

Bonnes pratiques, débogage et optimisation

On va voir :

Quelques bonnes et mauvaises pratiques de programmation

Outils de débogage :

- ▲ gdb
- ▲ valgrind

Outils de "profilage" :

- ▲ gprof
- ▲ gcov

Options utiles de gcc

Les "bugs"

Des bugs (cachés ou non) peuvent avoir de conséquences fâcheuses :

- ▶ plantages (aléatoires), pertes de données
- ▶ "exploitations" : porte d'entrée aux problèmes de sécurité

Lorsqu'un "bug" arrive :

- ▶ c'est (généralement) votre faute !

S'il un programme s'exécute sans "bug" :

- ▶ cela n'implique pas que vous avez bien programmé !
- ▶ les bugs peuvent être "non déterministes" ("Heisenbug" ...)



Les bugs dans un code

Mieux vaut prévenir que guérir :

- ▶ adopter de bonnes pratiques de programmation
- ▶ tester régulièrement son code
- ▶ détecter les problèmes le plus tôt possible dans le processus de programmation

Mais quand il faut guérir :

- ▶ utilisation d'outils de débogage



Bonnes pratiques

Servent à éviter la confusion, et améliorer la compréhension entre les différents programmeurs. Donc en conséquence, à limiter le risque d'erreurs.

- ▶ commenter le code
- ▶ avoir indentation correcte
- ▶ utiliser des noms de variables/fonction explicites
- ▶ "garder le code simple" (KIS) :
- ▶ préférer des fonctions courtes
- ▶ éviter la redondance de code
- ▶ lors de la première version préférez un algorithme simple (et plus lent) à un algorithme complexe (et plus rapide)



Bonnes pratiques

Pour les projets conséquents, ou à plusieurs :

- ▶ code "modulaire"
- ▶ documentez vos fonctions
- ▶ respectez une convention de nommage
- ▶ utilisez un utilitaire de versionnage

Note : on peut faire un code correct sans ces pratiques, mais c'est périlleux (e.g : ioccc)



Pratiques mauvaises/dangereuses/interdites

Ne pas initialiser les variables

- ▶ l'erreur pourra passer inaperçue, car souvent elle sera initialisée à 0 la première fois, mais après ce sera plus aléatoire...

Ne pas tester les codes retours

- ▶ lire les manuels des fonctions que vous utilisez

L'utilisation de fonctions réputées dangereuses

- ▶ sprintf(), strcpy(), strcat(), vsprintf(), gets() ne vérifient pas si il y a assez de place
 - ▶ fonctions non ré-entrant dans un code multithread
- Ne pas désallouer/fermer ce qui ne sert plus (mémoire, descripteurs de fichiers...)

Note : avec ces pratiques, un code ne sera pas "correct".



Tester

En cas de projet conséquent, il faut régulièrement :

- ▶ tester si le code compile
 - ▶ tester s'il donne les résultats attendus
- Séparer le processus de développement en petites parties. Par exemple, on peut dégager deux processus indépendants :
- ▶ le "refactoring" : on ne change pas les fonctionnalités, on ne fait que réorganiser/clarifier/simplifier/optimiser le code.
 - ▶ l'ajout de fonctionnalités.

Ne pas les faire en même temps, et vérifier après chaque étape.



Tests

- ▶ "Test unitaire" : vérifier le bon fonctionnement d'une partie (unité, module) du logiciel.
- ▶ Créez et intégrez une batterie de tests qui teste automatiquement chaque partie les unes après les autres
- ▶ Les tests doivent être "méchants" : testez sur beaucoup d'entrées, et essayez de couvrir tous les cas



Programmation par contrats

Assertion : expression qui doit être évaluée à vraie à un moment donné

Dans le paradigme "Programmation par contrats", 3 types d'assertions :

- ▶ pré-conditions
- ▶ post-conditions
- ▶ invariants

En C/C++ : on peut tester une assertion avec assert



assert

```
assert(expr)
  ▲ dans assert.h
  ▲ se déactive avec l'option -DNDEBUG
  ▲ attention : pas pour tester les codes retours dans un vrai
    programme !

#define mon_assert(expr) {\
  if (!(expr)) { \
    fprintf(stderr, "assert %s failed %s:%d\n", \
            STRING(expr), FILE, \
            __ASSERT_FUNCTION, __LINE__); \
    abort(); \
  } \
}
```



Outils d'aide au développement

Utilisation d'environnement de développement

Outils de gestion de version

- ▲ subversion (SVN), Git, mercurial...
- ▲ possible de faire des branches stable / développement
- ▲ il est possible de reprendre une ancienne version pour tracker l'apparition d'un bug.



