Projet de Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

Partie 2: Outils

- Chapitre 1: Compilation
- Chapitre 3: Librairie
- Chapitre 3: Tests
- Chapitre 4: Documentation
- Chapitre 5: Débogage
- Chapitre 6: Gestion des Versions

Agenda

- Chapitre 5: Débogage
  - Introduction
  - `printf()`
  - `gdb`
  - Gestion de la Mémoire

Agenda

- Chapitre 5: Débogage
  - Introduction
  - `printf()`
  - `gdb`
  - Gestion de la Mémoire

Introduction

- Plus des 4/5è du temps de programmation ne concernent pas l'écriture de nouvelles lignes de code
  - mais bien le débogage de lignes déjà écrites
    - recherche et correction de bugs
- Rendre plus efficace le débogage est donc essentiel d'un point de vue économique
- Des outils adaptés existent mais s'ils ne sont pas disponibles sur la machine sur laquelle le programme est lancé, il faut recourir à des techniques plus rustiques

Introduction (2)

- Parmi les symptômes des bugs les plus courants, on trouve
  - débordement de tableaux
    - chaînes de caractères comprises
    - dans la pile ou dans le tas
      - erreur en retour de fonction
      - erreur lors des appels de fonctions gérant la mémoire
    - problèmes supplémentaires avec l'allocation dynamique
  - lecture de zones mémoire non initialisées
    - y compris les pointeurs
  - utilisation d'espaces mémoires dynamiques déjà libérés
    - déjà réutilisés ou en voie de réutilisation
  - K. Tsipenyuk, B. Chess, G. McGraw. *Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors*. In IEEE Security & Privacy, 3(6), pg. 81-84. November/December 2005.

Introduction (3)

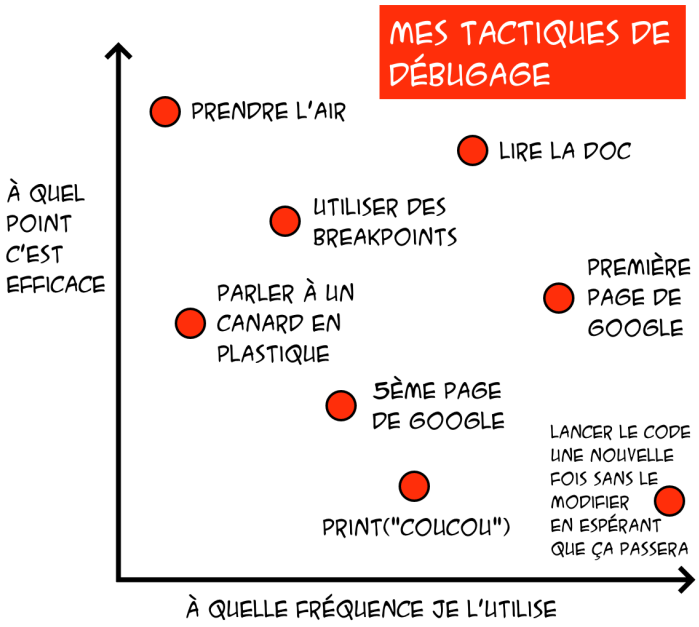
- Une erreur mémoire se présente généralement sous la forme d'un **segmentation fault**
  - *erreur de segmentation*
  - écriture/accès à une zone mémoire non autorisée
- Attention, c'est le noyau (i.e., OS) qui lance le segfault, pas le programme

Introduction (4)

- Un segfault est difficile à déboguer
  - comportement inattendu non détecté au moment de l'erreur
  - une erreur mémoire n'aboutit pas forcément à un segfault
    - ✓ peut aboutir à un comportement indéterminé
      - aléatoire en fonction des exécutions
    - ✓ peut ne pas apparaître
    - ✓ peut modifier le comportement hors de la logique du C
    - ✓ le comportement suspect n'apparaît pas au moment de l'erreur mais plus loin dans le programme

Introduction (5)

- Les grandes "tactiques" du débogage



Agenda

- Chapitre 5: Débogage
  - Introduction
  - `printf()`
    - ✓ Principe
    - ✓ Flots de Sortie
    - ✓ Macros
    - ✓ Discussion
  - gdb
  - Gestion de la Mémoire

Principe

- Comment avoir une idée précise du moment où le programme plante?
  - utilisation de fonctions à effets de bord
- le `printf()` est le moyen le plus classique
  - simple et facile à mettre en oeuvre
  - flexible
    - on affiche ce qu'on veut
  - sortie à l'écran ou dans un fichier (redirection)

