

Sous-programmes - Paramètres

L. Mounier

UGA - M2 CCI - PL1

19 octobre 2023

Pourquoi “structurer” un programme ?

- limites d'un programme “monolithique” :
(i.e., avec uniquement un programme principal)
 - maîtrise de la complexité, lisibilité, maintenance
 - conception par composition d'actions non élémentaires
 - ré-utilisation de portions de code
 - paramétrage de certaines opérations
 - etc.
- structure en **sous-programmes** :
notions d'action/fonction

Sous-Programmes (actions et fonctions)

Notation Algo :

Inserer : **action** (...)

...

EstPresent : **fonction** (...) → booléen

...

retourner vrai

Traduction en C :

- uniquement une notion de “fonction”
- type prédéfini `void` indique l'absence de valeur de retour

```
void Inserer (...) {
```

```
    ....
```

```
}
```

```
int EstPresent () {
```

```
    ....
```

```
    return 1
```

```
}
```

Structure d'un programme C

```
#include <stdio.h> // librairies pour les fonctions/variables externe

int a, b, c ; // variables globales

void F1 (...) {
    int x1, x2 ; // variables locales a F1
    ...
}

void F2 (...) {
    int x1, x2 ; // variables locales a F2
    ...
}

int main() {
    int x1, x2 ; // variables locales a main
    ...
    return 0 ;
}
```

Portées et durées de vie

Déterminer :

- à quelle **définition** se rapporte une **utilisation** de variable ?
- quand une variable est-elle **“active”** ?

→ Trois grandes règles :

- 1 les variables locales à un bloc/sous-programme ont une durée de vie égale à celle de ce bloc/sous-programme
- 2 les identificateurs locaux à un bloc/sous-programme ne sont pas visibles dans les blocs englobants
- 3 en cas de conflit de nom, on considère l'identificateur défini dans le bloc englobant le plus imbriqué

Portées et durée de vie (exemple)

Valeurs des variables en chaque point du programme ?

```
int a, b, c ; // variables globales

void F() {
    int x, a ; // variables locales a F
    a = 1 ;
    x = a + c ;
    c = x ;
}

void main() {
    a=2 ;
    c=2 ;
    F() ;
    F() ;
}
```

Passage de paramètres

Paramètres formels et paramètres effectifs :

A : une action (la donnée x : un entier)

↔ x est un paramètre **formel**

A ($y + 12$)

↔ $y+12$ est le paramètre **effectif**

Deux modes de passage de paramètre en notation algo :

- paramètres “données” :

Ecrire : une action (la donnée x : un entier)

{la valeur de x n'est pas modifiée par Ecrire}

- paramètres “résultats” :

Lire : une action (le résultat x : un entier)

{la valeur de x est modifiée par Lire}

Passage de paramètres en C

Un seul mode de passage = passage **par valeur**

⇒ un paramètre formel ~

- variable locale au sous-programme
- initialisée avec la **valeur** du paramètre effectif

⇒ ne permet pas un mode de passage par résultat ???

```
int r=12 ;
```

```
void Incrementer (int x) {  
    x = x+1 ;  
}
```

```
    Incrementer (r) ; // r n'est pas modifie !!!  
printf ("%d\n", r) // affiche 12 ...
```


Notion de pointeurs

Différentes informations associées à une variable :

- son nom et son type
 - définis lors de sa déclaration
 - non modifiables à l'exécution (en C)
- sa valeur
 - modifiée à chaque affectation
- son **adresse en mémoire**
 - fixée en début d'exécution
 - peut être accédée en cours d'exécution

Pointeurs :

- un type "adresse de ..."
- représente l'adresse d'une variable en mémoire

Les pointeurs en C

Déclaration d'une variable de type pointeur :

```
int x, y ; // x et y variable de type entier
int *p, *q ; // p et q variables de type "pointeur sur entier"
              // pour contenir l'adresse d'une var. de type entier
```

Affectation de pointeur, opérateur & :

```
p = &x ; // l'adresse de x est affectee a p
q = p ; // la valeur de p est affectee a q
```

Déréférencement de pointeur, opérateur * :

```
x = *p + 1 ; // x est incrementee de 1
*q = y + 1 ; // y+1 est affecte a x
```

Comparaison de pointeur : opérateurs ==, !=

Pointeurs sur tableaux et structures

Pointeurs sur structures, notation `->` :

```
typedef struct { int numerateur ; int denominateur } Fraction ;  
Fraction f ; // f variable de type Fraction  
Fraction *pf ; // pf pointeur sur Fraction  
...  
f.numerateur = 2 ; f.denominateur = 5 ;  
pf = &f ;  
(*pf).denominateur = 0 ;  
pf -> denominateur = 0 ; // syntaxe equivalente
```

Pointeurs sur tableaux :

```
int T[10] ;  
int *p ; // p pointeur sur entier  
...  
p = T ; // T designe l'adresse de T[0]  
*p = 12 ; // affectation de T[0]  
*(p+1) = 12 ; // affectation de T[1]
```

Paramètres résultat en C

Incr : une action (la donnée x : un entier, le résultat y : un entier)
{ y vaut x+1 à la fin de l'exécution }

```
void Incr (int x, int y) {  
    y = x+1 ;  
}  
...  
int r ;  
Incr (12, r) ; // r n'est pas modifié !!!
```

→ on transmet **un pointeur** sur la variable à modifier :

```
void Incr (int x, int * py) {  
    *py = x+1 ;  
}  
...  
int r ;  
Incr (12, &r) ; // r est modifié ...
```

Question : Incr : une action (la donnée-résultat x : un entier) ?

Cas des tableaux en paramètres

```
int T[10] ; // T est de type pointeur sur entier
```

tableau en paramètre = **un pointeur** sur les éléments du tableaux
~> passage par **données-résultat**

```
#define N 10
```

```
void incrementer (int T[]) { ... T[i] = T[i] + 1 ; ... }
```

```
int main () {  
    int Tab[N] ;          // ici la longueur N du tableau est globale  
    incrementer(Tab) ;   // incremente tous les elements de T  
}
```

Exercice : traduire en C le lexique suivant

TNom : le type chaîne de 20 caractères

TPersonne : le type (age : un entier, nom : un TNom)

TEffectif : le type tableau sur 0..100 de TPersonne

copier : une action

(la donnée p1 : un TPersonne, le résultat p2 : un TPersonne)

{ copie p1 dans p2 }

ReNommer : une action

(la donnée-résultat p : un TPersonne, la donnée s : un TNom)

{ donne le nom s à p }

Trier : une action (la donnée-résultat e : un TEffectif)

{ trie le tableau e }