Tracker les bugs : outils à disposition

voir les choses "suspectes", même sur un code qui semble marcher correctement :

- ▶ `gcc -Wall`
 - ▶ un code devrait toujours compiler sans warning !
 - ▶ on peut raffiner les tests de warning, ex : `"-Wno-sign-compare"`
 - ▶ on peut (des)activer un test dans le code :


```
#pragma GCC diagnostic ignored "-Wsign-compare"
...
#pragma GCC diagnostic warning "-Wsign-compare"
```




Tracker les bugs : outils à disposition

- ▶ `gcc -fstack-protector-all`
- ▶ en C++ : `g++ -D GLIBCXX_DEBUG` pour des tests sur les conteneurs de la STL
- ▶ Valgrind : passer un coup de valgrind de temps en temps, même sur un code sans suspicion, ne fait pas mal...

En cas de bug avéré :

- ▶ compiler avec les infos de débogage : `gcc -g`
 - ▶ attention, dès fois cela fait des choses bizarres avec `"-Ox"`
- ▶ `gdb`
- ▶ `valgrind`



Valgrind

valgrind détecte (des fois) :

- ▶ les variables non initialisées
- ▶ les fuites mémoires
- ▶ les dépassements de tableaux

Il y a (rarement) des faux positifs (dans certaines bibliothèques). Mais en général : si il y a un warning, c'est qu'un truc n'est pas bon dans votre code. Càd, un truc à corriger au plus tôt !

Principe (idée) : exécute le code dans un processeur virtuel.
Exécution 10 à 30x plus lente...



Valgrind : utilisation

- ▶ Compiler avec l'option -g (rajout des symboles de débogage dans le fichier binaire)
 - ▶ Exécuter la commande, précédée de valgrind
 - ▶ Les avertissements seront envoyés sur la sortie erreur :
- ```
==21068== Invalid write of size 8
==21068== at 0x400A6F: add(char const*, elm_t*) (vector.cpp:28)
==21068== by 0x400AF7: main (vector.cpp:39)
==21068== Address 0x5a81c88 is 8 bytes inside a block of
size 16 free'd
==21068== at 0x4C2A30B: operator delete(void*)
vg_replace_malloc.c:575
...
```



## Autres outils de la suite Valgrind

`valgrind -tool=<toolname>`

- ▶ memcheck (par défaut) : rapporte les problèmes d'accès mémoire (non alloué, non initialisé, inaccessible), les fuites mémoires, double-free...

- ▶ massif : profilage de tas

- ▶ cachegrind, callgrind : profilage de cache

- ▶ helgrind , DRD : déboguer programmes multithreadés



## gdb

gdb est le déboguer par défaut de la suite GNU Permet, entre autres :

- ▶ d'exécuter jusqu'à un ou des points d'arrêts
- ▶ d'exécuter pas à pas
- ▶ de regarder l'état des variables, pile, registres....

Comment ça marche (idée, sur x86) :

- ▶ gdb a accès à tout l'espace mémoire du processus qu'il débogue
- ▶ quand on met un point d'arrêt sur une ligne, gdb remplace la première instruction machine correspondante à la ligne par une instruction "INT 3" (opcode : 0xCC)
- ▶ l'exécution de "INT 3" provoque une interruption, qui rend la main à gdb (qui peut remettre l'instruction initiale à la place de INT 3)



## `gdb` : lancement

Compiler le programme à déboguer avec l'option "-g"

- attention, ça fait souvent des choses bizarres avec -Ox

Lancer le gdb :

- `gdb ./executable`
- si arguments : `gdb --args ./executable arguments...`

Dans l'interface de gdb :

- `run` : lance l'exécution

Attacher un programme en cours d'exécution :

- lancer gdb
- `attach pid`

- `gdb -tui` : avec une interface textuelle



## `gdb` : lister le code

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list fct` : lister le code depuis le début de la fonction *fct*
- `list fichier:fct` : lister le code depuis le début de la fonction *fct* dans le fichier *fichier*
- `list +, list -` : avancer (reculer) dans le fichier
- `step` : avance d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante



## `gdb` : points d'arrêts

Point d'arrêt (`breakpoint`) : arrête le processus quand il atteint une ligne

- ▶ `break fct` : rajoute un point d'arrêt au début de la fonction *fct*
  - ▶ `break n` : rajoute un point d'arrêt à la ligne *n*
  - ▶ `break fichier:ligne` ou `break fichier:fct`
- ▶ possibilités de point d'arrêts conditionnels
- ▶ `info breakpoints` : lister les points d'arrêts

Retirer un point d'arrêt :

- ▶ `clear fct`
- ▶ `delete nb`



## `gdb` : variables et "watchpoints"

- ▶ `print var` : affiche la valeur d'une variable (ou expression)
- ▶ `display var` : affiche à chaque pas

Modifier une variable :

- ▶ `set var = x`
- ▶ `watch var`

*watchpoint* : arrête le programme quand une variable est modifiée



## `gdb` : pile et threads

- ▶ `backtrace` : affiche la pile d'appels
  - ▶ `up / down` : monter ou descendre dans les *frames*
  - ▶ `frame num` : changer de *frame*
- Multithread :
- ▶ `info threads` : liste les threads
  - ▶ `thread num` : change le thread courant



## `gdb` : registres et assembleur

- ▶ `info registers` : affiche les registres
- ▶ `layout asm` : affiche le code assembleur
- ▶ `layout src` : affiche le code source

Il existe des interfaces graphiques à `gdb`...



