Test et débogage

1. **Besoin de tester des programmes**
2. **Test et différents classification**
3. **Mise en œuvre des tests unitaires**
3. **Débogage**

Exemple de programme

```
/* Calcul de la racine carrée d'un nombre.
 * param : x le nombre
 * return : y la racine carrée de n
*/

double sqrt(double x) ;
```

La question

- L'exécution est-elle conforme au comportement attendu ?

Erreurs possibles :

- erreur lors de la conception (algorithme ou spécification logiquement faux)
- erreur lors de la réalisation (faute de frappe, mauvaise compréhension du langage de programmation)
- etc

Au fait, pour notre exemple, la réponse est non : il faut vérifier la conformité de l'exécution de la fonction par rapport aux valeurs d'entrées.

Le moyen

- Pour tester la conformité de l'exécution, une seule solution possible : **tester le code exécuté**

- Quel est le problème ?
  - Avant l'appel, vérifier que la valeur est strictement positive,
  - Après l'appel, vérifier que le résultat est correct

- Dans certains cas, le nombre de configurations à tester peut être très important.

- Il va donc falloir sélectionner des données de test pertinentes, mettant à mal l'implémentation.

La mise en oeuvre

Une fois les données de test sélectionnées, comment passer à la mise en œuvre des tests ?

- Disposer d'un cadre adapté à l'écriture de test
  - comparaison obtenu / attendu

- Disposer d'un cadre adapté à l'exécution des tests
  - pouvoir grouper un ensemble de test et les jouer de façon automatique.

Erreur vs. Bogue

- **Un bogue** :
  - C'est un défaut de programmation
  - Ex. : diviser par 0, boucle infinie, trop de ressources consommées, etc

- **Une erreur** :
  - C'est une donnée ou un évènement non désirée
  - Ex : donnée invalide, plus de mémoire, fichier corrompu, etc

- **Attention** :
  - Une erreur non traitée est un bogue

Bogues fréquents en C

- Mauvais cast (ou cast implicite) : perte de précision
  - `(float)(x/y)` au lieu de `(float)x)/((float)y)`
- Accès en dehors des tableaux : en particulier le `'\0'` des chaînes.
- Erreur de format `printf` et `scanf`
- Variable/mémoire non initialisée
- Confusion entre = et ==
- Ne pas traiter le code de retour d'une fonction
- Boucle infinie
- etc

Test d'un programme

- Définition 1 (G. J. Myers) : Tester, c'est exécuter un programme dans le but de constater des bogues et de trouver des erreurs.

- Définition 2: Tester consiste à vérifier que le comportement d'un programme est conforme au comportement attendu. Cette vérification est faîte sur un ensemble fini de cas de test, sélectionnés de façon adéquate parmi l'ensemble potentiellement infini des cas possibles.

- Définition 3 : Le test est un moyen d'assurer la qualité des logiciels, en vérifiant sur un ensemble de cas pertinents (car potentiellement capables de mettre à jour une faute) que le comportement du code exécuté est conforme au comportement attendu.

==> **Difficulté** : choisir quoi tester et quand.

==> Réussir tous les tests ne prouve pas l'absence de bogues !

Niveaux de tests

- Tests unitaires :

  Les tests unitaires visent à vérifier le fonctionnement de portions de code indépendamment des programmes qui les utilisent.

  - Exemples d'unité de test : une fonction, un module, etc

- Tests d'intégration :

  Les tests d'intégration visent à vérifier le fonctionnement de la composition de deux portions de codes (deux unités de test) interagissant l'une avec l'autre.

  - Exemple de test d'intégration : intégration d'une unité d'affichage d'une table et d'une unité de traitement de table (tri, ajout, retrait, sélection, etc.)

- Tests systèmes :

  Les tests systèmes visent à vérifier le comportement du système complet :

  - test des performances, d'intégration avec l'environnement d'exploitation (Hardware, OS, librairies, etc.)

Classification selon les objectifs

- Tests de conformité :

  Les tests de conformité ont pour but de vérifier la conformité entre le comportement à l'exécution et le comportement spécifié.

