

Langage C

ENSTA - TC 1ère année

François Pessaux

U2IS

2016-2017

francois.pessaux@ensta-paristech.fr


Les fonctions

- Une fonction
 - Prend une ou des valeurs en argument.
 - Retourne **une** valeur.
- D'où, une fonction est définie par :
 - le **type** de la valeur qu'elle **retourne**,
 - son **nom**,
 - les **types** des **paramètres** qu'elle **attend**,
 - et son **corps** : la description du calcul qu'elle effectue, spécification **exécutable**.

Exemples

```
int twice (int i)
{
    return (i * 2) ;    // Corps: retourne bien un int.
}

float square_minus (float x, float y)
{
    float t = (x - y) ; /* Variable locale t. */
    return (t * t) ;
}
```

-  Des **virgules** entre les paramètres ! Pas des points-virgules !

Rappels / remarques importants

- Écrire une fonction ne l'exécute pas ! Il faut l'**appeler** !
- Appel : nom + arguments entre **parenthèses** séparés par des **virgules** :

```
...
v = twice (2) ;
v = square_minus (v, -v) + 42 ;
...
```

- Corps de la fonction uniquement exécuté lors de l'appel à la fonction.
- Paramètre **formel** : nom donné dans la **définition** de la fonction : i pour twice, x et y pour square_minus.
- Paramètre **effectif** : expression donnée à un paramètre formel lors de l'**appel**.
- Appel :
 - 1 Évaluation (calcul) des expressions arguments effectifs → **valeurs**.
 - 2 « Association » de chaque paramètre formel avec sa valeur.
 - 3 Calcul du corps de la fonction sous ces hypothèses d'association.
 - 4 Transmission à l'appelant de la valeur retournée par ce corps.

Les fonctions ne retournant « rien »

- Si pas d'argument : absence dans la **définition**, absence dans l'**appel**.
- Fonction retournant « rien » :
 - « Retourne » le type **void**.
 - Aucune expression après l'instruction return.
 - Dernière instruction return optionnelle.
 - return peut être nécessaire si plusieurs points de terminaison de la fonction.

```
int const_fun ()
{
    return (42) ;
}

void just_print (int i)
{
    printf ("I just print %d and %d\n", i, const_fun ()) ;
    return ;
}
```

Variables locales, variables globales

- On peut déclarer des variables **locales** dans une fonction.
- Plus généralement : déclaration de variables locales dans des **blocs** ({ ... }).
- Corps de la fonction ≡ bloc.
- ⇒ Vivantes seulement jusqu'à la fin de la fonction / du bloc.
- Variables globales (définies hors de fonctions) restent accessibles dans la fonction.

```
int glob = 5 ; /* Variable globale. */

int f (int x)
{
    int i = x * 2 ; /* Variable locale à la fonction f. */
    {
        int j = i + 5 ; /* Variable locale au bloc interne à f. */
        i = i + j ;
    } /* Fin du bloc: j n'est plus accessible. */
    return (i + 3 + glob) ;
}
```

Déclaration / prototype (1)

- Utilisation d'une fonction en **dehors de son fichier hôte** ?
- Utilisation d'une fonction **avant sa définition** ?

⇒ Son **type** doit être connu.

- Donner juste le type : **déclaration** ou « *prototype* » (et non définition).
- Rend visible par toute personne ayant accès à cette **déclaration**.
- Règle valable pour les autres types de variables.
- Déclaration d'une entité : son **nom** + son **type**.
- Déclaration :
 - Faite au « *toplevel* » d'un fichier d'en-tête (.h).
 - Utilisée via les directives #include.

Déclaration / prototype (2)

- Pour une fonction :
 - son type de retour,
 - son nom,
 - les types (et les noms optionnellement) de ses arguments.
- Pour une variable :
 - le mot clef **extern** (sinon ça serait une définition de variable!),
 - son type,
 - son nom.

```
/* Déclaration d'une variable (globale) entière. */  
extern int errno ;  
/* Déclaration de la fonction arc-tangente de y/x. */  
float atan2 (float y, float x) ;
```

- Indique « **comment** » utiliser la fonction.
- ... Mais pas ce qu'elle fait (et surtout pas comment).