## gdb : raccourcis

- ▶ entrée : précédente commande
- ▶ r : run
- ▶ l : list
- ▶ c : continue
- ▶ s : step
- ▶ n : next
- ▶ bt : backtrace
- ▶ i : info
- ▶ b : breakpoints
- ▶ i b : info breakpoints
- ▶ ...



## Déboguer : aller plus loin

Il est possible d'intégrer des outils de débogage dans son code.

Exemple : backtrace

```
void sigsegv (int)
{
 void *bt[DEBUGMEM_MAXBT];
 int sizebt = backtrace (bt, DEBUGMEM_MAXBT);
 char **strings = backtrace_symbols (bt, sizebt);
 for (int i=0; i<sizebt; i++)
 fprintf (stderr, "%s\n", strings [i]);
 exit(1);
}

...
signal(SIGSEGV, (sighandler_t) sigsegv);
signal(SIGBUS, (sighandler_t) sigsegv);
...
```



## Optimiser son code

Là, on suppose que notre code marche bien. On veut l'optimiser :

Options de gcc :

- ▶ -Ox
  - ▶ -O0 : pas d'optimisation
  - ▶ -O1 : optimisations modérées
  - ▶ -O2 : pleines optimisations
  - ▶ -O3 : comme -O2, en encore plus agressif
  - ▶ -Os : optimisation en mémoire (taille de d'exécutable)
- ▶ -march=native : compile pour le processeur de la machine
- ▶ -ffastmath : active certaines optimisations sur les flottants (ne respecte plus la norme IEEE 754)
- ...

Optimisations de gcc : passer des variables en registres, rendre des fonctions *inline*, dérécuriver, déboucler, réorganisation des instructions...



À savoir :

- ▶ les malloc/free (new/delete), c'est plutôt lent. Préférer d'autres méthodes d'allocations en cas de grosse demande
- ▶ les realloc peuvent être très lents (déplacement en mémoire)
- ▶ les appels systèmes, c'est très lent

En C++ : Certains conteneurs sont plus lents que d'autres :

- ▶ utiliser le conteneur le plus adapté
  - ▶ array, c'est bcp plus rapide que vector
  - ▶ remplir un vecteur avec un push\_back, c'est lent
- Certaines choses rendent l'inlinisation impossible :
- ▶ les accesseurs séparés dans un autre fichier source
  - ▶ les fonctions membres virtual...



## Outils de profilage

Profilage : analyse dynamique de l'exécution d'un code.

Outils :

- ▶ gprof
  - ▶ gcov
  - ▶ C++ : g++ -D\_GLIBCXX\_PROFILE
- [https://gcc.gnu.org/onlinedocs/libstdc++/manual/profile\\_mode.html](https://gcc.gnu.org/onlinedocs/libstdc++/manual/profile_mode.html)



## gprof

▶ Calcule le temps passé dans chaque fonction, et le graphe d'appel.

▶ Le compilateur rajoute du code, qui va générer un fichier gmon.out contenant les informations de profilage.

▶ Inconvénient : le code ne doit pas être optimisé (-Ox) , sinon cela peut faire des choses bizarres

- ▶ cela ne dit pas vraiment le temps passé dans chaque fonction quand ce sera optimisé, mais cela donne néanmoins de bonnes approximations

▶ Note : le code devient notablement plus lent



## `gprof` : utilisation

- ▶ Compiler avec l'option `-pg`
  - ▶ attention, souvent cela fait des choses bizarres avec `-Ox` !
- ▶ Lancer le programme normalement. Il va générer le fichier `gmon.out`
- ▶ Une fois terminé, lancer `gprof executable`
  - ▶ L'affichage est en 2 parties
    - ▶ Le temps passé dans chaque fonction
    - ▶ le graphe d'appel



## `gcov`

- ▶ Teste la "couverture". Pour chaque ligne, affiche le nombre de fois que la ligne a été exécutée
- ▶ Compiler avec `-fprofile-arcs -ftest-coverage`
- ▶ Exécuter le code.
- ▶ Exécuter `gcov fichier_source`
- ▶ Il va générer un fichier texte `fichier_source.gcov`



Suite :

- ▲ Threads et synchronisation
- ▲ Réseau...

