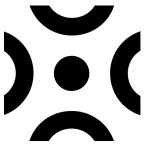
Perfectionnement à la programmation en C

Fiche de TP 1

L2 Informatique - 2024–2025

Compilation et débogage



Objectifs du TP:

- Utiliser un débogueur
- Utiliser des outils de diagnositique (`valgrind`, `sanitizer`)

1 Rappels

Pour compiler un programme on utilisera `clang` ou `gcc` et toujours en précisant :

- le nom de l'exécutable à produire : `-o mon_programme`
- les options d'avertissements : `-Wall`
- le standard suivi : `-std=c17`

La commande minimale pour compiler un programme `C` est donc

```
gcc -Wall -std=c17 mon_fichier.c -o mon_programme
```

Le compilateur est votre premier allié. Il faut lui demander de l'aide et l'écouter/le lire. On pourra alors ajouter les options suivante :

- `-pedantic` pour imposer un suivi **strict** de la norme du `C` suivie ;
- `-Wfatal-errors` pour limiter le nombre de messages d'erreur (en cas d'erreurs à la compilation) ;
- `-Werror` pour considérer tout avertissement comme une erreur (se combine bien avec l'option précédente) ;
- `-Wextra` pour encore plus d'avertissements. La pertinence de ces derniers dépend de l'avancement de l'écriture du programme (se combine mal avec l'option précédente) ;

2 Tests et premier pas avec gdb

Bob a (mal) implémenté une recherche dichotimique dans un tableau trié d'entiers.

```
int dichotomie(int v, int *tab, int size) {
    int left = 0;
    int right = size;
    while (left < right) {
        int mid = (left + right) / 2;
        if (tab[mid] == v)
            return 1;
        if (tab[mid] < v)
            left = mid;
        else
            right = mid;
    }
    return -1;
}
```

Exercice 1

1. Écrire un programme testant la fonction sur quelques exemples.

   On pourra utiliser le tableau {0, 2, 4, 6, 8} et chercher les valeurs entre 0 et 9.

2. Dans quels cas la fonction ne produit-elle pas l'effet souhaité ?

Aidons Bob à corriger sa fonction en utilisant le programme `gdb`.
`gdb` est un débogueur, comme `python tutor` mais en ligne de commande.
Pour pouvoir l'utiliser sur un programme `C`, il faut ajouter l'option `-g` lors de la compilation de ce dernier.
On exécute ensuite `gdb` en lui fournissant l'executable comme paramètre :

```
$ gdb ./main
GNU gdb (GDB) 13.2
[bla bla bla de gdb...]
Reading symbols from ./mon_programme...
(gdb)
```

Si `gdb` vous demande si vous voulez activer `debuginfod`, répondez `no`.

3. À ce stade, le programme de Bob est prêt à être lancé dans l'environnement de deboguage. Taper la commande `run` pour démarrer son exécution.

   Si une erreur de la forme `cannot exec: -c ...` se produit, il faut ajouter au fichier `~/.bashrc` la ligne `export SHELL=/bin/bash` (et créer le fichier s'il n'existe pas)

Le programme se comporte comme attendu : mal dans certains cas. On peut l'interrompre avec la combinaison `Ctrl C`.

4. À quoi correspondent les différentes informations dans ce message ?

5. L'exécution du programme est juste interrompue. Le programme n'est pas tué. On peut reprendre son exécution avec la commande `continue` (ou `c`) ou exécuter instruction par instruction avec la commande `next` (ou `n`).

   Essayez.

6. Pour visualiser l'état des variables du programme, deux commandes sont utiles : `print <nom de variable>` et `display <nom de variable>`.

- `print left` : affiche la valeur de la variable `left` (si elle existe à l'endroit où le programme a été interrompu)
- `display left` : comme `print left` mais le contenu de la variable est affiché à chaque fois que le programme est interrompu.

Affichez pas à pas l'état des variables pour identifier la source du problème de la fonction `dichotomie` de Bob.
Pour quitter `gdb`, on peut utiliser la commande `quit` ou bien utiliser la combinaison `Ctrl D`.

