Langage C avancé : Séance 8

Outils de recherche et de correction d'erreurs

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Erreurs dans un code C (1)

Un code source peut contenir des erreurs qui provoquent des messages d'avertissement ou erreurs de compilation, des arrêts prématurés de l'exécution du code binaire, ou une exécution complète mais qui fournit des résultats faux.

On a déjà vu que le compilateur propose des options qui affichent des messages d'avertissement sur du code dont la syntaxe est correcte mais qui risquent des arrêts prématurés, de produire d'autres résultats que ceux attendus: -Wall –Wextra …
Ces options repèrent certaines des erreurs les plus habituelles mais ne garantissent pas l'absence d'erreurs.

Il est, de toute façon, fortement conseillé d'utiliser ces options.

Erreurs et messages d'avertissement de compilation dans un code C (1)

- Les erreurs de compilation sont dues à des erreurs de syntaxe C, des fonctions ou des variables globales utilisées mais non définies, des fonctions ou des variables globales définies plusieurs fois, etc.
  Ce sont les plus faciles à détecter et à corriger: le compilateur refuse de générer un code binaire en affichant des messages qui donnent en général assez d'information pour corriger ce type d'erreurs.

Les messages d'avertissements et d'erreurs sont différents suivant les compilateurs, la version du langage C, etc.
Pour les codes destinés à être diffusés, il peut être intéressant de compiler le code source avec plusieurs compilateurs et sur plusieurs systèmes différents.

Erreurs et messages d'avertissement de compilation dans un code C (2)

- Dans le sous-répertoire erreurs_compilation/err1, compiler en tapant :

```
gcc main.c –o code
```

Corriger les erreurs éventuelles jusqu'à ce que le code compile.
Exécutez le code binaire.

- Recompiler avec les options de vérification, en tapant :

```
gcc –Wall –Wextra main.c –o code
```

Modifier le code source pour tenir compte des messages d'avertissement.
Exécutez le code binaire

Erreurs et messages d'avertissement de compilation dans un code C (3)

- Premier enseignement : il peut y avoir plusieurs façons de corriger les erreurs

Exemple:

```
double f(int n) {
  …
}

int main() {
  double x = f(10, 3.4);
}
```

?

```
double f(int n, double x) {
  …
}

int main() {
  double x = f(10, 3.4);
}
```

```
double f(int n) {
  …
}

int main() {
  double x = f(10);
}
```

Les 2 modifications donnent une syntaxe correctes.
Pour choisir, il faut comprendre ce qu'est censé faire le code

Erreurs et messages d'avertissement de compilation dans un code C (4)

- Deuxième enseignement : avoir une syntaxe correcte ne suffit pas, le code peut se compiler (éventuellement avec des messages d’avertissement) mais s’exécuter en fournissant des résultats faux ou en s’arrêtant sur une erreur.

Donc, il faut utiliser toutes les options vérifications à la compilation et tenir compte des messages d’avertissement.

On verra souvent qu’il faut aussi faire des vérifications à l’exécution (voir plus loin).

Erreurs dans l'appel des fonctions ou dans l'utilisation des variables globales

- Dans le sous-répertoire erreurs_compilation/err2, compiler en tapant :

```
gcc main.c f.c –o code
```

Le compilateur refuse de créer un binaire exécutable parce que la variable globale m est définie 2 fois. Modifier f.c en ajoutant « extern » devant la définition de m dans f.c:

```
double m;
```

```
extern double m;
```

Le message du compilateur signifie que la variable globale m ne peut pas être définie séparément dans plusieurs fichiers, il faut la définir dans un seul ficher et utiliser « extern » dans les autres.

Erreurs dans l'appel des fonctions ou l'utilisation des variables globales

- Recompiler et exécuter le code en tapant ./code
  Le code s'exécute mais affiche des valeurs incohérentes, sur ma machine:
  > f: x = 0 m = 2.47033e-323
  > f: r = 0
  > main: f(10) = 9

- Le problème vient de ce fait le compilateur quand on tape:

```
gcc main.c f.c –o code
```

  - Compilation séparée de main.c (fichier caché main.o)
  - Compilation séparée de f.c (fichier caché f.o)
  - Rassemblement des fichiers main.o et f.o dans l'exécutable code

Le compilateur vérifie que toutes les fonctions et les variables globales utilisées sont définies une et une seule fois, mais pas que les utilisations des fonctions ou variables globales sont conformes à leur définition.

