TP13 : Idées pour la correction

1. a) On devrait obtenir la somme $1+2+3+4$. A la place on obtient une valeur aberrante qui dépend d'ailleurs de l'exécution et de la machine. C'est normal, dans `somme` on accède à une case en dehors du tableau ce qui provoque un comportement indéfini.

   b) Le sanitizer indique un `stack-buffer-overflow` c'est à dire : on a essayé d'accéder à de la mémoire sur la pile à un endroit où on n'a pas le droit. C'est cohérent : le tableau qu'on manipule dans le `main` est statique donc a priori sur la pile. On aurait eu de la même façon un `heap-buffer-overflow` si ce tableau était alloué sur le tas. On corrige en ligne 6 via `i < n`.

2. a) La compilation provoque un warning. A l'exécution on obtient une erreur de segmentation. C'est la mention générique que vous obtiendrez lorsqu'une mauvaise manipulation de la mémoire survient (déréférencement de `NULL`, pointeur non initialisé, oubli d'un `'\0'` en fin de chaîne...). Compiler avec l'option `-fsanitize=address` permet généralement de préciser le problème.

   b) Ici lorsqu'on utilise un sanitizer et l'option `-g` on obtient l'information selon laquelle la ligne 18 du `main` a provoqué une lecture à un endroit interdit. C'est normal : dans la fonction `minimum`, on renvoie l'adresse d'une variable locale à la fonction (c'est d'ailleurs ce que dit le warning) donc qui cesse d'exister quand on en sort. On ne peut donc pas la lire !

   Pour corriger, il faut que l'adresse renvoyée par `minimum` soit celle d'un objet vivant sur le tas car lui continue d'exister en dehors de cette fonction :

```c
int* minimum(int a, int b)
{
    int* res = malloc(sizeof(int));
    if(a < b)
    {
      *res = a;
    }
    else
    {
      *res = b;
    }
    return(res);
}
```

   Bien sûr il faudra ajouter un `free(p)` dans le `main` pour libérer cette mémoire.

   Remarque importante : Certaines personnes ont essayé de corriger le problème via le code présent dans local_pb.c. A priori ça ne corrige rien puisqu'on continue de renvoyer l'adresse de quelque chose sur la pile. Pourtant, quand on compile, même avec `-fsanitize=address`, tout semble se passer normalement... jusqu'à ce qu'on essaie de faire plusieurs appels à la fonction `minimum` (vous devriez obtenir un affichage bizarre). Je confirme que ce code est faux et d'ailleurs vous pouvez le constater en l'exécutant dans C tutor (https://pythontutor.com/c.html#mode=edit) qui vous montre qu'en mémoire ce code ne peut pas marcher.

   La raison pour laquelle le sanitizer ne proteste pas sur les machines du lycée est juste... qu'il est trop vieux. Les machines du lycée utilisent gcc en version 9.4.0 et se dernier se fait visiblement bluffer. Avec une version 13+ de gcc, vous vous ferez insulter comme il se doit par le sanitizer.

   Ceci est une très bonne occasion de constater que les sanitizers ne sont pas des outils magiques et que ce n'est pas parce qu'ils ne râlent pas que le code est correct !

3. a) Après un temps plus ou moins long, 0 est renvoyé (ou pas) ce qui n'est pas le résultat attendu.

   b) Le problème se situe en ligne 14. En théorie, ceci est une boucle infinie car `i` peut croître indéfiniment. En pratique, on produit un dépassement de capacité. Or en C un dépassement de capacité est un comportement indéfini. Donc il peut se passer n'importe quoi et il se trouve que si vous avez de la chance ce n'importe quoi consiste lorsqu'on ajoute 1 au plus gros entier positif possible à repasser dans les négatifs ce qui invalide la condition `i >= 1`, fait sortir de la boucle et renvoyer 0.

   c) Fonction à savoir écrire en 2 minutes. Une version impérative est possible.

```c
int pgcd(int a, int b)
{
  if (a % b == 0) return b;
  else return pgcd(b, a % b);
}
```

4. a) La fonction `tab_n` renvoie un tableau de taille `n` ne contenant que de `n` et `vaut_3` renvoie `true` si et seulement si la première case du tableau en entrée vaut 3. L'affichage attendu est : $0, 1, 0$.

   b) La compilation sans option ne provoque pas d'erreur mais à l'exécution on obtient l'affichage $1, 1, 1$. En compilant avec `-Wall`, on obtient un warning `assignment used as truth value` et en effet, `tab[0] = 3` est une assignation, pas un test d'égalité. C'est ce qui cause le problème : cette assignation réussit dans tous les cas donc renvoie autre chose que 0 donc est considérée comme vraie et donc on renverra systématiquement 1. On corrige en utilisant `==` en ligne 17.

