Initiation à la programmation en C

le débogage

Antoine Miné

Année 2006–2007

Site du cours: `http://www.di.ens.fr/~mine/enseignement/prog2006/`

Le but de cette fiche est de présenter quelques techniques pour vous aider à comprendre et éliminer les erreurs de programmation en C.

Cette fiche ne concerne que les erreurs d'*exécution*. Elle suppose donc que votre programme compile correctement. C'est à dire, n'a aucune erreur de syntaxe, de typage, ou de symbole (variable, fonction) non défini ; ces erreurs étant généralement facile à corriger grâce aux messages explicites du compilateur.

Contenu de la fiche

1 Qu'est-ce qu'un bogue ? **1**
2 Techniques élémentaires **4**
3 Le débogueur GDB **8**
4 Le vérifieur Valgrind **11**
5 Le profileur Gprof **12**

1 Qu'est-ce qu'un bogue ?

Un bogue[1] désigne toute erreur de programmation qui rend un programme partiellement ou totalement inutilisable. Ce terme est très vague. Il regroupe en particulier les cas suivants :

- le programme s'interrompt brutalement en affichant un message tel que `Segmentation fault` ou `Invalid floating-point operation` (voir 1.2),
- le programme ne se termine pas,
- le programme ne calcule pas le résultat escompté,
- le programme permet à un utilisateur malveillant d'espionner un autre utilisateur, de modifier ou de détruire ses données ou de se faire passer pour lui.

Il est généralement admis que tous les programmes contiennent des bogues, même si certains sont plus bogués que d'autres et que la gêne occasionnée varie très fortement d'un bogue à l'autre. Un exemple extrême est celui de l'explosion d'Ariane 5 qui a coûté 500 millions de dollars[2].

1.1 Protection par le système

Les systèmes d'exploitation modernes font leur possible pour limiter les conséquences des erreurs de programmation. Un programme utilisateur n'accède jamais directement aux ressources

[1] Le mot bogue a été choisi pour sa proximité phonétique avec le terme anglais *bug*, lui-même dérivé de l'anecdote suivante : en 1946 à Harvard, un insecte coincé dans les relais électriques d'un ordinateur *Mark II* aurait causé une anomalie de fonctionnement.

[2] Il s'agissait bien d'une erreur de logiciel et non d'un problème matériel. En l'occurrence, le logiciel de guidage s'est désactivé suite à un dépassement de capacité non prévu dans une opération arithmétique. Quand la fusée a commencé à dériver, le logiciel de contrôle a donné l'ordre d'autodestruction.

matérielles, mais seulement par l'intermédiaire du système. Celui-ci effectue de nombreuses vérifications et interdit les opérations dangereuses[3]. Il est donc impossible à votre programme de "planter" le système, d'endommager un composant matériel ou d'effacer un fichier du système, sauf bogue du système[4]. Au pire, le système terminera brutalement le programme (voir 1.2). Au mieux, l'instruction n'aura aucun effet.

Protection mémoire. La mémoire est un exemple important de ressource partagée. Le système alloue une certaine partie de la mémoire à chaque programme et s'assure qu'il n'accède pas aux autres parties de la mémoire. En particulier, un programme ne peut pas lire ou écrire dans la mémoire réservée aux autres programmes, dans la mémoire du système ou celle réservée aux entrées / sorties des périphériques. En cas d'accès invalide, le programme est immédiatement interrompu par le système avec un message du type `Segmentation faut` (voir 1.2). Ceci est très vite arrivé en C si on essaye d'accéder à un tableau en dehors de ses bornes ou si on déréférence un pointeur construit n'importe comment.

Protection des fichiers. Sous Unix, il est également impossible de lire, modifier ou effacer un fichier appartenant à un autre utilisateur sans son consentement. Si vous tentez une telle opération, la fonction de la bibliothèque standard échouera et renverra un code d'erreur mais votre programme ne sera pas interrompu. Notez que dès que vous lancez un programme, vous lui donnez implicitement des droits sur vos fichiers (que le programme lancé vous appartienne ou non !). Il peut donc détruire vos données, les envoyer à un site pirate, etc.

Trous de sécurité. Il peut arriver qu'un bogue ne se déclenche que dans des conditions très particulières, jamais réunies en fonctionnement normal. Le bogue semble alors assez inoffensif. Toutefois, un pirate informatique doué sera capable d'exploiter ce bogue pour effectuer des opérations qui devraient lui être interdites. Ceci peut aller du simple plantage du système à la prise de contrôle de l'ordinateur visé, en passant par la récupération de données confidentielles. De nombreux problèmes de sécurité informatique ont pour cause des bogues, en particulier dans les serveurs Internet.

Bogues du système. Il arrive qu'un des composants du système (par exemple, un pilote de périphérique) soit lui-même bogué. Les effets peuvent être catastrophiques. En effet, le système s'exécute au niveau de privilège maximum et a un accès non restreint à toutes les ressources du système. Il travaille "sans filet" et personne n'est là pour rattraper ses erreurs. D'où l'importance de mettre régulièrement son système à jour !

1.2 Opérations invalides et opérations bénignes

Certaines instructions en langage machine sont invalides. Elles provoquent des *exceptions*, qui sont des signaux particuliers envoyés au système d'exploitation. Celui-ci répond en "tuant" immédiatement le programme fautif. Un message tel que `Segmentation fault` apparaît alors à l'écran. En fait, ce message ne provient ni du programme, ni du système, mais du shell qui suit de près ce qu'il advient des programmes lancés depuis la ligne de commande et informe l'utilisateur du destin tragique de ceux-ci[5].

Le langage C étant assez bas niveau, il ne se soucie pas trop des dangers liés aux opérations invalides. Il traduira une construction C dangereuse par une une construction dangereuse en langage machine en supposant que le programmeur sait ce qu'il fait.

