du caractère `'o'`.

`char a[7] = "coucou"; char *b = a + 3; if (&a[1] == &b[1])` ... Faux. Le membre de gauche de l'égalité est équivalent à `a + 1` (un pointeur vers le deuxième caractère de `a`), tandis que le membre de droite pointe sur le deuxième caractère de `b` qui est le cinquième caractère de `a`.

`char a[7] = "coucou"; char *b = a + 3; if (&a[3] == b)` ... Vrai. Il s'agit dans les deux cas d'un pointeur vers le quatrième caractère de `a`.

`char a[7] = "coucou"; char *b = a + 3; if (&a[5] == b + 5)` ... Ceci n'est pas un programme valide. En effet, `b + 5` pointe au-delà du tableau `a`, ce qui est incorrect.

`char *a = "coucou"; char *b = "coucou"; if (a = b)` ... Vrai. La valeur d'une expression d'affectation est la valeur affectée, donc `if (a = b) ...` équivaut à `a = b; if (b) ...`. De plus, dans une conditionnelle, le pointeur `NULL` est faux, et tout autre pointeur est vrai.

`char a[7] = "coucou"; char b[7] = "coucou"; if (a = b)` ... Ceci n'est pas un programme valide. Voir le troisième cas.

2 Ligne de commande

Écrire un programme qui suppose que tous ses arguments sont des nombres entiers écrits sous forme décimale et qui calcule et affiche leur somme. Si un des arguments n'est pas un nombre entier, ne rien afficher, mais faire retourner au programme la valeur `EXIT_FAILURE`.

On utilise la fonction standard `strtol` (déclarée dans `stdlib.h`). Son prototype est

```
long strtol(char *nptr, char **endptr, int base);
```

Normalement, la chaîne `nptr` contient seulement des chiffres en base `base` (éventuellement précédés d'un signe + ou -). La fonction `strtol` convertit cette suite de chiffres en un nombre entier (avec le type `long`).

Si `endptr` est le pointeur nul, c'est tout ce qui se passe. Sinon, `endptr` doit pointer sur une variable de type `char *`. La fonction `strtol` place dans cette variable l'adresse du premier caractère autre qu'un chiffre qu'elle rencontre. Si tout se passe bien, c'est un caractère nul. Sinon, ce n'est ni un chiffre, ni un caractère nul : la chaîne n'est pas une représentation valide d'un nombre entier. Tester si `**endptr` est nul permet donc de repérer les erreurs.

Il y a un cas particulier : si `nptr` est une chaîne vide, `**endptr` est un caractère nul, et `strtol` renvoie le nombre 0.

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main (int argc, char **argv) {
    long somme = 0;
    char *e;
    int i;
    for (i = 1; i < argc; i++) {
        somme += strtol(argv[i], &e, 10);
        if (*e != 0 || argv[i][0] == 0) {
            fprintf(stderr, "L'argument %d n'est pas numérique.\n", i);
            exit(EXIT_FAILURE);
        }
    }
    printf("%ld\n", somme);
    return EXIT_SUCCESS;
}
```

3 Comparaison de deux chaînes

Écrire une fonction qui compare deux chaînes de caractères. La fonction devra renvoyer 1 *si les chaînes sont identiques et* 0 *sinon. Par exemples :*

`chaines_identiques("coucou", "coucou")` $\Longrightarrow$ `1`
`chaines_identiques("coucou", "bonjour")` $\Longrightarrow$ `0`
`chaines_identiques("cou", "coucou")` $\Longrightarrow$ `0`

Voici deux versions, l'une utilisant un indice commun pour parcourir les deux chaînes, l'autre utilisant un pointeur qui parcourt chaque chaîne. Les deux se valent en termes d'efficacité et de clarté.

```
int chaines_identiques(char *s, char *t) {
    int i;
    for (i = 0; s[i] != 0 && t[i] != 0; i++)
        if (s[i] != t[i]) return 0;
    return 1;
}
```

```
int chaines_identiques(char *s, char *t) {
    char *a; char *b;
    for (a = s, b = t; *a != 0 && *b != 0; a++, b++)
        if (*a != *b) return 0;
    return 1;
}
```

4 Lire et écrire des caractères

Écrire deux programmes qui comptent les lignes entrées sur `stdin`*, l'un au moyen de* `getchar`*, puis l'autre au moyen de* `fgets`*. Pour le deuxième, on pourra dans un premier temps supposer que les lignes ne font pas plus de 100 caractères.*

L'idiome `while ((c = getchar()) != EOF)` est très répandu, c'est la manière habituelle de parcourir un fichier caractère par caractère. Rappel : la variable `c` doit être un `int`, sinon ce programme s'arrête dès qu'il rencontre le caractère `(char)-1`.

