IFT159
Analyse et programmation
Thème 6 — Les fonctions (suite)

Gabriel Girard

Département d'informatique
UNIVERSITÉ DE SHERBROOKE

20 octobre 2015

Thème 6 — Les fonctions (suite)

Les fonctions (suite)

Programmation modulaire et cohésion

Définition

Programmation modulaire et cohésion

Définition

- Un des objectifs de la programmation modulaire est de produire des fonctions avec un haut degré de cohésion.
- Une fonction est dite **fonctionnellement cohésive** si elle n'accomplit qu'un seule tâche.
- On détecte si une fonction est cohésive ou non par

Programmation modulaire et cohésion

Définition

- Un des objectifs de la programmation modulaire est de produire des fonctions avec un haut degré de cohésion.
- Une fonction est dite **fonctionnellement cohésive** si elle n'accomplit qu'un seule tâche.
- On détecte si une fonction est cohésive ou non par
  - le titre de la fonction ;

Programmation modulaire et cohésion

Définition

- Un des objectifs de la programmation modulaire est de produire des fonctions avec un haut degré de cohésion.
- Une fonction est dite **fonctionnellement cohésive** si elle n'accomplit qu'un seule tâche.
- On détecte si une fonction est cohésive ou non par
  - le titre de la fonction ;
  - le commentaire qui décrit la fonction.

Exemples

Exemple

Exemples

Exemple

- « calcul des dividendes » , cette fonction calcule les dividendes d'un actionnaire

Exemples

Exemple

- « calcul des dividendes » , cette fonction calcule les dividendes d'un actionnaire
- « répartition du poids » , cette fonction calcule le poids idéal à appliquer sur n mobiles

Exemples

Exemple

- « calcul des dividendes » , cette fonction calcule les dividendes d'un actionnaire
- « répartition du poids » , cette fonction calcule le poids idéal à appliquer sur n mobiles

Contre-exemples

Exemples

Exemple

- « calcul des dividendes » , cette fonction calcule les dividendes d'un actionnaire
- « répartition du poids » , cette fonction calcule le poids idéal à appliquer sur n mobiles

Contre-exemples

- « Calcul du reste et du quotient » : cette fonction calcule le quotient entier entre deux nombres ainsi que son reste.

Exemples

Exemple

- « calcul des dividendes » , cette fonction calcule les dividendes d'un actionnaire
- « répartition du poids » , cette fonction calcule le poids idéal à appliquer sur n mobiles

Contre-exemples

- « Calcul du reste et du quotient » : cette fonction calcule le quotient entier entre deux nombres ainsi que son reste.
- « Calcul de la moyenne et de l'écart-type » : cette fonction calcule la moyenne et l'écart-type pour une liste de notes.

Programme et cohésion fonctionnelle

Intérêt

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance
- simplifie le traitement des erreurs et de la robustesse

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance
- simplifie le traitement des erreurs et de la robustesse

Un programme

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance
- simplifie le traitement des erreurs et de la robustesse

Un programme

- n'a pas besoin d'être composé uniquement de modules fonctionnellement cohésifs ;

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance
- simplifie le traitement des erreurs et de la robustesse

Un programme

- n'a pas besoin d'être composé uniquement de modules fonctionnellement cohésifs ;
- impossible en pratique la plupart du temps ;

Programme et cohésion fonctionnelle

Intérêt

- simplifie la lecture et la compréhension
- augmente la maintenance
- simplifie le traitement des erreurs et de la robustesse

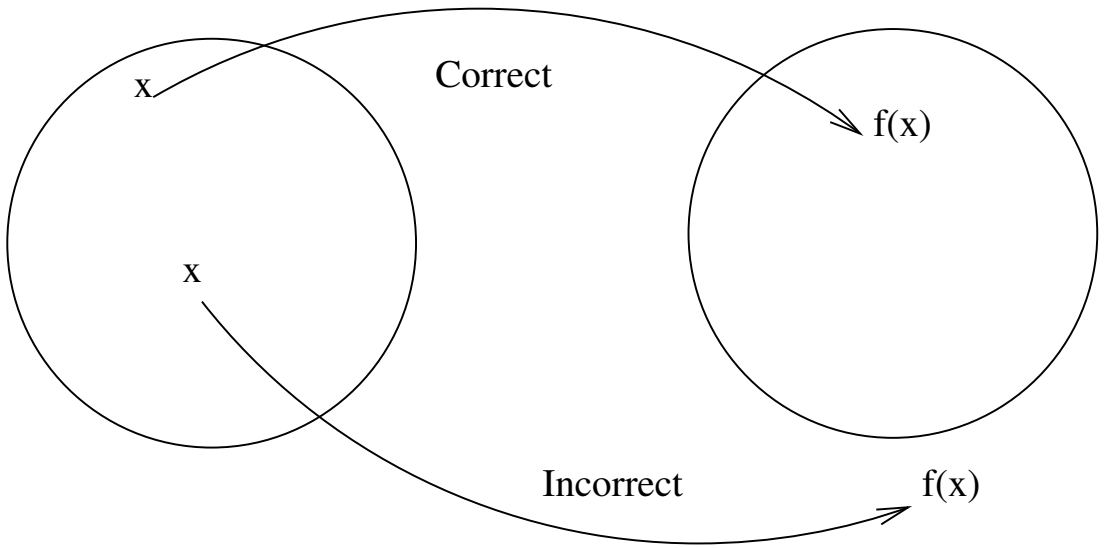
Un programme

- n'a pas besoin d'être composé uniquement de modules fonctionnellement cohésifs ;
- impossible en pratique la plupart du temps ;
- est un objectif à suivre cependant.

