Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Conventions d'écriture et outils de mise au point

Introduction

On va utiliser dans cette partie des outils d'aide à la mise au point de programmes écrits en C. Le C est un langage de bas-niveau : une des conséquences est que le compilateur fait moins de vérifications à la compilation que pour un langage comme Ada par exemple. De même, l'exécution est nettement moins encadrées que pour un programme interprété comme ceux écrits en Java.
Pour faciliter la lecture du code C que l'on va écrire, on se conformera à des règles d'écriture (*coding-style* en anglais). Il en existe plusieurs, on résume ci-dessous celles que l'on vous recommande de suivre.

Règles d'écriture

Indentation et structure du code

Un code C mal-indenté est parfaitement illisible... il est donc important de bien indenter chaque sous-blocs en utilisant la touche `TAB` de votre clavier (votre éditeur préféré saura en général indenter correctement du code C, ne luttez pas contre lui...).
La taille d'une tabulation peut se régler dans votre éditeur : on considère souvent que 4 ou 8 espaces sont des tailles raisonnables (4 permettant de garder des lignes assez courtes, alors que 8 délimite très clairement les blocs). Si votre éditeur le permet, réglez le pour qu'il utilise des espaces à la place de tabulation physiques (l'option s'appelle en général *soft-tabs*), afin de garantir que votre code aura la même apparence chez vous et chez les autres.
Une ligne trop longue est très dure à lire : on ne doit pas écrire de ligne plus longue que 80 caractères. On rappelle qu'en C, vous pouvez fractionner les chaines de caractères très simplement, par exemple au lieu d'écrire :

```
printf("une chaine tres tres tres tres tres tres tres tres tres tres tres longue");
```

écrivez plutôt :

```
printf("une chaine tres tres tres tres tres "
  "tres tres tres tres tres tres longue");
```

ce qui revient exactement au même (le compilateur fusionne les chaines consécutives).
On déconseille en général de mettre plusieurs instructions par ligne, par exemple :

```
x = 4; y = 5;
```

On peut cependant se l'autoriser pour les assertions :

```
p = malloc(10); assert(p != NULL);
```

ou pour les traces :

```
printf("Salut!"); fflush(stdout);
```

Les accolades se placent à la ligne lorsqu'on écrit une fonction :

```
void fct(void)
{
    printf("Bonjour\n");
}
```

Par contre, elles se placent en fin de ligne pour les boucles ou les conditionnels :

```
if (x == 5) {
    y = 10;
} else {
    y = 15;
}
```

Mettez systématiquement des accolades pour délimiter les blocs, même si le bloc ne fait qu'une seule instruction : c'est une source d'erreur fréquente que d'écrire :

```
if (x == 5)
    y = 10;
```

car si vous rajoutez une ligne dans le corps du `if`, il ne fera en fait pas partie du bloc. Ecrivez donc plutôt :

```
if (x == 5) {
    y = 10;
}
```

Une fonction correspond à une action : il faut découper son code en fonctions courtes, pour faciliter la lecture et la mise au point. On entend souvent un contre-argument : « oui mais c'est plus efficace si je mets le code directement, ça évite un appel de fonction ! ». C'est une mauvaise raison : en effet, c'est le travail du compilateur d'optimiser le code, pas le votre (le travail du programmeur consiste à implanter des algorithmes efficaces, par exemple un tri en $O(n \times log\ n)$ et pas en $O(n^2)$, car le compilateur ne sait évidemment pas transformer un mauvais programme en un bon !).

Noms des variables, fonctions, etc

Les noms de variables, fonctions et paramètres s'écrivent en minuscules, avec des blancs soulignés pour séparer les mots : par exemple, `int32_t somme;` ou :

```
void trier_tableau(int64_t tab[], uint64_t taille)
```

Les constantes définies avec `#define` s'écrivent en majuscules : par exemple, `#define TAILLE 10`. Il n'y a pas de bonne raison d'écrire une valeur en dur dans le code (par exemple : `uint32_t tab[10];`), il faut utiliser autant que possible des constantes pour faciliter la lecture et l'évolution du code (si demain votre tableau doit contenir 20 entiers et plus 10, vous serez bien content de n'avoir qu'à changer le `#define` et pas à parcourir tout votre code pour remplacer tous les 10 par des 20 !).
Les noms de types s'écrivent en minuscules, avec `_t` à la fin pour indiquer qu'il s'agit d'un type : par exemple, `struct liste_t`.
Evitez d'utiliser le mot clé `typedef` pour définir des alias de types : l'abus de ce mot clé rend rapidement le code incompréhensible car on ne sait plus quelle sorte de données on manipule (structure, pointeur, etc).
Vous devez utiliser des noms parlants ! Chacun peut trouver tel ou tel nom plus compréhensible qu'un autre, mais de façon général `a`, `b`, `p1`, `p2`, ... ne sont pas des noms faciles à comprendre lorsqu'on relit son code après avoir oublié ce qu'il fait. On s'autorise par contre souvent les variables `i`, `j`, ... comme indices de boucles (mais vous avez le droit de trouver mieux !).