```
#include <stdio.h>
int main () {
    int lignes = 0;
    int c;
    while ((c = getchar()) != EOF)
        if (c == '\n') ++lignes;
    printf("%d\n", lignes);
    return 0;
}
```

Voici maintenant une version utilisant `fgets`. Cette version a plusieurs inconvénients. D'abord elle est plus difficile à écrire, parce qu'il faut appeler `fgets` plusieurs fois pour lire entièrement les lignes trop longues.

Ensuite et surtout les lignes qui contiennent un caractère nul ne sont pas gérées correctement. C'est pourquoi il est souvent déconseillé d'utiliser la fonction `fgets`, même si on la trouve dans beaucoup de programmes existants.

```
#include <stdio.h>
#include <string.h>
#define TAILLE_TAMPON 100
int main () {
    unsigned lignes;
    char tampon[TAILLE_TAMPON];
    while (fgets(tampon, TAILLE_TAMPON, stdin)) {
        while (! strchr(tampon, '\n'))
            fgets(tampon, TAILLE_TAMPON, stdin);
        ++lignes;
    }
    return 0;
}
```

Les deux programmes présentés ici ne comptent pas la dernière ligne d'un fichier si elle n'est pas terminée par un saut de ligne. Il existe des implémentations de C sur lesquelles tout fichier texte doit se terminer par un saut de ligne ; ce n'est pas le cas d'Unix ni de DOS.

5 Comptage de mos

Écrire un programme qui compte les mots du fichier passé sur son entrée standard. Un mot est une séquence de caractères qui ne sont pas des caractères d'espacement.

Le principe de ce programme est de compter un mot chaque fois qu'on passe d'un caractère d'espacement à un caractère non d'espacement, ou l'inverse. Il faut faire attention à traiter correctement le début et la fin du fichier, qui peuvent chacun être blanc ou non.

```
#include <ctype.h>
#include <stdio.h>
int main () {
    int mots = 0;
    int b = ' ', c;
    while ((c = getchar()) != EOF) {
        if (!isspace((unsigned char)c) && isspace((unsigned char)b)) ++mots;
        b = c;
    }
    printf("%d\n", mots);
    return 0;
}
```

6 Graphe d'une fonction (le retour)

Écrire une fonction qui dessine le graphe d'une fonction numérique. Le prototype de cette fonction sera

```
void dessine_graphe(double (*f)(double));
```

Il s'agit juste d'utiliser un pointeur sur une fonction. Pour lire le type de l'argument de `dessine_graphe`, on va de l'intérieur vers l'extérieur :

- on part de `f` ;
- l'étoile signifie que `f` est un pointeur vers un type τ ; τ est le type de `*f` ;
- `double g(double)` signifie que `g` est une fonction prenant un argument de type `double` et renvoyant un `double`. Donc `f` est un pointeur vers une telle fonction.

On suppose définies quatre variables globales : `x0` et `y0` sont les coordonnées du graphe correspondant au pixel en bas à gauche de la fenêtre de dessin, et `pas_x` et `pas_y` sont les échelles horizontale et verticale du graphe.

```
void dessine_graphe(double (*f)(double)) {
    int taille_horizontale, i;
    taille_horizontale = gr_size_x();
    for (i = 0; i < taille_x; i++) {
        gr_plot(i, (*f)(x0 + i * pas_x) / pas_y - y0);
    }
}
```

7 Concaténation de fichiers

Écrire un programme qui concatène (met bout à bout) les fichiers qui lui sont passés en arguments. C'est la commande Unix `cat`*. Par exemple,* `cat toto tutu` *affiche sur sa sortie standard le contenu du fichier* `toto` *puis celui du fichier* `tutu`*. Ne pas oublier de traiter le cas où un fichier n'existe pas.*

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char **argv) {
    int err = EXIT_SUCCESS;
    int i;
    int c;
    FILE *f;
    for (i = 1; i < argc; ++i) {
        if ((f = fopen(argv[i], "r"))) {
            while ((c = fgetc(f)) != EOF) putchar(c);
            fclose(f);
        } else {
            fprintf(stderr, "Impossible de lire le fichier %s\n", argv[i]);
            err = EXIT_FAILURE;
        }
    }
    return err;
}
```

8 Recherche d'une sous-chaîne

Écrire une fonction qui recherche une chaîne dans une autre. Renvoyer le plus petit indice auquel la sous-chaîne commence, ou -1 si la sous-chaîne n'est pas présente.