Les fonctions (suite)

Validité (Exactitude)

Définition Une fonction est dite *valide* (*exacte* ou *correcte*) si pour des données valides en entrée elle produit un résultat valide (qui est dans l'image de la fonction).



Vérification de l'exactitude

Dépend de deux critères

- *préconditions* : ensemble des données d'entrées pour laquelle la fonction est correcte.
- *postconditions* : ensemble des données en sortie.

procédure

- Un commentaire décrit les préconditions.
- Un commentaire décrit les postconditions.
- `doxygen` définie deux commandes `\pre` et `\post`.

Exactitude

Exemple

Fonction « Calculer moyenne des notes »

Exactitude

Exemple

Fonction « Calculer moyenne des notes »

- **pre** : la liste passée en argument est une liste non-vide de nombres réels compris entre 0 et 100.

Exactitude

Exemple

Fonction « Calculer moyenne des notes »

- **pre** : la liste passée en argument est une liste non-vide de nombres réels compris entre 0 et 100.
- **post** : la valeur de retour est la moyenne à 10^{-2} près des notes de la liste ; c'est un nombre réel compris entre 0 et 100.

Exemple de documentation — I

Moyenne d'une liste de notes

```
/**
   \brief calcule la moyenne d'un ensemble de
          notes

   \pre  l'argument est une suite non-vide de
         nombres de reels.
   \post la valeur de retour est la moyenne
         des nombres.
.......

*/
float calculer_moyenne (float[],...)
```

Exemple de documentation — II

Tri d'un vecteur

```
/**
  \brief trie une liste de reels dans l'ordre
         croissant

  \pre  l'argument est une suite non-vide de
        nombres reels
  \post la liste est triee dans l'ordre
        croissant des elements.
  ....
*/
bool trier_liste (float[], ...)
```

Exemple de documentation — III

La factorielle

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre n est un entier >= 0
  \post la valeur de retour est la
        factorielle de n

  \param[in] n un entier positif ou nul
  \return la factorielle du nombre  n
*/
```

Exemple de documentation — III

La factorielle

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre n est un entier >= 0
  \post la valeur de retour est la
        factorielle de n

  \param[in] n un entier positif ou nul
  \return la factorielle du nombre  n
*/
```

Attention

Cette fonction n'est pas exacte !

Exemple de documentation — IV

La factorielle

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre n est un entier positif
       ou nul inferieur ou egal a 12
  \post la valeur de retour est la
        factorielle du nombre  n