Principe (2)

- Exemple

```
void derivation(int valeurs[], int derivees[]){
  for(int k=0; k<TAILLE_MAX;++k)
    derivees[k] = valeurs[k+1] - valeurs[k];
}//fin derivation()

int main(){
  int valeurs[TAILLE_MAX];
  int derivees[TAILLE_MAX];

  for(int k=0; k<TAILLE_MAX; ++k)
    valeurs[k] = 3*k*k;

  derivation(valeurs, derivees);

  return 0;
}//fin programme
```

Principe (3)

- Rajoutons le code suivant

```
int main(){
  //déclarations et initialisation du tableau valeurs
  printf("Valeurs: ");
  affiche_tab(valeurs);
  derivation(valeurs, derivees);
  printf("Dérivées: ");
  affiche_tab(derivees);

  return 0;
}//fin programme
```

- Trace d'exécution

  Valeurs: [0 3 12 27 48 75 108 147 192 243 300 363 432 507 588]
  Dérivées: [3 9 15 21 27 33 39 45 51 57 63 69 75 81 **-588**]

Principe (4)

- Ajout de `printf()` pour voir ce qui se passe

```
void derivation(int valeurs[], int derivees[]){
  for(int k=0; k<TAILLE_MAX;++k){
    printf("%d/%d -> %d\n", k, TAILLE_MAX, valeurs[k]);
    printf("%d/%d -> %d\n", k+1, TAILLE_MAX, valeurs[k+1]);
    derivees[k] = valeurs[k+1] - valeurs[k];
  }//fin for - k
}//fin derivation()
```

- Trace d'exécution

```
0/15 -> 0
1/15 -> 3
...
14/15 -> 588
14/15 -> 588
15/15 -> 0
```

Flots de Sortie

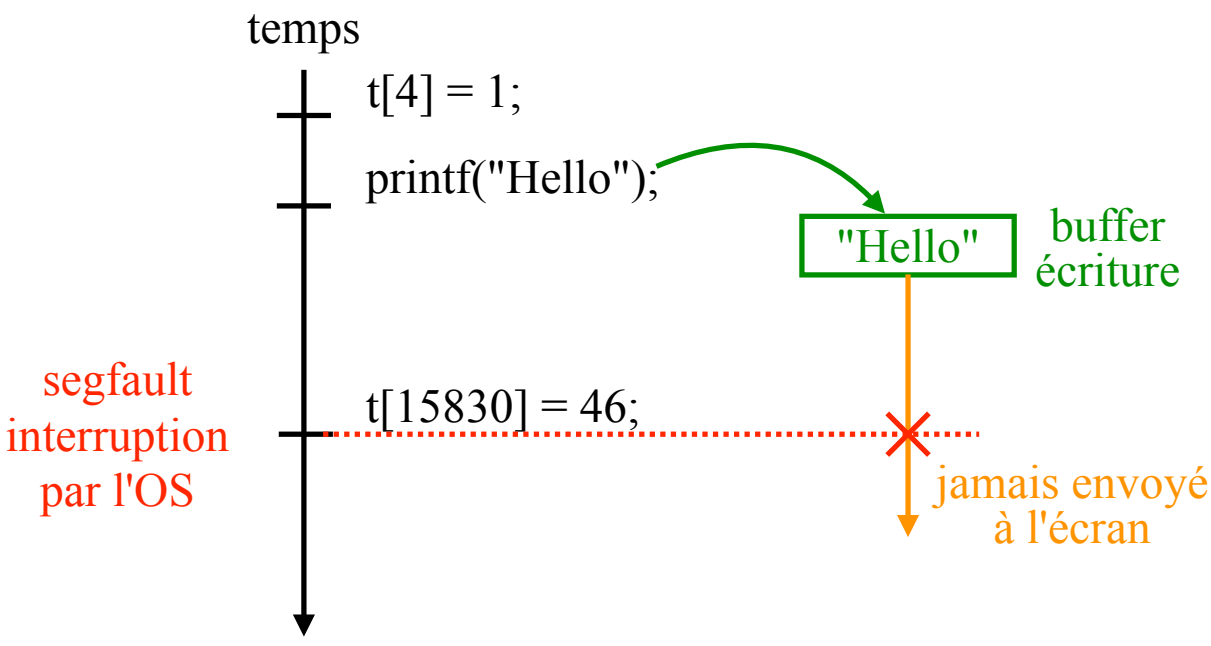
- Attention à l'effet du *buffer* d'écriture

```c
#include <stdio.h>

int main(){
  int t[5];

  t[4] = 1;
  printf("Hello");
  t[15830] = 46;
  printf("World");

  return 0;
}//fin programme
```

```
$>gcc -o main programme.c
$>./main
Segmentation Fault: 11
```