[3] Cet accès indirect a d'autres applications que la protection des programmes et matériels. Elle permet de *virtualiser* les ressources et permettre à plusieurs programmes d'accéder au clavier, à l'écran, etc. sans se marcher sur les pieds. Elle permet enfin d'*abstraire* les périphériques pour fournir au programme une interface unifiée, indépendante du type de matériel présent.

[4] En réalité, un programme peut facilement, par des moyens détournés, rendre le système inutilisable. Par exemple, en occupant toute la mémoire disponible, en saturant le disque, en accaparant les périphériques, etc.

[5] Un programme lancé en ligne de commande entretient une relation filiale avec le shell qui l'a lancé. On dit en effet que le shell est le *père*, tandis que le programme est le *fils*.

Messages d'erreur. Voici une liste des messages d'erreur ainsi que leur cause probable. Cette liste est très partielle. Elle varie d'une système à l'autre et d'un shell à l'autre :

1. `Segmentation fault` : accès mémoire illégal (interdit par le système). Il s'agit probablement d'un dépassement de tableau ou d'un déréférencement de pointeur invalide.
2. `Bus error` : accès mémoire non aligné. Sur certains microprocesseurs un déréférencement de pointeur sur un entier ou flottant doit obligatoirement se faire sur une adresse multiple de la taille de l'objet déréférencé (c'est le cas sur les processeurs Sparc des Suns mais pas sur les Intels des PCs). L'erreur est donc probablement due à une arithmétique de pointeurs invalide ou à une violation du typage.
3. `Floating point exception` : division (ou modulo `%`) par zéro *dans les entiers*. Contrairement à ce que l'intitulé peut faire croire, les calculs flottants ne provoquent pas cette erreur.
4. `Broken pipe` : écriture dans un tube (*pipe*) qui a été fermé par l'interlocuteur.

Opérations bénignes. Notons d'abord que certaines opérations C à priori non définies ont en fait un comportement parfaitement défini par la norme quoique non intuitif. Par exemple, un dépassement de capacité dans les entiers non signés sur b bits correspondra à un calcul *modulo* 2^b. De même, dans les flottants, tous les dépassements de capacité et les divisions par zéro génèrent des nombres spéciaux ($+\infty$, $-\infty$ ou *Not a number*) mais pas d'exception[6]. De plus, la majeure partie des opérations non définies par la norme ont un comportement déterministe et bénin sur la majorité des ordinateurs. Toutefois, ce comportement dépend du compilateur, des options de compilation et n'est pas toujours documenté. Il ne faut donc jamais le tenir pour acquis ! Un exemple est le dépassement de capacité dans les entiers signés.

Les opérations invalides générant une exception ne couvrent donc qu'une faible proportion des comportements non définis du langage C. Notons que, si ces opérations ne déclenchent pas immédiatement une exception, elles peuvent quand même provoquer des bogues *fonctionnels* : le programme ne calcule pas ce qu'il devrait (par exemple, si sa correction dépend de propriétés de l'arithmétique qui ne sont plus vraies quand un dépassement de capacité se produit).

Localisation des erreurs. Pour déterminer la ligne du programme qui a déclenché une exception, il est utile de lancer le programme sous le débogueur GDB (voir 3). Attention, il arrive souvent que le bogue se trouve en amont et ne provoque une opération invalide que beaucoup plus loin (voir 1.3). Il arrive également que l'opération invalide ait lieu dans une fonction de la bibliothèque standard C et non une fonction utilisateur. La bibliothèque étant bien programmée, une telle erreur est généralement imputable à l'utilisateur de la fonction. Un premier exemple de telle erreur est le passage d'arguments invalides (pointeurs NULL, chaînes non terminées par le caractère \0, etc.). Un deuxième exemple est le non respect du contrat d'utilisation (fermer deux fois un fichier, ignorer un code d'erreur retourné une fonction appelée précédemment, etc.). Dans ces cas, pensez à relire la documentation de la fonction (commande `man`).

1.3 Erreurs non déterministes

Certaines erreurs semblent se manifester de manière aléatoire. De même, certaines erreurs ne semblent se manifestent que si le programme est compilé avec des options d'optimisation. Enfin, certains bogues semblent disparaître alors qu'on instrumente le programme avec des `printf` pour essayer de comprendre leur origine.

En fait, le bogue est toujours là. Son effet est *non déterministe* : il dépend du contexte de manière non spécifiée par le langage et difficilement prévisible. En particulier, il ne se traduit pas forcément par une opération invalide.

[6] Dans la configuration par défaut de la plupart des machines, au moins. Ce comportement peut toutefois être modifié.

Accès mémoires invalides. Ces bogues sont dus en majorité à des accès mémoire invalides. Pour illustrer le caractère non déterministe, nous donnons quelques conséquences possibles d'un accès aléatoire dans la mémoire :
- lire une zone accessible au programme ⇒ aucun effet ;
- lire ou écrire dans une zone protégée par le système ⇒ `Segmentation fault` ;
- lire ou écrire sans respecter les contraintes d'alignement ⇒ `Bus error` ;
- écrire dans une zone réservée au programme mais inutilisée dans la suite des calculs ⇒ aucun effet ;
- écrire dans une zone réservée au programme et utilisée dans la suite du programme ⇒ corruption silencieuse de la mémoire qui peut provoquer des résultats aberrants et même une `Segmentation fault` quand la donnée est utilisée (peut-être beaucoup plus tard).

L'effet du bogue dépend en particulier de l'adresse mémoire affectée à chaque objet par le compilateur. Celle-ci varie quand le programme est modifié (même légèrement) ou compilé avec des options d'optimisation différentes.

Variables non initialisées. Les bogues non déterministes peuvent aussi être dus à des variables non initialisées. On rappelle en effet qu'en C, les variables ne sont pas initialisées par défaut. Avant leur première affectation, leur valeur est aléatoire. Si une telle variable sert d'indice dans un tableau, on retrouve une erreur de mémoire non déterministe. La valeur aléatoire peut également se propager par des affectations en cascade et ne provoquer un bogue que très loin, dans une expression qui ne fait pas intervenir directement de variable non initialisée.