- Tests de régression :

  Les tests de régression ont pour but de vérifier que des modifications/évolutions n'ont pas eu d'effets négatifs inattendus, en montrant que les tests qui passaient jusqu'alors et non impactés par l'évolution/modification testée, passent toujours.

- Tests d'acceptation :

  Les d'acceptation ont pour but de valider le développement complet du logiciel. Il s'agit de vérifier que le logiciel peut être utilisé pour mener à bien un certain nombres de scénarios, fixés par le client.

Types de tests

- Boite noire : Le but est de vérifier que les spécifications définies sont vraies sur des exemples :
  - On ne connaît que l'interface
  - On teste les interfaces : telle entrée donne tel résultat
  - On teste en général les données limites ou particulières

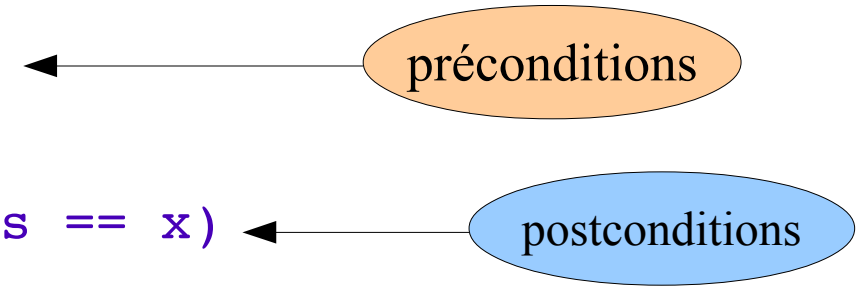
- Exemple de tests aux limites pou la fonction **`int nbJoursDansMois(int mois, int annee)`**
  - limite pour le paramètre mois :
  - limite pour le paramètre année :

- Boite blanche : Le but de ce test est d'exécuter chaque branche du code avec différentes conditions d'entrée afin de détecter tous les comportements anormaux.
  - On connaît les détails de l'implémentation
  - On prend en compte la structure du code

Mise en œuvre des tests unitaire (1)

- Les assertions : l'assertion est une technique de détection des erreurs de programmation :
  - La fonction **`assert`** teste une condition et avorte le programme
  - Fait partie de la librairie `assert.h`
- Exemple : retour sur la fonction sqrt

```
double sqrt(double x) {
        double res ;
        assert(x>0) ;
        /* ... */
        assert(res*res == x)
        return res ;
}
```



- Preconditions specify and check the input conditions to a function
- Postconditions specify and check the output conditions of a function

Mise en œuvre des tests unitaire (2)

```c
#include <assert.h>

int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    assert(add(12,10) == 22);

   /* bad assertion */
    assert(add(12, 12) == 25);

    return 0;
}
```

Mise en œuvre des tests unitaire (3)

Outils dédiés : des frameworks nous permettent d'écrire « facilement » des tests unitaires :

- `C` `CUnit`
- `Java` `Junit`
- `Php` `PHPUnit`

Cunit (1/4)

Cunit :

- Framework permettant d'administrer, d'écrire et d'exécuter les tests unitaires de programmes écrits en C.
- Fournit un ensemble d'assertions assurant le test des conditions logiques.

Organisation des tests :

- Un programme test Cunit est composé d'un ***registre*** de tests
- Un registre de test est composé d'une ou plusieurs ***suites*** (**CU_pSuite**) de tests
  - Chaque suite de tests est encadrée par une fonction d'initialisation et une fonction de nettoyage, toute de deux de type int *f(void)

Fonction test : void <Nom_fonc_test>(void)

Il n'y a aucune restriction sur le contenu de la fonction test, sauf qu'elle ne doit pas modifier les paramètres de la plate-forme CUnit. Par exemple, elle ne doit pas ajouter des test, modifier le registre de test ou initialiser un test. Elle peut appeler d'autres fonctions.

