Variables (locales et globales), fonctions et procédures

Nicolas Delestre et Michel Mainguenaud

{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr

Modifié pour l'ENSICAEN par :

Luc Brun

luc.brun@greyc.ensicaen.fr

Plan...

- Rappels
- Les sous-programmes
- Variables locales et variables globales
- Structure d’un programme
- Les fonctions
- Les procédures

Vocabulaire...

- Dans ce cours nous allons parler de “programme” et de “sous-programme”
- Il faut comprendre ces mots comme “programme algorithmique” indépendant de toute implantation

Rappels...

- La méthodologie de base de l’informatique est :
  1. Abstraire
     - Retarder le plus longtemps possible l’instant du codage
  2. **Décomposer**
     - "... chacune des difficultés que j’examinerai en autant de parties qu’il se pourrait et qu’il serait requis pour les mieux résoudre." Descartes
  3. Combiner
     - Résoudre le problème par d’abstractions

Par exemple...

- Résoudre le problème suivant :
  - Écrire un programme qui affiche en ordre croissant les notes d'une promotion suivies de la note la plus faible, de la note la plus élevée et de la moyenne
- Revient à résoudre les problèmes suivants :
  - Remplir un tableau de naturels avec des notes saisies par l'utilisateur
  - Afficher un tableau de naturels
  - Trier un tableau de naturel en ordre croissant
  - Trouver le plus petit naturel d'un tableau
  - Trouver le plus grand naturel d'un tableau
  - Calculer la moyenne d'un tableau de naturels
- Chacun de ces sous-problèmes devient un nouveau problème à résoudre
- Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait "quasiment" résoudre le problème initial

Sous-programme...

- Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial
- En algorithmique il existe deux types de sous-programmes :
  - Les
  - Les
- Un sous-programme est obligatoirement caractérisé par un nom (un identifiant) unique
- Lorsqu'un sous programme a été explicité (on a donné l'algorithme), son nom devient une nouvelle instruction, qui peut être utilisé dans d'autres (sous-)programmes
- Le (sous-)programme qui utilise un sous-programme est appelé **(sous-)programme appelant**

Règle de nommage...

- Nous savons maintenant que les variables, les constantes, les types définis par l'utilisateur (comme les énumérateurs) et que les sous-programmes possèdent un nom
- Ces noms doivent suivre certaines règles :
  - Ils doivent être explicites (à part quelques cas particuliers, comme par exemple les variables *i* et *j* pour les boucles)
  - Ils ne peuvent contenir que des lettres et des chiffres
  - Ils commencent obligatoirement par une lettre
  - Les variables et les sous-programmes commencent toujours par une minuscule
  - Les types commencent toujours par une majuscule
  - Les constantes ne sont composées que de majuscules
  - Lorsqu'ils sont composés de plusieurs mots, on utilise les majuscules (sauf pour les constantes) pour séparer les mots (par exemple JourDeLaSemaine)

Les différents types de variable...

- Définitions :
  - La **portée** d'une variable est l'ensemble des sous-programmes où cette variable est connue (les instructions de ces sous-programmes peuvent utiliser cette variable)
  - Une variable définie au niveau du programme principal (celui qui résout le problème initial, le problème de plus haut niveau) est appelée
    - Sa portée est totale : **tout** sous-programme du programme principal peut utiliser cette variable
  - Une variable définie au sein d'un sous programme est appelée
    - La portée d'un variable locale est uniquement le sous-programme qui la déclare
- Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale est localement masquée
  - Dans ce sous-programme la variable globale devient inaccessible

Structure d’un programme...

- Un programme doit suivre la structure suivante :

  Programme *nom du programme*

  Définition des constantes

  Définition des types

  Déclaration des variables globales

  Définition des sous-programmes

  début

  instructions du programme principal

  fin

Les paramètres...

