Travaux Pratiques d'algorithmique n°9

Cours d'informatique de Deuxième Année

—Deug Sciences S3-MI—

Compiler, déboguer et optimiser vos programmes

Le but de cette feuille est l'apprentissage des techniques nécessaires à une bonne utilisation du débogueur et autres outils de développement.

1 La compilation avec `gcc`

Voici quelques options de `gcc` (dont certaines ne vous sont pas étrangères) :

- `-Wall` : affiche tous les messages de recommandations (*warning*s) ;
- `-ansi` : utilise la norme *C ANSI* ;
- `-o` <*nom*> : permet de donner un nom au fichier obtenu par compilation (par défaut : `a.out`) ;
- `-lm` : permet de *link*er la librairie mathématique au programme ;
- `-v` : demande au compilateur d'afficher ses actions (pour *verbose* ou *mode bavard*) ;
- `-g` : nécessaire pour utiliser le débogueur ;
- `-profile` (`-pg` sur d'autres systèmes) : nécessaire pour utiliser `gprof` ;
- `-O` : optimisation classique (ne pas employer en cas d'utilisation du débogueur) ;
- `-O3` : optimisation maximale (ne pas employer en cas d'utilisation du débogueur) ;

La compilation d'un source *C* s'effectue selon plusieurs étapes. En fait, lorsque vous lancez la commande `gcc`, un certain nombre de programmes sont appelés les uns à la suite des autres, chacun correspondant à une phase de la compilation. Pour vous en rendre compte, utilisez l'option `-v` pour compiler votre programme.

Pour information, les trois étapes principales sont les suivantes :

- Le pré-processing.

  `gcc` fait appel au programme `cpp` (appelé le pré-processeur) dont le but est de :

  - substituer les macro-définitions par leur valeur (`#define`) ;

- ◦ substituer les `#include` par le contenu de fichiers ainsi désignés ;
  - ◦ permettre la compilation conditionnelle avec `#if(n)def VARIABLE #endif`.

  Pour vous rendre compte de ce que fait le pré-processeur, compilez un petit programme avec `cpp` au moyen de la ligne suivante : `gcc -E essai.c -o essai.cpp` et jetez un œil au fichier ainsi généré : il est très long et votre code se trouve à la fin.

- La compilation en elle-même.

  `gcc` fait appel au programme `cc1` qui permet de passer d'un code écrit en *C* à un fichier objet en `.o`. C'est durant cette étape que la vérification de la syntaxe, de l'existence des types et des prototypes des fonctions est effectuée (c'est ici que les messages d'erreur du type `essai.c:10` sont générés).

- L'édition de liens (linkage).

  `gcc` fait appel au programme `collect2` (ou `ld` sur d'autres systèmes) dont le but est de lier tous les fichiers `.o` et toutes les librairies en un fichier binaire exécutable (c'est par exemple ici qu'intervient l'option `-lm`). C'est à cette étape que l'on vérifie si toutes les fonctions appelées et toutes les variables utilisées existent bien (c'est ici que les messages d'erreur du type `/tmp/cca00453.o` sont générés).

2 Le débogueur `xxgdb`

Un débogueur est un outil très utile qui permet de comprendre pourquoi votre programme “plante” ou ne se comporte pas comme vous le désirez. Un tel logiciel vous permet de suivre l'exécution de votre programme pas à pas. Vous pouvez ainsi à tout moment vérifier la valeur d'une variable, d'une structure, visualiser la pile des appels de fonctions ... Le débogueur dont vous disposez sous Linux est `xxgdb`. Pour l'utiliser vous devez compiler votre programme avec l'option `-g` comme ceci : `gcc -Wall -ansi -g -o essai essai.c`

Ensuite, tapez la commande : `xxgdb essai`

Vous verrez alors apparaître plusieurs fenêtres. Celle en bas à gauche contient votre programme écrit en *C*. Celle en haut à gauche permet d'entrer les commandes de contrôle du débogueur. Une autre sur la droite comporte des boutons permettant une gestion à la souris. Dans ce TP nous étudierons les commandes utilisables dans la fenêtre du haut, mais `xxgdb` permet de mettre en œuvre certaines commandes courantes au moyen de la souris.

Pour obtenir de l'aide, tapez `help` dans la fenêtre du haut ou bien `help` *<rubrique>* ou “ *<Control-h> i* ” sous *emacs*.

2.1 Les points d'arrêt

Vous avez la possibilité d'indiquer au débogueur des lignes précises, ou les fonctions qui vous intéressent plus particulièrement. Pour cela, on utilise des points d'arrêt (*break points*). Ainsi, le programme s'exécutera normalement jusqu'au moment où un point d'arrêt sera rencontré. Le débogueur vous rendra alors la main pour vous permettre d'exécuter votre programme en mode pas à pas (par exemple pour contrôler la valeur des variables). Les points d'arrêt sont numérotés incrémentalement au fur et à mesure de leur création.

