



Langage C – Cours 8

Lélia Blin

lelia.blin@irif.fr

2023 - 2024

Modularités du code (suite)

Makefile

Debug

Variables globales

- Dans un fichier prog.c on peut déclarer et initialiser une variable globale en dehors des définitions de fonctions
- Cette variable sera visible par toutes les fonctions du programme
- Il faut penser à bien l'initialiser

Example

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

int cpt=0;

void inc();
void dec();
int getcpt();

int main(){
    inc();
    inc();
    printf("cpt vaut %d\n",getcpt());
    dec();
    printf("cpt vaut %d\n",getcpt());
    return EXIT_SUCCESS;
}

void inc(){
    ++cpt;
}

void dec(){
    if(cpt>0){
        --cpt;
    }
}

int getcpt(){
    return cpt;
}
```

glob.c

Variables globales vs plusieurs fichiers

- On peut avoir une variable globale dans plusieurs fichiers
- Par exemple, dans glob2. c dans le programme main, on va initialiser la valeur de cpt
- Et dans cpt.c on va mettre les fonctions inc, dec et getcpt
- Il peut y avoir plusieurs fichiers où la même variable existe
- **MAIS un seul endroit où elle est initialisée,**
 - Sinon le linker à la compilation fera une erreur
 - Dans les autres fichiers on va déclarer la variables comme extern
- Même si elle est est présente dans plusieurs fichiers, au final il n'y aura qu'une seule variable

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "cpt.h"
```

```
extern int cpt;
```

```
int main(){
    cpt=10;
    inc();
    inc();
    printf("cpt vaut %d\n",getcpt());
    dec();
    printf("cpt vaut %d\n",getcpt());
    return EXIT_SUCCESS;
}
```

**Si on initialise aussi ici
erreur avec le linker**

```
#include "cpt.h"
```

```
int cpt=0;
```

```
void inc(){
    ++cpt;
}
```

```
void dec(){
    if(cpt>0){
        --cpt;
    }
}
```

```
int getcpt(){
    return cpt;
}
```

glob2.c

cpt.c

```
extern void inc();
extern void dec();
extern int getcpt();
```

cpt.h

Bonne pratique

- En fait le mieux, c'est de déclarer la variable extern dans le fichier .h lié au fichier où l'on crée la variable
 - dans notre cas cpt.h

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "cpt.h"

int main(){
    cpt=10;
    inc();
    inc();
    printf("cpt vaut %d\n",getcpt());
    dec();
    printf("cpt vaut %d\n",getcpt());
    return EXIT_SUCCESS;
}
```

glob2.c

```
#include "cpt.h"

int cpt=0;

void inc(){
    ++cpt;
}

void dec(){
    if(cpt>0){
        --cpt;
    }
}

int getcpt(){
    return cpt;
}
```

cpt.c

```
#ifndef CPT_H
#define CPT_H

extern int cpt;

extern void inc();
extern void dec();
extern int getcpt();

#endif
```

cpt.h

Le mot clef static

- **Rappel** : on ne peut pas avoir deux fonctions avec le même nom dans des programmes utilisés ensemble
- Mais si on déclare une fonction/variable avec le mot-clef **static**, la fonction n'est visible que dans le fichier où elle est déclarée
 - On peut donc avoir dans deux fichiers séparés, deux fonctions avec le même nom si elles sont déclarées comme **static**
- Si une variable est déclarée avec le mot clef **static**, sa création est permanente
 - Si elle est déclarée dans une fonction, elle ne sera pas créée à chaque appel de la fonction mais une seule fonction
 - Sa valeur sera gardée
- On peut déclarer une variable static à l'intérieure ou à l'extérieur d'une fonction
 - dans une fonction : la variable n'est visible que par la fonction
 - à l'extérieur : elle est visible par toutes les fonctions du fichier

Exemple

```
#include <stdio.h>
#include <stdlib.h>

unsigned f(unsigned v){
    static unsigned x=10;
    ++x;
    return (v+x);
}

int main(){
    unsigned m=f(2);
    printf("m vaut : %u\n",m);
    m=f(2);
    printf("m vaut : %u\n",m);
}
```

➔ **Affiche 13**

➔ **Affiche 14**

stat.c

Exemple

