Programmation C avancée

Concepts & Outils
pour le développement

maj 03/2024

Les Tests

HYUNDAI
TUCSON
INSURANCE INSTITUTE
FOR HIGHWAY SAFETY
CEF1004

Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
  - au niveau d'une fonction : tests unitaires
  - au niveau d'un ou plusieurs modules : tests d'intégration
  - au niveau général, applicatif : tests de recette

TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis du code validant les tests.
- La méthode XP (extreme programming) repose en partie sur partie sur TDD.
- TDD repose sur des cycles courts consistant :
  - à écrire un test fonctionnel
  - vérifier que le test plante
  - à écrire un test unitaire
  - vérifier que le test plante
  - écrire le code minimal pour que le test fonctionne
  - vérifier que le test passe

TDD

- A la fin de l'écriture d'un code et lorsque tous les tests passent, on peut vouloir refactoriser le code (copier/coller, simplification, unification, …)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce que à nouveau il valide l'ensemble des tests.

TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de scores de bowling :

exemple grandement inspirée de la présentation « TDD in C » par Olve Maudal (dispo. slideshare)

bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
  - Si le joueur élimine les 10 quilles du premier coup, il fait **strike** et il ne joue pas de 2ème boule
  - Si le joueur élimine les 10 quilles au deuxième coup, il fait **spare**
- Le score d'un set fait :
  - le score précédent + la somme des deux lancés si pas de strike ni de spare
  - le score précédent + 10 + le score du premier lancé du prochain set si spare
  - le score précédent + 10 + le score du prochain set si strike.
- Pour le 10ème set, le joueur peut avoir un troisième lancé

bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
  - void roll(BowlingGame *,int nbPinsDown) : enregistre un nouveau lancé
  - int score(BowlingGame *) : renvoie le score actuel

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {
    struct Set sets[10] ;
}
```

- Il faut connaître le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancés

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets

```
struct Game {
    struct Set s[10];
}
```

- Il faut connaître le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancés

PAS TDD !

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

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

int main(){
	assert(false && « c'est parti ») ;
}
```

```
$ make bow
gcc -Wall  bow.c   -o bow
$ ./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```

ok, le système de test fonctionne !

bowling : cas vide

- Commençons par le cas vide

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

void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game) ;
}

int main(){
  test_empty();
}
```

```
$ gcc -Wall bow.c
bow.c: In function ‘test_empty’:
bow.c:6:10: warning: implicit declaration of function ‘bg_init’ [-Wimplicit-function-declaration]
   struct BowlingGame *game=bg_init();
          ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
   struct BowlingGame *game=bg_init();
```

Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H


struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int );
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *) ;

#endif
```

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```


Ajout de bowling.c

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```

Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
  return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return -1;
}
void bg_free(struct BowlingGame *g){}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

bowling : cas vide

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

Ajout du score

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

```
$ gcc bowling.c bow.c -
Wall
$ ./a.out
$ valgrind ./a.out
ok
```

bowling : test tout à 1

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

void test_empty(){ … }

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game) ;
}

int main(){
  test_empty();
  test_all_ones() ;
}
```

```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

bowling : test tout à 1

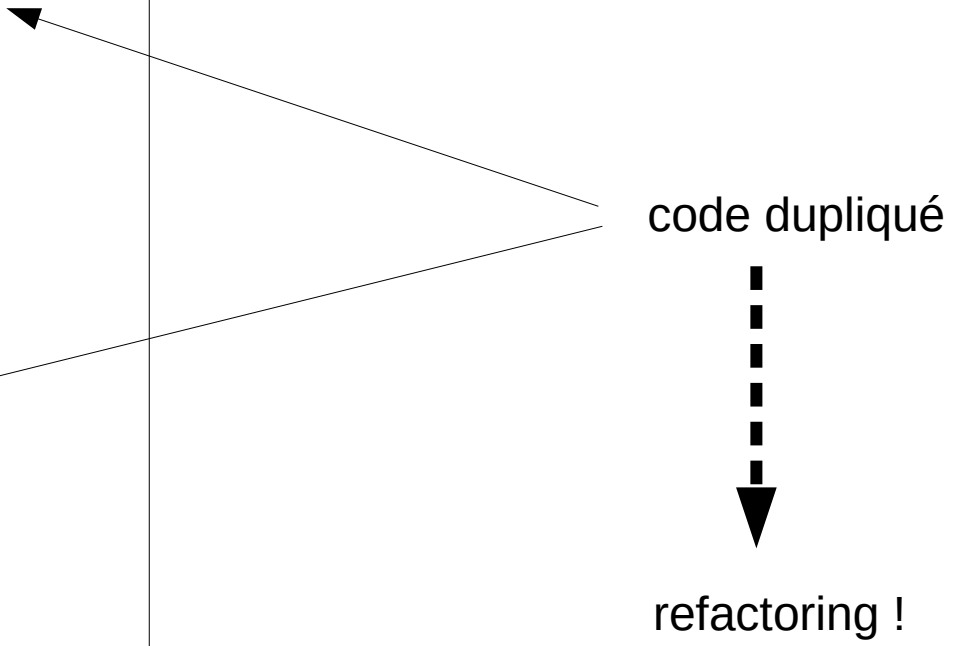
```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

```
$ gcc bowling.c bow.c -Wall
$ ./a.out
$
```

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```



bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

```
void rolls(struct BowlingGame *game,int n, int v){
  int i;
  for(i=0;i<n;++i)
    bg_roll(game,v);
}

void test_empty(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

bowling : un spare

```
void test_one_spare(){
  struct BowlingGame *game=bg_init();
  bg_roll(game,5);
  bg_roll(game,5);
  bg_roll(game,3);
  rolls(game,17,0);
  assert(bg_score(game)==16 && "test one spare");
}
```