- Un paramètre d'un sous-programme est une variable locale particulière qui est associée à une variable ou constante (numérique ou définie par le programmeur) du (sous-)programme appelant :
  - Puisque qu'un paramètre est une variable locale, un paramètre admet un type
  - Lorsque le (sous-)programme appelant appelle le sous-programme il doit indiquer la variable (ou la constante), de même type, qui est associée au paramètre
- Par exemple, si le sous-programme *sqr* permet de calculer la racine carrée d'un réel:
  - Ce sous-programme admet un seul paramètre de type réel positif
  - Le (sous-)programme qui utilise *sqr* doit donner le réel positif dont il veut calculer la racine carrée, cela peut être :
    - une variable, par exemple *a*
    - une constante, par exemple *5.25*

Les passage de paramètres...

- Il existe trois types d'association (que l'on nomme **passage de paramètre**) entre le paramètre et la variable (ou la constante) du (sous-)programme appelant :
  - Le **passage de paramètre en**
  - Le **passage de paramètre en**
  - Le **passage de paramètre en**

Le passage de paramètres en entrée...

- Les instructions du sous-programme ne l'entité (variable ou constante) du (sous-)programme appelant
  - En fait c'est la valeur de l'entité du (sous-) programme appelant qui est copiée dans le paramètre (à part cette copie il n'y a pas de relation entre le paramètre et l'entité du (sous-)programme appelant)
  - C'est le `seul` passage de paramètre qui admet l'utilisation d'une constante
- Par exemple :
  - le sous-programme *sqr* permettant de calculer la racine carrée d'un nombre admet un paramètre en entrée
  - le sous-programme **écrire** qui permet d'afficher des informations admet n paramètres en entrée

Le passage de paramètres en sortie...

- Les instructions du sous-programme une valeur à ce paramètre (valeur qui est donc aussi affectée à la variable associée du (sous-)programme appelant)
- Il y a donc une liaison forte entre le paramètre et l'entité du (sous-) programme appelant
  - C'est pour cela qu'on ne peut pas utiliser de constante pour ce type de paramètre
- La valeur que pouvait posséder la variable associée du (sous-)programme appelant par le sous-programme
- Par exemple :
  - le sous-programme **lire** qui permet de mettre dans des variables des valeurs saisies par l'utilisateur admet n paramètres en sortie

Le passage de paramètres en entrée/sortie...

- Passage de paramètre qui combine les deux précédentes
- A utiliser lorsque le sous-programme doit la valeur de la variable du (sous-)programme appelant
- Comme pour le passage de paramètre en sortie, on ne peut pas utiliser de constante
- Par exemple :
  - le sous-programme **échanger** qui permet d'échanger les valeurs de deux variables

Les fonctions...

- Les fonctions sont des sous-programmes admettant des paramètres et retournant un (comme les fonctions mathématiques y=f(x,y,...)
  - les paramètres sont en nombre fixe (≥ 0)
  - une fonction possède un seul type, qui est le type de la valeur retournée
  - le passage de paramètre est : c'est pour cela qu'il n'est pas précisé
    - lors de l'appel, on peut donc utiliser comme paramètre des variables, des constantes mais aussi des résultats de fonction
  - la valeur de retour est spécifiée par l'instruction **retourner**
- Généralement le nom d'une fonction est soit un nom (par exemple *minimum*), soit une question (par exemple *estVide*)

Les fonctions...

- On déclare une fonction de la façon suivante :

  fonction *nom de la fonction* **(***paramètre(s) de la fonction***) :** *type de la valeur retournée*

  Déclaration *variable locale 1 : type 1; . . .*

  début

  instructions de la fonction avec au moins une fois l'instruction ***retourner***

  fin

- On utilise une fonction en précisant son nom suivi des paramètres entre parenthèses
  - Les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètre

Exemple de déclaration de fonction...

fonction abs **(**unEntier : **Entier) : Entier**

début

    si unEntier $\geq$ 0 **alors**

        retourner unEntier

    finsi

    retourner

fin

Remarque : Cette fonction est équivalente à :