  \param[in] une entier n tel que 0 <= n <= 12
  \return la factorielle du nombre  n
*/
```

Exemples

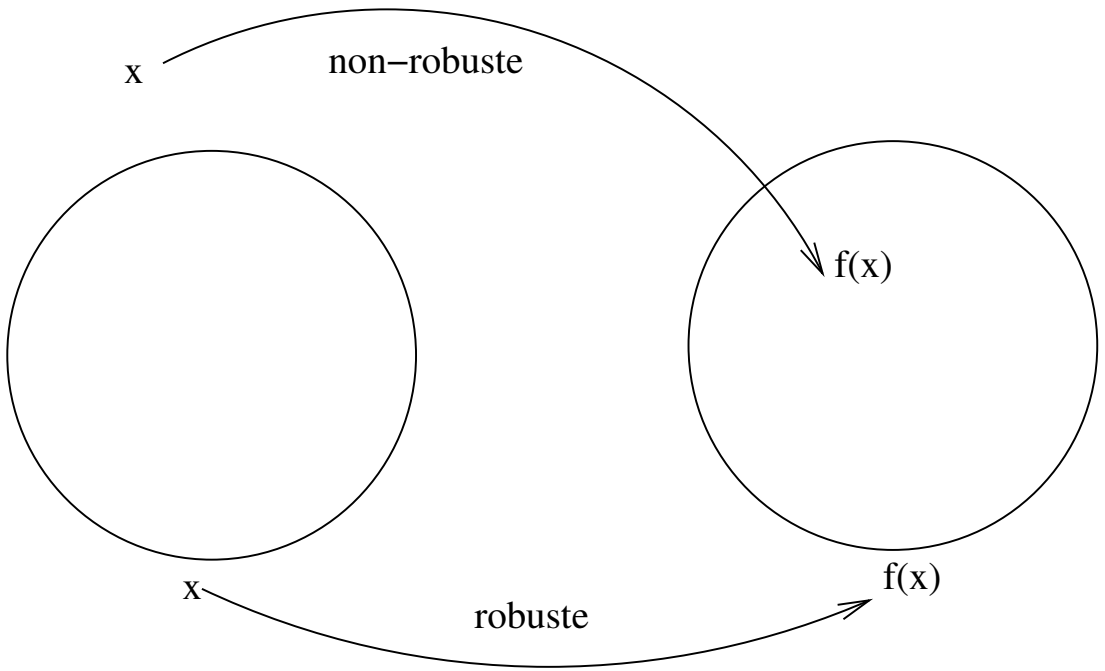
Exemple

Reprenons le programme qui trouve les n premiers multiples.

Robustesse

Définition Une fonction est dite *robuste* si pour des données invalides en entrée elle produit un message d'erreur ou un résultat invalide (qui n'est pas dans l'image de la fonction).

Robustesse



Vérification de la robustesse

Une fonction robuste

- valide les arguments en entrée ;
- dispose d'un moyen pour transmettre
  - si une erreur s'est produite
  - la catégorie d'erreur
- ne s'occupe pas du typage en général

Vérification de la robustesse

Une fonction robuste

- valide les arguments en entrée ;
- dispose d'un moyen pour transmettre
  - si une erreur s'est produite
  - la catégorie d'erreur
- ne s'occupe pas du typage en général

Méthodes

- Utilisation des continuations
- Utilisation des exceptions
- Utilisation des codes d'erreurs
- Utilisation des `assert`

Continuations et exceptions

Les continuations et les exceptions

Continuations et exceptions

Les continuations et les exceptions

- ne sont pas vues en IFT 159

Continuations et exceptions

Les continuations et les exceptions

- ne sont pas vues en IFT 159
- continuations en IFT 359

Continuations et exceptions

Les continuations et les exceptions

- ne sont pas vues en IFT 159
- continuations en IFT 359
- exceptions en IFT 339 et IFT 232

Continuations et exceptions

Les continuations et les exceptions

- ne sont pas vues en IFT 159
- continuations en IFT 359
- exceptions en IFT 339 et IFT 232
- efficace, général, mais coûteuses

Les codes d'erreurs : introduction

Les codes d'erreur

- sont des valeurs retournées par une fontion lorsqu'un problème est détecté pendant l'exécution
- doivent être retournés séparément des valeurs de retour ou ne pas être dans l'image de la fonction.
- facilitent la récupération d'une erreur par un test (`switch`)
- facilitent sa propagation

Les codes d'erreurs : introduction

Approche

- Une méthode consiste à utiliser la valeur de retour pour indiquer, à l'aide d'un booléen ou d'un nombre, l'état du calcul.
- Par convention, 0 indique que tout c'est bien passé.
- C'est le fontionnement classique pour beaucoup d'opérateurs.

Les codes d'erreurs

Documentation

- Les codes d'erreurs doivent être clairement documentés au début de la fonction.
- Ils doivent doivent être clairement définis
  - comme des constantes,
  - comme des macros (#define).
- Ceci permet d'utiliser la fonction dans un test .

Exemple : factorielle robuste

Documentation doxygen

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre est un entier n | 0 < n <= 12
  \post la valeur de retour est la factorielle du
        nombre  n

  Cette fonction est robuste.

  \param[in] n un entier | 0 < n <= 12

  \return la factorielle du nombre  n
  \return VALEUR_TROP_PETITE (-1) si n < 0
  \return VALEUR_TROP_GRANDE (-2) si n > 12
 */
```

Exemple : factorielle robuste

Code

```
const int VALEUR_TROP_PETITE = -1;
const int VALEUR_TROP_GRANDE = -2;
...
int factorielleRobuste(int n)
{
    int factorielle(int n);
    int resultat;

    if (n < 0) resultat = VALEUR_TROP_PETITE;
    else if (n > 12) resultat = VALEUR_TROP_GRANDE;
    else
        for (int i = 1; i != n; i += 1)
        {
            resultat *= i;
        }
    return resultat;
}
```

Assertions

Les assertions

- simples à utiliser et à désactiver
- désactivation : perte de la robustesse
- désactivation : gain en performance
- pratique pour la période de tests
- devrait être utilisées systématiquement

Utilisation d'assertion

Instruction `assert`

- nécessite `#include <cassert>`
- teste une condition durant l'exécution
- provoque un arrêt en cas d'échec
- peut être ignoré lors de la compilation par l'ajout de `#define NDEBUG`

Utilisation d'assertion

Instruction `assert`

- nécessite `#include <cassert>`
- teste une condition durant l'exécution
- provoque un arrêt en cas d'échec
- peut être ignoré lors de la compilation par l'ajout de `#define NDEBUG`

Utilisation

```
assert(condition);
```

Utilisation d'assertion

Exemple : énoncé

On veu implanter une fonction qui permet de calculer la moyenne des notes. On peut se servir du assert pour valider les préconditions.