```
$ gcc bow.c bowling.c -Wall
allali@hebus:~/SVN_LaBRI/ENSEIRB/PG106/Cours$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16 && "test one spare"' failed.
```

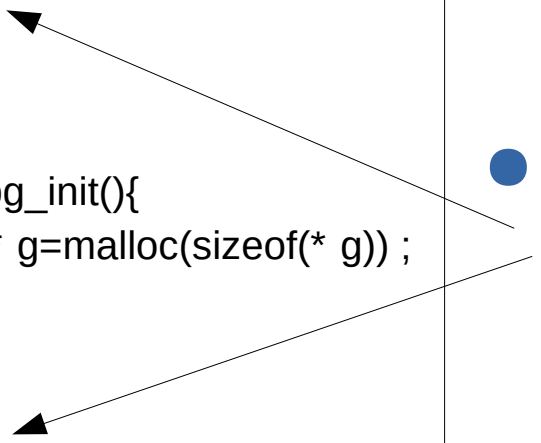
bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.

bowling : conception

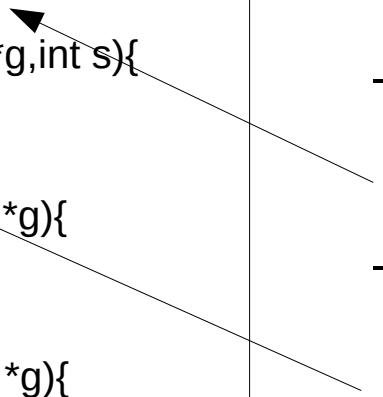
```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```



- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```



- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

Refactoring !

- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : refactoring

- On revient en arrière :

```
int main(){
  test_empty();
  test_all_ones();
  // test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

- On modifie le code : ajout d'un tableau de score, du coup en cours et mise à jour de la fonction de calcul

bowling : refactoring

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int rolls[21] ;
        int current;
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
  g->score+=s ;
   g->rolls[g->current++]=s ;
}
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
 return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

```
int main(){
  test_empty();
  test_all_ones();
  test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

bowling : one spare (back)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```



```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) {
      if (g->rollls[i]+g->rolls[i+1]==10){
            // this is a spare...
            score= ... ; // ?
       }
       score+=g->rolls[i] ;
  }
  return score ;
}
```

ca ne marchera pas car il faut compter
par set. On doit encore faire un refactoring !

```
int main(){
  test_empty();
  test_all_ones();
  //test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : refactoring (again)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```

```
struct BowlingGame *bg_init(){
  int i ;
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  for(i=0;i<21;++i) g->rolls[i]=0 ;
  return g ;
}

int bg_score(struct BowlingGame *g){
  int score=0, frame;
  for(frame=0;frame<10;++frame) {
        score+=g->rolls[2*frame]+g->rolls[2*frame+1] ;
  }
  return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : one spare (again)

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

```
int bg_score(struct BowlingGame *g){
  int score=0, frame, hits;
  for(frame=0;frame<10;++frame) {
        hits=g->rolls[2*frame]+g->rolls[2*frame+1] ;
        score+=hits ;
        if (hits==10) // spare
              score+=+g->rolls[2*frame+2] ;
  }
  return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : TDD

- Et ainsi de suite :
  - ajout d'un test avec un strike
  - cas pour la fin de partie
  - …
- Le cycle à suivre en TDD est :
  - écriture d'un test
  - le test ne passe pas : ROUGE
  - écriture du code
  - le test passe : VERT
- Lorsqu'on ré-écrit des tests, on ne touche pas au code jusqu'à ce que ça repasse au vert.
- Lorsqu'on ré-écrit le code, on ne touche pas aux tests jusqu'à ce que ça repasse au vert.

Tests

- Rappel : Il existe plusieurs types de tests.
- Les plus importants sont :
  - Les tests unitaires et les tests fonctionnels
  - Les tests d'intégration
  - Les tests de recette

Les tests unitaires

- Les tests unitaires ont pour objet de valider le **fonctionnement** d'une fonction.
- Pour qu'un test unitaire soit correct, il faut tester le fonctionnement « normal » ainsi qu'aux limites (cas NULL, domaine de valeur).
- Le test unitaire repose sur la **spécification** de la fonction et son implémentation.
- Les tests fonctionnels reposent eux uniquement sur la spécification et ne sont pas limités à une seule fonction.

Les tests unitaires

- Ils doivent mettre en œuvre l'ensemble des fonctionnalités décrites dans les **spécifications**, et explorer le fonctionnement du module dans des conditions non spécifiées (discutable...)
  - Doivent également tester le code dédié à la gestion des erreurs
  - Définition de jeux de tests représentatifs
  - Comparaison à des résultats attendus

Les tests unitaires

- Le code dédié aux tests unitaires doit faire partie du code du module
  - Permet la ré-utilisabilité des tests en même temps que du code du module proprement dit
  - Facilite l'extensibilité des tests

Les tests unitaires

- Tout module `module.c` doit disposer d'au moins un fichier `module_test.c` contenant la procédure de test unitaire
  - Contient une fonction `main()`
  - Construit par la commande « `make test` »
  - Renvoie un code de succès « `exit (0)` » ou d'échec
    - Permet la conduite automatique des tests au moyen de scripts shell
- Procédure documentée dans le manuel de maintenance

Cas de structures pré-déclarer et de fonctions internes

- Il arrive qu'un module ne fasse que la pré-déclaration de « struct » et dispose de fonctions internes (non déclarées dans le .h)

module.h :

```
#pragma once

struct Module ;

struct Module *new_module() ;
void release_module(struct Module *) ;
void do_module(struct Module *,int i)
...
```

module.c

```
#include ''module.h''
struct Module{
.....
} ;

void compact_data(struct Module *) {
…}
struct Module *new_module() ;
void release_module(struct Module *) ;
.....
```

module_test.c

```
#include <module.h>
..... ?!? ...
```

header interne

module.h :

```
#pragma once

struct Module ;

struct Module *new_module() ;
void release_module(struct Module *) ;
void do_module(struct Module *,int i)
...
```

module.c

```
#include ”module_impl.h”


void compact_data(struct Module *) {
…}
struct Module *new_module() ;
void release_module(struct Module *) ;
…..
```

module_test_fonctionnel.c

