Programmation « orientée système »
LANGAGE C – DEBUGGING & PROFILING

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- apprendre à gérer les erreurs dans vos programmes : trouver et corriger vos propres erreurs

  ☞ utilisation d'un dévermineur

- optimiser la rapidité de vos codes

Jean-Cédric Chappelier

Erreurs en programmation

Il existe plusieurs types d'erreurs :

1. erreurs de syntaxe :
   le programme est mal écrit et le compilateur ne comprend pas ce qui est écrit.

   Erreurs relativement faciles à trouver : le compilateur signale le problème, indiquant souvent l'endroit de l'erreur.
2. erreurs d'implémentation : la syntaxe du programme est correcte (il compile), mais ce que fait le programme est erroné
   (par exemple une division par zéro se produit, ou une variable n'a pas été initialisée correctement).

   Ces erreurs ne se détectent qu'à l'exécution du programme, soit par un arrêt prématuré (p.ex. division par zéro), soit par des résultats erronés (p.ex. mauvaise initialisation).

Erreurs en programmation

Il existe plusieurs types d'erreurs :

③ erreurs d'algorithme : l'algorithme implémenté ne fait pas ce que l'on croit (ce qu'il devrait)
C'est assez proche du cas précédent, mais ici c'est plus la méthode globale qui est erronée, plutôt qu'une étourderie ou un manque de précision dans une des étapes du codage de l'algorithme.

Il existe pour ce type d'erreurs des tests formels permettant de trouver les erreurs : validation et vérification formelle d'algorithmes.

④ erreurs de conceptions : ici c'est carrément l'approche du problème qui est erronée, souvent en raison d'hypothèses trop fortes ou non explicitées.
Elles relèvent du domaine de l'ingénierie informatique (le « génie logiciel »).

Jean-Cédric Chappelier

Erreurs en programmation

Il existe plusieurs types d'erreurs :

① erreurs de syntaxe
② erreurs d'implémentation
③ erreurs d'algorithme
④ erreurs de conceptions

Nous nous intéressons dans ce cours aux erreurs d'implémentation (②), voire d'algorithme (③), mais d'un point de vue pratique :
c'est-à-dire mise en œuvre de **procédures de déverminage**.

On s'intéresse donc à trouver vos erreurs **lors de l'exécution**.

Jean-Cédric Chappelier

Dévermineur

L'utilisation d'un « dévermineur » (« debugger » en anglais) permet d'ausculter en détails l'exécution d'un programme, et en particulier

- localiser les erreurs
- exécuter un programme pas à pas
- suivre la valeur de certaines variables

gdb

① Pour pouvoir déboguer un programme, il faut le **compiler avec l'option** `-g`
Cela indique au compilateur de rajouter des informations supplémentaires dans le programme, utiles au dévermineur.

```
gcc -g -o monprogramme monprogramme.c
```

② Ensuite il faut lancer le **dévermineur**. Il en existe de nombreux, souvent intégrés dans les IDE. On peut aussi le lancer directement à la ligne de commandes (`gdb`) ou utiliser une GUI spécique comme `ddd` ou `Kdbg` :

```
gdb monprogramme
ddd monprogramme
```

En mode texte/terminal (`gdb`), on peut visualiser le code source à l'aide de (*une fois entrée dans* `gdb`)) :

```
layout next
```

③ Il faut ensuite exécuter le programme à corriger/étudier dans le dévermineur.
Cela se fait à l'aide de la commande `run` dans `gdb`/`ddd` :
`run` ou `run` *`arguments`*

Jean-Cédric Chappelier

gdb (2)

④ On peut décider de suspendre l'exécution du programme à des endroits précis en y plaçant des breakpoints (points d'arrêt)

ou :

```
break 123
break demander_nombre
```

⑤ Une fois le programme stoppé à un point d'arrêt, on peut continuer à l'exécuter *pas à pas*

- soit avec la commande `next` qui exécute les pas de programme au même niveau que le point d'arrêt (ne « descend » pas dans les fonctions appelées)
- soit avec la commande `step` qui exécute les pas élémentaires du programme et donc entre dans les fonctions appelées

ou alors *en continu* jusqu'au prochain breakpoint en utilisant `cont`.

Exemple next/step

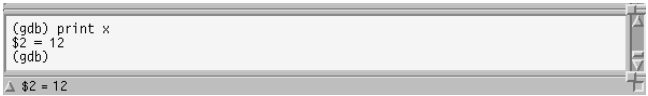
```
...

z = f(a);
y = g(b);

...
```

```
double f(int nb)
{
    double x = 0.0;
    for (int i = 0; i <= nb; ++i) {
        ...
```

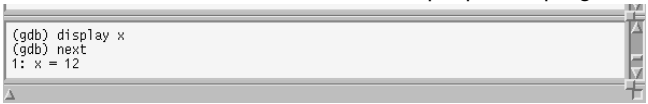
gdb (3)

⑥ On peut regarder le contenu d'une variable

- soit en mettant la souris dessus (`ddd`)
- soit à l'aide de la commande `print`
  qui affiche la valeur de la variable à ce moment là



- soit à l'aide de la commande `display`.
  La valeur de la variable est alors affichée à chaque pas de programme.