Exemple : programme

```
#include <cassert>
...
float calculer_moyenne(....)
{
   assert(nb_notes > 0);
   ...
}
```

Gestion de la validation

Approche

- utilisation de deux fonctions
  - une traite l'exactitude
  - l'autre traite la robustesse
- que ce soit avec ou sans code d'erreur
- sépare les préoccupations
- améliore la maintenance

Documentation

Le commentaire de la fonction

- indique un comportement robuste s'il y a lieu
- indique la gestion des cas d'erreur (code d'erreur ou autre)
- indique les cas non spécifiés si la robustesse n'est pas traité

Exemple : factorielle non robuste (documentation)

Documentation doxygen

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre est un entier n | 0<=n<=12
  \post la valeur de retour est la
        factorielle du nombre n

     Cette fonction n'est pas robuste. Si n n'est
        pas valide le comportement de la fonction
        n'est pas specifie.

  \param[in] n un entier | 0 <= n <= 12
  \return la factorielle du nombre  n
*/
```

Exemple : factorielle non robuste (I)

Implémentation (I)

```
int factorielle1(int n)
{
    int resultat;

    resultat = 1;
    for (int i = 1; i <= n; i += 1)
    {
        resultat *= i;
    }

    return resultat;
}
```

Exemple : factorielle non robuste (II)

Implémentation (II)

```
int factorielle2(int n)
{
    int resultat;

    resultat = 1;
    // N.B. la condition de la boucle
    for (int i = 1; i != n; i += 1)
    {
        resultat *= i;
    }

    return resultat;
}
```

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;
2. coder la fonction chapeau qui ajoute la robustesse ;

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;
2. coder la fonction chapeau qui ajoute la robustesse ;
    1. valider les entrées ;

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;
2. coder la fonction chapeau qui ajoute la robustesse ;
    1. valider les entrées ;
    2. appeler la fonction non robuste ;

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;
2. coder la fonction chapeau qui ajoute la robustesse ;
    1. valider les entrées ;
    2. appeler la fonction non robuste ;
3. intégrer les fonctions dans le programme ;

Mise en place de solution robuste

Étapes avec la méthode des codes de retour

1. coder la fonction non robuste ;
2. coder la fonction chapeau qui ajoute la robustesse ;
    1. valider les entrées ;
    2. appeler la fonction non robuste ;
3. intégrer les fonctions dans le programme ;
4. effectuer les tests d'intégration ;

Exemple : factorielle non robuste

Documentation doxygen

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre est un entier n | 0 < n <= 12
  \post la valeur de retour est la factorielle du
        nombre n

  Cette fonction n'est pas robuste.

  \param[in] n un entier | 0 < n <= 12
  \return la factorielle du nombre n
 */
```

Exemple : factorielle non robuste

Code

```
int factorielle(int n)
{
    int resultat;

    resultat = 1;
    for (int i = 1; i <= n; i += 1)
    {
        resultat *= i;
    }

    return resultat;
}
```

Exemple : factorielle robuste

Documentation doxygen

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre est un entier n | 0 < n <= 12
  \post la valeur de retour est la factorielle du
        nombre  n

  Cette fonction est robuste.

  \param[in] n un entier | 0 < n <= 12
  \return la factorielle du nombre  n

  \return VALEUR_TROP_PETITE (-1) si n < 0
  \return VALEUR_TROP_GRANDE (-2) si n > 12
 */
```

Exemple : factorielle robuste

Code

```
const int VALEUR_TROP_PETITE = -1;
const int VALEUR_TROP_GRANDE = -2;
...
int factorielleRobuste(int n)
{
    int factorielle(int n);
    int resultat;

    if (n < 0) resultat = VALEUR_TROP_PETITE;
    else if (n > 12) resultat = VALEUR_TROP_GRANDE;
    else resultat = fatorielle(n);

    return resultat;
}
```

Mise en place de solution robuste

Étapes avec la méthode des assertions

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;
3. tester pour des entrées valides ;

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;
3. tester pour des entrées valides ;
4. intégrer la fonction dans le programme ;

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;
3. tester pour des entrées valides ;
4. intégrer la fonction dans le programme ;
5. effectuer les tests d'intégration ;

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;
3. tester pour des entrées valides ;
4. intégrer la fonction dans le programme ;
5. effectuer les tests d'intégration ;
6. au besoin, désactiver les `assert`.

Mise en place de solution robuste

Étapes avec la méthode des assertions

1. coder la fonction sans validation ;
2. ajouter des assertions ;
3. tester pour des entrées valides ;
4. intégrer la fonction dans le programme ;
5. effectuer les tests d'intégration ;
6. au besoin, désactiver les `assert`.

Attention !

Désactiver les assertions supprime la robustesse.

Exemple : factorielle robuste (documentation)

Documentation doxygen

```
/**
  \brief calcule la factorielle d'un nombre
  \pre le parametre est un entier n | 0 < n <= 12
  \post la valeur de retour est la factorielle du
        nombre  n

  Cette fonction est robuste.