```
#include <module.h>
……..
```

module_test.c

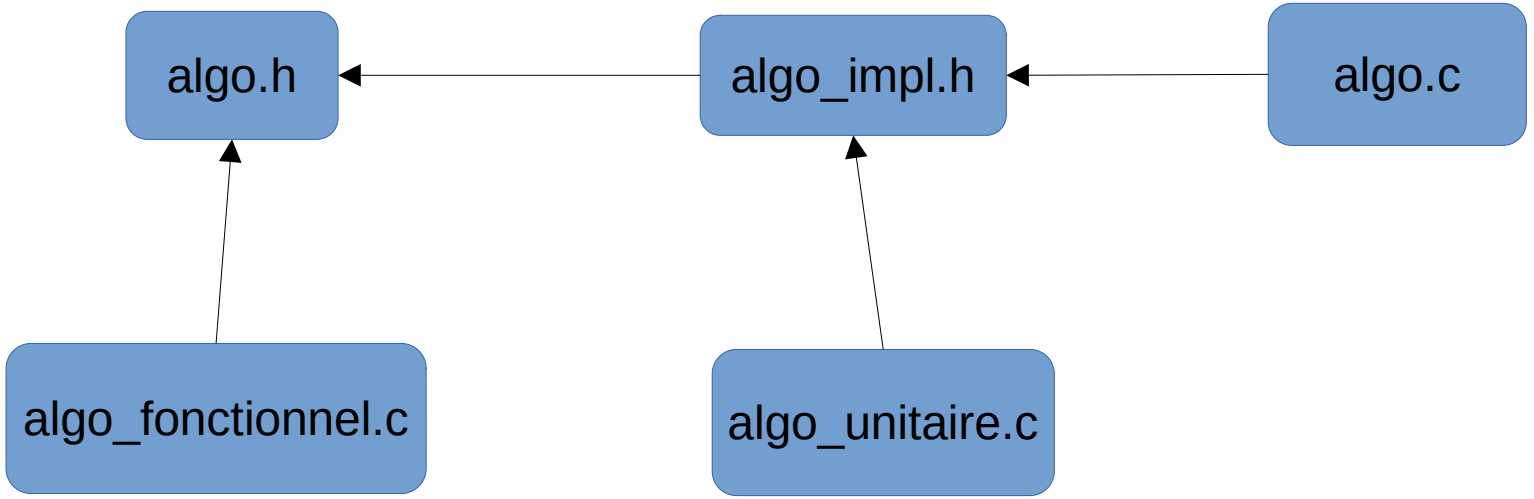
```
#include <module_impl.h>
……..
```

module_impl.h

```
#include ”module.h”
struct Module{
…..
} ;
void compact_data(struct Module *) ;
```

Unitaire vs fonctionnel

- le test unitaire va s'appuyer sur l'implémentation en plus de la spécification
- le test fonctionnel va uniquement se baser sur la spécification.



framework

- Il n’existe pas de framework standard pour l’écriture de tests en C
- Pour les premiers tests, on pourrait tout simplement utiliser la macro assert (#include<assert.h>) mais celle-ci est indiquée pour ajouter des vérifications exhaustives dans le code et pose problème.
- Chaque test prendra la forme d’une fonction, une fonction main appellera chaque fonction l’une après l’autre

exemple :

```
void test_array_malloc(){
      struct Array *p = array_malloc(5) ;
      assert(p!=NULL) ;
}
void test_array_malloc0(){
       struct Array *p=array_malloc(0) ;
      assert(p==NULL) ;
}
....
int main(){
     test_array_malloc() ;
     test_array_malloc0() ;
}
```

```
void test_array_malloc(){
      struct Array *p = array_malloc(5) ;
      if (p==NULL)
            exit(5) ;
}
void test_array_malloc0(){
       struct Array *p=array_malloc(0) ;
      if (p !=NULL)  exit(6) ;
}
....
int main(){
     test_array_malloc() ;
     test_array_malloc0() ;
}
```

test et assert

- On n'utilisera pas « assert » pour écrire des tests
- En effet, on veut pouvoir exécuter nos tests sur le code en mode debug et en mode non debug
- ⇒ n'ayant pas de framework standard, on pourra écrire nos propre macro :
  - RUN(t) : qui affiche le test puis l'exécute
  - CMP(a,b) : qui compare deux valeurs et stop le programme si elles diffèrent
  - STRCMP(a,b) : idem, mais pour les chaînes de caractères (strcmp)
  - ...

macros d’exemple

- En l’absence de « standard », on pourra écrire nos propres macros telles que :