Une première difficulté des bogues non déterministes est d'établir un scénario permettant de les reproduire afin de les étudier (choix d'un jeu de données, options de compilation, type d'ordinateur, etc.). Une deuxième difficulté est de déterminer l'origine réelle du bogue (variable non initialisée, calcul d'indice dans un tableau, etc.) qui n'a parfois qu'un lien ténu avec sa manifestation. Le logiciel Valgrind (présenté en 4) fournit une aide précieuse en détectant un certain nombre de constructions génératrices de bogues non déterministes au moment même où elles sont exécutées.

2 Techniques élémentaires

2.1 Bogues courants

Cette partie décrit quelques bogues courants. Si votre programme ne marche pas, prenez un peu de temps pour lire ce qui suit (tous les conseils ne s'appliquent pas à tous les types de programmes).

Référence non définie. Si vous utilisez une bibliothèque externe de fonctions, n'oubliez pas de compiler avec l'option `-l` correspondante. Pour utiliser les fonctions mathématiques, terminez votre commande par l'option de compilation `-lm`.

Problèmes syntaxiques. Avec l'option `-Wall`, GCC avertit l'utilisateur lorsqu'il rencontre des constructions syntaxiques suspectes. Citons, entre autres :
- utilisation de `=` au lieu de `==` dans un test (conditionnelle, boucle) ;
- erreurs sur les priorités des opérateurs : dans le doute, mettre des parenthèses (en particulier pour `&&`, `||`, `<<`, `>>`) ;
- utilisation de conditionnelles ou boucles imbriquées sans délimiteurs de blocs `{` et `}`.

Attention, tous ces avertissements ne sont qu'indicatifs. Ils peuvent se déclencher sur du code correct ou, au contraire, laisser passer des constructions dangereuses.

L'option `-Wextra`[7] ajoute des avertissements complémentaires, portant sur des constructions considérées comme légèrement "moins suspectes" (c'est à dire, correspondant plus souvent à des

[7] En réalité, GCC connaît de nombreuses options de la forme `-W`*xxx* qui activent des avertissement non reportés par `-Wall` ou `-Wextra`. Voir `man gcc`. Toutefois, celles-ci sont généralement moins utilisées car elles rendent GCC très "bavard", même sur des programmes corrects.

fausses alertes en pratique). Je vous conseille toutefois de compiler votre programme avec les deux options `-Wall -Wextra` et de tenir compte des remarques du compilateur, même si le programme semble marcher.

Variables non initialisées. Pensez à initialiser vos variables avant des les utiliser. Leur valeur initiale n'est pas forcément zéro mais un nombre aléatoire.

Return manquant. Une fonction dont le type de retour n'est pas `void` doit toujours renvoyer une valeur avec `return`. Sinon, la valeur de retour est aléatoire.

Déclarations implicites. Si le type d'une variable ou le prototype d'une fonction (type des arguments et type de retour) n'est pas connu, le type `int` est supposé. Ceci arrive en particulier quand vous utilisez une fonction externe (par exemple, de la bibliothèque C) mais oubliez la directive `#include` correspondante. Si le type réel de n'est pas `int`, cela peut poser de graves problèmes, donc n'oubliez pas le `#include`. Ce problème n'est pas détecté par le compilateur si l'option `-Wall` n'est pas mise !

Arithmétique des ordinateurs. Voici quelques spécificités de l'arithmétique en C qu'il faut connaître pour éviter les bogues :

- les entiers ont une capacité limitée, dépendante du type et de la machine ; sur nos machine les `int` sont dans $[-2147483648; 2147483647]$ et les `unsigned` dans $[0; 4294967296]$ (32 bits) ;
- un dépassement de capacité dans les entiers signés renvoie une valeur non déterminée ;
- un dépassement de capacité dans les entiers non signés renvoie le résultat modulo ;
- une division ou le modulo par zéro dans les entiers provoque une `Floating point exception` ;
- la division entière tronque (arrondi vers 0) ;
- le modulo a `%` b est positif si a est positif, négatif sinon ;
- le type `char` peut être signé ($[-128; 127]$) ou non signé ($[0; 255]$) ; dans le doute, toujours préciser `unsigned char` ou `signed char` ;
- les décalages a `<<` b et a `>>` b sont indéfinis si b est négatif ou plus grand que le nombre de bits du type de a ;
- les conversions flottants vers entiers tronquent (arrondi vers 0) ;
- les calculs dans les flottants sont sujets à arrondi ;
- un dépassement de capacité ou un calcul incorrect dans les flottants génère un des flottants spéciaux : $+\infty$, $-\infty$ ou *NaN* (signifiant *Not a Number*) ; ceux-ci se propagent alors dans les calculs suivants.

Ordre d'évaluation. L'ordre d'évaluation des sous-expressions et des arguments d'une fonction est indéfini. Ceci peut poser un problème si l'évaluation des sous-expressions provoque des *effets de bord*[8]. Par exemple, `x=puts("a")+puts("b");` affichera tantôt `ab`, tantôt `ba` à l'écran. Par contre, `x=y++;` ou `x=--y;` sont biens définis car `y` sera modifié après (respectivement, avant) l'affectation dans `x`. Dans le doute, décomposez les instructions complexes en plusieurs instructions plus simples.

Affectations multiples. Une conséquence du point précédent est qu'une instruction ne doit modifier une variable donnée qu'une fois au plus. Ainsi `x=(y=2)+(y=3);` est indéfini car `y` est modifié deux fois. De même il est interdit d'utiliser et de modifier une même variable dans une instruction (par exemple dans `a[i++]=i;`). Il y a cependant une exception : la nouvelle valeur d'une variable peut dépendre de son ancienne valeur. Ceci rend légal la construction courante `i=i+1;`.