```
#include <stdio.h>
#include <stdlib.h>

static unsigned x=10;

unsigned g(unsigned v){
    ++x;
    return (v*x);
}

unsigned f(unsigned v){
    ++x;
    return (v+x);
}

int main(){
    unsigned m=f(2);
    printf("m vaut : %u\n",m);
    m=g(2);
    printf("m vaut : %u\n",m);
}
```

➔ **Affiche 12**

➔ **Affiche 24**

stat2.c

Makefile

- Un fichier Makefile permet d'automatiser la compilation
- En pratique, vous avez votre projet avec différents fichiers et un appel à make (avec éventuellement permet de faire automatiquement les opérations de compilation)
- Pourquoi ?
 - Le fichier Makefile gère les dépendances entre fichiers, donc la compilation lors du développement est facilité
 - L'installation est plus simple (pas besoin de lister dans un readme toutes les opérations à faire pour compiler votre projet)
- En pratique, un fichier Makefile est une liste de règles de la forme suivante :

```
cible : liste de dépendances  
<TAB> commandes unix
```



La tabulation ici est obligatoire

Exemple

```
all : prog

prog : prog.c
      gcc -Wall -o prog prog.c

clean :
      rm -rf prog
```

- Si on fait **make prog**
 - On a besoin de prog.c
- Si on fait **make clean**
 - Cela appelle la commande liée à la règle clean (efface prog)
- Si on fait **make** tout seul
 - On prend la première règle
 - Ici cela nous dit que pour faire make, on a besoin de prog
 - Et la règle au-dessus nous dit comment obtenir prog

stat.c

Exemple

```
all : glob2

cpt.o : cpt.c cpt.h
    gcc -Wall -c cpt.c

glob2.o : glob2.c cpt.h
    gcc -Wall -c glob2.c

glob2 : glob2.o cpt.o
    gcc -Wall -o glob2 glob2.o cpt.o

clean :
    rm -rf glob2 *.o
```

Makefile

Utilisation de macros

- On peut définir des macros au début pour rendre plus systématique la création de Makefile et aussi leur réutilisation

- Pour les définir :

```
nom_macro=définition de la macro
```

- Pour utiliser une macro, on l'appelle de la façon suivante **\$(nom_macro)**
- Ensuite, quand on appelle le Makefile, toutes les macros sont remplacés par leur définition

Exemple

```
CC=gcc  
CFLAGS=-Wall  
DEPS=cpt.h  
EXEC=glob2  
  
all : $(EXEC)  
  
cpt.o : cpt.c $(DEPS)  
      $(CC) $(CFLAGS) -c cpt.c  
  
glob2.o : glob2.c $(DEPS)  
         $(CC) $(CFLAGS) -c glob2.c  
  
glob2 : glob2.o cpt.o  
       $(CC) $(CFLAGS) -o glob2 glob2.o cpt.o  
  
clean :  
      rm -rf $(EXEC) *.o
```

Nom du compilateur

Option de compilation

Liste des .h utiles

Nom de l'exécutable

Makefile

Variables et règles génériques

- Pour simplifier le Makefile et sa réutilisation, on peut aussi recourir à des variables internes, par exemple :
 - `$@` : le nom de la cible
 - `$<` : le nom de la première dépendance
 - `^` : la liste des dépendances
 - `$?` : la liste des dépendances plus récentes que la cible
 - `*` le nom du fichier sans suffixe
- On peut aussi faire des règles génériques s'appliquant à tous les fichiers du même type :

```
% .o : %.c  
    règle générique pour les fichiers .o
```

Exemple

```
CC=gcc
CFLAGS=-Wall
DEPS=cpt.h
EXEC=glob2

all : $(EXEC)

cpt.o : cpt.c $(DEPS)
    $(CC) $(CFLAGS) -c cpt.c

glob2.o : glob2.c $(DEPS)
    $(CC) $(CFLAGS) -c glob2.c

glob2 : glob2.o cpt.o
    $(CC) $(CFLAGS) -o glob2 glob2.o cpt.o

clean :
    rm -rf $(EXEC) *.o
```

Makefile

Exemple

```
CC=gcc
CFLAGS=-Wall
DEPS=cpt.h
EXEC=glob2

all : $(EXEC)

%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c $<

glob2 : glob2.o cpt.o
    $(CC) $(CFLAGS) -o $@ $^

clean :
    rm -rf $(EXEC) *.o
```

Makefile

Vous pouvez réutiliser ce Makefile tel quel en changeant :

- Le nom de l'exécutable
- Les .h dans la macro DEPS
- Les .o dans la règle pour glob2 (et aussi le nom glob2)

Débugger

- Il est important de faire des affichages d'informations afin de trouver les bugs dans vos programmes
- Une méthode simple consiste à faire des printf régulièrement
- Pour éviter de les mettre puis de les enlever ces printf, on peut penser à développer un mode debug
- Pour cela, on peut faire un fichier debug.h avec une seule ligne