fonction abs **(**unEntier : **Entier) : Entier**

    Déclaration tmp: **Entier**

début

    si unEntier $\geq$ 0 **alors**

        tmp $\leftarrow$ unEntier

    sinon

        tmp $\leftarrow$ -unEntier

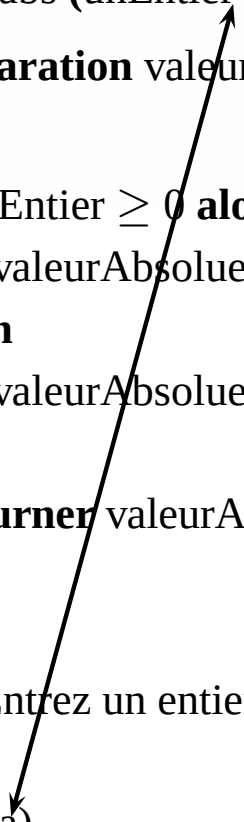
    finsi

    retourner tmp

fin

Exemple de programme...

```
Programme exemple1
    Déclaration a : Entier, b : Naturel
    fonction abs (unEntier : Entier) : Naturel
        Déclaration valeurAbsolue : Naturel
    début
        si unEntier ≥ 0 alors
            valeurAbsolue ← unEntier
        sinon
            valeurAbsolue ← -unEntier
        finsi
        retourner valeurAbsolue
    fin
début
    écrire("Entrez un entier :")
    lire(a)
    b ← abs(a)
    écrire("la valeur absolue de ",a," est ",b)
fin
```



Lors de l'exécution de la fonction *abs*, la variable *a* et le paramètre *unEntier* sont associés par un passage de paramètre en entrée : La valeur de *a* est copiée dans *unEntier*

Un autre exemple...

```
fonction minimum2 (a,b : Entier) : Entier
début
    si a ≥ b alors
        retourner b
    finsi
    retourner a
fin
fonction minimum3 (a,b,c : Entier) : Entier
début
    retourner minimum2(a,minimum2(b,c))
fin
```

Les procédures...

- Les procédures sont des sous-programmes qui ne retournent aucun résultat
- Par contre elles admettent des paramètres avec des passages :
  - en entrée, préfixés par **Entrée** (ou **E**)
  - en sortie, préfixés par **Sortie** (ou **S**)
  - en entrée/sortie, préfixés par **Entrée/Sortie** (ou **E/S**)
- Généralement le nom d'une procédure est un verbe

Les procédures...

- On déclare une procédure de la façon suivante :

  procédure *nom de la procédure* **(**

)

  Déclaration *variable(s) locale(s)*

  début

  instructions de la procédure

  fin

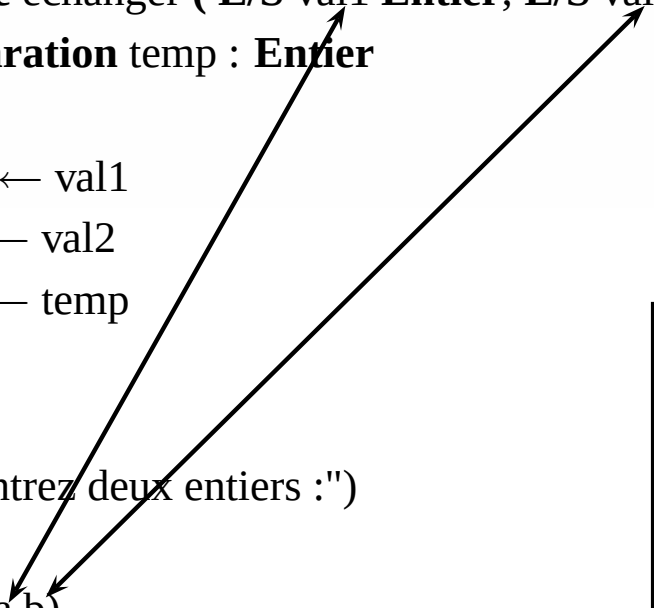
- Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses

Exemple de déclaration de procédure...

procédure calculerMinMax3 **(E** a,b,c : **Entier** ; **S** m,M : **Entier)**
début
    m ← minimum3(a,b,c)
    M ← maximum3(a,b,c)
fin

Exemple de programme...

```
Programme exemple2
    Déclaration a : Entier, b : Naturel
    procédure echanger ( E/S val1 Entier; E/S val2 Entier;)
        Déclaration temp : Entier
    début
        temp ← val1
        val1 ← val2
        val2 ← temp
    fin
début
    écrire("Entrez deux entiers :")
    lire(a,b)
    echanger(a,b)
    écrire("a=",a," et b = ",b)
fin
```



Lors de l'exécution de la procédure *echanger*, la variable *a* et le paramètre *val1* sont associés par un passage de paramètre en entrée/sortie : Toute modification sur val1 est effectuée sur a (de même pour b et val2)

Autre exemple de programme...