```
#define RUN(t) fputs("\033[32mrunning\033[0m : "#t"\n",stderr); t
```

```
#define CMP_EQ(a,b) if (a!=b){  \
    fputs("\033[31merror\033[0m : "#a" != "#b" in ",stderr);    \
    fputs(__func__,stderr);                    \
    fputs("\n",stderr);                  \
    exit(EXIT_FAILURE); }                    \
  else fputs("\033[32mOK\033[0m : "#a" != "#b"\n",stderr);
```

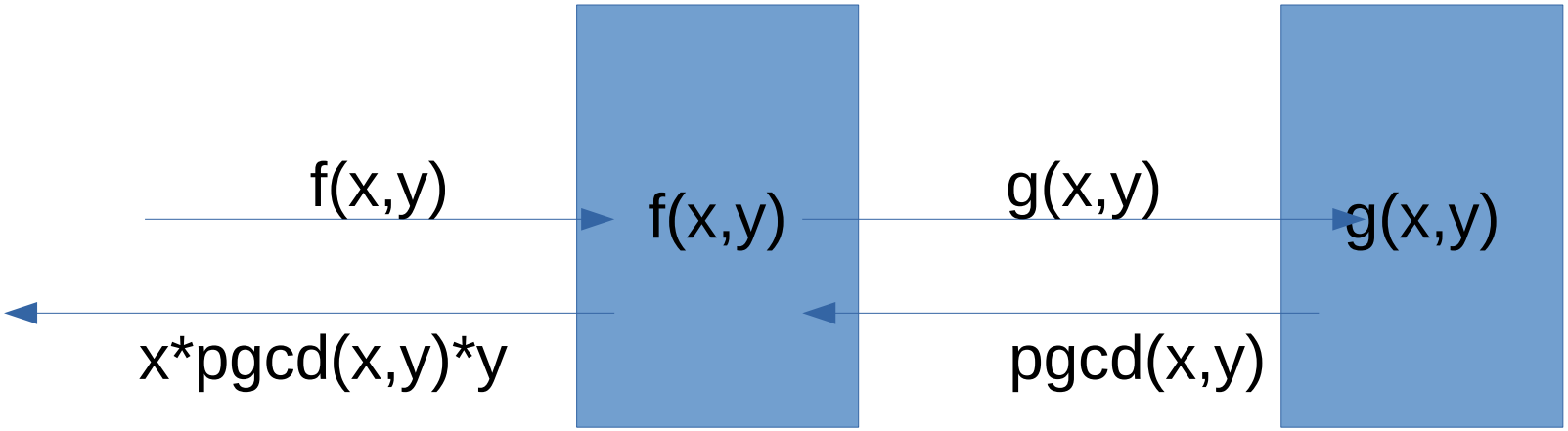
```
#define CMP_STR_EQ(a,b) if (strcmp(a,b)!=0){  \
    fputs("\033[31merror : "#a" == "#b" in ",stderr); \
    fputs(__func__,stderr);                  \
    fputs("\n",stderr);                \
    exit(EXIT_FAILURE); }                    \
  else fputs("\033[32mOK\033[0m : "#a" == "#b"\n",stderr);
```

Les tests unitaires : faussaires

- Dans le cas des tests unitaires, on souhaite que le tests concerne uniquement une fonction indépendamment des autres modules.
- Si le test plante, alors cela doit indiquer un bug dans la fonction correspondante.
- Que faire si une fonction ou un module repose sur un autre code ?

Tests inter module: faussaires

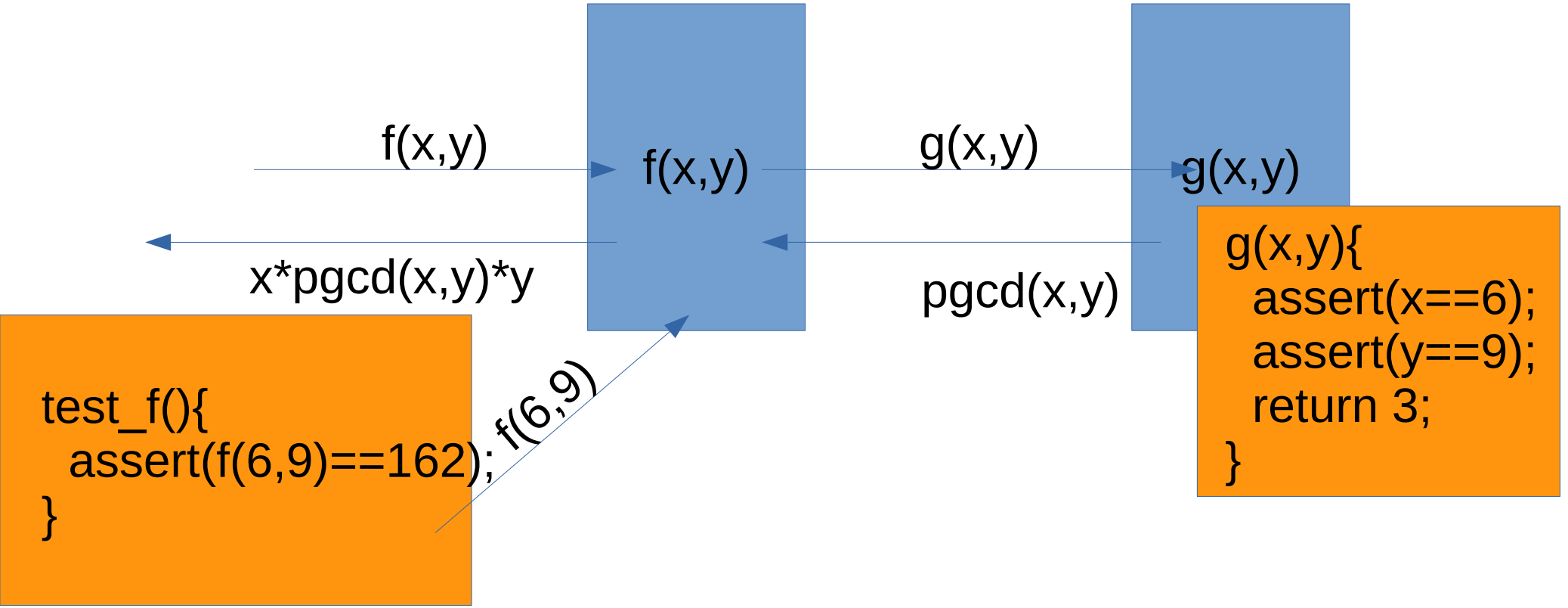
- Pour tester un module indépendamment d'un autre module



Pour tester f(), on écrit une fausse fonction g qui va tester
la valeur de ses paramètres et renvoyer une valeur pré-calculée

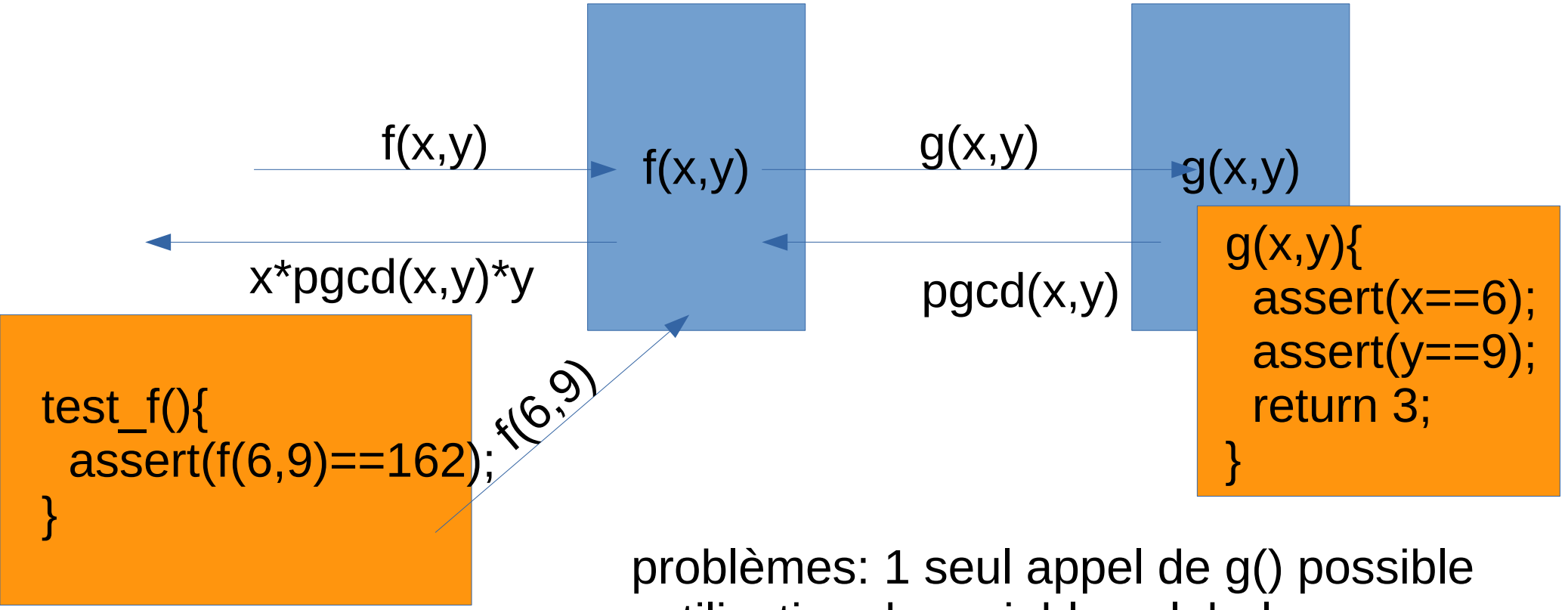
Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module



Tests inter module: faussaires

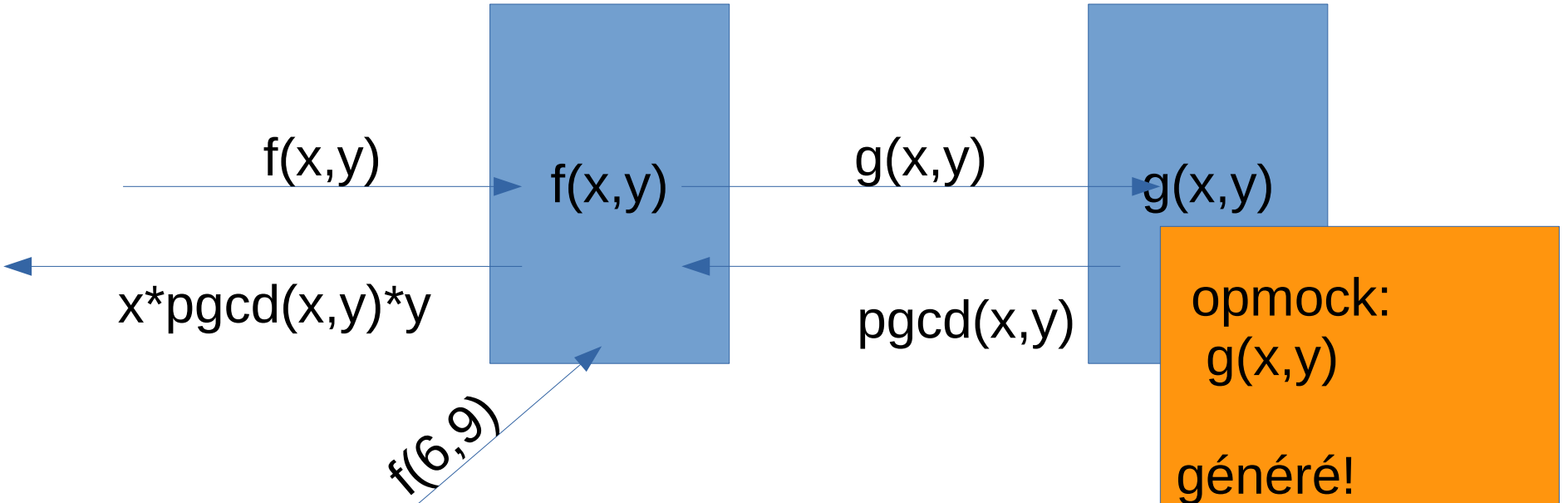
- Pour tester un module indépendamment d'un autre module



problèmes: 1 seul appel de g() possible
➔utilisation de variables globales
➔utilisation de générateur de mock (ex: opmock)

Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module




