Cours PRG2

Programmation C

C10 *(2024_1)*

Contenu

- Gestion des erreurs
- Gestion des exceptions
- Tests unitaires

Gestion des erreurs (1/4)

- Variable **`errno`** et fonction **`perror()`**

`#include <errno.h>`

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

int main() {
    int *ptr;
    size_t size = 1000000000000;

    // Tentative d'allouer une quantité massive de mémoire
    ptr = (int *)malloc(size * sizeof(int));
    if (ptr == NULL) {
        printf("Erreur numéro %d\n", errno);
        perror("Erreur lors de l'allocation de mémoire");

        exit(EXIT_FAILURE);
    }
}
```

Référence à la variable globale `errno`

```
Erreur numéro 12
Erreur lors de l'allocation de mémoire: Cannot allocate memory
```

```c
int main() {

    FILE *fp = fopen("fichier.txt", "r");

    if (fp == NULL) {

        if (errno == ENOENT) {
            printf("Fichier non trouvé\n");
        } else {
            printf("Erreur inattendue\n");
        }

        perror("Erreur lors de l'ouverture du fichier");
    }
}
```

```
Fichier non trouvé
Erreur lors de l'ouverture du fichier: No such file or directory
```


La variable `errno` (propre à chaque contexte d'exécution) est accessible partout dans le code ; il s'agit d'une macro dont la définition est déclarée dans le fichier *include* `errno.h`. La fonction `perror()` quant à elle est également déclarée dans ce fichier d'entête.

La gestion des erreurs avec les fonctions de bibliothèque repose sur la vérification des valeurs de retour, l'utilisation de `errno` pour identifier les erreurs spécifiques et l'utilisation de fonctions comme `perror()` pour **afficher des messages d'erreur informatifs**. Cette approche permet de détecter et de gérer efficacement les erreurs lors de l'utilisation des fonctions de la bibliothèque standard.

L'argument de `perror()` est une chaîne de caractères qui sera concaténée au message prédéfini associé au code d'erreur.

Gestion des erreurs (2/4)

- Codes d'erreur prédéfinies

```
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO          5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
/* ... */
```

Les messages d'erreur sont définis dans la `libc`

```c
int main() {
    FILE *fichier;
    int *tableau;

  fichier = fopen("mon_fichier.txt", "r");
  if (fichier == NULL) {
    if (errno == ENOENT) {
      printf("'mon_fichier.txt' n'existe pas.\n");
    } else {
      perror("Erreur ouverture fichier");
    }
  } else {
    fclose(fichier);
  }

  tableau = malloc(100000000 * sizeof(int));
  if (tableau == NULL) {
    if (errno == ENOMEM) {
      printf("Mémoire insuffisante.\n");
    } else {
      perror("Erreur allocation mémoire");
    }
  } else {
    free(tableau);
  }

  fichier = fopen("fichier_lecture_seule.txt", "w");
  if (fichier == NULL) {
    if (errno == EACCES) {
      printf("Ce fichier est en lecture seule.\n");
    } else {
      perror("Erreur ouverture fichier");
    }
  } else {
    fclose(fichier);
  }

  return 0;
}
```


Pour faciliter la gestion des erreurs, la bibliothèque `errno` propose des codes d'erreur prédéfinis sous forme de `#define` comme présenté ci-dessus ; la liste n'est de loin pas complète (il s'agit uniquement des codes de base). Ces codes sont associés à des messages d'erreur explicites qui décrivent la nature de l'erreur. En utilisant ces codes d'erreur, il est possible d'afficher des messages d'erreur clairs et informatifs aux utilisateurs, leur permettant de mieux comprendre l'origine du problème.

Gestion des erreurs (3/4)

- Constantes **NAN** (*Not a Number*)
- Utilisée pour indiquer une erreur de calcul

```c
#include <math.h>
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#define NAN /*implementation defined*/

int main(void)
{
    const double f = NAN;
    uint64_t fn;

    memcpy(&fn, &f, sizeof f);
    printf("NAN:   %f %" PRIx64 "\n", f, fn);
}
```

Défini dans `math.h`

⇨ `NAN:   nan 7ff8000000000000`

nan, nanf, nanl, nand32, nand64, nand128

Defined in header <math.h>

float	nanf(const char* arg);	(1)	(since C99)
double	nan(const char* arg);	(2)	(since C99)
long double	nanl(const char* arg);	(3)	(since C99)
_Decimal32	nand32(const char* arg);	(4)	(since C23)
_Decimal64	nand64(const char* arg);	(5)	(since C23)
_Decimal128	nand128(const char* arg);	(6)	(since C23)

Exemple :

```c
float diviser_flottants(float a, float b) {
 if (b == 0.0f) {
   return NAN;
 }

 float resultat = a / b;