Commentaires

On se demande très souvent à quel point le code doit être commenté. La règle est en fait simple : votre code doit être compréhensible par quelqu'un qui ne l'a pas écrit et qui ne sait pas ce qu'il fait. Si vous utiliser des noms de fonctions, variables, etc parlants, vous n'avez pas besoin de mettre beaucoup de commentaires en général.
Il ne sert à rien d'écrire des commentaires qui ne font que paraphraser le code, par exemple :

```
// on divise x par y
x = x / y;
```

n'a aucun intérêt... par contre :

```
// on est sur ici que y n'est pas nulle
x = x / y;
```

est beaucoup plus pertinent, car le commentaire montre qu'on a pensé à éviter une division par zéro. Vous pouvez aussi ajouter des commentaires listant les pré-conditions de vos fonctions, par exemple :

```
// pre-condition : liste != NULL
void tri_liste(struct liste_t *liste)
{
    ...
}
```

Conseils divers

En C classique, les booléens sont représentés par des entiers, ce qui signifie que le code suivant est correct :

```
int cond = 0; // x est false

if (!cond) {
    ...
}

int val = 5;
if (val) {
    ...
}

char *ptr = malloc(10);
if (ptr) {
    ...
}
```

En C99, on a de vrais booléens : il faut donc les utiliser. De façon générale, il est plus lisible (et absolument tout aussi efficace, le compilateur génère exactement le même code !) d'écrire :

```
bool cond = false;

if (!cond) {
    ...
}

int32_t val = 5;
if (val != 0) {
    ...
}

char *ptr = malloc(10);
if (ptr != NULL) {
    ...
}
```

Enfin, on dispose en C d'un mécanisme très puissant : les assertions. Elles permettent de vérifier des pré-conditions pendant la phase de mise au point sans pénaliser l'exécution du code final. Par exemple :

```
assert(y != 0);
x = x / y;
```

génère bien un test pour vérifier que `y` n'est pas nulle avant de faire la division, ce qui est utile pour mettre au point son programme et trouver rapidement l'erreur s'il s'avère qu'`y` peut en fait être nulle. Lorsqu'on rend le produit fini, il suffit de rajouter l'option `-DNDEBUG` sur la ligne de commande de GCC pour que les assertions soient effacées, donc aucune pénalité dans le code final.
Erreur classique : en C, il est fréquent de confondre l'affectation `=` avec la comparaison `==`. Ce type d'erreurs peut être difficile à détecter car une affectation renvoie une valeur (celle qui est affectée) ce qui veut dire que :

```
if (x = 5) { // en fait, on voulait écrire x == 5...
    ...
}
```

sera évalué comme toujours vrai.
Une astuce couramment employée consiste à utiliser la symétrie de l'opérateur de comparaison en écrivant `if (5 == x)` plutôt que `if (x == 5)`.

En résumé

Le C est un langage bas-niveau : on peut rapidement écrire des choses difficiles à comprendre. Ce n'est pas une raison pour en plus écrire salement (c'est même une raison pour au contraire faire des efforts de présentation !).
De façon très générale, quelque-soit le langage utilisé, c'est bien sûr toujours une bonne pratique de découper son code en sous-modules : un gros fichier .c est en général parfaitement illisible... On verra plus tard comment écrire des modules en C99.