  \param[in] n un entier | 0 < n <= 12
  \return la factorielle du nombre  n
 */
```

Exemple : factorielle robuste (I)

Code (I)

```
int factorielle1(int n)
{
    assert(n >= 0);
    assert(n <= 12);

    int resultat;

    resultat = 1;
    for (int i = 1; i <= n; i += 1)
    {
        resultat *= i;
    }

    return resultat;
}
```

Fiabilité

Définition

Une fonction est dite *fiable* si elle est à la fois exacte et robuste.

Fiabilité

Définition

Une fonction est dite *fiable* si elle est à la fois exacte et robuste.

Pour être fiable, une fonction doit

Fiabilité

Définition

Une fonction est dite *fiable* si elle est à la fois exacte et robuste.

Pour être fiable, une fonction doit

- utiliser des fonctions valides (exactes)

Fiabilité

Définition

Une fonction est dite *fiable* si elle est à la fois exacte et robuste.

Pour être fiable, une fonction doit

- utiliser des fonctions valides (exactes)
- utiliser des fonctions robustes

Les fonctions (suite)

Entorse à la programmation modulaire

Utilisation du `return`

Entorse à la programmation modulaire

Utilisation du `return`

- Afin d'éviter d'avoir trop de `if` emboités ou trop de `else if`, il est toléré d'avoir plusieurs `return`.

Entorse à la programmation modulaire

Utilisation du `return`

- Afin d'éviter d'avoir trop de `if` emboités ou trop de `else if`, il est toléré d'avoir plusieurs `return`.
- Ceux-ci ne sont utilisés que pour retourner des types d'erreurs en plus du `return` normal.

Entorse à la programmation modulaire

Utilisation du `return`

- Afin d'éviter d'avoir trop de `if` emboités ou trop de `else if`, il est toléré d'avoir plusieurs `return`.
- Ceux-ci ne sont utilisés que pour retourner des types d'erreurs en plus du `return` normal.
- Chaque type d'erreur devrait être défini comme une constante.

Entorse à la programmation modulaire

Utilisation du `return`

- Afin d'éviter d'avoir trop de `if` emboités ou trop de `else if`, il est toléré d'avoir plusieurs `return`.
- Ceux-ci ne sont utilisés que pour retourner des types d'erreurs en plus du `return` normal.
- Chaque type d'erreur devrait être défini comme une constante.

Justification

En cas d'erreur ou d'invalidation des données, le flux normal du programme est interrompu. Il est aisé de reconstruire normalement le flux du programme à l'aide de `if/else if`.

Exemple : factorielle robuste

Code

```
const int VALEUR_TROP_PETITE = -1;
const int VALEUR_TROP_GRANDE = -2;
...
int factorielleRobuste(int n)
{
    int factorielle(int n);
    int resultat;

    if (n < 0) return(VALEUR_TROP_PETITE);
    if (n > 12) return(VALEUR_TROP_GRANDE);

    resultat = fatorielle(n);

    return resultat;
}
```

Mise en place de solution robuste

Stratégie à adopter

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)
3. Tests unitaires avec des valeurs valides et invalides

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)
3. Tests unitaires avec des valeurs valides et invalides
4. Tests d'intégration avec des valeurs valides et invalides

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)
3. Tests unitaires avec des valeurs valides et invalides
4. Tests d'intégration avec des valeurs valides et invalides
5. Tests système et tests d'acceptation avec les valeurs attendues

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)
3. Tests unitaires avec des valeurs valides et invalides
4. Tests d'intégration avec des valeurs valides et invalides
5. Tests système et tests d'acceptation avec les valeurs attendues
6. Désactivation des `assert`

Mise en place de solution robuste

Stratégie à adopter

1. Définition d'une fonction non-robuste avec des `assert`
2. Définition des fonctions de validation (avec de multiples points de sortie)
3. Tests unitaires avec des valeurs valides et invalides
4. Tests d'intégration avec des valeurs valides et invalides
5. Tests système et tests d'acceptation avec les valeurs attendues
6. Désactivation des `assert`
7. Déploiement

Exemple : factorielle robuste

Code