 return resultat;
}
```


Bien que les nombres flottants soient essentiels dans de nombreux domaines, ils peuvent également être sujets à des erreurs et des imprécisions. C'est pourquoi la gestion des erreurs flottantes est un aspect crucial de la programmation en C.

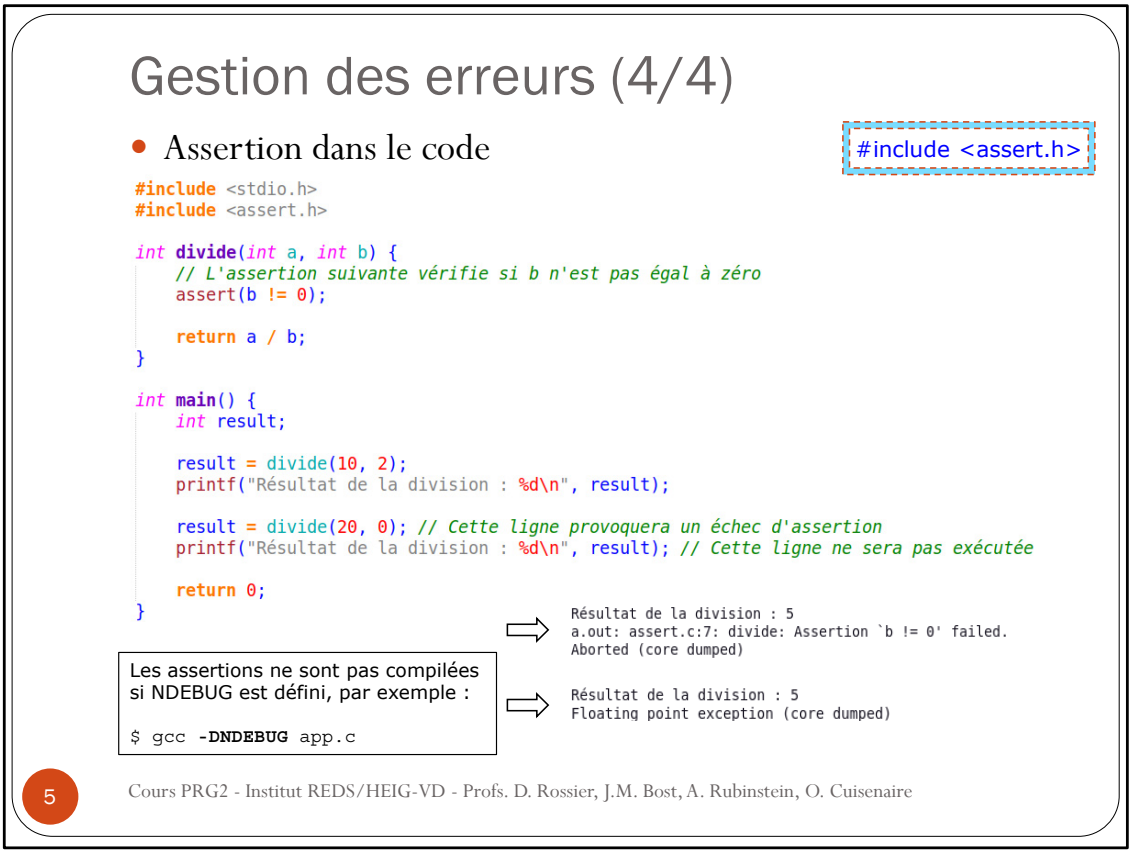
Les erreurs fréquentes avec les opérations sur les nombres flottant sont la division par 0, le dépassement de capacité et les imprécisions d'arrondis.

C'est pourquoi, les fonctions opérant sur les nombres flottant peuvent retourner une valeur qui n'est pas à considérer comme un nombre, mais comme un état d'erreur. C'est le cas de la constante "Not a Number" (NaN) ou **NAN**, ainsi que les fonctions de la même famille (`nanf`, `nanl`, `nand32`, `nand64`, etc.).

De même, le résultat d'un calcul peut être trop grand pour être stocké dans le type de données flottant choisi, ce qui entraîne une perte de précision ou une valeur incorrecte.

Les nombres flottants sont représentés avec un nombre limité de bits, ce qui peut conduire à des erreurs d'arrondi lors des calculs complexes.

La librairie `math` fourni des fonctions comme `isnan()` et `isinf()` pour identifier respectivement les valeurs `NaN` et infinies.



En langage C, les assertions représentent un outil précieux pour vérifier le bon fonctionnement du code et détecter d'éventuelles erreurs dès le stade du développement. Elles permettent d'exprimer des conditions qui, si elles ne sont pas satisfaites, indiquent un problème dans le programme.

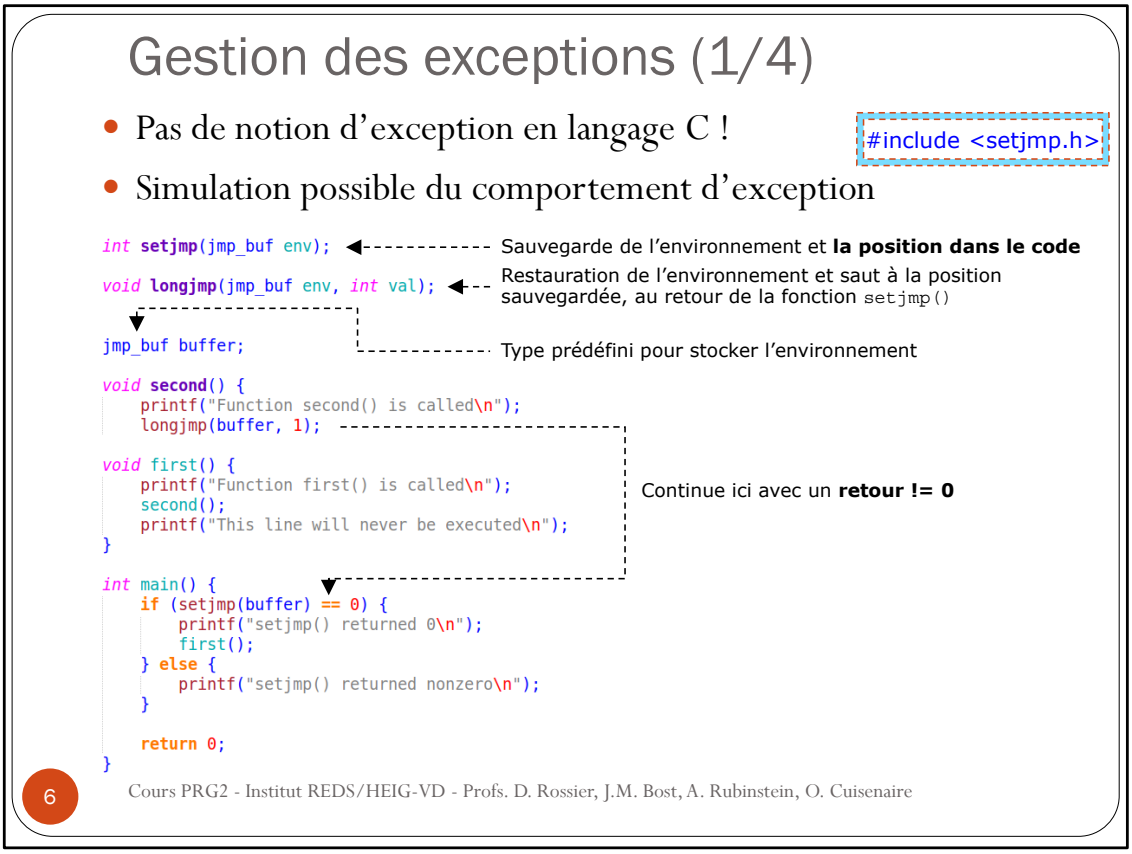
L'assertion la plus commune est implémentée par la macro `assert()`. Elle prend en paramètre une expression booléenne. Si l'expression est évaluée à ***`false`***, l'assertion **échoue et le programme s'arrête**, affichant un message d'erreur indiquant le fichier, la ligne et l'expression en cause.

Les assertions sont particulièrement utiles pour vérifier les préconditions et lever une erreur si elles ne sont pas satisfaites. Elles permettent également d'identifier des **erreurs de logique** dans le code qui pourraient passer inaperçues lors de l'exécution normale.

En plaçant des assertions à des points stratégiques du code, il est possible de détecter les cas où le programme se comporte de manière inattendue.

Pa ailleurs, les assertions ajoutent des commentaires explicites au code, ce qui peut améliorer sa lisibilité et sa compréhension pour les autres développeurs.

Il est important de noter que les assertions ne sont généralement pas activées dans les versions finales des programmes, car elles peuvent ralentir l'exécution. Elles sont principalement utilisées pendant le développement et la phase de test pour identifier et corriger les erreurs.



La simulation d'exceptions en C est possible en utilisant les fonctions `setjmp()` et `longjmp()` (de la librairie *setjmp*). Cependant, cette approche est moins flexible et moins puissante que les mécanismes de gestion des exceptions intégrés dans d'autres langages.

La fonction `setjmp()` prend un pointeur vers une structure `jmp_buf` comme argument et retourne une valeur entière. La structure `jmp_buf` sert à stocker l'état d'exécution du programme au moment où la fonction est appelée. La valeur de retour de *setjmp()* est 0 si la fonction est appelée pour la première fois, ou une valeur non nulle si elle est appelée dans un bloc `longjmp()` .

La fonction *longjmp()* prend un pointeur vers une structure *jmp_buf* et une valeur entière comme arguments. Elle restaure l'état d'exécution du programme à celui qui était stocké dans la structure *jmp_buf* et saute à l'instruction suivante de la fonction où *setjmp()* a été appelée. La valeur entière passée à *longjmp()* peut être utilisée pour transmettre des informations supplémentaires sur l'exception.

Gestion des exceptions (2/4)

- Exemple d'utilisation