Exploration des fichiers binaires

Pour voir ce qui se passe, recompiler les fichiers en plusieurs étapes:

```
gcc –c main.c
gcc -c f.c
gcc main.o f.o –o code
```

Et utiliser la commande objdump qui montre le contenu des fichiers compilés (l'exécutable final ou les fichiers objets .o):

```
objdump –t <nom de fichier>
```

On peut aussi utiliser la commande nm qui affiche les mêmes informations sous une forme différente:

```
nm <nom de fichier>
```

Exploration des fichiers binaires

```
objdump -t main.o
```

```
…
0000000000000000 g     O .bss   0000000000000004 m
0000000000000000 g     F .text  0000000000000049 main
0000000000000000         *UND*  0000000000000000 f
0000000000000000         *UND*  0000000000000000 printf
…
```

```
objdump -t f.o
```

```
…
0000000000000000 g     F .text  0000000000000074 f
0000000000000000         *UND*  0000000000000000 m
0000000000000000         *UND*  0000000000000000 printf
…
```

UND : signifie que le symbole (fonction, variable globale) est utilisé dans le fichier .o mais sa définition est à l'extérieur.

Exploration des fichiers binaires

```
objdump -t code
```

(extraits)

```
0000000000001192 g     F .text  0000000000000074    f
0000000000000000       F *UND*  0000000000000000    printf@GLIBC_2.2.5
0000000000004014 g     O .bss   0000000000000004    m
0000000000001149 g     F .text  0000000000000049    main
```

Tous les symboles sont bien définis (f, m, main), c.-à-d. le compilateur les a trouvés dans un des .o.
Seule la fonction printf est non définie dans le binaire mais l'exécutable utilise la fonction printf du système.
Utilisez la commande ldd pour voir ce que l'exécutable utilise dans la machine:

```
ldd code
```

```
linux-vdso.so.1 (0x00007fff391f7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff50749f000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff5076cc000)
```

Exploration des fichiers binaires

L’information qui nous intéresse ici est que les symboles (fonctions, variables globales) sont enregistrés dans les .o avec leur nom, mais pas avec leur type (variables globales), pas avec leur type de retour ou leurs types et nombre de paramètres (fonctions).

Ce qui fait que le compilateur ne voit pas qu’il y des différences entre les symboles de même nom mais dans des .o différents.

Ceci est particulier au C !!

Correction de l'erreur

Pour corriger ce type d'erreur, on procède généralement comme suit, pour chaque variable globale:

- On met la déclaration de la variable globale dans un fichier d'entête (se terminant souvent par .h) avec le mot clef « extern »
- On remplace la déclaration de cette variable globale dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la variable globale), on définit la variable globale (sans le mot clef « extern ») – C accepte de déclarer une variable globale 2 fois (avec et sans « extern »)

On peut réutiliser le même fichier d'entête pour plusieurs variables globales.

Correction de l'erreur

Pour chaque fonction:

- On met le prototype de la fonction dans un fichier d'entête (se terminant souvent par .h) avec ou sans le mot clef « extern » (qui est ajouté par défaut)
- On remplace le prototype de cette fonction dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la fonction), on définit la fonction (avec le code interne de la fonction)

On peut réutiliser le même fichier d'entête pour les prototypes de plusieurs fonctions.

Cela permet au compilateur C de contrôler la cohérence des fonctions et variables globales.

Voir le répertoire symboles_globaux pour un exemple incomplet à compléter.

Un code avec une syntaxe correcte qui compile peut contenir des erreurs

Quand on exécute un code binaire C (produit de la compilation d'un code source), il peut se produire des erreurs à l'exécution ou des résultats faux.
Dans ce cas, il y a plusieurs moyens pour les repérer et les corriger:

- Réviser le code source
  En général, valable pour un petit code et si on a une bonne idée du problème
- Utiliser printf pour afficher des variables au cours du calcul
- Utiliser la fonction assert pour contrôler des conditions au cours du calcul
- Utiliser un outil de débug
  Il s'agit outils qui permettent d'exécuter un code instruction par instruction et d'afficher des variables du code pendant l'exécution.
- Utiliser un outil de test de la mémoire dynamique
  Si on soupçonne un problème avec l'utilisation de la mémoire dynamique.