Pour plus de détails sur l'utilisation pratique de `gdb`/`ddd`, voir l'exercice correspondant dans la série d'exercices.

Jean-Cédric Chappelier

Déverminage

Pour utiliser un programme de déverminage, compiler avec l'option `-g`

```
gcc -g -o monprogramme monprogramme.cc
```

Lancer le dévermineur : `gdb monprogramme` ou `ddd monprogramme`

Démarrer mon programme dans `gdb`/`ddd` : `run` ou `run` *`arguments`*

Suspendre l'exécution du programme à des endroits précis : utiliser le bouton « breakpoints » 

ou la commande `break`

Exécuter pas à pas : `next` ou `step`

Regarder le contenu d'une variable :

- soit en mettant la souris dessus
- soit `print` *`nom_variable`*
- soit `display` *`nom_variable`*
  La valeur de la variable est alors affichée à chaque pas de programme.

Jean-Cédric Chappelier

Plan

- debuging
- optimiser la rapidité de vos codes

Optimisation

On va maintenant s'intéresser à rendre les programmes un peu plus efficaces, mais...

...**AVANT TOUT** ce qu'il **ne faut pas** faire :

- penser à optimiser avant de concevoir !
  - ☞ Il faut avant tout bien concevoir ses programmes (de façon **claire**) et se focaliser sur le choix des *bons algorithmes*
- optimiser au détriment de la modularité
  - ☞ **jamais** de copier/coller (même si utiliser une fonction « ralentit » le code !)

  La modularité et la maintenabilité (lisibilité) de votre code seront toujours plus importantes que d'essayer de gagner quelques secondes.

Jean-Cédric Chappelier

(Mauvaise) Optimisation (suite)

- ne pas chercher à optimiser à la place du compilateur
  - ☞ se concentrer sur les aspects généraux non triviaux pour les compilateurs :
  - optimisations mathématiques (partage de calcul)
  - éviter les répétitions de calculs, mais préférer stocker des résultats intermédiaires en mémoire
  - éviter les accès mémoires inutiles (laissez le compilateur utiliser des registres)
  - optimisations liées aux informations dynamiques
    (sur lesquelles le compilateur ne peut strictement rien savoir !)

Éviter les répétitions de calculs

Exemple classique des boucles imbriquées...

...explicites :

```
for (size_t i = 0; i < n; ++i)
  for (size_t int j = 0; j < m; ++j)
     a[i][j] = 5 * i + j;
```

...ou implicites (`strlen` fait une boucle sur les caractères de la chaîne) :

```
for (size_t i = 0; i < strlen(s); ++i)
```

Préférer :

```
for (size_t i = 0; i < n; ++i) {
  const size_t temp = 5 * i;
  for (size_t j = 0; j < m; ++j)
     a[i][j] = temp + j;
}
```

et :

```
size_t const temp = strlen(s);
for (size_t i = 0; i < temp; ++i)
```

Éviter les répétitions (2)

Autre exemple classique :

```
for (int i = 0; i < k - 1; ++i)
```

ou encore pire :

```
for (int i = 0; i <= k-1; ++i)
```

Préférez :

```
int const up = k-1;
for (int i = 0; i < up; ++i)
```

et pour le second :

```
for (int i = 0; i < k; ++i)
```

Éviter les répétitions (3)

On peut aussi parfois écrire la boucle dans l'autre sens (si cela s'y prête)

```
for (int i = 0; i <= EXPR; ++i)
```

pourrait peut être s'effectuer aussi bien par

```
for (int i = EXPR; i >= 0; --i)
```

où ici *EXPR* n'est évalué qu'une fois (lors de l'initialisation)

Mais faites bien attention aux *conditions aux limites* ! !

Par exemple

```
for (size_t i = strlen(s)-1; i >= 0; --i)
```

n'est **pas** équivalent à

```
for (size_t i = 0; i < strlen(s); ++i)
```

☞ Pourquoi ?