```c
#include <stdio.h>
#include <setjmp.h>

jmp_buf buffer;

void divide(int x, int y) {
    if (y == 0) {
        printf("Division by zero error!\n");
        longjmp(buffer, 1);
    }
    printf("Result of division: %d\n", x / y);
}

int main() {
    int numerator, denominator;

    if (setjmp(buffer) == 0) {

        printf("Enter numerator: ");
        scanf("%d", &numerator);

        printf("Enter denominator: ");
        scanf("%d", &denominator);

        divide(numerator, denominator);

    } else {
        printf("Exception handled in main()\n");
    }

    return 0;
}
```

Procède à la façon d'un *goto*

```
Enter numerator: 55
Enter denominator: 3
Result of division: 18
```

```
Enter numerator: 34
Enter denominator: 0
Division by zero error!
Exception handled in main()
```


L'exemple ci-dessus montre l'utilisation des fonctions `setjmp()` et `longjmp()` dans le cas de récupération d'une exception durant l'exécution de la fonction `divide()`.

Il est à noter que la levée de l'exception n'est provoquée ni par le *runtime* ni par le système d'exploitation, mais par le code de la fonction qui appelle `longjmp()` en cas d'erreur.

Gestion des exceptions (3/4)

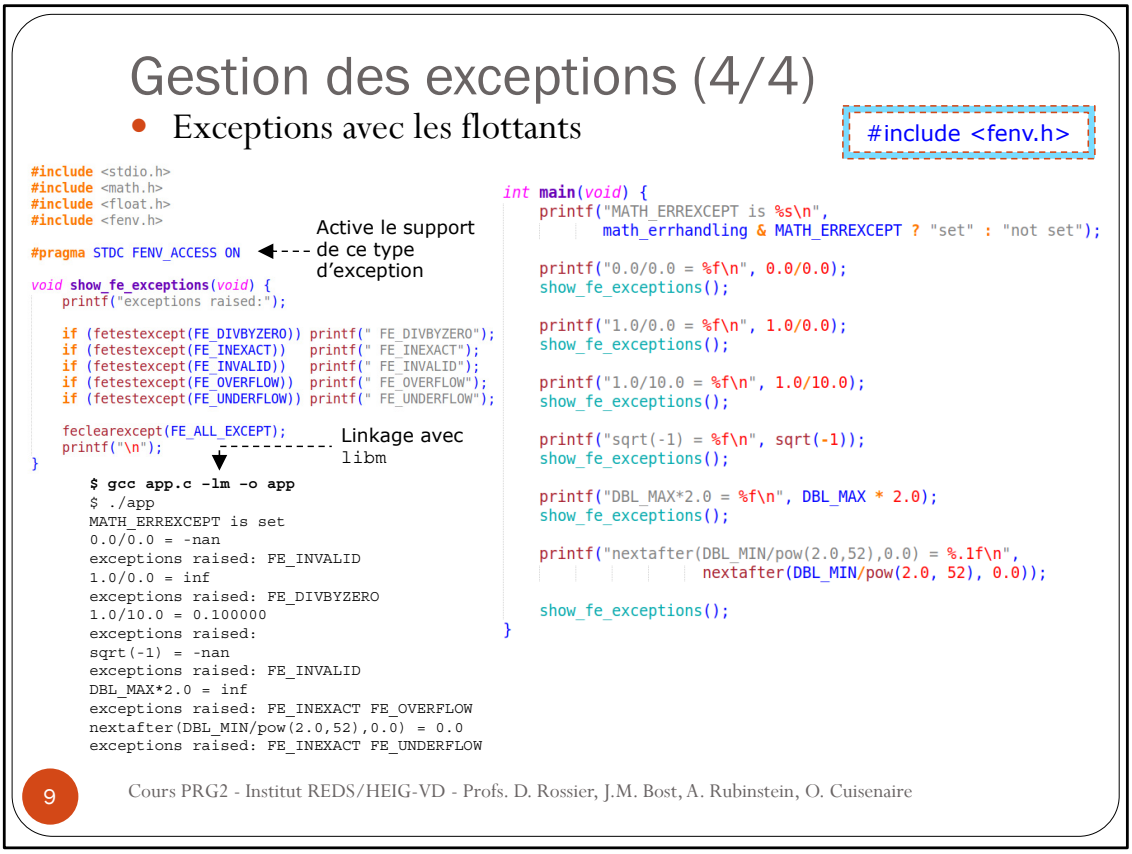
- Implémentation d'une structure `try-catch`

`#include <setjmp.h>`