[8] On appelle effet de bord toute modification de l'environnement global : état de la mémoire, de l'écran, des fichiers, etc. Cette modification est généralement irréversible et est observable par d'autres programmes (et l'utilisateur). L'exécution d'un programme est composée de "calculs purs" et d'effets de bord.

Opérateurs logiques. L'ordre d'évaluation du *et* `&&` et du *ou* `||` logiques se fait toujours de gauche à droite. De plus, l'expression de droite n'est évaluée que si le résultat ne peut pas être déduit de la seule valeur de gauche. C'est à dire, a `&&` b n'évaluera pas b si a est faux, et a `||` b n'évaluera pas b si a est vrai. En particulier, si le tableau `t` a `n` cases, le test `i<n && t[i]!=0` est correct tandis que `t[i]!=0 && i<n` peut provoquer un dépassement de tableau.

Opérateur ternaire. a `?` b `:` c commence toujours par évaluer a. Ensuite, il évalue soit b (si a est vrai), soit c (si a est faux), mais pas les deux.

Indices de tableau. Les tableaux sont indicés à partir de 0, pas 1. Si `t` est déclaré par `int t[10];`, les indices valables de `t` vont de 0 à 9. Un accès à un indice strictement négatif ou strictement supérieure à 9 provoquera une erreur non déterministe.

Chaînes de caractères. Toutes les fonctions standard supposent qu'une chaîne de caractères se termine par le caractère \0 (de valeur numérique 0). Si le terminateur \0 n'est pas présent, elles peuvent corrompre la mémoire au-delà de l'espace alloué pour la chaîne ou bien provoquer une `Segmentation fault`. Si possible, préférez les fonctions qui prennent en argument la taille maximale de la chaîne (`fgets`, `snprintf`, `strncat`, etc.). Enfin, lorsque vous allouez de la mémoire pour stoker vos chaînes, prenez garde à compter la place occupée par le terminateur : il faut donc allouer un octet de plus que la taille maximale de la chaîne.

Fonctions dangereuses. Certaines fonctions de la bibliothèque standard sont intrinsèquement dangereuses. Il ne faut jamais les utiliser. Quand c'est le cas, cela sera indiqué dans la page de man correspondante, ainsi que les fonctions qu'il vaut mieux utiliser à la place. Ces vielles fonctions n'existent encore que pour des raisons de compatibilité avec les anciens programmes. Quelques exemples importants : ne jamais utiliser `gets`, `sprintf`, `strcat`, `strcpy` mais respectivement `fgets`, `snprintf`, `strncat` et `strncpy`.

Pointeurs. Ils sont la cause de nombreuses erreurs, en particulier si on utilise la mémoire dynamique (`malloc`, `realloc`, `free`) ou l'arithmétique de pointeurs. Il ne faut pas :
- retourner un pointeur sur une variable locale (l'espace mémoire pointé devient invalide dès que la fonction retourne),
- déréférencer un pointeur NULL (de nombreuses fonctions C standard retournent NULL pour signaler une erreur, il faut donc toujours vérifier leur valeur de retour),
- déréférencer un pointeur non initialisé (il peut pointer n'importe où),
- utiliser l'arithmétique de pointeurs pour dépasser des bornes de la mémoire réservée à une variable ou un bloc de mémoire dynamique,
- utiliser un bloc de mémoire dynamique après l'avoir libéré (`free`) ou redimensionné (`realloc` retourne un nouveau pointeur).

Codes d'erreur. De nombreuses fonctions de la bibliothèque standard renvoient un code d'erreur pour indiquer si l'opération s'est bien déroulée. Celui-ci vaut généralement -1 en cas d'échec (ou NULL pour un pointeur). Citons, entre autres : les opérations de fichiers (`fopen`, `open`, `write`, etc.) et celles qui allouent de la mémoire (`malloc`, `strdup`, etc). Il est important de bien tester le code d'erreur. En cas de problème, mieux vaut en avertir l'utilisateur (`fprintf(strerr,"blha",...);`) et quitter immédiatement (`exit(1);`) que continuer et risquer une erreur plus grave ou un comportement non déterministe. La nature exacte de l'erreur est indiqué par la valeur de la variable standard `errno`.

2.2 Instrumenter avec `printf`

Un méthode classique pour vérifier le bon comportement d'un programme est d'ajouter des *messages de débogage* qui décrivent pas à pas son déroulement. Pour cela, des appels judicieux à `printf` suffisent. On pourra afficher périodiquement la valeur de quelques variables importantes. On pourra également afficher un message quand certaines fonctions sont appelées, dans certaines branches conditionnelles ou à chaque itération de certaines boucles.

Dans le cas où le programme est interactif ou si il affiche déjà des informations à l'écran, il est préférable de stocker ces messages dans un fichier annexe, dit fichier *de log*. On commencera donc le programme par `FILE* log = fopen("log","w");`. On utilisera ensuite `fprintf(log,"blha\n",...);` pour laisser un message. Pensez à appeler la fonction `fflush(log);` après chaque message. Ceci assure que le message est immédiatement consigné dans le fichier et ne reste pas retenu un temps indéfini dans la mémoire tampon de la bibliothèque standard. En effet, si vous souhaitez examiner le contenu du fichier de log au fur et à mesure qu'il se remplit, il est important qu'il soit toujours à jour. De plus, si le programme plante, le contenu du tampon est perdu et avec, tous les messages entre le dernier `fflush` et le plantage (ce sont souvent les plus importants).

