IFT159
Analyse et programmation
Chapitre 7 — Les fonctions (suite)

Gabriel Girard

Département d'informatique

UNIVERSITÉ DE SHERBROOKE

22 septembre 2008

Chapitre 7 — Les fonctions (suite)

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.

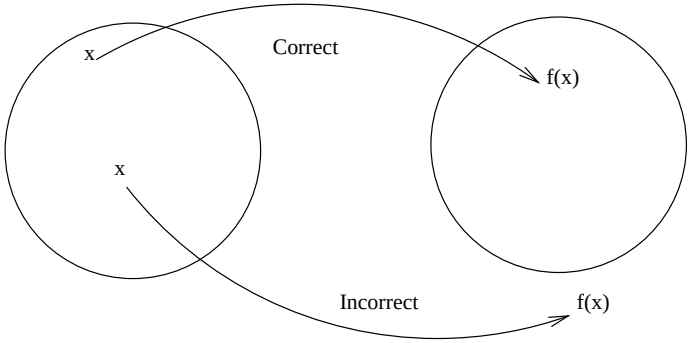
Exemples

Exemple

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

Exactitude

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



Exactitude

procédure

- Un commentaire décrit l'ensemble des données d'entrées pour laquelle la fonction est correcte : ce sont les préconditions.
- Un commentaire décrit l'ensemble des données en sortie : ce sont les postconditions.
- En général, on veut que les fonctions soient exacte.

Exemples

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 un nombre réel compris entre 0 et 100 (formule x.)

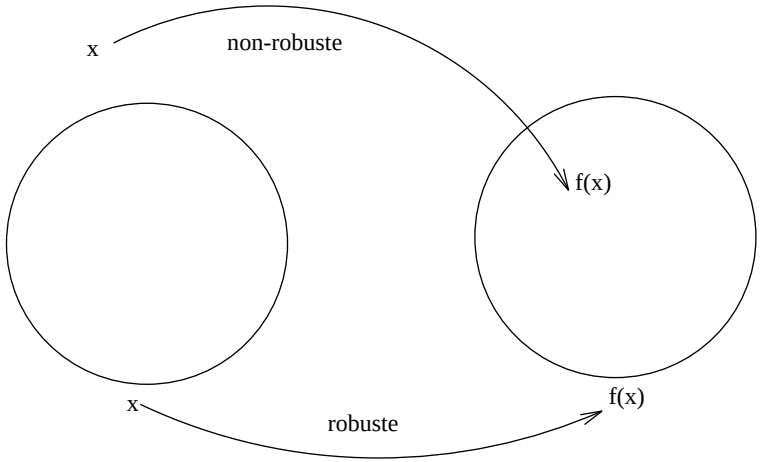
Exemple

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



Robustesse

Procédure

- Un commentaire au début d'une fonction spécifie si elle est robuste ou non.
- Une fonction robuste :
  - valide les arguments en entrée ;
  - envoie un code d'erreur précis.
- Les codes d'erreurs doivent être clairement documentés au début de la fonction.
- Ceci permet d'utiliser la fonction dans un test.

Robustesse

Procédure

- Les codes d'erreurs doivent être retournés séparément des valeurs de retour ou ne pas être dans l'image de la fonction.
- 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.

Fiabilité

Définition

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

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`.

Exemples

Reprenons l'exemples des multiples.

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

Exemple

Écrire un programme qui calcule le factoriel d'un nombre

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.

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).

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

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

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.

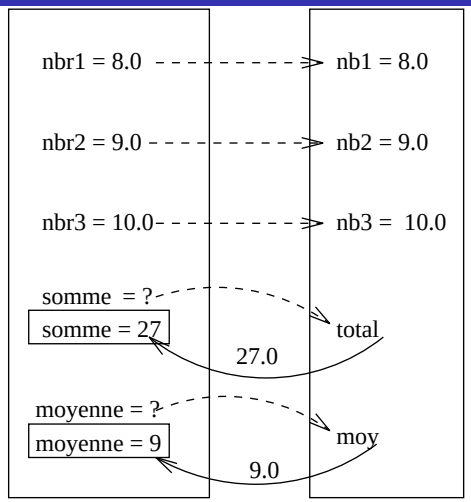
Méthode de passage d'un paramètre

Définition

Passage par valeur C'est le cas du paramètre d'entrée. La fonction travaille avec une copie de la valeur. Une modification du paramètre dans le corps de la fonction n'entraîne pas d'effet de bord.

Passage par référence C'est le cas du paramètre de sortie (et d'entrée/sortie). On vous donne accès directement à la variable passée en paramètre. en transmettant la référence (l'adresse en mémoire) de cette variable. Une modification du paramètre entraîne une modification de la variable transmise par un effet de bord.

Paramètres de sortie



Fonctions avec paramètres de sortie

Syntaxe de la déclaration (ligne prototype)

type nomFonction (*type*, *type*, *type*&, *type*&)

Syntaxe de la définition (paramètres formels)

type nomFonction (*type* par1, *type* par2, *type*& par3, *type*& par4)

Syntaxe de l'appel

nomFonction (p1, p2, p3, p4)

Raffinement successif

1. Analyse globale.
2. Conception globale (diagramme structurel).
3. Analyse et 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.
- Analyse/conception du niveau principal.
  1. Diagramme structurel (1^{er} *niveau*).
  2. Analyse du module principal.
  3. Algorithme du module principal.
- Analyse/conception des modules (fonctions).
  1. Conception *globale* (diagramme structurel).
  2. Analyse du module principal.
  3. Conception du module principal (algorithme).
- Implémentation des modules.

Raffinement successif

1. Analyse globale du problème.
2. Analyse/conception/implémentation du niveau principal.
   1. Conception *globale* (diagramme structurel).
   2. Analyse du module principal.
   3. Conception du module principal (algorithme).
   4. Implémentation.
3. Pour chacun des modules, on réapplique les trois étapes.
   1. Conception du module (diagramme structurel).
   2. Analyse du module.
   3. Conception du module (algorithme).
   4. 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.