Flots de Sortie (2)

- Explication



Flots de Sortie (3)

- Il existe deux flots de sortie standard
  - `stdout`: sortie standard
    - ✓ associé au descripteur de fichiers 0
    - ✓ bufferisée
      - les données sont écrites dans une zone tampon
      - transmission au système à la fin de chaque ligne (si la destination est un terminal)
      - lorsque le tampon est plein (si la destination est un fichier)
    - ✓ améliore l'efficacité du programme
  - `stderr`: sortie standard d'erreur
    - ✓ sortie standard d'erreur
    - ✓ associé au descripteur de fichiers 2
    - ✓ non bufferisé
      - les données écrites sont immédiatement transmises au système pour affichage immédiat

Flots de Sortie (4)

- Il est donc préférable de ne pas utiliser `printf()`
- Mais plutôt, utiliser `fprintf(stderr, ...)`
  - sinon, des messages écrits avec `printf()` peuvent ne pas encore avoir été envoyés au système quand le programme plante
  - risque de croire que le programme s'est planté en amont du `printf()`
- On peut également utiliser la fonction `fflush(stdout)` pour forcer la vidange du tampon sur la sortie standard

Macros

- Certaines macros prédéfinies peuvent être utile pour le debug
  - `__LINE__`
    - ✓ numéro de ligne sur laquelle la macro a été utilisée
    - ✓ entier décimal constant
  - `__FUNCTION__`
    - ✓ nom de la fonction/procédure dans laquelle la macro a été utilisée
  - `__FILE__`
    - ✓ nom du fichier dans lequel la macro a été utilisée
    - ✓ chemin complet et non juste le résumé du include
  - `__DATE__`
    - ✓ date à laquelle le pré-processeur a été utilisé
  - `__TIME__`
    - ✓ heure à laquelle le pré-processeur a été utilisé

Macros (2)

- Exemple

```
#include <stdio.h>

void affiche_macros(){
  fprintf(stderr, "%d\n", __LINE__);
  fprintf(stderr, "%s\n", __FUNCTION__);
  fprintf(stderr, "%s\n", __FILE__);
  fprintf(stderr, "%s\n", __DATE__);
  fprintf(stderr, "%s\n", __TIME__);
}//fin affiche_macros()

int main(){
  affiche_macros();

  return 0;
}//fin programme
```

Macros (3)

- Passage du pré-processeur

```
$> gcc -E macros.c
```

```
# inclusion de stdio.h
void affiche_macros(){
  fprintf(stderr, "%d\n", 4);
  fprintf(stderr, "%s\n", affiche_macros);
  fprintf(stderr, "%s\n", "macros.c");
  fprintf(stderr, "%s\n", "Mar 25 2015");
  fprintf(stderr, "%s\n", "14:40:31");
}

int main(){
  affiche_macros();

  return 0;
}
```


Macros (4)

- Ces macros sont utiles pour les messages d'erreur
- Exemple

```
void fonction(int *t, int n){
  for(int k=0; ...){
    //du code
    if(erreur)
      fprintf(stderr, "Erreur détectée ligne %d, fonction
      %s, fichier %s\n", __LINE__, __FUNCTION__,
      __FILE__);

    //du code
  }//fin for - k
  //du code
}//fin fonction()
```

Macros (5)

- Exemple complet
  - manipulation de vecteurs
- Fichier `vector.h` (partiel)

```
#define DEBUG(message, indice) \
  fprintf(stderr, "ERREUR (%s%d): ligne %d, fonction \
  [%s], fichier [%s]\n",  message, indice, __LINE__, \
         __FUNCTION__, __FILE__)

#define EPSILON 1e-5

typedef struct Vector_t Vector;

Vector *create_vector();
float norm(Vector *v);
void normalize(Vector *v);
void set_vector_x(Vector *v, float x);
```


Macros (6)

- Fichier `vector_main.c`

```
#include "vector.h"

static void normalize_vector_array(Vector **vectors, int n){
  for(int k=0; k<n; k++){
    float normV = norm(vectors[k]);

    if(normV < EPSILON)
      DEBUG("norme nulle pour k=", k);
    else
      normalize(vectors[k]);
  }//end for - k
}//end normalize_vector_array()
```

Macros (7)