   La compilation avec `-fsanitize=address` provoque aussi une erreur de fuite mémoire. C'est normal : l'appel à `tab_n` dans `vaut_3` alloue de la mémoire sur le tas qui malgré le `free(tab)` final n'est jamais libéré puisque le `return` fait sortir de la fonction avant qu'on atteigne cette instruction. On corrige via :

```c
int vaut_3(int n)
{
    int* tab = tab_n(n);
    if (tab[0] == 3)
    {
        free(tab);
        return 1;
    }
    else
    {
        free(tab);
        return 0;
    }
}
```

5. a) b) Vous pouvez obtenir des affichages très variés et qui dépendent (ou pas) de l'exécution. C'est ça qui est chouette avec les comportements UB, c'est qu'on ne sait jamais à quoi s'attendre...

   Le problème s'identifie mieux avec `-fsanitize=address` qui informe d'un `stack-buffer-overflow` et indique même que c'est la variable `p` qui pose problème. C'est normal, la condition de la boucle en ligne 9 finit par faire des accès hors bornes au tableau `p` ce qui produit un code au comportement indéfini. Et indéfini ça veut dire indéfini donc n'importe quoi peut se passer !

6. a) En compilant sans option, l'exécution montre qu'on n'a pas concaténé les deux chaînes : le début de la chaîne 2 a disparu. Ceci est du à un problème d'indice en ligne 17.

   b) Après cette modification, sans option de compilation, le code semble se comporter normalement. Mais si on compile avec `-fsanitize=address`, le sanitizer vous informe d'un `heap-buffer-overflow` causé par `strlen` à la ligne 27 du `main`. C'est normal : dans `concatener`, on a oublié de réserver de la place pour le caractère final `'\0'`. On corrige via :

```
char* concatener(char* s1, char* s2)
{
    int n1 = strlen(s1);
    int n2 = strlen(s2);
    char* s = (char*) malloc(sizeof(char)*(n1+n2+1));
    for (int i = 0; i < n1; i++)
    {
        s[i] = s1[i];
    }
    for (int i = n1; i < n1+n2; i++)
    {
        s[i] = s2[i-n1];
    }
    s[n1+n2] = '\0';
    return s;
}
```

   On peut néanmoins se demander : puisque la chaîne `s` du `main` n'a pas de caractère de fin de chaîne, comment `strlen` a-t-elle pu calculer sa taille quand on compilait sans le sanitizer ? La réponse est : parce qu'on a eu de la chance. Quand un programme est lancé, le système lui accorde de la mémoire sous forme de "pages" d'octets initialisés à 0. Vous n'avez pas la main sur ça et c'est différent de `malloc`. Du coup dans ce petit code, comme on a juste fait un `malloc` quelque par dans cette page vierge, la case mémoire qui suit immédiatement le dernier caractère de la chaîne est en fait `'\0'` donc `strlen` fonctionne.

   Le problème aurait été visible (peut-être) si on avait appelé ce code après avoir mis du bazar dans la mémoire sur la page. Par exemple, si on alloue des choses dessus puis qu'on les libère, lorsqu'on allouera `s` il n'y aura plus de garantie que le caractère suivant immédiatement le dernier caractère de `s` soit le symbole de fin de chaîne. Par exemple avec le code chaine_pb.c, j'obtiens l'affichage non désiré (qui peut varier selon les machines et exécutions) :

```
totititi***************, 27
```

7. a) On devrait avoir (les listes étant à chaque fois des listes chaînées d'entiers) :

   - `est_vide` indique si une liste est vide.
   - `queue` et `tete` renvoient respectievement la queue et la tête d'une liste.
   - `cons` ajoute un élément en tête d'une liste donnée en l'étendant.
   - `affiche` permet d'afficher les éléments d'une liste.
   - `libere_liste` libère la mémoire occupée sur le tas par une liste.
   - `concatenation` renvoie la concaténation de deux listes.

b) Si on compile sans option, pas d'erreur. A l'exécution, on a un problème qui s'explicite lorsqu'on compile avec `-fsanitize=address` et `-g` : on a un `heap-use-after-free` causé par l'appel à `queue` dans l'appel à `libere_liste` fait en ligne 98 du `main`.

Pour comprendre on peut déjà constater que l'appel à `libere_liste` en ligne 97 n'a lui pas posé de problème. Donc ce n'est pas forcément `libere_liste` le problème mais son utilisation. En ligne 98, on l'utilise sur une chaîne `c` qui est le résultat de la concaténation d'une chaîne `test` avec elle même. Or `test` vient d'être libérée et si on observe le fonctionnement de `concatenation`, cette libération a libéré indirectement la fin de `c`.

Par conséquent, lorsqu'on libère `c`, à la moitié de cette liste, on tente de libérer récursivement des maillons qui ont déjà été libérés donc d'utiliser de la mémoire sur le tas après un `free`. Pour corriger le problème, il suffit de supprimer la libération de `test` dans le `main` car celle de `c` s'en chargera indirectement.

Remarque : Il vaudrait en fait mieux réécire la concaténation de sorte à en décorréler complètement le résultat des listes en entrée. Ce n'est pas difficile, il suffit en ligne 71 de renvoyer une copie de `l2` plutôt que `l2` via une fonction de copie que je vous laisse écrire.