```c
#define TRY do { jmp_buf ex_buf; if (setjmp(ex_buf) == 0) {
#define CATCH } else {
#define END_TRY } } while (0)

void risky_function(int x) {
    if (x == 0) {
        printf("Error: Division by zero!\n");
        longjmp(ex_buf, 1);
    }
    printf("Result: %f\n", 1.0 / x);
}

int main() {
    TRY
    {
        printf("Trying...\n");
        risky_function(0);
        printf("This line will not be reached.\n");
    }
    CATCH
    {
        printf("Caught an exception!\n");
    }
    END_TRY

    return 0;
}
```

Pas besoin de tester le retour en cas d'erreur

Une construction de type *try-catch*, connue dans d'autres langages, peut être mis en oeuvre en combinant l'utilisation de `#define` et des fonctions `setjmp()` et `longjmp()`.

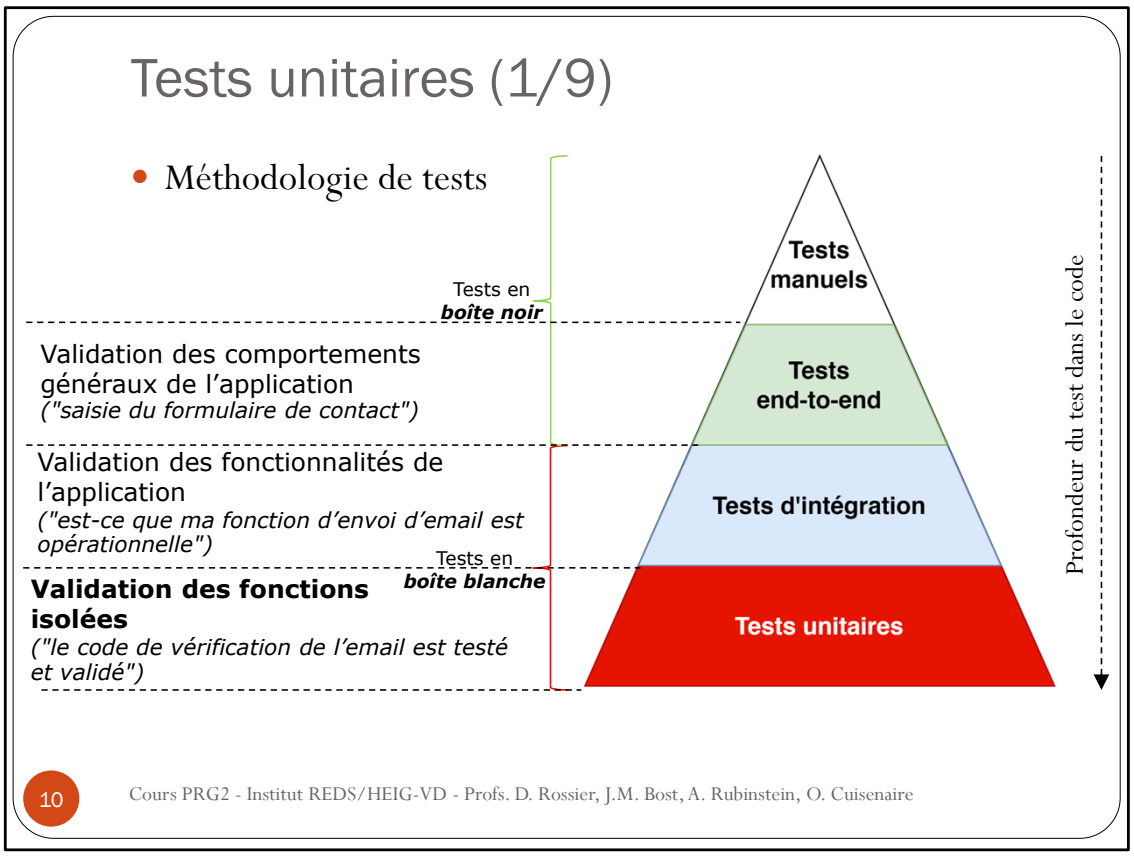


Le standard C définit plusieurs macros, définies dans la librairie `fenv`, pour détecter les différentes exceptions flottantes qui peuvent survenir lors de calculs en virgule flottante, telles que :

`FE_DIVBYZERO`	Division par zéro
`FE_OVERFLOW`	Dépassement de capacité
`FE_UNDERFLOW`	Sous-dépassement de capacité
`FE_INVALID`	Opération invalide
`FE_INEXACT`	Résultat inexact

Ces macros peuvent être utilisées pour tester l'état des registres d'état flottant après une opération en virgule flottante et lever une exception si une erreur est détectée.

L'instruction `#pragma` permet l'activation du support de ce type d'exception de telle sorte que le programme **ne plante pas** en cas d'erreur.



Lors du développement d'un programme, il est important de s'assurer qu'il fonctionne correctement. Pour de très petits programmes, cela peut se faire à la fin du développement, en testant le comportement du programme complet. En revanche, pour des programmes de taille plus importante — dont