- Fichier `vector_main.c`

```
int main(){
  Vector *v[5];
  for(int k=0; k<5; k++) {
     v[k] = create_vector();
     if (v[k] == NULL)
        // Free already allocated memory
        return 1;
  }

  for(int k=0; k<4; k++)
    set_vector_x(v[k], k+1);

  normalize_vector_array(v, 5);

  return 0;
}//end program
```

oubli d'un vecteur

Macros (8)

- Compilation & exécution

```
$>gcc -o main vector_main.c vector.c
$>./main
ERREUR (norme nulle pour k=4): ligne 11, fonction
[normalize_vector_array], fichier [vector_main.c]
```

Macros (9)

- Plus de détails sur les macros pré-définies disponibles ici
  - https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html

Discussion

- Avantages
  - facile à mettre en oeuvre
  - rapide
  - exécution et debug du vrai code
- Inconvénients
  - nécessite de bien connaître le code
  - nécessite de pré-localiser l'erreur
  - long et fastidieux
    - ✓ génération de grosses masses de traces pour ensuite partir à la pêche aux indices
- Idéalement, on devrait pouvoir le code source s'exécuter pas à pas
  - débogage symbolique

Agenda

- Chapitre 5: Débogage
  - Introduction
  - `printf()`
  - `gdb`
  - Gestion de la Mémoire

gdb

- Débogueur symbolique
- Permet à l'utilisateur
  - d'examiner de façon symbolique les valeurs des variables et du pointeur d'instructions d'un processus ayant terminé anormalement
    - ✓ nécessite un core dump
  - d'exécuter pas à pas un processus actif et d'agir sur le contenu de ses variables

gdb (2)

- Comment utiliser le débogage symbolique?
  - compilation avec l'option `-g`
    - ✓ permet l'ajout d'une table de symboles à la fin de l'exécutable
  - désactiver l'optimisation du code avec l'option `-O0` (ou, au plus, `-O1`)
    - ✓ permet que les numéros de lignes affichés correspondent bien aux lignes du code source

```
$> gcc -g -O0 brol.c -o main
```


gdb (3)

- Utilisation du débogueur

```
$> gdb main
```

lancer le débogueur

```
$> gdb main core.1558
```

analyse du fichier core produit

gdb (4)

- Quelques commandes du débogueur
  - `run [arguments]`
    - ✓ (re)lance le programme
  - `^C`
    - ✓ rend la main à gdb
  - `c`
    - ✓ reprend l'exécution du programme
  - `s`
    - ✓ exécute la ligne courante, en rentrant dans les appels de fonctions
  - `n`
    - ✓ exécute la ligne courante, sans entrer dans les appels de fonctions
  - `u`
    - ✓ exécute jusqu'à la sortie de la boucle

gdb (5)

- Quelques commandes du débogueur (suite)
  - `f`
    - ✓ exécute jusqu'au retour de la fonction
  - `break [nom | ligne]`
    - ✓ positionne un point d'arrêt au début de la fonction ou du numéro de ligne donné
  - `cond numéro condition`
    - ✓ définit une condition associée au point d'arrêt
  - `dis | ena numéro`
    - ✓ désactive ou réactive le point d'arrêt de numéro donné
  - `watch zone`
    - ✓ définit un point de surveillance sur la zone mémoire donnée
  - `del numéro`
    - ✓ supprime le point d'arrêt ou surveillance de numéro donné

gdb (6)

- Quelques commandes du débogueur (suite)
  - `help`
    - ✓ affiche une aide succincte
  - `quit`
    - ✓ termine l'exécution de gdb

Agenda

- Chapitre 5: Débogage
  - Introduction
  - `printf()`
  - `gdb`
  - Gestion de la Mémoire
    - ✓ Allocation Dynamique
    - ✓ Erreurs avec l'Allocation Dynamique
    - ✓ `valgrind`

Alloc. Dynamique

- Il est fréquent de ne pas connaître à l'avance (i.e., à la compilation) la quantité de mémoire nécessaire pour une donnée
  - données lues au clavier, dans un fichier, un flux réseau, ...
- Nécessité de pouvoir réserver dynamiquement de la mémoire
  - utilisation du tas (heap)

Alloc. Dynamique (2)

- La fonction `malloc()` permet de réserver un certain nombre d'octets sur le tas
- On passe en paramètre le nombre d'octets souhaités
  - utiliser l'opérateur **`sizeof`**`()`
- `malloc()` retourne
  - un pointeur sur la zone mémoire allouée
  - `NULL` en cas d'erreur

```
int n;
scanf("%d", &n);
int *tab = malloc(n * sizeof(int));

for(int i=0; i<n; i++)
  scanf("%d", &tab[i]);
```