7. *(Pour aller plus loin)* Plutôt que d'exécuter péniblement instruction par instruction, on peut demander à `gdb` de poursuivre l'exécution jusqu'à une instruction particulière du programme. On appelle ça *un point d'arrêt* (ou *breakoint*). Pour définir un point d'arrêt on peut indiquer :

   - le nom de la fonction sur laquelle s'arrêter : `(gdb) break dichotomie`
   - le nom du fichier avec le numéro de la ligne de l'instruction : `break main.c:20`

   En lançant le programme avec `run` ou en poursuivant son exécution avec `continue`, `gdb` interrompra l'exécution quand il tombera sur l'instruction indiqué par un point d'arrêt.

   Identifiez une ligne de la fonction dichotomie qui pourrait servir de `point d'arrêt` puis testez à nouveau.

Exercice 2 (*Dictionnaire de mots*)

Le code suivant ne se comporte pas comme attendu.

```c
#include <stdio.h>
#include <string.h>

typedef struct {
    char *mots[256];
    int taille;
} Dico;

/* Insertion triée */
void inserer_mot(Dico *dico, char *mot) {
    int i;
    if (dico->taille >= 256)
        return;

    for (i = dico->taille - 1; i >= 0 && strcmp(dico->mots[i], mot) > 0; i--) {
        dico->mots[i + 1] = dico->mots[i];
    }
    dico->mots[i + 1] = mot;

    dico->taille++;
}

void afficher_dico(Dico *dico) {
    int i;
    for (i = 0; i < dico->taille; i++)
        printf("%s\n", dico->mots[i]);
}

int main() {
    Dico dico = {0};
    char mot[256] = "";

    dico.taille = 0;

    /* Mots de base */
    inserer_mot(&dico, "zoo");
    inserer_mot(&dico, "anaconda");
    inserer_mot(&dico, "mitochondrie");

    printf("Dico de base\n"
           "============\n");
    afficher_dico(&dico);
    printf("------------\n");

    /* Ajout de mots de l'utilisateur, Ctrl D pour stopper la saisie */
    while (scanf(" %s", mot) > 0) {
        inserer_mot(&dico, mot);
    }
```

```
    printf("Dico complet\n"
           "============\n");
    afficher_dico(&dico);
    printf("------------\n");

    return 0;
}
```

1. Constater le problème.
2. En plaçant un point d'arrêt avant la saisie utilisateur, observer l'évolution du contenu du dictionnaire `dico` avec la commande `print` ou `display` après chaque ajout.

   On peut placer le point d'arrêt via la commande `(gdb) break mon_programme.c:47` où 47 est le numéro de la ligne du `while`
3. Comment faudrait-il corriger le programme afin qu'il liste les mots entrés par l'utilisateur ?

---

3 Algorithme de tri

Le tri rapide est un algorithme de tri utilisant la technique du diviser pour régner. Pour trier un tableau:

- on selectionne une première valeur du tableau appelée *pivot*
- on organise les valeurs du tableau de manière à placer en premier les valeurs plus petites que le pivot, suivies du pivot, suivi des valeurs plus grandes que le pivot ;
- on trie récursivement les petites valeurs, puis les grandes valeurs.

Cela en fait un algorithme de tri relativement simple à écrire. Cependant, voici une implémentation n'aimant pas les doublons.

```
// Tri rapide en place
void quicksort(int tab[], int n) {
    // Si le tableau est vide ou ne contient qu'un élément
    // il est déjà trié
    if (n == 0 || n == 1)
        return;

    // Sinon on selectionne le pivot
    int pivot = tab[0];

    // Et on place toutes les valeurs plus petites que le pivot
    // au début du tableau
    int idx = 1;
    for (int i = 1; i < n; i++) {
        if (tab[i] <= pivot) {
            int tmp = tab[i];
            tab[i] = tab[idx];
            tab[idx++] = tmp;
        }
    }

    // Le pivot est juste après ces valeurs en l'échangeant avec la
    // dernière valeur plus petite que lui
```

```
    tab[0] = tab[idx-1];
    tab[idx-1] = pivot;

    // On trie ensuite la première partie du tableau
    quicksort(tab, idx);
    // Puis la seconde
    quicksort(tab + idx, n - idx);
}
```