```
const int VALEUR_TROP_PETITE = -1;
const int VALEUR_TROP_GRANDE = -2;
int factorielleRobuste(int n)
{
    int factorielle(int n);
    if (n < 0) return(VALEUR_TROP_PETITE);
    if (n > 12) return(VALEUR_TROP_GRANDE);
    return fatorielle(n);
}
int factorielle_non_robuste(int n)
{
    assert (n >= 0);
    assert (n <= 12);
    int resultat = 1;
    for (int i = 1; i <= n; i += 1)
          resultat *= i;
    return resultat;
}
```

Exemples

Reprenons l'exemples des multiples.

Les fonctions (suite)

Tests et mise au point

- On contrôle le niveau d'erreurs en créant des fonctions, en restreignant leur dimension et en utilisant convenablement les paramètres.

Tests et mise au point

- On contrôle le niveau d'erreurs en créant des fonctions, en restreignant leur dimension et en utilisant convenablement les paramètres.
- On doit d'abord tester une fonction lors de sa conception en faisant des traces manuelles.

Tests et mise au point

- On contrôle le niveau d'erreurs en créant des fonctions, en restreignant leur dimension et en utilisant convenablement les paramètres.
- On doit d'abord tester une fonction lors de sa conception en faisant des traces manuelles.
- Lors de l'implémentation, on doit tester chaque fonction individuellement (programmes pilotes).

Tests et mise au point

- Mise au point descendante (fonctions bidons).

Tests et mise au point

- Mise au point descendante (fonctions bidons).
- Mise au point ascendante (programme pilote).

Tests et mise au point

- Mise au point descendante (fonctions bidons).
- Mise au point ascendante (programme pilote).
- Approche combinée.

Tests et mise au point

- Mise au point descendante (fonctions bidons).
- Mise au point ascendante (programme pilote).
- Approche combinée.
- Tests d'intégration.

Les fonctions (suite)

Utilisation des fonctions

- Expressions logiques.

Utilisation des fonctions

- Expressions logiques.

- **Exemple :**
  Écrire un programme qui lit des caractères et qui effectue un traitement qui dépend du type de caractère (chiffre, lettre ou autre).

Utilisation des fonctions

- Récursivité.
  La résursivité consiste à implanter une solution comme une fonction qui s'appelle elle-même jusqu'à ce que la solution soit trouvée.

Utilisation des fonctions

- Récursivité.
  La résursivité consiste à implanter une solution comme une fonction qui s'appelle elle-même jusqu'à ce que la solution soit trouvée.

- **Exemple :**
  Écrire un programme qui trouve le factoriel d'un nombre.

Exercices

1. Écrire un programme qui calcule et affiche de façon récursive les n premiers nombres de Fibonnaci. Les nombres de Fibonnaci sont définis de la façon suivante :