```
#define DEBUG 1
```

- Quand on passe en mode non-debug on change la valeur de DEBUG de 0 à 1
- Et après dans le code, on peut faire des tests selon la valeur de cette macro

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include "debug.h"

unsigned fact(unsigned n){
    unsigned v=0;
    if(DEBUG){
        printf("DEBUG : Appel fact(%u)\n",n);
    }
    if(n==0){
        v=1;
    }else{
        v=n*fact(n-1);
    }
    if(DEBUG){
        printf("DEBUG : ----> Valeur renvoyee %u\n",v);
    }
    return v;
}

int main(){
    if(DEBUG){
        printf("DEBUG : Lancement du programme\n");
    }
    int v=fact(8);
    printf("Le resultat est : %u\n",v);
    return EXIT_SUCCESS;
}
```

prog-deb.c

Outils

- Il existe aussi des outils pour déboguer/corriger vos programmes
- Par exemple :
 - **gdb** : le déboggeur de gnu, qui permet de faire avancer le programme pas à pas, d'arrêter son exécution, de voir la valeur des variables à un instant etc
 - **valgrind** : pour détecter les problèmes liés à la manipulation de la mémoire en particulier les fuites mémoire

Exemple

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

char *ch;

unsigned somme_n(unsigned n){
    unsigned r=0;
    for(unsigned i=0;i<=n;++i){
        r+=i;
    }
    return r;
}

int main(){
    printf("Entrer un entier positif :\n");
    unsigned v=0;
    scanf("%u",&v);
    unsigned res=somme_n(v);
    int r=sprintf(ch,"Le resultat est %u\n",res);
    assert(r>=0);
    printf("%s",ch);
    return EXIT_SUCCESS;
}
```

crash-prog.c

gdb

- Pour compiler le programme précédent pour gdb :
 - **gcc -g -Wall -o crash-prog crash-prog.c**
- Si on exécute ce programme normalement, on obtient une erreur de segmentation

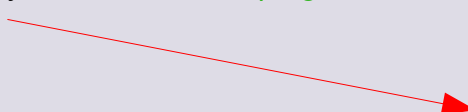
```
./crash-prog  
Entrer un entier positif :  
10  
Erreur de segmentation
```

- Mais, on n'a aucune information de ce qui a pu causer l'erreur de segmentation
- On va donc utiliser gdb
- Pour le lancer avec le programme on fait :
 - **gdb crash-prog**

Exemple

```
gdb crash-prog
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from crash-prog...
(gdb)
```



**Ici gdb attend qu'on lui
dise quoi faire en lui
donnant des commandes à
exécuter**

Commandes

- **run**

- Elle exécute le programme jusqu'à ce qu'il termine où rencontre une erreur

```
(gdb) run
Starting program: /ens/sangnier/TestC/crash-prog
Entrer un entier positif :
10

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e502e1 in __GI___IO_str_overflow (fp=0x7fffffffe710, c=76)
    at strops.c:133
133  strops.c: Aucun fichier ou dossier de ce type.
(gdb)
```

- **backtrace**

- on peut ensuite appeler backtrace qui permet d'avoir la liste des appels de fonction ayant amené à l'erreur

Exemple

```
(gdb) backtrace
#0  0x00007ffff7e502e1 in __GI__IO_str_overflow (fp=0x7ffffffffffe710, c=76)
    at strops.c:133
#1  0x00007ffff7e4eba1 in __GI__IO_default_xsputn (n=<optimized out>,
    data=<optimized out>, f=<optimized out>) at genops.c:399
#2  __GI__IO_default_xsputn (f=0x7ffffffffffe710, data=<optimized out>, n=16)
    at genops.c:370
#3  0x00007ffff7e35d80 in __vfprintf_internal (s=s@entry=0x7ffffffffffe710,
    format=format@entry=0x555555556022 "Le resultat est %u\n",
    ap=ap@entry=0x7ffffffffffe850, mode_flags=mode_flags@entry=0)
    at ../libio/libioP.h:948
#4  0x00007ffff7e430d0 in __vsprintf_internal (string=0x0,
    maxlen=maxlen@entry=18446744073709551615,
    format=0x555555556022 "Le resultat est %u\n",
    args=args@entry=0x7ffffffffffe850, mode_flags=mode_flags@entry=0)
    at iovsprintf.c:96
#5  0x00007ffff7e22f14 in __sprintf (s=<optimized out>, format=<optimized out>)
    at sprintf.c:30
#6  0x0000555555555201 in main () at crash-prog.c:20
(gdb)
```

Probleme a la ligne 20 de crash-prog.c