```
test_f(){
  g_ExpectAndReturn(6,9,3,comp_int,comp_int);
  assert(f(6,9)==162);
}
```

Les faussaires

- L'utilisation des faussaires (mock) en C n'est pas trivial, vous verrez que c'est beaucoup plus « naturel » dans d'autres langages supportant l'introspection (Java/python par ex…)
- Les faussaires sont aussi utiles pour paralléliser le développement : un développeur doit développer le module A et un autre le module B reposant sur A :
- Dans ce cas, on écrit une version faussaire du module A permettant d'entamer le développement du module B et parallèle du développement de A.

Tests d'implémentation

- Tout module `module.c` gérant un type `Module` doit contenir une méthode `moduleVerifie` destinée à vérifier la cohérence de l'instance de `Module` qui lui est passée en paramètre
  - Utile seulement s'il existe des conditions vérifiables
  - Sert à vérifier la cohérence des objets de type `Module` calculés par les méthodes du module
    - Assertion ou test en mode « debug »
    - Utilisation d'un drapeau « `MODULE_DEBUG` »
  - Mise à la disposition des tiers désireux d'étendre les fonctionnalités du module

Tests d'implémentation

```
int
matriceFaitQqch (
Matrice *     source,
Matrice *     destination,
int           paramètre)
{
...
#ifdef MATRICE_DEBUG                    /* Test de pré-condition     */
  if (matriceVerifie (source) != 0) { /* Test avec retour d'erreur */
    ...
    return (1);
  }
#endif /* MATRICE_DEBUG */
...
#ifdef MATRICE_DEBUG                      /* Test de post-condition */
  assert (matriceVerifie (destination) == 0);  /* Assertion (exit) */
#endif /* MATRICE_DEBUG */