Jean-Cédric Chappelier

Éviter les opérations coûteuses

Exemples classiques :

préférez `x*x` à `pow(x, 2.0)` ; ( `i >> 2` plutôt que `i/4`)

L'exemple précédent :

```
for (size_t i = 0; i < n; ++i) {
  const size_t temp = 5*i;
  for (size_t j = 0; j < m; ++j)
     a[i][j] = temp+j;
}
```

serait encore plus efficace comme ceci :

```
for (size_t i = 0, temp = 0; i < n; ++i) {
  for (size_t j = 0; j < m; ++j)
     a[i][j] = temp+j;
  temp += 5;
}
```

☞ remplace une multiplication par une addition

Jean-Cédric Chappelier

Optimiser les expressions mathématiques

① choisir une solution algébriquement plus simple, lorsqu'elle existe :

```
int s = n * (k + (n-1) >> 1);
```

au lieu de :

```
int s = 0;
    for (int i = k; i < k+n; ++i)
       s += i;
```

	affectations $(\mathcal{O}(\log(n)))$	additions $(\mathcal{O}(\log(n)))$	multiplications $(\mathcal{O}(\log(n)\log^{(2)}(n)\log^{(3)}(n)))$	décalages $(\mathcal{O}(\log(n)))$
solution 1	1	2	1	1
solution 2	$2+n$	$3n$	0	0

Jean-Cédric Chappelier

Optimiser les expressions mathématiques

② factorisez les calculs (surtout les plus coûteux)

```
y = log(3*x+2) + x + 2;
... // sans modification de x
z = x + 4*b + 3 + log(3*x+2);
... // sans modification de x
t = u * v - x + 4 * r - log(3*x+2) - 12;
```

sera plus efficace sous la forme :

```
const double temp = log(3*x+2) + x + 2;
y = temp;
... // sans modification de x
z = temp + 4*b + 1 ;
... // sans modification de x
t = u * v + 4 * r - 10 - temp;
```

Laissez le compilateurs utiliser des registres

Les accès explicites à la mémoire peuvent être coûteux.
Préférez des variables locales :

```
void somme(double* resultat, size_t taille, double a_sommer[])
{
  *resultat = 0.0;
  for (size_t i = 0; i < taille; ++i)
    *resultat += a_sommer[i];
}
```

sera plus optimal sous la forme :

```
void somme(double* resultat, size_t taille, double a_sommer[])
{
  double temp = 0.0;
  for (size_t i = 0; i < taille; ++i)
    temp += a_sommer[i];
  *resultat = temp;
}
```


Jean-Cédric Chappelier

Optimisation : Résumé

NE PAS y penser dans un premier temps.
Se concentrer avant tout sur la conception, la modularité et la lisibilité.

Ensuite, dans un second temps, en préservant non seulement la tâche effectuée (bien s'en assurer !), mais aussi la modularité et la lisibilité, cherchez à

- supprimer les répétitions de calculs inutiles
- localiser les opérations (variables locales)
- optimiser les aspects algébriques

Pour optimiser son programme, on peut également chercher « expérimentalement » (mais ça ne doit pas empêcher de réfléchir avant !) les endroits les plus critiques lors de l'exécution
☞ outils de « profiling »

Profiling

Le « *profiling* » est un moyen de déterminer le temps consommé par chacune des parties de votre programme.

Pour réaliser ces mesures de manière automatique, on peut ajouter une option de compilation `-pg` :

```
gcc -pg programme.c -o programme
```

L'exécution de `programme` produit alors un fichier `gmon.out`
L'utilitaire `gprof` utilise ce fichier et `programme` pour donner des statistiques sur le déroulement du programme :

```
gprof programme
```

Exemple :

```
gcc -pg fibonacci.c -o fibonacci
fibonacci
```

(... s'exécute ...)

```
gprof fibonacci
```

`man gprof` pour plus de détails

Profiling

pourcentage du temps

Temps propre

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 89.51     6.70     6.70        1     6.70     6.70  Fibonacci
  8.02     7.29     0.60        1     0.60     0.60  FibonacciIteratif
  2.47     7.48     0.18        1     0.18     0.18  demander_nombre
  0.00     7.48     0.00        1     0.00     0.00  _GLOBAL__I_main
  0.00     7.48     0.00        1     0.00     0.00  __static_initialization_and_destruction_0
```

Temps total

```
...

index %time    self  children    called     name
-----------------------------------------------
                             331160280           Fibonacci [2]
                6.70    0.00       1/1           main [1]
[2]     89.5    6.70    0.00       1+331160280 Fibonacci [2]
                             331160280             Fibonacci [2]
-----------------------------------------------
                0.60    0.00       1/1           main [1]
[3]      8.0    0.60    0.00       1         FibonacciIteratif [3]
```

Profiling : pour aller plus loin

Une fois comprises les bases, il peut être utile d'utiliser aussi les outils suivants :

- `callgrind` est une extension de `valgrind` qui permet d'analyser le graphe d'appel des fonctions

  ```
  valgrind -tool=callgrind ./mon_pgm
  ```

  `kcachegrind` permet de visualuser les information de `callgrind`
  (mais n'est plus maintenu depuis 2013) ;

- `gcov` : analyse de couverture (quelles lignes sont exécutées combien de fois) ;
  `lconv` permet de mieux visualiser les résultats de `gcov` ;

- sur Linux : `perf` est une suite d'outils d'analyse de performance ;

- `gprof2dot` (`https://github.com/jrfonseca/gprof2dot`) permet de convertir les informations de divers profilers (dont `Linux perf`, `gprof`, `callgrind`)