```
int r=sprintf(ch,"Le resultat est %u\n",res);
```

Commandes

- **break**

- Cette fonction permet de mettre des points d'arrêts dans le programme
- Quand on fera un run, le programme s'arrêtera là
- On peut dire **break crash-prog.c:20**
 - Met un point d'arrêt à la ligne 20 du programme crash-prog
- Ou encore **break crash-prog.c:nom-fonction**
 - Le programme s'arrête avant l'exécution de la fonction

- **print**

- Permet d'afficher la valeur de variables
- par exemple **print j** -> affiche le contenu de la variable j

Exemple

```
(gdb) break crash-prog.c:20
Breakpoint 1 at 0x555555551e3: file crash-prog.c, line 20.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /ens/sangnier/TestC/crash-prog
Entrer un entier positif :
10

Breakpoint 1, main () at crash-prog.c:20
20    int r=sprintf(ch,"Le resultat est %u\n",res);
(gdb) print res
$1 = 55
(gdb) print ch
$2 = 0x0
(gdb)
```



La valeur de ch est NULL

Exemple

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

char ch[100];

unsigned somme_n(unsigned n){
    unsigned r=0;
    for(unsigned i=0;i<=n;++i){
        r+=i;
    }
    return r;
}

int main(){
    printf("Entrer un entier positif :\n");
    unsigned v=0;
    scanf("%u",&v);
    unsigned res=somme_n(v);
    int r=sprintf(ch,"Le resultat est %u\n",res);
    assert(r>=0);
    printf("%s",ch);
    return EXIT_SUCCESS;
}
```

crash-prog-corr.c

Commandes

- **quit**
 - Pour quitter le débbugger
- On peut aussi se déplacer dans le programme après un breakpoint
 - **next** (ou **n**) : exécute la commande courante
 - **step** (ou **s**) : exécute la commande courante pas à pas (si il s'agit d'une fonction, on va rentrer dans le corps de la fonction)
 - **continue** (ou **c**) : continue jusqu'au prochain points d'arrêts ou jusqu'à la fin
- **info breakpoints**
 - Affiche les points d'arrêt
- **clear nom_breakpoints**
 - permet d'annuler un point d'arrêt
- **where**
 - Permet de voir l'état de la pile d'appels

Exemple

```
(gdb) break fact
Breakpoint 1 at 0x1150: file prog-deb.c, line 6.
(gdb) run
Starting program: /ens/sangnier/TestC/prog-deb
DEBUG : Lancement du programme

Breakpoint 1, fact (n=8) at prog-deb.c:6
6      unsigned v=0;
(gdb) c
Continuing.
DEBUG : Appel fact(8)

Breakpoint 1, fact (n=7) at prog-deb.c:6
6      unsigned v=0;
(gdb) c
Continuing.
DEBUG : Appel fact(7)

Breakpoint 1, fact (n=6) at prog-deb.c:6
6      unsigned v=0;
(gdb) where
#0  fact (n=6) at prog-deb.c:6
#1  0x000055555555189 in fact (n=7) at prog-deb.c:13
#2  0x000055555555189 in fact (n=8) at prog-deb.c:13
#3  0x0000555555551cb in main () at prog-deb.c:25
(gdb)
```


Exemple

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

int *creer_tab(unsigned l){
    int *t=malloc(l*sizeof(int));
    assert(t!=NULL);
    int v=0;
    for(int *t2=t;t2<t+l;++t2){
        *t2=v;
        ++v;
    }
    return t;
}

void affiche_tab(int *t,unsigned n){
    assert(n>0);
    printf("[");
    for(int *t1=t;t1<t+n-1;++t1){
        printf("%d ",*t1);
    }
    printf("%d]\n",*(t+n-1));
}

int main(){
    int *tab1=creer_tab(5);
    affiche_tab(tab1,5);
    tab1=creer_tab(2);
    affiche_tab(tab1,2);
    return EXIT_SUCCESS;
}
```

valgrind

- Pour compiler le programme précédent pour gdb :
 - **gcc -g -Wall -o mall-prog mall-prog.c**
- Si on exécute ce programme normalement, tout se passe bien

```
./mall-prog  
[0 1 2 3 4]  
[0 1]
```

- Mais, en fait ce programme a des problèmes
- Il fait des fuites mémoire
 - Il réserve des endroits dans le tas mémoire avec **malloc**
 - et il ne libère pas ces espaces en faisant un **free** ensuite

Exemple