  return (0);                                  /* On y est arrivé */
}
```

Tests d'implémentation

- Ces tests ne sont valables que pour des structures de données complexes
- Attention à ce que l'appel à ces tests n'induit pas d'effet de bord (modification de l'état de la structure).

Les tests d'intégration

- Les tests d'intégrations valident le fonctionnement global d'un module :
  - Cohérence des fonctions entre elles
  - Exécution de série d'opérations types
- Les tests d'intégrations peuvent également valider le fonctionnement de plusieurs modules entre eux
  - par exemple : couplage d'un module de graphe avec un module d'arbre

Tests d’acceptation / recette

- Ont pour but d'attester la validité du projet dans son ensemble
  - Mettent en œuvre des jeux de tests de taille réelle
  - Utilisés comme éléments contractuels pour la phase de recette du logiciel

Tests d’acceptation / recette

- Tout projet doit disposer d'un ou plusieurs fichiers contenant la procédure de tests de recette
  - Programmes ou scripts shell
- Procédure documentée dans le manuel de maintenance

Couverture, performance, non-régression

- Certaines caractéristiques peuvent être observées sur les tests (indépendamment de leurs types) :
  - Couverture : correspond au pourcentage du code effectivement exécuté par les tests. Permet de mettre en avant du code mort ou du code non testé
  - Non-Régression : correspond au fait qu'au fur et à mesure du développement, les tests passés continuent d'être validés
  - Performance : les tests permettent également de faire de la veille sur les ressources utilisées (CPU, mémoire…). Le suivi dans le temps permet de mettre en avant des dysfonctionnements

Couverture

- L'option –-coverage de gcc permet de générer des traces d’exécution

main.c:

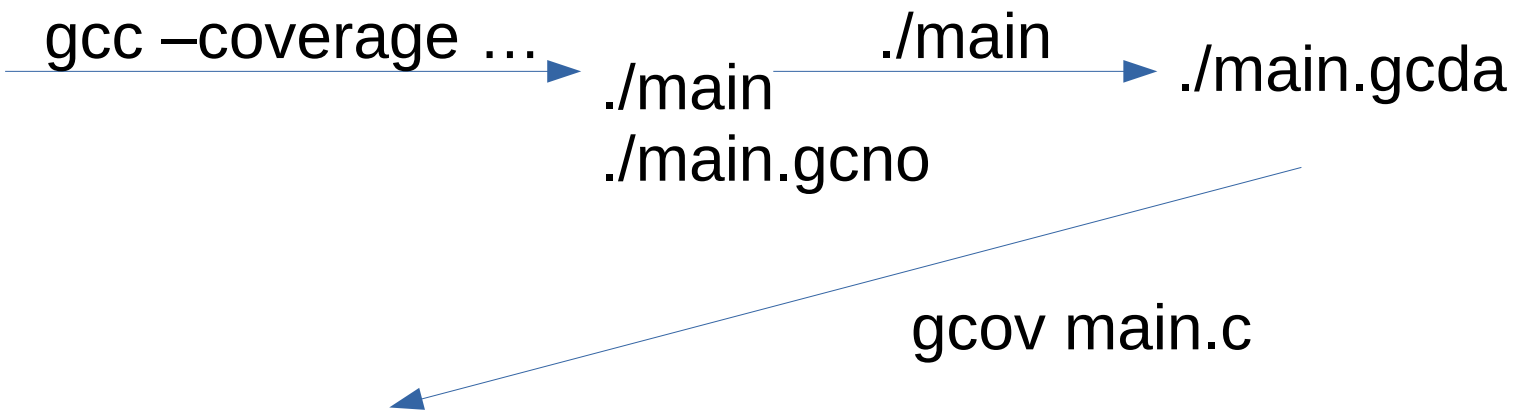
```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```

Couverture

- L'option –-coverage de gcc permet de générer des traces d'exécution

main.c:

```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```



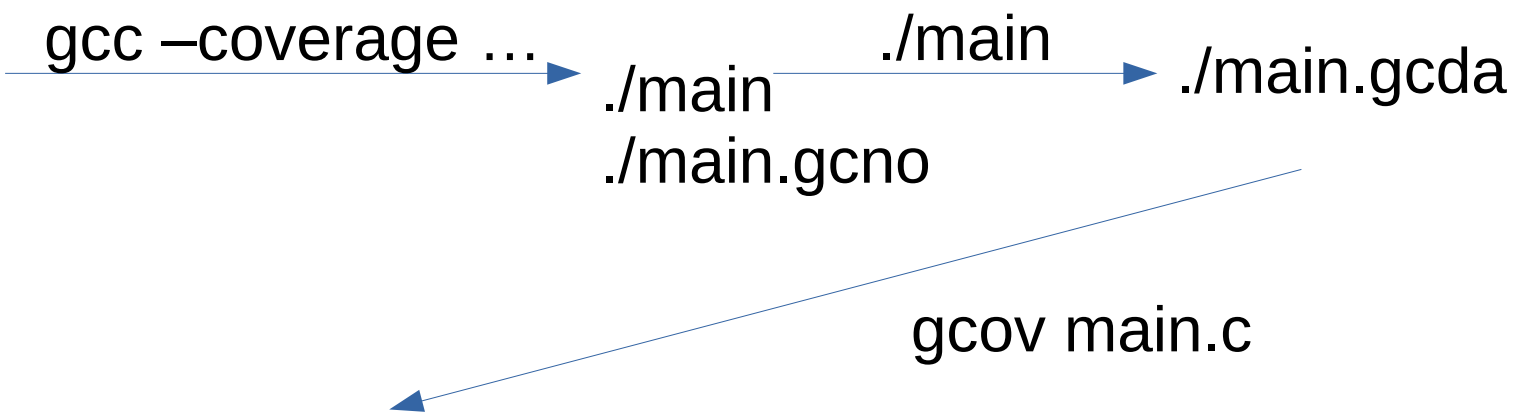

```
File 'main.c'
Lines executed:80.00% of 5
main.c:creating 'main.c.gcov'
```

Couverture

- L'option –-coverage de gcc permet de générer des traces d'exécution

main.c:

```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```



gcc --coverage

- L'option --coverage est une « meta » option de gcc, elle implique les options :
  - -fprofile-arcs et -ftest-coverage lors de la compilation
  - -lgcov lors de l'édition de lien
- Il est possible de compléter avec l'option

  -fprofile-abs-path pour permettre à gcov de retrouver plus facilement les sources associées au binaire.

Couverture

- La couverture est une propriété très intéressante à observer
- Pour les tests, elle permet de mettre à jour du code non testé ou inutile
- Permet également lors de tests de recette ou de bêta de voir simplement quelles parties sont les plus utilisées, quelles erreurs n'arrivent « jamais » ou « souvent »...

Programmation C avancée

Performance

Analyse de performance (1)

- La performance d'un logiciel est évaluée en fonction des ressources nécessaires à l'obtention du résultat demandé
  - Temps mis
  - Ressources consommées (mémoire centrale, espace disque, bande passante réseau, ...)