le développement peut prendre plusieurs semaines à plusieurs années — cette solution **n'est pas réaliste**. D'une part, la probabilité que, une fois terminé, un tel programme fonctionne sans jamais avoir été testé est presque nulle ; d'autre part, les erreurs sont difficiles à localiser lorsqu'elles peuvent se trouver n'importe où dans un gros programme.

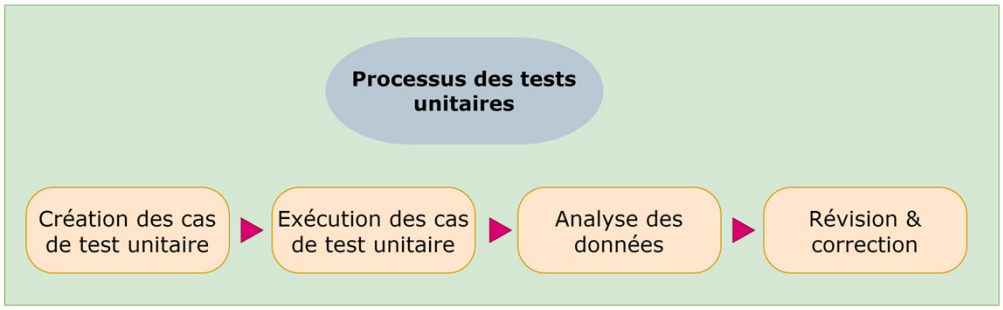
Dès lors, le test de programmes non triviaux doit se faire **petit à petit** : dès qu'une partie individuellement utile du programme, appelée **unité** (*unit*), a été écrite, celle-ci peut (et doit) être testée indépendamment. Si des problèmes apparaissent lors du test d'une unité, ceux-ci doivent être corrigés immédiatement avant de continuer. Une fois le test d'une unité terminé, celle-ci peut être supposée correcte (pour l'instant), et le développement du programme peut se poursuivre avec la prochaine unité.

En procédant de la sorte, le programme final, composé d'unités testées individuellement, a beaucoup plus de chances de fonctionner que si la totalité du code avait été écrite sans être testée.

Cette pratique, consistant à tester individuellement les différentes unités d'un programme, est appelée **test unitaire** (*unit testing*).

Tests unitaires (2/9)

- **Différentes phases** d'élaboration des tests unitaires
  - Cas de test unitaire (***Use Case***)
  - Exécution des *Use Cases*
  - Analyse des données
  - Révision & correction



La mise en place de tests unitaires nécessite plusieurs phases distinctes.

La première phase consiste à **créer des cas de test unitaires**. Les cas de test unitaires sont des scénarios qui décrivent comment l'unité de code doit se comporter dans différentes situations. Ils doivent être précis et couvrir **tous les aspects de la fonctionnalité** de l'unité de code.