```
/TestC$ valgrind ./mall-prog
==1239491== Memcheck, a memory error detector
==1239491== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1239491== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1239491== Command: ./mall-prog
==1239491==
[0 1 2 3 4]
[0 1]
==1239491==
==1239491== HEAP SUMMARY:
==1239491==    in use at exit: 28 bytes in 2 blocks
==1239491== total heap usage: 3 allocs, 1 frees, 1,052 bytes allocated
==1239491==
==1239491== LEAK SUMMARY:
==1239491==    definitely lost: 28 bytes in 2 blocks
==1239491==    indirectly lost: 0 bytes in 0 blocks
==1239491==    possibly lost: 0 bytes in 0 blocks
==1239491==    still reachable: 0 bytes in 0 blocks
==1239491==    suppressed: 0 bytes in 0 blocks
==1239491== Rerun with --leak-check=full to see details of leaked memory
==1239491==
==1239491== For lists of detected and suppressed errors, rerun with: -s
==1239491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Example

```
valgrind --leak-check=full ./mall-prog
==1243417== Memcheck, a memory error detector
==1243417== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1243417== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1243417== Command: ./mall-prog
==1243417==
[0 1 2 3 4]
[0 1]
==1243417==
==1243417== HEAP SUMMARY:
==1243417==   in use at exit: 28 bytes in 2 blocks
==1243417== total heap usage: 3 allocs, 1 frees, 1,052 bytes allocated
==1243417==
==1243417== 8 bytes in 1 blocks are definitely lost in loss record 1 of 2
==1243417==   at 0x483877F: malloc (vg_replace_malloc.c:307)
==1243417==   by 0x10917E: creer_tab (mall-prog.c:6)
==1243417==   by 0x1092C2: main (mall-prog.c:28)
==1243417==
==1243417== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==1243417==   at 0x483877F: malloc (vg_replace_malloc.c:307)
==1243417==   by 0x10917E: creer_tab (mall-prog.c:6)
==1243417==   by 0x1092A3: main (mall-prog.c:26)
==1243417==
==1243417== LEAK SUMMARY:
==1243417==   definitely lost: 28 bytes in 2 blocks
==1243417==   indirectly lost: 0 bytes in 0 blocks
==1243417==   possibly lost: 0 bytes in 0 blocks
==1243417==   still reachable: 0 bytes in 0 blocks
==1243417==     suppressed: 0 bytes in 0 blocks
==1243417==
==1243417== For lists of detected and suppressed errors, rerun with: -s
==1243417== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Exemple

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

int *creer_tab(unsigned l){
    int *t=malloc(l*sizeof(int));
    assert(t!=NULL);
    int v=0;
    for(int *t2=t;t2<t+l;++t2){
        *t2=v;
        ++v;
    }
    return t;
}

void affiche_tab(int *t,unsigned n){
    assert(n>0);
    printf("[");
    for(int *t1=t;t1<t+n-1;++t1){
        printf("%d ",*t1);
    }
    printf("%d]\n",*(t+n-1));
}

int main(){
    int *tab1=creer_tab(5);
    affiche_tab(tab1,5);
    free(tab1);
    tab1=creer_tab(2);
    affiche_tab(tab1,2);
    free(tab1);
    return EXIT_SUCCESS;
}
```

mall-prog-corr.c

Exemple

```
valgrind ./mall-prog-corr
==1252719== Memcheck, a memory error detector
==1252719== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1252719== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1252719== Command: ./mall-prog-corr
==1252719==
[0 1 2 3 4]
[0 1]
==1252719==
==1252719== HEAP SUMMARY:
==1252719==    in use at exit: 0 bytes in 0 blocks
==1252719== total heap usage: 3 allocs, 3 frees, 1,052 bytes allocated
==1252719==
==1252719== All heap blocks were freed -- no leaks are possible
==1252719==
==1252719== For lists of detected and suppressed errors, rerun with: -s
==1252719== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Exemple

```
(gdb) break fact
Breakpoint 1 at 0x1150: file prog-deb.c, line 6.
(gdb) run
Starting program: /ens/sangnier/TestC/prog-deb
DEBUG : Lancement du programme

Breakpoint 1, fact (n=8) at prog-deb.c:6
6      unsigned v=0;
(gdb) c
Continuing.
DEBUG : Appel fact(8)

Breakpoint 1, fact (n=7) at prog-deb.c:6
6      unsigned v=0;
(gdb) c
Continuing.
DEBUG : Appel fact(7)

Breakpoint 1, fact (n=6) at prog-deb.c:6
6      unsigned v=0;
(gdb) where
#0  fact (n=6) at prog-deb.c:6
#1  0x000055555555189 in fact (n=7) at prog-deb.c:13
#2  0x000055555555189 in fact (n=8) at prog-deb.c:13
#3  0x0000555555551cb in main () at prog-deb.c:25
(gdb)
```