Analyse de performance (2)

- Lorsque la performance est insuffisante, il faut déterminer l'origine de cette insuffisance afin d'essayer d'y porter remède
  - Problèmes d'implémentation, pas de spécification
  - Évaluation sur des cas réels, après maquettage
- Les remèdes pourront être :
  - Matériels : ajout ou remplacement de ressources
    - Plus le temps passe, plus les matériels sont puissants !
  - Logiciels : recodage de routines critiques ou bien modifications profondes de la structure du logiciel

Analyse de performance (3)

- Lorsque le manque de performance concerne le temps, les problèmes d'accès à la mémoire en sont les causes principales
  - En moyenne, près de la moitié des cycles consommés par les processeurs sont des cycles d'attente de la mémoire
- Il est donc essentiel de concevoir les algorithmes afin de minimiser les attentes mémoire

Principes de localité

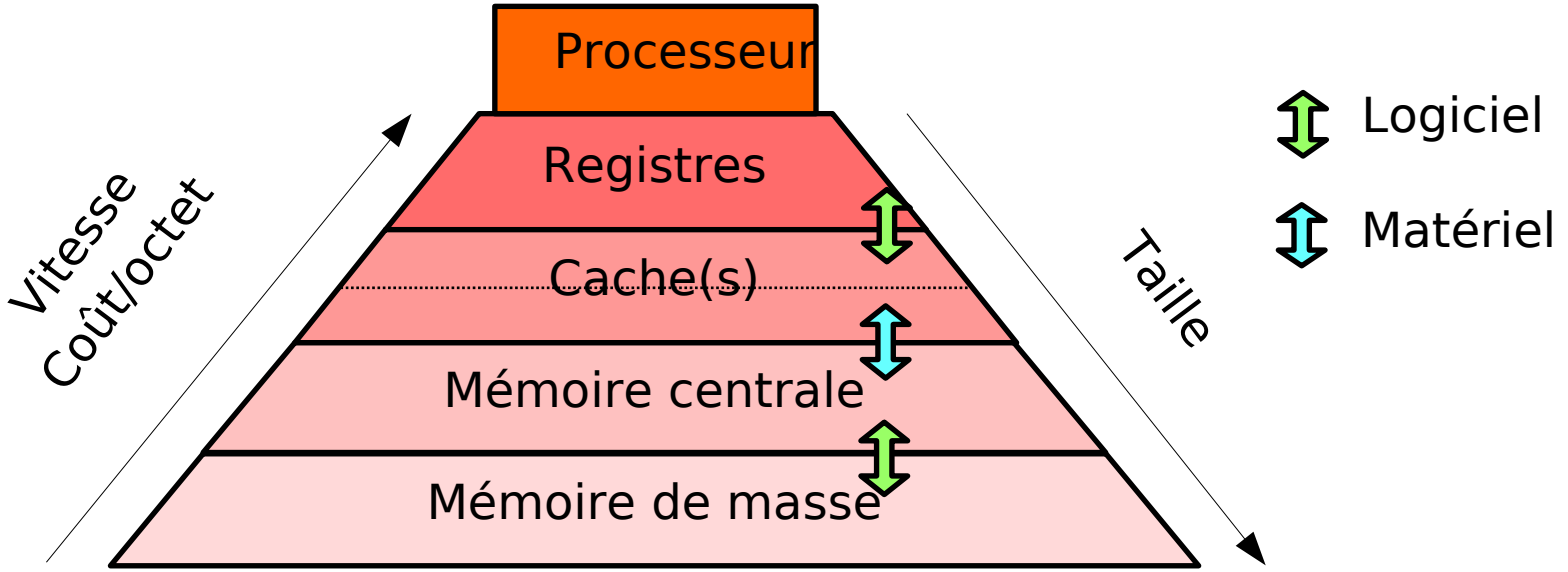
- Dans tout programme, il est possible de mettre en évidence un phénomène de localité des accès mémoire
  - Localité temporelle : plus une zone mémoire a été accédée récemment, plus sa probabilité de ré-accès est élevée
    - Lecture des instructions par le processeur (boucles), ...
  - Localité spatiale : plus une zone mémoire est proche de la dernière zone mémoire accédée, et plus la probabilité qu'elle soit à son tour accédée est importante
    - Parcours de tableaux, ...

Hiérarchie mémoire (1)

- Pour minimiser l'attente de la mémoire, il faut que les informations les plus fréquemment utilisées soient disponibles le plus rapidement possible
- On s'appuie sur les principes de localité pour mettre en place une hiérarchie de la mémoire
  - Mémoires rapides de faible capacité, proches du processeur
  - Mémoires de grande capacité aux temps d'accès plus longs, situées plus à distance

Hiérarchie mémoire (2)

- Met en œuvre les principes de localité
- Rend disponibles plus rapidement les données les plus fréquemment utilisées



- Les algorithmes doivent s'appuyer dessus !

Mémoires cache (1)

