

Programmation et structures de données en C

cours 2: découpage, compilation et débogage structures et listes

Jean-Lou Desbarbieux, Stéphane Doncieux
et Mathilde Carpentier
2I001 UPMC 2022/2023

Sommaire

Débogage : outils et méthode

Découpage d'un programme et compilation

Makefile

Débogage : outils et méthode

Qu'est-ce qu'un bug ?

Défaut de conception à l'origine d'un dysfonctionnement.

Exemples de dysfonctionnements :

- ▶ plantage du programme (seg fault, bus error, ...)
- ▶ fuites mémoires
- ▶ comportement indésirable ou erreurs
- ▶ ...

Éviter les bugs pendant l'écriture

- ▶ *"La ligne de code la plus sûre au monde est celle que l'on n'écrit pas !"* Ecrire du code aussi simple que possible, en réutilisant des fonctions bien éprouvées.
- ▶ Travail en binome (pair-programming) : un qui écrit, un qui relit et vérifie (rôles échangés régulièrement)
- ▶ Utilisation d'un style de programmation facilitant la lecture...

Éviter les bugs pendant l'écriture

```
static int e,n,j,o,y;int main(){
for(++;o;(n=~-getchar());e+=11==n,y++)
o=n>0xe^012>n&&' ' ^n^65?!n:!o?++j:o;
printf( "%8d%8d%8d\n" ,e^n,j+=!o&&y,y);}
```

*Dave Burton, prix du programme en 1 ligne le plus complexe,
26eme International Obfuscated C Code Context (2019).*

Si vous voulez écrire du code illisible, participez à cette
compétition ! Sinon écrivez du code lisible !!!

Règles d'écritures à suivre :

- ▶ Écrire un code aéré : une instruction par ligne et lignes vides
- ▶ INDENTER !!! tab ou 3 espaces dans un nouveau bloc
- ▶ Utiliser des noms de fonction et de variable évocateurs
- ▶ Mettre des commentaires
- ▶ Écrire des fonctions compactes (couper au-delà de 30 l)

Faciliter la detection de bugs

Utiliser assert

```
#include <assert.h>
```

```
int main(void) {
    int i=3;
    assert(i==4);
    return 1;
}
```

```
$ ./prog_assert
Assertion failed: (i==4), function main,
file prog_assert.c, line 4.
Abort trap: 6
```

A utiliser pour détecter si une condition que vous pensez vérifiée ne l'est pas.

DéTECTER les bugs à la compilation

Utilisez `-Wall` et supprimez les causes des warnings :

```
int main(void) {
    int i;
    if (i==3)
        printf("i=3\n");
    if (i=4)
        printf("i=4\n");
}
```

```
$ gcc -Wall -o warning warning.c
warning.c: In function `main':
warning.c:4:5: warning: implicit declaration of function `printf' [-Wimplicit-function-declaration]
    printf("i=3\n");
           ^~~~~~

warning.c:4:5: warning: incompatible implicit declaration of built-in function `printf'
warning.c:4:5: note: include '<stdio.h>' or provide a declaration of `printf'
warning.c:6:7: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
    if (i=4) {
           ^
           ^~~~~~

warning.c:7:5: warning: incompatible implicit declaration of built-in function `printf'
warning.c:7:5: note: include '<stdio.h>' or provide a declaration of `printf'
warning.c:3:6: warning: `i' is used uninitialized in this function [-Wuninitialized]
    if (i==3) {
           ^
```

Chasser les bugs à l'exécution

Différentes méthodes :

- ▶ outils de débogage :
 - ▶ valgrind
 - ▶ gdb & ddd
 - ▶ ...
- ▶ "printf method"

Chasser les bugs à l'exécution : valgrind

- ▶ Logiciel permettant (entre autres) de vérifier l'utilisation de la mémoire :
- ▶ Déetecte :
 - ▶ l'utilisation de variables non initialisées
 - ▶ l'utilisation de mémoire libérée
 - ▶ les fuites mémoires
- ▶ Utilisation :
 - ▶ compilation avec l'option `-g`
 - ▶ exécution : `bash$ valgrind ./monprog`

Chasser les bugs à l'exécution : gdb & ddd

- ▶ Le programme doit être compilé avec l'option `-g`
- ▶ `ddd` interface graphique pour `gdb`
- ▶ Permet d'exécuter pas à pas.
- ▶ Permet de poser des points d'arrêt.
- ▶ Permet d'observer les variables.