Alloc. Dynamique (3)

- Il est important de toujours tester la valeur de retour de `malloc()`
  - permet de détecter les problèmes
- Test avec traitement d'erreur dans le cadre du déroulement du programme
  - on utilise pas `assert()` pour cela

```
int n;
int *tab = malloc(n * sizeof(int));
if(tab==NULL)
  //Gestion de l'erreur
```


Alloc. Dynamique (4)

- La fonction `free()` permet de libérer une zone mémoire allouée avec `malloc()`

```
free(tab);
```

- Bonne pratique
  - ne pas utiliser `free()` avec les zones non allouées par `malloc()`
  - ne pas appeler `free()` plusieurs fois sur la même zone
- Pour éviter les fuites mémoires (*memory leak*), il est préférable d'appeler `free()` dès que la zone n'est plus utilisée

Erreurs

- L'allocation dynamique est un mécanisme fragile
  - blocs libres et occupés chainés par des pointeurs situés avant et après chaque bloc
    - ✓ facile d'écrire en dehors des bornes
    - ✓ *heap smashing*
  - pas de vérification de cohérence
- Les erreurs sont parfois détectées bien après l'instruction qui les cause
  - segmentation fault dans un `malloc()` parce que le `free()` précédent s'est fait sur un bloc corrompu

Erreurs (2)

- Il existe des outils et des bibliothèques destinés à détecter les erreurs d'accès à la mémoire
  - bibliothèques de gestion de mémoire instrumentées pour détecter les écritures en dehors des bornes
    - ✓ compilation spécifique avec ces bibliothèques
    - ✓ compilation spécifique avec des variables d'environnement
      - `MALLOC_CHECK_` sous Linux
  - outils de vérification dynamique de chaque accès mémoire lors de l'exécution
    - ✓ compilation particulière
      - `valgrind`

valgrind

- Banc de tests d'exécution de programmes
- Permet de détecter les erreurs d'exécution et/ou de réaliser un profilage du code
- Basé sur un émulateur de langage machine
  - interprétation pas à pas de chaque instruction du programme binaire
  - réalisation d'un certain nombre de vérifications
- Inconvénients
  - lent car émulation de chaque instruction
  - rechercher des jeux de test les plus petits possibles

valgrind (2)

- `valgrind` travaille par application d'outils
  - fonctions de test prédéfinies
- Outils disponibles
  - `memcheck`
    - ✓ vérification mémoire poussée
      - utilisation de mémoire non initialisée, accès hors blocs mémoires alloués, accès à des zones déjà libérées, ...
  - `addrcheck`
    - ✓ version allégée de `memcheck`
      - pas de test d'accès aux zones mémoires non initialisées

valgrind (3)

- Outils disponibles (suite)
  - `cachegrind`
    - ✓ simulateur de comportement de la hiérarchie de caches
      - ‣ analyse poussée des performances
  - `massif`
    - ✓ analyse de l'occupation du tas
  - `helgrind`
    - ✓ analyse du comportement des programmes multi-threads
      - ‣ détection de zones mémoire non protégées et accédées par plusieurs tâches
- Possibilité de créer ses propres outils

valgrind (4)

```
#include <stdlib.h>

int main(){
  int *x = malloc(10 * sizeof(int));
  x[10] = 0;
}//fin programme
```

- Compilation obligatoire avec les options `-g` et `-O0`

```
$> gcc -g -O0 brol.c -o brol
```

valgrind (5)

- Lors de l'exécution sous `valgrind`
  - des messages d'erreurs sont générés à la volée
    - ✓ pour chaque erreur potentielle détectée
  - à la fin, messages d'erreurs concernant les fuites mémoire

valgrind (6)

```
$>valgrind --leak-check=full ./brol

==31769== Using Valgrind-3.8.1 and LibVEX; rerun with -h for
copyright info
==31769== Command: ./brol
==31769==
==31769== Invalid write of size 4
==31769==    at 0x100000F22: main (brol.c:5)
==31769==  Address 0x1000040e8 is 0 bytes after a block of size
40 alloc'd
==31769==    at 0xC713: malloc (vg_replace_malloc.c:274)
==31769==    by 0x100000F19: main (brol.c:4)
...
==31769== 40 bytes in 1 blocks are definitely lost in loss
record 2 of 9
==31769==    at 0xC713: malloc (vg_replace_malloc.c:274)
==31769==    by 0x100000F19: main (brol.c:4)
```