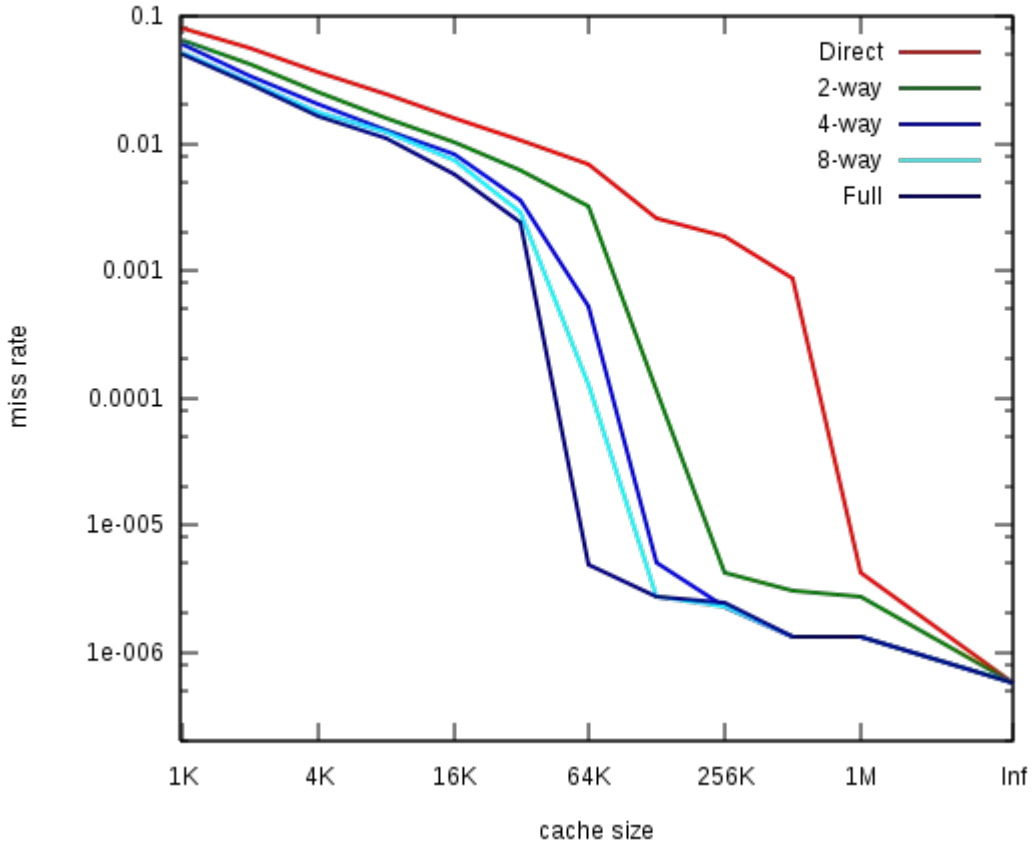
- Mémoire(s) rapide(s) située(s) entre le processeur et la mémoire centrale
- On trouve habituellement plusieurs niveaux de cache sur les architectures modernes
  - Cache de premier niveau
    - Sur le chip du processeur lui-même
    - Quelques dizaines de Ko seulement
    - Caches dissociés pour les instructions et les données
  - Caches de second/troisième niveau
    - Dans le boîtier du processeur ou à proximité
    - De quelques centaines de Ko à quelques Mo ; 2008: intel i7 8Mo; 2017: Intel Core i9 9900K 512K – 2Mo – 16Mo

      2020 : AMD Ryzen 9 5900X: 12 x (64k - 512K) - 64Mo

Mémoires cache (2)

- Les caches ne manipulent pas des octets ou des mots, mais des lignes (« *cache lines* »)
  - Lorsque le processeur demande un mot mémoire, qui n'est pas déjà présent dans le cache, le cache charge toute la ligne contenant le mot voulu à partir de la mémoire (de 8 à 512 octets)
  - Mise en œuvre du principe de localité spatiale
- Lorsque le cache est plein, la ligne la plus ancienne est effacée pour faire de la place à la nouvelle
  - Politique de remplacement de type LRU

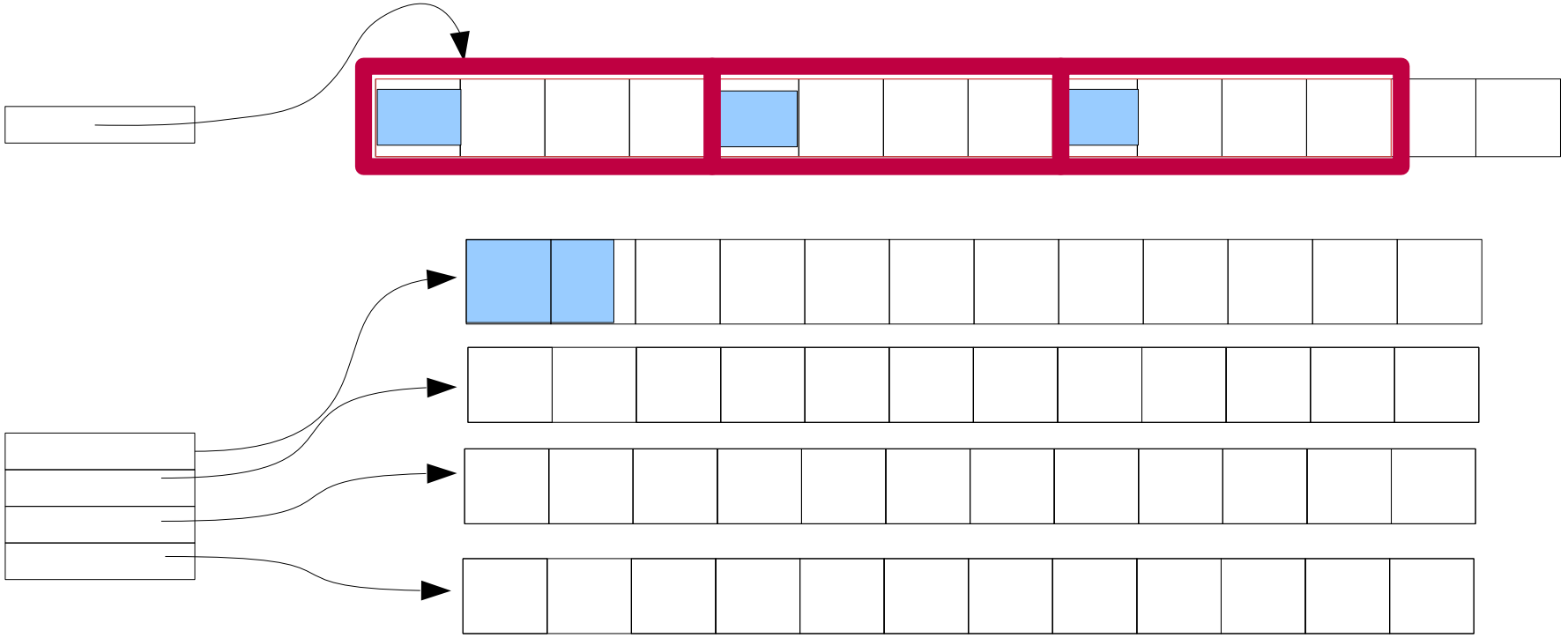
Cache: Taille vs. Fautes



benchmark: SPEC CPU2000

Programmation « *cache friendly* »

- Pour qu'un algorithme soit efficace sur une architecture disposant d'une hiérarchie mémoire, il faut :
  - Effectuer la majorité des parcours de façon continue
    - Croissante ou décroissante
    - Maximise l'utilisation des données des lignes nouvellement chargées
  - Effectuer les parcours en écriture de façon continue
    - Les écritures sont plus chères car on doit répercuter les modifications en mémoire centrale

Programmation « *cache friendly* »

- Manipuler plusieurs tableaux de sous-structures plutôt qu'un unique tableau de structures de grandes tailles
  - Les lignes de cache ne stockent que des données utiles