Cunit (2/4)

Structure type d'un programme Cunit :

(1) Initialiser le registre de tests : CU_initialize_registry(void)

(2) Ajouter une suite de tests au registre : CU_add_suite()

(3) Ajouter des tests à une suite de tests : CU_add_test()

(4) Lancer les tests avec l'interface appropriée

- Mode non interactive : CU_basic_run_tests(void)
- Mode interactive : CU_curses_run_tests(void)

(5) Nettoyer le registre de tests : CU_cleanup_registry();

Structure d'un test :

- Fonction de type **void <Nom_fonc_test>(void)** qui utilise des assertions :
  - macro instructions commencant par **CU_ASSERT** (définies dans Cunit/CUnit.h) :
    - **CU_ASSERT_TRUE, CU_ASSERT_FALSE, CU_ASSERT_EQUAL, CU_ASSERT_NOT_EQUAL, CU_ASSERT_PTR_EQUAL**

Cunit (3/4)

Récupération des résultats: Le code du testeur pourrait avoir besoin des résultats intermédiaires grâce à aux fonctions de « Cunit/TestRun.h » :

- unsigned int CU_get_number_of_suites_failed(void)
- unsigned int CU_get_number_of_tests_run(void)
- unsigned int CU_get_number_of_tests_failed(void)
- unsigned int CU_get_number_of_asserts(void)
- unsigned int CU_get_number_of_successes(void)
- unsigned int CU_get_number_of_failures(void)

Gestion des erreurs(Cunit/CUError.h): Cunit propose plusieurs fonctions permettant de repérer différentes erreurs (erreur d'initialisation de registre, duplication du nom d'une suite de test, etc) :

- CU_ErrorCode CU_get_error(void)
- char* CU_get_error_msg(void)

Cunit (4/4)

```
include "CUnit/CUnit.h"
#include "CUnit/Basic.h"
#include <stdio.h>

int max (int n1, int n2 )
{ if (n2 > n1) return n2;
   return n1;
}

//Test case functions
void max_test_1(void) {
  CU_ASSERT_EQUAL( max(1,2), 2);
  CU_ASSERT_EQUAL( max(2,1), 2);

}
void max_test_2(void) {
  CU_ASSERT_EQUAL( max(2,2), 2);
  CU_ASSERT_EQUAL( max(0,0), 0);
  CU_ASSERT_EQUAL( max(-1,-1), -1);
}
void max_test_3(void) {
  CU_ASSERT_EQUAL( max(-1,-2), -1);
}
```

Cunit (4/4)

```
// Test Runner Code goes here
int main(void){
    CU_pSuite pSuite = NULL;
/** initialize the CUnit test registry **/
   if (CUE_SUCCESS != CU_initialize_registry())
      return CU_get_error();

/** add a suite to the registry **/
   pSuite = CU_add_suite("max_test",NULL,NULL);
   if (NULL == pSuite) {
      CU_cleanup_registry();
      return CU_get_error();
   }

/** add the tests to the suite **/
   If ( (NULL == CU_add_test(pSuite,"max_test_1",max_test_1))||
        (NULL == CU_add_test(pSuite,"max_test_2",max_test_2))||
        (NULL == CU_add_test(pSuite,"max_test_3",max_test_3))
      )
   {
      CU_cleanup_registry();
      return CU_get_error();
   }
```

Cunit (4/4)

```
// Run all tests using the basic interface

    CU_basic_set_mode(CU_BRM_VERBOSE);

    CU_basic_run_tests();

    printf("\n");

    CU_basic_show_failures(CU_get_failure_list());

    printf("\n\n");

/* Clean up registry and return */
   CU_cleanup_registry();
   return CU_get_error();

}
```

Débogage

- **Processus** :
  - Trouver le bogue
  - Trouver la cause du bogue
  - Corriger le bogue

- **Problème : c'est dur !!**
  - Chercher une aiguille dans un botte de foin
  - Le symptôme peut être très éloigné du bogue réel

Techniques de débogage

- **Lecture du code** :
  - Inefficace si l'auteur == le lecteur
  - Inefficace car l'esprit voit ce qu'il veut voire

- **Execution instrumentée** :
  - Ajouts d'assertions
  - Ajout de `printf`

- **Exécution contrôlée** :
  - Utilisation de débogueur
  - Permet d'exécuter pas-à-pas, de mettre des points d'arrêts, de surveille ou modifier des variables en temps réel
  - Exemple : gdb
  - La plupart des environnements de développement ont des débogueurs intégrés.