Exercice 3

1. Diagnostiquer le problème à l'aide de `gdb`.

   Hint: on pourra utiliser la commande `backtrace` *pour examiner la pile d'appel*

2. Quelle(s) lignes semblent concernée(s) ?

Un autre tri, se prétant bien aux listes chainées, mais qui s'applique aussi pour un tableau est le tri fusion. Dans ce cas:

- On découpe le tableau en deux sous-tableaux de même taille (à la parité près) ;
- On trie récursivement les deux sous-tableaux ;
- On fusionne les deux tableaux triés en un tableau trié.

```
// Tri en utilisant un tableau temporaire
void mergesort(int tab[], int n) {
    // Si le tableau contient 0 ou 1 élément, il est déjà trié
    if (n < 2) return;

    int mid = n / 2;

    // Allocation d'un tableau pour la fusion
    int tmp[n];

    mergesort(tab, mid);
    mergesort(tab + mid, n - mid);

    // On fusionne
    int i = 0;
    int j = mid;
    while (i < mid && j < n) {
        if (tab[i] > tab[j]) {
            tmp[i + j - mid] = tab[i];
            i++;
        } else {
            tmp[i + j - mid] = tab[j];
            j++;
        }
    }

    while (i < mid) {
        tmp[i + n - mid] = tab[i];
        i++;
    }
```

```
    while (j < n) {
        tmp[j] = tab[j];
        j++;
    }

    // Et on recopie le résultat dans le tableau original
    for (int i = 0; i < n; i++) tab[i] = tmp[i];
}
```

De même que pour le tri rapide, il est aisé de faire des erreurs discrètes lors de l'implémentation de cet algorithme.

Exercice 4

1. Tester la fonction sur des tableaux de 1000, 10000, 100000, 1000000 entrées.
2. Quel est le problème ?
3. Proposer une solution.

Variation de cette exercice :
Ce code calcule *bêtement* et récursivement la somme de deux entiers.

```
#include <stdio.h>

int add(int a, int b) {
    if (a < 0) return add(a + 1, b - 1);
    if (a > 0) return add(a - 1, b + 1);
    return b;
}

int main(void) {
    for (int i = 1; i < 10000000; i *= 10) {
        printf("add(%d, %d)\t= %d\n", i - 1, 1, add(i - 1, 1));
    }
    return 0;
}
```

On ne cherchera pas ici à trouver une solution mais à comprendre le problème.

1. Estimer à l'aide de `gdb` la taille de l'espace mémoire réservé à chaque appel de la fonction `add`.

---

4 Instrumentation avec les *sanitizers*

En utilisant `gdb`, on peut exécuter pas à pas un programme pour observer son comportement. Il est aussi possible d'intégrer directement dans un programme, et à faible coup, des outils réalisant automatiquement un ensemble de diagnostiques lors de l'exécution du programme.

Exercice 5 (*Limite du C ?*)
On souhaite calculer les valeurs du triangle de Pascal données par la fonction binomiale :

$$binomial(p, n) = \frac{n!}{p!(n-p)!}$$

1. Exprimer en fonction de n la valeur de $binomial(1, n)$
2. Tester le programme suivant. Observe-t-on les valeurs attendues ?

```c
int factoriel(int n) {
    if (n == 0) return 1;
    return n * factoriel(n-1);
}
int binomial(int p, int n) {
    return factoriel(n) / (factoriel(p) * factoriel(n - p));
}

int main(void) {
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < i; j++) {
            printf(" %d", binomial(j, i));
        }
        printf("\n");
    }
    return 0;
}
```

Lorsqu'on écrit un programme et que ce dernier ne se comporte pas comme attendu, il s'agit généralement d'une erreur dans l'écriture du code.

Ici, l'erreur n'est pas algorithmique, pas de pointeur non plus, ce qui rend sa détection compliquée. L'erreur se cache en fait sur les limites de représentation des valeurs en `C`.

En effet, dans le calcul de la binomiale, on calcule des factorielles. La factorielle de 13 vaut 6'227'020'800 là où la valeur maximale d'un `int` sur nos machines est de $2^{31} - 1$ soit 2'147'483'648.