GDB

Récupérer l'archive contenant les sources utilisées pendant cette séance. Vous pouvez compiler toutes les sources fournies en tapant simplement `make` suivi du nom du binaire à générer (par exemple, `make fact` pour compiler le premier programme).
GDB est un débogeur, c'est à dire un logiciel qui est capable notamment d'exécuter pas à pas un programme et d'afficher le contenu des variables utilisées dans le programme. C'est donc un outil très intéressant pour la mise au point de programme, si on veut éviter d'avoir à rajouter des traces partout dans son code pour trouver ses erreurs !
Vous pouvez trouver en ligne des documentations complètes des très nombreuses options supportées par GDB. En pratique, on utilisera qu'un sous-ensemble très restreint des capacités de ce logiciel.
Pour lancer le débogage d'un programme `fact`, il suffit de taper la commande `gdb ./fact` qui charge le programme dans GDB. Ensuite, on peut utiliser les commandes GDB suivantes :

- `run` lance l'exécution du programme : si on n'a pas fixé de point d'arrêt, le programme s'exécutera complètement jusqu'à sa fin (ou jusqu'à ce qu'une erreur termine son exécution), ce qui peut être pratique par exemple pour localiser la ligne de code provocant une erreur de segmentation (à noter que si le programme `prog` prend des arguments sur la ligne de commande, on peut les donner à GDB en tapant simplement `run arg1 arg2 ...`) ;
- `break` place un point d'arrêt : on s'en sert typiquement pour arrêter l'exécution du programme au début d'une fonction, avant de tracer pas à pas l'exécution de la fonction en question (par exemple, `break fact` place un point d'arrêt avant le début de la fonction `fact`, ce qui veut dire qu'une exécution lancée par `run` se mettra en pause avant d'exécuter le corps de `fact`) ;
- `cont` peut être employée une fois qu'on s'est arrêté sur un point d'arrêt, pour continuer l'exécution du programme (jusqu'à la fin ou jusqu'au point d'arrêt suivant) : attention, si vous tapez `run`, vous relancer l'exécution du programme depuis le début ;
- `step` avance l'exécution d'une instruction (c'est à dire d'une ligne de C terminée par un ;) : cette commande permet d'avancer pas à pas dans une fonction pour voir à quel moment une erreur se produit ;

— `next` : cette commande est similaire à `step`, avec la différence qu'elle exécute un appel de fonction de façon atomique (dit autrement, si vous faites `step` alors que votre point d'arrêt est sur `fact`, alors vous rentrez dans la fonction et exécutez son code instruction par instruction, alors que si vous faites `next`, la fonction est appelée sans point d'arrêt et vous passer directement à la ligne suivante dans la fonction appelante) ;
— `display` permet d'afficher le contenu d'une variable, par exemple `display x` affiche le contenu de la variable `x` : il faut bien sûr que la variable existe et soit visible au point où on en est de l'exécution du programme (par exemple, si `x` est une variable locale de `fact`, elle ne sera visible qu'une fois que l'exécution pas à pas sera rentrée dans `fact`) ;
— enfin, il est possible d'annuler certaines des commandes précédentes : `delete #` efface le point d'arrêt numéro #, et `undisplay #` arrête d'afficher la variable numéro #.

Vous trouverez beaucoup plus d'information sur GDB dans cet l'aide-mémoire disponible sur la page principale du cours.
Pour vous exercer, compilez le programme `fact.c` (`make fact`) et lancez son exécution dans GDB. Vous afficherez les valeurs de `x`, `f` et `n` et exécuterez la fonction `fact` pas à pas pour voir évoluer la valeur de `n`. Pensez bien à utiliser `next` lorsque vous arriverez sur le `printf`, sinon vous allez entrer dans du code de la bibliothèque C qui n'est pas intéressant à lire.

Valgrind

Valgrind est un outil d'analyse de la mémoire : il est capable notamment de tracer l'allocation et la désallocation de blocs mémoires, pour éviter par exemple des fuites mémoires (c'est à dire des zones allouées qui ne sont jamais désallouées), ainsi que de détecter des accès à des zones mémoires interdites ou non-allouées. C'est un outil essentiel pour la mise au point de programmes utilisant beaucoup d'allocations dynamiques, comme par exemple les exercices que l'on va faire avec des listes chainées.
Valgrind s'exécute grâce à la commande suivante : `valgrind --leak-check=full ./prog` si `prog` est le binaire du programme qu'on veut analyser.
Pour vous exercer, compiler le programme `segf.c` (`make segf`) qui fait provoque volontairement une erreur de segmentation (lancez le directement, vous verrez que le message ne nous aide pas beaucoup à localiser l'erreur). Executez ensuite `valgrind --leak-check=full ./segf` : vous devez obtenir une trace assez longue qui détaille l'analyse du programme. Les lignes qui nous intéressent ressemblent à (attention, les adresses peuvent être différentes d'une machine à l'autre) :

```
==19921==  Invalid write of size 4
==19921==    at 0x40049D: main (segf.c:9)
==19921==  Address 0xa is not stack'd, malloc'd or (recently) free'd
```

Ce message nous indique :
— que l'on a fait une erreur d'écriture, sur une donnée de taille 4 octets ;
— que cette erreur se situe à la ligne 9 du fichier `segf.c` (on nous donne même l'adresse mémoire où l'instruction fautive est stockée, mais ça ne nous avance pas beaucoup) ;
— que l'adresse où on a essayé d'écrire était l'adresse 10 (0xa en hexadécimal).