Une fois votre programme corrigé, il est souhaitable de désactiver ces messages. D'abord pour éviter de submerger l'utilisateur de textes et fichiers inutiles. Ensuite, pour gagner du temps (l'affichage avec `printf` et `fprintf` est en effet très lent). Il serait toutefois dommage de supprimer totalement les lignes `printf` car les messages de débogage auront peut-être besoin d'être rétablis par la suite. Une solution classique consiste à placer ces lignes entre les directives `#ifdef DEBUG` et `#endif`. Considérez l'exemple suivant :

```
#include <stdio.h>

/* #define DEBUG */

int main(int argc, const char** argv)
{
#ifdef DEBUG
  printf("debug: début de main, %i arguments\n",argc);
#endif
  ...
#ifdef DEBUG
  printf("debug: fin normale de main\n");
#endif
  return 0;
}
```

Les deux commandes `printf("debug: ...");` ne seront pas exécutées tant que le symbole `DEBUG` n'est pas défini (en fait, elles ne seront même pas compilées). Si, par contre, vous décommentez la ligne `#define DEBUG`, alors les messages de débogage seront activés. En fait, il n'est même pas utile de modifier le programme, il suffit de le recompiler avec l'option `-DDEBUG` (équivalente à un `#define DEBUG` en tête de programme).

2.3 Assertions

Une *assertion* est une expression booléenne qui est censée être toujours vraie en un point de programme donné. Mais peut-être que l'assertion est fausse à cause d'un bogue dans votre programme. La fonction `assert`, définie dans l'en-tête `assert.h`, permet de vérifier une assertion. Si la condition a est fausse pendant l'évaluation de `assert(`a`)` ; alors le programme sera interrompu avec un message indiquant, entre autres, le numéro de ligne de la commande `assert` qui a échoué. Il est donc utile d'émailler son programme d'`assert`s qui vérifient périodiquement la bonne santé des données.

Une utilisation particulièrement intéressante d'`assert` est la vérification des arguments d'une fonction. Supposez, par exemple, que votre fonction `f` prenne en argument un pointeur `p` qui ne doit jamais être NULL. Alors, plutôt que de faire entièrement confiance à l'appelant de la fonction, vous pouvez ajouter la ligne `assert(p);` en tête de `f`, histoire d'être vraiment sûr.

Les assertions peuvent être automatiquement désactivées par la directive `#define NDEBUG` placée avant le `#include <assert.h>` ou, plus simplement, par recompilation avec l'option `-DNDEBUG`. Il s'agit donc d'une technique duale de celle décrite en 2.2. Le gain de rapidité ne se justifie généralement pas et il vaut mieux laisser les assertions actives.

2.4 *Wrappers*

Il peut être pénible d'avoir à tester le code de retour de chaque fonction de la bibliothèque standard, en particulier si on les utilise beaucoup. Une solution élégante pour éviter ce travail consiste à définir une fois pour toutes des fonctions qui "enrobent" celles de la bibliothèque C (en anglais *wrappers*). Elles s'occupent de tester leur code d'erreur et, en cas de problème, affichent un message clair et quittent proprement. Voici des exemples pour `fopen` et `malloc` :

```
#include <stdio.h>  /* pour fopen */
#include <stdlib.h> /* pour malloc et exit */
#include <errno.h>  /* pour errno (codes d'erreurs) */
#include <string.h> /* pour strerror (messages d'erreur) */

FILE* my_fopen(const char* path, const char* mode)
{
  FILE* f = fopen(path,mode);
  if (!f) {
    fprintf(stderr,"N'arrive pas à ouvrir %s car %s\n",path,strerror(errno));
    exit(1);
  }
  return f;
}

void* my_malloc(size_t size)
{
  void* p = malloc(size);
  if (!p) {
    fprintf(stderr,"N'arrive pas à allouer %lu octets\n",(unsigned long)size);
    exit(1);
  }
  return p;
}
```

Vous pouvez maintenant utiliser `my_fopen` et `my_malloc` sans vous préoccuper des codes d'erreur. Notez qu'avec cette méthode, le programme quitte immédiatement en cas de problème[9]. Elle est donc bien adaptée à la gestion propre des erreurs fatales, pas des problèmes transitoires que le programme sait contourner.

3 Le débogueur GDB

L'utilitaire GDB permet suivre interactivement l'exécution de votre programme. Il permet en particulier de l'interrompre à souhait et d'examiner le contenu des variables. C'est un outil très utile aussi bien pour découvrir l'origine d'une erreur à l'exécution que pour vérifier le bon fonctionnement d'un programme.

Toutes les commandes de GDB vues ici sont récapitulées dans la figure 1. De plus que toutes les commandes sont documentées dans l'aide intégrée, accessible par la commande `help`.

3.1 Lancement de GDB

Avant de déboguer son programme, il est nécessaire le recompiler avec l'option `-g`[10]. Il est aussi important de ne pas utiliser les options d'optimisation `-O`, `-O2` ou `-O3`[11].

[9] La fonction `atexit` qui permet cependant de définir une action à exécuter au moment où `exit` est appelée.

[10] Cela n'est pas strictement indispensable. Mais GDB sera beaucoup moins puissant sans les informations ajoutés dans le binaire par l'option `-g`.

[11] La encore, il est possible de déboguer un programme optimisé, mais c'est beaucoup plus dur. Vous constaterez que l'exécution fait des "sauts en arrière", que certaines variables ont disparu ou ne sont pas mises à jour après une affectation... En fait, les notions de numéro de ligne et de variable n'ont plus vraiment de sens sur un binaire optimisé.

Ensuite, au lieu de lancer l'exécutable, on lance la commande `gdb` en passant en argument le nom de l'exécutable. GDB propose alors un shell interactif. L'invite `(gdb)` indique que GDB est prêt à recevoir vos ordres. Tapez alors la commande `run` pour voir votre programme s'exécuter. Quand GDB rend la main, la commande `quit` (ou l'appui sur Contrôle+D) permet de quitter. Voici un exemple de session :

```
% gcc essai36.c -Wall -Wextra -g
% gdb ./a.out
GNU gdb 6.6
...
(gdb) set args bla bli blo
(gdb) run
Starting program: /home/mine/ATER/2006/prog/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004a0 in main () at essai36.c:5
5         a[i] =  12;
(gdb) quit
```

On voit déjà le bénéfice de GDB : en cas d'opération invalide, la ligne contenant l'opération fautive est indiquée clairement. Comme nous le verrons en 3.3, il est possible d'obtenir de nombreuses informations complémentaires (comme la valeur de `i`, en tapant simplement la commande `print i`).