   La suite de Fibonacci est : 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$\text{fib}(0) = 0$$
$$\text{fib}(1) = 1$$
$$\text{fib}(n) = \text{fib}(n\text{-}1) + \text{fib}(n\text{-}2).$$

Exercices

1. Écrire un programme qui calcule et affiche de façon récursive les n premiers nombres de Fibonnaci. Les nombres de Fibonnaci sont définis de la façon suivante :
   La suite de Fibonacci est : 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
   $$\text{fib}(0) = 0$$
   $$\text{fib}(1) = 1$$
   $$\text{fib(n)} = \text{fib(n-1)} + \text{fib(n-2)}.$$

2. Écrire une fonction qui élève un nombre entier à la puissance entière n sachant que
   $$X^i = X * X^{i-1}.$$

Les fonctions (suite)

Paramètres de sortie

syntaxe

Paramètres de sortie

syntaxe

- Une fonction peut avoir à retourner plus d'une valeur.
- Lors de la conception on peut aisément spécifier plusieurs paramètres de sortie.
- En C++, on spécifie un paramètre de sortie en ajoutant un & après le type (`type &`).

Paramètres de sortie

syntaxe

- Une fonction peut avoir à retourner plus d'une valeur.
- Lors de la conception on peut aisément spécifier plusieurs paramètres de sortie.
- En C++, on spécifie un paramètre de sortie en ajoutant un & après le type (`type &`).

Exemple

1. Écrire une fonction qui échange le contenu de deux variables (`swap`).
2. Écrire une fonction qui reçoit trois valeurs en entrée et qui retourne la somme et la moyenne.

Méthode de passage d'un paramètre

Passage par valeur

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Passage par référence

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Passage par référence

- paramètre de sortie ou d'entrée/sortie ;

Méthode de passage d'un paramètre

Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Passage par référence

- paramètre de sortie ou d'entrée/sortie ;
- donne accès directement à la variable passée en paramètre en transmettant la référence (l'adresse en mémoire) de cette variable ;

Méthode de passage d'un paramètre

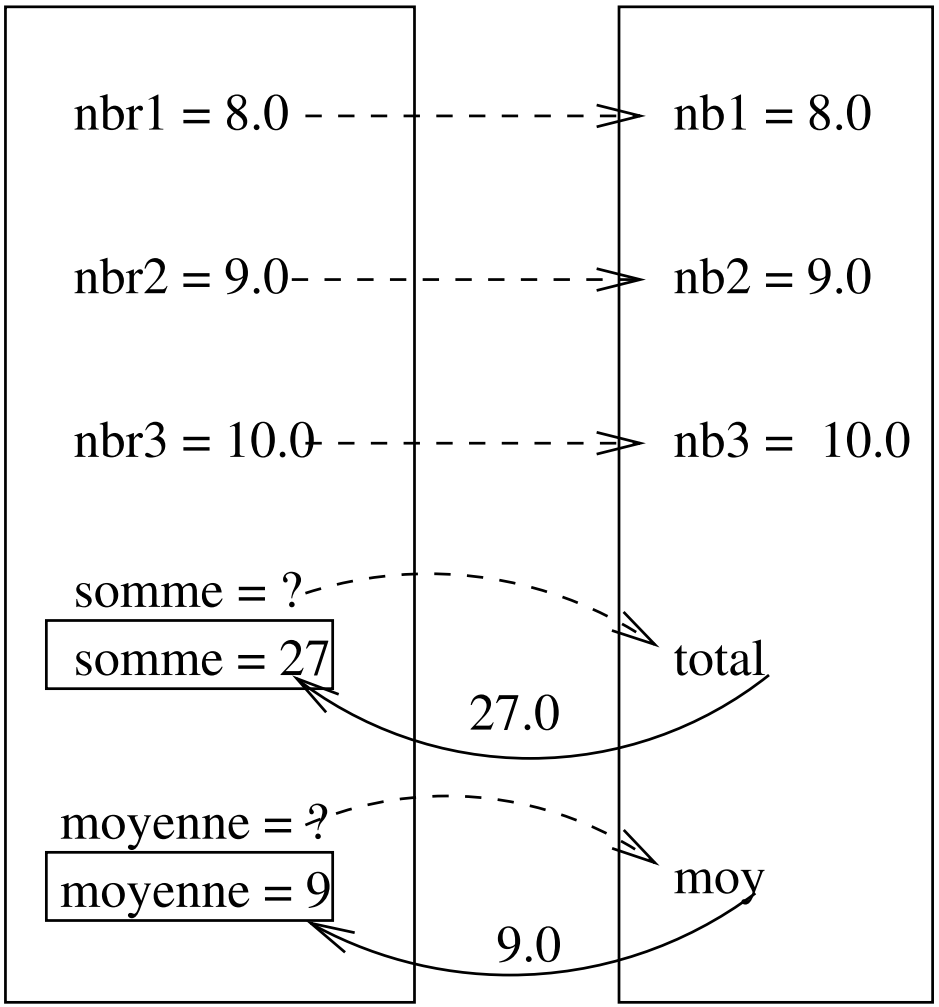
Passage par valeur

- paramètre d'entrée ;
- travaille avec une copie de la valeur de la variable ;
- pas d'effet de bord ;

Passage par référence

- paramètre de sortie ou d'entrée/sortie ;
- donne accès directement à la variable passée en paramètre en transmettant la référence (l'adresse en mémoire) de cette variable ;
- effet de bord possible ;

Paramètres de sortie



Fonctions avec paramètres de sortie

Syntaxe de la déclaration (ligne prototype)

