PG120:
Outils pour la programmation C
6 séances de 2h en cours intégrés

Julien Allali / allali@enseirb-matmeca.fr

Les tests

- L’objectif d’un test est de dérouler un scénario et de vérifier que le comportement est conforme à ce qui était prévu.
- On peut écrire plusieurs type des tests :
  - au niveau d’un fonction : tests unitaires
  - d’un module : tests d’intégration
  - entre plusieurs modules : tests fonctionnels
  - sur un programme : tests de recette
- La qualité des tests dépendra de la richesses des cas testés

Les tests unitaires

- Ces tests sont « simples » à écrire : ils doivent vérifier le comportement d'une seule fonction
- On testera plusieurs cas, dont les cas limites : pointeur à NULL, paramètre entier à -1 ou INT_MAX (dans limits.h)
- Chaque test prend la forme d'une fonction et devra tester un comportement. On pourra donc avoir plusieurs fonctions tests qui testeront la même fonction.

les tests d’intégrations

- Le deuxième type de tests que l’on va écrire sont les test d’intégration
- Ce test travaille sur un module (un fichier .h par exemple) et vérifie des scénarios
- On essayera de faire des tests exhaustifs qui utiliseront au plus le code du module

framework

- Il n’existe pas de framework standard pour l’écriture de tests en C
- Pour les premiers tests, on peut tout simplement utiliser la macro assert (#include<assert.h>) mais celle-ci est plutôt indiquée pour ajouter des vérifications exhaustives dans le code.
- 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)
  - ...

spécification

- Le test doit reposer sur la spécification de la fonction (que fait array_malloc si le paramètre est ≤ 0 ?)
- Il arrive de devoir préciser ces spécifications (et revoir l'implémentation) lors de l'écriture des tests.
- Les spécifications de la fonction doivent être dans le header : on utilisera pour cela les commentaires en suivant le format Doxygen

Cas particulier

- Comment faire si une structure est juste pré-déclarée dans le .h ? ⇒ On peut créer un deuxième .h :

dico.h :

```
#ifndef DICO_H
#define DICO_H

struct Dico ;
…
#endif
```

dico_impl.h :

```
#ifndef DICO_IMPL_H
#define DICO_IMPL_H
#include ''dico.h''
struct Dico {
   char **data ;
   int size ;
   …..
} ;
…
#endif
```

dico.c :

```
#include ''dico_impl.h''
….
```

dico_test.c :

```
#include ''dico_impl.h''
….
```

prog.c :

```
#include <dico.h>
….
```

Automatisation

- Dans le Makefile, on ajoute une règle **tests** qui va compiler les différents programmes de tests et les lancer

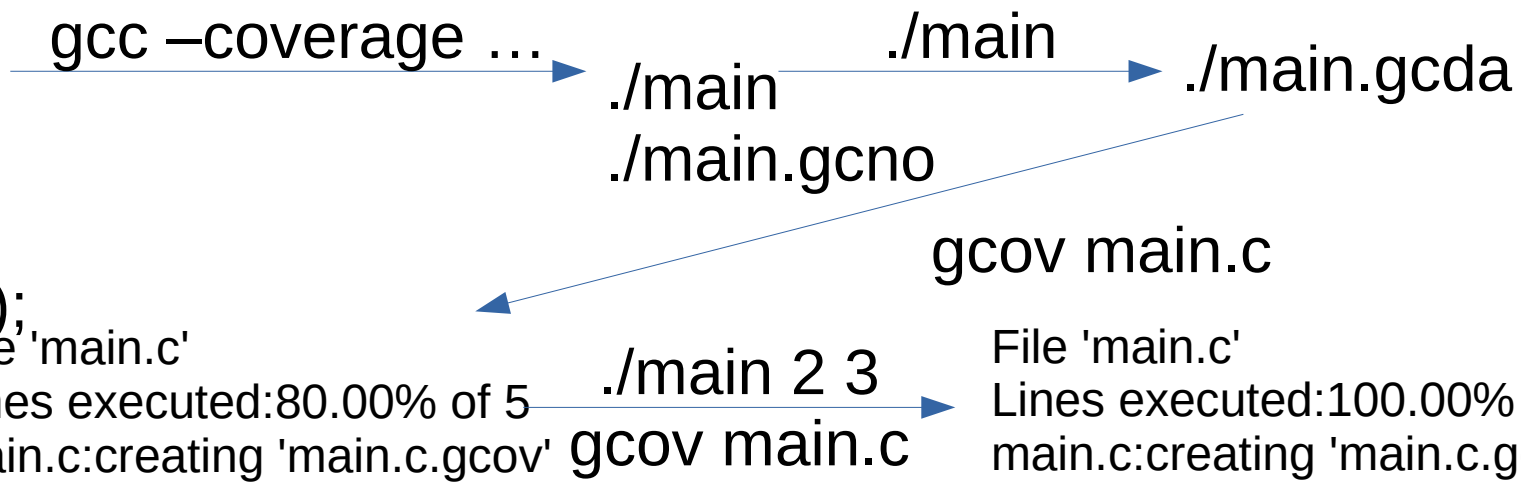
Tests

- Travailler dans une branche, on la fusionnera à la fin
- Commencer par écrire un fichier ritest.h qui contiendra les macros que vous utiliserez pour vos tests : RUN(t) pour lancer un test, CMP(a,b) qui compare deux valeurs et interrompt le programme (on utilisera __func__ pour afficher le nom du test) et CMP_STR pour les chaînes de caractères. Vous écrirez un petit fichier de test ritest_test.c
- Créer les fichiers d'entête array.h et array_impl.h, la structure sera juste pré-déclarée dans array.h
- Déplacer le main de array.c dans un fichier array_example.c
- créer un fichier array_test.c dont le but sera de tester le module array.c
- Écrire des tests pour les fonctions du modules : revoir le code d'implémentation du module si cela est nécessaire.
- Dans le Makefile, ajouter la règle pour compiler array_test ainsi qu'une règle **tests** pour exécuter les tests.

couverture

- Lorsque l'on compile du code avec l'option « --coverage », le compilateur ajoute du code permettant de tracer celui-ci

```
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 »...

Couverture

- Recompiler le module array.c avec l’option de couverture
- Compiler votre test et exécuter celui-ci
- Afficher la couverture avec **gcov**