Voici quelques commandes relatives aux points d'arrêt :

- `b` (<*fichier*>) <*numéro de ligne*> (break) insère un point d'arrêt à la ligne indiquée dans le fichier (si aucun nom de fichier n'est spécifié, le fichier considéré est celui actuellement affiché dans la fenêtre du bas) ;
- `b` <*nom de fonction*> insère un point d'arrêt à la ligne où commence la fonction ;
- `disable` <*numéro*> suspend le point d'arrêt de numéro spécifié (`enable` pour le réactiver) ;
- `delete` (<*numéro*>) supprime le point d'arrêt de numéro spécifié (si pas de numéro spécifié : supprime tous les points d'arrêt) ;
- `help breakpoints` donne la liste de toutes les commandes concernant les points d'arrêt.

2.2 Exécuter le programme

Voici les commandes qui permettent de contrôler l'exécution du programme :

- `r` (<*arguments*>) (run) exécute le programme jusqu'au premier point d'arrêt (les arguments sont ceux que l'on mettrait sur la ligne de commande dans le cas d'une exécution normale du programme) ;
- `s` (step) exécute uniquement l'instruction suivante ;
- `n` (next) exécute uniquement l'instruction suivante (sans entrer dans les appels de fonction ; une fonction étant alors vue comme une seule instruction) ;
- `c` (continue) le programme reprend et s'arrête au prochain point d'arrêt rencontré ;
- `p` (<*expression*>) (print) permet de visualiser la valeur de l'expression (en l'absence d'expression, la dernière expression fournie à la commande `p` est affichée) ; les syntaxes "`p *t`", "`p t->champs`", etc sont valides si `t` est un pointeur ;
- `x` <*adresse*> (examine) examine la mémoire à l'adresse spécifiée ; indique par exemple si on a le droit d'accéder à cette adresse ; des messages comme "`Cannot access memory at address`" ou "`_dl_runtime_resolve+8`" indiquent que l'on n'a pas accès à cet endroit de la mémoire ; <*adresse*> peut avoir les formats suivants : `p` (si `p` est un pointeur), `t`, `&t[12]`, `t+12` (si `t` est un tableau) ...
- `q` (quit) permet de quitter `xxgdb`.

2.3 La pile des appels de fonction

Lorsque votre programme "plante" dans une fonction `x`, appelée par une fonction `y`, elle-même appelée par une fonction `z`, on a souvent besoin de remonter dans la pile d'exécution (ou pile des appels de fonction) pour vérifier le contenu des variables à chaque niveau.

Pour cela, vous pouvez utiliser les commandes suivantes :

- `where` permet de visualiser le contenu de la pile des appels de fonctions ;
- `up` et `down` permettent respectivement de remonter dans le code de la fonction appelante, et de descendre dans le code de la fonction appelée ; ainsi à chaque niveau, il est possible de visualiser la valeur des variables locales à chaque fonction appelée.

2.4 Comment exploiter un fichier `core`

Dans le cas d'un programme interactif (c'est-à-dire faisant très souvent appel aux fonctions d'entrée comme `scanf` ou `getchar`), il est parfois difficile d'utiliser un débogueur. Si un programme de ce type (ou d'un autre type) venait à "planter", il existe une méthode pour connaître précisément et a posteriori l'endroit où l'erreur a eu lieu.

Pour cela, il faut taper dans le terminal (c'est-à-dire la `xterm`) d'où vous allez lancer le programme la ligne suivante : `ulimit -c unlimited` Cette commande permet aux programmes qui y seront ensuite lancés, de créer des fichiers `core` sans limite de taille ; par défaut la limite est à 0, c'est-à-dire que vous ne pouvez pas en créer. Il n'est possible de modifier cette limite qu'une seule fois ; en cas d'erreur, il vous faut donc ouvrir un autre terminal. Un `core` est un fichier qui est la copie de la mémoire du programme au moment où il "plante".

Ensuite la marche à suivre est celle-ci :

- Compilez le programme avec l'option `-g` : `gcc -Wall -ansi -g -o essai essai.c`
- Ensuite exécutez le : `./essai` ; s'il plante, il va générer un `core`.
- Dans ce cas, lancez le débogueur au moyen de la ligne suivante : `xxgdb essai core`
  Ce dernier analyse le fichier `core` et affiche l'endroit exact où le programme a planté ; il est alors possible d'afficher la valeur des variables, la pile des appels de fonction ...

Vous pouvez faire un essai avec les instructions suivantes : `int *p=NULL; *p = 2;`
Pensez à effacer tous vos fichiers `core` ; il sont inutiles après débogage et prennent souvent beaucoup de place.