En ajoutant lors de la compilation l'option `-fsanitize=undefined`, on peut demander au programme de diagnostiquer lui même ce genre de problème.

1. Ajouter `-fsanitize=undefined` lors de la compilation et tester à nouveau le programme.

Cette option intègre l'outil de détection de comportements dit non définis (*Undefined Behavior* ou *UB*). On peut le retrouver dans la nature sous le nom *UBSan*.

Pour corriger le problème, on peut :

- changer le type `int` pour le type `long int` laissant un peu plus de marge pour le calcul
- changer la méthode de calcul de la fonction *binomial*, en changeant astucieusement l'ordre des multiplications et divisions notamment

Dans le cas général, si le dépassement de capacité est un risque nuisible, on doit vérifier qu'il n'ait pas lieu avant de faire le calcul.[1]

Exercice 6 (*Ça passe ou ça casse*)

En `C`, un programme qui marche n'est pas toujours un programme correct. Par exemple, le programme suivant est louche mais ne provoque pas nécessairement d'erreur ni même de fuite mémoire.

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

struct Cellule {
    struct Cellule *suivant;
    int valeur;
};
typedef struct Cellule Cellule, *Liste;
```

[1] la norme C23 introduit un mécanisme pour gérer ces situations avec les macro/fonctions `chk_add`, `chk_add` et `chk_add`

```c
Cellule *extraitTete(Liste *liste) {
    Cellule *cell = *liste;
    // Si la liste est non vide, on retire la cellule de tête
    if (*liste) *liste = (*liste)->suivant;
    return cell;
}

Cellule *insereTete(Liste liste, int valeur) {
    Cellule *tete = malloc(sizeof(Cellule));
    if (tete == NULL) return NULL;
    tete->suivant = liste;
    tete->valeur = valeur;
    return tete;
}

void afficheListe(Liste l) {
    for (Cellule *cell = l; cell != NULL; cell = cell->suivant) {
        printf("%d,", cell->valeur);
    }
    printf("\n");
}

void liberer(Liste *l) {
    while (*l) {
        free(extraitTete(l));
    }
}

int main() {
    Liste l = NULL;
    for (int i = 1; i < 9; i++) {
        l = insereTete(l, (i * 7) % 13);
    }
    afficheListe(l);

    Cellule *cell = extraitTete(&l);
    liberer(&l);

    for (int i = 1; i < 8; i++) {
        l = insereTete(l, - (i * 7) % 13);
    }
    afficheListe(cell); // Affiche les valeurs de `l` ??

    liberer(&cell);

    return 0;
}
```

1. Proposer une explication. Quelle erreur est la cause du problème ?

Pour établir un diagnostic on utilisera deux outils. Le premier, `valgrind`, n'intervient une fois l'exécutable produit.

```
$ valgrind ./mon_programme
==4105== Memcheck, a memory error detector
==4105== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
```

```
==4105== Using Valgrind-3.24.0 and LibVEX; rerun with -h for copyright info
==4105== Command: ./mon_programme
==4105==
4,10,3,9,2,8,1,7,
==4105== Invalid read of size 4
==4105==    at 0x10920F: afficheListe (main.c:28)
==4105==    by 0x10937F: main (main.c:52)
==4105==  Address 0x4a68228 is 8 bytes inside a block of size 16 free'd
==4105==    at 0x484575B: free (vg_replace_malloc.c:989)
==4105==    by 0x109268: liberer (main.c:35)
==4105==    by 0x109319: main (main.c:47)
==4105==  Block was alloc'd at
==4105==    at 0x4842794: malloc (vg_replace_malloc.c:446)
==4105==    by 0x1091C7: insereTete (main.c:19)
==4105==    by 0x1092E3: main (main.c:42)
...
```

`valgrind` simule une partie des opérations de la machine et intercepte les accès mémoire afin de vérifier s'ils sont corrects lors de l'exécution du programme.

Dans l'exemple précédent, `valgrind` détecte que le programme accède à `4` octets d'un bloc de `16` octets alloué dans `insereTete` mais libéré depuis dans la fonction `liberer`.

La seconde méthode consiste à intégrer directement à l'exécutable le code permetant ce type de diagnostic. Celui-ci peut alors être plus fin et ralentit moins l'exécution du programme que `valgrind`.

Pour cela, on compile le programme en ajoutant aux options l'option `-fsanitize=address`. Cette option active l'*AddressSanitizer* (ou *ASan*) en l'intégrant à l'exécutable. Cela a pour effet de controler les accès mémoire et les allocations dynamiques.

2. Compiler en intégrant cette option et exécuter le programme.

Cette fois-ci le programme plante et fournit un message verbeux indiquant le problème, la ligne de code le provoquant en ajoutant ensuite des informations relative à la zone mémoire concernée.

Exercice 7 (*Un peu à côté*)

Le code suivant isole la partie erronnée d'un code plus gros et est adapté pour l'exercice.

Ce code initialise un tableau à partir de la valeur passée en argument au programme dans la ligne de commande et indique si la valeur par défaut est utilisé ou si c'est celle de l'utilisateur qui est utilisée.