Afficher des valeurs intermédiaires pendant le calcul

Afficher des valeurs intermédiaires avec printf

Quand on soupçonne que des variables prennent des valeurs non correctes pendant le calcul, on peut afficher ces valeurs en plusieurs endroits du code source.

Avantages:

- Ce moyen est souvent celui qui reste disponible, si aucun autre outil de recherche d'erreur ne l'est.
- Si on a une idée précise de la variable et de l'endroit qui pose problème, c'est un moyen de contrôle simple.

Inconvénients:

- C'est un processus qui peut-être long, surtout si un code source est de grande taille et qu'on n'a pas d'idée précise où se situe le problème.
- Ajouter des instructions au code (même des printf), le modifie et peut changer son comportement.

  Dans certains cas, un code peut s'arrêter sur une erreur sans printf et s'exécuter jusqu'à la fin avec des printf

Exemple

Dans le répertoire racine, un fichier C (main.c) contient une fonction qui calcule la racine carrée d'un nombre par une méthode de type Newton.

Le programme principal demande à l'utilisateur de rentrer un nombre et affiche la racine carrée calculée de ce nombre.

Compiler et exécuter plusieurs fois ce code en rentrant des valeurs quelconques positives, nulles et négatives (2, 3, 0, -1, -6).
Quel est le comportement du code ?

Afficher les valeurs successives de r dans la fonction racine.

Utilisation de la fonction assert

Tester des conditions dans le code source avec assert

C propose une fonction « assert » qui teste une condition.
Si la condition est vraie, l'exécution continue, sinon l'exécution s'arrête en affichant un message d'erreur.
Ne pas oublier d'inclure le fichier assert.h.
Voir le répertoire tests/racine2.

Il est possible de désactiver les asserts en compilant le code avec l'option NDEBUG (no debug):

```
gcc -DNDebug main.c -o racine
```

Dans ce cas, les instructions asserts seront ignorées.

Si la variable qui peut poser problème, est une variable initialisée par l'utilisateur, à la place de assert (ou en plus de assert), il faut plutôt ajouter un test explicite:

```
scanf("%lg", &x);
if (x < 0) { printf("entrer un nombre positif ou nul"); exit(-1); }
```

Conseils avec les instructions « assert »

En général, on procède comme suit :

- pendant la phase d'écriture du code, on compile en mode debug et on active les asserts et les tests sur le données d'entrée

```
gcc –g main.c –o racine
```

- quand le code a été testé et avant de le fournir aux utilisateurs, on compile en mode optimisé et on désactive les assert mais on garde les tests sur les données d'entrée

```
gcc –O2 –DNDebug main.c –o racine
```

La fonction assert est plutôt destinée aux développeurs de code et pas aux utilisateurs.

Préconditions avec les instructions « assert »

Dans chaque fonction du code source, on ajoute des "assert" au début de la fonction pour contrôler la validité des paramètres de cette fonction (si cela à un sens)

Exemple: produit de matrices A, B avec résultat la matrice C (le type matrice utilisé est celui vu à la séance précédente)

```
void produit(const Matrice *A, const Matrice *B, Matrice *C) {

  assert(A.n() == C.n());
  assert(B.m() == C.m());
  assert(A.m() == B.n());

  for (int i=0; i<C.n(); i++)
    for (int j=0; j<C.m(); j++)
      for (int k=0; k<A.m(); k++)
        C.c[i*C.m() + j] += A.c[i*A.m() + k]*B.c[k*B.m() + j];
}
```

Postconditions avec les instructions « assert »

Dans chaque fonction du code source qui produit des résultats, on ajoute des "assert" en fin de la fonction pour contrôler la validité des résultats de cette fonction (si cela à un sens)

Exemple: calcul par une formule de Taylor de sin(x)

```
double sinus(double x) {
  double s = x – x*x*x/6 + …;
  assert(s>-1 && s<1);
}
```

L'utilisation de pré- et post-conditions est parfois appelée "programmation par contrat" ou "programmation défensive"

Utilisation d’un outil de debug : exemple avec gdb

Utilisation d'un outil de debug

Les outils logiciels de debug permettent :

- d'exécuter instruction par instruction un code binaire,
- d'afficher la valeur d'une variable du code à un instant
- d'exécuter le code jusqu'à une instruction donnée
- d'exécuter le code tant qu'une condition est vraie
- d'exécuter le code tant qu'une variable ne change pas de valeur
- …

C'est donc un outil très puissant pour examiner ce qui se passe pendant l'exécution d'un code.