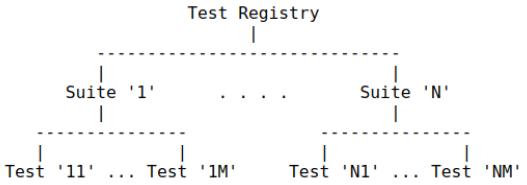
Une fois que les cas de test unitaires ont été créés, ils doivent être exécutés. Cela permet de vérifier que l'unité de code se comporte comme prévu. Les tests unitaires peuvent être exécutés manuellement ou automatiquement.

Après l'exécution des cas de test unitaires, les données doivent être analysées. Cela permet de déterminer si les tests ont réussi ou échoué. Si un test échoue, cela signifie qu'il y a un bug dans l'unité de code.

Si un test échoue, l'unité de code doit être corrigée. Une fois que le code a été corrigé, les tests unitaires doivent être exécutés à nouveau pour s'assurer qu'ils réussissent.

Tests unitaires (3/9)

- *Framework* **CUnit**
- Ensemble de macros d'assertion

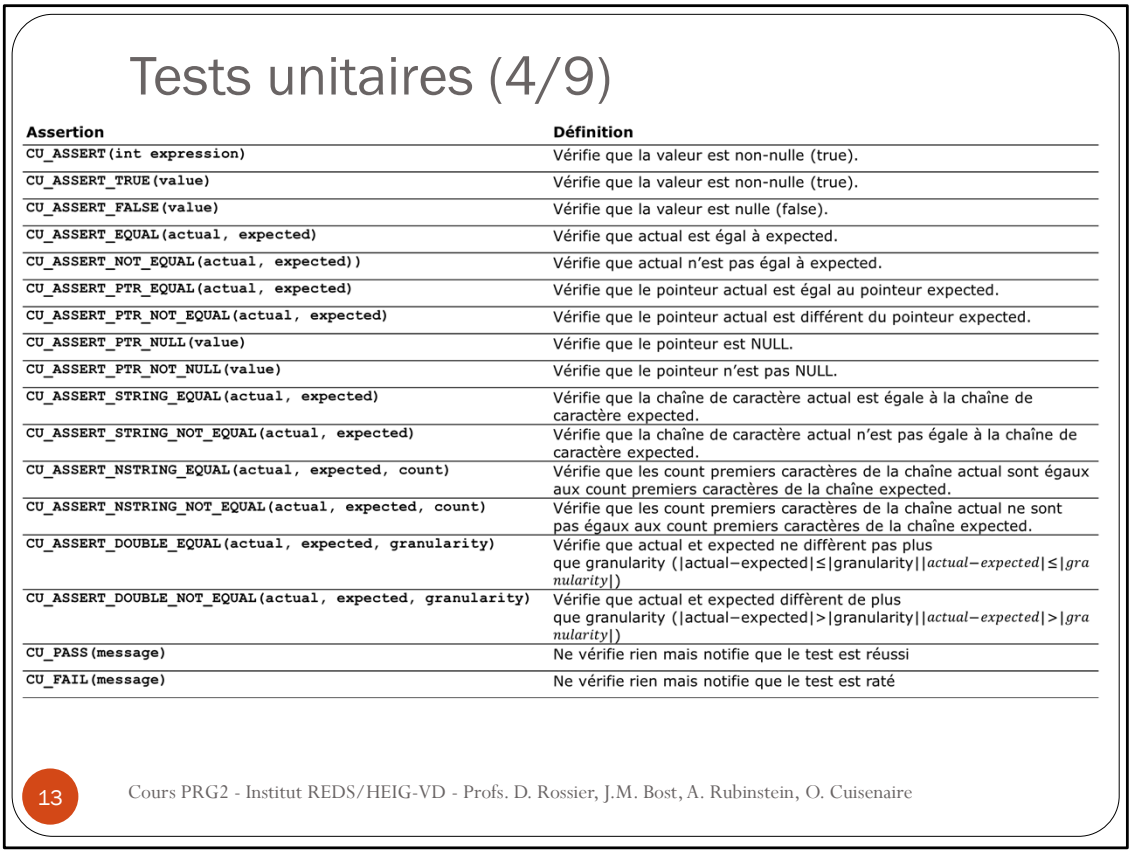


- Procédure
  - **Ecrire** les fonctions pour les tests (et les fonctions d'initialisation/nettoyage des suites si nécessaires)
  - **Initialiser** le catalogue (*test registry*) avec `CU_initialize_registry()`
  - **Ajouter** les suites de test dans le catalogue avec `CU_add_suite()`
  - **Ajouter** les tests dans les suites de tests avec `CU_add_test()`
  - **Exécuter** les tests au moyen d'une interface avec soit `CU_console_run_tests()` ou `CU_curses_run_tests()`
  - **Terminer** proprement l'exécution des tests avec `CU_cleanup_registry()`

CUnit est un *framework* de test unitaire open source et léger pour le langage de programmation C. Il permet aux développeurs d'écrire et d'exécuter **des tests unitaires** pour vérifier le bon fonctionnement de leurs unités de code. *CUnit* est facile à utiliser et à apprendre, et il offre un ensemble complet de fonctionnalités pour tester le code C.