On compare la sous-chaîne recherchée (`mot`) à la chaîne `s`, puis on se décale d'un cran dans `s` et on refait la comparaison, et ainsi de suite. Il faut faire attention de ne pas dépasser la fin de `s`.

La boucle intérieure compare `mot` avec la sous-chaîne de `s` qui commence à la position `i`, et s'interrompt dès qu'elle trouve deux caractères différents. Si on sort normalement de la boucle, on a `(mot[k] == 0)`, et on a trouvé une occurence de `mot` ; sinon on a `(mot[k] != 0)`, et il faut continuer la recherche.

```
#include <string.h>
int indice_sous_chaine (char *s, char *mot) {
    int i, k;
    for (i = 0; i <= (int)strlen(s) - (int)strlen(mot); i++) {
        for (k = 0; mot[k] != 0; k++)
            if (s[i+k] != mot[i]) break;
        if (mot[k] == 0) return i;
    }
    return -1;
}
```

Voici la version que j'ai écrite au tableau lors du TD :

```
int indice_sous_chaine (char *s, char *mot) {
    int i, k;
    for (i = 0; i < strlen(s) - strlen(mot); i++) {
        for (k = 0; mot[k] != 0; k++)
            if (s[i+k] != mot[k]) break;
        if (mot[k] == 0) return i;
    }
    return -1;
}
```

L'erreur qu'il fallait trouver était dans le test d'arrêt `i < strlen(s) - strlen(mot)` : il faut une comparaison au sens large, sinon on rate le cas où la seule occurence de `mot` dans `s` est juste à la fin de `s`.

Il y avait en fait une deuxième erreur dans ce test. Lorsqu'on le compile avec `gcc`, celui-ci signale :

```
grep.c: In function 'indice_sous_chaine':
grep.c:6: warning: comparison between signed and unsigned
```

Cet avertissement concerne la comparaison `i <= strlen(s) - strlen(mot)`. Pour le comprendre, il faut quelques explications sur la façon dont C gère des expressions combinant plusieurs types entiers.

La fonction `strlen` renvoie un type entier sans signe, appelé `size_t`.[1]. Or quand on fait la différence de deux expressions de type sans signe, on a une différence du même type sans signe (les règles de types pour `-` et pour `+` sont les mêmes). Ainsi l'opérateur de comparaison `<` a une opérande de type entier avec signe (`i`) et une opérande de type sans signe.

Lorsqu'un opérateur (`+`, `-`, `*`, `/`, mais aussi les comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`) a deux opérandes de types différents, le compilateur insère une ou deux conversions de types pour se ramener à deux opérandes de même type. Les règles exactes pour choisir ce type commun sont complexes, mais le principe de base est le suivant : on prend le plus petit type qui est capable de représenter toutes les valeurs *positives* des deux opérandes. Par exemple, avec une opérande de type `int` et une de type `size_t`, si (ce qui est en général le cas) `size_t` est au moins aussi grand que `unsigned int`, les deux opérandes sont convertits en des `size_t`.

Donc l'expression `i <= strlen(s) - strlen(mot)` est fausse si `i` est un « petit » entier et `s` est plus courte que `mot`. En effet `strlen(s) - strlen(mot)` est un grand entier(`UINT_MAX - strlen(mot) + strlen(s)` si le type `size_t` est identique à `unsigned int`).

La conversion explicite des résultats de `strlen` en un type avec signe donne au programme le comportement attendu (ne pas rentrer dans la boucle si `mot` est plus court que `s`).

> *Lorsqu'une expression signée est comparée à une expression non signée de même taille, le comportement peut être inattendu.*
>
> B. W. Kernighan et D. M. Ritchie, *Le langage C* (deuxième édition),
> annexe A, paragraphe 6.5 « Conversions arithmétiques »

[1]Le type `size_t` est un déclaré dans les en-têtes `stdlib.h`, `stdio.h`, `string.h` et d'autres encore. Ce type est un type entier sans signe (en général identique à `unsigned int` ou à `unsigned long`), qui est assez grand pour contenir la taille d'un objet. C'est le type retourné par l'opérateur `sizeof` et de ce qu'on passe à `malloc`, c'est le type qu'il faut systématiquement utiliser comme indice de tableau quand on fait les choses proprement.