Il existe plusieurs outils de debug (on dit aussi debugger): gdb (linux), lldb (linux), codeview (intégré à visual studio dans windows), …

Gdb est souvent à la base des outils de débug disponibles dans les environnements de développement (codeblocks, visual code, eclipse,…)

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un débugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de l'une des commandes

```
gcc main.c –o code
gcc –O2 main.c –o code
gcc –g –O2 main.c –o code
```

Il faut utiliser ou mieux:

```
gcc –g main.c –o code
gcc –g –O0 main.c –o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake –S srcDir –B buildDir –DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un débugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de l'une des commandes

```
gcc main.c –o code
gcc –O2 main.c –o code
gcc –g –O2 main.c –o code
```

Il faut utiliser ou mieux:

```
gcc –g main.c –o code
gcc –g –O0 main.c –o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake –S srcDir –B buildDir –DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb : utilisation (1)

Pour exécuter le code binaire « code » sous le contrôle de gdb, il faut taper:

```
gdb code
```

Pour arrêter gdb, taper ctrl-d (et répondre y pour confirmer).

Si le message suivant (ou similaire) s'affiche :

```
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n])
```

Quitter gdb et tapez la commande (dans le terminal):

```
echo "set debuginfod enabled on" >> $HOME/.gdbinit
```

Debugger gdb : utilisation (2)

Quand on lance gdb (avec le nom du code binaire exécutable), gdb se met en attente d'une commande.

Les principales commandes sont:

- ctrl-x a: permet de voir le code source en même temps qu'on tape des commandes de gdb (ctrl-x a à nouveau pour sortir)
- run : lance l'exécution (jusqu'à la fin ou une erreur d'exécution ou un point d'arrêt (voir ci-dessous)
- quit : sort de gdb (confirmer en tapant y)
- b xxx.c:n (break) : met un point d'arrêt au début d'une instruction à la ligne nunéro n du fichier xxx.c

  Gdb affiche un numéro de point d'arrêt qui permettra de le supprimer, de le désactiver, de le réactiver, …

- p v : affiche la valeur de la variable v

Debugger gdb : utilisation (3)

- del k : supprime le point d'arrêt numéro k
- disable k : si le point d'arrêt numéro k est actif, le désactiver
- enable k : si le point d'arrêt numéro k est inactif, le réactiver
- cont (continue) : si l'exécution est arrêtée à un point d'arrêt, continuer jusqu'à la fin ou au point d'arrêt suivant
- n (next) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, « sauter au-dessus de l'appel)
- s (step) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, gdb se met au début du code de la fonction)
- w x (watch) : où x est le nom d'une variable simple (pas une structure), si on tape cont ou run à la suite de la commande watch, l'exécution se poursuit tant que la variable x ne change pas de valeur

Debugger gdb : utilisation (4)

- `b xxx.c:n if « condition »` : pose un point d'arrêt conditionnel à la ligne n du fichier xxx.c (l'exécution s'arrêtera au point d'arrêt seulement si la condition est vraie)

  Exemple: extrait du fichier « toto.c »:

```
21: for (i=0; i<1000; i++)
22:     a[i] = 2*b[i]-b[i-1]-b[i+1];
```

Si on tape : `b toto.c:22 if (i==500)`
l'exécution s'arrêtera à l'itération numéro 501 de la boucle sur i

- `bt` : affiche la pile d'appels (la liste des lignes par où est passé le code pour arriver à la ligne courante)

Debugger gdb : utilisation (5)

Il existe (beaucoup) d'autres commandes, voir, par exemple, le manuel de gdb (https://sourceware.org/gdb/current/onlinedocs/gdb.html)

Pour se rappeler les commandes de gdb, le mieux est de s'exercer sur des codes:

- Recompiler le code dans le répertoire sqrt en mode debug et utiliser gdb pour exécuter le code, mettre des points d'arrêts, afficher des variables, etc.