Quelques fonctions déjà vues

- **printf** : affichage à l'écran.
 - Prototype un peu bizarre car nombre et types des arguments variables.
 - `void printf (const char *restrict format, ...)`; ☹
- **main** : La fonction point d'entrée de tout programme.
 - `int main ()`;
 - ★ Retourne un entier : le code de retour du programme.
 - ★ Ne prend pas d'argument.
 - `int main (int argc, char *argv[])`;
 - ★ Forme alternative : arguments en provenance de la ligne de commande.
 - ★ `argc` : **nombre d'entrées** dans `argv`.
 - ★ `argv` : tableau de **chaînes de caractères**.
 - ★ 1^{er} élément : **nom + chemin** de l'exécutable.
 - ★ Éléments suivants : arguments effectifs.
 - ★ Ex : `> ./myprog 1 hop 4`

⇒ `argv =`

0	1	2	3
"myprog"	"1"	"hop"	"4"

Appel(s) de fonction(s)

Déroulement d'un appel de fonction

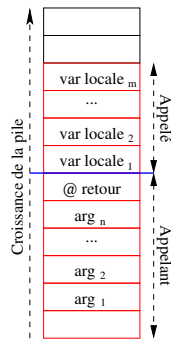
- Illustration sur le calcul de la racine carrée d'un réel a .
- Méthode itérative de Newton.
- \sqrt{a} donnée par $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$
- n : Nombre d'itérations (à fixer arbitrairement).
- $x_0 > 0$: Point de départ (à fixer arbitrairement).

Protocole d'appel

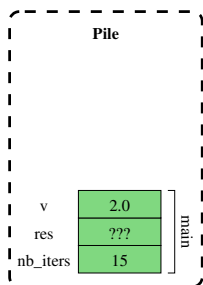
- Appeler une fonction c'est :
 - stocker la valeur de ses paramètres,
 - aller exécuter son code,
 - et en revenir avec le résultat.
- Différentes conventions d'appel selon les langages, les architectures.
- Principe restant majoritairement le même : utilisation de la **pile**.
- Différences dans « qui (appelant/appelé) fait quoi ».
- Différences dans l'agencement mémoire d'informations (registres vs pile).

Protocole d'appel de manière générique

- Côté appelant :
 - ▶ Évaluation des arguments à passer → mis sur la pile.
 - ▶ Adresse de retour mise sur la pile.
- Côté appelé :
 - ▶ Réserve des variables locales sur la pile.
 - ▶ Exécution du code.
 - ▶ Stockage de la valeur à retourner (en registre ou pile).
 - ▶ Aller ... à l'adresse de retour.
- Côté appelant :
 - ▶ Récupérer la valeur retournée par l'appelé.
 - ▶ Supprimer de la pile les arguments passés lors de l'appel.



Déroulement d'un appel de fonction

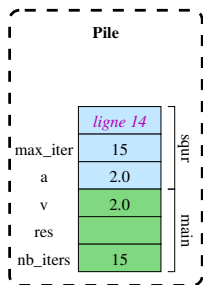


```

1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double squr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = squr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }
    
```

Définition et initialisation des variables locales de main

Déroulement d'un appel de fonction

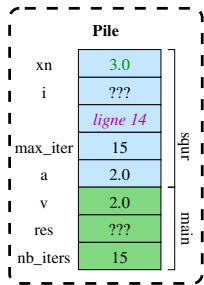


```

1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double squr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = squr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }
    
```

Appel de la fonction squr

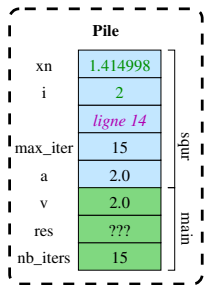
Déroulement d'un appel de fonction



```
1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     ► double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }
```

Définition et initialisation des variables locales de sqr

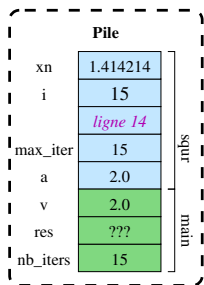
Déroulement d'un appel de fonction



```
1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7     ► xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }
```

Itérations de la boucle (ici $i = 2$, $xn = 1.41499$)

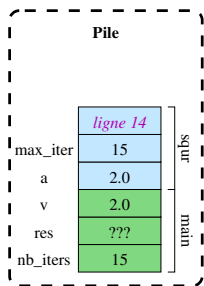
Déroulement d'un appel de fonction



```
1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     ► return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }
```

Sortie de la boucle, prêt à retourner de la fonction sqr

Déroulement d'un appel de fonction



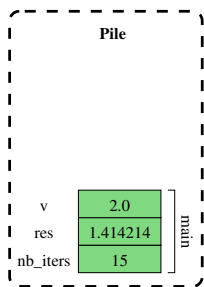
```

1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }

```

Retour de la fonction : suppression des variables locales, retourner à l'adresse spécifiée sur la pile

Déroulement d'un appel de fonction



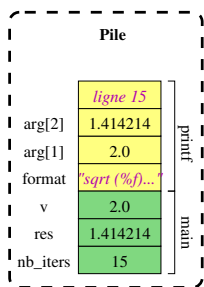
```

1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }

```

Affectation de la valeur de retour de la fonction sqr

Déroulement d'un appel de fonction



```

1 #define X0 (3.0) // Germe: quelconque > 0
2
3 double sqr (double a, int max_iter) {
4     int i ;
5     double xn = X0 ;
6     for (i = 0; i < max_iter; i++)
7         xn = 0.5 * (xn + (a / xn)) ;
8     return (xn) ;
9 }
10
11 int main () {
12     int nb_iters = 15 ;
13     double res, v = 2.0 ;
14     res = sqr (v, nb_iters) ;
15     printf ("sqrt (%f) = %f\n", v, res) ;
16     return (0) ;
17 }

```

Appel à la fonction printf ...etc ...