2.5 Exemple de session

Voici un exemple d'utilisation du débogueur `xxgdb` sur le fichier `~lhullier/s3mi/essai.c` ici recopié :

```
/*  1 */  #include <stdio.h>
/*  2 */  #include <stdlib.h>
```

```
/*  3 */  int Bbb( int n );
/*  4 */  int Aaa( int n )
/*  5 */  {
/*  6 */    int b = n + 3;
/*  7 */    b *= (n+2);
/*  8 */    if( n > 0 )
/*  9 */      return b+Bbb(n-1);
/* 10 */    else
/* 11 */      return b;
/* 12 */  }
/* 13 */  int Bbb( int n )
/* 14 */  {
/* 15 */    int b = n;
/* 16 */    b = b + 4;
/* 17 */    if( n > 0 )
/* 18 */      return b+Aaa(n-1);
/* 19 */    else
/* 20 */      return b;
/* 21 */  }
/* 22 */  int main( void )
/* 23 */  {
/* 24 */    int i;
/* 25 */    char t[10];
/* 26 */    for( i=3; i>=0; i-- )
/* 27 */      printf("%d\n",Aaa(i));
/* 28 */    t[2000] = 3;
/* 29 */    return 0;
/* 30 */  }
```

Avant de faire quoi que ce soit, écrivez sur papier la structure des appels de fonctions (quelles sont les fonctions qui sont appelées et avec quels paramètres ...) au moins pour `i` valant 3 dans le `main`.

Pour compiler et lancer le débogueur, les lignes de commandes sont les suivantes :

```
gcc -Wall -ansi -g -o essai essai.c
xxgdb essai
```

Ensuite, tapez les commandes suivantes dans la fenêtre du haut :

1.	`b main`	pose d'un point d'arrêt sur la fonction `main`
2.	`b 15`	pose d'un point d'arrêt en ligne 15
3.	`r`	lancement du programme ; l'exécution s'arrête sur le point d'arrêt du `main`
4.	`s`	on avance d'une instruction en mode pas à pas
5.	*[ENTER]*	taper *[ENTER]* permet d'effectuer à nouveau la commande précédente ; ici on avance d'une instruction et on entre donc dans la fonction `Aaa`
6.	`c`	on continue le programme ; l'exécution s'arrête sur le point d'arrêt de la ligne 15 (fonction `Bbb`)
7.	`where`	on regarde la pile d'exécution
8.	`p n`	on affiche la valeur de la variable `n`
9.	`c`	on continue le programme ; l'exécution s'arrête à nouveau sur le point d'arrêt de la ligne 15
10.	`where`	on regarde la pile d'exécution
11.	`p n`	on affiche la valeur de la variable `n`
12.	`up`	on remonte dans la pile d'exécution : on passe donc au niveau de la fonction `Aaa` appelante
13.	`p n`	on affiche la valeur de la variable `n` (celle de la fonction `Aaa`)
14.	`delete`	on supprime tous les points d'arrêt
16.	`c`	on continue le programme ; l'exécution s'arrête à la ligne 28 où le programme plante
17.	`x &t[2000]`	on teste si on peut accéder à l'adresse indiquée
18.	`quit`	on quitte `xxgdb`

Quelles sont les valeurs et informations affichées par les commandes `p` et `x` ? Êtes-vous convaincu que cela correspond effectivement à la réalité ?

3 L'optimiseur de performance `gprof`

Pour optimiser au mieux un programme, il est nécessaire de savoir dans quelles fonctions le programme passe le plus de temps. Les outils utilisés pour cela sont appelés "profiler" ; celui qui est disponible sous Linux est appelé `gprof`. Pour l'utiliser, vous devez compiler votre programme en utilisant l'option `-profile` (`-pg` sur d'autres systèmes), soit :

```
gcc -Wall -ansi -profile -o essai essai.c
```

Ensuite, vous exécutez votre programme normalement. Un fichier `gmon.out` sera alors créé et contiendra des statistiques sur votre programme que `gprof` saura décoder.

Les statistiques données par `gprof` sont nombreuses, celles qui nous intéressent sont celles qui concernent le nombre d'appels de fonctions et le temps passé à les exécuter. Dans le cas où votre programme passerait 90% de son temps à exécuter la même fonction, il faudrait veiller à optimiser cette dernière si vous souhaitez augmenter la vitesse du programme. Vous pouvez aussi comparer le nombre d'appels aux fonctions `malloc` et `free` ; au cas où ces nombres seraient différents, il est certain que votre gestion de la mémoire n'est pas satisfaisante.

Cette méthode d'optimisation est surtout valable pour les programmes non-interactifs, c'est-à-dire ceux faisant peu appel aux fonctions d'entrée comme `scanf`, `getchar`, etc.

Voici la marche à suivre :

```
gcc -Wall -ansi -profile -o essai essai.c
essai
gprof essai | less
```