Comprendre les erreurs d’utilisation de la mémoire dynamique : l’outil valgrind

Erreurs d’exécution : utilisation de la mémoire dynamique

Un grand nombre d’erreurs d’exécution sont dues à une mauvaise utilisation de la mémoire dynamique:

- Utilisation d’une mémoire dynamique non (ou pas encore) réservée
- Utilisation d’un pointeur en dehors de la mémoire dynamique réservée
- Utilisation d’une mémoire dynamique après qu’elle ait été rendue au système.

L’outil valgrind

valgrind est un logiciel conçu à l’origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l’environnement KDE dans linux.
Il s’est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- valgrind crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, valgrind peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l’efficacité de l'outil, est que le temps d’exécution est (considérablement) allongé.

L’outil valgrind

Valgrind est un logiciel conçu à l’origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l’environnement KDE dans linux.
Il s’est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- valgrind crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, valgrind peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l’efficacité des vérifications, est que le temps d’exécution est (considérablement) allongé.

Utilisation de valgrind

Utiliser valgrind est simple, il suffit de taper valgrind suivi du nom du code binaire:

```
valgrind code
```

valgrind affiche à l'écran, en plus des affichages normaux du code, tout ce qu'il a noté dans l'utilisation de la mémoire dynamique.

Compte-tenu du (potentiellement) grand nombre de message affiché, il est préférable de rediriger les affichages écran dans un fichier qu'on examinera à la fin de l'exécution :

```
valgrind code >& log
```

(crée un fichier de nom log qui contient les messages de valgrind et les affichages du code)

Exemple d'utilisation de valgrind

Se mettre dans le répertoire memoire_dynamique/exemple2,
compiler en mode debug et exécuter la commande

```
valgrind ./code >& log
```

On examinera en séance le fichier log avec les messages valgrind.

Comprendre les erreurs d’utilisation de la mémoire dynamique : l’outil sanitizer

L’outil sanitizer

Plusieurs outils ont été développés chez google, puis intégrés dans les compilateurs (d’abord clang, puis gcc et icx (le compilateur Intel))

C’est un fichier binaire (une librairie) à ajouter à la compilation. Il contient des fonctions qui interceptent toutes les utilisations de la mémoire dynamique.
Pour compiler le code, il faut ajouter une option –fsanitize=… et, éventuellement, une librairie (exemples pour gcc et clang) :

```
gcc –g –fsanitize=address main.c –static–libasan –o code

clang –fsanitize=memory –fno–omit–frame–pointer –g main.c –o code
```

Exemple d'utilisation des sanitizers

Le répertoire memoire_dynamique/exemple2 contient plusieurs versions avec différents types d'erreurs possibles.

Un Makefile compile chaque fichier séparément pour générer des codes, avec et sans les options sanitizer.

- Utiliser valgrind avec les exécutables compilés sans les options sanitizer
- Exécuter directement, les exécutables générés avec les options sanitizer

Interpréter les sorties.

La librairie sanitizer est peut-être un futur outil standard.
Cet outil n'est pas encore aussi efficace que valgrind mais est beaucoup plus rapide.
Par contre, il faut recompiler tout les fichiers sources.

Dans la pratique

Aucun des outils présenté ici n'est capable de trouver toutes les erreurs possibles.

Il faut donc, en général, les combiner

Cas des vecteurs de taille fixe – débordement de tableau

Compiler et exécuter le code contenu dans le répertoire « vecteurs-statiques ».

Expliquer l'erreur dans ce qui est affiché.

Dans la séance, on utilisera gdb pour repérer la cause de l'erreur.

Ce genre d'erreur (débordement de tableau) est souvent utilisé par les virus informatiques qui parviennent à écraser des zones mémoires normalement inaccessibles dans un code et à modifier le comportement du code.
Il est donc important de corriger ce type d'erreur.

Malheureusement, les outils de debug arrivent difficilement à trouver la cause de l'erreur (dans l'exemple, on a repéré la zone écrasée en affichant toutes les variables en plusieurs points du code).