```
$ ./mon programme 42
Initialisation par défaut ? Non
$ ./mon programme
Initialisation par défaut ? Oui
```

Voici le code en question.

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

#define SIZE 2

int main(int argc, char *argv[]) {
    long init = argc > 1 ? atoi(argv[1]) : 0;
    long ok = argc > 1;
    long tab[SIZE][SIZE];
    for (int m = SIZE; m >= 0; m--) {
```

```
        for (int n = SIZE; n >= 0; n--) {
            tab[m][n] = init;
        }
    }
    fprintf(stderr, "Initialisation par défaut ? ");
    if (ok)
        fprintf(stderr, "Non\n");
    else
        fprintf(stderr, "Oui\n");
    // ...
    return 0;
}
```

1. Tester en spécifiant la valeur `0` comme argument du programme.[2]
2. Proposer une explication.
3. Tester avec *ASan*.
4. L'explication donnée à la question 2 est-elle correcte ? Proposer une nouvelle explication dans le cas contraire.

Variation de cet exercice :

```
#include <stdio.h>

#define SIZE 25

int main(void) {
    // Calcul des premiers nombes de fibonacci
    int n, m;
    int tab[SIZE] = {1, 1};
    for (m = 2; m <= SIZE; m++) {
        tab[m] = 0;
        for (n = m - 2; n < m; n++)
            tab[m] += tab[n];
    }
    // Puis affichage
    for (m = 0; m < SIZE; m++) {
        printf("%d ", tab[m]);
    }
    return 0;
}
```

Dans cet exemple calculant les premières valeurs de la suite de Fibonacci, le comportement peut dépendre du compilateur. Cela signale la présence d'une erreur.

1. Compiler ce programme avec `gcc` et `clang` et comparer le résultat obtenu
2. Proposer une explication.
3. Tester avec *ASan*.
4. L'explication donnée à la question 2 est-elle correcte ? Proposer une nouvelle explication dans le cas contraire.

[2] le comportement peut varier d'un compilateur à l'autre

5 Quelques mots

Les outils de débogage sont complexes et variés. Il est difficile et long de les maitriser mais ils sont une chance quand les erreurs surviennent et que le classique `fprintf` atteint ses limites.

Si `gdb` peut paraître fastidieux à utiliser, les outils tels que `valgrind`, *ASan* et *UBSan* sont simples à mettre en place. Il n'est jamais trop tôt pour les intégrer dans votre pratique de la programmation en `C` car ils permettent de détecter rapidement les erreurs et donc de les régler avant qu'elles deviennent encombrantes.

Nous verrons dans quelques TP comment les intégrer sans avoir à les taper dans la ligne de commande.

Pour plus d'informations sur ces outils :

- Cheatsheet GDB
- Guide de démarrage de valgrind et sa documentation (plus précisément de son outils memcheck)
- La documentation de GCC pour les *sanitizers* (et oui, il existe tout un tas d'options utilisable avec `gcc`)
- La documentation d'ASan
- La documentation d'UBSan