Grâce à ce message, on peut immédiatement localiser l'instruction fautive : `*p = 5;` à la ligne 9 du fichier `segf.c` : c'est bien à cette ligne qu'on a essayé d'écrire dans une adresse mémoire où l'on n'avait pas le droit de le faire. Mais en pratique, l'erreur vient plutôt de la ligne 8 `p = (uint32_t *)10;` car c'est à cette ligne qu'on force le pointeur `p` à pointer sur l'adresse 10, où on n'a sûrement pas le droit d'écrire.
Valgrind est aussi utile pour détecter des fuites mémoires : par exemple, si dans un programme vous allouez une zone de 10000 octets avec `malloc` et vous oubliez de la libérer avec `free`, vous obtiendrez un message du genre à la fin des traces de Valgrind :

```
==20097== LEAK SUMMARY:
==20097==    definitely lost: 10,000 bytes in 1 blocks
```

Exercices

On commence par travailler sur le fichier `simple.c` qui contient beaucoup de petits bugs faciles à trouver. Certains empêchent la compilation du fichier, vous devez donc déjà les corriger à la main. Ensuite, vous

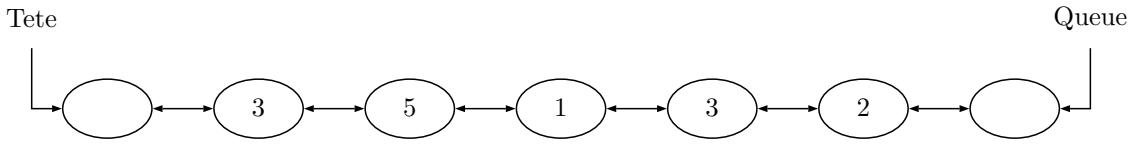
aurez peut-être besoin de GDB pour trouver certains bugs un peu plus obscurs.
Ensuite, on travaille sur le fichier `mem.c` qui utilise beaucoup la mémoire : ce fichier compile et s'exécute sans erreur, mais il contient en fait plusieurs bugs plus difficiles à trouver. L'outil Valgrind peut vous aider à les détecter, ainsi qu'une lecture attentive du code !
Enfin, on vous fourni un module de gestion de listes chainées, qui contient lui-aussi un certain nombre de bugs : dans le fichier `liste_bugs.c`, on définit un type de cellules contenant des entiers naturels sur 32 bits :

```
struct cell_t {
    uint32_t val;
    struct cell_t *suiv;
    struct cell_t *prec;
};
```

Chaque cellule contient deux liens de chainage : un vers la cellule suivant, et un vers la cellule précédente : on travaille donc avec des listes doublement chainées.
On définit ensuite le type d'une liste :

```
struct liste_t {
    struct cell_t *tete;
    struct cell_t *queue;
};
```

Une liste est donc une structure qui contient un pointeur vers la première cellule de la liste et un pointeur vers la dernière.
Une lecture attentive de la fonction de création d'une liste `init_liste` vous permet de comprendre que les cellules de tête et de queue ne sont pas des cellules normales : il s'agit de sentinelles, c'est à dire de cellules sans valeurs qui servent à délimiter la liste et à faciliter l'écriture de certaines fonction (comme la suppression d'un élément par exemple). Notez bien que les listes ne sont pas circulaires.
Par exemple, la liste composée des valeurs 3, 5, 1, 3 et 2 pourra être représentée par le dessin suivant :



On fourni un programme de test qui utilise le module de gestion des listes, vous pouvez le compiler avec `make testliste` : ce programme principal ne comprend (normalement !) aucune erreur, toutes les erreurs sont dans le module.
Pour trouver tous les problèmes, vous utiliserez GDB et Valgrind, mais aussi des outils technologiquement moins avancés mais redoutablement efficaces : une feuille de papier et un crayon, pour faire des dessins des listes que vous manipulez !