CUnit dispose d'un catalogue (*test registry*) constitué d'un ensemble de suites de tests. Ce catalogue doit être initialisé au début du programme de test et nettoyer en fin de programme.

La création de tests unitaires avec *CUnit* consiste à créer une suite de tests, d'ajouter le(s) test(s) à cette suite, d'exécuter les tests et de récolter les résultats.



CUnit fournit un ensemble complet d'**assertions** pour vérifier les résultats des tests. Ces assertions permettent aux développeurs de **comparer des valeurs**, de vérifier **l'état des pointeurs** et d'évaluer d'autres **conditions importantes**.

Le tableau ci-dessus présente les assertions principales.

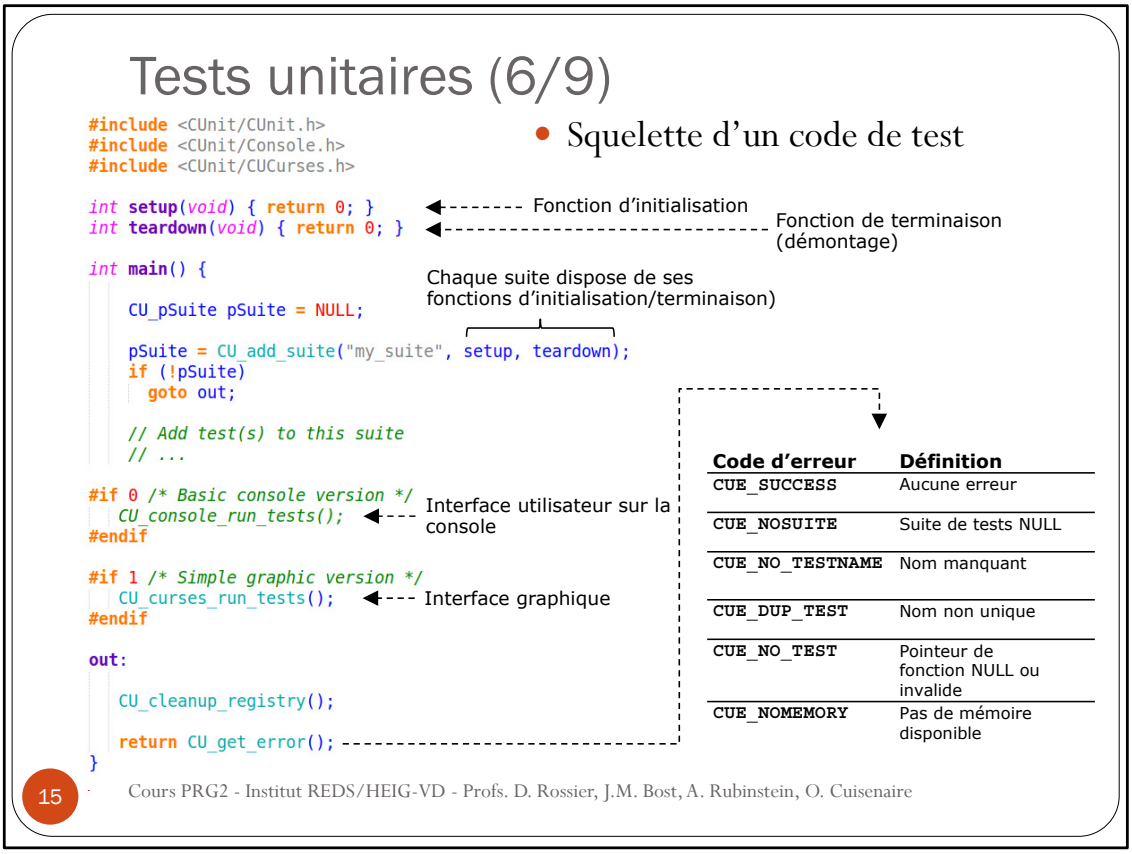
Tests unitaires (5/9)

- Fichiers *include* du *framework* **CUnit**

Header File	Description
`#include <CUnit/CUnit.h>`	ASSERT macros for use in test cases and includes other framework headers.
`#include <CUnit/CUError.h>`	Error handing functions and data types. *Included automatically by CUnit.h.*
`#include <CUnit/TestDB.h>`	Data type definitions and manipulation functions for the test registry, suites, and tests. *Included automatically by CUnit.h.*
`#include <CUnit/TestRun.h>`	Data type definitions and functions for running tests and retrieving results. *Included automatically by CUnit.h.*
`#include <CUnit/Automated.h>`	Automated interface with xml output.
`#include <CUnit/Basic.h>`	Basic interface with non-interactive output to stdout.
`#include <CUnit/Console.h>`	Interactive console interface.
`#include <CUnit/CUCurses.h>`	Interactive console interface (*nix).

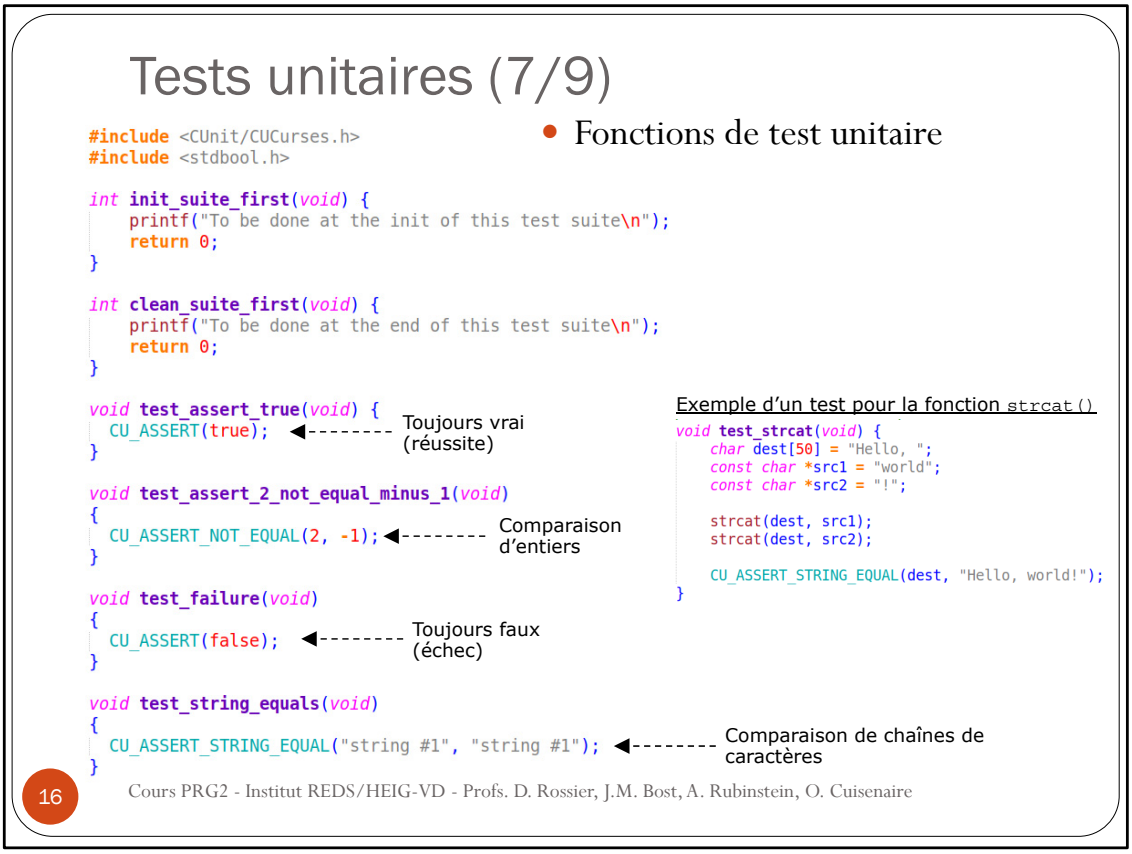
L'utilisation de *CUnit* nécessite l'inclusion de fichiers d'entête tels que présentés ci-dessus. On remarque différent fichier selon le mode d'interaction que l'on souhaite avoir, par exemple avec une simple console ou une interface graphique basique.