Arguments du programme. Si vous précisez des arguments supplémentaires sur la ligne de commande `gdb`, ceux-ci sont passés à GDB et non à votre programme. Pour donner des arguments à votre programme, il faut utiliser la commande `set args` *arg1 arg2* $\cdots$ dans GDB, juste avant la commande `run`.

3.2 Exécution sous GDB

Une fois le programme lancé avec `run`, il poursuit son exécution jusqu'à qu'il se termine ou qu'il soit interrompu par GDB. Par terminaison, on entend l'issue du programme telle qu'on l'aurait observé en lançant le programme directement depuis le shell et non sous GDB (issue normale ou exception pour cause d'opération invalide). Par interruption, on entend une suspension temporaire du programme ne faisant pas parti de son fonctionnement normal. Une première méthode d'interruption est l'appui sur Contrôle+C (ceci ne tue pas le programme, comme le ferait un appui sur Contrôle+C lors d'une exécution normale). Dans tous ces cas, GDB reprend la main et propose à l'utilisateur d'entrer des nouvelles commandes.

Reprise de l'exécution. La commande `continue` permet de reprendre l'exécution du programme après une interruption. Notez que ceci est différent de `run` qui recommence son exécution à partir du début (c'est à dire, par la fonction `main`). Il n'est évidement pas possible de reprendre l'exécution d'un programme qui s'est terminé, normalement ou sur une erreur à l'exécution.

Exécution pas à pas. Comme `continue`, les commandes `next` et `step` reprennent l'exécution du programme, mais elles n'exécutent qu'une seule ligne. Quand le programme atteint la ligne suivante, il est de nouveau interrompu. Leur comportement diffère quand la ligne courante contient des appels de fonction :

- `step` s'interrompra dès le premier appel de fonction, juste après l'évaluation de ses arguments, et se positionnera sur la première ligne du corps de la fonction ;
- `next` exécutera jusqu'au bout toutes les fonctions et ne s'interrompra que quand le programme atteindra réellement la ligne suivante.

Les commandes `step` et `next` peuvent être suivies d'un entier n indiquant qu'il faut exécuter n fois `step` ou `next` avant de rendre la main.

Enfin, signalons la commande `finish` qui continue l'exécution jusqu'à ce que la fonction courante se termine.

Notez que, dans tous les cas, on a la garantie que le programme sera interrompu *au plus tard* au moment précisé. Il peut être interrompu avant (par exemple, si l'utilisateur tape Contrôle+C). Notez que, quelque soit la raison pour laquelle GDB rend la main, celui-ci "oublie" la date butoir associée à la dernière commande.

Points d'arrêt. La commande `break` permet de poser un point d'arrêt. À chaque fois que le programme passe par ce point, il est interrompu et GDB rend la main. L'argument de `break` peut être un nom de fonction, auquel cas le point d'arrêt est placé juste avant la première instruction de la fonction. Dans l'exemple suivant, on se sert d'un point d'arrêt en début de programme afin de suivre son exécution pas à pas :

```
$ gdb a.out
GNU gdb 6.6
...
(gdb) break main
Breakpoint 1 at 0x4004f0: file a.c, line 6.
(gdb) run
Starting program: /home/mine/ATER/2006/prog/a.out
...
Breakpoint 1, main () at a.c:6
....
(gdb) next
....
(gdb) next
....
```

Notez qu'un point d'arrêt peut être placé avant la commande `run`.

`break` peut également être suivi d'un numéro de ligne, auquel cas le point d'arrêt est placé au début de la ligne. Le numéro de ligne peut être préfixé par un nom de fichier source et le symbole `:` ce qui est pratique en cas d'ambiguïté (par exemple, `break truc.c:95` place un point d'arrêt au début de la ligne 95 du fichier `truc.c`, pas du fichier courant).

Une fois posé, un point d'arrêt reste actif jusqu'à être explicitement supprimé. Il peut exister plusieurs points d'arrêts. GDB leur affecte un numéro, précisé par le message `Breakpoint` n `at`... La commande `delete breakpoints` n permet alors de supprimer un point d'arrêt à partir de son numéro. Si n est omis, ils sont tous supprimés.

3.3 Inspecter l'état courant

GDB permet d'examiner l'état mémoire d'un programme interrompu. Ceci n'est bien sûr pas possible avant qu'il ait commencé son exécution (avant le premier `run`) ou après qu'il l'ait terminé correctement (message `Program exited normally.`). Il est par contre s'est possible d'examiner l'état mémoire d'un programme qui a été tué par une opération invalide.

Évaluation d'expressions. La commande `print` permet d'évaluer une expression. Elle accepte la plupart des opérateurs C. De plus, l'expression peut faire intervenir toutes les variables visibles au point courant. Il s'agit des variables globales ainsi que des variables locales de la fonction en cours d'exécution (y compris ses arguments). Il est donc facile de connaître le valeur courante d'une variable. Par défaut, `print` affiche tous les champs (resp. indices) d'une structure (resp. tableau), mais il est possible de n'afficher qu'une partie grâce à l'opérateur `.` (resp. `[]`). Enfin, il est possible de déréférencer un pointeur.

La pile d'appels. On appelle *pile d'appels* séquence des appels de fonctions depuis `main` jusqu'à la position courante[12]. La commande `bt` permet d'afficher la pile d'appels courante. La commande `bt full` affiche, de plus, les variables locales et les arguments de chaque fonction de la pile.