Chasser les bugs à l'exécution : "printf method"

Mettre des `printf` pour trouver d'où vient le problème...

Exemple de `printf` à réutiliser tel quel :

```
printf("ligne : %d fonction : \"%s\" fichier : %s\n",
       __LINE__, __PRETTY_FUNCTION__, __FILE__);
```

- ▶ `__LINE__` est remplacé par le numéro de ligne de l'instruction
- ▶ `__PRETTY_FUNCTION__` est remplacé par le nom de la fonction dans laquelle est l'instruction
- ▶ `__FILE__` est remplacé par le nom du fichier

Pour aller plus vite :

```
#define printdebug printf("ligne : %d fonction : \"%s\" "
fichier : %s\n", __LINE__, __PRETTY_FUNCTION__, \
__FILE__)
```

et ensuite, chaque fois que vous le souhaitez :

```
printdebug ;
```

Bonnes pratiques, bilan :

1. Ecrire du code lisible et documenté
2. Mettre des assert
3. Se donner les moyens de détecter les bugs : affichage approprié
4. Enlever tous les warnings avec -Wall
5. Exécuter avec valgrind (même s'il n'y a pas de bugs apparent) **et supprimer les warnings !**
6. S'il y a un bug :
 - 6.1 Lancer avec valgrind
 - 6.2 Utiliser ddd ou la "printf method" pour voir les valeurs des différentes variables impliquées et remonter à la source du problème
7. Une fois le problème corrigé, ne pas hésiter à ajouter des assert pour éviter les retours en arrière...

Découpage d'un programme

.h, .c : exemple

Fichier mes_fonctions.h :

```
extern float ma_variable;
int ma_fonction1(int, float);
void ma_fonction2(float, char[10]);
```

Fichier mes_fonctions.c :

```
float ma_variable=12.;
int ma_fonction1(int, float) {
    ...
}
void ma_fonction2(float, char[10]) {
    ...
}
```

Fichier mon_programme.c,
utilisant les fonctions définies dans
mes_fonctions.c :

```
#include "mes_fonctions.h"

int main() {
    int i=0,j;
    float f=ma_variable;
    j=ma_fonction1(i,f);
    ...
}
```

Compilation, macros et préprocesseur

Les étapes permettant de passer d'un fichier source à un executable :

- ▶ Traitement de chaque fichier source indépendamment :
 - ▶ prétraitement : gestion des macros et autres directives au préprocesseur
 - ▶ compilation : transformation du source obtenu en un fichier objet
- ▶ Édition des liens entre les fichiers objets pour générer la bibliothèque ou l'exécutable.

Compilation avec GCC

Préprocesseur, compilateur, éditeur de lien selon les options.

gcc [options] source1.c source2.c...

Options couramment utilisées :

- ▶ -c : prétraitement + compilation (ne pas faire l'édition de lien)
- ▶ -o fichier_sortie : nom du fichier de destination (fichier .o ou exécutable selon les cas). Si non spécifié, a.out pour un exécutable, source.o pour un fichier objet.
- ▶ -Wall : affiche tous les warnings
- ▶ -g : inclure les informations de débogage

Pour information :

- ▶ -E : ne fait que le prétraitement et envoie le résultat sur la sortie standard.

Compilation, macros et préprocesseur : exemple

Compilation de l'exemple précédent :

- ▶ Un header : mes_fonctions.h
- ▶ Deux fichiers sources : mes_fonctions.c, mon_programme.c

1. preprocessing et compilation des sources :

```
gcc -c -o mes_fonctions.o mes_fonctions.c  
gcc -c -o mon_programme.o mon_programme.c
```

2. édition des liens :

```
gcc -o mon_programme mon_programme.o  
mes_fonctions.o
```

(pas de traitement à faire sur le header, il sera inclus dans les fichiers .c par la macro #include par le préprocesseur)

Compilation, macros et préprocesseur : macros

Instructions exécutées avant compilation.

- ▶ `#define` association d'une étiquette à une valeur
- ▶ `#include` inclusion d'un fichier
- ▶ `#ifdef` ou `#ifndef`
- ...
- `#endif`

Makefile

Makefile

À quoi ça sert : simplifier la compilation, prendre en compte automatiquement les dépendances...