```
type nom_fonction (type, type &, type &);
```

Syntaxe de la définition (paramètres formels)

```
type nom_fonction (type nom_P_in,
                   type &nom_P_in_out,
                   type &nom_P_out);
```

Syntaxe de l'appel (paramètres effectifs)

```
x = nom_fonction (p1, p2, p3)
```

Documentation `doxygen`

Variantes de `param`

- `\param[in]` pour les paramètres lus seulement
- `\param[in,out]` pour les paramètres lus et modifiés
- `\param[out]` pour les paramètres modifiés seulement

Documentation `doxygen`

Variantes de `param`

- `\param[in]` pour les paramètres lus seulement
- `\param[in,out]` pour les paramètres lus et modifiés
- `\param[out]` pour les paramètres modifiés seulement

Documentation de la fonction `swap`

```
/**
  \brief Echange le contenu de deux variables

  \param[in,out] n la premiere variable
  \param[in,out] m la deuxieme variable
  */
```

Remarques

Passage par référence

- est différent du passage de pointeur (IFT 339)
- est préférable au passage par valeur pour les grandes données
- mais est critiqué
  - pour son manque de lisibilité ;
  - les erreurs qui peuvent survenir.

Remarques

Passage par référence

- est différent du passage de pointeur (IFT 339)
- est préférable au passage par valeur pour les grandes données
- mais est critiqué
  - pour son manque de lisibilité ;
  - les erreurs qui peuvent survenir.

Le mot clé `const`

- permet d'éviter une modification du paramètre
- conserve l'efficacité du passage par référence

Les fonctions (suite)

Raffinement successif

1. Analyse globale.

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).
3. Analyse/conception du module du premier niveau.

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).
3. Analyse/conception du module du premier niveau.
4. Analyse/conception des modules du second niveau.

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).
3. Analyse/conception du module du premier niveau.
4. Analyse/conception des modules du second niveau.
5. Analyse/conception descendante de tous les modules niveaux par niveaux jusqu'au plus primitif.

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).
3. Analyse/conception du module du premier niveau.
4. Analyse/conception des modules du second niveau.
5. Analyse/conception descendante de tous les modules niveaux par niveaux jusqu'au plus primitif.
6. Implémentation des modules niveau par niveau.

Raffinement successif

- Analyse globale.

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.
  2. Algorithme du module principal.

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.
  2. Algorithme du module principal.
- Analyse/conception des modules (fonctions).

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.
  2. Algorithme du module principal.
- Analyse/conception des modules (fonctions).
  1. Analyse du module principal.

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.
  2. Algorithme du module principal.
- Analyse/conception des modules (fonctions).
  1. Analyse du module principal.
  2. Conception du module principal (algorithme).

Raffinement successif

- Analyse globale.
- Analyse/conception du niveau principal.
  1. Analyse du module principal.
  2. Algorithme du module principal.
- Analyse/conception des modules (fonctions).
  1. Analyse du module principal.
  2. Conception du module principal (algorithme).
- Implémentation des modules.

Raffinement successif

1. Analyse globale du problème.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
    1. Analyse du module principal.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
    1. Analyse du module principal.
    2. Conception du module principal (algorithme).

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
    1. Analyse du module principal.
    2. Conception du module principal (algorithme).
    3. Implémentation.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
    1. Analyse du module principal.
    2. Conception du module principal (algorithme).
    3. Implémentation.
3. Pour chacun des modules, on réapplique les trois étapes.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
    1. Analyse du module principal.
    2. Conception du module principal (algorithme).
    3. Implémentation.
3. Pour chacun des modules, on réapplique les trois étapes.
    1. Analyse du module.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
   1. Analyse du module principal.
   2. Conception du module principal (algorithme).
   3. Implémentation.
3. Pour chacun des modules, on réapplique les trois étapes.
   1. Analyse du module.
   2. Conception du module (algorithme).

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
   1. Analyse du module principal.
   2. Conception du module principal (algorithme).
   3. Implémentation.
3. Pour chacun des modules, on réapplique les trois étapes.
   1. Analyse du module.
   2. Conception du module (algorithme).
   3. Implémentation.

Exemple

On veut écrire un programme qui fait la mise à jour de votre compte en banque. Le programme traite les transactions (dépôt (d), retrait (r) et terminer (t)) une à la fois, ajuste le solde et garde trace du nombre de transactions de chaque type. À la fin on imprime le solde de départ, le nouveau solde et le nombre de transactions de chaque type. On doit refuser un retrait si le solde devient négatif.

Les fonctions (suite)

Exemples (analyse et robustesse)

- *Gemini V* - Manque son site d'aterrissage
  Le calcul a oublié de considérer le mouvement de la Terre autour du soleil (mauvaise analyse).

Exemples (analyse et robustesse)

- *Gemini V* - Manque son site d'aterrissage
  Le calcul a oublié de considérer le mouvement de la Terre autour du soleil (mauvaise analyse).
- *Apollo* - Le logiciel n'était ni convivial ni robuste.

Exemples (analyse et robustesse)

- *Gemini V* - Manque son site d'aterrissage
  Le calcul a oublié de considérer le mouvement de la Terre autour du soleil (mauvaise analyse).
- *Apollo* - Le logiciel n'était ni convivial ni robuste.
  - *Apollo 8* - Effacement de toute la mémoire
    Erreur provoquée par l'astronaute et des mauvaises spécifications.

Exemples (analyse et robustesse)

- *Gemini V* - Manque son site d'aterrissage
  Le calcul a oublié de considérer le mouvement de la Terre autour du soleil (mauvaise analyse).
- *Apollo* - Le logiciel n'était ni convivial ni robuste.
  - *Apollo 8* - Effacement de toute la mémoire
    Erreur provoquée par l'astronaute et des mauvaises spécifications.
  - *Apollo 11* - Alarme à l'atterissage sur la lune.
    Le commutateur du radar était en mauvaise position.

Exemples (analyse et robustesse)

- *Gemini V* - Manque son site d'aterrissage
  Le calcul a oublié de considérer le mouvement de la Terre autour du soleil (mauvaise analyse).
- *Apollo* - Le logiciel n'était ni convivial ni robuste.
  - *Apollo 8* - Effacement de toute la mémoire
    Erreur provoquée par l'astronaute et des mauvaises spécifications.
  - *Apollo 11* - Alarme à l'atterissage sur la lune.
    Le commutateur du radar était en mauvaise position.
- *USS Yorktown* - Panne des moteurs
  La saisie erronnée d'un «0» au clavier a provoqué l'arrêt du système de propulsion.

Problème de modularité et mauvais tests

- Jupiter ou Vinking - Perte de la sonde
  Téléchargement et effacement du module de contrôle de l'atterissage provoque une coupure des communications

Problème de modularité et mauvais tests

- Jupiter ou Vinking - Perte de la sonde
  Téléchargement et effacement du module de contrôle de l'atterissage provoque une coupure des communications
- Thérac 25 (1985) - Six personnes ont reçu une surdose de radiation pendant leur traitement contre le cancer. Trois en sont mortes.
  Conclusion : tests et documentation inadéquats.

Paroles célèbres

- ≪There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone.≫ Bjarne Stroustrup

Paroles célèbres

- ≪There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone.≫
  Bjarne Stroustrup
- ≪The use of COBOL cripples the mind ; its teaching should therefore be regarded as a criminal offense.≫
  E.W. Dijkstra

Paroles célèbres

- ≪There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone.≫
  Bjarne Stroustrup
- ≪The use of COBOL cripples the mind ; its teaching should therefore be regarded as a criminal offense.≫
  E.W. Dijkstra
- ≪It is practically impossible to teach good programming style to students that have had prior exposure to BASIC. As potential programmers, they are mentally mutilated beyond hope of regeneration.≫
  E. W. Dijkstra