[12] Comme pour une pile d'assiettes, on ne voit que celle "du dessus", *i.e.*, on accède qu'aux variables locales de la fonction courante. Un appel de fonction va empiler un nouvel environnement qui cache celui de la fonction appelante. Au retour, on dépile cet environnement pour retrouver celui de la fonction appelante.

commande	effet
Contrôle+D ou `quit`	quitte GDB
`help`	aide intégrée
`run`	commence l'exécution du programme (au début)
Contrôle+C	interrompt l'exécution du programme
`continue`	reprend l'exécution du programme
`next`	exécute la ligne courante, y compris ses appels de fonction
`step`	exécute la ligne courante mais s'interrompt aux fonctions appelées
`next` *nb*	exécute *nb* fois `next`
`step` *nb*	exécute *nb* fois `step`
`finish`	exécute jusqu'au retour de la fonction courante
`up`	remonte d'un cran dans la pile d'appels
`down`	descend d'un cran dans la pile d'appels
`bt full`	affiche la pile d'appels complète et la valeur des variables locales
`print` *expression*	affiche la valeur d'une expression C
`break` *fonction*	place un point d'arrêt au début d'une fonction
`break` *ligne*	place un point d'arrêt en début de ligne
`break` *fichier*:*ligne*	comme ci-dessus en précisant le nom du fichier source
`delete`	supprime tous les points d'arrêt
`delete breakpoints` n	supprime le point d'arrêt numéro n
`set args` *a1 a2* ...	change les arguments en ligne de commande du programme

Fig. 1 – Quelques commandes utiles de GDB.

Il est possible de naviguer dans la pile grâces aux commandes `up` et `down` qui remontent vers l'appelant ou redescendent vers l'appelé. Ceci n'affecte pas l'exécution du programme mais sélectionne seulement l'environnement dans lequel s'exécute la commande `print`. Il est ainsi possible d'accéder aux variables locales des fonctions appelantes, et pas uniquement à celles de la fonction en cours d'exécution.

4 Le vérifieur Valgrind

4.1 Introduction

Valgrind est un outil d'analyse dynamique qui exécute un programme binaire tout en vérifiant, instruction par instruction, qu'il n'effectue pas d'opération dangereuse. Valgrind vérifie en particulier :

- l'utilisation de variables non initialisées,
- certains accès mémoire invalides dans les variables locales,
- certains accès mémoire invalides dans des blocs de mémoire dynamique,
- les utilisations incorrectes de `malloc` et `free`,
- les fuites de mémoire (blocs de mémoire dynamique jamais libérés).

Un des atouts de Valgrind est sa capacité à détecter très tôt les manipulations dangereuses de mémoire. En effet, il est beaucoup plus stricts dans ses vérifications que le système d'exploitation (au détriment de la vitesse d'exécution). Ceci permet d'éliminer de nombreux bogues non déterministes (voir 1.3) avant même qu'ils ne se manifestent. Notez toutefois qu'il est surtout efficace contre les erreurs liées à la mémoire dynamique, et l'est moins contre les accès invalides dans les variables locales et globales.

Valgrind a d'autres utilisations, liées au débogage des programmes parallèles et à l'optimisation du cache et des allocations dynamiques. On ne détaillera pas ici ces utilisations, et renvoyons le lecteur au site officiel de Valgrind : `http://valgrind.org/`.

4.2 Utilisation de Valgrind

Comme pour GDB (voir 3), il est fortement conseillé de compiler son programme avec l'option `-g`[13]. Valgrind se lance par la commande `valgrind`, suivie du nom de l'exécutable, suivie éventuellement des arguments à passer à l'exécutable[14]. Contrairement à GDB, la vérification est automatique et non interactive. Voici un exemple :

```
$ gcc bonjour.c -g
$ valgrind ./a.out
==23232== Memcheck, a memory error detector.
...
==23245==
Bonjour tout le monde!
==23245==
==23245== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 5 from 1)
==23245== malloc/free: in use at exit: 0 bytes in 0 blocks.
==23245== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==23245== For counts of detected errors, rerun with: -v
==23245== All heap blocks were freed -- no leaks are possible.
```

Valgrind affiche les erreurs au fur et à mesure qu'il les détecte. Quand le programme termine, il affiche un résumé des erreurs trouvées et des fuites de mémoire.

En cas d'erreur, Valgrind indiquera non seulement sa position dans le source mais aussi sa *pile d'appels* complète (ce terme est expliqué en 3.3).

Notez que l'exécution sous Valgrind est beaucoup plus lente qu'une exécution normale. Donc, à n'utiliser que pour déboguer !

5 Le profileur Gprof

Un profileur (*profiler* en anglais) tel que Gprof permet d'évaluer le nombre de fois que chaque fonction du programme est appelée, et le temps total passé dans chacune. Son but n'est donc pas l'élimination des bogues, mais l'optimisation, c'est à dire, rendre son programme plus rapide.

On est souvent tenté de réécrire des parties conséquentes de son programme en espérant améliorer son temps d'exécution. Nous rappelons donc ici quelques leçons fondamentales en matière d'optimisation :

1. pour la plupart des programmes, 80% du temps d'exécution est passé dans seulement 20% du programme ;
2. il est plus facile d'optimiser un programme qui marche que de déboguer un programme optimisé (souvent rendu illisible) ;
3. avec l'option `-O3`, le compilateur optimise souvent très bien tout seul ;
4. on gagne parfois beaucoup plus à changer d'algorithme qu'à optimiser une méthode naïve ;
5. terminons par cette maxime de D. Knuth :

   L'optimisation prématurée est à la source de tous les maux.

Il est donc important de partir d'un programme écrit clairement et qui utilise des algorithmes adaptés. Il faut ensuite utiliser un profileur tel que Gprof pour déterminer expérimentalement les quelques fonctions qui conditionnent la rapidité du programme. Ce ne sont pas forcément celles qui sont intrinsèquement les plus lentes ; le nombre d'appels est aussi un facteur important. À ce moment, et à ce moment seulement, on peut envisager quelques réécritures.

[13] Ceci permet à Valgrind d'afficher, pour chaque erreur, le numéro de ligne correspondant dans le source. Sinon, il n'affichera que le nom de la fonction.

[14] Les arguments à passer à Valgrind, si nécessaires, sont à placer entre le nom de la commande, `valgrind`, et le nom de l'exécutable.