Exemple de makefile

```
all: mon_programme

mes_fonctions.o: mes_fonctions.h mes_fonctions.c
    gcc -c -o mes_fonctions.o mes_fonctions.c

mon_programme.o: mon_programme.c mes_fonctions.h
    gcc -c -o mon_programme.o mon_programme.c

mon_programme: mon_programme.o mes_fonctions.o
    gcc -o mon_programme mon_programme.o mes_fonctions.o

clean :
    rm -f *.o mon_programme
```

Makefile : règle

Un Makefile est composé de règles structurées de la façon suivante :

```
cible: dependances
      action 1
      action 2
      ...
```

- ▶ La `cible` est un nom de fichier à créer (ou mettre à jour) ou une action.
- ▶ La partie `dépendances` indique le ou les fichiers dont la cible dépend (séparés par des espaces)
- ▶ Les lignes `action` indiquent les instructions à réaliser pour construire le fichier, le mettre à jour ou réaliser l'action

ATTENTION : il faut mettre une tabulation devant les actions (pas des espaces).

Makefile : utilisation

```
bash$ make cible
```

Recherche le fichier `Makefile` qui est dans le répertoire courant et exécute la règle cible

```
bash$ make
```

Recherche le fichier `Makefile` qui est dans le répertoire courant et exécute la **première règle** du fichier.

Makefile : exécution d'une règle

```
bash$ make cible
```

Détail de l'exécution de la règle :

1. Recherche de la règle dans le fichier Makefile
2. Vérification des dépendances :
 - ▶ Exécution des règles associées (s'il y en a)
 - ▶ Si au moins une dépendance n'existe pas : échec
 - ▶ Si au moins une des dépendances est plus récente que la cible : déclenchement des actions

Makefile : variables

```
objets = fichier1.o fichier2.o
flags = -Wall
cc= gcc

mon_executable: $(objets)
    $(cc) $(flags) -o mon_executable $(objets)

fichier1.o: fichier1.c fichier1.h
    $(cc) $(flags) -c -o fichier1.o fichier1.c

fichier2.o: fichier2.c fichier2.h
    $(cc) $(flags) -c -o fichier2.o fichier2.c
```

Makefile : cibles "classiques"

```
all et clean  
  
objets = fichier1.o fichier2.o  
flags = -Wall  
cc= gcc  
  
all: mon_executable  
  
mon_executable: $(objets)  
    $(cc) $(flags) -o mon_executable $(objets)  
  
fichier1.o: fichier1.c fichier1.h  
    $(cc) $(flags) -c -o fichier1.o fichier1.c  
  
fichier2.o: fichier2.c fichier2.h  
    $(cc) $(flags) -c -o fichier2.o fichier2.c  
  
clean:  
    rm -rf $(objets) mon_executable
```

Makefile : variables automatiques

Les variables dites automatiques permettent de faire référence à des éléments de la règle :

- ▶ \$@ : cible de la règle
- ▶ \$< : nom de la première dépendance
- ▶ \$? : toutes les dépendances plus récentes que le but
- ▶ \$^ : toutes les dépendances
- ▶ \$+ : idem mais chaque dépendance apparaît autant de fois qu'elle est citée et l'ordre d'apparition est conservé

Exemple :

```
mon_executable: $(objets)
    $(cc) $(flags) -o $@ $^
```

Makefile : règles implicites

Règle qui va s'appliquer à tous les fichiers respectant un certain patron indiqué avec des %. Utilisé avec des variables automatiques.

Exemple pour compiler des fichiers sources C :

```
% .o : % .c % .h  
    $(CC) $(flags) -c $<
```

→ la première dépendance (le fichier .c) est compilé pour créer le fichier objet. Cela créera le fichier .o automatiquement, mais si on voulait le spécifier dans la règle, on pourrait ajouter `-o $@` à la commande gcc.

Makefile : conclusion

Quelques points de prudence :

- ▶ Ne pas oublier les tabulations devant les règles
- ▶ Ne pas faire d'erreur dans l'appel à `rm`, pas possible de revenir en arrière si vous faites une erreur...

C'est tout pour aujourd'hui !