Le fichier principal est `CUnit/CUnit.h`



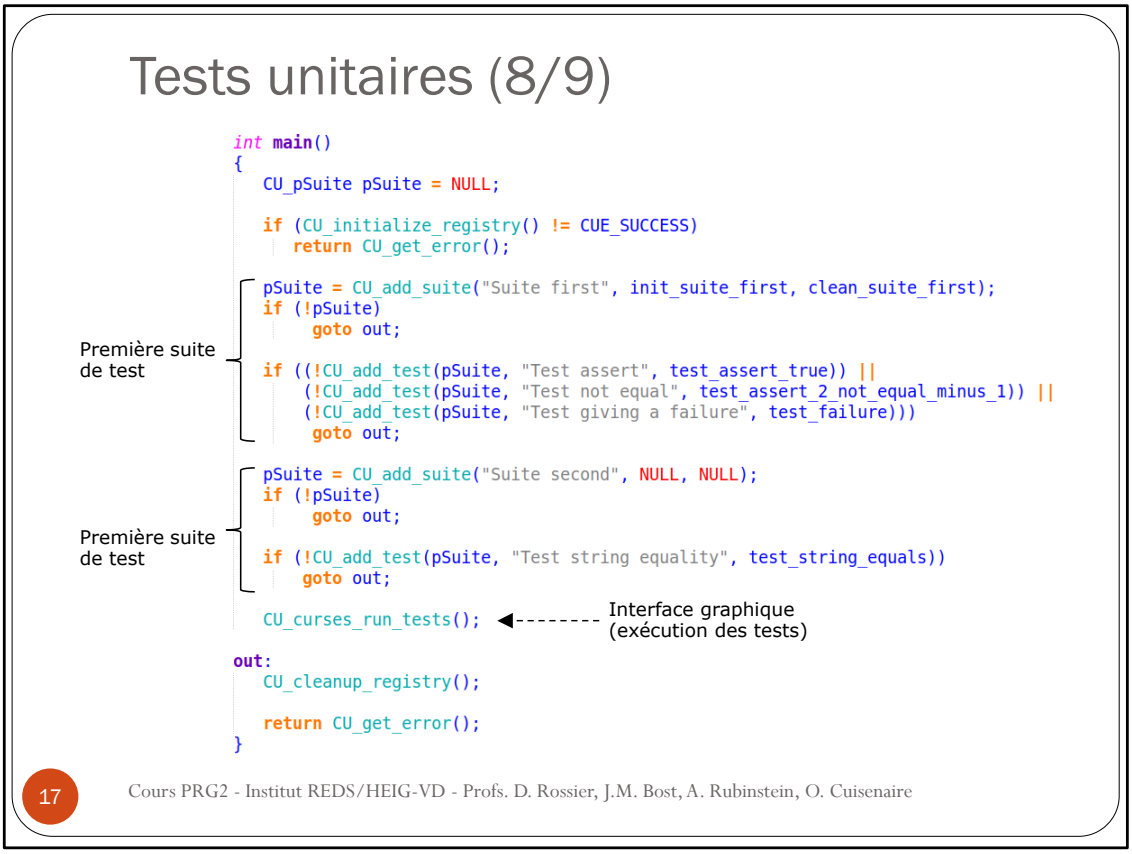
Le code ci-dessus représente un squelette de programme effectuant des tests unitaires. Ce code doit être *linké* avec les fonctions à tester.

S'il n'y a pas de fonction d'initialisation et/ou de terminaison, il est possible de passer l'argument `NULL` dans la fonction `CU_add_suite()`.

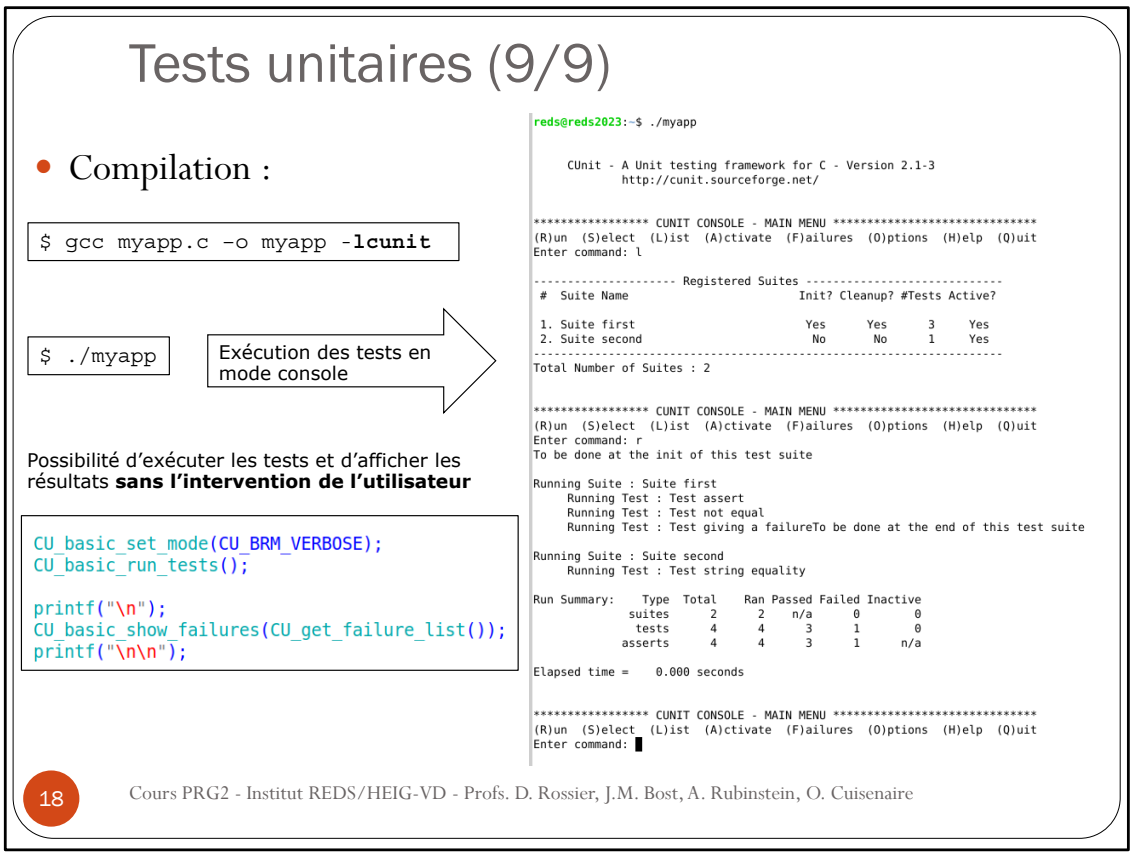


Les fonctions qui effectuent les tests unitaires doivent être implémentées par le programmeur. En effet, ces tests ne sont pas générés «automatiquement». Il s'agit de déterminer les cas d'utilisation et d'effectuer les tests associés.

L'utilisation des macros d'assertions du *framework CUnit* s'avère très pratique puisqu'un échec d'une telle assertion provoquera une situation d'erreur qui pourra être gérée et détaillée lors de l'affichage des erreurs. Ces macros font usage des fonctions `setjmp()` et `longjmp()` évoquées précédemment.



Les suites de tests sont ensuite crées et les fonctions de test enregistrées dans les suites respectives. Dans l'exemple ci-dessus, l'interface graphique permettra à l'utilisateur de sélectionner les suites de tests à exécutés et pourra obtenir des informations sur le résultat de ces tests.



Nous avons déjà évoqué la nécessité de *linker* les fonctions à tester avec le programme de test. La compilation nécessite également de lier la librairie `libcunit` qui contient le *framework CUnit*.

Hormis la possibilité d'interagir soit via une console, soit via une interface graphique, le *framework* offre également des fonctions permettant d'automatiser l'exécution des tests et l'affichage des erreurs, **sans l'intervention de** l'utilisateur. Un exemple de code est montré ci-dessus.