5.1 Utiliser Gprof

Avant d'utiliser Gprof, il est nécessaire de recompiler entièrement votre programme avec l'option `-pg` de GCC. Si vous optez pour une compilation séparée, l'option `-pg` doit être utilisée pour les compilations de **tous** vos fichiers `.c` **et pour l'édition de liens**.

Exécutez ensuite votre programme normalement. L'exécution génère un fichier de profilage nommé `gmon.out`. Ce fichier n'est pas lisible directement. On utilise la commande `gprof` pour le convertir en texte lisible. Gprof prend en argument le nom de l'exécutable (`a.out` par défaut) et du fichier de profilage (`gmon.out` si non précisé), dans cet ordre.

L'exécution de Gprof peut prendre un peu de temps. De plus, sa sortie est très longue. On a donc intérêt à la stoker dans un fichier (grâce à l'opérateur de redirection `>` du *shell*) qu'on consultera avec un visualisateur ou un éditeur de texte.

La session suivante illustre l'utilisation de Gprof :

```
% gcc essai1.c -Wall -Wextra -pg
% rm gmon.out
% ./a.out
Bonjour le monde!
% gprof a.out gmon.out > prof
% less prof
Flat profile:

Each sample counts as 0.01 seconds.
...
```

Attention, si un fichier `gmon.out` existe déjà, l'exécution ne remplacera pas ce fichier mais y ajoutera de nouvelles informations. La sortie de Gprof ne distinguera pas les anciennes informations des nouvelles. Ceci permet d'accumuler facilement les statistiques de plusieurs exécutions, mais n'est pas toujours souhaitable. En particulier, si le fichier `gmon.out` provient d'un programme différent ou si le programme a été modifié entre-temps, on obtient à coup sûr des incohérences. Il faut penser dans ce cas à effacer `gmon.out` avant l'exécution de `a.out`.

Une fois le profilage de votre programme terminé, pensez à le recompiler complètement *sans* l'option `-pg`. En effet, un programme qui génère des informations de profilage ne se prête pas à une utilisation normale. Il en est considérablement ralenti. De plus, il sature inutilement le disque de fichiers `gmon.out`.

5.2 Interpréter la sortie de Gprof

Le fichier généré par Gprof étant assez complexe, nous ne le décrivons pas intégralement ici. La documentation complète est accessible au format Info par la commande `info gprof` (plus complète que la page de `man`).

La sortie est généralement composée de deux tables : un *profil plat* suivi d'un *graphe d'appels*. Notez que la signification des colonnes est incluse dans le fichier généré, *après* chaque table.

Profil plat. Le profil plat liste les fonctions du programme[15] classées par contribution décroissante au temps de calcul total. Voici un exemple[16] :

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 58.75      2.73     2.73 534597709     0.00     0.00  mod
 25.52      3.92     1.19   139998     0.00     0.00  isprem
 16.72      4.70     0.78                             frame_dummy
  0.00      4.70     0.00        1     0.00     2.80  doit
  0.00      4.70     0.00        1     0.00     1.12  truc
```

[15] Les fonctions qui ne "semblent" pas contribuer au temps d'exécution du programme sont omises par défaut. L'option `-z` de Gprof permet de les inclure.

[16] On note au passage que les calculs de Gprof sont imprécis car le total des pourcentages vaut 100.99 !

Le nom de la fonction apparaît en dernière colonne. Le temps passé dans chaque fonction est indiqué par la première (temps en %) et la troisième (temps en secondes) colonne. La deuxième colonne donne le temps cumulé de toutes les lignes précédentes. La quatrième colonne indique le nombre de fois que la fonction a été appelée.

Graphe d'appel. Le graphe d'appel permet d'analyser le temps pris par une fonction en fonction de son *contexte d'appel*, c'est à dire, de la fonction appelante. Il est divisé en entrées, une pour chaque fonction. Voici un exemple d'entrée correspondant à la fonction `isprem` (ceci se voit à l'indentation particulière de la dernière colonne) :

```
index % time    self  children    called     name
                0.34    0.78   39999/139998      truc [5]
                0.85    1.95   99999/139998      doit [3]
[1]     83.4    1.19    2.73  139998         isprem [1]
                2.73    0.00 534597709/534597709     mod [4]
```

Examinons la ligne correspondant à la fonction courante, `isprem`. Le temps est divisé entre le temps passé en propre dans `isprem` (colonne `self`) et celui passé dans les fonctions appelées par `isprem` (colonne `children`). Les autres lignes donnent des informations contextuelles. Au dessus, on trouve une ligne pour chaque fonction qui appelle directement `isprem`. Les colonnes `self` et `children` indiquent toujours le temps passé dans `isprem` et les fonctions qu'elle appelle, mais seulement lorsque `isprem` est appelée directement depuis la fonction indiquée en dernière colonne. Au dessous, il s'agit des fonctions directement appelées par `isprem` et de leur temps de calcul en propre et des fonctions qu'elles appellent à leur tour, toujours dans le cas où elles ont été appelées par `isprem`. La colonne `called` indique, à gauche du `/`, le nombre d'appels à ces fonctions sous des contraintes contextuelles similaires. À droite du `/` est rappelé le nombre d'appels total.

5.3 Remarques sur la pertinence des résultats

Du fait de son fonctionnement interne, Gprof n'est pas un outil très fiable. D'abord, il effectue ses mesures à une fréquence faible, afin de ne pas trop ralentir l'exécution du programme (environ 100 fois par seconde). De plus, comme en mécanique quantique, l'outil perturbe la mesure. Le temps d'exécution mesuré n'est pas celui du programme normal, mais celui d'un programme spécialement modifiée. Enfin, comme GDB (voir 3) ou Valgrind (voir 4), Gprof effectue une analyse dynamique. Celle-ci n'est donc valable que pour une exécution particulière et ne donne pas d'indication sur le temps d'exécution en moyenne sur des jeux de données bien réparties, ni sur le cas le pire.