Programme *exemple3*

Déclaration entier1,entier2,entier3,min,max : **Entier**

fonction minimum2 **(**a,b : **Entier) : Entier**

. . .

fonction minimum3 **(**a,b,c : **Entier) : Entier**

. . .

procédure calculerMinMax3 **(E** a,b,c : **Entier** ; **S** min3,max3 : **Entier)**

début

min3 ← minimum3(a,b,c)

max3 ← maximum3(a,b,c)

fin

début

écrire("Entrez trois entiers :")

lire(entier1) ;

lire(entier2) ;

lire(entier3)

calculerMinMax3(entier1,entier2,entier3,min,max)

écrire("la valeur la plus petite est ",min," et la plus grande est ",max)

fin

Fonctions/procédures récursives

Une fonction ou une procédure récursive est une fonction qui s'appelle elle-même.

Exemple :

```
fonction factorielle (n: Naturel) : Naturel
début
    si n = 0 alors
        retourner 1
    finsi
    retourner n*factorielle(n-1)
fin
```

Liste des appels

factorielle(4)		4*factorielle(3)	=	4*3*2*1
↓		↑		
factorielle(3)		3*factorielle(2)	=	3*2*1
↓		↑		
factorielle(2)		2*factorielle(1)	=	2*1
↓		↑		
factorielle(1)		1*factorielle(0)	=	1*1
↓		↑		
factorielle(0)	→	1		

Récursivité : Caractérisation

On peut caractériser un algorithme récursif par plusieurs propriétés:

- Le mode d'appel : direct/indirect
- Le nombre de paramètres sur lesquels porte la récursion : arité
- Le nombre d'appels récursifs : ordre de récursion
- Le genre de retour : .

Récursivité : mode d'appel

Une fonction récursive s'appellant elle même a un mode d'appel (ex: factorielle). Si la récursivité est effectuée à travers plusieurs appels de fonctions différentes le mode d'appel est .

```
fonction pair (n: Naturel) : Booléen
début
    si n = 0 alors
        retourner vrai
    finsi
    retourner imPair(n-1)
fin
```

```
fonction imPair (n: Naturel) : Booléen
début
    si n = 0 alors
        retourner faux
    finsi
    retourner pair(n-1)
fin
```

Récursivité : Arité/bien fondé

- L’arité d’un algorithme est
- Récursivité bien fondé :
  Une récursivité dans laquelle les paramètres de la fonction appelée sont «plus simple» que ceux de la fonction appelante. Par exemple `factorielle(n)` appelle `factorielle(n-1)`.
  Exemple de récursivité mal fondée:

GNU: Gnu is not Unix

Récursivité : Ordre de récursion

- L'ordre de récursion d'une fonction est le nombre d'appels récursifs lancés à chaque appel de fonction. Par exemple, factorielle(n) ne nécessite qu'un seul appel à la fonction factorielle avec comme paramètre $n - 1$. C'est donc une fonction récursive d'ordre 1.
  Par exemple, la fonction suivante basée sur la formule $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$, est d'ordre 2.

```
fonction comb (Entier p, n) : Entier
début
    si p = 0 ou n = p alors
        retourner 1
    sinon
        retourner comb(p, n - 1) + comb(p - 1, n - 1)
    finsi
fin
```

Récursicité : genre de retour

Le genre d'un algorithme récursif est déterminé par le traitement effectué sur la valeur de retour.

- Si la valeur de retour est retrounée sans être modifié l'algorithme est dit terminal (Par exemple `pair/imPair` est terminal)
- Sinon l'algorithme est dit (ex factorielle qui multiplie la valeur de retour par n).

Remarques sur la récursivité

- La récursivité peut simplifier considérablement certains problèmes.
- Un appel de fonction/procédure à un coût non négligeable. Les programmes récursifs sont souvent plus coûteux que leurs homologues non récursifs.