La récursivité

Fonctions (mutuellement) récursives

- En C, (comme dans beaucoup de langages) : toutes les fonctions sont **implicitement** mutuellement récursives (si besoin).
 - Mais la première définie doit « connaître » la suivante.
- ⇒ Utilisation de **déclarations**.

```
#include <stdbool.h>
bool natural_even (unsigned int n) ; /* Décl. de natural_even. */
bool natural_odd (unsigned int n) ; /* Décl. de natural_odd. */

bool natural_even (unsigned int n) /* Déf. de natural_even. */
{
    if (n == 0) return (true) ;
    else return (natural_odd (n - 1)) ;
}

bool natural_odd (unsigned int n) /* Déf. de natural_odd. */
{
    if (n == 0) return (false) ;
    else return (natural_even (n - 1)) ;
}
```

« Récursion, boucles, ça se termine quand ? »

- Pour obtenir un **résultat**, un calcul **doit terminer**.
- Boucle avec condition d'arrêt **toujours fausse** ⇒ boucle « infinie ».
- Récursion avec **cas de base** (≡ condition d'arrêt) **jamais atteint** ⇒ récursion « infinie ».
- Dans ces cas ... on peut attendre longtemps.
- La **terminaison** est une propriété **très importante**.

Comment s'assurer de la terminaison ?

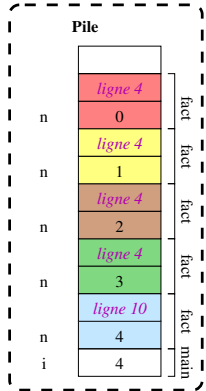
- Pas de méthode automatique.
- Nécessite une **preuve de terminaison**.
- \Rightarrow Problème de raisonnement, pas de calcul.
- Intuition : démontrer que chaque appel récursif s'effectue sur un argument « plus petit »
 - \Rightarrow Nécessité d'un ordre $<$ sur le domaine des arguments.
 - Ordre bien fondé : \nexists suite décroissante infinie.
- $\forall n \in \mathbb{N}$, `fact (n)` : si `n == 0` alors `1` sinon `n * fact (n - 1)`
Facile : appel récursif sur `n-1` qui est bien $< n$.
- $Ack(m, n) = \begin{cases} n+1 & \text{Si } m=0 \\ Ack(m-1, 1) & \text{Si } m>0 \text{ et } n=0 \\ Ack(m-1, Ack(m, n-1)) & \text{Si } m>0 \text{ et } n>0 \end{cases}$
Nettement moins évident ! Mais elle termine (ordre lexicographique).

Appels de fonctions récursives

- Principe d'appel **identique** aux autres fonctions.
 - À chaque appel on empile les informations nécessaires.
 - Lorsque l'on sort d'un appel, on dépile.
- \Rightarrow La pile augmente avec la profondeur de récursion.
- En cas de récursion trop profonde, possibilité de dépassement de pile (« *stack overflow* »).
- Pour autant **la récursivité n'est pas à bannir !**
 - Permet d'exprimer simplement et élégamment certains algorithmes.
 - Incontournable pour certains problèmes (sinon, obligation de se gérer manuellement une pile en fait).
 - Les compilateurs savent dé-récursiver la récursion dite **terminale**.

Appels de fonction(s) récursive(s)

Déroulement d'appels de fonction récursive



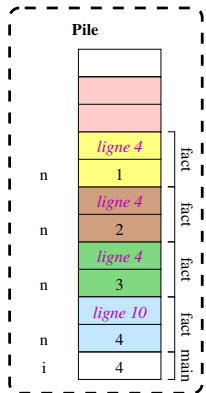
```

1 ►unsigned int fact (unsigned int n)
2 {
3     if (n == 0) return (1) ;
4     else return (n * fact (n - 1)) ;
5 }
6
7 int main ()
8 {
9     unsigned int i = 4 ;
10    printf ("Fact(%d)=%d\n", i, fact (i)) ;
11    return (0) ;
12 }

```

État de la pile à l'entrée de l'appel à fact avec l'argument 0.

Déroulement d'appels de fonction récursive



```

1 unsigned int fact (unsigned int n)
2 {
3     if (n == 0) return (1) ;
4     ► else return (n * fact (n - 1)) ;
5 }
6
7 int main ()
8 {
9     unsigned int i = 4 ;
10    printf ("Fact(%d)=%d\n", i, fact (i)) ;
11    return (0) ;
12 }

```

État de la pile au retour de l'appel à fact avec l'argument 0.